Introduction to Writing High Performance Julia

Arch D. Robison

Intel Corporation

Goal

Learn how to write Julia code that is:

- Generic
- Performant
- Concise

Disclaimers

Presentation focused on numerics

- String issues not covered

Julia and its compiler are evolving.

- Code dumps and performance are for Julia 0.4.5

Covers only single-threaded execution

- See afternoon workshop for parallelism

My machine is not your machine

Map maker’s dilemma

Main Topics

Julia tool chain

Hardware considerations

Julia type system

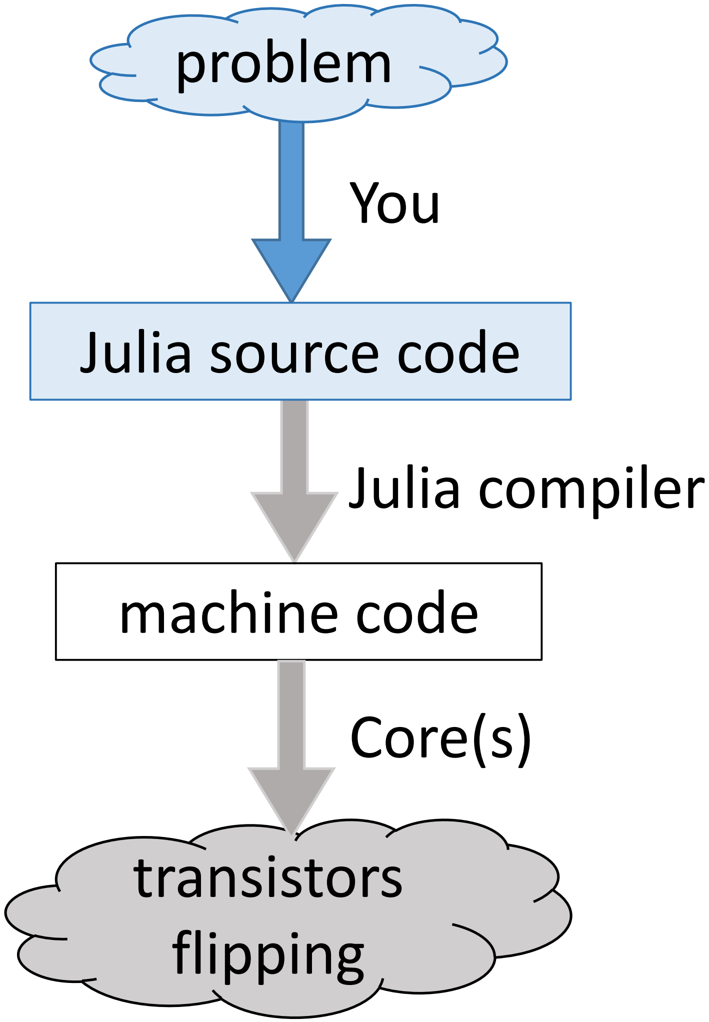
- break + homework problems

Optimizations: automatic vs. manual

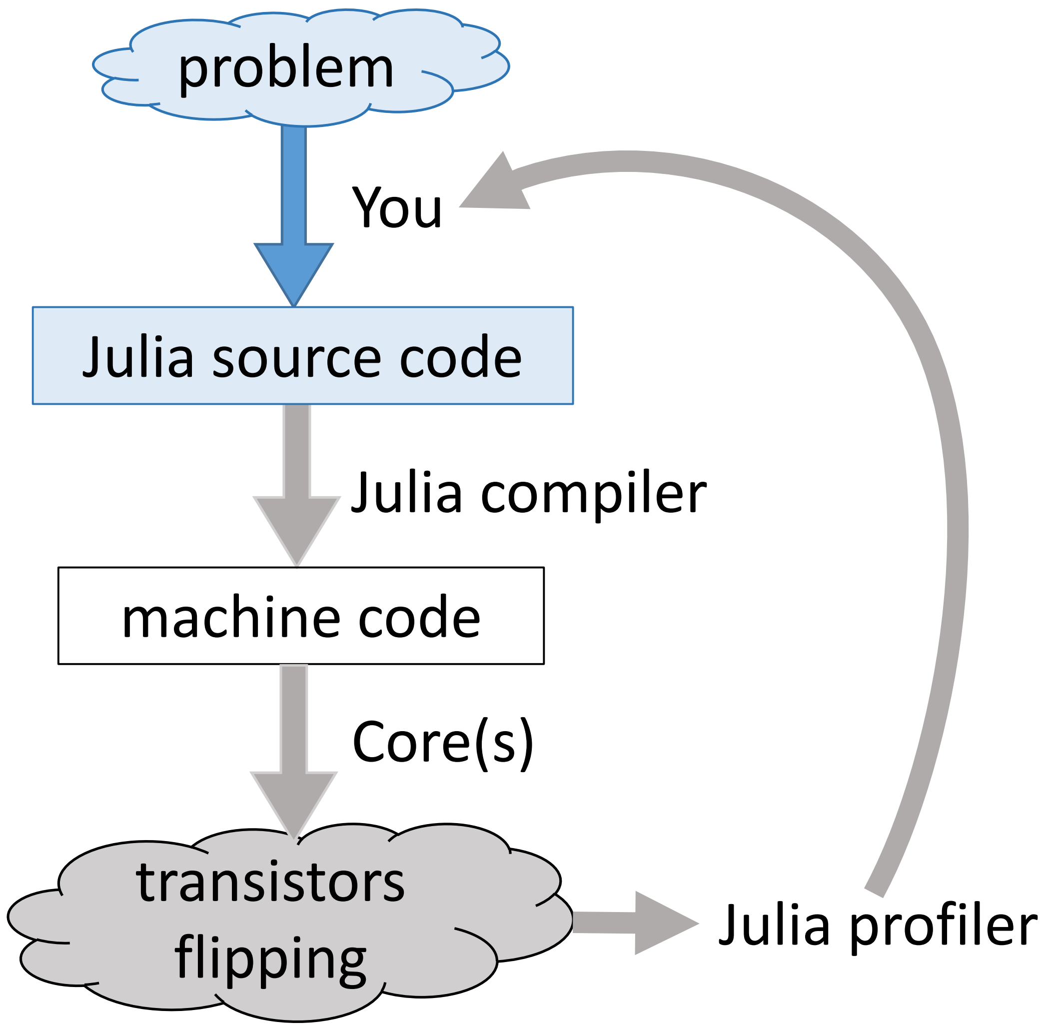
- break + homework problems

Vectorization (SIMD loops)

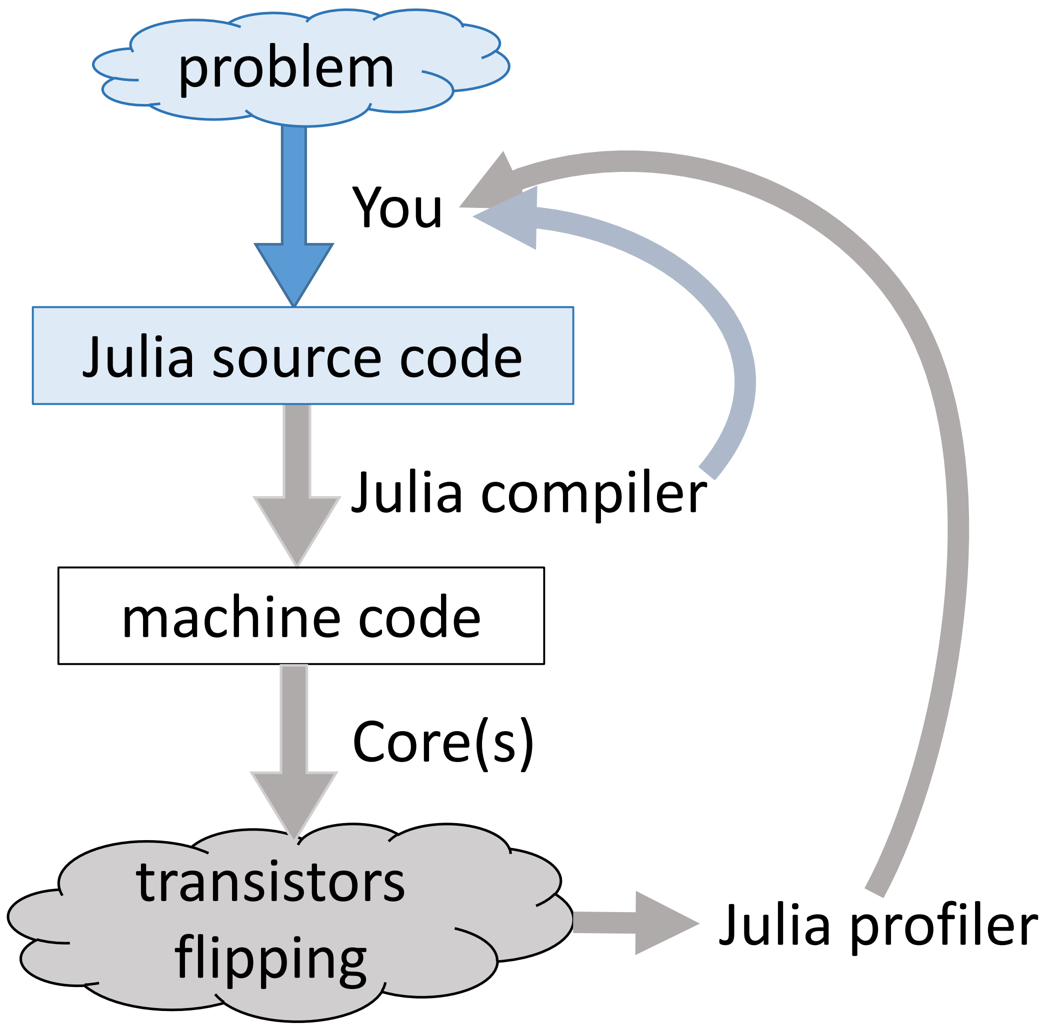
The Julia Tool Chain



Completing the Loop



One More Loop



Two Time Profilers

Profile module (built into Julia)

```
julia> @profile foo()

julia> Profile.print()
```

Intel® VTune™ Amplifier

- Graphical interface
- Has both source and assembly views
- Requires building Julia from source.
- Add “USE_INTEL_JITEVENTS = 1” to Make.user before building Julia

Timing/Profiling Gotchas

Not warming up system

- JITing code
- Caches

Unstable processor frequency

Ignoring warmup if it is important

Timing too short a run

Timing something that optimizer removes

- Do not rely on obfuscation
- Print value that requires doing the computation!

Warming Up System

```
function triple(a)
    i = 1
    n = length(a)
    while i<=n
        a[i] *= 3
        i += 1
    end
end

a = rand(Float32,10000)
@time triple(a)
@time triple(a)
println(hash(a))
```

0.002785 seconds (1.97 k allocations: 108.609 KB)

0.000014 seconds (4 allocations: 160 bytes)

Heap Allocation Profiling

`julia --track-allocation=user bar.jl`

Creates file `bar.jl.mem`

File `bar.jl`

```
function tally(x)
    s = 0
    for v in x
        s += v
    end
    s
end

function wrapper()
    y = rand(1000)
    println(tally(y))
    Profile.clear_malloc_data()
    println(tally(y))
end

wrapper()
```

Call workload from wrapper function to avoid misattribution bug.

Force compilation of workload

Clear allocation counters.

Run the compiled workload.

Heap Allocation Profile

File bar.jl.mem

```
   - function tally(x)
       0     s = 0
       0     for v in x
   32000         s += v
       -     end
       0     s
       - end
       -
       - function wrapper()
       0     y = rand(1000)
       0     println(tally(y))
       0     Profile.clear_malloc_data()
     592     println(tally(y))
       - end
       -
       - wrapper()
```

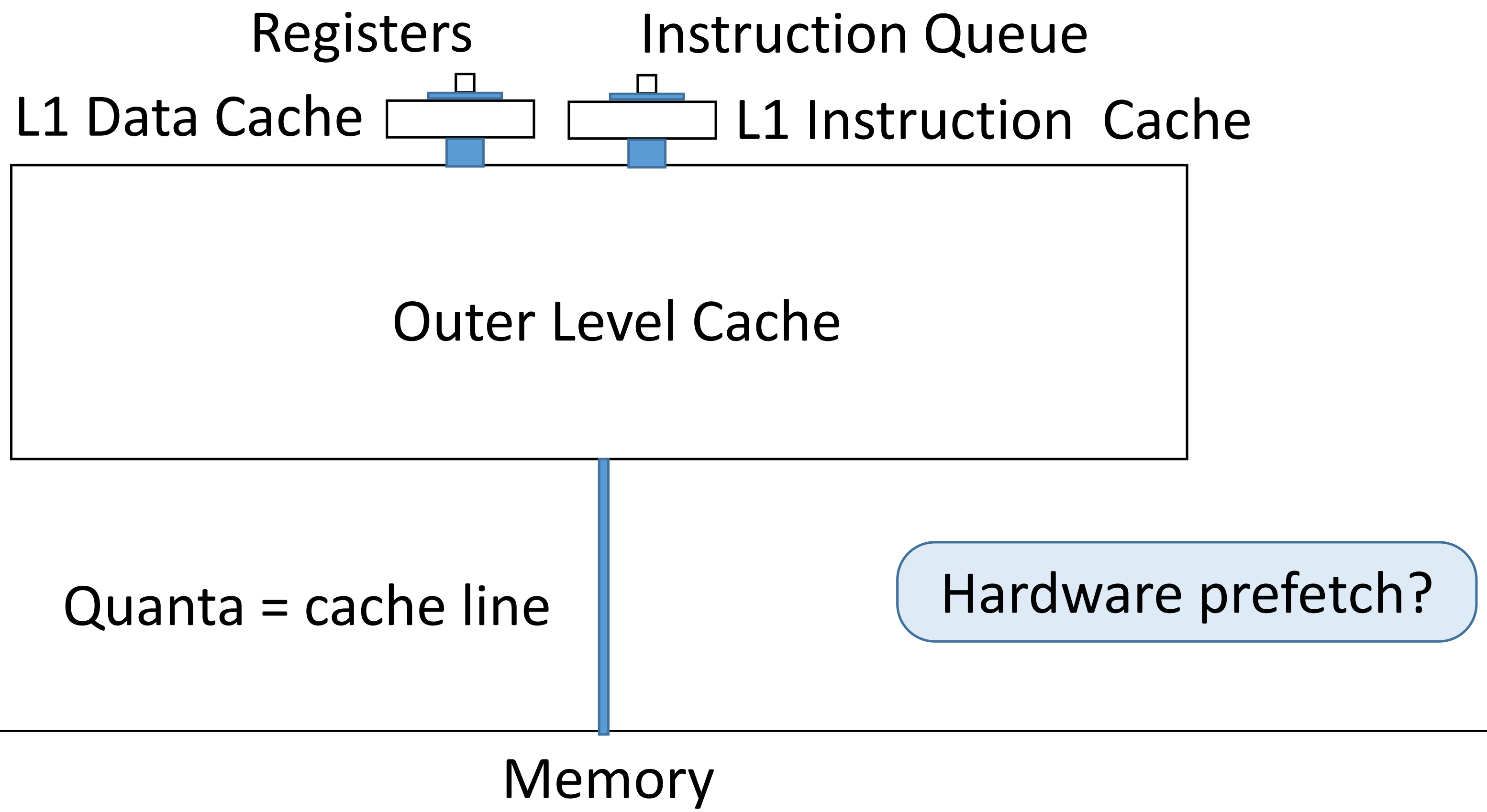
Hardware Resources

Memory

Compute

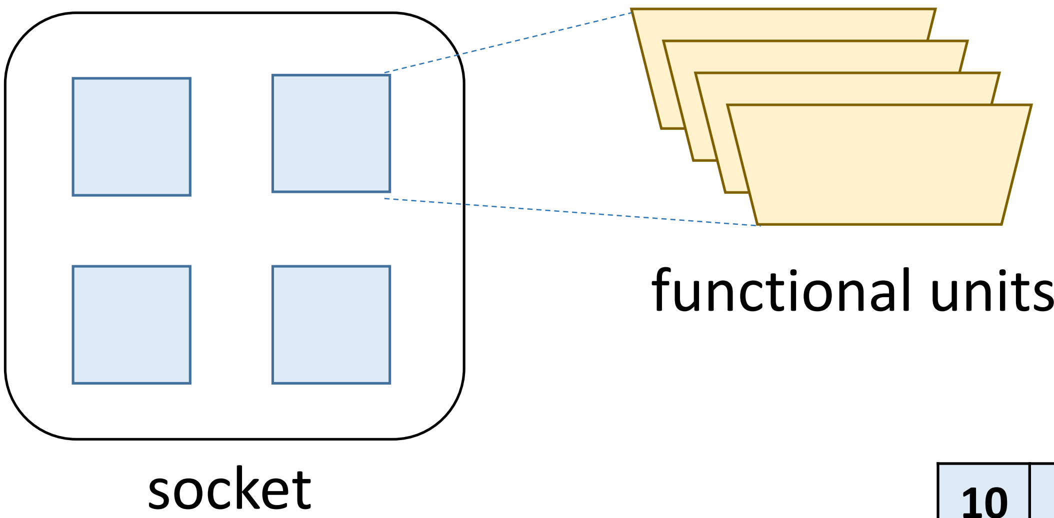
Memory Hierarchy

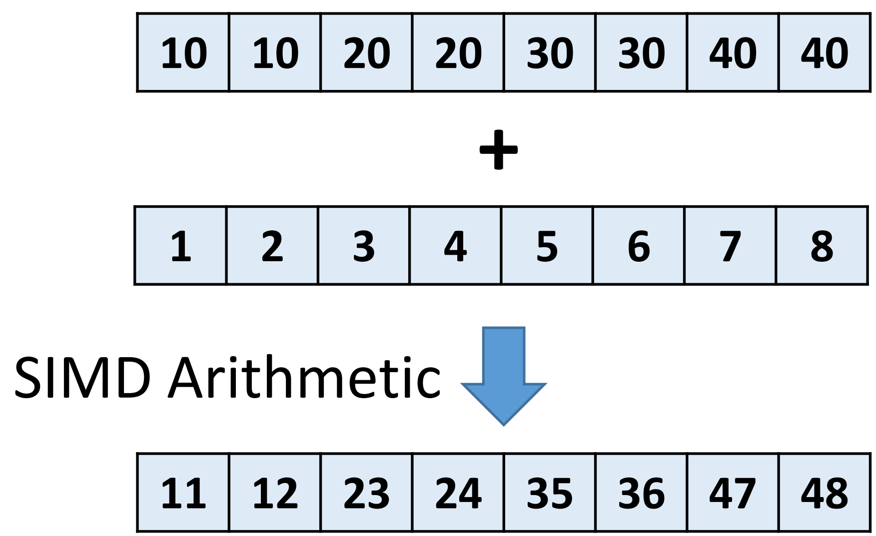
Size
Latency
Bandwidth



Compute Resources

Cores
SIMD Width
Latency
Throughput





Ideal Use of Hardware

SIMD units going at full speed

Most memory accesses hit L1 cache

No stalls from cache misses

- Effective prefetching
- Out of order pipeline
- Hardware multithreading

Float32 often faster than Float64

Uses half the bandwidth

Has half the cache footprint

SIMD can process twice as many values

Semantics vs. Implementation

Semantics — Reason about **correctness**

Implementation — Reason about **performance**

C Semantics for Variables

```
int foo() {
   int x;
   x = 2;
   x = 3.1;
   return x;
}
```

A variable is a **location** in memory.

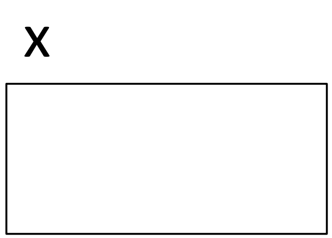
x

3

C vs. Julia Semantics for Variables

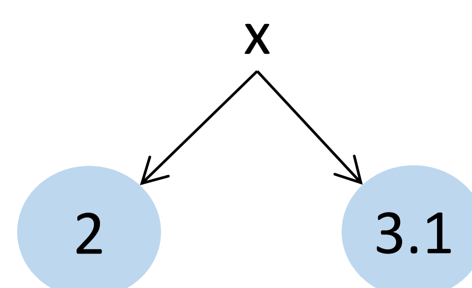
```c
int foo() {
    int x;
    x = 2;
    x = 3.1;
    return x;
}
```

A variable is a **location** in memory.




```julia
function foo()
    local x
    x = 2
    x = 3.1
    x
end
```

A variable is a **name** bound to a value.



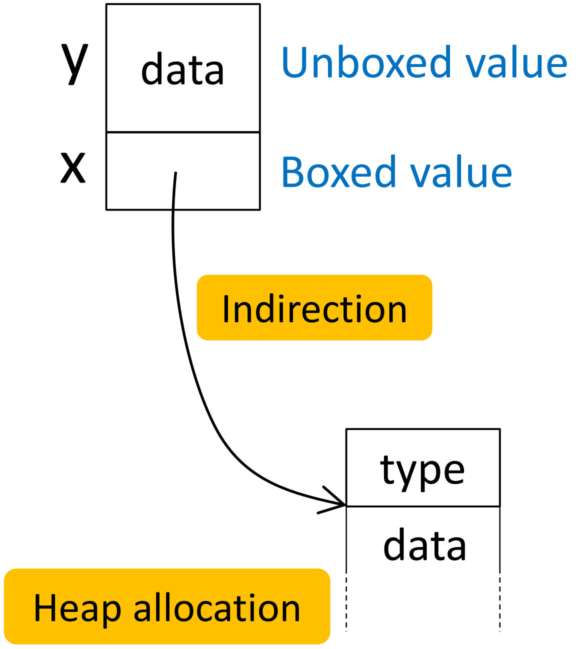
Boxing

Used for objects without known compile-time type.

Compiler works to avoid it.

```
function bar()
    local x, y
    x = 2
    x = 3.1
    y = 4.0
    x+y
end
```

Generic dispatch



Recap

Julia source
- Name is bound to value
- Optional type declarations

Julia compiler

Machine code
- Values stored in locations.
- Values of unpredictable type must be boxed.

Semantics

Implementation

Julia Compilation & Introspection

Parse source into syntax tree

Expand macros

Lower syntax tree

code_lowered

Type Inference

code_warntype, code_typed

Build LLVM code

Optimize LLVM code

code_llvm

Emit machine code

code_native

code_lowered

```
function bar(x)
    y = 1
    x-y
end
```

```
julia> code_lowered(bar,(Int,))
1-element Array{Any,1}:
 :($(Expr(:lambda, {:x}, {{:y},{{:x,:Any,0},{:y,:Any,18}},{}},
:(begin  # /tmp/bar.jl, line 2:
        y = 1 # line 3:
        return x - y
    end))))
```

```
julia> @code_lowered bar(0))
```
...output same as above...

Alternative macro form

code_warntype

Macro form: @code_warntype

```
function bar(x)
    y = 1
    x-y
end
```

```
julia> code_warntype(bar,(Int,))
Variables:
  x::Int64
  y::Int64

Body:
  begin  # none, line 2:
      y = 1 # none, line 3:
      return (Base.box)(Int64,(Base.sub_int)(x::Int64,y::Int64))
  end::Int64
```

code_llvm

Macro form: `@code_llvm`

```
function bar(x)
    y = 1
    x-y
end
```

```
julia> code_llvm(bar,(Int,))

define i64 @julia_bar_21400(i64) {
top:
  %1 = add i64 %0, -1
  ret i64 %1
}
```

code_native

Macro form: @code_native

```
function bar(x)
    y = 1
    x-y
end
```

```
julia> code_native (bar,(Int,))
 .text
Filename: /tmp/bar.jl
Source line: 3
        pushq   %rbp
        movq    %rsp, %rbp
Source line: 3
        leaq    -1(%rdi), %rax
        popq    %rbp
        ret
```

Summary

Parse source into syntax tree

↓

Expand macros

↓

Lower syntax tree

↓ code_lowered

Type Inference

↓ code_warntype

Build LLVM code

↓

Optimize LLVM code

↓ code_llvm

Emit machine code

code_native

Most useful level for *understanding* type-related performance issues.

Most useful level for *detecting* performance issues.

Concrete vs. Non-Concrete Types

Non-Concrete
(require *boxing*)

`Any`

`Integer`

`Union{Int32,Int64}`

`Vector{T}`

Concrete

`Vector{Int}`

```
type Foo
    x::Int
    y::Float32
end
```

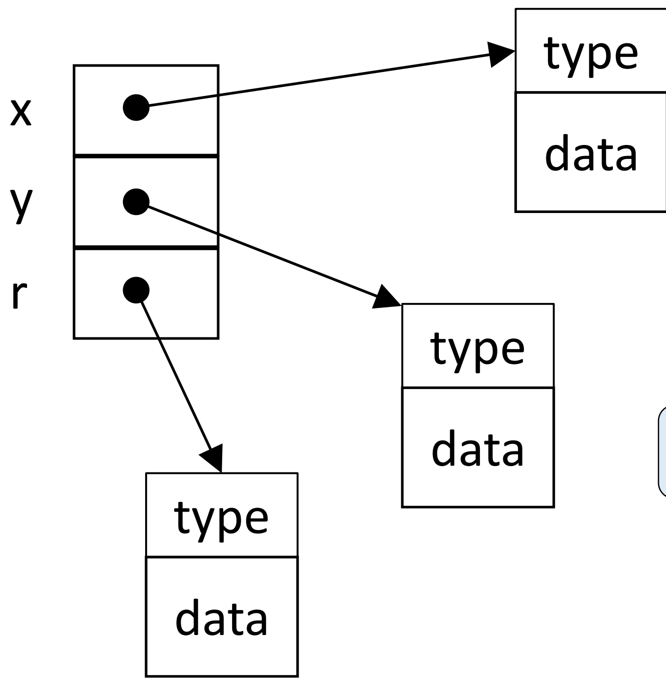
`Int`

`Tuple{Int,Float32}`

Quicksand Types?

```
type Circle
    x
    y
    r
end
```

`Circle` is a concrete type.
But `x`, `y`, and `r` are implicitly Any.

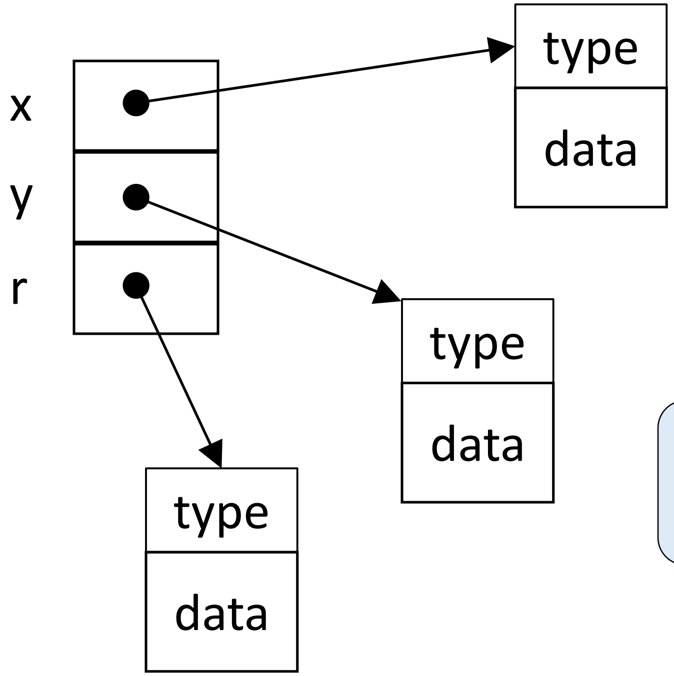


Requires boxing.

Still Not Concrete

```
type Circle
    x :: Real
    y :: Real
    r :: Real
end
```

Still requires boxing, since Real is abstract type.



Boxes known to hold some concrete subtype of Real.

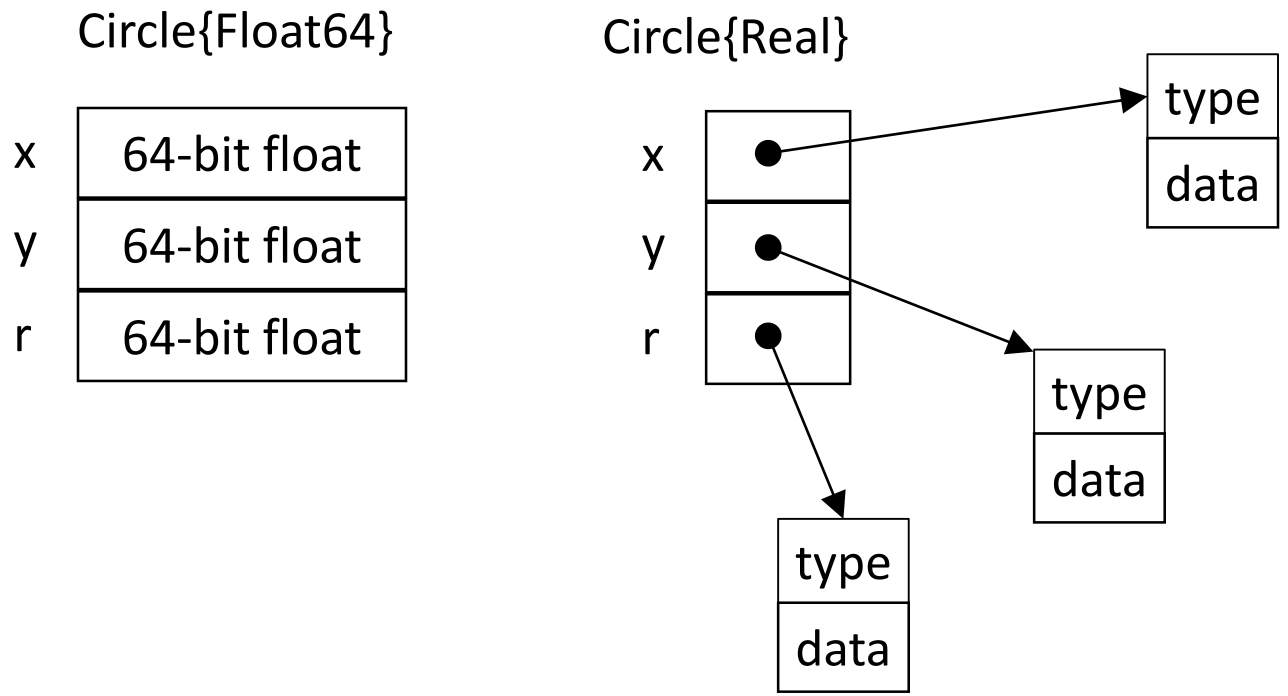
Concrete

```
type Circle
    x :: Float64
    y :: Float64
    r :: Float64
end
```

x	64-bit float
y	64-bit float
r	64-bit float

Generalize with Parametric Types

```
type Circle{T<:Real}
    x :: T
    y :: T
    r :: T
end
```



Big Impact

```
type Circle{T<:Real}
    x :: T
    y :: T
    r :: T
end

touch(c,d) = (c.x - d.x)^2 + (c.y-d.y)^2 ≤ (c.r + d.r)^2

function counttouch(a)
    k = 0
    for i=2:length(a), j=1:i-1
        k += touch(a[i],a[j])
    end
    k
end

for T in [Real,Float64]
    a = Circle{T}[Circle{T}(rand(),rand(),rand()*.1) for i=1:1000]
    println(T)
    for trial=1:3
        @time counttouch(a)
    end
end
```

```
Real
  0.160440 seconds (3.99 M allocations: 61.080 MB, 33.08% gc time)
  0.094418 seconds (3.98 M allocations: 60.752 MB, 3.94% gc time)
  0.095533 seconds (3.98 M allocations: 60.752 MB, 3.40% gc time)
Float64
  0.008329 seconds (6.33 k allocations: 295.941 KB)
  0.001684 seconds (1 allocation: 16 bytes)
  0.001685 seconds (1 allocation: 16 bytes)
```

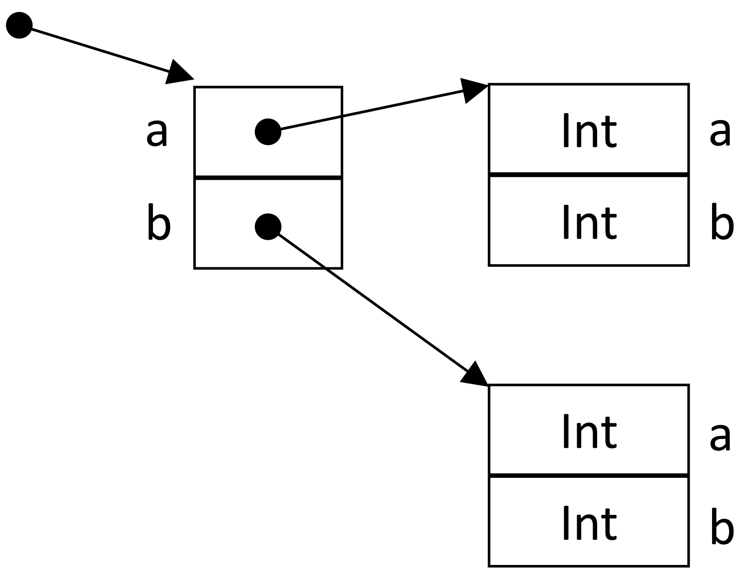
Type vs. Immutable

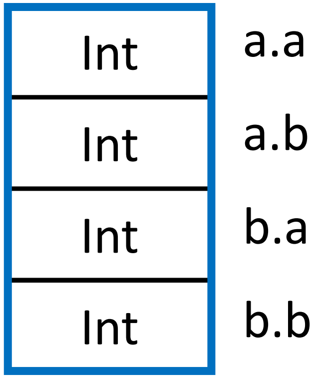
```
type MTwo{T}
    a :: T
    b :: T
end

f() = MTwo(MTwo(1,2),MTwo(3,4))
```

```
immutable ITwo{T}
    a :: T
    b :: T
end

g() = ITwo(ITwo(1,2),ITwo(3,4))
```





Compiler's Knowledge of Types

Context	Compiler Treatment
parameter global const variable	usually known exactly
local variable return value	inferred
fields of structures	as declared
global variable	unknown

Be Nice to Type Inference!

The big performance issue in Julia.

- Julia functions are polymorphic.
- Hardware is monomorphic.

Impact of type uncertainty:

- Boxing
  - Heap allocation
  - Garbage collection (GC)
- Run-time dispatch of calls
  - Table scanning
  - Call cannot be inlined.

Example

```
function qux(x)
    if x≥0
        y=x
    else
        y=0
    end
    y+1
end
```

```
julia> code_llvm(qux,(Int,))

define i64 @julia_qux_21202(i64) {
top:
  %1 = icmp slt i64 %0, 0
  br i1 %1, label %L1, label %if

if:                                   ; preds = %top
  %phitmp = add i64 %0, 1
  br label %L1

L1:                                   ; preds = %if, %top
  %y.0 = phi i64 [ %phitmp, %if ], [ 1, %top ]
  ret i64 %y.0
}
```

Ouch!

```
function qux(x)
    if x≥0
        y=x
    else
        y=0
    end
    y+1
end
```

julia> code_llvm(qux,(Float32,))

```
define %jl_value_t* @julia_qux_21208(float) {
top:
  %1 = alloca [5 x %jl_value_t*], align 8
  %.sub = getelementptr inbounds [5 x %jl_value_t*]* %1, i64 0, i64 0
  %2 = getelementptr [5 x %jl_value_t*]* %1, i64 0, i64 2
  %3 = getelementptr [5 x %jl_value_t*]* %1, i64 0, i64 3
  store %jl_value_t* inttoptr (i64 6 to %jl_value_t*), %jl_value_t** %.sub,
  %4 = getelementptr [5 x %jl_value_t*]* %1, i64 0, i64 1
  %5 = load %jl_value_t*** @jl_pgcstack, align 8
  %.c = bitcast %jl_value_t** %5 to %jl_value_t*
  store %jl_value_t* %.c, %jl_value_t** %4, align 8
  store %jl_value_t** %.sub, %jl_value_t*** @jl_pgcstack, align 8
  store %jl_value_t* null, %jl_value_t** %2, align 8
  store %jl_value_t* null, %jl_value_t** %3, align 8
  %6 = getelementptr [5 x %jl_value_t*]* %1, i64 0, i64 4
  store %jl_value_t* null, %jl_value_t** %6, align 8
  %7 = fcmp ult float %0, 0.000000e+00
  br i1 %7, label %L1, label %if

if:                                               ; preds = %top
  %8 = call %jl_value_t* @jl_box_float32(float %0)
  br label %L1

L1:                                               ; preds = %if, %top
  %storemerge = phi %jl_value_t* [ %8, %if ], [ inttoptr (i64 14067693868654
  store %jl_value_t* %storemerge, %jl_value_t** %2, align 8
  store %jl_value_t* %storemerge, %jl_value_t** %3, align 8
  store %jl_value_t* inttoptr (i64 140676938686592 to %jl_value_t*), %jl_val
  %9 = call %jl_value_t* @jl_apply_generic(%jl_value_t* inttoptr (i64 140
%jl_value_t*), %jl_value_t** %3, i32 2)
  %10 = load %jl_value_t** %4, align 8
  %11 = getelementptr inbounds %jl_value_t* %10, i64 0, i32 0
  store %jl_value_t** %11, %jl_value_t*** @jl_pgcstack, align 8
  ret %jl_value_t* %9
}
```

GC-related stuff

Boxing x

Run-time dispatch

Root Problem

```
function
qux(x)
    if x≥0
        y=x
    else
        y=0
    end
    y+1
end
```

```
julia> code_warntype(qux,(Float32,))
Variables:
  x::Float32
  y::Any
  ####fx#7042#7043::Float32

Body:
  begin  # none, line 2:
      NewvarNode(:y)
      ####fx#7042#7043 = (Base.box)(Float32,(Base.sitofp)(Float32,0))
 unless
(Base.box)(Base.Bool,(Base.or_int)((Base.lt_float)(####fx#7042#7043::F
loat32,x::Float32)::Bool,(Base.box)(Base.Bool,(Base.and_int)((Base.eq_
float)(####fx#7042#7043::Float32,x::Float32)::Bool,(Base.box)(Base.Boo
l,(Base.or_int)((Base.eq_float)(####fx#7042#7043::Float32,9.223372f18)
::Bool,(Base.sle_int)(0,(Base.box)(Int64,(Base.fptosi)(Int64,####fx#70
42#7043::Float32)))::Bool)))))) goto 0 # none, line 3:
      y = x::Float32
      goto 1
      0:  # none, line 5:
      y = 0
      1:  # none, line 7:
      return y::Union{Float32,Int64} + 1::Union{Float32,Int64}
  end::Union{Float32,Int64}
```

Wrong Way to Fix (Usually)

```
function qux(x::Int)
    if x≥0
        y=x
    else
        y=0
    end
    y+1
end
```

Now code is type-stable, but not generic. ☹

Better Way: Use Conversion

```
function qux(x)
    if x≥0
        y=x
    else
        y=zero(x)
    end
    y+1
end
```

```
function qux{T}(x::T)
    if x≥0
        y=x
    else
        y=zero(T)
    end
    y+1
end
```

```
function qux{T}(x::T)
    if x≥0
        y=x
    else
        y=convert(T,0)
    end
    y+1
end
```

```
function qux{T}(x::T)
    if x≥0
        y=x
    else
        y=T(0)
    end
    y+1
end
```

After Repair

```
function qux(x)
    if x≥0
        y=x
    else
        y=zero(x)
    end
    y+1
end
```

```
julia> code_llvm(qux,(Float32,))

define float @julia_qux_21422(float) {
top:
  %1 = fcmp ult float %0, 0.000000e+00
  br i1 %1, label %L1, label %if

if:                                   ; preds = %top
  %phitmp = fadd float %0, 1.000000e+00
  br label %L1

L1:                                   ; preds = %if, %top
  %y.0 = phi float [ %phitmp, %if ],
                   [ 1.000000e+00, %top ]
  ret float %y.0
}
```

Mixed-Mode Arithmetic Not a Problem

```
function f(x)
    y = x + 1.0
    z = y + 2im
    z
end
```

Mixed-mode arithmetic is fine.
Type promotions are predictable.

```
julia> code_warntype(f,(Int,))
Variables:
  x::Int64
  y::Float64
  z::Complex{Float64}
...
```

Common Problem: Reductions

Initialize accumulation variable with value of right type!

Slow way to sum collection x

```
function tally(x)
    s = 0
    for v in x
        s += v
    end
    s
end
```

Usually much faster

```
function tally(x)
    s = zero(eltype(x))
    for v in x
        s += v
    end
    s
end
```

Type Stability Revisited

```
# Solve quadratic equation ax^2+bx+c
function roots(a,b,c)
    d = b^2-4*a*c
    if d≥0
        # Real roots
        (-b+√d)/2a, (-b-√d)/2a
    else
        # Complex roots
        (-b+im*√-d)/2a, (-b-im*√-d)/2a
    end
end
```

@code_warntype Revisited

```
julia> @code_warntype roots(1,1,1)
Variables:
  a::Int64
  b::Int64
  c::Int64
  d::Int64
  ##xs#7101::Tuple{}
  ##re#7102::Float64

Body:
  begin  # /localdisk/adrobiso/julia-0.4.5/roots.jl, line 2:
```

...lots of code...

```
 end::Tuple{Number,Number}
```

So far, so good....

...oops!

Possible Solution

```
function roots(a,b,c)
    d = b^2-4*a*c
    if d≥0
        complex((-b+√d)/2a), complex((-b-√d)/2a)
    else
        (-b+im*√-d)/2a, (-b-im*√-d)/2a
    end
end
```

Pop Quiz

Which of the following functions are type stable?

```
re(x) = x<0 ? 0 : 1

mi(x) = x==0 ? -x : x

fa(x) = x==0 ? 1 : sin(x)/x

so(x,y) = x<y ? x : y

la{T}(x::T, y::T) = x<y ? x : y
```

Using Promotions

```
la{T}(x::T, y::T) = x<y ? x : y

la(x,y) = la(promote(x,y)...)
```

Global Variables

No type inference for reassignable global variables.

- Julia is dynamic -- more assignments could be added later.

Work arounds

- Avoid global variables
- Declare single-assignment global variables `const`
- Wrapper trick
- Use explicit type-check or force conversion
- Pass as parameter to helper function

Using `const`

Slow

```
function foo(x)
    s = 0
    for v in x
        s += v≥β
    end
    s
end

β = 0.5
```

With const

```
function foo(x)
    s = 0
    for v in x
        s += v≥β
    end
    s
end

const β = 0.5
```

Gained performance, but lost generality.

Note on const

const means that identifier is never rebound

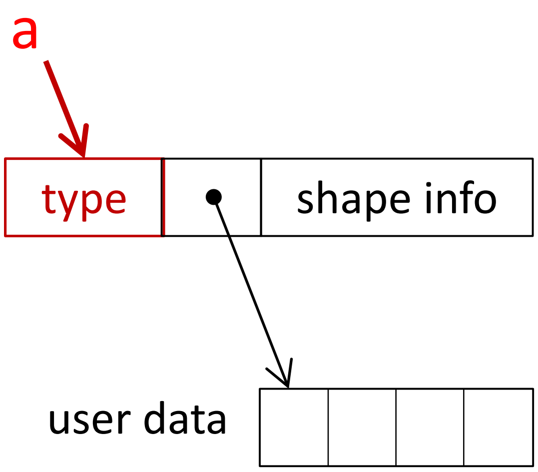
Does NOT mean that object is invariant

```
julia> const a = [1,2,3];

julia> push!(a,4);

julia> a[:] = 0;

julia> a = [0,0,0,0];
Warning: redefining constant a
```



Ref Hack

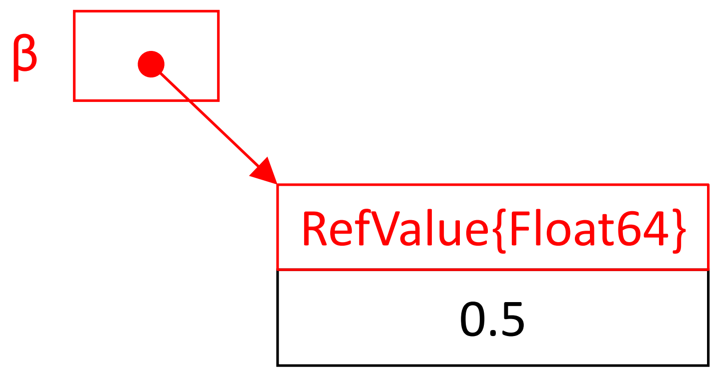
Excerpt from base/refpointer.jl
in Julia standard library.

```
type RefValue{T} <: Ref{T}
    x::T
end
```

```
function foo(x)
    s = 0
    for v in x
        s += v≥β[]
    end
    s
end

const β = Ref(0.4)

β[] = 0.5  # Assignment allowed
```



Thanks to Kristoffer Carlsson for pointing
out that Ref could be used for this hack.

Using Type Assertion

Slow

```
function foo(x)
    s = 0
    for v in x
        s += v≥β
    end
    s
end

β = 0.5
```

Explicit check

```
function foo(x)
    s = 0
    for v in x
        s += v ≥ β::Float64
    end
    s
end

β = 0.5
```

Gained performance, but lost generality.

Helper Function

Original slow version

```
function foo(x)
    s = 0
    for v in x
        s += v≥β
    end
    s
end

β = 0.5
```

Faster version

```
function foo_aux(x,b)
    s = 0
    for v in x
        s += v≥b
    end
    s
end

foo(x) = foo_aux(x,β)

β = 0.5
```

generic dispatch

Gotcha

Still slow!

```
function foo(x)
    s = 0
    b = convert(eltype(x), β)
    for v in x
        s += v ≥ b
    end
    s
end

β = 0.5
```

Compiler cannot infer result type of `convert` unless it infers the type of **both** arguments.

Type Assertion to the Rescue

Fast

```
function foo(x)
    s = 0
    b = convert(eltype(x), β) :: eltype(x)
    for v in x
        s += v ≥ b
    end
    s
end

β = 0.5
```

help type inference

Julia 0.4 Note

Julia 0.4 defines:

```
call{T}(::Type{T}, arg) = convert(T, arg)::T
```

So

```
b = convert(eltype(x), β) :: eltype(x)
```

can be written as:

```
b = eltype(x)(β)
```

Type Guidelines for Julia

Julia specializes functions

- Customizes function to its parameter types
- Type declarations on parameters do not help performance

Type inference does forward flow analysis

- Code performs slowly when inference cannot deduce concrete types.

Look out for type instability for variables assigned on multiple paths.

- Pay attention to how 0 is used.

Spot lack of concrete types

- `@code_warntype`

Avoid using global variables in kernels

- Use `const` where applicable
- Ref hack
- Helper function trick

Exercises 1-2

Download from http://tinyurl.com/HPJ2016

Do these two problems:

- ex1.jl
- ex2.jl

A Solution to Exercise 1

```
# Compute alternating sum of array a
function altsum(a)
    s = 0
    s = zero(eltype(a))
    c = 1
    for i in 1:length(a)
        s += c*a[i]
        c = -c
    end
    s
end
```

A Solution to Exercise 2

```
# Compute successor of i in its hailstone cycle
function h(i)
    if i%2==0
        ~~i/2~~
        div(i,2)
    else
        3*i+1
    end
end
```

Reuse Temporary Arrays

Reuse objects instead of reallocating them

New garbage collector in Julia 0.4 reduces impact

Example

```
function next_state(s)
   t = similar(s)
   n = length(s)
   for i=1:n
      t[i] = (s[i]+s[i==n?1:i+1]) % 2
   end
   t
end

function evolve(nstep,state)

   for i=1:nstep
      state = next_state(state)

   end
   state
end
```

```
function next_state!(t, s)

   n = length(s)
   for i=1:n
      t[i] = (s[i]+s[i==n?1:i+1]) % 2
   end

end

function evolve(nstep,state)
   next= similar(state)
   for i=1:nstep
      next_state!(next,state)
      next,state = state,next
   end
   state
end
```

Comprehension Caveat

```
function next_state( s )
    t = similar(s)
    n = length(s)
    for i=1:n
        t[i] = (s[i]+s[i==n?1:i+1]) % 2
    end
    t
end
```

If s is Vector{Int8}, do these functions have similar behavior?

```
function next_state( s )
    n = length(s)
eltype(s)[(s[i]+s[i==n?1:i+1]) % 2 for i=1:n]
end
```

Loop vs. Array Operations vs. BLAS

Array Style (“vectorized”)

```
function foo(c, w, i, j, Δx, Δy)
    Δw = w[:,i]-w[:,j]
    c[:,1] += Δw*Δx
    c[:,2] += Δw*Δy
end
```

Loop Style

```
function foo(c, w, i, j, Δx, Δy)
    (m,n) = size(w)
    for k=1:m
        Δw = w[k,i]-w[k,j]
        c[k,1] += Δw*Δx
        c[k,2] += Δw*Δy
    end
end
```

Exploit BLAS

```
function foo(c, w, i, j, Δx, Δy)
    BLAS.ger!(T(1), w[:,i]-w[:,j], [Δx,Δy], c)
end
```

Recommendations

Use array style if convenient and performance is not critical

- Allocation overhead
- Poor cache behavior

Loop style versus BLAS depends on circumstance

- BLAS highly optimized for large matrices

Read Dahua Lin's exposition

- http://julialang.org/blog/2013/09/fast-numeric/
- Optimization of array style has improved some since it was written

Three Kinds of Optimizations

Automatic

Needs a nudge

Manual

Two Key Questions

An optimization transforms code.

For an instance of code, is the transform:

- **Always** legal?
- **Likely** profitable?

Constant Propagation

```
const a = 2

function f(i)
    x = a+1
    if i>0
        y = i + x + 4
    else
        y = i + 7
    end
    z = y+1
    z
end
```

```
julia> code_llvm(f,(Int,))
define i64 @julia_f_20996(i64) {
top:
  %1 = add i64 %0, 8
  ret i64 %1
}
```

Floating-Point

```
const a = 2

function f(i)
    x = a+1
    if i>0
        y = i + x + 4
    else
        y = i + 7
    end
    z = y+1
    z
end
```

Floating-point addition is not associative!

```
julia> code_llvm(f,(Float64,))

define double @julia_f_20998(double) {
top:
  %1 = fcmp ule double %0, 0.000000e+00
  br i1 %1, label %L, label %if

if:                              ; preds = %top
  %2 = fadd double %0, 3.000000e+00
  %3 = fadd double %2, 4.000000e+00
  br label %L1

L:                               ; preds = %top
  %4 = fadd double %0, 7.000000e+00
  br label %L1

L1:                              ; preds = %L, %if
  %y.0 = phi double [ %4, %L ], [ %3, %if ]
  %5 = fadd double %y.0, 1.000000e+00
  ret double %5
}
```

Order Of Operations Matters

```
const a = 2

function f(i)
    x = a+1
    if i>0
        y = i + (x + 4)
    else
        y = i + 7
    end
    z = y+1
    z
end
```

```
julia> code_llvm(f,(Float64,))

define double @julia_f_20996(double)
{
top:
  %1 = fadd double %0, 7.000000e+00
  %2 = fadd double %1, 1.000000e+00
  ret double %2
}
```

Explicitly grouping less varying operands can help.

Some Unsafe Algebraic Rules for Floating Point

	Counterexample
x+0 → x	x = -0.0
0*x → 0	x = Inf
x/a → x*(1/a)	x = 3.0; a=5.0
(x+y)+z → x+(y+z)	x = 0.1; y = 0.1; z = 1.0
a*x + a*y → a*(x+y)	x = 0.1; y = 0.1; z = 0.5

Apply these rules by hand,
or use @fastmath.

Some Rules That **Do** Work (ignoring signaling NaNs as in Julia)

$x + (-0.0) \rightarrow x$

$1*x \rightarrow x$

$x/a \rightarrow x*(1/a)$ if $a=2$^k

$x+y \rightarrow y+x$

$x*y \rightarrow y*x$

$-(-x) \rightarrow x$

$x +(-y) \rightarrow x - y$

@fastmath

```
const a = 2

function f(i)
    x = a+1
    @fastmath begin
        if i>0
            y = i + x+4
        else
            y = i + 7
        end
        z = y+1
    end
    z
end
```

Now down to one addition!
(but result might differ)

```
julia> code_llvm(f,(Float64,))

define double @julia_f_20996(double) {
top:
  %1 = fadd fast double %0,
8.000000e+00
  ret double %1
}
```

@fastmath grants permission to apply “unsafe algebra”.

Algebra Summary

Compilers are good about rearranging integer arithmetic

- They know everything you learned in grade school, and more.

Less so for floating point

- IEEE rules make much algebra unsafe
- Careful ordering of floating-point can pay off
  - Can enable constant folding and hoisting
- Or use `@fastmath` judiciously

What do these functions do?

```
f(x::Int) = -~x
g(x::Int) = ~-x
```

Inlining

Replaces call site with copy of callee's body

Always legal?

- Yes, as long as correct callee can be determined.

Likely profitable?

- Saves overhead of calling convention
- Enables further specialization of callee
  - Constant propagation
  - Branch elimination
- Might increase instruction cache misses ☹

Example

```
f(x,y) = div(x,y)*y
g(x) = f(x,2)
```

```
julia> code_lowered(g,(Uint,))
julia> code_lowered(g,(UInt,))
1-element Array{Any,1}:
 :($(Expr(:lambda, Any[:x], Any[Any[Any[:x,:Any,0]],Any[],0,Any[]], :(begin  # none, line 1:
        return (Main.f)(x,2)
    end))))
```

no inlining yet

Example

Most operations in Julia are defined by more Julia code.

```
f(x,y) = div(x,y)*y
g(x) = f(x,2)
```

```
julia> code_typed(g,(Uint,))
1-element Array{Any,1}:
 :($(Expr(:lambda, Any[:x], Any[Any[Any[:x,UInt64,0]],Any[],Any[],Any[]], :(begin  # none, line 1:
        return (Base.box)(UInt64,(Base.mul_int)((Base.box)(UInt64,(Base.box)(Int64,(Base.flipsign_int)((Base.box)(Int64,(Base.box)(UInt64,(Base.udiv_int)(x::UInt64,(Base.box)(UInt64,(Base.box)(Int64,(Base.flipsign_int)(2,2)))))),2))),(Base.box)(UInt64,(Base.check_top_bit)(2))))
    end::UInt64))))
```

f and div inlined

Example

f(x,y) = div(x,y)*y
g(x) = f(x,2)

```
julia> code_llvm(g,(UInt,))

define i64 @julia_g_21066(i64) {
pass:
  %1 = and i64 %0, -2
  ret i64 %1
}
```

simplifies to bitwise AND.

Can disable inlining via command line:
`julia --inline=no`

Forcing inlining with @inline

Inlining heuristic guesses performance gain versus cost of code expansion.

- Sometime you might know better

@inline is a slight win here.
Saves calling overhead, but does not enable other transformations.

```julia
@inline function h(n)
    if n%2==0
        n>>1
    else
        3*n+1
    end
end

function hail(n)
    k = 0
    while n>1
        n = h(n)
        k += 1
    end
    k
end
```

Inlining Can Slow Code Down Too!

```
@noinline f(x) = cos(x)^2 - sin(x)^2

function foo(a)
    for i in eachindex(a)
        a[i] = f(a[i])
    end
end

a = Any[sin(i) for i=1:1000000]
@time foo(a)
@time foo(a)
@time foo(a)
```

Bounds Checking in Julia

Julia checks array subscripts by default

- Overhead is usually small (~10% for tight loop)
- But can have BIG impact when it prevents vectorization.

```
function foo(a,b,p)
   for i=1:length(p)
      a[p[i]] += b[i]
   end
end
```

Typical instruction sequence for one check

```
...
  %15 = add i64 %"#s1.0", -1
  %16 = icmp ult i64 %15, %7
  br i1 %16, label %idxend, label %oob

oob:                                        ; preds = %L
  %17 = alloca i64, align 8
  store i64 %"#s1.0", i64* %17, align 8
  call void @jl_bounds_error_ints(%jl_value_t* %2, i64* %17, i64 1)
  unreachable

idxend:                                     ; preds = %L
...
```

Eliminating Bounds Checks

```
function foo(a,b,p)
    for i=1:length(p)
        @inbounds a[p[i]] += b[i]
    end
end
```

Out of bounds subscript could result in random corruption!

Command-line control

`--check-bounds=yes`	Ignore @inbounds
`--check-bounds=no`	Treat everything as @inbounds

Truncating Integers

`Int8(n)`	Checked conversion
`n % Int8`	Modulo conversion

```
julia> Int8(-200)
ERROR: InexactError()
 in call at essentials.jl:56

julia> -200 % Int8
56
```

Hoisting Invariants

Probably the most frustrating

- Compiler sometimes does it
- Other time you have to do it manually

Key problem is question “always legal?”

- Often depends on alias analysis

Hoisting Example

```
type Bar{T}
    x::Vector{T}
    y::Vector{T}
end

function foo(b,a)
    for i=1:length(q.x)
        b.y[i] += (2a+1)*b.x[i]
    end
end

b = ...instance of Bar{Float32}...
foo(b,3.0f0)
```

Hoisting Loads of Fields

```
type Bar{T}
    x::Vector{T}
    y::Vector{T}
end

function foo(q,a)
    x = q.x
    y = q.y
    for i=1:length(x)
        y[i] += (2a+1)*x[i]
    end
end

b = ...instance of Bar{Float32}...
foo(b,3.0f0)
```

Guidelines for Invariant Hoisting

Don't bother hoisting local scalar stuff

Hoist indirect loads known to be loop-invariant

- That includes fields of composite types

Julia compiler could do better in future

Unroll Loops?

```
# Original loop
function axpy(a,x,y)
    @inbounds for
i=1:length(y)
        y[i] += a*x[i]
    end
end
```

Unrolling this loop causes big slowdown because it thwarts vectorization by LLVM.

```
# Partially unrolled
function axpy(a,x,y)
    n = length(y)
    for i=1:4:n-3
        y[i] += a*x[i]
        y[i+1] += a*x[i+1]
        y[i+2] += a*x[i+2]
        y[i+3] += a*x[i+3]
    end
    # Remainder loop
    for i=n-n%4+1:n
        y[i] += a*x[i]
    end
end
```

Avoid Manual Unrolling

Let JIT do it

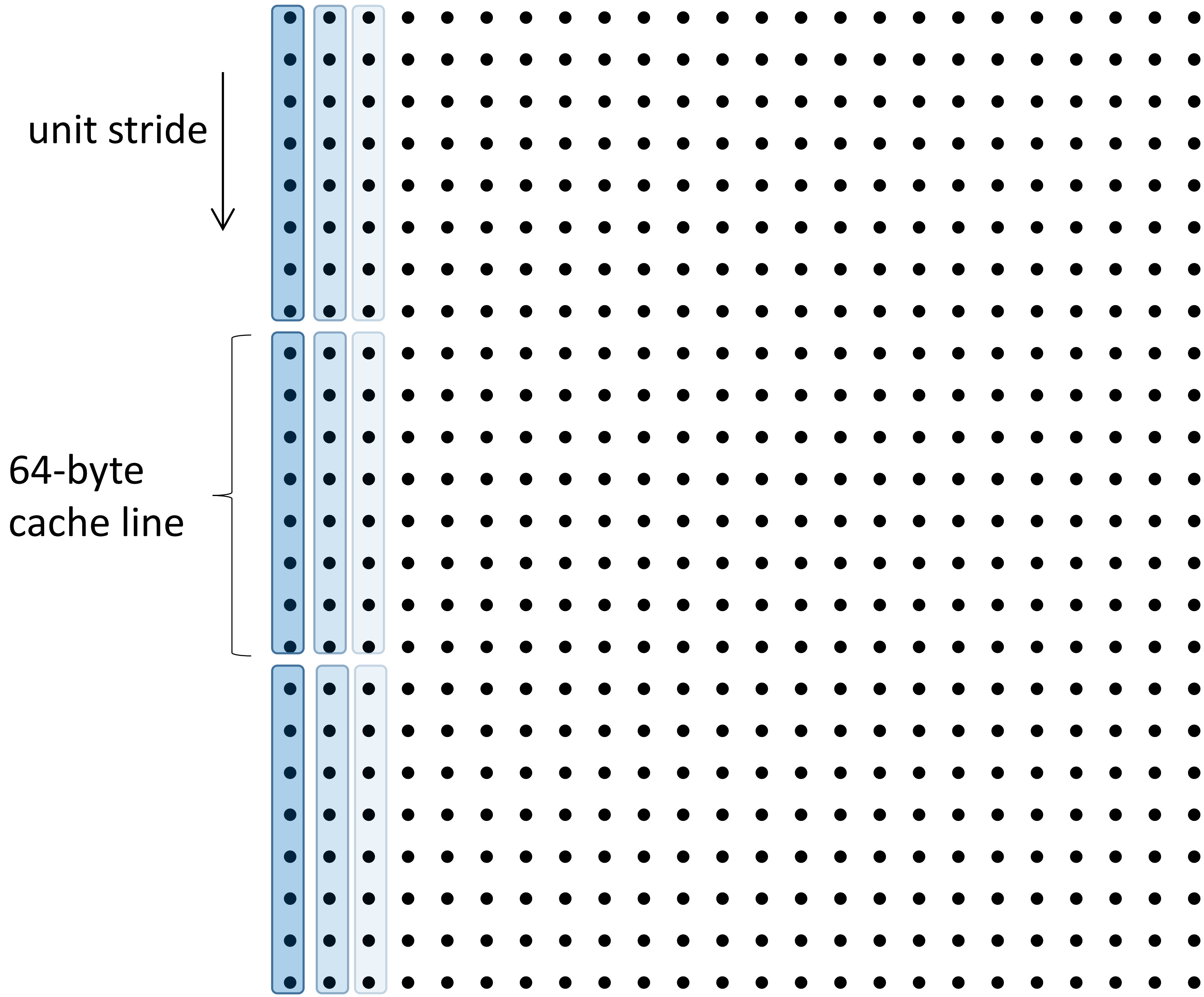
- Optimal unroll factor depends on hardware
  - Instruction latencies
  - Instruction queue size
- Best done *after* some other optimizations happen
- Makes code harder to read

Might occasionally pay off to do manually

- But can backfire badly if it thwarts vectorization

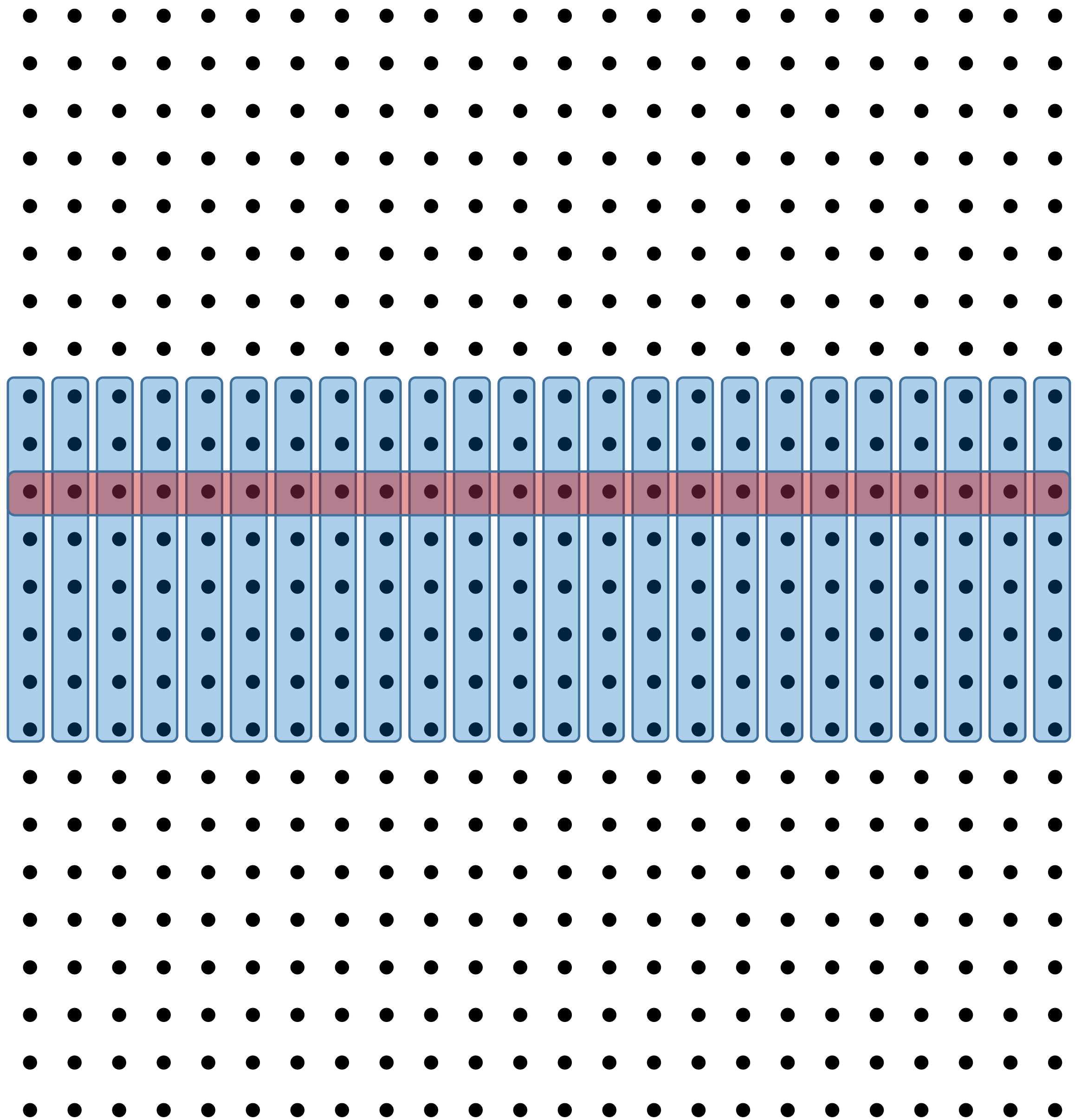
Julia Arrays are Column Major

24x24 array of Float64



Julia Arrays are Column Major

24x24 array of Float64



Example

Slow for big matrices

```
function incmatrix(a)
    (m,n) = size(a)
    for i=1:m, j=1:n
        a[i,j] += 1
    end
end
```

Faster for big matrices

```
function incmatrix(a)
    (m,n) = size(a)
    for j=1:n, i=1:m
        a[i,j] += 1
    end
end
```

Fast and more concise

```
function incmatrix(a)
    for k=eachindex(a)
        a[k] += 1
    end
end
```

Yet Another Cache

Translation Look-aside Buffer (TLB)

- Operates at page level granularity
- Pages = ~4 kB typically.
- System may support “huge” pages too (~2 MB, ~ 1 GB)

Implications

Give thought to how a matrix will be traversed when choosing what a column represents.

Order of a loop nest can make a difference.

Minimize random access

- All else equal, prefer random reads over random writes

Experts sometimes use “blocked algorithms”

- Decompose work into cache-sized pieces of work
- Look up “cache oblivious algorithm”

Exercise 3 and 4

Should be in previous archive from
http://tinyurl.com/HPJ2016

Do these two problems:

- ex3.jl
- ex4.jl

Part 1 of a Solution to Exercise 3

Partial fix: use parametric type ...

```
type Star{T}
    mass :: T # Mass
    pos  :: T # Coordinate
    vel  :: T # Velocity
end
```

... and instantiate with concrete type

```
univ = Star{Float64}[Star(rand(),rand(),rand()) for i=1:100]
```

Part 2 of a Solution to Exercise 3

Replace global variable with local parameter

```
function compute_force(univ)
    n = length(univ)
    force = zeros(n)
    for i=1:n, j=1:n
        force[i] += (univ[i].mass*univ[j].mass)*
                    sign(univ[j].pos-univ[i].pos)
    end
    force
end
```

Likewise for apply_force!

```
function step(m)
    dt = 1/m
    for k=1:m
        force = compute_force(univ)
        apply_force!(force, dt, univ)
    end
end
```

Overhead of dynamic dispatch here is dominated by callee time.

Note on Solution to Exercise 3

Allocating array `force` only once did not help

- Actually slowed down example
- @simd repaired performance

Solution to Exercise 4

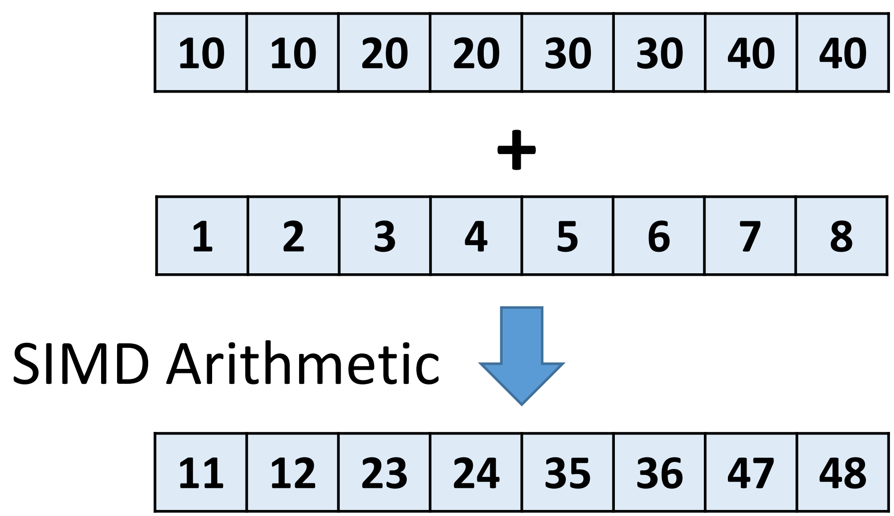
- Swap order of i and j in loop
- Add `@inbounds`

```
for j=2:length(b), i=2:length(a)
@inbounds for i=2:length(a), j=2:length(b)
    match = f[i-1,j-1] + s[a[i],b[j]]
    delete = f[i-1,j] + d
    insert = f[i,j-1] + d
    f[i,j] = max(match, insert, delete)
end
```

Vectorization

Program transformation for exploiting SIMD units

- Not to be confused with other use of “vectorization” to mean array-oriented operations.



Vectorization of a Loop

```
function axpy( a, x, y )
    @simd for i=1:length(x)
        @inbounds y[i] = y[i]+a*x[i]
    end
end
```

```
function axpy( a::Float32, x::Array{Float32,1}, y::Array{Float32,1} )
    @inbounds for i=1:4:length(x)
        # Four logical iterations per physical iteration
        t1 = (x[i],x[i+1],x[i+2],x[i+3])   # Load tuple
        t2 = (y[i],y[i+1],y[i+2],y[i+3])   # Load tuple
        t3 = a*t1                          # Scalar times tuple
        t4 = t2+t3                         # Tuple add
        (y[i],y[i+1],y[i+2],y[i+3]) = t4   # Tuple store
    end
    ... Scalar loop for remaining iterations ...
end
```

Note: example assumes tuple math exists.

Serial Order of Evaluation

$t1_1=x[1]$

$t2_1=y[1]$

$t3_1=a*t1_1$

$t4_1=t2_1+t3_1$

$y[1]=t4_1$

$t1_2=x[2]$

$t2_2=y[2]$

$t3_2=a*t1_2$

$t4_2=t2_2+t3_2$

$y[2]=t4_2$

$t1_3=x[3]$

$t2_3=y[3]$

$t3_3=a*t1_3$

$t4_3=t2_3+t3_3$

$y[3]=t4_3$

$t1_4=x[4]$

$t2_4=y[4]$

$t3_4=a*t1_4$

$t4_4=t2_4+t3_4$

$y[4]=t4_4$

Current @simd Order in Julia

$t1_1$=x[1]	$t1_2$=x[2]	$t1_3$=x[3]	$t1_4$=x[4]
$t2_1$=y[1]	$t2_2$=y[2]	$t2_3$=y[3]	$t2_4$=y[4]
$t3_1$=a*$t1_1$	$t3_2$=a*$t1_2$	$t3_3$=a*$t1_3$	$t3_4$=a*$t1_4$
$t4_1$=$t2_1$+$t3_1$	$t4_2$=$t2_2$+$t3_2$	$t4_3$=$t2_3$+$t3_3$	$t4_4$=$t2_4$+$t3_4$
y[1]=$t4_1$	y[2]=$t4_2$	y[3]=$t4_3$	y[4]=$t4_4$

Future @simd Order in Julia?

$$
\begin{tikzcd}
t1_1=x[1] \arrow[r] \arrow[d] & t1_2=x[2] \arrow[r] \arrow[d] & t1_3=x[3] \arrow[r] \arrow[d] & t1_4=x[4] \arrow[d] \\
t2_1=y[1] \arrow[r] \arrow[d] & t2_2=y[2] \arrow[r] \arrow[d] & t2_3=y[3] \arrow[r] \arrow[d] & t2_4=y[4] \arrow[d] \\
t3_1=a*t1_1 \arrow[r] \arrow[d] & t3_2=a*t1_2 \arrow[r] \arrow[d] & t3_3=a*t1_3 \arrow[r] \arrow[d] & t3_4=a*t1_4 \arrow[d] \\
t4_1=t2_1+t3_1 \arrow[r] \arrow[d] & t4_2=t2_2+t3_2 \arrow[r] \arrow[d] & t4_3=t2_3+t3_3 \arrow[r] \arrow[d] & t4_4=t2_4+t3_4 \arrow[d] \\
y[1]=t4_1 \arrow[r] & y[2]=t4_2 \arrow[r] & y[3]=t4_3 \arrow[r] & y[4]=t4_4
\end{tikzcd}
$$

For now, do not rely on the horizontal orderings.

The Two Key Questions Again

For an instance of a loop, is vectorization:

- **Always** legal?
- **Likely** profitable?

Implicit vs. Explicit Vectorization

Implicit vectorization

Automatic

- Compiler proves that transposition/reassociation is legal

  OR

- Inserts run-time checks

Explicit vectorization with `@simd`

Programmer intervention

- Experimental feature
- Programmer vouches that transposition/reassociation is okay

Example of Run-Time Check

```
function axpy( a::Float32, x::Array{Float32,1}, y::Array{Float32,1} )
    n = length(x)
    if y[1:n] does not overlap x[1:n]
        @inbounds for i=1:4:length(x)
            y[…] += a*x[…]
        end
    end
    … Scalar loop for remaining iterations …
end
```

Limitations of run-time check

- Cost is often quadratic in number of arrays.
- Punts on tricky subscript patterns, such as in sparse matrix code.

  … = w[k[i]]   *# "gather"*

  z[k[i]] = …   *# "scatter"*

Vectorization of Reduction

```
function summation(x)
    s = zero(x[1])
    @simd for i=1:length(x)
        @inbounds s += x[i]
    end
    s
end
```

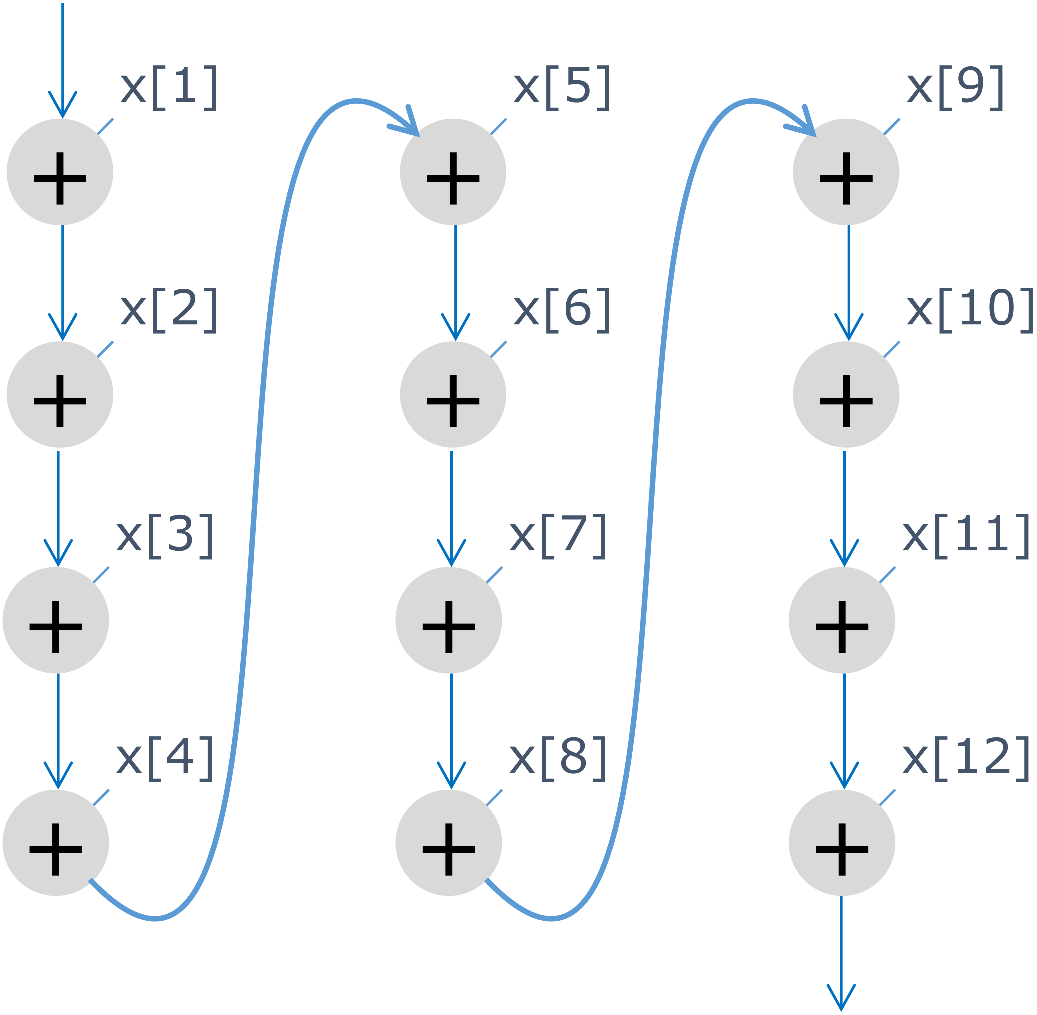
A *reduction variable* is accumulated inside a loop, and otherwise **not** used until loop finishes.

```
function summation(x::Array{Float32,1})
    t =(0f0,0f0,0f0,0f0)
    @inbounds for i=1:4:length(x)
        # Four logical iterations per physical iteration
        t += (x[i],x[i+1],x[i+2],x[i+3])
    end
    s = (t[1]+t[2]) + (t[3]+t[4])
    ... deal with remaining iterations ...
    s
end
```

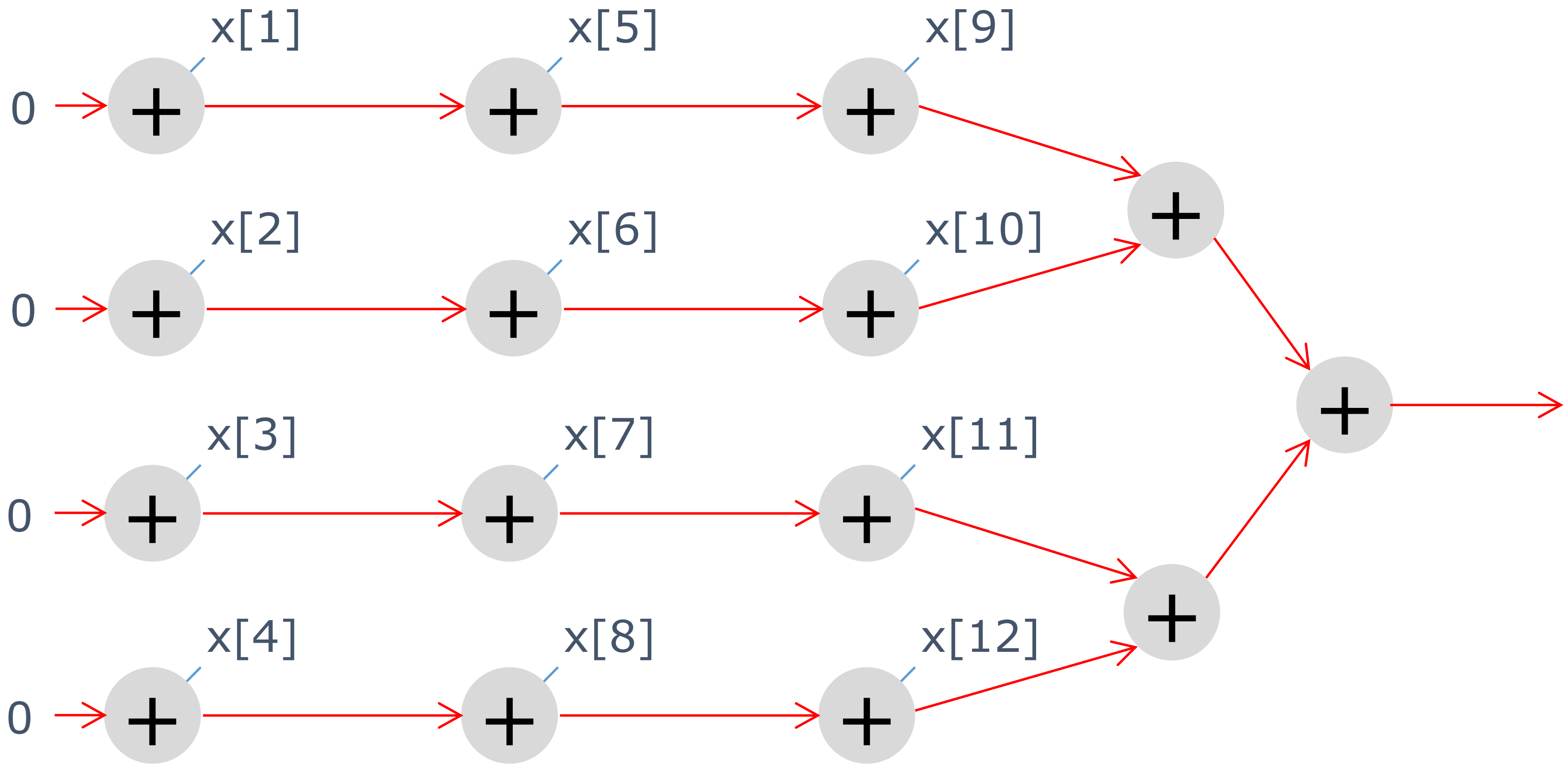
Note: example assumes tuple math exists.

111

Serial Order of Summation



Vectorization Reorders Reduction



Impact of Reassociation Requirement

Implicit vectorization works for

- **integer** reductions
  - `+`, `*`, `&`, `|`, `$`, `min`, `max`
- `@fastmath` **floating-point** reductions
  - `+`, `*`

Use `@simd` for **floating-point** reductions

- `+`, `*`

Not yet implemented in Julia

- floating-point min/max

114

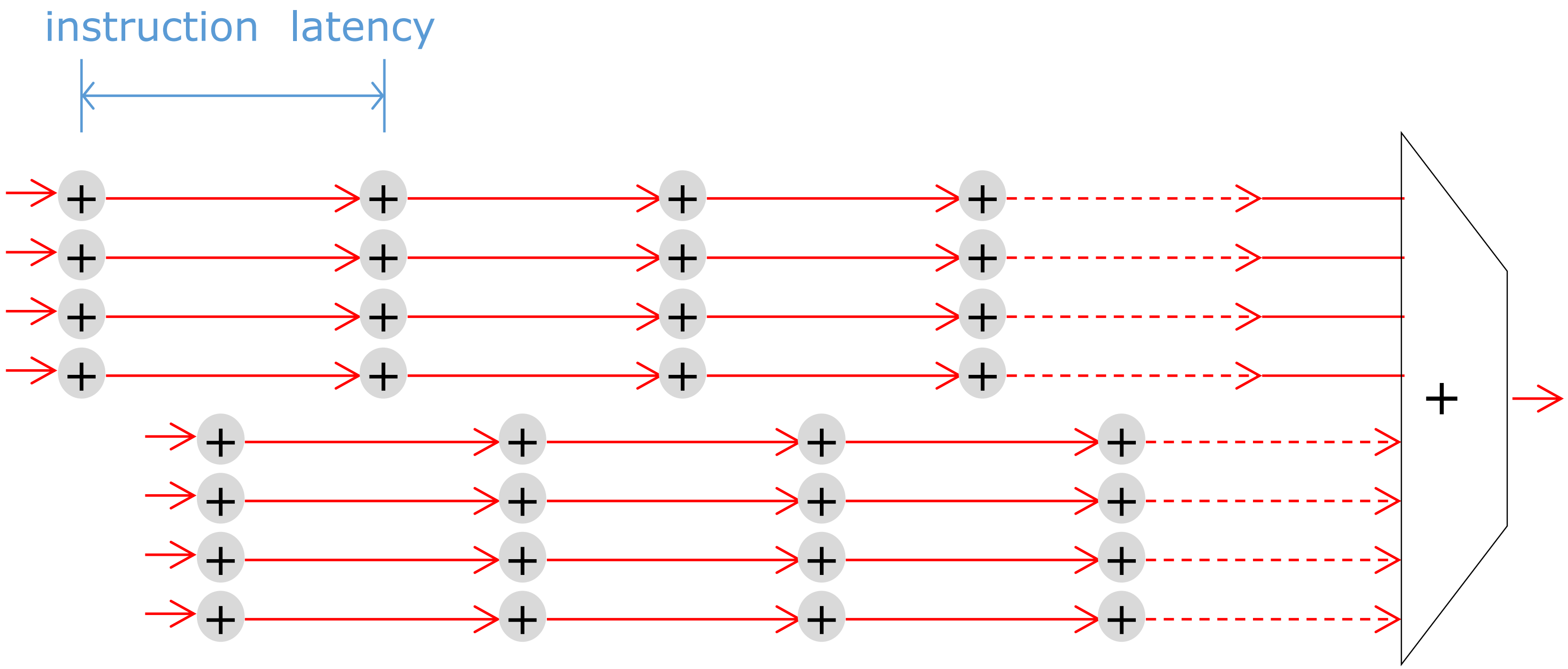
Occasional Speedup Surprise

`@simd` observed to speed up `summation` example by 12x

- On hardware with vector size of 8!

115

Instruction Level Parallelism



Permission to reassociate/commute operations can improve instruction-level parallelism

Vectorization Recommendations

No cross-iteration dependencies

Trip count must be obvious

Loop body must be straight-line code

Subscripts should be unit-stride

No cross-iteration dependencies

Each iteration must not read or write a location written by another iteration

- Except for reduction variables, which must be local scalars
- No iteration waits on another
- An academic issue for now in Julia.

`@simd` spec not same as classic vectorizable loop

- Classic definition allowed limited forms of dependencies
- `@simd` tells LLVM "there are no cross-iteration dependencies"

Trip Count Must Be Obvious

```
@simd for
i=range
    …
end
```

`length(range)` should return integer

`m:n` form of *range* works fine

Loop body should be straight-line code.

```
@simd for i=range
    … no control-flow altering constructs …
end
```

All method calls must be inlined

- Type inference must determine any call targets
- Learn how to write type-stable code

No exception constructs

- Turn off bounds checking (@inbounds)

Short a&&b, a||b, and a?b:c constructs sometimes work

- If LLVM converts it to “select” operation before vectorizer sees it
- Use function ifelse to be sure.

Example with ?: that works

```
function clip( x, a, b, )
    @simd for i=1:length(x)
        @inbounds x[i] = x[i]<a ? a : x[i]>b ? b : x[i]
    end
end

# Shows that code vectorizes for Float32
code_llvm( clip, (Array{Float32,1},Float32,Float32))
```

Skimming code_llvm output

Look for “vector.body” and <*size* x *type*>

```
vector.ph:                              ; preds = %L.preheader

   ...
vector.body:                            ; preds = %vector.body, %vector.ph
   ...
   %wide.load17 = load <8 x float>* %25, align 4
   %26 = fcmp uge <8 x float> %wide.load, %broadcast.splat19
   ...
   %36 = and <8 x i1> %27, %33
   ...
   store <8 x float> %predphi26, <8 x float>* %25, align 4
   ...
   %index.next = add i64 %index, 24
   %38 = icmp eq i64 %index.next, %n.vec
   br i1 %38, label %middle.block, label %vector.body
```

Subscripts should be unit-stride.

```
function stride2( a, b, x, y )
    @simd for i=1:length(y)
        @inbounds y[i] = a * x[2*i] + b
    end
end

code_llvm(stride2,
(Float32,Float32,Array{Float32,1},Array{Float32,1}))
```

Code vectorizes for Float32, but badly

- Ran about 1.37x faster without `@simd` for me
- Stride-2 load synthesized from raft of separate loads

123

2D Arrays Can Work

```
function updateV( irange, jrange, U, Vx, Vy, A )
    for j in jrange
        @simd for i in irange
            @inbounds begin
                Vx[i,j] += (A[i,j+1]+A[i,j])*(U[i,j+1]-U[i,j])
                Vy[i,j] += (A[i+1,j]+A[i,j])*(U[i+1,j]-U[i,j])
            end
        end
    end
end

# Shows that code vectorizes for Float32
R = typeof(1:8)
A = Array{Float32,2}
code_llvm(sweep,(R,R,A,A,A,A,A))
```

In loop nest, put unit-stride loop innermost

Programmer Responsibilities

All vectorization (currently)

- No cross-iteration dependencies
- Straight-line loop body
  - `@inbounds`
  - All calls inlined (be nice to type inference)
- Unit-stride subscripts

Implicit vectorization

- Just a few arrays accessed
- Use `@fastmath` for floating-point reductions

Explicit vectorization

- Use `@simd`
- Ensure there are no cross-iteration dependencies.
- Local scalars for reductions.

Rethinking Algorithms

Cumulative sum rightwards

```
function rsum!(a)
    (m,n) = size(a)
    @inbounds for i=1:m
        s = zero(eltype(a))
        for j=1:n
            s += a[i,j]
            a[i,j] = s;
        end
    end
end
```

Cumulative sum downwards

```
function dsum!(a)
    (m,n) = size(a)
    @inbounds for j=1:n
        s = zero(eltype(a))
        for i=1:m
            s += a[i,j]
            a[i,j] = s
        end
    end
end
```

Why can’t the inner loop be vectorized as written?

Restructuring for SIMD

Cumulative sum rightwards

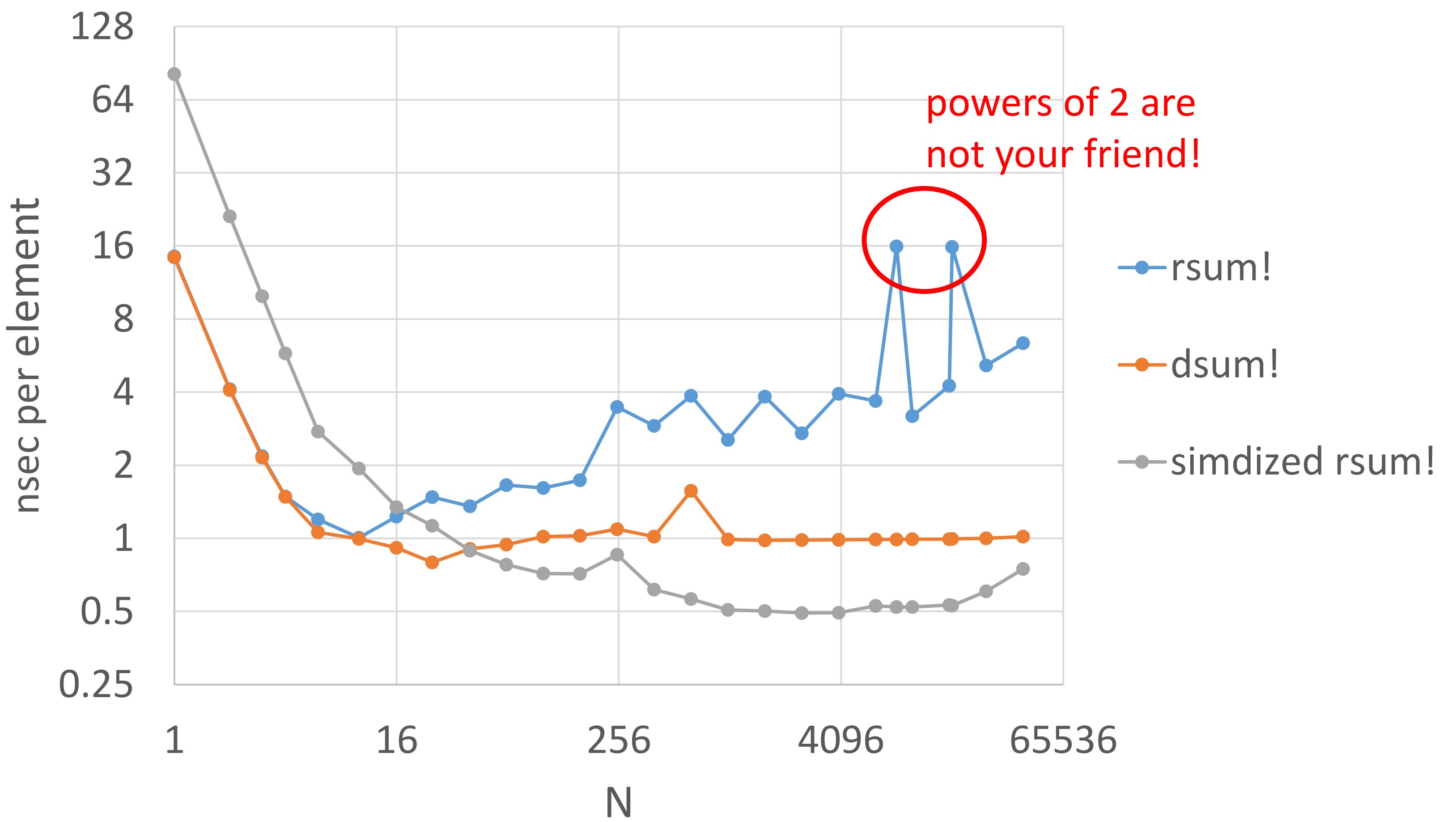
```
function rsum!(a)
  (m,n) = size(a)
  @inbounds for i=1:m
    s = zero(eltype(a))
    for j=1:n
      s += a[i,j]
      a[i,j] = s;
    end
  end
end
```

Cumulative sum rightwards

```
function rsum!(a)
  (m,n) = size(a)
  s = zeros(eltype(a),m)
  @inbounds for j=1:n
    @simd for i=1:m
      s[i] += a[i,j]
      a[i,j] = s[i];
    end
  end
end
```

Additional benefit: lets out-of-order hardware hide latency of +=

Time per Element for Cumulative Sum Functions
NxN array of Float32



Processor: Intel(R) Xeon(R) CPU E5-2630 v3 (20 MB cache) running Julia 0.4.5

Review - Hardware

Memory hierarchy

- Memory bandwidth can be a limiting resource
- Cache lines are the quanta of information interchange (~64 bytes)
- Julia arrays are column major.

Hardware can keep multiple operations in flight

- Sometimes limited by latency, sometimes by throughput

SIMD (Single Instruction Multiple Data)

- Can compute multiple results for the cost of one result
- Requires same operation for all results

Review - Transforms

Transform	Recommended Responsibility
Constant propagation	Compiler
Algebraic simplification	Compiler for integers or `@fastmath` You for other floating-point
Inlining	Compiler usually You can use `@inline` Disable with `–inline=no`
Eliminating bounds checks	Use `@inbounds` Use `–check-bounds=yes` to force checking
Hoisting loop invariants	Compiler for local scalar calculations You for field/subscript references
Unrolling loops	Compiler
Vectorization	Compiler You must use @inbounds You can use `@simd` to assist

Review - Types

Concrete types run much faster

- Avoid boxing and generic dispatch overhead

Pay attention to type inference in compute kernels

- Local variables, fields, return types
- Avoid global variables

Use parametric types, not abstract types, for generality

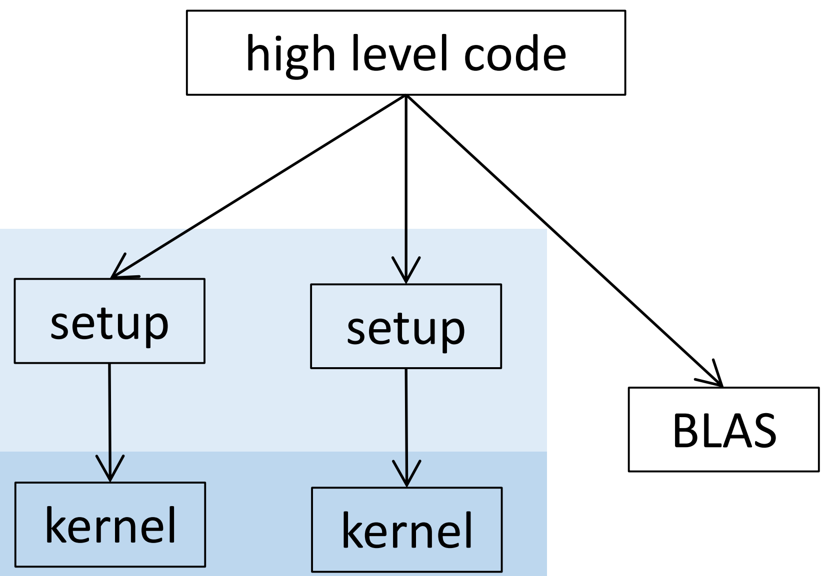
- Use abstract types for overloading or to prevent accidents

Suggested Program Structure

use types to direct control flow and
protect against accidents

do loads/stores for global vars.

inferable concrete types
no global variable references
@simd loops if possible
help compiler



Trademark Notice

Intel is a trademark of Intel Corporation in the U.S. and/or other countries.