Prospect:
A Library and Compiler for High-Level, High-Performance Scientific Computing in Julia

Lindsey Kuper
Parallel Computing Lab, Intel Labs

April 14, 2016

High Performance Scripting Project

Contributors: Todd Anderson, Raj Barik, Chunling Hu, Victor Lee, Hai Liu, Geoff Lowney, Paul Petersen, Hongbo Rong, Tatiana Shpeisman, Ehsan Totoni, Leonard Truong, Youfeng Wu

You're a scientist or engineer...

Your problem: designing a bridge, decrypting a message, picking a stock portfolio, processing audio signals, training a car to drive itself, …

Your expertise: differential equations, Fourier analysis, linear algebra, matrix computations, …

Not your expertise: memory management, scheduling parallel tasks

Productivity languages and the “human compiler” problem

Productivity languages: Matlab, Python, R, Julia, …

How do you “scale up” a productivity-language prototype? The answer today: Get an expert to port the code to an efficiency language

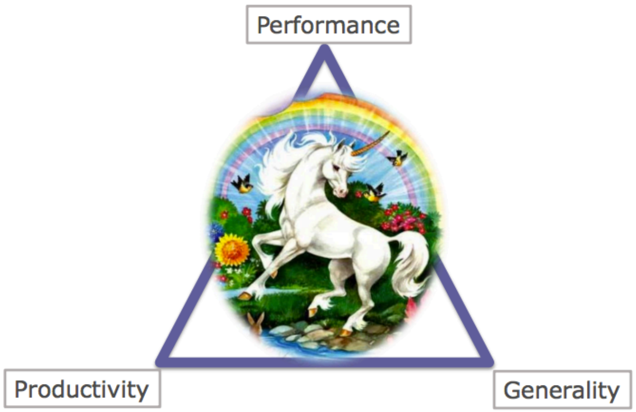
The result is fast…and also brittle, hard to experiment with, and hard to maintain

Can we do better?

How about high-performance DSLs?

Idea: trade off generality for productivity and efficiency

Delite (Brown *et al*. 2011),
SEJITS (Catanzaro *et al.* 2009),
Halide (Ragan-Kelley *et al*. 2013),
Copperhead (Catanzaro *et al*. 2011), …



[Olokutun *et al*., 2012]

Amazing results! But, two challenges:

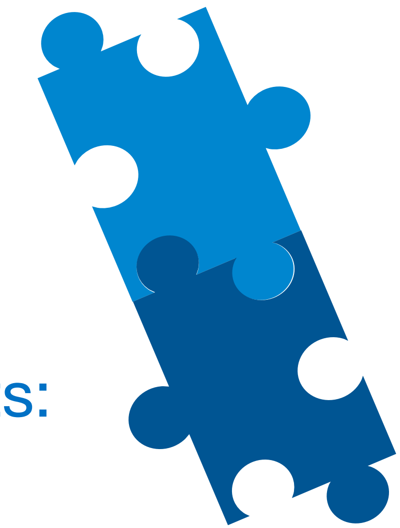
- The learning curve
- The rest of the productivity story…

The rest of the DSL productivity story

Several dimensions to productivity beyond offering the “right” abstractions for a domain:

- Fast compilation time
- Robust to a wide variety of inputs
- Debuggable using familiar techniques
- Available on the platforms users want to use

Our system: Prospect



A combination compiler-library solution

- Accelerate existing language constructs:
  - map, reduce, comprehension
- Support additional domain-specific constructs (`runStencil`)
  - …with two implementations: library-only and native

Run in library-only mode during development and debugging

Run in native mode for high performance at deployment

Prospect in practice

Implemented as a julia package:

github.com/IntelLabs/ParallelAccelerator.jl

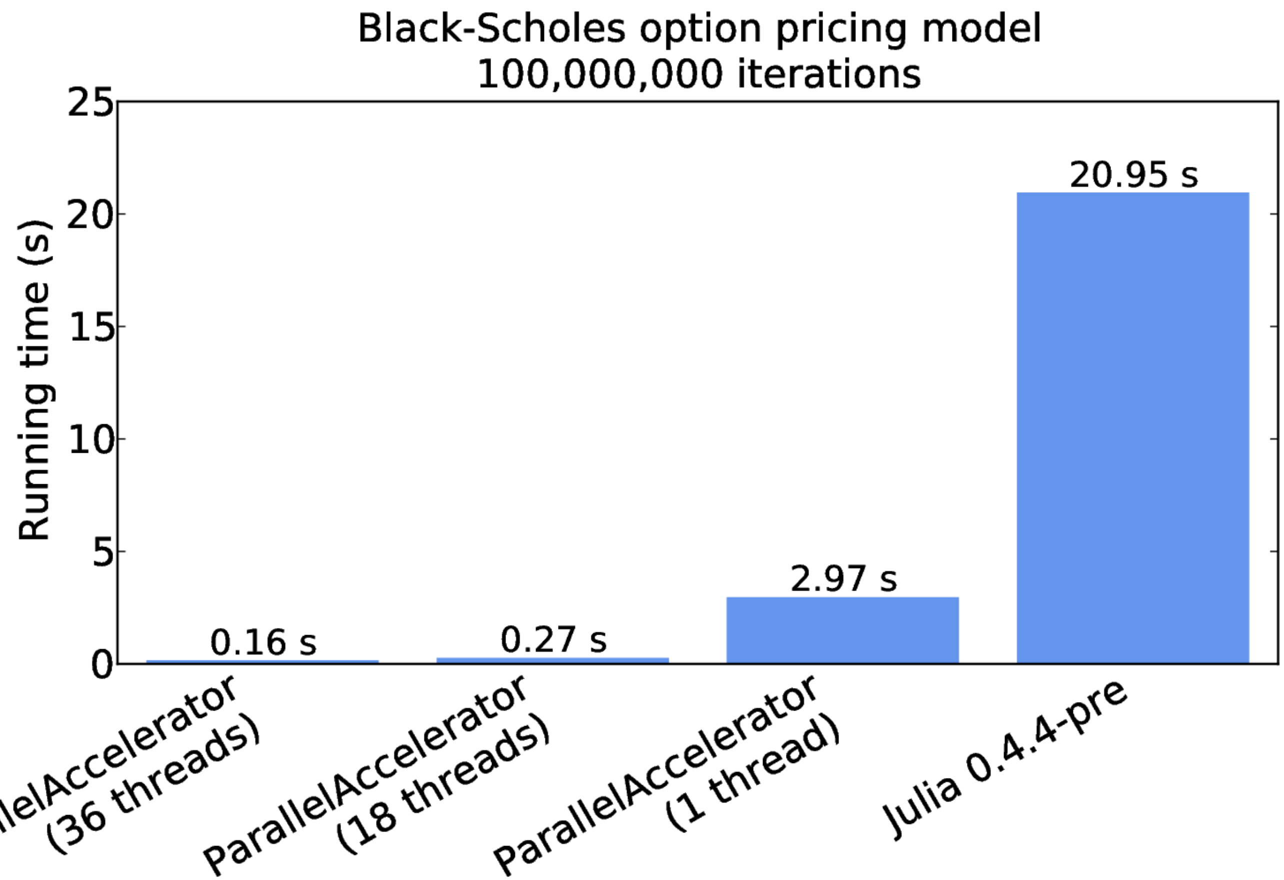
Provides an `@acc` macro to annotate code to be optimized

Under the hood, it’s a Julia-to-C++ compiler, written in Julia

Approach:

- Identify implicit parallel patterns in a subset of Julia code
- Compile to explicit parallel for loops
- Eliminate run-time overheads

A quick preview of results…



Data from 01/31/2016
2 Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.3GHz processors, 18 cores each (36 cores total)
128 GB RAM

Aside: why Julia?

- Open source
- Faster than many scientific computing languages
- Good support for array-style programming
- Under active development, strong community
- A Julia compiler in Julia works pretty well!

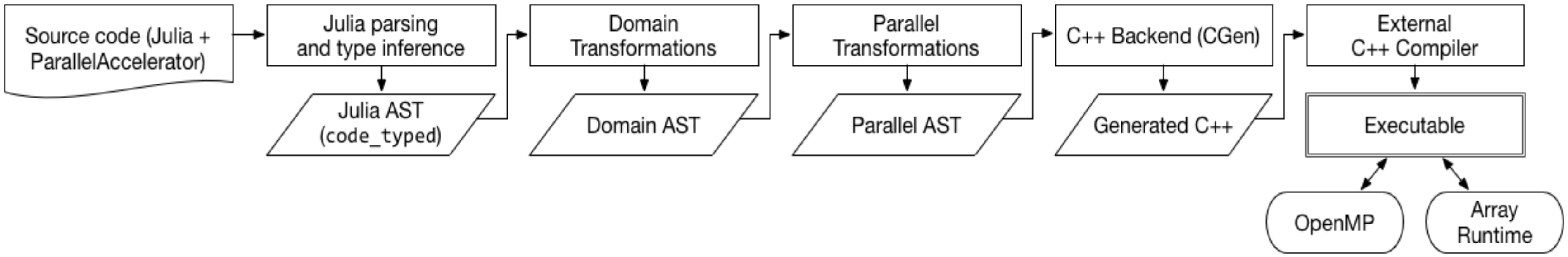
Parallel patterns

- Map: Translate pointwise array operations like `.+`, `.-`, `.*`, and `./` to data-parallel map operations
- Reduce: Translate `minimum`, `maximum`, `sum`, `prod`, `any`, and `all` to data-parallel reduce operations
- Array comprehensions: Translate to in-place map operations

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

- Special `runStencil` form for stencil computations

Prospect compiler pipeline



Domain Transformations: replaces some Julia AST nodes with new "domain nodes" for map, reduce, comprehension, and stencil

Parallel Transformations: replaces domain nodes with "parfor" nodes representing parallel for loops

CGen: converts parfor nodes into OpenMP loops

Example: Black-Scholes

```
using ParallelAccelerator

@acc function blackscholes(sptprice::Array{Float64,1},
                           strike::Array{Float64,1},
                           rate::Array{Float64,1},
                           volatility::Array{Float64,1},
                           time::Array{Float64,1})
    logterm = log10(sptprice ./ strike)
    powterm = .5 .* volatility .* volatility
    den = volatility .* sqrt(time)
    d1 = (((rate .+ powterm) .* time) .+ logterm) ./ den
    d2 = d1 .- den
    NofXd1 = cndf2(d1)
    ...
    put = call .- futureValue .+ sptprice
end

put = blackscholes(sptprice, initStrike, rate, volatility, time)
```

Black-Scholes demo

runStencil example: Gaussian blur

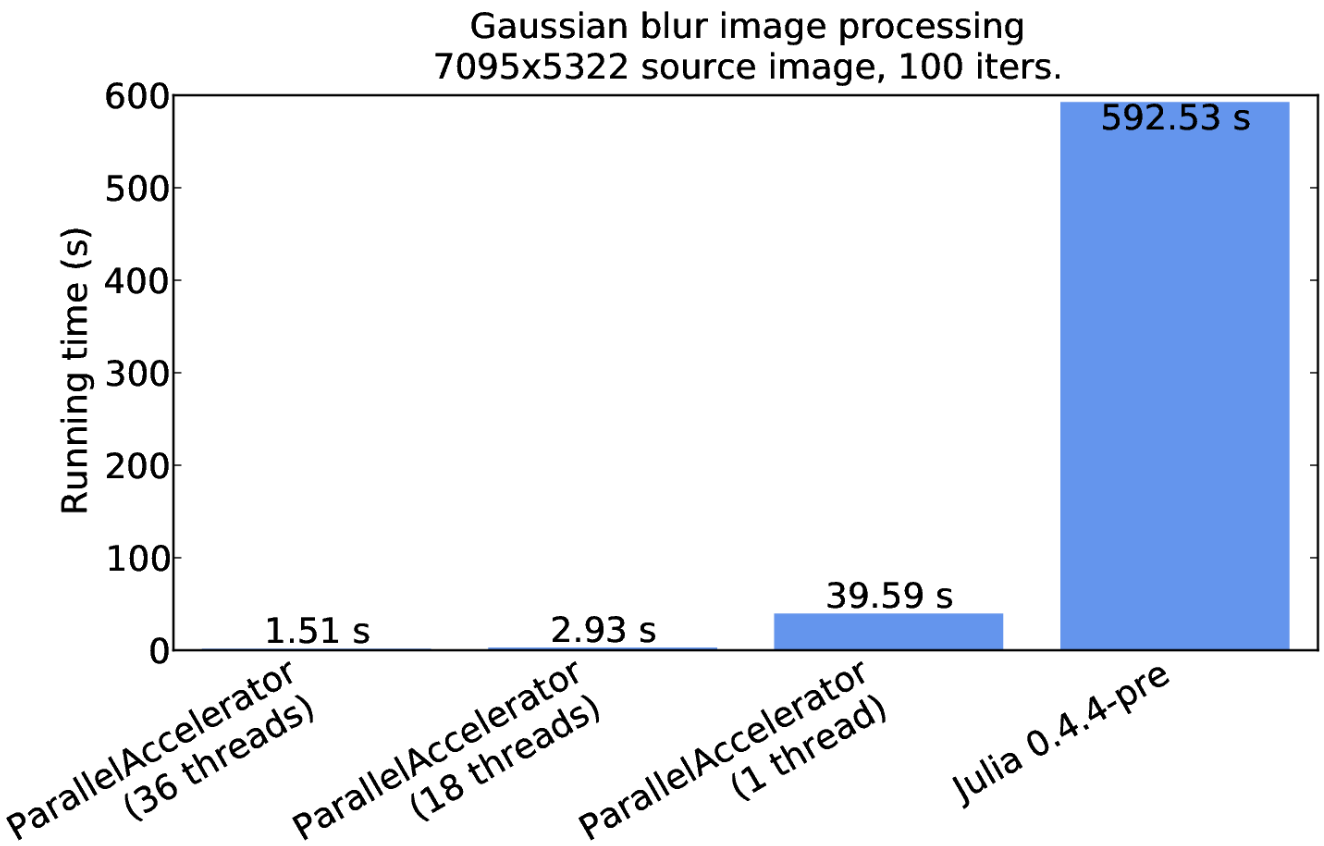
```
using ParallelAccelerator

@acc function blur(img::Array{Float32,2}, iterations::Int)
    buf = Array(Float32, size(img)...)
    runStencil(buf, img, iterations, :oob_skip) do b, a
       b[0,0] =
            (a[-2,-2] * 0.003  + a[-1,-2] * 0.0133 + a[0,-2] * ...
             a[-2,-1] * 0.0133 + a[-1,-1] * 0.0596 + a[0,-1] * ...
             a[-2, 0] * 0.0219 + a[-1, 0] * 0.0983 + a[0, 0] * ...
             a[-2, 1] * 0.0133 + a[-1, 1] * 0.0596 + a[0, 1] * ...
             a[-2, 2] * 0.003  + a[-1, 2] * 0.0133 + a[0, 2] * ...
       return a, b
    end
    return img
end

img = blur(img, iterations)
```

Gaussian blur demo

More results



Data from 03/02/2016
2 Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.3GHz processors, 18 cores each (36 cores total)
128 GB RAM

Caveats

Package load time

- Can be mitigated using `ParallelAccelerator.embed()`

Compiler limitations

- Only a subset of Julia is accelerated
- Compiler tries to transitively compile the whole call chain
- If anything fails to compile, fall back to standard Julia

To learn more…

Guest post on the Julia blog:

julialang.org/blog/2016/03/parallelaccelerator

Our GitHub repo:

github.com/IntelLabs/ParallelAccelerator.jl

Thanks!