A tour of ParallelAccelerator.jl

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

Lindsey Kuper

Parallel Computing Lab, Intel Labs

January 18, 2017

Project Contributors:
Todd Anderson, Hai Liu, Ehsan Totoni, Jan Vitek, Tatiana Shpeisman

The productivity/performance tradeoff

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?

But can't it be done automatically?

After decades of research, automatic parallelization has proved elusive

Most auto-parallelization techniques only work in a limited setting

Efficient compilation of dynamic languages is hard, too, because we may not know the types of program expressions until runtime

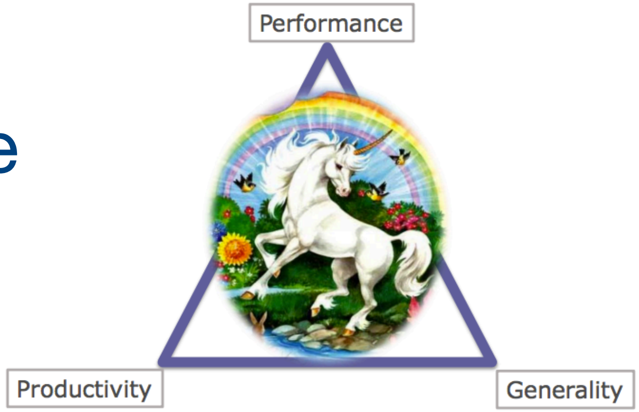
The result: no "sufficiently smart compiler" for you!

Performance, productivity, *generality*: a "pick two out of three" trilemma

Idea: sacrifice generality for productivity and performance

Delite (Brown *et al*., 2011),

SEJITS (Catanzaro *et al.*, 2009), …

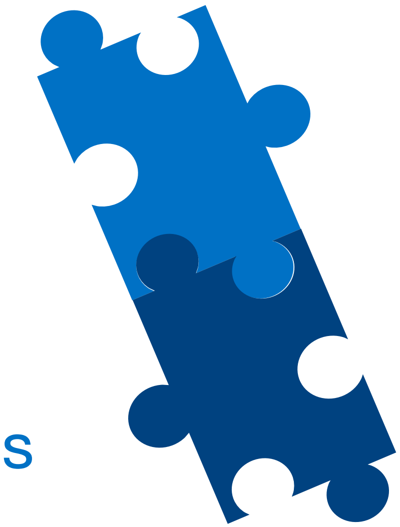


[Olokutun *et al.*, 2012]

But, some issues with high-performance DSLs:

- Steep learning curve
- Functionality cliffs
- Lack of robustness

ParallelAccelerator



A *non-invasive* DSL embedded in Julia

- Accelerate existing language constructs
  - Aggregate array operations; array comprehensions
- Support additional domain-specific constructs (`runStencil`)
  - …with two implementations: library and native

A combination compiler-library solution

- Run in library mode during development and debugging
- Run in native mode for high performance at deployment

ParallelAccelerator

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:

- Find *implicit data-parallel patterns* in a subset of Julia code
- Compile to explicit parallel for loops
- Minimize run-time overheads

Example: Black-Scholes option pricing

```
using ParallelAccelerator

@acc 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

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

Example: Black-Scholes option pricing

```
using ParallelAccelerator

@acc 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

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

Example: Black-Scholes option pricing

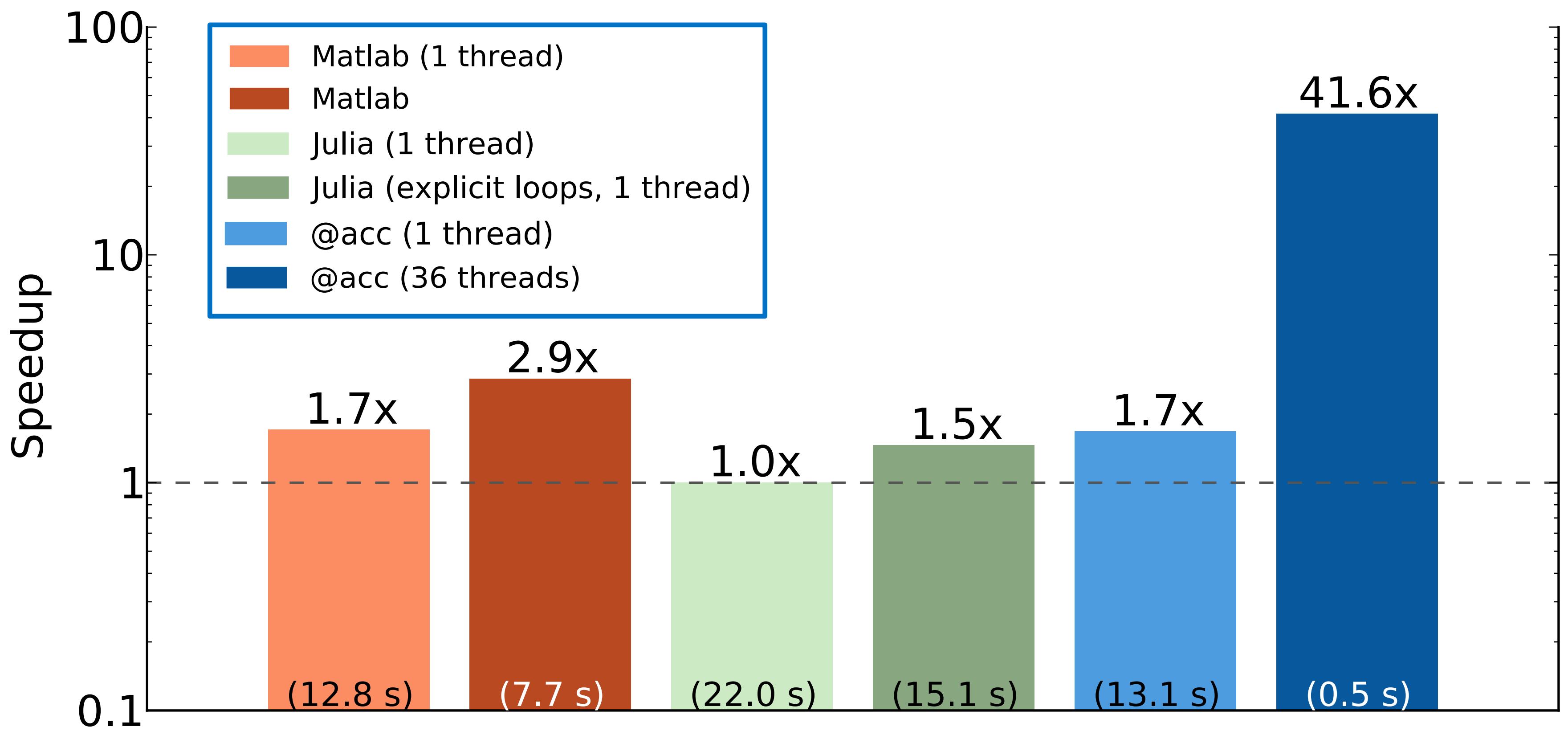
```
using ParallelAccelerator

@acc 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

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

Black-Scholes performance results

Running on arrays of 100 million elements:



Data collected on 12/31/2016
2 Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.3GHz processors, 18 cores each (36 cores total)
128 GB RAM
Julia version 0.5.0; Matlab version R2015a

Data-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

runStencil example: Gaussian blur

```
using ParallelAccelerator

@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] * 0.003  + 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)
```

runStencil example: Gaussian blur

```
using ParallelAccelerator

@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] * 0.003  + 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)
```

runStencil example: Gaussian blur

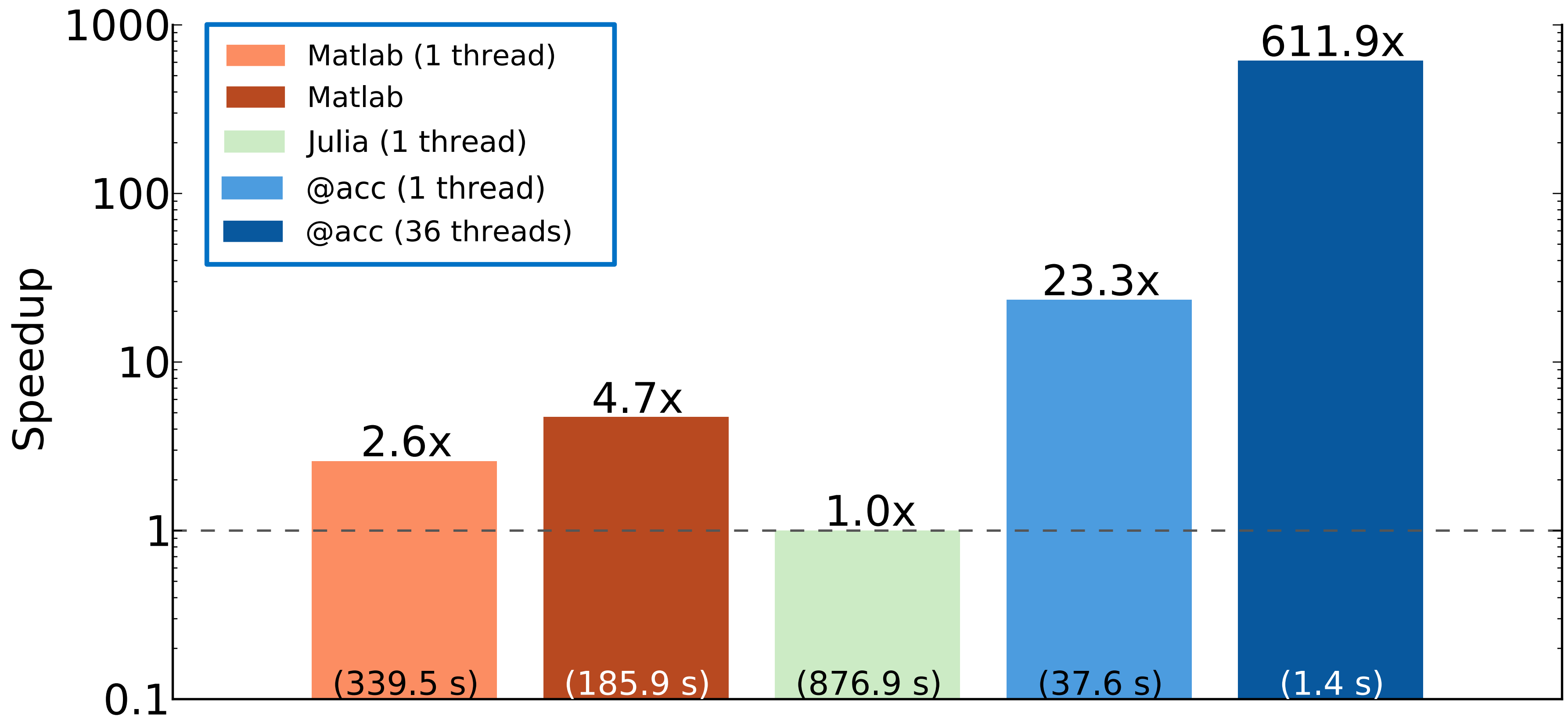
```
using ParallelAccelerator

@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] * 0.003  + 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 performance results

Running on a 7095x5322 source image for 100 iterations:



Data from 12/31/2016
2 Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.3GHz processors, 18 cores each (36 cores total)
128 GB RAM
Julia version 0.5.0; Matlab version R2015a

Why Julia?

Open source

Faster than many scientific computing languages

Supports array-style programming

Under active development; strong community

Julia's support for programming in the large, macro system, and introspection capabilities (`code_typed`) made it feasible!

ParallelAccelerator caveats

Package load time is too long

Only a few workloads investigated so far; we need more

If code isn't in array style, ParallelAccelerator can't help you

Compiler limitations:

- Only a subset of Julia is accelerated
- Compiler tries to transitively compile the whole call chain; if anything fails to compile, it falls back to standard Julia
- The new native threading backend addresses these limitations, but is ~2x slower

To learn more…

The Julia blog:

julialang.org/blog/2016/03/parallelaccelerator

Our GitHub repo:

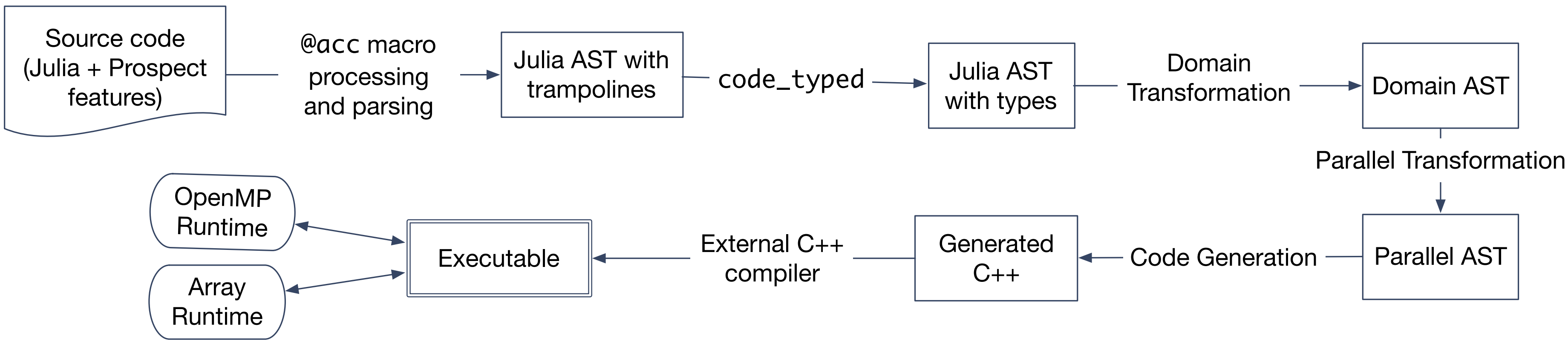
github.com/IntelLabs/ParallelAccelerator.jl

Thanks!

@lindsey

lkuper

ParallelAccelerator compiler pipeline



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

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

CGen: converts parfor nodes into OpenMP loops