Parallelizing Julia with a Non-invasive DSL

Todd A. Anderson[1], Hai Liu[1], Lindsey Kuper[1], Ehsan Totoni[1], Jan Vitek[2], and Tatiana Shpeisman[1]

1 **Parallel Computing Lab, Intel Labs**
2 **Northeastern University / Czech Technical University Prague**

Abstract

Computational scientists often prototype software using productivity languages that offer high-level programming abstractions. When higher performance is needed, they are obliged to rewrite their code in a lower-level efficiency language. Different solutions have been proposed to address this trade-off between productivity and efficiency. One promising approach is to create embedded domain-specific languages that sacrifice generality for productivity and performance, but practical experience with DSLs points to some road blocks preventing widespread adoption. This paper proposes a *non-invasive* domain-specific language that makes as few visible changes to the host programming model as possible. We present ParallelAccelerator, a library and compiler for high-level, high-performance scientific computing in Julia. ParallelAccelerator's programming model is aligned with existing Julia programming idioms. Our compiler exposes the implicit parallelism in high-level array-style programs and compiles them to fast, parallel native code. Programs can also run in "library-only" mode, letting users benefit from the full Julia environment and libraries. Our results show encouraging performance improvements with very few changes to source code required. In particular, few to no additional type annotations are necessary.

1998 ACM Subject Classification D.1.3 Parallel Programming

Keywords and phrases parallelism, scientific computing, domain-specific languages, Julia

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.57

1 Introduction

Computational scientists often prototype software using a *productivity language* [8, 19] for scientific computing, such as MATLAB, Python with the NumPy library, R, or most recently Julia. Productivity languages free the programmer from having to think about low-level issues such as how to manage memory or schedule parallel threads, and they typically come ready to perform common scientific computing tasks through extensive libraries. The productivity-language programmer can thus work at a level of abstraction that matches their domain expertise. However, a dilemma arises when the programmer, having produced a prototype, wants to handle larger problem sizes. The next step is to manually port the code to an *efficiency language* like C++ and parallelize it with tools like MPI. This takes considerable effort and requires a different skill set. While the result can be fast, it is harder to maintain or experiment with.

Ideally the productivity language could be automatically converted to efficient code and parallelized. Unfortunately, automatic parallelization has proved elusive, and efficient compilation of dynamic languages remains an open problem. An alternative is to embed, in the high-level language, a domain-specific language (DSL) specifically designed for high-performance scientific computing. That domain-specific language will have a restricted set of features, carefully designed to allow for efficient code generation and parallelization. Representative examples are DSLs developed using the Delite framework [6, 27, 28] for Scala,

31st European Conference on Object-Oriented Programming (ECOOP 2017).
Editor: Peter Müller; Article No. 57; pp. 57:1–57:29
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Copperhead [7] and DSLs developed using the SEJITS framework [8] for Python, and Accelerate [9] for Haskell. Brown *et al.* posit a “pick 2-out-of-3” trilemma between performance, productivity, and generality [6]. DSLs choose performance and productivity at the cost of generality by targeting a particular domain. This allows implementations to make stronger assumptions about programmer intent and employ domain-specific optimizations.

Practical experience with DSLs points to some road blocks preventing widespread adoption. DSLs have a learning curve that may put off users reluctant to invest time in learning new technologies. DSLs have functionality cliffs; the fear of hitting a limitation late in the project discourages some users. DSLs can lack in robustness; they may have long compile times, be unable to use the host’s debugger, or place limits on supported libraries and platforms.

This paper proposes a *non-invasive* domain-specific language that makes as few visible changes to the host programming model as possible. It aims to help developers parallelize scientific code with minimal alterations using a hybrid compiler and library approach. With the compiler, ParallelAccelerator provides a new parallel execution model for code that uses parallelizable constructs, but any program can also run single-threaded with the default semantics of the host language. This is also the case when the compiler encounters constructs that inhibit parallelization. In library mode, all features of the host language are available. The initial learning curve is thus small; users can start writing programs in our DSL by adding a single annotation. During development, the library mode allows users to sidestep any compilation overheads, and to retain access to all features of the host language including its debugger and libraries. The contribution of this paper is a design that leverages existing technologies and years of research in the high-performance computing community to create a system that works surprisingly well. The paper also explores the combination of features needed from a dynamic language for this to work.

ParallelAccelerator is embedded in the Julia programming language. Julia is challenging to parallelize: its code is untyped, all operators are dynamically bound, and `eval` allows loading new code at any time. While some features cannot be parallelized, their presence in other parts of the program does not prevent us from generating efficient code. Julia is also interesting because it is among the fastest dynamic languages of the day. Julia’s LLVM-based just-in-time compiler generates efficient code, giving the language competitive baseline performance. Languages like MATLAB or Python have much more “fat” that can be trimmed.

Julia provides high-level constructs in its standard library for scientific computing as well as bindings to high-performance native libraries (*e.g.*, BLAS libraries). While many library calls can run in parallel, the real issue is that library calls do not compose in parallel. Thus porting to an efficiency language is often about making the parallelism explicit and manually composing constructs to achieve greater performance. ParallelAccelerator focuses on programs written in array style, a style of programming that is widely used in scientific languages such as MATLAB and R; it identifies the implicit parallelism in array operations and automatically composes them. The standard Julia compiler does not optimize array-style code.

ParallelAccelerator provides the `@acc` annotation, a short-hand for “accelerate”, which instructs the compiler to focus on the annotated block or function and attempts to parallelize its execution. Plain host-language functions and `@acc`-annotated code can be intermingled and invoke each other freely. ParallelAccelerator parallelizes array-style constructs already existing in Julia, such as element-wise array operations, reductions on arrays, and array comprehensions, and introduces only a single new construct, `runStencil` for stencil computations.

While adding annotations is easy, we do rely on users to write code in array style.

ParallelAccelerator is implemented in Julia and is released as a package.[1] There is a small array runtime component written in C. The compiler uses a combination of type specialization and devirtualization of the Julia code and generates monomorphic OpenMP C++ code for every `@acc`-annotated function and its transitive closure. Two features of Julia made our implementation possible. The first is Julia's macro system that allows us to intercept calls to an accelerated function for each unique type signature. The second is access to Julia's type inference results. Accurate type information allows us to generate code that uses efficient unboxed representations and calls to native operations on primitive data types. Our results demonstrate that ParallelAccelerator can provide orders-of-magnitude speedup over Julia on a variety of scientific computing workloads, and can achieve performance close to optimized parallel C++.

2 Background

2.1 Julia

The Julia programming language [3] is a high-level dynamic language that targets scientific computing. Like other productivity languages, Julia has a read-eval-print loop for continuous and immediate user interaction with the program being developed. Type annotations can be omitted on function arguments and variables, in which case they behave as dynamically typed variables [4]. Julia has a full complement of meta-programming features, including macros that operate on abstract syntax trees and `eval` which allows users to construct code as text strings and evaluate it in the environment of the current module.

Julia comes with an extensive base library, largely written in Julia itself, for everyday programming tasks as well as a range of functions appropriate for scientific and numerical computing. In particular, the notation for array and vector computation is close to that of MATLAB, which suggests an easy transition to Julia for MATLAB programmers. For example, the following are a selected few array operators and functions:

- Unary: `- ! log exp sin cos`
- Binary: `.+ .- .* ./ .== .!= .> .< .>= .<=`

Julia supports multiple dispatch; that is to say, functions can be (optionally) annotated with types and Julia allows functions to be overloaded based on the types of their arguments. Hence the same `-` (negation) operator that usually takes scalar operands can be overloaded to take an array object as its argument, and returns the negation of each of its elements in a new array. For instance, `-[1,2,3]` evaluates to `[-1,-2,-3]`, and `[1,2,3] .* [3,2,1]` evaluates to `[3,4,3]` where `.*` stands for element-wise multiplication. The resolution of any function call is typically dynamic: at each call, the runtime system will check the tags of arguments and find the function definition that is the most applicable for these types.

The Julia execution engine is an aggressively specializing just-in-time compiler that emits intermediate representation for the LLVM compiler. Julia has a fast C function call API, a garbage collector and, since recently, native threads. The compiler does not optimize array-style code, so users tend to write (error-prone) explicit loops.

There are a number of design choices in Julia that facilitate the job of ParallelAccelerator. Optional type annotations are useful, in particular on data type declarations. In Python,

[1] Source code is available on GitHub: `https://github.com/intellabs/ParallelAccelerator.jl`.

programmers have no way to limit the range of values the fields of a class can take. In R or MATLAB, things are even worse, as there are not even classes; all data types are built up dynamically out of basic building blocks such as lists and arrays. In Julia, if a field is declared to be of some type `T`, then the runtime system will insert checks at every assignment (unless the compiler can prove they are redundant). Julia differentiates between abstract types, which cannot be instantiated but can have subtypes, and concrete types, which can be instantiated but cannot have subtypes. In Java terminology, concrete types are `final`. This property is helpful because the memory layout of a concrete type is thus known, and the compiler can optimize them (*e.g.*, by stack allocation or field stripping). The `eval` function is not allowed to execute in the scope of the current function, as it does in JavaScript, which means that the damage that it can do is limited to changing global variables (and if the variables are typed, those changes must be type-preserving) and defining new functions. Julia's reflection capacities are limited, so it is not possible to modify the shape of data structures or add local variables to existing frames as in JavaScript or R.

2.2 Related Work

One way to improve the performance of high-level languages is to reduce interpreter overhead, as some of these languages are still executed by interpreting abstract syntax trees or bytecode. For instance, there is work on compiling MATLAB to machine code [10, 5, 22, 16], but due to the untyped and dynamic nature of the language, a sophisticated just-in-time compiler performing complex type inference is needed to get any performance improvements. Several projects have explored how to speed up the R language. Riposte [29] uses tracing techniques to extract commonly taken operation sequences and efficiently schedule vector operations. Early versions of FastR [14] exclusively relied on runtime specialization to remove high-level overheads; more recent versions also generate native code [25]. Pydron [21] provides semi-automatic parallelization of Python programs but requires explicit programmer annotations of side-effect free, parallelizable functions. Numba [17] is a JIT compiler for Python, and allows the user to define NumPy math kernels called UFuncs in Python and run them in parallel on either CPU or GPU. Julia is simpler to optimize, because it intentionally omits some of the most dynamic features of other productivity languages for scientific computing. For instance, Julia does not allow a function to delete arbitrary local variables of its caller (which R allows). ParallelAccelerator takes advantage of the existing Julia compiler. That compiler performs one and only one major optimization: it aggressively specializes functions on the run-time type of their arguments. This is how Julia obtains similar benefits to FastR but with a simpler runtime infrastructure. ParallelAccelerator only needs a little help to generate efficient parallel code. The main difference between ParallelAccelerator and Riposte is the use of static analysis rather than dynamic liveness information. The difference between ParallelAccelerator and Pydron is the reduction in the number of programmer-provided annotations.

Another way to improve performance is to trade generality for efficiency with domain-specific languages (DSLs). Delite [6, 27, 28] is a Scala compiler framework and runtime for high-performance embedded DSLs that leverages Lightweight Modular Staging [24] for runtime code generation. Our compiler design is inspired by Delite's Domain IR and Parallel IR, but does not prevent users from using the host language (by contrast, Delite-based DSLs such as OptiML support only a subset of Scala [26]). Copperhead [7] provides composable primitives for data-parallel operations embedded in a subset of Python, and leverages implicit data parallelism for efficiency. DSLs such as Patus [11, 12] target stencil computations. PolyMage [20] and Halide [23] are highly optimized DSL implementations for image processing pipelines. ParallelAccelerator addresses the lack of generality of these DSLs by providing full access to the host language. The SEJITS methodology [8] similarly allows full access to the host language, and employs specializers (micro-compilers) for specific computational "motifs" [2], which are "stovepipes" [15] from kernels to execution platform. Individual specializers use domain-specific optimizations to efficiently implement specific kernels, but do not share a common intermediate representation or runtime like ParallelAccelerator, limiting composability.

Pochoir [30] is a DSL for stencil computations, embedded in C++ using template metaprogramming. Like ParallelAccelerator, it can be used in two modes. In the first mode, which is analogous to ParallelAccelerator's library-only mode, the programmer can compile Pochoir programs to unoptimized code using an ordinary C++ compiler. In the second mode, the programmer compiles the code using the Pochoir compiler, which acts as a preprocessor to the C++ compiler and transforms the code into parallel C++.

Improving the speed of array-style code in Julia is the goal of packages such as Devectorize.jl [18]. It provides a macro for automatically translating array-style code into devectorized code. Like Devectorize.jl, the focus of ParallelAccelerator is on speeding up code written in array style. However, since our approach is compiler-based rather than library-based, we can do much more in terms of compiler optimizations, and the addition of parallelism provides a substantial further speedup.

3 Motivating Examples

We illustrate how ParallelAccelerator speeds up scientific computing codes by example. Consider a trivial `@acc`-annotated function declared and run in the Julia REPL as shown in Figure 1.

```
julia> @acc f(x) = x .+ x .* x
f (generic function with 1 method)

julia> f([1,2,3])
5-element Array{Int64,1}:
2
6
12
```

Figure 1 A "hello world" motivating example for ParallelAccelerator.

When compiling an `@acc`-annotated function such as `f`, ParallelAccelerator optimizes high-level array operations, such as pointwise array addition (`.+`) and multiplication (`.*`). It compiles them to C++ with OpenMP directives, so that a C++ compiler can then generate high-performance native code. The C++ code that ParallelAccelerator produces for `f` is shown in Figure 2. In line 1, the function name `f` is mangled to produce a unique C function name and the input array `x` can be seen followed by the pointer argument `ret` that is used to return the output array. In line 7, the length of the array `x` is saved and used as the upper bound of the for loop in line 12. In line 8, memory is allocated for the output array, similarly matching the length of `x`. The parallel OpenMP for loop defined on lines 9-22 iterates through each element of `x`, multiplies each element by itself, adds each element to that product, and stores the result in the corresponding index in the output array. On line 21, the output array is returned from the function by storing it in `ret`.

The performance improvements that ParallelAccelerator delivers over standard Julia are

```
void f271(j2c_array<double> &x, j2c_array<double> *ret)
{
    int64_t idx, len;
    double tmp1, tmp2, ssa0, ssa1;
    j2c_array< double >  new_arr;

    len = x.ARRAYSIZE(1);
    new_arr = j2c_array<double>::new_j2c_array_1d(NULL, len);
#pragma omp parallel private(tmp1, ssa1, ssa0, tmp2)
    {
#pragma omp for private(idx)
        for (idx = 1; idx<=len; idx++)
        {
            tmp1 = x.ARRAYELEM(idx);
            ssa1 = tmp1 * tmp1;
            tmp2 = x.ARRAYELEM(idx);
            ssa0 = tmp2 + ssa1;
            new_arr.ARRAYELEM(idx) = ssa0;
        }
    }
    *ret = new_arr;
}
```

Figure 2 The generated C++ code for the motivating example from Figure 1.

in part a result of exposing parallelism and exploiting parallel hardware, and in part a result of eliminating run-time inefficiencies such as unneeded array bounds checks and intermediate array allocations. In fact, ParallelAccelerator often provides a substantial performance improvement over standard Julia even when running on one thread; see Section 6 for details.

3.1 Black-Scholes option pricing

The Black-Scholes formula for option pricing is a classic high-performance computing benchmark. Figure 3 shows an implementation of the Black-Scholes formula, written in a high-level array style in Julia. The arguments to the `blackscholes` function, `sptprice`, `strike`, `rate`, `volatility`, and `time`, are all arrays of floating-point numbers. `blackscholes` performs several computations involving pointwise addition (`.+`), subtraction (`.-`), multiplication (`.*`), and division (`./`) on these arrays. To understand this example, it is not necessary to understand the details of the Black-Scholes formula; the important thing to notice about the code is that it does many pointwise array arithmetic operations. When run on arrays of 100 million elements, this code takes 22 seconds to run under standard Julia.

The many pointwise array operations in this code make it a good candidate for speeding up with ParallelAccelerator. Doing so requires only minor changes to the code: we need only import the ParallelAccelerator library, then annotate the `blackscholes` function with `@acc`. With the addition of `@acc`, the running time drops to 13.1s on one thread, and when we enable 36-thread parallelism, running time drops to 0.5s, for a total speedup of 41.6× over standard Julia.

3.2 Gaussian blur

In this example, we consider a stencil computation that blurs an image using a Gaussian blur. Stencil computations, which update the elements of an array according to a fixed pattern called a stencil, are common in scientific computing. In this case, we are blurring an image, represented as a 2D array of pixels, by setting the value of each output pixel to a weighted

```
function blackscholes(sptprice, strike, rate, volatility, time)
    logterm = log10(sptprice ./ strike)
    powterm = .5 .* volatility .* volatility
    den = volatility .* sqrt(time)
    d1 = (((rate .+ powterm) .* time) .+ logterm) ./ den
    d2 = d1 .- den
    NofXd1 = 0.5 .+ 0.5 .* erf(0.707106781 .* d1)
    NofXd2 = 0.5 .+ 0.5 .* erf(0.707106781 .* d2)
    futureValue = strike .* exp(- rate .* time)
    c1 = futureValue .* NofXd2
    call = sptprice .* NofXd1 .- c1
    put  = call .- futureValue .+ sptprice
end
```

Figure 3 An implementation of the Black-Scholes option pricing algorithm. Adding an `@acc` annotation to this function improves performance by 41.6× on 36 cores.

```
function blur(img, iterations)
    w, h = size(img)
    for i = 1:iterations
      img[3:w-2,3:h-2] =
       img[3-2:w-4,3-2:h-4] * 0.0030 + img[3-1:w-3,3-2:h-4] * 0.0133 + ... +
       img[3-2:w-4,3-1:h-3] * 0.0133 + img[3-1:w-3,3-1:h-3] * 0.0596 + ... +
       img[3-2:w-4,3+0:h-2] * 0.0219 + img[3-1:w-3,3+0:h-2] * 0.0983 + ... +
       img[3-2:w-4,3+1:h-1] * 0.0133 + img[3-1:w-3,3+1:h-1] * 0.0596 + ... +
       img[3-2:w-4,3+2:h-0] * 0.0030 + img[3-1:w-3,3+2:h-0] + 0.0133 + ...
    end
    return img
end
```

Figure 4 An implementation of a Gaussian blur in Julia. The `...`s elide parts of the weighted average.

average of the values of the corresponding input pixel and its neighbors. (At the borders of the image, we do not have enough neighboring pixels to compute an output pixel value, so we skip those pixels and do not assign to them.) Figure 4 gives a Julia implementation of a Gaussian blur that blurs an image (`img`) a given number of times (`iterations`). The `blur` function is in array style and does not explicitly loop over all pixels in the image; instead, the loop counter refers to the number of times the blur should be iteratively applied. When run for 100 iterations on a large grayscale input image of 7095x5322 pixels, this code takes 877s to run under standard Julia.

Using ParallelAccelerator, we can rewrite the `blur` function as shown in Figure 5. In addition to the `@acc` annotation, we replace the `for` loop in the original code with ParallelAccelerator's `runStencil` construct.[2] The `runStencil` construct uses *relative* rather than absolute indexing into the input and output arrays. The `:oob_skip` argument tells `runStencil` to skip pixels at the borders of the image; unlike in the original code, we do not have to adjust indices to do this manually. Section 4.5 covers the semantics of `runStencil` in detail. With these changes, running under the ParallelAccelerator compiler, running time drops to 38s on one core and only 1.4s when run on 36 cores — a speedup of over 600× over standard Julia.

[2] Here, `runStencil` is being called with Julia's built-in `do`-block syntax (`http://docs.julialang.org/en/release-0.5/manual/functions/#do-block-syntax-for-function-arguments`).

```
@acc function blur(img, iterations)
    buf = Array(Float32, size(img)...)
    runStencil(buf, img, iterations, :oob_skip) do b, a
       b[0,0] =
            (a[-2,-2] * 0.0030 + a[-1,-2] * 0.0133 + ... +
             a[-2,-1] * 0.0133 + a[-1,-1] * 0.0596 + ... +
             a[-2, 0] * 0.0219 + a[-1, 0] * 0.0983 + ... +
             a[-2, 1] * 0.0133 + a[-1, 1] * 0.0596 + ... +
             a[-2, 2] * 0.0030 + a[-1, 2] * 0.0133 + ...
       return a, b
    end
    return img
end
```

Figure 5 Gaussian blur implemented using ParallelAccelerator.

3.3 Two-dimensional wave equation simulation

The previous two examples focused on orders-of-magnitude performance improvements enabled by ParallelAccelerator. For this example, we focus instead on the flexibility that ParallelAccelerator provides through its approach of extending Julia, rather than offering an invasive DSL alternative to programming in standard Julia.

This example uses the two-dimensional wave equation to simulate the interaction of two waves. The two-dimensional wave equation is a partial differential equation (PDE) that describes the propagation of waves across a surface, such as the vibrations of a drum head. From the discretized PDE, one can derive a formula for the future (f) position of a point (x, y) on the surface, based on its current (c) and past (p) positions (r is a constant):

$$f(x,y) = 2c(x,y) - p(x,y) + r^2\left[c(x-\Delta x,y) + c(x+\Delta x,y) + c(x,y-\Delta y) + c(x,y+\Delta y) - 4c(x,y)\right]$$

The above formula expressed in array-style code is:

```
f[2:s-1,2:s-1] = 2*c[2:s-1,2:s-1] - p[2:s-1,2:s-1]
                 + r^2 * (  c[1:s-2,2:s-1] + c[3:s,2:s-1]
                          + c[2:s-1,1:s-2] + c[2:s-1,3:s]
                          - 4*c[2:s-1,2:s-1]  )
```

This code is excerpted from a MATLAB program found “in the wild”[3] that simulates the propagation of waves on a surface. It is written in an idiomatic MATLAB style, making heavy use of array notation. It represents f, c, and p as three 2D arrays of size `s` in each dimension, and updates the `f` array based on the contents of the `c` and `p` arrays (skipping the boundaries of the grid, which are handled separately).

A direct port (preserving the array style of the MATLAB code) of the full program to Julia has a running time of 4s on a 512×512 grid. The wave equation shown above is part of the main loop of the simulation. Most of the simulation's execution time is spent in computing the above wave equation. To speed up this code with ParallelAccelerator, the key is to observe that the wave equation is performing a stencil computation. The expression

```
runStencil(p, c, f, 1, :oob_skip) do p, c, f
    f[0,0] = 2*c[0,0] - p[0,0]
             + r*r * (c[-1,0] + c[1,0] + c[0,-1] + c[0,1] - 4*c[0,0])
end
```

[3] See footnote 8 for a link to the original code.

takes the place of the wave equation formula. Like the original code, it computes an updated value for `f` based on the contents of `c` and `p`. Unlike the original code, it uses relative rather than absolute indexing: assigning to `f[0,0]` updates *all* elements of `f`, based on the contents of the `c` and `p` arrays. With this minor change and with the addition of an `@acc` annotation, the code runs in 0.3s, delivering a speedup of 15× over standard Julia.

However, for this example the key point is not the speedup enabled by `runStencil` but rather the way in which `runStencil` combines seamlessly with standard Julia features. Although the wave equation dominates the running time of the main loop of the simulation, most of the code in that main loop (which we omit here) handles other important details, such as how the simulation should behave at the boundaries of the grid. Most stencil DSLs support setting the edges to zero values, which in physical terms means that a wave reaching the edge would be reflected back toward the middle of the grid. However, this simulation uses “transparent” boundaries, which simulate an infinite grid. With ParallelAccelerator, rather than needing to add special support for this sophisticated boundary handling, we can again use `:oob_skip` in our `runStencil` call, as was done in Figure 5 for the Gaussian blur example in the previous section, and instead express the boundary-handling code in standard Julia.

The sophisticated boundary handling that this simulation requires illustrates one reason why ParallelAccelerator takes the approach of extending Julia and identifying existing parallel patterns, rather than providing an invasive DSL: it is difficult for a DSL designer to anticipate all the features that the user of the DSL might need. Therefore, ParallelAccelerator does not aim to provide an invasive DSL alternative to programming in Julia. Instead, the user is free to use domain-specific constructs like `runStencil` (say, for the wave equation), but can combine them seamlessly with standard, fully general Julia code (say, for boundary handling) that operates on the same data structures.

4 Parallel patterns in ParallelAccelerator

In this section, we explain the implicit parallel patterns that the ParallelAccelerator compiler makes explicit. We also give a careful examination of their differences in expressiveness, safety guarantees and implementation trade-offs, which hopefully should give a few new insights even to readers who are already well versed with concepts like map and reduce.

4.1 Building Blocks

Array operators and functions become the building blocks for users to write scientific and numerical programs and provide a higher level of abstraction than operating on individual elements of arrays. There are numerous benefits to writing programs in array style:

- We can safely index array elements without bounds checking once we know the input array size.
- Many operations are amenable to implicit parallelization without changing their semantics.
- Many operations do not have side effects, or when they do, their side effects are well-specified (*e.g.*, modifying the elements of one of the input arrays).
- Many operations can be further optimized to make better use of hardware features, such as caches and SIMD (Single Instruction Multiple Data) vectorization.

In short, such array operators and functions already come with a degree of domain knowledge embedded in their semantics, and such knowledge enables parallel implementations and optimization opportunities. In ParallelAccelerator, we identify a range of parallel patterns corresponding to either existing functions from the base library or language features. We are then able to translate these patterns to more efficient implementations without sacrificing program safety or altering program semantics. A crucial enabling factor is the readily available type information in Julia's typed AST, which makes it easy to identify what domain knowledge is present (*e.g.*, dense array or sparse array), and generate safe and efficient code without runtime type checking.

4.2 Map

Many element-wise array functions are essentially what is called a *map* operation in functional programming. For a unary function, this operation maps from each element of the input array to an element of the output array, which is freshly allocated and of the same size as the input array. For a binary function, this operation maps from each pair of elements from the input arrays to an element of the output array, again requiring that the two input arrays and the output array are of the same size. Such arity extension can also be applied to the output, so instead of just one output, a *map* can produce two output arrays. Internally, ParallelAccelerator translates element-wise array operators and functions to the following construct that we call *multi-map*, or *mmap* for short:

$$\underbrace{(B_1, B_2, \ldots)}_{n} = mmap(\underbrace{(x_1, x_2, \ldots)}_{m} \rightarrow \underbrace{(e_1, e_2, \ldots)}_{n}, \underbrace{A_1, A_2, \ldots}_{m})$$

Here, *mmap* takes an anonymous lambda function and maps it over m input arrays $A_1, A_2, \ldots$ to produce n output arrays $B_1, B_2, \ldots$. The lambda function performs element-wise computation from m scalar inputs to n scalar outputs. The sequential operational semantics of *mmap* is to iterate over the array length using an index, call the lambda function with elements read from the input arrays at this index, and write the results to the output arrays at the same index.

In addition to *mmap*, we introduce an *in-place multi-map* construct, or *mmap*! for short:[4]

$$mmap!(\underbrace{(x_1, x_2, \ldots)}_{m} \rightarrow \underbrace{(e_1, e_2, \ldots)}_{n}, \underbrace{A_1, A_2, \ldots}_{m}), \text{where } m \geq n$$

The difference between *mmap*! and *mmap* is that *mmap*! modifies the first n out of its m input arrays, hence we require that $m \geq n$. Below are some examples of how we translate user-facing array operations to either *mmap* or *mmap*!:

$$\begin{array}{lcl}
\texttt{log(A)} & \Rightarrow & mmap(x \rightarrow \texttt{log}(x), \texttt{A}) \\
\texttt{A .* B} & \Rightarrow & mmap((x, y) \rightarrow x\texttt{*}y, \texttt{A}, \texttt{B}) \\
\texttt{A -= B} & \Rightarrow & mmap!((x, y) \rightarrow x\texttt{-}y, \texttt{A}, \texttt{B}) \\
\texttt{A .+ c} & \Rightarrow & mmap(x \rightarrow x\texttt{+c}, \texttt{A})
\end{array}$$

In the last example above, we are able to inline a scalar variable `c` into the lambda because type inference is able to tell that `c` is not an array.

Once we guarantee the inputs and outputs used in *mmap* and *mmap*! are of the same size, we can avoid all bounds checking when iterating through the arrays. Operational safety is further guaranteed by having *mmap* and *mmap*! only as internal constructs to our compiler, rather than exposing them to users, and translating only a selected subset of higher-level

[4] The ! symbol is part of a legal identifier, suggesting mutation.

operators and functions when they are safe to parallelize. This way, we do not risk exposing the lambda function to our users, and can rule out any unintended side effects (*e.g.*, writing to environment variables, or reading from a file) that may cause non-deterministic behavior in a parallel implementation.

4.3 Reduction

Beyond maps, ParallelAccelerator also supports reduction as a parallel construct. Internally it has the form $r = \mathit{reduce}(\oplus, \phi, A)$, where $\oplus$ stands for a binary infix reduction operator, and ϕ represents an identity value that accompanies a particular $\oplus$ for a specific element type. Mathematically, the *reduce* operation is equivalent to computing $r = \phi \oplus a_1 \oplus \cdots \oplus a_n$, where $a_1, \ldots, a_n$ represent all elements in array A of length n. Reductions can be made parallel only when the operator $\oplus$ is associative, which means we can only safely translate a few functions to *reduce*, namely:

$$\begin{array}{lcl}
\texttt{sum(A)} & \Rightarrow & \mathit{reduce}(\texttt{+}, 0, \texttt{A}) \\
\texttt{product(A)} & \Rightarrow & \mathit{reduce}(\texttt{*}, 1, \texttt{A}) \\
\texttt{any(A)} & \Rightarrow & \mathit{reduce}(\texttt{||}, \texttt{false}, \texttt{A}) \\
\texttt{all(A)} & \Rightarrow & \mathit{reduce}(\texttt{\&\&}, \texttt{true}, \texttt{A})
\end{array}$$

Unlike the multi-map case, we make a design choice here not to support multi-reduction so that we can limit ourselves to only operators rather than the more flexible lambda expression. This is mostly an implementation constraint that can be lifted as soon as our parallel backend supports custom user functions.

4.4 Cartesian Map

So far, we have focused on a selected subset of base library functions that can be safely translated to parallel constructs such as *mmap*, *mmap*!, and *reduce*. Going beyond that, we also look for ways to translate larger program fragments. One such target is a Julia language feature called *comprehension*, a functional programming concept that has grown popular among scripting languages such as Python and Ruby. In Julia, comprehensions have the syntax illustrated below:

$$A = [f(x_1, x_2, \ldots, x_n)\ \texttt{for}\ x_1\ \texttt{in}\ r_1,\ x_2\ \texttt{in}\ r_2, \ldots, x_n\ \texttt{in}\ r_n]$$

where variables $x_1, x_2, \ldots, x_n$ are iterated over either range or array objects $r_1, r_2, \ldots, r_n$, and the result is an n-dimensional dense array A, each value of which is computed by calling function f with $x_1, x_2, \ldots, x_n$. Operationally it is equivalent to creating a rank-n array of a dimension that is the Cartesian product of the range of variables $r_1, r_2, \ldots, r_n$, and then going through n-level nested loops that use f to fill in all elements. As an example, we quote from the Julia user manual[5] the following `avg` function, which takes a one-dimensional input array `x` of length n and uses an array comprehension to construct an output array of length $n - 2$, in which each element is a weighted average of the corresponding element in the original array and its two neighbors:

```
avg(x) = [ 0.25*x[i-1] + 0.5*x[i] + 0.25*x[i+1] for i in 2:length(x)-1 ]
```

[5] http://docs.julialang.org/en/release-0.5/manual/arrays/#comprehensions

This example explicitly indexes into arrays using the variable `i`, something that cannot be expressed as a vanilla map operation as discussed in Section 4.2. Most comprehensions, however, can still be parallelized provided the following conditions can be satisfied:

1. There are no side effects in the comprehension body (the function f).
2. The size of all range objects $r_1, r_2, \ldots, r_n$ can be computed ahead of time.
3. There are no inter-dependencies among the range indices $x_1, x_2, \ldots, x_n$.

The first condition is to ensure that the result is still deterministic when its computation is made parallel, and ParallelAccelerator implements a conservative code analysis to identify possible breaches to this rule and reject such code. The second and third are constraints that allow the result array to be allocated prior to the computation, and in fact Julia's existing semantics for comprehension already imposes these rules. Moreover, comprehension syntax also rules out ways to either mention or index the output array within the body. Therefore, in our implementation it is always safe to write to the output array without additional bounds checking. It must be noted, however, that since arbitrary user code can go into the body, indexing into other array variables is still allowed and must be bounds-checked.

In essence, a comprehension is still a form of our familiar map operation, where instead of input arrays it maps over the range objects that make up the Cartesian space of the output array. Internally ParallelAccelerator translates comprehension to a parallel construct that we call *cartesianmap*:

$$A = \mathit{cartesianmap}(\underbrace{(i_1, \ldots)}_{n} \rightarrow f(\underbrace{r_1[i_1], \ldots}_{n}), \; \underbrace{\mathit{len}(r_1), \ldots}_{n})$$

In the above translation, we slightly transform the input ranges to their numerical sizes, and liberally make use of array indexing notation to read the actual range values so that $x_j = r[i_j]$ for all $1 \leq i \leq \mathit{len}(r_j)$ and $1 \leq j \leq n$. This transformation makes it easier to enumerate all range objects uniformly. Furthermore, given the fact that *cartesianmap* iterates over the Cartesian space of its output array, we can decompose a *cartesianmap* construct into two steps: allocating the output array, and in-place *mmap*!-ing over it, with the lambda function extended to take indices as additional parameters. We omit the details of the transformation here.

4.5 Stencil

A stencil computation computes new values for all elements of an array based on the current values of neighbors. The example we give for comprehensions in Section 4.4 would also qualify as a stencil computation. It may seem plausible to just translate stencils as if they were parallel array comprehensions, but there is a crucial difference: stencils have both input and output arrays, and they are all of the same size. This property allows us to eliminate bounds checking not just when writing to outputs, but also when reading from inputs. Because Julia does not have a built-in language construct for us to identify stencil calls in its AST, we introduce a new user-facing language construct, `runStencil`:

$$\texttt{runStencil}(\underbrace{(A, B, \ldots)}_{m} \rightarrow f(\underbrace{A, B, \ldots}_{m}), \underbrace{A, B, \ldots}_{m}, n, s)$$

The `runStencil` function takes a function f as the stencil kernel specification, then m image buffers (dense arrays) A, B, $\ldots$, and optionally a trip-count n for an *iterative stencil loop* (ISL), and a symbol s that specifies how to handle stencil boundaries. We also require that:

- All buffers are of the same size.

```
runStencil(Apu, Apv, pu, pv, Ix, Iy, 1, :oob_src_zero) do Apu, Apv, pu, pv,
    Ix, Iy
  ix = Ix[0,0]
  iy = Iy[0,0]
  Apu[0,0] = ix * (ix*pu[0,0] + iy*pv[0,0]) + lam*(4.0f0*pu[0,0]-(pu[-1,0]+
      pu[1,0]+pu[0,-1]+pu[0,1]))
  Apv[0,0] = iy * (ix*pu[0,0] + iy*pv[0,0]) + lam*(4.0f0*pv[0,0]-(pv[-1,0]+
      pv[1,0]+pv[0,-1]+pv[0,1]))
end
```

Figure 6 Stencil code excerpt from the opt-flow workload (see Section 6) that illustrates multi-buffer operation.

- Function f has arity m.
- In f, all buffers are relatively indexed with statically known indices, *i.e.*, only integer literals or constants.
- There are no updates to environment variables or I/O in f.
- For an ISL, f may return a set of buffers, rotated in position, to indicate the sequence of buffer swapping in between two consecutive stencil loops.

For boundary handling, we have built-in support for a few common cases such as wrap-arounds, but users are free to do their own boundary handling outside of the `runStencil` call, as mentioned in Section 3.

An interesting aspect of our API is that input and output buffers need not be separated and that any buffer may be read or updated. This allows flexible stencil specifications over multiple buffers, combined with support for rotating only a subset of them in case of an ISL. Figure 6 shows an excerpt from the opt-flow workload (see Section 6) that demonstrates this use case. It has six buffers, all accessed using relative indices, and only two buffers `Apu` and `Apv` are written to. A caveat is that care must be taken when reading from and writing to the same output buffer. Although there are stencil programs that require this ability, output arrays must always be indexed at 0 (*i.e.*, the current position) in order to avoid non-determinism. We currently do not check for this situation.

Our library provides two implementations of `runStencil`: one a pure-Julia implementation, and the other the parallel implementation that our compiler provides. In the latter case, we translate the `runStencil` call to an internal construct called *stencil!* that has some additional information after a static analysis to derive kernel extents, dimensions, and so on. Since there is a Julia implementation, code that uses `runStencil` can run in Julia simply by importing the ParallelAccelerator library, even when the compiler is turned off, which can be done by setting an environment variable. Any `@acc`-annotated function, including those that use `runStencil`, will run in this library-only mode, but through the ordinary Julia compilation path.[6]

5 Implementing ParallelAccelerator

The standard Julia compiler converts programs into ASTs, then transforms them to LLVM IR, and finally generates native assembly code. The ParallelAccelerator compiler intercepts this AST, introduces new AST node types for the parallel patterns discussed in Section 4,

[6] The Julia version of `runStencil` did not require significant extra development effort. To the contrary, it was a crucial early step in implementing the native ParallelAccelerator version, because it allowed us to prototype the semantics of the feature and have a reference implementation.

performs optimizations, and finally generates OpenMP C++ code (as shown in Figure 7). Since we assume there is an underlying C++ compiler producing optimized native code, our general approach to optimization within ParallelAccelerator itself is to only perform those optimizations that the underlying compiler is not capable of doing, due to the loss of semantic information present only in our Domain AST (Section 5.1) and Parallel AST (Section 5.2) intermediate languages.

Functions annotated with the `@acc` macro are replaced by a trampoline. Calls to those functions are thus redirected to the trampoline. When it is invoked, the types of all arguments are known. The combination of function name and argument types forms a key that the trampoline looks for within a cache of accelerated functions. If the key exists, then the cache has optimized code for this call. If it does not, the trampoline uses Julia's `code_typed` function to generate a type-inferred AST specialized for the argument types. The trampoline then coordinates the passage of this AST through the rest of our compilation pipeline, installs the result in the cache, and calls the newly optimized function. This aggressive code specialization mitigates the dynamic aspects of the language.

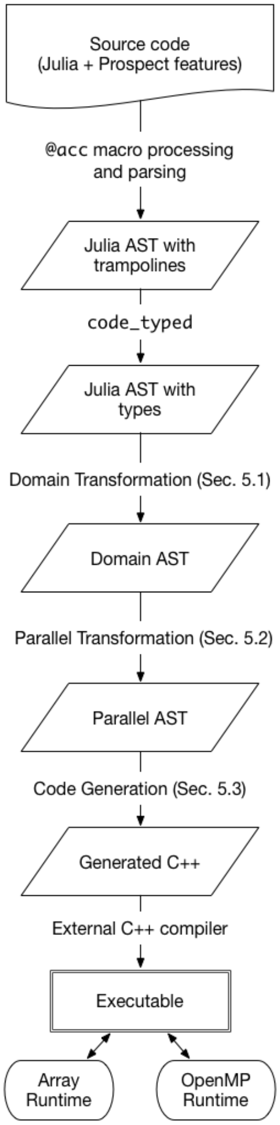
Source code
(Julia + Prospect features)
@acc macro processing and parsing
Julia AST with trampolines
code_typed
Julia AST with types
Domain Transformation (Sec. 5.1)
Domain AST
Parallel Transformation (Sec. 5.2)
Parallel AST
Code Generation (Sec. 5.3)
Generated C++
External C++ compiler
Executable
Array Runtime
OpenMP Runtime

Figure 7 The ParallelAccelerator compiler pipeline.

5.1 Domain Transformation

Domain transformation takes a Julia AST and returns a Domain AST where some nodes have been replaced by "domain nodes" such as *mmap*, *reduce*, *stencil*!, etc. We pattern-match against the incoming AST to translate supported operators and functions to their internal representations, and perform necessary safety checks and code analysis to ensure soundness with respect to the sequential semantics.

Since our compiler backend outputs C++, all transitively reachable functions must be optimized. This phase processes all call sites, finds the target function(s) and recursively optimizes them.

The viability of this strategy crucially depends on the compiler's ability to precisely determine the target functions at each call site. This, in turn, relies on knowing the type of arguments of functions. While this is generally an intractable problem for a dynamic language, ParallelAccelerator is saved by the fact that optimizations are performed at runtime, the types of the argument to the original `@acc` call are known, and the types of global constants are also known. Lastly, at the point when an `@acc` is actually invoked, all functions needed for its execution are likely to have been loaded. Users can help by providing type annotations, but in our experience these are rarely needed.

5.2 Parallel Transformation

Parallel transformation lowers domain nodes down to a common "parallel for" representation that allows a unified optimization framework for all parallel patterns. The result of this phase is a Parallel AST which extends Julia ASTs with *parfor* nodes. Each parfor node represents one or more tightly nested for loops where every point in the loops' iteration space is independent and is thus amenable to parallelization.

This phase starts with standard compiler techniques to simplify and reorder code so as to maximize later fusion. Next, the AST is lowered by replacing domain nodes with equivalent parfor nodes. Lastly, this phase performs fusion.

A parfor node consists of *pre-statements*, *loop nests, reductions, body,* and *post-statements.* Pre- and post-statements are statements executed once before and after the loop. Pre-statements do things such as output array allocation, storing the length of the arrays used by the loop or the initial value of the reduction variable. The loop nests are encoded by an array where each element represents one of the nested loops of the parfor. Each such element contains the index variable, the initial and final values of the index variable, and the step. All lowered domain operations have a loop nest. The reduction is an array where each element represents one reduction computation taking place in the parfor. Each reduction element contains the name of the reduction variable, the initial value of the reduction variable, and the function by which multiple reduction values may be combined. The body consists of the statements that perform the element-wise computation of the parfor. The body is generated in three parts, input, computation, and output, which makes it easier to perform fusion. In the input, individual elements from input arrays are stored into variables. Conversely, in the output, variables containing results are stored into their destination array. The computation is generated by the domain node which takes the variables defined for the input and output sections and generates statements that perform the computation.

Parfor fusion lowers loop iteration overhead, eliminates some intermediate arrays that would otherwise have to be created and typically has cache benefits by allowing array elements to remain in registers or cache across multiple uses. When two consecutive domain node types are lowered to parfors, we check whether they can be fused. The criteria are:

- Loop nests must be equivalent: Since loop nests are usually based on some array's dimensions, the check for equivalence often boils down to whether the arrays used by both parfors are known to have the same shape. To make this determination, we keep track of how arrays are derived from other arrays and maintain a set of array size equivalence classes.
- The second parfor must not access any piece of data created by the first at a different point in the iteration space: This means that the second parfor does not use a reduction variable computed by the first parfor and all array accesses must only access array elements corresponding to the current point in the iteration space. This also means that we do not currently fuse parfors corresponding to *stencil*! nodes.

The fusing of two parfors involves appending the second parfor's pre-statements, body, reductions, and post-statements to those of the first parfor's. Also, since the body of the second parfor uses loop index variables specific to the second parfor's loop nest, the second parfor's loop index variables are replaced with the corresponding index variables of the first parfor. In addition, we eliminate redundant loads and stores and unneeded intermediate arrays. If an array created for the first parfor is not live at end of the second parfor, then the array is eliminated by removing its allocation statement in the first parfor and by removing all assignments to it in the fused body.

5.3 Code Generation

The ParallelAccelerator compiler produces executable code from Parallel AST through our CGen backend which outputs C++ OpenMP code. That code is compiled into a native shared library with a standard C++ compiler. ParallelAccelerator creates a proxy function that handles marshalling of arguments and invokes the shared library with Julia's `ccall` mechanism. It is this proxy function that is installed in the code cache.

CGen makes a single depth-first pass over the AST. It uses the following translation strategy for Julia types, parfor nodes, and method invocations. Unboxed scalar types, such as 64-bit integers, are translated to the semantically equivalent C++ type, *e.g.*, `int64_t`. Composite types, such as `Tuples`, become C structs. Array types are translated into reference-counted C++ objects provided by our array runtime. Parfor nodes are lowered into OpenMP loops with reduction clauses and operators where appropriate. The parfor nodes also contain metadata from the parallel transformation phase that describes the private variables for each loop nest, and these are translated into OpenMP private variables. Finally, there are three kinds of method invocations that CGen has to translate: intrinsics, foreign functions, and other Julia functions. Intrinsics are primitive operations, such as array indexing and arithmetic functions on scalars, and CGen translates these into the equivalent native functions or operators and inlines them at the call sites. Julia calls foreign functions through its `ccall` mechanism, which includes the names of the library and the function to invoke. CGen translates such calls into normal function calls to the appropriate dynamic libraries. Calls to Julia functions, whether part of the standard library or user-defined, cause CGen to add the function to a worklist. When CGen is finished translating the current method, it will translate the first function on the worklist. In this way, CGen recursively translates all reachable Julia functions in a breadth-first order.

CGen imposes certain limitations on the Julia features that ParallelAccelerator supports. There are some Julia features that could be supported in CGen to some degree with additional work, such as string processing and exceptions. More fundamentally, since CGen must declare a single C type for every variable, CGen cannot support Julia union types (including type `Any`), which occur if Julia type inference determines that a particular variable could have different types at different points within a function. Global variables are always type-inferred as `Any` and so are not supported by CGen. CGen also does not support reflection or meta-programming, such as `eval`. Whenever CGen is provided an AST containing unsupported features, CGen prints a message indicating which feature caused translation to fail and installs the original, unmodified AST for the function in the code cache so that the program will still run, albeit unoptimized.

5.3.1 Experimental JGen Backend

We are developing an alternative backend, JGen, that builds on the experimental threading infrastructure provided recently in Julia 0.5. JGen generates Julia task functions for each parfor node in the AST. The arguments to the task function are determined by the parfor's liveness information plus a *range* argument that specifies which portion of the parfor's iteration space the function should perform. The task's body is a nested for loop that iterates through the space and executes the parfor body.

JGen replaces parfor nodes with calls to Julia's threading runtime, specifying the scheduling function, the task, and arguments to the task. The updated AST is stored in the code cache. When it is called, Julia applies its regular LLVM-based compilation pipeline to generate native code. Each Julia thread calls the backend's scheduling function which uses

Workload	Description	Input size	Stencil	Comp. time
opt-flow	Horn-Schunck optical flow	5184x2912 image	✓	11s
black-scholes	Black-Scholes option pricing	100M iters		4s
gaussian-blur	Gaussian blur image processing	7095x5322 image, 100 iters	✓	2s
laplace-3d	Laplace 3D 6-point stencil	290x290x290 array, 1K iters	✓	2s
quant	Quantitative option pricing	524,288 paths, 256 steps		9s
boltzmann	2D lattice Boltzmann fluid flow	200x200 grid	✓	9s
harris	Harris corner detection	8Kx8K image	✓	4s
wave-2d	2D wave equation simulation	512x512 array	✓	6s
juliaset	Julia set computation	1Kx1K resolution, 10 iters		4s
nengo	Nengo NEF algorithm	$N_A = 1000$, $N_B = 800$		7s

Table 1 Description of workloads. The last column shows the compile time for `@acc` functions. This compile-time cost is only incurred the first time that an `@acc`-accelerated function is run during a Julia session, and is not included in the performance results in Figures 8 and 9.

the thread id to perform a static partitioning of the complete iteration space. Alternative implementations such as a dynamic load-balancing scheduler are possible. JGen supports all Julia features.

Code generated by the JGen backend is currently significantly slower (about 2×) than that generated by CGen, due to factors such as C++ compiler support for vectorization that is currently lacking in LLVM. Moreover, the Julia threading infrastructure on which JGen is based is not yet considered ready for production use.[7] Therefore all the performance results we present for ParallelAccelerator in this paper use the CGen backend. However, in the long run, JGen may become the dominant backend for ParallelAccelerator as it is more general.

6 Empirical Evaluation

Our evaluation is based on a small but hopefully representative collection of scientific workloads listed in Table 1. We ran the workloads on a server with two Intel® Xeon® E5-2699 v3 ("Haswell") processors, 128GB RAM and CentOS v6.6 Linux. Each processor has 18 physical cores (36 cores total) with base frequency of 2.3GHz. The cache sizes are 32KB for L1d, 32KB for L1i, 256KB for L2, and 25MB for the L3 cache. The Intel® C++ compiler v15.0.2 compiled the generated code with `-O3`. All results shown are the average of 3 runs (out of 5 runs, first and last runs discarded).

Our speedup results are shown in Figure 8. For each workload, we measure the performance of ParallelAccelerator running in single-threaded mode (labeled "`@acc` (1 thread)") and with multiple threads ("`@acc` (36 threads)"), compared to standard Julia running single-threaded ("Julia (1 thread)"). We used Julia 0.5.0 to run both the ParallelAccelerator and standard Julia workloads.

For all workloads except opt-flow and nengo, we also compare with a MATLAB implementation. The boltzmann, wave-2d, and juliaset workloads are based on previously existing MATLAB code found "in the wild" with minor adjustments made for measurement purposes.[8] For the other workloads, we wrote the MATLAB code. MATLAB runs used

[7] See `http://julialang.org/blog/2016/10/julia-0.5-highlights`.

[8] The original implementations are: `http://www.exolete.com/lbm` for boltzmann, `https://www.piso.at/julius/index.php/projects/programmierung/13-2d-wave-equation-in-octave` for wave-2d, and `http://www.albertostrumia.it/Fractals/FractalMatlab/Jul.html` for juliaset.

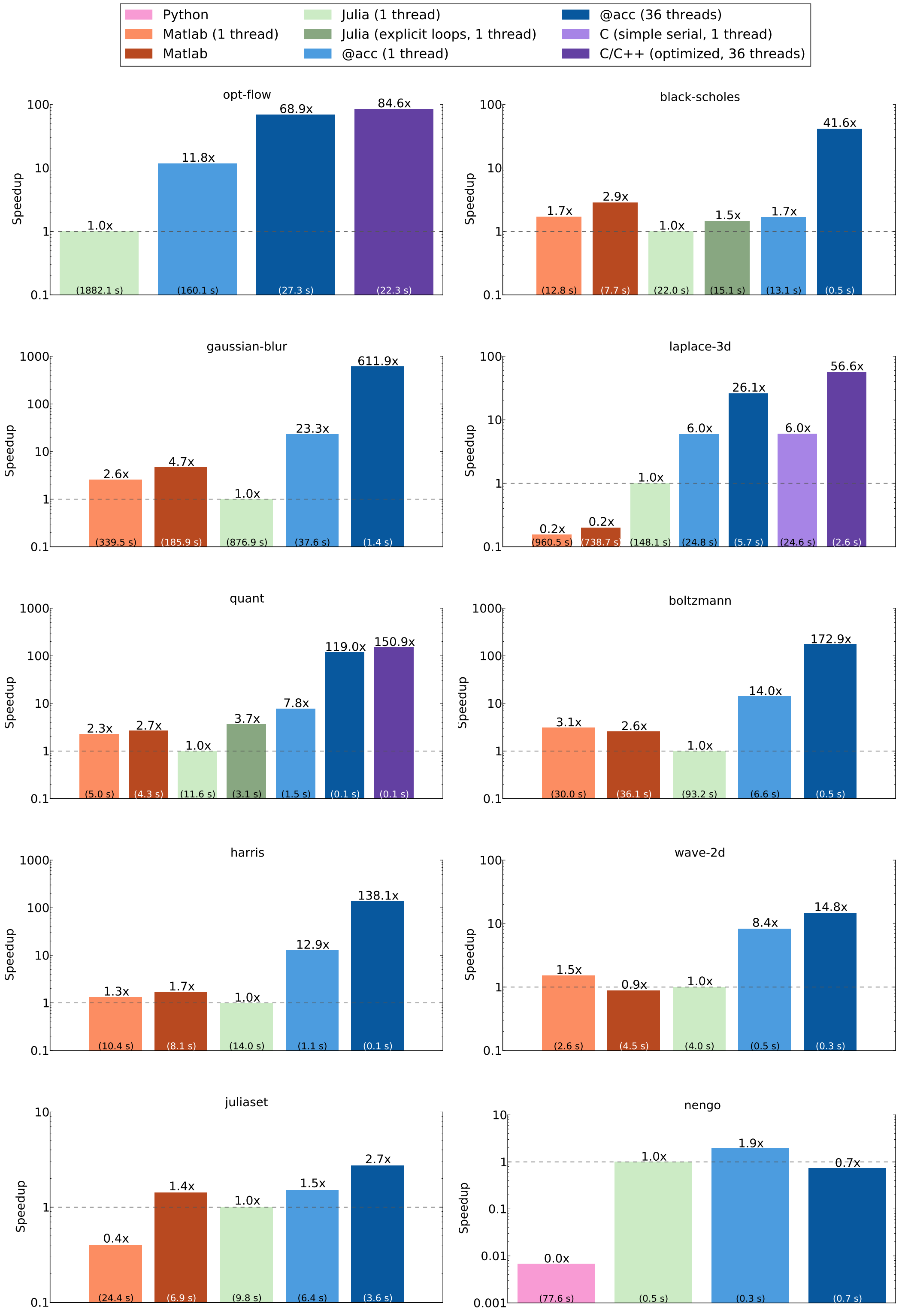

Figure 8 Speedups. Improvements relative to Julia are at the top of each bar. Absolute running times are at the bottom of each bar. Note: This figure is best viewed in color.

version R2015a, 8.5.0.197613. The label "Matlab (1 thread)" denotes runs of MATLAB with the `-singleCompThread` argument. MATLAB sometimes does not benefit from implicit parallelization of vector operations (see boltzmann or wave-2d). For the opt-flow and nengo workloads, we compare with Python implementations, run on version 2.7.10. Finally, for opt-flow, laplace-3d, and quant, expert parallel C/C++ implementations were available for comparison. In the rest of this section, we discuss each workload in detail. Julia and ParallelAccelerator code for all the workloads we discuss is available in the ParallelAccelerator GitHub repository.

6.1 Horn-Schunck optical flow estimation

This is our longest-running workload. It takes two 5184x2912 images (*e.g.*, from a sequence of video frames) as input and computes the apparent motion of objects from one image to the next using the Horn-Schunck method [13]. The implementation in standard Julia[9] ran in 1882s. Single-threaded ParallelAccelerator showed 11.8-fold improvement (160s). Running on 36 threads results in a speedup of 68.9× over standard Julia (27s). For comparison, a highly optimized parallel C++ implementation runs in 22s. That implementation is about 900 lines of C++ and uses a hand-tuned number of threads and handwritten barriers to avoid synchronization after each parallel for loop. With ParallelAccelerator, only 300 lines of code are needed (including three `runStencil` calls). We achieve performance within a factor of two of C++. This suggests it is possible, at least in some cases, to close much of the performance gap between productivity languages and expert C/C++ implementations. We do not show Python results because the Python implementation timed out. If we run on a smaller image size (534x388), then Python takes 1061s, Julia 20s, ParallelAccelerator 2s and C++ 0.2s.

The Julia implementation of this workload consists of twelve functions in two modules, and uses explicit for loops in many places. To port this code to ParallelAccelerator, we added the `@acc` annotation to ten functions using `@acc begin ... end` blocks (omitting the two functions that perform file I/O). In one `@acc`-annotated function, we replaced loops with three `runStencil` calls. For example, the `runStencil` call shown in Figure 6 replaced a 23-line, doubly nested for loop. Elsewhere, we refactored code to use array comprehensions and aggregate array operations in place of explicit loops. These changes tended to shorten the code. However, it is difficult to give a line count of the changes because in the process of porting to ParallelAccelerator, we also refactored the code to take advantage of Julia's support for multiple return values, which simplified the code considerably. We also had to make some modifications to work around ParallelAccelerator's limitations; for example, we moved a nested function to the top level because ParallelAccelerator can only operate on top-level functions within a module. The overall structure of the code remained the same.

Finally, opt-flow is the only workload we investigated in which it was necessary to add some type annotations to compile the code with ParallelAccelerator. In particular, type annotations were necessary for variables used in array comprehensions. Interestingly, though, this is the case only under Julia 0.5.0 and not under the previous version, 0.4.6, which suggests that it is not a fundamental limitation but rather an artifact of the way that Julia currently implements type inference.

[9] For this workload, we show results for Julia 0.4.6, because a performance regression in Julia 0.5.0 caused a large slowdown in the standard Julia implementation that would make the comparison unfair. All other workloads use Julia 0.5.0.

6.2 Black-Scholes option pricing model

This workload (described previously in Section 3.1) uses the Black-Scholes model to calculate the prices of European options for 100 million iterations. The Julia version runs in 22s, the single-threaded MATLAB implementation takes 12.8s, and the default MATLAB 7.7s. The gap between the single-threaded Julia and MATLAB running times bears discussion. In Julia, code written with explicit loops is often faster than code written in vectorized style with aggregate array operations.[10] This is in contrast with MATLAB, which encourages writing in vectorized style. We also measured a devectorized Julia version of this workload ("Julia (explicit loops, 1 thread)"). As Figure 8 shows, it runs in 15.1s, much closer to single-threaded MATLAB. ParallelAccelerator on one thread gives a 1.7× speedup over the array-style Julia implementation (13.1s), and with 36 threads the running time is 0.5s, a total speedup of 41.6× over Julia and 14.5× over the faster of the two MATLAB versions. Because the original code was already written in array style, this speedup was achieved non-invasively: the only modification necessary to the code was to add an `@acc` annotation.

6.3 Gaussian blur image processing

This workload (described previously in Section 3.2) uses a stencil computation to blur an image using a Gaussian blur. The Julia version runs in 877s, single-threaded MATLAB in 340s, and the default MATLAB in 186s. The ParallelAccelerator implementation uses a single `runStencil` call and takes 38s on one thread, a speedup of 23.3× over Julia. 36-thread parallelism reduces the running time to 1.4s — a total speedup of over 600× over Julia and about 130× over the faster of the two MATLAB versions. The code modification necessary to achieve this speedup was to replace the loop shown in Figure 4 with the equivalent `runStencil` call shown in Figure 5 — an 8-line change, along with adding an `@acc` annotation.

6.4 Laplace 3D 6-point stencil

This workload solves the Laplace equation on a regular 3D grid with simple Dirichlet boundary conditions. The Julia implementation we compare with is written in devectorized style, using four nested for loops. It runs in 148s, outperforming default MATLAB (961s) and single-threaded MATLAB (739s). The ParallelAccelerator implementation uses `runStencil`.

ParallelAccelerator on one thread runs in 24.8s, a speedup of 6× over standard Julia. This running time is roughly equivalent to a simple serial C implementation (24.6s). Running under 36 threads, the ParallelAccelerator implementation takes 5.7s, a speedup of 26× over Julia and 130× over the faster of the two MATLAB versions. An optimized parallel C implementation that uses SSE intrinsics runs in 2.6s. The running time of the ParallelAccelerator version is therefore nearly within a factor of two of highly optimized parallel C. The `runStencil` implementation is very high-level: the body of the function passed to `runStencil` is only one line. The C version requires four nested loops and many calls to intrinsic functions. Furthermore, the C code is less general: each dimension must be a multiple of 4, plus 2. Indeed, this was the reason we chose the problem size of 290x290x290. The ParallelAccelerator implementation, though, can handle arbitrary NxNxN input sizes.

For this example, since the original Julia code was written in devectorized style, the necessary code changes were to replace the loop nest with a single `runStencil` call and to add the `@acc` annotation. We replaced 17 lines of nested for loops with a 3-line `runStencil` call.

[10] See, *e.g.*, `https://github.com/JuliaLang/julialang.github.com/issues/353` for a discussion.

6.5 Quantitative option pricing model

This workload uses a quantitative model to calculate the prices of European and American options. An array-style Julia implementation runs in 11.6s. As with black-scholes, we also compare with a Julia version written in devectorized style, which runs 3.7× faster (3.1s). Single-threaded MATLAB and default MATLAB versions run in 5s and 4.3s, respectively. Single-threaded ParallelAccelerator runs in 1.5s. With 36 threads, the running time is 0.09s, a total speedup of 119× over array-style Julia, and 45× over the faster of the two MATLAB versions. For comparison, an optimized parallel C++ implementation written using OpenMP runs in 0.08s. ParallelAccelerator is about 1.3× slower than the parallel C++ version.

Since this workload was already written in array style, and the bulk of the computation takes place in a single function, it should have been easy to port to ParallelAccelerator by adding an `@acc` annotation. However, we encountered a problem in that the `@acc`-annotated function calls the `inv` function (for inverting a matrix) from Julia's linear algebra standard library.[11] We had difficulty compiling `inv` through CGen because Julia has an unusual implementation of linear algebra, making it hard to generate code for most of the linear algebra library functions (except for BLAS library calls, which are straightforward to translate). As a workaround, we wrote our own implementation of `inv` for the `@acc`-annotated code to call, specialized to the array size needed for this workload. With that change, ParallelAccelerator worked well. The need for workarounds like this could be avoided by using the JGen backend described in Section 5.3.1, which supports all of Julia.

6.6 2D lattice Boltzmann fluid flow model

This workload uses the 2D lattice Boltzmann method for fluid simulation. The ParallelAccelerator version uses `runStencil`. The Julia implementation runs in 93s, and single-threaded MATLAB and default MATLAB in 30s and 36s, respectively (making this an example of a workload where MATLAB's default implicit parallelization hurts rather than helps). Single-threaded ParallelAccelerator runs in 6.6s, a speedup of 14× over Julia. With 36 threads we get a further speedup to 0.5s, for a total speedup of 173× over Julia and 56× over the faster of the two MATLAB versions.

This workload is the only one we investigated in which a use of `runStencil` is longer than the code it replaces. The ParallelAccelerator version of the code contains a single 65-line `runStencil` call, replacing a 44-line while loop in the standard Julia implementation.[12] In addition to the replacement of the while loop with `runStencil`, other, smaller differences between the ParallelAccelerator and Julia implementations arose because ParallelAccelerator does not support the transfer of `BitArray`s between C and Julia. Therefore the modifications needed to run this workload with ParallelAccelerator came the closest to being invasive changes of any workload we studied. That said, the code is still recognizably "Julia" and our view is that the resulting 173× speedup justifies the effort.

6.7 Harris corner detection

This workload uses the Harris corner detection method to find corners in an input image. The ParallelAccelerator implementation uses `runStencil`. The Julia implementation runs in

[11] See `http://docs.julialang.org/en/stable/stdlib/linalg/#Base.inv`.

[12] That said, the Julia implementation was a direct port from the original MATLAB code, which was written with extreme concision in mind (see `http://exolete.com/lbm/` for a discussion), while the `runStencil` implementation was written with more of an eye toward readability.

14s; single-threaded MATLAB and default MATLAB run in 10.4s and 8.1s, respectively. The single-threaded ParallelAccelerator version runs in 1.1s, a speedup of 13× over Julia. The addition of 36-thread parallelism results in a further speedup to 0.1s, for a total speedup of 138× over Julia and 80× over the faster of the two MATLAB versions.

The Harris corner detection algorithm is painful to implement without some kind of stencil abstraction. The ParallelAccelerator implementation of this workload uses five `runStencil` calls, each with a one-line function body. The standard Julia code, in the absence of `runStencil`, has a function that computes and returns the application of a stencil to an input 2D array. This function has a similar interface to `runStencil`, but is less general (and, of course, cannot parallelize as `runStencil` does). Therefore the biggest difference between the ParallelAccelerator and Julia implementations of this workload is that we were able to remove the `runStencil` substitute function from the ParallelAccelerator version, which eliminated over 30 lines of code. The remaining differences between the versions, including addition of the `@acc` annotation, are trivial.

6.8 2D wave equation simulation

This workload is the wave equation simulation described in Section 3.3. The ParallelAccelerator implementation uses `runStencil`. The Julia implementation (4s) outperforms the default MATLAB implementation (4.5s); however, the single-threaded MATLAB implementation runs in 2.6s, making this another case where MATLAB's default implicit parallelization is unhelpful. The single-threaded ParallelAccelerator version runs in 0.5s, a speedup of 8× over Julia. The addition of 36-thread parallelism results in a further speedup to 0.3s, for a total speedup of about 15× over Julia and 10× over the faster of the two MATLAB versions.

The Julia implementation of this workload is a direct port from the MATLAB version and is written in array style, so it is amenable to speedup with ParallelAccelerator without any invasive changes. The only nontrivial modification necessary is to replace the one-line wave equation shown in Section 3.3 with a call to an `@acc`-annotated function containing a `runStencil` call with an equivalent one-line body.

6.9 Julia set computation

This workload computes the Julia set fractal[13] for a given complex constant at a resolution of 1000x1000 for ten successive iterations of a loop. The Julia implementation (9.8s) is written in array style. It outperforms the single-threaded MATLAB version (24s), but is slightly slower than the default MATLAB version (6.9s). On one thread, ParallelAccelerator runs in 6.4s, a speedup of 1.5× over standard Julia. With 36 threads we achieve a further speedup to 3.6s, for a total speedup of 2.7× over Julia and about 2× over the faster of the two MATLAB versions. The only modification needed to the standard Julia code is to add a single `@acc` annotation. The speedup enabled by ParallelAccelerator is modest because each iteration of the loop is dependent on results from the previous iteration, and so ParallelAccelerator is limited to parallelizing array-style operations within each iteration.

6.10 Nengo NEF algorithm

Finally, for our last example we consider a workload that demonstrates poor parallel scaling with ParallelAccelerator. This workload is a demonstration of the Neural Engineering

[13] See `https://en.wikipedia.org/wiki/Julia_set`.

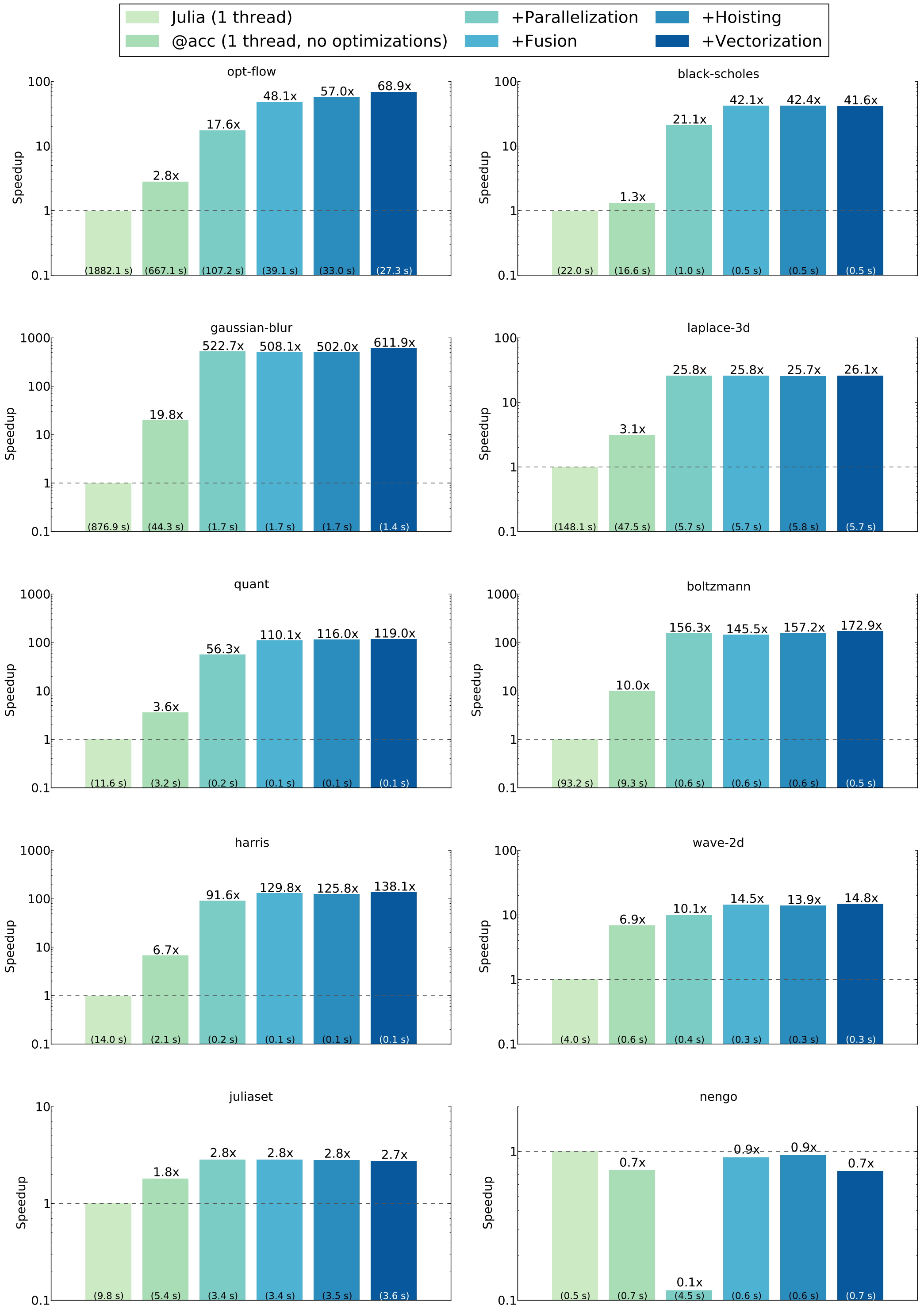

Figure 9 The effects of individual ParallelAccelerator optimizations on a variety of workloads. Speedup after each successive optimization (compared to Julia baseline) is shown at the top of each bar (higher is better). The absolute running times are shown at the bottom of each bar.

Framework (NEF) algorithm used by Nengo, a Python software package for simulating neural systems [1]. It builds a network from two populations of neurons. We ported the NEF Python code[14] to Julia and attempted to parallelize it with ParallelAccelerator. With an input size of 1000 neurons for the first population and 800 for the second population, the original Python code runs in 78s. We observed an impressive speedup to 0.5s simply by porting the code to Julia. On one thread, the ParallelAccelerator version runs in 0.3s, a 1.9× speedup over standard Julia, but on 36 threads we observed a slowdown to 0.7s.

The Julia port of the NEF algorithm is about 200 lines of code comprising several functions. For the ParallelAccelerator implementation, we annotated two of the functions with `@acc`, and we replaced roughly 30 lines of code in those functions that had been written using explicit for loops with their array-style equivalents. Doing so led to the modest speedup gained by running with ParallelAccelerator on one thread. However, this workload offers little opportunity for parallelization with ParallelAccelerator, although it might be possible to obtain better results on a different problem size or with fewer threads.

6.11 Impact of Individual Optimizations

Figure 9 shows a breakdown of the effects of parallelism and individual optimizations implemented by our compiler. The leftmost bar in each plot (labeled "Julia (1 thread)") shows standard Julia running times, for comparison. The second bar ("`@acc` (1 thread, no optimizations)") shows running time for ParallelAccelerator with `OMP_NUM_THREADS=1` and optimizations disabled. The difference between the first and second bars in each plot illustrates the impact of avoiding the run-time overhead of allocation and management of arrays for intermediate computations and checking array access bounds. As the figure shows, the difference can be substantial. This is due to the lack of optimization for array-style code in the Julia compiler. The third bar in each plot ("+Parallelization") shows the impact of enabling parallel execution with `OMP_NUM_THREADS=36`. Again, the benefits are usually noticeable. The last three bars in each plot ("+Fusion", "+Hoisting", "+Vectorization") each cumulatively add an additional compile-time optimization. For many workloads, these optimizations have no significant impact or even a slight negative impact. Some workloads, such as opt-flow and gaussian-blur, see a noticeable speedup from vectorization. For opt-flow, black-scholes, and quant there is a noticeable speedup from fusion. In general, the usefulness of these optimizations depends on the application, but the performance to be gained from them appears small in comparison to the improvement we see from parallelization and run-time overhead elimination.

7 Limitations and Future Work

While we are confident that the reported results and claimed benefits of ParallelAccelerator will generalize to other workloads and other languages, we do acknowledge the following limitations and opportunities to extend our work.

- **Workloads:** Our empirical evaluation is limited in size to the workloads we were able to obtain or write ourselves. A larger set of programs would increase confidence in the applicability and generality of the approach. We considered porting more MATLAB or R codes to Julia, as they are often naturally vectorized, but the differences in semantics

[14] Available at `http://nengo.ca/docs/html/nef_algorithm.html`.

of the base libraries complicates that task. It turned out that each program in our benchmark suite represented substantial work. Larger programs in terms of code size would also help validate the scalability of the compilation strategy. We intend to engage with the Julia community to port more Julia code currently written with loops to the array style supported by ParallelAccelerator.

- **Programming model:** ParallelAccelerator only parallelizes programs that are written in array style; it will not touch explicit loops. Thus it is less suitable for certain applications, for instance, string and graph processing applications that use pointers. Additionally, the ParallelAccelerator compiler must be able to statically resolve all operations being invoked. For this it needs to have fairly accurate approximations of the types of every value. Furthermore, some reflective operations cannot be invoked within an `@acc`-annotated function. We can generalize the ParallelAccelerator strategy to accept more programming styles, although automatic parallelization may be more challenging. As for the type specialization, it has worked surprisingly well so far. We hypothesize that the kinds of array codes we work with do not use complex types for performance reasons. They tend to involve arrays of primitives as programmers try to match what will run fast on the underlying hardware. One of the reasons why allocation and object operations are currently not supported is that ParallelAccelerator was originally envisioned as running on GPUs or other accelerators with a relatively limited programming model, but for CPUs, we could relax that restriction.
- **Code bloat:** The aggressive specialization used by ParallelAccelerator has the potential for massive code bloat. This could occur if `@acc`-annotated functions were called with many distinct sets of argument types. We have not witnessed it so far, but it could be a problem and would require a smarter specialization strategy. The aggressive specialization may lead to generating many native functions that are mostly similar. Instead of generating a new function for each new type signature, we could try to share the same implementation for signatures that behave similarly.
- **User feedback:** There is currently limited feedback when ParallelAccelerator fails to parallelize code (*e.g.*, due to union types for some variables). While the code will run, users will see a warning message and will not see the expected speedups. Unlike invasive DSLs, with additional work we can map parallelization failures back to statements in the Julia program. We are considering how to provide better diagnostic information.
- **Variability:** The benefits of parallelization depend on both the algorithm and the target parallel architecture. For simplicity, we assume a shared-memory machine without any communication cost and parallelize all implicitly parallel operations. However, this can result in poor performance. For example, parallel distribution of array elements across operations can be inconsistent, which can have expensive communication costs (*i.e.*, cache line exchange). We are considering how to expose more tuning parameters to the user.

8 Conclusion

Typical high-performance DSLs require the use of a dedicated compiler and runtime for users to use the domain-specific language features. Unfortunately, DSLs often face challenges that limit their widespread adoption, such as a steep learning curve, functionality cliffs, and a lack of robustness. Addressing these shortcomings requires significant engineering effort. Our position is that designing and implementing a DSL is difficult enough without having to tackle these additional challenges. Instead, we argue that implementors should focus only on providing high-level abstractions and a highly optimizing implementation, but users of the DSL should enjoy rapid development and debugging, using familiar tools on the platform of their choice. This is where the ability to disable the ParallelAccelerator compiler during development and then enable it again at deployment time comes in: it allows us to offer users high performance and high-level abstractions while still giving them an easy way to sidestep problems of compilation time, robustness, debuggability, and platform availability.

In conclusion, ParallelAccelerator is a non-invasive DSL because it does not require wholesale changes to the programming model. It allows programmers to write high-level, high-performance array-style code in a general-purpose productivity language by identifying implicit parallel patterns in the code and compiling them to efficient native code. It also eliminates many of the usual overheads of high-level array languages, such as intermediate array allocation and bounds checking. Our results demonstrate considerable speedups for a number of scientific workloads. Since ParallelAccelerator programs can run under standard Julia, programmers can develop and debug their code using a familiar environment and tools. ParallelAccelerator also demonstrates that with a few judicious design decisions, scientific codes written in dynamic languages can be parallelized. While it may be the case that scientific codes are somewhat more regular in their computational kernels than general-purpose codes, our experience with ParallelAccelerator was mostly positive: there were very few cases where we needed to add type annotations or where the productivity-oriented aspects of the Julia language prevented our compiler from doing its job. This is encouraging as it suggests that dynamism and performance need not be mutually exclusive.

Acknowledgments Anand Deshpande and Dhiraj Kalamkar wrote the parallel C version of laplace-3d. Thanks to our current and former colleagues at Intel and Intel Labs who contributed to the design and implementation of ParallelAccelerator and to the collection of workloads we studied: Raj Barik, Neal Glew, Chunling Hu, Victor Lee, Geoff Lowney, Paul Petersen, Hongbo Rong, Jaswanth Sreeram, Leonard Truong, and Youfeng Wu. Thanks to the Julia Computing team for their encouragement of our work and assistance with Julia internals. Prof. Vitek's research has been supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 695412).

References

1 The Nengo neural simulator. `http://nengo.ca`, 2016.

2 Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine Yelick. The landscape of parallel computing research: A view from Berkeley. Technical report, UC Berkeley, 2006. URL: `www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html`.

3 Jeff Bezanson, Stefan Karpinski, Viral Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *CoRR*, abs/1209.5145, 2012. URL: `http://arxiv.org/abs/1209.5145`.

4 Gavin Bierman, Erik Meijer, and Mads Torgersen. Adding dynamic types to C$^\sharp$. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 76–100, Berlin, Heidelberg, 2010. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=1883978.1883986`.

5 João Bispo, Luís Reis, and João M. P. Cardoso. Techniques for efficient MATLAB-to-C compilation. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Li-*

braries, Languages, and Compilers for Array Programming, ARRAY 2015, pages 7–12, New York, NY, USA, 2015. ACM. URL: http://doi.acm.org/10.1145/2774959.2774961, doi:10.1145/2774959.2774961.

6 Kevin J. Brown, Arvind K. Sujeeth, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 89–100, Washington, DC, USA, 2011. IEEE Computer Society. URL: http://dx.doi.org/10.1109/PACT.2011.15, doi:10.1109/PACT.2011.15.

7 Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: Compiling an embedded data parallel language. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 47–56, New York, NY, USA, 2011. ACM. URL: http://doi.acm.org/10.1145/1941553.1941562, doi:10.1145/1941553.1941562.

8 Bryan Catanzaro, Shoaib Kamil, Yunsup Lee, Krste Asanovic, James Demmel, Kurt Keutzer, John Shalf, Kathy Yelick, and Armando Fox. SEJITS: Getting productivity and performance with selective embedded JIT specialization. In *Workshop on Programmable Models for Emerging Architecture (PMEA)*, 2009. URL: http://parlab.eecs.berkeley.edu/publication/296.

9 Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, DAMP '11, pages 3–14, New York, NY, USA, 2011. ACM. URL: http://doi.acm.org/10.1145/1926354.1926358, doi:10.1145/1926354.1926358.

10 Maxime Chevalier-Boisvert, Laurie Hendren, and Clark Verbrugge. Optimizing MATLAB through just-in-time specialization. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 46–65, Berlin, Heidelberg, 2010. Springer-Verlag. URL: http://dx.doi.org/10.1007/978-3-642-11970-5_4, doi:10.1007/978-3-642-11970-5_4.

11 Matthias Christen, Olaf Schenk, and Helmar Burkhart. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 676–687, Washington, DC, USA, 2011. IEEE Computer Society. URL: http://dx.doi.org/10.1109/IPDPS.2011.70, doi:10.1109/IPDPS.2011.70.

12 Matthias Christen, Olaf Schenk, and Yifeng Cui. Patus for convenient high-performance stencils: Evaluation in earthquake simulations. In *Proceedings of the 2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 1–10, Washington, DC, USA, 2012. IEEE Computer Society. URL: http://dx.doi.org/10.1109/SC.2012.95, doi:10.1109/SC.2012.95.

13 Berthold K. P. Horn and Brian G. Schunck. Determining optical flow. *Artif. Intell.*, 17(1-3):185–203, August 1981. URL: http://dx.doi.org/10.1016/0004-3702(81)90024-2, doi:10.1016/0004-3702(81)90024-2.

14 Tomas Kalibera, Petr Maj, Floreal Morandat, and Jan Vitek. A fast abstract syntax tree interpreter for R. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, pages 89–102, New York, NY, USA, 2014. ACM. URL: http://doi.acm.org/10.1145/2576195.2576205, doi:10.1145/2576195.2576205.

15 Shoaib Ashraf Kamil. *Productive High Performance Parallel Programming with Auto-tuned Domain-Specific Embedded Languages.* PhD thesis, EECS Department, University of California, Berkeley, January 2013. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-1.html.

16 Vineet Kumar and Laurie Hendren. MIX10: Compiling MATLAB to X10 for high performance. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 617–636, New York, NY, USA, 2014. ACM. URL: http://doi.acm.org/10.1145/2660193.2660218, doi:10.1145/2660193.2660218.

17 Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pages 7:1–7:6, New York, NY, USA, 2015. ACM. URL: http://doi.acm.org/10.1145/2833157.2833162, doi:10.1145/2833157.2833162.

18 Dahua Lin. Devectorize.jl. https://github.com/lindahua/Devectorize.jl, 2015.

19 Derek Lockhart, Gary Zibrat, and Christopher Batten. PyMTL: A unified framework for vertically integrated computer architecture research. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 280–292, Washington, DC, USA, 2014. IEEE Computer Society. URL: http://dx.doi.org/10.1109/MICRO.2014.50, doi:10.1109/MICRO.2014.50.

20 Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. PolyMage: Automatic optimization for image processing pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 429–443, New York, NY, USA, 2015. ACM. URL: http://doi.acm.org/10.1145/2694344.2694364, doi:10.1145/2694344.2694364.

21 Stefan C. Müller, Gustavo Alonso, Adam Amara, and André Csillaghy. Pydron: Semi-automatic parallelization for multi-core and the cloud. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 645–659, Berkeley, CA, USA, 2014. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=2685048.2685100.

22 Ashwin Prasad, Jayvant Anantpur, and R. Govindarajan. Automatic compilation of MATLAB programs for synergistic execution on heterogeneous processors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 152–163, New York, NY, USA, 2011. ACM. URL: http://doi.acm.org/10.1145/1993498.1993517, doi:10.1145/1993498.1993517.

23 Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM. URL: http://doi.acm.org/10.1145/2491956.2462176, doi:10.1145/2491956.2462176.

24 Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 127–136, New York, NY, USA, 2010. ACM. URL: http://doi.acm.org/10.1145/1868294.1868314, doi:10.1145/1868294.1868314.

25 Lukas Stadler, Adam Welc, Christian Humer, and Mick Jordan. Optimizing R language execution via aggressive speculation. In *Proceedings of the 12th Symposium on Dynamic Languages*, DLS 2016, pages 84–95, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2989225.2989236, doi:10.1145/2989225.2989236.

26 Arvind Sujeeth. OptiML language specification 0.2. stanford-ppl.github.io/Delite/optiml/downloads/optiml-spec.pdf, 2012.

27 Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. OptiML: An implicitly parallel domain-specific language for machine learning. In Lise Getoor and Tobias Scheffer, editors, *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, ICML '11, pages 609–616, New York, NY, USA, June 2011. ACM.

28 Arvind K. Sujeeth, Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, Victoria Popic, Michael Wu, Aleksandar Prokopec, Vojin Jovanovic, Martin Odersky, and Kunle Olukotun. Composition and reuse with compiled domain-specific languages. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 52–78, Berlin, Heidelberg, 2013. Springer-Verlag. URL: `http://dx.doi.org/10.1007/978-3-642-39038-8_3`, `doi:10.1007/978-3-642-39038-8_3`.

29 Justin Talbot, Zachary DeVito, and Pat Hanrahan. Riposte: A trace-driven compiler and parallel VM for vector code in R. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 43–52, New York, NY, USA, 2012. ACM. URL: `http://doi.acm.org/10.1145/2370816.2370825`, `doi:10.1145/2370816.2370825`.

30 Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The Pochoir stencil compiler. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM. URL: `http://doi.acm.org/10.1145/1989493.1989508`, `doi:10.1145/1989493.1989508`.