Advances in Engineering Software 132 (2019) 29–46

Contents lists available at ScienceDirect

Advances in Engineering Software

journal homepage: www.elsevier.com/locate/advengsoft

Rapid software prototyping for heterogeneous and distributed platforms

Tim Besard[b], Valentin Churavy[*,a], Alan Edelman[a], Bjorn De Sutter[b]

[a] Massachusetts Institute of Technology, USA
[b] Ghent University, Belgium

ARTICLE INFO

Keywords:
Julia
Generic programming
Heterogeneous systems
CUDA
Distributed computing

ABSTRACT

The software needs of scientists and engineers are growing and their programs are becoming more compute-heavy and problem-specific. This has led to an influx of non-expert programmers, who need to use and program high-performance computing platforms.

With the continued stagnation of single-threaded performance, using hardware accelerators such as GPUs or FPGAs is necessary. Adapting software to these compute platforms is a difficult task, especially for non-expert programmers, leading to applications being unable to take advantage of new hardware or requiring extensive rewrites.

We propose a programming model that allows non-experts to benefit from high-performance computing, while enabling expert programmers to take full advantage of the underlying hardware. In this model, programs are generically typed, the location of the data is encoded in the type system, and multiple dispatch is used to select functionality based on the type of the data. This enables rapid prototyping, retargeting and reuse of existing software, while allowing for hardware specific optimization if required.

Our approach allows development to happen in one source language enabling domain experts and performance engineers to jointly develop a program, without the overhead, friction, and challenges associated with developing in multiple programming languages for the same project.

We demonstrate the viability and the core principles of this programming model in Julia using realistic examples, showing the potential of this approach for rapid prototyping, and its applicability for real-life engineering. We focus on usability for non-expert programmers and demonstrate that the potential of the underlying hardware can be fully exploited.

1. Introduction

It is a truth universally acknowledged that efficient programming of high-performance computing (HPC) systems, be it GPUs or clusters, is difficult and best left to experts. It is therefore often the case that in domains such as engineering and scientific computing, software development happens in distinct phases or modes. First, there is rapid prototyping, then there is performance engineering. Existing approaches to bridge the gap between these two worlds fail in several ways. For example, some enforce overly rigid frameworks, e.g., by restricting programmers to predetermined data types or functions. Others combine multiple languages or dialects thereof for the different phases, and struggle with what is known as the two-language problem [1]. The failures of existing approaches eventually result in a loss of productivity and innovation capacity, because the developed solutions all too often are one-offs, not readily used by others or hard to integrate inside other systems. As a result, engineers, scientists and other domain experts with large scale problems are still searching for a solution that lets them innovate quickly, scale initial prototypes to real-world datasets, and continuously improve, adapt and develop their code, without having to sacrifice either expressiveness or performance.

Rapid prototyping, fluent collaboration, and speedy innovation require simplicity and expressiveness (at an abstract level), easy code reuse, extensibility, and composability. This implies that code should be agnostic to and ideally work with all relevant data-types. It should hence be developed in terms of abstract or generic types.

Performance engineering requires to some extent similar features. The reason is that on the one hand performance engineering involves knowledge of and tuning for specific hardware at hand. This requires the ability to specialize low-level aspects of computations and communications to fit the hardware. However, while it is often believed that changing the used hardware type requires changes in the source language or calling libraries, we are convinced that this notion has hindered the use of hardware accelerators.

* Corresponding author.
E-mail addresses: tim.besard@ugent.be (T. Besard), vchuravy@mit.edu (V. Churavy), edelman@mit.edu (A. Edelman), bjorn.desutter@ugent.be (B.D. Sutter).

https://doi.org/10.1016/j.advengsoft.2019.02.002
Received 18 October 2018; Accepted 3 February 2019

On the other hand performance also requires knowledge of the domain and tuning for the context in which the solution is used. As for the latter, naive system-level performance engineers waste their energy on teraflops. Sophisticated scientists and engineers know when they can take a larger time step or search step in their differential equation or optimization than their software library considers "safe". This is not cheating or reckless. This is experienced performance engineering. The kind one probably will not find in a textbook. Sparse linear algebra takes applications far. Structured linear algebra can take applications farther. New algorithms are imagined all the time, but does one have to rewrite huge parts of code to even find out whether a new algorithm will show promise?

Any proposed approach must therefore answer the following question: How easy is it to apply special knowledge to gain performance, be it knowledge from the application domain, or knowledge about the hardware and system level? In light of this question, this paper goes past the notion of writing performant code that accomplishes a task perhaps as a "one-off", for oneself or a small team. Rather, we explore the tension between obtaining large performance from hardware accelerators and the productivity benefits that come from maximizing simplicity. In our vision, a code writer should not ask only how much performance he can get, but rather how easily can he get performance, how many people can he share the reached performance with, is he siloed or is his performance extensible, and will others and even himself be able to make use of his hard work in even a few short years, or when his organization buys new hardware next month.

In line with this vision, this paper sets out an approach that tears down the traditional wall between rapid prototyping and performance engineering. This paper argues and demonstrates that this is possible with *one language*, i.e., one co-design of composable and extensible programming abstractions that implement the necessary separations of concerns on the one hand, and of compilation tools capable of specialization on the other hand. With such a design, we can prevent that developed applications and, equally important if not more, libraries are no longer siloed or one-offs.

A core concept in our approach is that of array abstractions. Arrays are natural language elements for engineers and scientists. So the use of arrays and abstractions of arrays and of operations on them (`map`, `reduce`,...) simplifies their lives and improves their expressiveness and productivity. Arrays can be dense, sparse, triangular, or structured. Arrays can also be on a CPU, GPU, homogeneously distributed, or heterogeneously distributed. Software developers typically think of the mathematical structure (dense,...) and the hardware structure (CPU,...) separately. They seem so very different. This paper argues that this does not need to be the case, and proposes the advantages of remembering that structure and storage can both be treated as abstractions. As we will demonstrate, this allows us to melt the distinction between mathematical structure and architectural structure, and hence enable all necessary forms of performance engineering.

To build our case and demonstrate the viability and advantages of our approach, we build on Julia [1–3] and its rich set of array abstractions [4]. We have extended it with a powerful GPU compiler [5], and combine array abstractions we developed for GPU computing [6] and distributed computing [7] to enable transparent distributed heterogeneous computing. All of our work is open-source, and has been contributed to and integrated with the upstream repositories.

The main contributions of this paper are the following:

- We present a set of array abstractions and implementations thereof in the CuArrays.jl and DistributedArrays.jl packages that enable rapid development of solutions for real-world engineering problems, and at the same time enable the exploitation of heterogeneous high-performance hardware.
- We demonstrate the composability of the abstractions and underlying infrastructure, and show how it facilitates separation of concerns regarding what is computed, where the underlying data is stored, and how communication happens.
- We discuss portability issues and argue how the proposed abstractions handle them.
- We present a performance evaluation that demonstrates the extent of the composability and the ease with which the potential of heterogeneous hardware can be exploited.

This paper is structured as follows. First, Section 2 introduces three examples with which we will later demonstrate the usability of our approach. All three examples are instances of rapid prototyping and exploration that could build the foundation for a larger project. Section 3 and 4 introduce the characteristics of Julia and its array abstractions that help us achieve our goals of composability and usability.

Section 5 then introduces the two packages CuArrays.jl and DistributedArrays.jl that implement the aforementioned array abstractions for respectively GPUs and for distributed environments. Due to the nature of array abstractions in Julia, these two packages transparently compose to provide a solution for distributed heterogeneous computing in Julia. Section 6 shows how this applies to each of our examples, and Section 7 analyzes the performance of these applications. Section 8 then compares our approach to other research and discusses related work. Finally, Section 9 draws conclusions, summarizes the current status of our work, and discusses future work.

2. Use cases

For the purpose of explaining concepts in this paper, we introduce three examples that are relevant to computer-based engineering techniques: the power method to calculate eigenvalues, gradient descent to minimize a loss function, and the Kronecker product of two matrices. These examples represent different levels of application complexity, and demonstrate different aspects of our approach. We have implemented the examples in Julia [2], using high-level, idiomatic code that stays as close as possible to the original mathematical descriptions. To emphasize this, these and other code listings that contain code that would be written by regular users, e.g., during application prototyping, are put in a green box. Code that requires more in-depth knowledge of the language is listed in orange, while code that would only be written by expert programmers, e.g., as part of a library, is shown in a red box.

2.1. Power iteration

The power method serves as the first, simplest example. This is an eigenvalue algorithm, approximating the dominant eigenvalue of a diagonalizable matrix by means of an iterative algorithm [8]. The associated eigenvalue is then computed using the Rayleigh quotient. The Julia implementation in Listing 1 mirrors the high-level descriptions of these algorithms from the corresponding Wikipedia pages, and uses simple operations on arrays, such as the dot product, matrix-vector multiplication, the Euclidean norm of a vector, and element-wise division. The parameter `p` of the `domeigen` function defines the number of iterations the method should perform.

Note that like all other listings in this paper, Listing 1 is not pseudo code. It is pretty printed Julia source code. The ability to write such code, using a Unicode character set, allows engineers to produce very readable source code, at the mathematical level of abstraction at which they prefer to reason and to express their ideas.

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

2.2. Proximal gradient descent

Listing 2 implements a more complex example that combines array operations with a generically typed external library that extends the

```
using LinearAlgebra
using Random

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

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

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

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

  return bₖ, λ
end
```

Listing 1. Power method implementation approximating the dominant eigenvector and eigenvalue of a matrix.

```
using ForwardDiff: gradient, derivative
using LinearAlgebra

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

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

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

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

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

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

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

  model = linear_regression
  loss = mean_squared_error
  optimize = proximal_gradient_descent

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

Listing 2. Implementation of the proximal gradient descent method, minimizing a squared error loss function.

base language. The array operations now also include higher-order abstractions that compose with arbitrary user code.

Specifically, the example implements proximal gradient descent to minimize the squared error loss of a linear regression model. The example uses the ForwardDiff.jl package [9] to determine the gradient and derivative of the loss function as defined by the user. This package implements forward-mode automatic differentiation in Julia. Under the hood, it specializes user code to generate efficient machine code for computing derivatives. The ability to differentiate arbitrary user code distinguishes this Julia package from others. Many existing machine learning frameworks either require engineers to pick functions from a fixed library of functions for which gradients have been defined, while others can compute custom derivatives but only if the original function had been specified as a computational graph. By enabling us to differentiate arbitrary imperative code, the ForwardDiff.jl package improves productivity as well as flexibility of machine learning frameworks built on top of this package.

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

Note that both the model and loss functions, of which lines 5 and 9 show examples, are defined independently from the optimization algorithm in `proximal_gradient_descent`. The model and loss functions are passed to the optimization algorithm as arguments, and they are simply passed on to anonymous functions (lambdas) that are themselves fed to the `gradient` and `derivative` functions from the ForwardDiff.jl library on 12 and 15. This generalizes the implementation and makes it possible for the developer to iterate independently on each aspect of the implementation (loss, model, and optimization algorithm).

From the compiler's perspective, the `gradient` (line 12) and `derivative` (line 15) functions return dynamically-generated code. The Julia run-time compiler then generates specialized and statically optimized machine code. The design of the Julia language and its compiler, described in detail in Section 3, makes it possible to deliver good performance and enable code generation for accelerators, such as GPUs, that require static code.

The simple code of Listing 2 performs various operations on arrays much like those in Listing 1, but it also uses abstractions that compose with user code. For example, the loss function on line 9 calls the standard library operation `sum` with the user-defined function `abs2`, which is applied to all elements before they are summed. We will later discuss how this makes it possible to separate the concerns of application code from how the underlying abstractions are implemented.

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

2.3. Kronecker product

Finally, we describe a scenario where a more advanced user prototypes an algorithm by means of declarative code instead of an imperative subprogram. Specifically, Listing 3 implements the Kronecker product of two matrices, $\mathbf{A}\otimes\mathbf{B}$, where every element of the first matrix is multiplied with every element of the second matrix:

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

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

As opposed to defining an imperative function that constructs an output matrix and eagerly computes the value for every element, we define a `Kronecker` type that lazily computes individual values when requested. This is called a structured matrix [10]. It is also a common pattern in the Julia programming language, which provides many such array as part of the standard library. This example will demonstrate how our approach is composable with such infrastructure from the standard library.

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

Expressing computation declaratively using lazy arrays has several advantages: first and foremost, it saves on memory usage and avoids unnecessary computations. Furthermore, we can provide optimized implementations of certain methods by using problem-specific knowledge. For example, in the case of the Kronecker product we know from Lancaster and Farahat [11] that for matrices **A** and **B** the norm can be computed as:

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

We use this property of the Kronecker product to implement an optimized version of the `norm` function in Listing 4. This optimization greatly improves performance, as it prevents materialization of the Kronecker wrapper while reducing the size of matrices that need to be processed.

The approach from Listing 3 also composes with other lazy wrappers. For example, the Julia standard library avoids materializing matrix transpositions by using a `Transpose` wrapper that implements the expected indexing semantics. This wrapper type is also part of the `AbstractArray` hierarchy. It can hence be used as an input to our `Kronecker` type without materializing the wrapper. Other opportunities for composability will be discussed in Section 6.3.

3. The Julia perspective

The above examples are written in Julia, a high-level, high-performance dynamic programming language originally designed for technical computing [2]. This section explains how the design of this language and its run-time compiler enables our approach towards rapid prototyping for heterogeneous platforms.

The Julia programming language features a type system with parametric polymorphism, multiple dispatch, metaprogramming capabilities, and other high-level features [12]. Many of these features encourage code reuse. For example, code can be fully untyped, as

```
1  struct Kronecker{T,N,AT} <: AbstractArray{T,N}
2    A::AT
3    B::AT
4    function Kronecker(A::AT, B::AT) where
5                        {T, N, AT<:AbstractArray{T,N}}
6      new{T,N,AT}(A, B)
7    end
8  end
9
10 Base.size(K::Kronecker) = size(K.A) .* size(K.B)
11
12 function Base.getindex(K::Kronecker, i::Int, j::Int)
13   I,Ix = divrem(i-1, size(B,1))
14   J,Jx = divrem(j-1, size(B,2))
15   K.A[I+1,J+1] * K.B[Ix+1,Jx+1]
16 end
```

Listing 3. Declarative implementation of the Kronecker product of two matrices.

shown in the examples from Section 2. Such dynamically typed code is especially interesting during prototyping, where specifying types or reasoning about their hierarchy is an undesirable non-functional aspect.

Dynamically-typed code also makes it possible to reuse that code with differently-typed values, as long as all functions that are called are applicable for the argument types at hand. This is demonstrated in Listing 5. When the Julia runtime invokes such code, the Julia compiler *specializes* the code based on run-time type information. The generated machine code is specific to the types and hand, and consequently avoids the performance penalty of performing operations on boxed values or dispatching dynamically to other methods. Furthermore, code specialization makes it possible to generate statically-typed machine code, which is essential for heterogeneous computing devices such as GPUs.

Next to method specialization, multiple dispatch is another cornerstone of the Julia programming language. With multiple dispatch, function calls resolve to methods based on the run-time values of each of their arguments (as opposed to single-dispatch polymorphism as with C++ where only the run-time value of the first argument influences dispatch). Multiple dispatch allows programmers to write smaller method definitions with limited responsibilities [12]. This facilitates code reuse, as it enables fine-grained overloading of functionality when behavior needs to differ, e.g., when defining new types that are part of an existing type hierarchy.

Incorporating all arguments in dispatch also makes it possible to overload methods that would be out-of-reach with single dispatch. For example, Listing 6 defines a dual number type, an extension of real numbers with an epsilon component for the purpose of, e.g., automatic differentiation. Using multiple dispatch, we implement methods for algebraic addition and multiplication that propagate epsilon components by extending respectively the `+` and `*` functions from the standard library on line 12 to 16. The definition on line 14 is not possible in a single-dispatch language such as Python, where special methods `__mul__` and `__rmul__` exist specifically for the purpose of defining commutative multiplication as a workaround to overcome the limitations of single-dispatch. Such a workaround does not generalize, however, and fails to compose with optimized functionality such as matrix-matrix multiplication as implemented in NumPy. As a result, users would be forced to reimplement larger pieces of functionality, while complicating reuse of existing functionality. This pattern is especially common for operators, and the Julia standard library uses multiple dispatch extensively to implement these methods [2].

With the definitions from Listing 6, functionality from the standard library that relies on addition and multiplication can be reused, even if it combines dual numbers with other types. For example, line 22 shows the multiplication of a matrix with floating-point dual numbers and a matrix with integer complex numbers. The resulting matrix contains elements of type `Dual`, with complex floating-point values as real and epsilon components. This is a very powerful demonstration of code reuse, where the minimal definitions from Listing 6 compose with extensive functionality from the standard library that implements complex numbers, 2-dimensional arrays, and operations on these types.

4. Abstractions for array programming

This section discusses existing standard and higher-order array abstractions commonly used in Julia. They also form the basis of our approach as presented in later sections.

Many data science and engineering problems are commonly expressed in terms of vectorized operations, especially during initial prototyping. This natural, concise representation makes it easier to iterate over different prototype implementations. They avoid the typical non-functional boilerplate of scalar processing of data items such as specifying loop bounds, indexing calculations, etc. For example, the high-level code from Section 2 is written entirely using array abstractions, resulting in readable high-level code.

Many high-level languages such as R or Python require the use of array operations in order to achieve high performance. These operations are then implemented in a low-level, high-performance language. This illustrates the two-language problem as it exists with many high-level programming languages. The Julia programming language does not suffer from this problem, as the language has been co-designed with a JIT-compiler that generates high-quality machine code. The performance of scalar, loop-based programs is typically on par with implementations in a low-level language like C. As a result, the array operations themselves are also implemented in Julia [4], and do not require a low-level language to achieve high performance. This greatly

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

Listing 4. Optimized computation of the matrix norm for Kronecker products.

```
1  using LinearAlgebra
2  function user_code(a, b)
3    inv(a) * norm(b)
4  end
5
6  user_code(rand(2,2), rand(2,2))
7
8  using SparseArrays
9  user_code(rand(2,2), sparse(rand(2,2)))
10
11 user_code(1, 2)
12
13 user_code("", "")
14 # ERROR: no method matching inv(::String)
```

Listing 5. Illustration of code reuse through dynamic typing.

lowers to barrier to contributing to the Julia project or any of its packages. Indeed, the number of contributors to the main Julia language repository is greater than that of the Python reference implementation, despite the latter being a significantly older and well-known project.

At the same time, the availability of a JIT compiler enables powerful, higher-order abstractions that compose with arbitrary user code. The `reduce` abstraction is a prime example of such an abstraction. Listing 7 illustrates how the first argument to the `reduce` function can be any transformation function that reduces two scalar values. The JIT 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 Section 5 and 6, we will further focus on portability through the use of different array types.

4.1. 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 the type of the arguments, selecting an implementation that maximizes performance or otherwise preserves the array type, e.g., to prevent slow memory transfers from or to a heterogeneous computing device.

At its core, `map` transforms collections of identical shape and size by applying a function elementwise over the collections, as shown in Listing 8. The function should accept as many arguments as the amount of containers passed to `map`.

The `broadcast` abstractions generalizes the behavior of `map` to containers of heterogeneous shapes by padding dimensions

```
1  struct Dual{N<:Number} <: Number
2    re::N
3    ep::N
4
5    # constructor with default value for epsilon component
6    Dual{N}(re::N, ep::N=zero(N)) where {N} = new{N}(re, ep)
7  end
8
9
10 using Base: *, +
11
12 *(x::Dual, y::Dual) = Dual(x.re * y.re, x.ep*y.re + x.re*y.ep)
13 *(x::Dual, y::Number) = Dual(x.re * y, x.ep*y)
14 *(x::Number, y::Dual) = Dual(x * y.re, x*y.ep)
15
16 +(x::Dual, y::Dual) = Dual(x.re + y.re, x.ep + y.ep)
17 ...
18
19
20 A = Dual.(rand(Float64, 2,2))
21 B = rand(Complex{Int}, 2)
22 A * B
```

Listing 6. Illustration of multiple dispatch facilitating reuse by allowing fine-grained method overloads.

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

Listing 7. Example use of the `reduce` abstraction.

accordingly. This greatly improves use with objects of different shape. For example:

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

The `reduce` abstraction reduces a container by applying a binary function along certain dimensions of an array, e.g., to compute the sum of an array by calling `reduce(+, array)`. 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 5.

4.2. Dot expressions

To improve the usability of broadcast, so-called *dot expressions* can be used in Julia to denote elementwise transformations [13]. The Julia parser lowers this syntactic sugar to invocations of the `broadcast` function, as illustrated with some examples in Table 1. Elementwise assignments call the `broadcast!` function, which performs in-place assignment to avoid allocating an output container.

Dot expressions that contain multiple elementwise applications are syntactically fused together and result in a single application of the broadcast abstraction [13]. Semantically, the parser generates a new anonymous function that contains the scalar operations from each dot expression, as shown in Table 1. This not only reduces the number of invocations of the broadcast machinery, but also enables the compiler to optimize the fused expression, e.g., by eliminating common subexpressions. Fusion also obviates the need for temporary arrays: Intermediate values, such as the result of `a.+ b` in Table 1, now exist as scalar values within the fused function, and do not need to be stored outside of that.

As of Julia 1.0 dot expressions are represented *lazily* through a first-class data structure [14]. The `Broadcasted` data structure represents the tree of a broadcast expression and is accessible to implementers of broadcast at run time. This 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.

Table 1
Lowering of different forms of broadcast syntax.

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

5. Heterogeneous programming with arrays

Programming with the array abstractions from the previous section makes it possible for application code to only deal with *what* needs to be computed, while an underlying array type takes care of *where* the data is stored, and *how* the computations are performed. Now, we will describe how an array type for storage and execution on a heterogeneous device can be implemented. We hence temporarily switch to the side of the programming expert, who has to deliver such an implementation to support the needs of rapid-prototyping engineers.

At its core, every array type starts with a parametric type definition that subtypes the `AbstractArray` type. In the case of an array type that is backed by actual device memory, as opposed to, e.g., an array that performs a computation like the example from Listing 3, the type contains a number of member fields that provide handles to device memory. In Listing 9, we define such a `HeterogeneousArray` type that contains a single field, `handle`, to store a pointer to device memory. The constructor on line 8 accepts any array data as input, and uploads it to the device by using an the `to_device` function that is provided by the illustrative device back-end package DeviceBackend.jl. A counterpart function on line 11 implements conversion back to a CPU array, downloading from device memory using the `from_device` function.

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

```
1  a = [1 2; 3 4]
2  b = [3 4; 5 6]
3  c = [5 6; 7 8]
4
5  map(x->x+1,          a)
6  map(+,               a, b)
7  map((x,y,z)->x+y+z,  a, b, c)
8
9  broadcast(+,         a, 1)
10 broadcast(+,         a, [-1; 1])
```

Listing 8. Example use of the `map` and `broadcast` abstractions.

```
using DeviceBacked: to_device, from_device

struct HeterogeneousArray{T,N} <: AbstractArray{T,N}
  # member field storing handle to device memory
  handle::Ptr{T}

  # constructor
  HeterogeneousArray(data::AbstractArray{T,N}) where {T,N} = new{T,N}(to_device(data))
end

Base.convert(::Type{Array}, array::HeterogeneousArray) = from_device(array.handle)
```

Listing 9. Storage handling for a heterogeneous array type.

```
using DeviceBacked: to_device, from_device

function Base.setindex!(array::HeterogeneousArray, i::Integer, value)
  to_device(array.handle, i, value)
  return
end

function Base.getindex(array::HeterogeneousArray, i::Integer)
  return from_device(array.handle, i)
end
```

Listing 10. Scalar indexing for a heterogeneous array type.

parameters are deduced by the constructor from the input data.

As part of the `AbstractArray` interface, custom array types should implement certain functionality, such as the `getindex` and `setindex!` methods to fetch and to store scalar elements from the array. Examples of these methods are defined in Listing 3 and 8 of Listing 10 where we rely on versions of the `from_device` and `to_device` functions of the device backend package to load from and store to device memory.

These scalar access methods are useful because they provide compatibility of the array type with existing code that explicitly iterates over the elements of arrays. For example, the "default" definition of matrix multiplication for `AbstractArray`s in the Julia standard library, which is designed for execution on a host CPU, uses the textbook algorithm with nested for loops that multiply and accumulate matrix elements. When that matrix multiplication is invoked on an array of type `HeterogeneousArray`, it still computes the correct result, albeit it very slowly: The nested loops is still executed on the host CPU, and every element accessed in the array on the device is transferred individually from the device to the host. This obviously is very slow, and defeats the entire purpose of auxiliary hardware devices. Still, it provides compatibility with existing scalar code. Such code can then be incrementally ported to use array abstractions, and the results can be verified at every step. This will be further illustrated in Section 6.

For an array type to be usable for engineering purposes, it has to provide efficient versions of relevant array abstractions. As detailed in Section 3, the design of the Julia programming language facilitates such overloads. In Listing 11 we demonstrate how a custom array type can implement a generic matrix-matrix multiplication that replaces the aforementioned generic, scalar version of the standard library. The example uses the `@on_device` macro provided by the DeviceBackend.jl package to mark code that should be executed on the device. Note that the implementation is still fully generic. It can be used with any element type (e.g., the `Dual` number type from Section 3) as long as multiplication and addition are defined for the type. When this method is invoked, the run-time compiler specializes the code on the actual run-time arguments, i.e., concrete instances of `HeterogeneousArray` with values for the `T` and `N` type parameters, and on the execution context, i.e., `@on_device`. The illustrative operations on lines 7 and 8 are syntactic sugar underneath of which the abstract `getindex` and `setindex!` as implemented in Listing 10 are still used. But in this context, they are executed on the device. This obviates the transfers of the elements to and from the host processor. In the `@on_device` context, the `from_device` and `to_device` are specialized to direct accesses in the device memory, and the computations are performed directly on the device. How this is achieved technically is out of the scope of this paper.

Abstractions such as matrix multiplication from Listing 11 as implemented for a heterogeneous array type define both *what* is executed, *where*, and *how*. By contrast, higher-order abstractions such as `broadcast` from Section 4.1 make the user responsible for specifying only *what* is computed. Section 6.3 will discuss how this makes it possible to compose different array types, where each type deals with

```
using DeviceBacked: @on_device

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

Listing 11. In-place multiplication for a heterogeneous array type.

```
using DeviceBacked: @on_device

function Base.copyto!(dest::HeterogeneousArray, op::Broadcasted)
  @on_device begin
    I = CartesianIndex(dest)
    dest[I] = op[I]
  end
end
```

Listing 12. Implementation of the broadcast interface for a heterogeneous array type.

different aspects of the computation. To support the broadcast abstraction, the example from Listing 12 provides an implementation of the `copyto!` function for `HeterogeneousArray` when it is also passed a `Broadcasted` tree. This method is responsible for executing a flattened representation of broadcast expressions in the context of a certain array type, and is part of the interface that makes up the broadcast interface. In the case of our `HeterogeneousArray` type, we make sure this operation happens on the device by using the `@on_device` macro.

The following sections discuss two concrete packages for which we relied on the discussed types of interfaces to provide array types for programming heterogeneous devices: CuArrays.jl for NVIDIA GPUs, and DistributedArrays.jl to program multiprocessor systems.

5.1. CuArrays.jl

The *CuArrays.jl* package [6] 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 4 are implemented on top of CUDAnative.jl [5], 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 [5].

Availability of a GPU compiler like CUDAnative.jl not only enables abstractions that compose with user code, but also extends the applicability of other operations. For example, matrix multiplication as implemented by cuBLAS only supports certain real and complex element types, and is limited to specific dense memory layouts. CuArrays.jl also provides a generically-typed implementation of matrix multiplication, similar to the aforementioned textbook implementation, but optimized for GPUs. Because the implementation is generically typed, it is applicable to all element types that define multiplication and addition and supports every memory layout with well-defined indexing semantics. This is relevant to, e.g., the example from Listing 2, where derivatives are computed by the ForwardDiff.jl package through a dual number type. This requires all used array operations, which includes matrix multiplication, to be applicable to arrays of such element types.

As an example of the low-level kernel programming interface, Listing 14 shows how to compute an element-wise addition of two `CuArray` GPU arrays using CUDAnative.jl. This package works at an abstraction level similar to CUDA C, where the programmer needs to provide a kernel function to be executed in parallel according to the Single Program, Multiple Data (SPMD) programming model. The `vadd` function on line10 is such a function, and is launched on line 16 at which point the CUDAnative.jl compiler specializes the function on the types of its arguments in order to generate efficient and GPU-compatible code. Although this is a low-level and explicit interface for programming a GPU, the kernel is still written in high-level Julia code: Kernels are generically typed and specialized upon first use, high-level language features such as metaprogramming or parametric types are available, etc. This greatly improves the productivity of kernel programming and makes it possible to reuse code that is independent from the specific execution environment [5]. This includes most of the Julia standard library, external packages that are not tied to CPU execution, and even vendor-neutral GPU kernels as implemented by the GPUArrays.jl package. At the same time, CUDAnative.jl still requires the developer to understand the SPMD model, and the performance characteristics of the underlying GPU hardware.

Line 7 of Listing 14 is an alternative, but semantically equivalent, high-level vector addition that uses CuArrays.jl to program the GPU. It uses the dot syntax from Section 4.2 as a shorthand for calling the `broadcast` function with a simple scalar function (here, `+`). This completely avoids the need to provide a SPMD kernel. The example demonstrates how users can use the `CuArray` type with powerful, higher-order abstractions that often obviate manual kernel programming. However, when flexibility is required, it is still perfectly possible to go deeper and use CUDAnative.jl to create custom SPMD kernels as with Listing 13. Both approaches can perfectly coexist in a single application.

```
using CuArrays

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


using CUDAnative

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

@cuda threads=4 vadd(c, a, b)
```

Listing 13. Low-level addition of GPU arrays using kernel programming interfaces from CUDAnative.jl.

```
1 using CuArrays
2
3 a = CuArray(rand(2,2))
4 b = CuArray(rand(2,2))
5 c = similar(a)
6
7 c .= a .+ b
```

Listing 14. High-level alternative to Listing 13, adding two GPU arrays using broadcast from CuArrays.jl.

Under the hood, the implementation of `broadcast` for `CuArray` transforms the scalar transformation to a valid SPMD kernel. Listing 15 shows a part of that implementation from the CuArrays.jl package. As explained in Section 5, the `copyto!` method is responsible for executing a broadcast expression in the context of a specific array type, here `CuArray`. The implementation defines an anonymous kernel on line 6, which calculates array indices using GPU intrinsics in accordance with the dimension-matching semantics of the broadcasting abstraction. The kernel is subsequently executed in parallel on line 14 using CUDAnative.jl. This is similar to the low-level use of CUDAnative.jl as shown in Listing 14. Note, however, that only the developers of CuArray.jl need to face this level of complexity; users of the package are spared of it.

Finally, high-level abstractions can also improve performance. As mentioned in Section 4.2, the Julia parser syntactically fuses multiple broadcast expressions together, resulting in fewer calls to the `copyto!` method from Listing 15. In the context of GPU programming, the advantages of broadcast fusion are profound: fewer kernel launches are required, memory allocations for temporary outputs can be avoided, and temporaries live on the stack and do not have to be loaded from global memory.

5.2. *DistributedArrays.jl*

The *DistributedArrays.jl* package builds upon Julia's distributed computing infrastructure to provide a Global Array-like interface [15]. 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 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 6.3 will show how this patterns makes it possible to compose array types that, like `DArray`, wrap other arrays.

Listing 16 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 16 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. 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.

```
1  using CUDAnative
2
3  function Base.copyto!(dest::CuArray, bc::Broadcasted)
4    op = Broadcast.preprocess(op)
5
6    function kernel(dest, op::Broadcasted)
7      i = (blockIdx().x-1) * blockDim().x + threadIdx().x
8      I = CartesianIndex(i)
9      dest[I] = op[I]
10     return
11   end
12
13   numthreads, numblocks = ...  # heuristic to maximize occupancy
14   @cuda threads=numthreads blocks=numblocks kernel(dest, op)
15
16   return dest
17 end
```

Listing 15. Low-level implementation of one of the methods that implement the broadcast abstraction, taken from CuArrays.jl.

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

Listing 16. Low-level implementation of in-place `map` taken from DistributedArrays.jl.

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

using DistributedArrays

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

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

Listing 17. High-level use of the `map!` abstraction with distributed arrays from DistributedArrays.jl

The distributed computing abstractions as used in Listing 16 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 [16], the Portable Batch System [17], 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 [18]. 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 16 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 17 shows how to use the `map!` abstraction from Listing 16 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.

6. Code portability

This section discusses how the examples from Section 2 and other codes can be ported to other platforms and environments by using the array types from Section 5. Section 6.1 focuses on the portability of standalone applications with respect to different array implementations for different heterogeneous platforms. Section 6.2 focuses on libraries that provide domain-specific functionality using array abstractions, for use in standalone applications and/or in compositions with other libraries. Such libraries should be generic with respect to array types not to hinder the portability of the applications or other domain libraries in which and with which they are used. Finally, Section 6.3 focuses on the portability and composability of libraries that define new array types and/or extend existing array abstractions.

6.1. Application portability

Array-based application code that does not rely on library functionality, such as the example from Listing 1, can be ported trivially. It suffices to use an appropriate array type by changing the array allocations to use a different constructor, for example, `CuArray(...)` instead of `Array(...)`. Operations on these arrays then dispatch to respective implementations in the corresponding array package. If that package does not provide certain operations, fallback methods from the Julia standard library are used. For example, when passing a `CuArray` to the `domeigen` function from Listing 1, the call to `rand!` dispatches to an optimized implementation in CuArrays.jl that uses the cuRAND library. Similarly, the multiplication on line 11 is lowered to a call to `mul!`. Several implementations of `mul!` are provided in CuArrays.jl, using the cuBLAS library when possible but falling-back to a generic matrix-matrix multiplication when required for, e.g., element types that are not supported by cuBLAS. This implementation is written in Julia, and uses CUDAnative.jl to compile code for the GPU and to execute it on the GPU.

In the case of array types that support computations with user code, we can also use code that is built around the higher-order array abstractions from Section 4.1. These abstractions compose with user code, and require the ability to generate code for the hardware that is targeted by the array type. For example, we can take the example from Listing 17 and change the call to `distribute` to create a `CuArray` instead. The CuArrays.jl package uses CUDAnative.jl to generate code for NVIDIA GPUs. Similarly, we can take the example from Listing 14 and execute it with arrays of type `DArray{Array}`, which would result in distributed execution on the CPU. `DArray` itself does not execute the user code but defers to the inner `Array`, which uses the Julia compiler to generate code for the CPU.

Application code can also perform scalar iterations over array elements, either because the application code is written that way or because (standard) library operations used in the application code are implemented as such. As explained in Section 5, this type of iteration defeats the purpose of heterogeneous programming as it cannot be implemented efficiently. Still, packages like CuArrays.jl and DistributedArrays.jl support this type of iteration because it greatly simplifies the effort of porting code. Initially, one can run the application on heterogeneous hardware without any change to the code, to verify

the functional correctness of the implementation. Subsequently, performance can be improved by reimplementing methods that rely on scalar iteration using array abstractions that can be executed efficiently on heterogeneous hardware. Identifying the methods that need to be reimplemented is facilitated by API calls that disallows scalar iteration. For example, both CuArrays.jl and DistributedArrays.jl provide a configuration value `allowscalar` that, when set to `false`, triggers errors upon use of inefficient scalar functionality.

Typical applications also contain multiple allocation sites. For example, the `domeigen` function from Listing 1 takes not only an array as argument, but also allocates an output container for the resulting eigenvector. To avoid hard-coding an array type, Julia provides functions such as `similar` to allocate new containers based on existing ones. These functions make it possible to write generic code that is independent from the chosen array type. The Julia standard library is built on top of these generic programming approaches, and rapid prototyping engineers can also use it, to facilitate reuse with different array types.

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

6.2. Library portability

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

Library code that works with arrays behaves similarly to application code as described in Section 6.1. As long as the library only uses functionality mandated by or implemented for `AbstractArray`, and allocates new containers using generic functions like `similar`, it is possible to reuse the library code with different array types.

However, where application code is often untyped, library code typically specifies types for function arguments [3]. For code to be portable, i.e., reusable with different array types, these signatures should use abstract array types such as `AbstractArray` or `AbstractSparseVector` and not their concrete CPU instantiations such as `Array` or `SparseVector`.

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

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

As a concrete library example, consider the already mentioned ForwardDiff.jl package. It implements methods to compute different kinds of derivatives of arbitrary user-defined computations on arrays and their elements [9]. For example, in the machine-learning example from Listing 2 the `gradient` and `derivative` functions are used to differentiate the loss function of a model for use by a gradient descent optimization algorithm. The ForwardDiff.jl package is an example of a high-quality, type-generic library. Simply changing the type of the arrays as passed to the derivatives makes the example from Listing 2 work on, e.g., a GPU, without requiring any other changes to either the code in Listing 2 or the underlying library.

However, the performance of the standard implementation of the ForwardDiff.jl package was not optimal when used with heterogeneous array types. To identify functionality that needs to be optimized, we disabled scalar iteration as described in Section 6.1. This revealed that certain methods of the `ForwardDiff.seed!` function were implemented using scalar for loops, one of which is shown in lines 2 to 8 of Listing 18. By reimplementing those methods using array abstractions (lines 11 to 15) they are better suited for execution on, e.g., a GPU. In this case, the replacement uses a `broadcast` expression as a substitute for the scalar for loop. The replacement code is certainly not more complex.

When the need to redefine a library function to obtain higher performance in a specific application arises, either during or after the rapid-prototyping phase, the redefinition does not necessarily needs to happen in the library itself. It can also be done in the application, by prefixing the function name with the contained module. For example, to implement the replacement of Listing 18 in an application rather than in the ForwardDiff.jl library, it suffices to write it down as `function ForwardDiff.seed!... end`. When a replacement definition in an application has exactly the same signature as the original definition in the library, the replacement overrides the library version.

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

```
1  # original, scalar implementation
2  function seed!(duals::AbstractArray{Dual{T,V,N}}, x,
3                 seeds::NTuple{N,Partials{N,V}}) where {T,V,N}
4      for i in 1:N
5          duals[i] = Dual{T,V,N}(x[i], seeds[i])
6      end
7      return duals
8  end
9
10 # replacement broadcasting version
11 function seed!(duals::AbstractArray{Dual{T,V,N}}, x,
12                seeds::NTuple{N,Partials{N,V}}) where {T,V,N}
13     duals[1:N] .= Dual{T,V,N}.(x[1:N], seeds[1:N])
14     return duals
15 end
```

Listing 18. Reimplementation of a method from ForwardDiff.jl using array abstractions.

functionality deep down the library as opposed to only functions that are called directly by the library.

Furthermore, the original definition in the library can easily be kept available for the purpose of verifying the replacement implementation. It suffices to use a dispatch signature that is limited to the heterogeneous array type of choice to avoid that the original definition is overridden. For example, by changing the definition on line 11 to specify `CuArray` instead of `AbstractArray` for the first argument, the broadcasting version would only be used for GPU arrays, and the known-good library implementation remains available, and can be used on `Array` types to verify the semantical equivalence of the original and the replacement definitions.

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

6.3. Array infrastructure portability

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

6.3.1. Kronecker products on the GPU

The example from Listing 3 uses a custom array type for efficiently computing the Kronecker product of two matrices, and provides an optimized implementation of the `norm` function computing the matrix norm using properties of the Kronecker product to improve performance. The `Kronecker` array type is generically typed, and only requires that the two input matrices should be part of the `AbstractArray` type hierarchy. No so-called *glue code* is required for the `Kronecker` type to work with concrete array types.

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

With only the `getindex` function for scalar indexing defined, array operations with objects of type `Kronecker{...}` dispatch to generic implementations as described in Section 5. However, any optimized method that calls functions on the underlying containers compiles to specialized code that uses functionality optimized for the contained array type. For example, with a Kronecker product of `CuArray`s and the optimized but still generically-typed implementation of the matrix norm from Listing 3, calls to the `norm` function result in an execution that combines the properties of the Kronecker product that allow for an efficient calculation of the norm with a well-optimized GPU implementation of the Euclidean norm that is available in the CuArrays.jl package and that in turn invokes the cuBLAS library. This powerful example illustrates how multiple array types, each dealing with separate concerns, seamlessly compose together to form a high-performance interface that can still be used generically.

Ideally, it should also be possible to use the broadcast abstraction from Section 4 in combination with custom array types. However, currently that does not yet work out of the box. One problem is the implementation of the type hierarchy in relation to broadcasting when wrappers are combined. For example, `Kronecker{CuArray}` is an `AbstractArray`, but not a `CuArray`. In the current language implementation, the compiler's use of available methods optimized for `CuArray` to specialize code depends on the presence of certain artifacts in the `Kronecker` class method implementations, such as whether or not those (by accident) defer explicitly to the inner `CuArray`. That dependency on the occurrence of those artifacts should be avoided, as it violates the separation of concerns and limits composibility and performance portability in ways a non-expert programmer cannot easily handle.

We expect this situation to improve in the future, since heavy use of array wrapper types is relatively new, and the current broadcast infrastructure has been designed as recently as Julia 1.0. For now, array packages such as CuArrays.jl and DistributedArrays.jl provide the necessary definitions for common array wrappers, such as the ones from the Julia standard library, to work as expected.

6.3.2. Distributed GPU arrays

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

Similar to the example in the previous section, 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 16 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.

7. Performance evaluation

This section analyzes the performance of different array types applied to the examples from Section 2. We work with the latest stable version of Julia, 1.0.1, using the official binaries from the homepage. For auxiliary packages, we also used the latest released versions at the time of writing: CUDAdrv 0.8.6, CUDAnative 0.9.1, and ForwardDiff 0.9.0. In the case of array packages, CuArrays.jl and DistributedArrays.jl, we used development branches to incorporate fixes and improvements to the array types that we developed while working on this paper.

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

Code that targets the CPU by using the `Array` or `DArray{Array}` types is allowed to take advantage of the supported 16 simultaneous CPU threads. In the case of `Array`, this is done by configuring the OpenBLAS library that empowers many of the array abstractions as implemented for `Array` to use 16 threads. This implies a single-process multi-threaded parallelization. In the case of `DArray{Array}`, single-threaded multi-process parallelization is used instead. This is done by configuring the Distributed standard library that is used by DistributedArrays.jl to launch 16 worker processes, while OpenBLAS is configured to use only thread per process to avoid oversubscription of the system. For measurements with a single GPU, we use the GeForce GTX 1080 in a single process. When targeting multiple GPUs, e.g., with the `DArray{CuArray}` type, we use one worker process per GPU.

We used the BenchmarkTools.jl package to collect accurate timings for the experiments in this paper [19]. Measurements are performed on an otherwise idle system, after tuning in order to determine the required execution and sample count for each experiment to yield accurate timings. In the charts below, we report the mean execution time.

The performance evaluation below is limited in scope. We rely on existing array packages to perform well in the contexts they were designed for, i.e., CuArrays.jl for GPU execution and DistributedArrays.jl for execution on multi-core CPU computers and distributed systems. The measurements in this section serve to illustrate how the realistic problems from Section 2, built on top of array abstractions from Section 4, can be used with the array types from Section 5 to effortlessly program heterogeneous systems and to benefit from the increased performance and/or enlarged scale these systems provide. This does not necessarily imply optimal or efficient use of the hardware, but we will show that our approach facilitates that goal.

7.1. Power iteration

The example from Listing 1 is a simple application that uses array abstractions. It can trivially be executed with a variety of array types, for which it suffices to change the initial allocation site. Fig. 1 shows how the execution time of the `domeigen` function evolves with the problem size. This time includes all run-time overhead such as the time to allocate output buffers, launch GPU kernels, and communicate data across compute nodes.

The results in Fig. 1 highlight several performance characteristics. First of all, it is clear how regular multi-threaded `Array`s have very low overhead, and scale with increasing problem size as would be expected

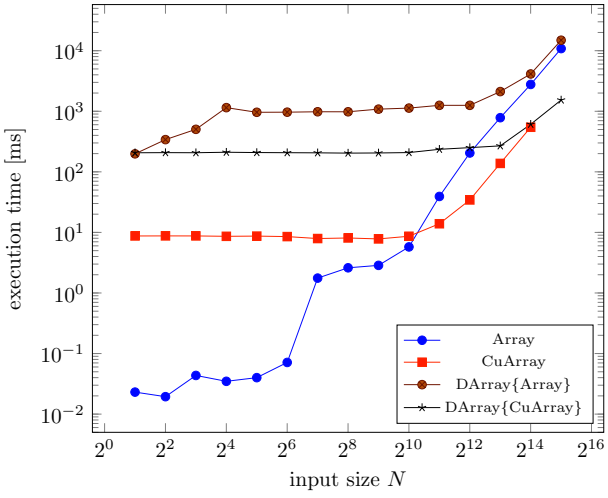

Fig. 1. Time to execute the `domeigen` function from Listing 1 and compute the dominant eigenvector and eigenvalue of a $N \times N$ matrix. We benchmark for 1000 iterations of the power method, approximating the reference eigenvalue with sufficient accuracy.

from working with $N \times N$ arrays. A distributed `DArray{Array}` works with multiple processes that require Inter-Process Communication (IPC). This comes with a significant overhead that will be discussed below, but with large problem sizes the performance shows to scale identically to multi-threaded arrays that do not require IPC. This shows how the use of `DArray{Array}` is viable for large problems, where performance of multiple processes with IPC is comparable to that of a multi-threaded application that does not require communication.

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

As GPUs typically have small memories, they are limited in the amount of data that can be processed. Although certain operations can be implemented with so-called out-of-core algorithms that support working set sizes larger than the available memory, and features like CUDA Unified Memory make it possible to do so without significantly changing code, these approaches come at a large performance cost [20–22]. We did not employ such techniques in the reported experiments. For that reason, the `CuArray` measurements stop at input size 2^{14}. The alternative solution of using multiple GPUs to extend the available memory requires careful management of data in order to reach good levels of performance. This data management has already been developed as part of DistributedArrays.jl, so we reuse that functionality via objects of type `DArray{CuArray}` to distribute data automatically across GPU devices. Fig. 1 shows how this again comes with a large initial overhead for small input sizes, but ultimately the approach scales past the limits of using a single GPU and delivers performance that is better than the projected performance of using a single GPU past its maximal problem size, consistent with the increase in computing power that arises from using multiple GPUs. It shows how multiple GPUs can be easily used together to extend the supported working set size of an array application, while further improving performance despite inefficiencies in the current IPC implementation.

Similarly, DistributedArrays.jl can be used to scale past single computers without changes to the application, by using one of the cluster managers as explained in Section 5.2. This makes it possible to support working set sizes that exceed the available main memory, and to improve performance by adding more computational power than a single computer has to offer. As we did not have such a system at our disposal at the time of writing this paper, we do not present measurements for a distributed system that consists of multiple computers.

7.2. Performance characteristics of DistributedArrays.jl

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

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

way to move to a parallel thread runtime where this would not be an issue.

Another slowdown is due to the many data copies occurring as part of IPC. The vector-matrix product on line 11 of Listing 1 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 Listing 1 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 Listing 1 with `DArray{CuArray}`, the performance benefit of using GPUs overcomes that overhead.

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

7.3. Kronecker product

Computation of the Kronecker product from Listing 3 illustrates a scenario where much less communication is required. The Euclidean norm can easily be computed on local parts of the input arrays, after which the partial scalar results can be communicated and used to compute the total norm. Fig. 2 show how this does not affect measurements with multi-threaded `Array`s, which do not require interprocess communication. The timings hence scale quadratically, as would be expected from processing the Kronecker product of $N \times N$ arrays.

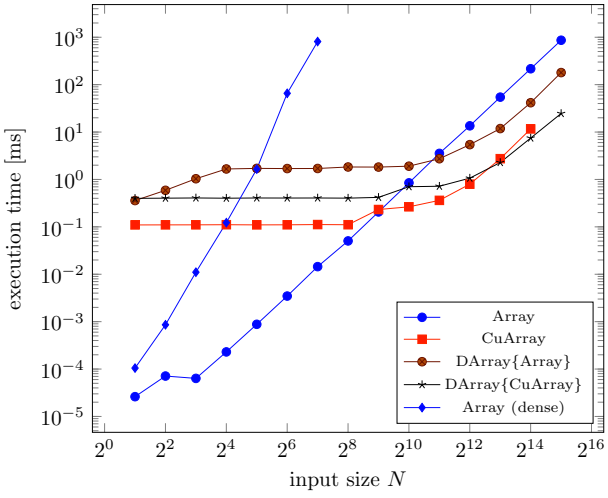

Fig. 2. Time to compute matrix norm of the Kronecker product of two $N \times N$ matrices. Measurements marked with "dense" first compute the Kronecker product in full, while other measurements uses the structured matrix type from Listing 3 and the accompanying norm calculation from Listing 4.

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

With fewer communication demands, the measurements for distributed CPU arrays using `DArray{Array}` show a much smaller overhead than were observed for the power iteration example. For significantly large problem sizes, not only is the scaling behavior identical to that of multi-threaded `Array`s that do not require interprocess communication, the performance is in fact higher. This shows that the distributed `DArray{Array}` is not only interesting for extending the working set size using distributed systems, but that it can also improve performance on a single computer as long as the application does not require significant IPC. This performance improvement can be explained by the Non-Unified Memory Access (NUMA) architecture of our 2-processor system. In the case of `Array`, the entire array is allocated once on one of the NUMA nodes and processing from threads on a different NUMA node results in relatively slow memory accesses. With DistributedArrays.jl, data is explicitly partitioned across workers on the system. This results in data allocated in the local NUMA node, therefore minimizing memory traffic across NUMA zones.

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

7.4. Proximal gradient descent

In Section 7.2 we mentioned a major performance penalty in the current implementation of DistributedArrays.jl due to inefficiencies with IPC. This is particularly noticeable in the machine learning example from Listing 2, where the main computational cost comes from matrix-vector multiplications as part of the `proximal_gradient_descent` method. These operations require significant communication, which is troublesome given the current implementation of IPC in DistributedArrays.jl. Indeed, Fig. 3 shows how distributed execution with `DArray`s is completely dominated by the cost of communication, and even drowns out any performance benefits that come from using GPU hardware. In contrast, local execution with `CuArray` shows significant run-time improvements compared to CPU-based `Array`s, but is limited in terms of the working set size. As such, while the performance of distributed execution is far from optimal at this point, it makes it possible to scale beyond single devices and benefit from, e.g., the increase in available memory.

This example illustrates how application performance and potential improvements of using different array types are currently subject to application characteristics and how those influence the (composition of) the underlying array libraries. For example, Fig. 3 shows how the example from Listing 2 benefits significantly from using a GPU, but currently does not improve when executed on a distributed system due to the heavy use of IPC. The example from Listing 1 does not rely as much on IPC, and Fig. 1 shows how it benefits from using multiple GPUs in a distributed setting. At the other end of the spectrum, the example from Listing 3 does hardly use any IPC and as a result Fig. 2 shows how use of distributed CPU and GPU resources yields significant speedups.

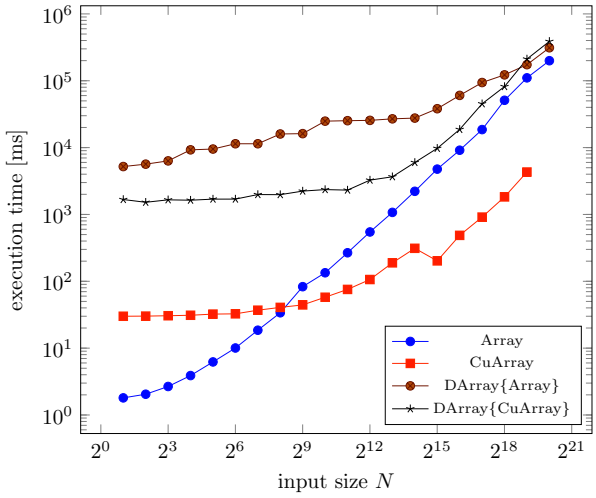

Fig. 3. Time to perform 25 iterations of proximal gradient descent from Listing 2 to optimize a network of 100 parameters for N outputs. The implementation uses linear regression as a user defined model and performs enough iterations for the loss to reach 0.01 given random inputs that are normally-distributed around 0 with a standard deviation of 1.

7.5. Optimization opportunities

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

After the initial prototyping phase in other high-level languages, developers typically rewrite (part of) their code in a high-performance language. With a high-level language that is designed for performance, as Julia is, this translation step can be avoided. Instead the Julia language features great performance from the get go, and makes it possible to optimize code within the language itself to the point where it reaches or even goes beyond the performance of statically compiled languages such as C or Fortran [1].

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

In the remainder of this section, we discuss two different types of optimizations and their impact on examples from this paper.

7.5.1. Array programming

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

As an example, consider how every iteration of the `for` loop in the `domeigen` function in Listing 1 allocates two temporary containers to store the outputs of the operations on lines 11 and 14. Listing 19 shows an alternative version of the code that pre-allocates two containers before the loop and uses in-place operations to prevent new allocations. This trivial optimization significantly improves performance, especially in the case of small inputs where the overhead of allocating memory is similar to the run time of the actual array operations. For example, with CPU arrays of size 64×64 or smaller, this optimization improves performance by up to 15%. Furthermore, the change is fully generic and equally applies to other array types. With `CuArray`s, where memory allocations aren't backed by a high-performance garbage collector, the improvements are about 5% for all matrix sizes as used in this evaluation.

7.5.2. Multiple dispatch

Beyond optimizing the use of array abstractions, it is always possible to use multiple dispatch for providing fine-grained method overloads that optimize critical pieces of underlying functionality. One obvious example as discussed in Section 6 are method overloads that avoiding scalar iteration, e.g., in an underlying library that is used by the application. Although the main purpose of these overloads is to improve performance when working with heterogeneous computing devices, the implementations are often generic and can be used for all array types.

Method overloads can also be specific to an array type, and provide functionality that only optimizes execution for that type. For example, the ForwardDiff.jl library as used in Listing 2 performs partial derivative evaluations on the input vector in chunks [9]. Performing the evaluations on small smaller chunks uses less memory but requires more evaluations of the target function. In the case of GPU execution, larger chunks also require more registers, which might result in inefficient use of the GPU's parallel compute units. ForwardDiff.jl uses a heuristic to optimize the chunk size and minimize the amount of chunks given the size of the input vector. Listing 20 shows how to override that heuristic for GPU arrays by hard-coding an empirically-chosen chunk size that performs well given a specific application and GPU. Note that a production-quality version of this function would need to specialize on

```
1  function domeigen(A, p)
2    ...
3
4    # power iteration
5    b_k = b_0
6    b_k+1 = similar(b_k)
7    for _ in 1:p
8      mul!(b_k+1, A, b_k)
9
10     # normalize
11     b_k .= b_k+1 ./ norm(b_k+1)
12   end
13
14   ...
15 end
```

Listing 19. Optimization of the power iteration loop from Listing 1, using pre-allocated buffers and in-place array operations.

```
using CuArrays
using ForwardDiff: Chunk, DEFAULT_CHUNK_THRESHOLD

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

Listing 20. Optimizing the use of ForwardDiff.jl from Listing 2 for GPU execution.

the performance characteristics of the GPU hardware that backs the input array.

8. Related work

In this paper we 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 [23], 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 [1] and causes composability [24] and extensibility problems. Once developers exhaust the functionality of the library and require custom functionality, e.g., because they want to take advantage of problem-specific knowledge as shown in Section 2.3, the library approach starts to break down and they have to resort to writing their code in the low-level language. Numba [25] 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 [14].

The idea of separating the algorithm (what to compute) from the schedule (how and where to compute) is most prominent in Halide [26,27]. 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 [28] 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 [29] and PyTorch [30] 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 previous work, we discussed the reason for this failure of composability [31] 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 [32], IBM's X10 [33], and Fortress [34], 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 [35], Thrust [36], OpenMP [37], OpenACC [38] and several others. There are also several approaches for distributed programming like MPI [39], Legion[40], and UPC++[41]. Trilinos [42], PetSC [43], and Kokkos [44], 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.

9. Conclusion and future work

Conclusion

We have shown the initial promise of a programming model that is particularly well suited for rapid prototyping, gradual performance improvements, and taking advantage of heterogeneous computing resources to tackle problems at scale. It realizes our vision of a unified development environment without walls between rapid prototyping and performance engineering. We have demonstrated that non-trivial applications can be expressed with array abstractions as offered by the Julia programming language, and how that enables portability through the use of different array types.

Our work on CuArrays.jl and DistributedArrays.jl has made it possible to execute realistic array applications on respectively GPUs and distributed systems. The presented GPU array type builds on our previous work for compiling Julia code for GPUs [5], and makes it possible to program the hardware without any knowledge of GPU programming. We also show how these array packages compose, and make it possible to target distributed CPUs and GPUs alike.

Status

The initial focus of our work has been on usability and functionality, supporting common abstractions while offering mechanisms for incrementally porting existing code. Nonetheless, performance improvements of applications that use our work are often significant, albeit inconsistent and dependent on the application characteristics and implementation details of the underlying array library. Specifically, distributed arrays suffer from several performance deficiencies, but prove useful to scale past individual systems and benefit from the extended memory and computing power that distributed systems have to offer.

Our work is compatible with the latest version of Julia, and our contributions to the packages as used in this paper are open-source and have been integrated with the upstream development branches.

Future work

Work is under way to improve performance problems with distributed arrays as discussed in Section 7.2. This includes moving to a parallel thread runtime to prevent worker starvation, and improvements to the communication primitives to support high-performance mechanisms such as RDMA and NVLink.

We are also working on generalized handling of array wrappers in order to avoid the composability issues as discussed in Section 6.3. These issues are equally relevant to regular Julia code, outside of heterogeneous programming, where array wrappers are increasingly used as a means to implement operations on arrays efficiently. Case in point, other developers have recently redesigned the broadcast infrastructure to better accommodate for deep array hierarchies, and to allow fine-grained decisions at each level on how broadcasts are processed. In turn, that work is valuable for heterogeneous programming, e.g., to control which broadcast expressions are fused into GPU kernels. Exploiting the synergies between all ongoing developments in this regard still requires some work.

Finally, there is ongoing work to generalize the abstractions from Section 4 in order to represent most if not all of the common tensor comprehensions. This includes the development of tensor and stencil compilers in Julia, and we predict that such developments could take advantage of the general strategy outlined here to separate the individual concerns that involve heterogeneous computing.

Acknowledgments

We are grateful to Peter Ahrens and Jarret Revels for valuable discussions and comments. We would also like to thank the Julia community at large, and in particular Jameson Nash, Matt Bauman, Andreas Noack, Chris Rackauckas and Yingbo Ma for their input on these subjects.

This research is supported in part by NSF DMS-1312831, NSF OAC-1835443, Darpa XDATA, and an ARAMCO MITEI grant. Tim Besard is supported by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT), the Research Foundation – Flanders (FWO 3G051318), and by Ghent University through the Concerted Research Action on distributed smart cameras.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.advengsoft.2019.02.002.

References

[1] Bezanson J, Karpinski S, Shah VB, Edelman A. Julia: a fast dynamic language for technical computing. 2012. arXiv:https://arxiv.org/abs/1209.5145.
[2] Bezanson J, et al. Julia: a fresh approach to numerical computing. SIAM Rev. 2017;59(1):65–98.
[3] Bezanson J, Chung B, Chen J, Karpinski S, Shah VB, Vitek J, et al. Julia: dynamism and performance reconciled by design. Proceedings of the international conference on object oriented programming systems languages and applications. ACM; 2018. To appear
[4] Bezanson J, Chen J, Karpinski S, Shah V, Edelman A. Array operators using multiple dispatch: a design methodology for array implementations in dynamic languages. Proceedings of ACM SIGPLAN international workshop on libraries, languages, and compilers for array programming. ACM; 2014.
[5] Besard T, Foket C, De Sutter B. Effective extensible programming: unleashing Julia on GPUs. IEEE Trans Parallel Distrib Syst 2018.
[6] Julia developers. CuArrays.jl: CUDA-accelerated arrays for Julia. 2018a. https://github.com/JuliaGPU/CuArrays.jl.
[7] Julia developers. DistributedArrays.jl: distributed arrays in Julia. 2018b. https://github.com/JuliaParallel/DistributedArrays.jl.
[8] Mises R, Pollaczek-Geiringer H. Praktische verfahren der gleichungsauflösung. J Appl Math Mech (ZAMM) 1929;9(1):152–64.
[9] Revels J, Lubin M, Papamarkou T. Forward-mode automatic differentiation in Julia. 2016. arXiv:https://arxiv.org/abs/1607.07892.
[10] Hogben L. Handbook of linear algebra. Chapman and Hall/CRC; 2006.
[11] Lancaster P, Farahat H. Norms on direct sums and tensor products. Math Comput 1972;26(118):401–14.
[12] Bezanson J. Abstractions in technical computing. Massachusetts Institute of Technology; 2015.
[13] Johnson SG. More dots: syntactic loop fusion in Julia. 2017. https://julialang.org/blog/2017/01/moredots.
[14] Bauman M. Extensible broadcast fusion. 2018. https://julialang.org/blog/2018/05/extensible-broadcast-fusion.
[15] Nieplocha J, Harrison RJ, Littlefield RJ. Global arrays: a portable shared-memory programming model for distributed memory computers. Proceedings of the ACM/IEEE conference on supercomputing. IEEE Computer Society Press; 1994. p. 340–9.
[16] Yoo AB, Jette MA, Grondona M. Slurm: simple linux utility for resource management. Proceedings of the workshop on job scheduling strategies for parallel processing. Springer; 2003. p. 44–60.
[17] Henderson RL. Job scheduling under the portable batch system. Proceedings of the workshop on job scheduling strategies for parallel processing. Springer; 1995. p. 279–94.
[18] Liu J, Wu J, Panda DK. High performance RDMA-based MPI implementation over infiniband. Int J Parallel Programm 2004;32(3):167–98.
[19] Chen J, Revels J. Robust benchmarking in noisy environments. 2016. arXiv:https://arxiv.org/abs/1608.04295.
[20] Harada T. A framework to transform in-core GPU algorithms to out-of-core algorithms. Proceedings of the 20th ACM SIGGRAPH symposium on interactive 3D graphics and games. ACM; 2016. p. 179–80.
[21] Landaverde R, Zhang T, Coskun AK, Herbordt M. An investigation of unified memory access performance in CUDA. Proceedings of the high performance extreme computing conference (HPEC). IEEE; 2014. p. 1–6.
[22] Sakharnykh N. Beyond GPU memory limits with unified memory on pascal. 2016. https://devblogs.nvidia.com/beyond-gpu-memory-limits-unified-memory-pascal/.
[23] Yalamanchili P, Arshad U, Mohammed Z, Garigipati P, Entschev P, Kloppenborg B, et al. ArrayFire - A high performance software library for parallel computing with an easy-to-use API. 2015. https://github.com/arrayfire/arrayfire.
[24] Malakhov A. Composable multi-threading for Python libraries. Proceedings of the Python in science conferences. 2016.
[25] Lam SK, Pitrou A, Seibert S. Numba: a LLVM-based Python JIT compiler. Proceedings of the second workshop on the LLVM compiler infrastructure in HPC. 2015. p. 7:1–6.
[26] Ragan-Kelley J, Adams A, Paris S, Levoy M, Amarasinghe S, Durand F. Decoupling algorithms from schedules for easy optimization of image processing pipelines. ACM Trans Graph 2012;31(4):32:1–32:12.
[27] Li T-M, Gharbi M, Adams A, Durand F, Ragan-Kelley J. Differentiable programming for image processing and deep learning in Halide. Proc ACM Trans Graph (SIGGRAPH) 2018;37(4):139:1–139:13.
[28] Denniston T, Kamil S, Amarasinghe S. Distributed halide. SIGPLAN Not 2016;51(8):5:1–5:12.
[29] Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, et al. Tensorflow: a system for large-scale machine learning. Proceedings of the 12th USENIX conference on operating systems design and implementation. USENIX Association; 2016. p. 265–83.
[30] Paszke A, Gross S, Chintala S, Chanan G. Pytorch, NeuroIPS 2017 Workshop. 2017. https://openreview.net/pdf?id=BJJsrmfCZ.
[31] Innes M, Karpinski S, Shah V, Barber D, Stenetorp P, Besard T, et al. On machine learning and programming languages. Proceedings of the SysML conference. ACM; 2018https://www.sysml.cc/doc/37.pdf.
[32] Chamberlain BL, Callahan D, Zima HP. Parallel programmability and the Chapel language. Int J High Perform Comput Appl 2007;21(3):291–312.
[33] Charles P, Grothoff C, Saraswat V, Donawa C, Kielstra A, Ebcioglu K, et al. X10: an object-oriented approach to non-uniform cluster computing. Proceedings of the ACM SIGPLAN Notices. 40. ACM; 2005. p. 519–38.
[34] Allen E, Chase D, Hallett J, Luchangco V, Maessen J-W, Ryu S, et al. The Fortress language specification. 2005. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.85.461&rep=rep1&type=pdf.
[35] Merrill D, NVIDIA-Labs. CUDA UnBound (CUB) Library. 2015. https://nvlabs.github.io/cub/.
[36] Bell N, Hoberock J. Thrust: a productivity-oriented library for CUDA. GPU computing gems Jade edition. Elsevier; 2011. p. 359–71.
[37] Dagum L, Menon R. OpenMP: an industry-standard API for shared-memory programming. IEEE Comput Sci Eng 1998:46–55.
[38] Wienke S, Springer P, Terboven C, an Mey D. OpenACC: first experiences with real-world applications. Proceedings of the 18th international conference on parallel processing. Springer-Verlag; 2012. p. 859–70.
[39] MPI Forum. MPI: a message-passing interface standard. Technical Report. 1994.
[40] Bauer M, Treichler S, Slaughter E, Aiken A. Legion: expressing locality and independence with logical regions. Proceedings of the international conference for high performance computing, networking, storage and analysis. IEEE Computer Society; 2012. p. 1–11.
[41] Bachan J, Bonachea D, Hargrove PH, Hofmeyr S, Jacquelin M, Kamil A, et al. The UPC++ PGAS library for exascale computing. Proceedings of the second annual PGAS applications workshop. ACM; 2017. p. 7:1–4.
[42] Heroux MA, Bartlett RA, Howle VE, Hoekstra RJ, Hu JJ, Kolda TG, et al. An overview of the trilinos project. ACM Trans Math Softw 2005:397–423.
[43] Balay S, Gropp WD, McInnes LC, Smith BF. Efficient management of parallelism in object-oriented numerical software libraries. Modern software tools for scientific computing. Springer; 1997. p. 163–202.
[44] Carter Edwards H, Trott CR, Sunderland D. Kokkos: enabling manycore performance portability through polymorphic memory access patterns. J Parallel Distrib Comput 2014:3202–16.