Practical Vectorization in Julia

Make those SIMD units (that you bought) work for you.

Arch D. Robison – 6/27/2014

Scope

Julia v0.3.0-prerelease

- Download from http://julialang.org/downloads/

Julia master

- `git clone` [`https://github.com/JuliaLang/julia.git`](https://github.com/JuliaLang/julia.git)
- `Do not configure with USE_SYSTEM_LLVM=1`

Julia 0.2 has no vectorization.

Outline

SIMD hardware

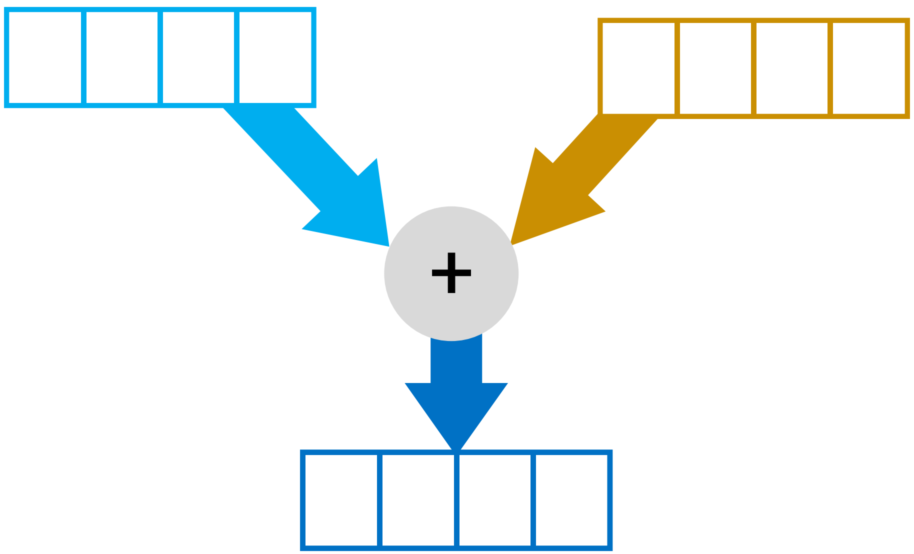
Vectorization basics

Recommendations on vectorization in Julia

Future directions

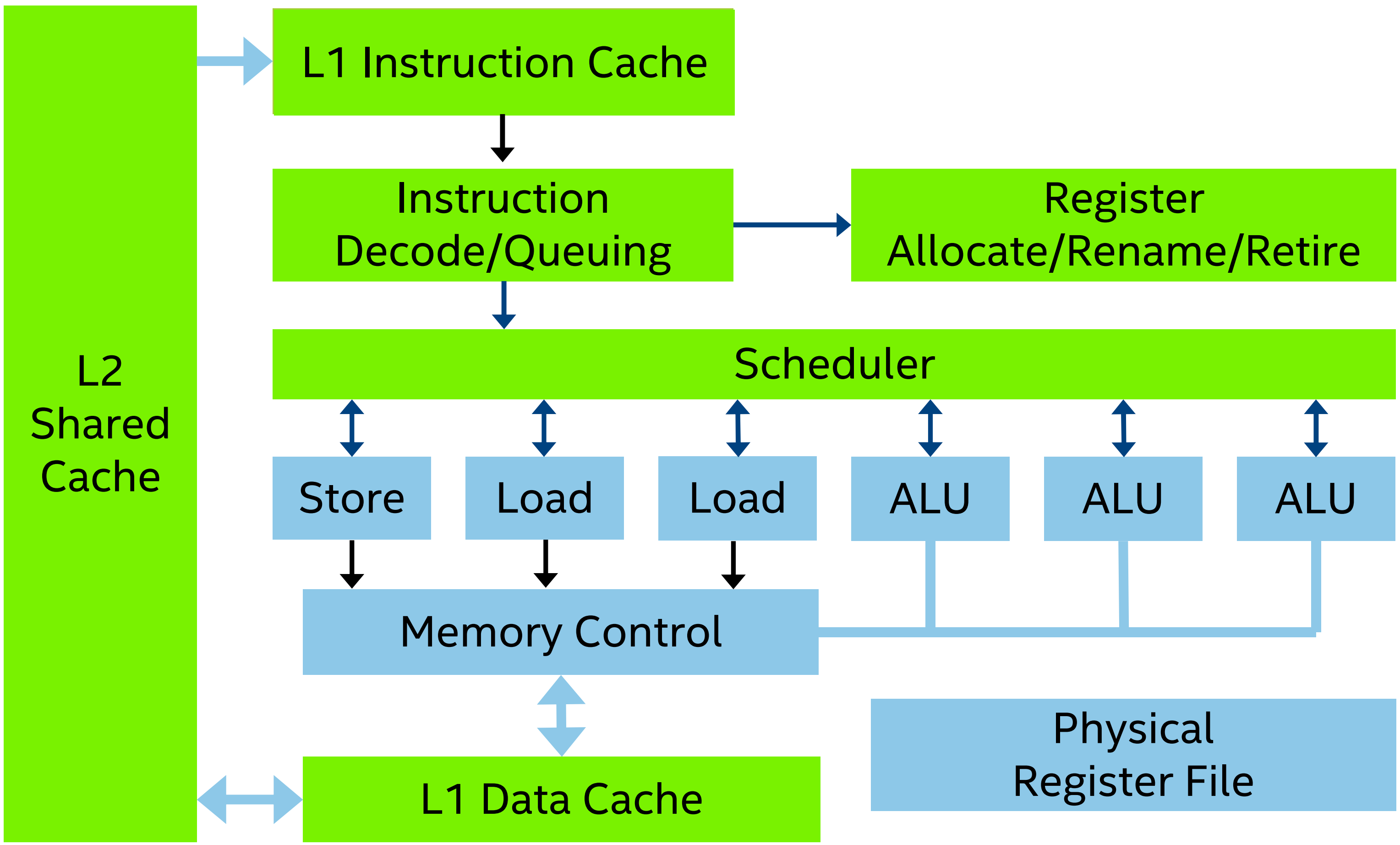
What is SIMD?

Single **I**nstruction **M**ultiple **D**ata



Single instruction operates across a tuple.

Why Hardware Designers Like SIMD



Optimization Notice

Vectorization

Program transformation for exploiting SIMD units

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
```

Requires Julia 0.3-dev or later

```
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

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

Vectorization Transposes the Order

$t1_1$=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*$t1_3$

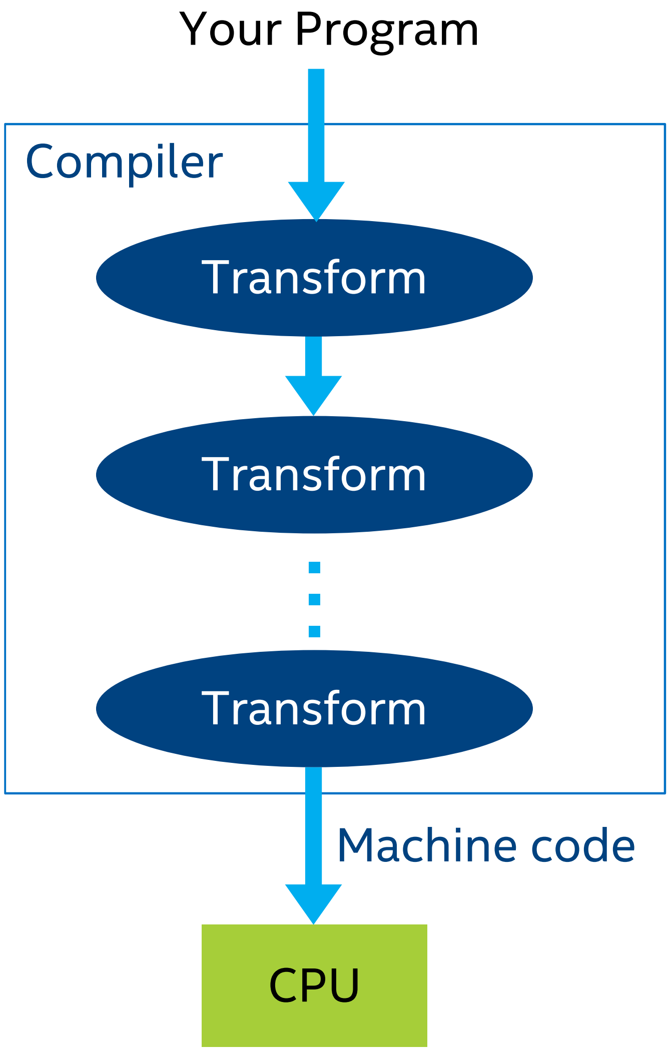
$t4_1$=$t2_1$+$t3_1$ $t4_2$=$t2_2$+$t3_2$ $t4_3$=$t2_3$+$t3_3$ $t4_4$=$t2_3$+$t3_3$

y[1] =$t4_1$ y[2] =$t4_2$ y[3] =$t4_3$ y[4] =$t4_4$

Vectorization transposes each chunk of iteration space.

Optimization Notice

Compilers 101



Three steps for a transform

1. Is it **legal**?
2. Is it **profitable**?
3. If so, **do** the transform.

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 swears 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"*

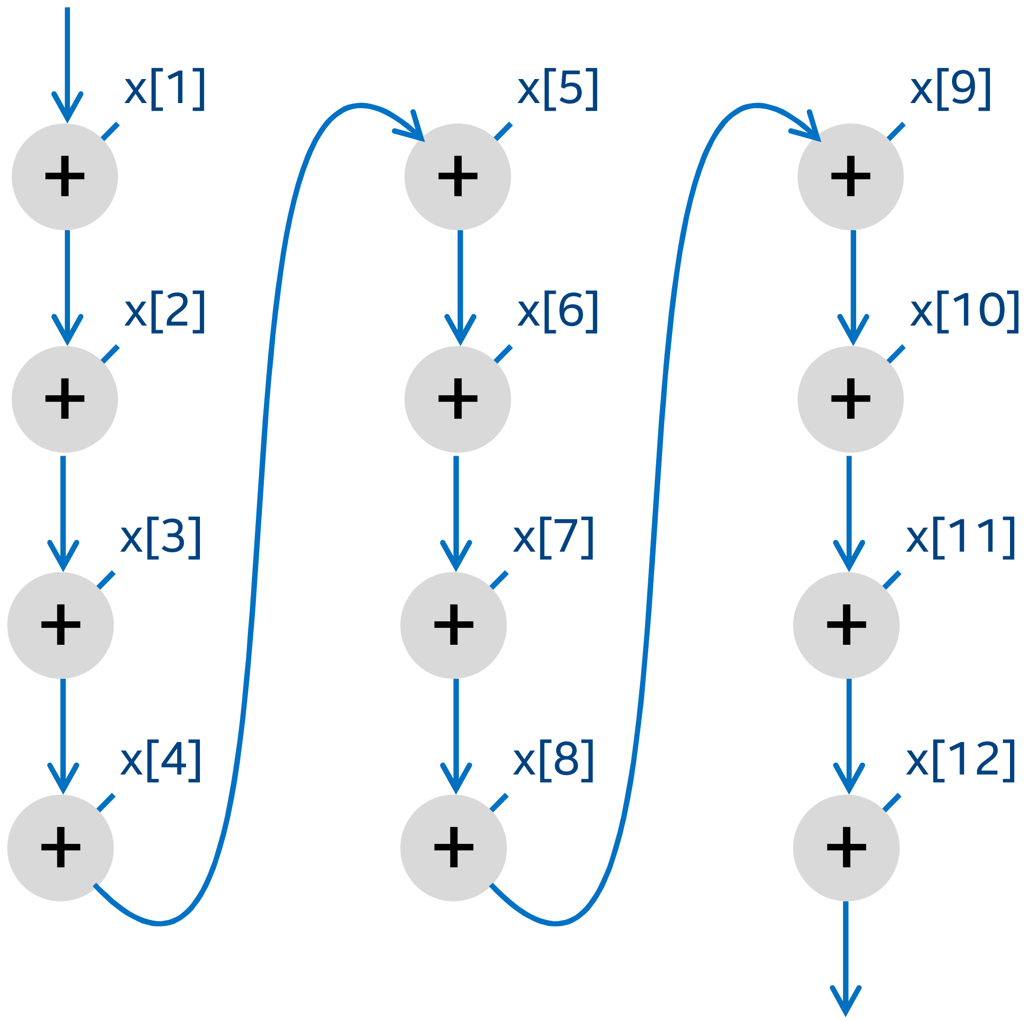
Vectorization of Reduction

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

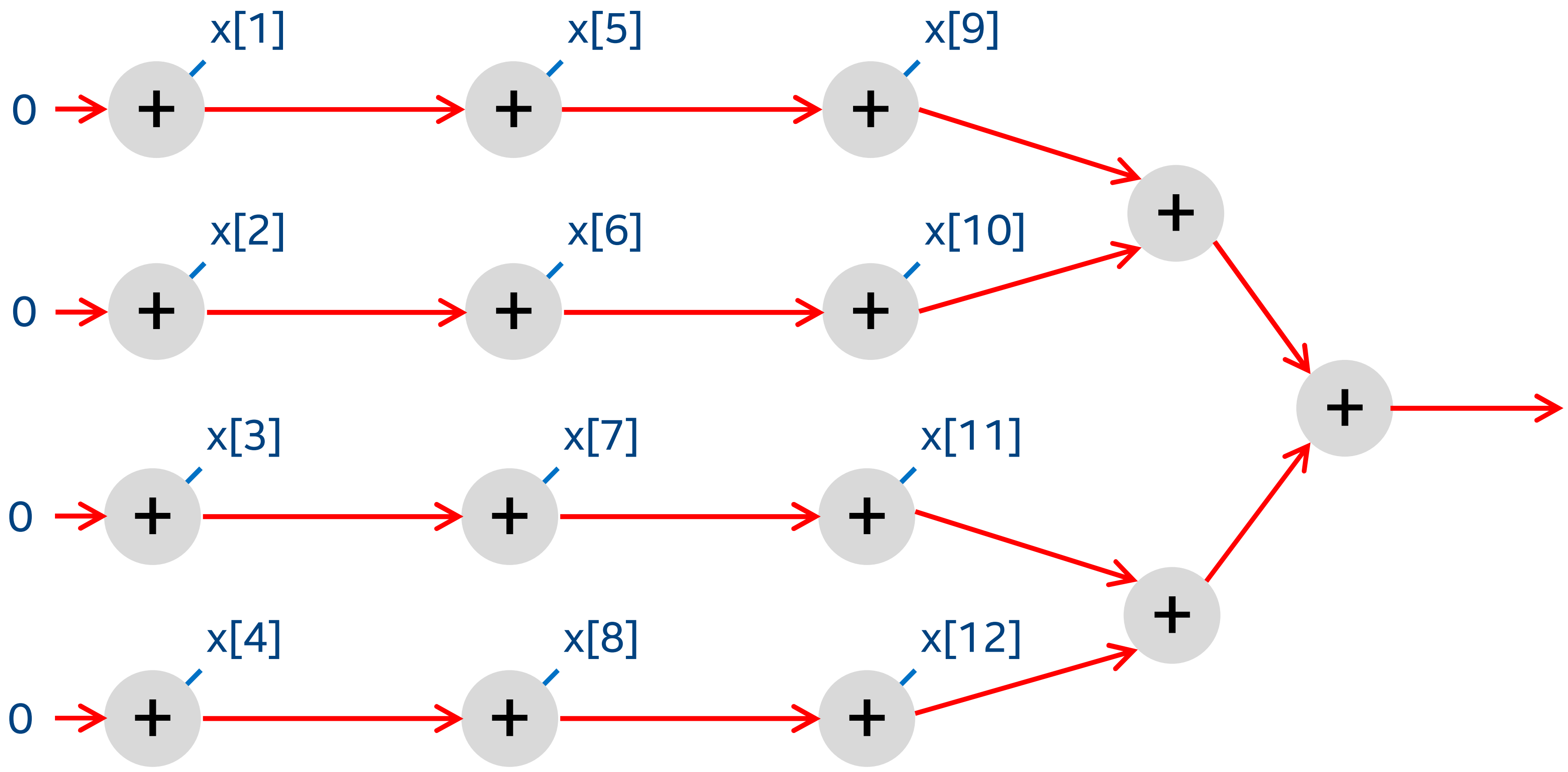
```
function summation(x::Array{Float32,1})
    t =(0f0,0f0,0f0,0f0)
    @inbounds for i=1:4:length(x)
        # Four logical iterations per physical iteration
        t += (x[i],x[i+1],x[i+2],x[i+3])
    end
    s = (t[1]+t[2]) + (t[3]+t[4])
    ... deal with remaining iterations ...
    s
end
```

Note: example assumes tuple math exists.

Serial Order of Summation



Vectorization Reorders Reduction



Impact of Reassociation Requirement

Implicit vectorization works for **integer** reductions

- +, *, &, |, $, min, max

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

- +, *

Not yet implemented

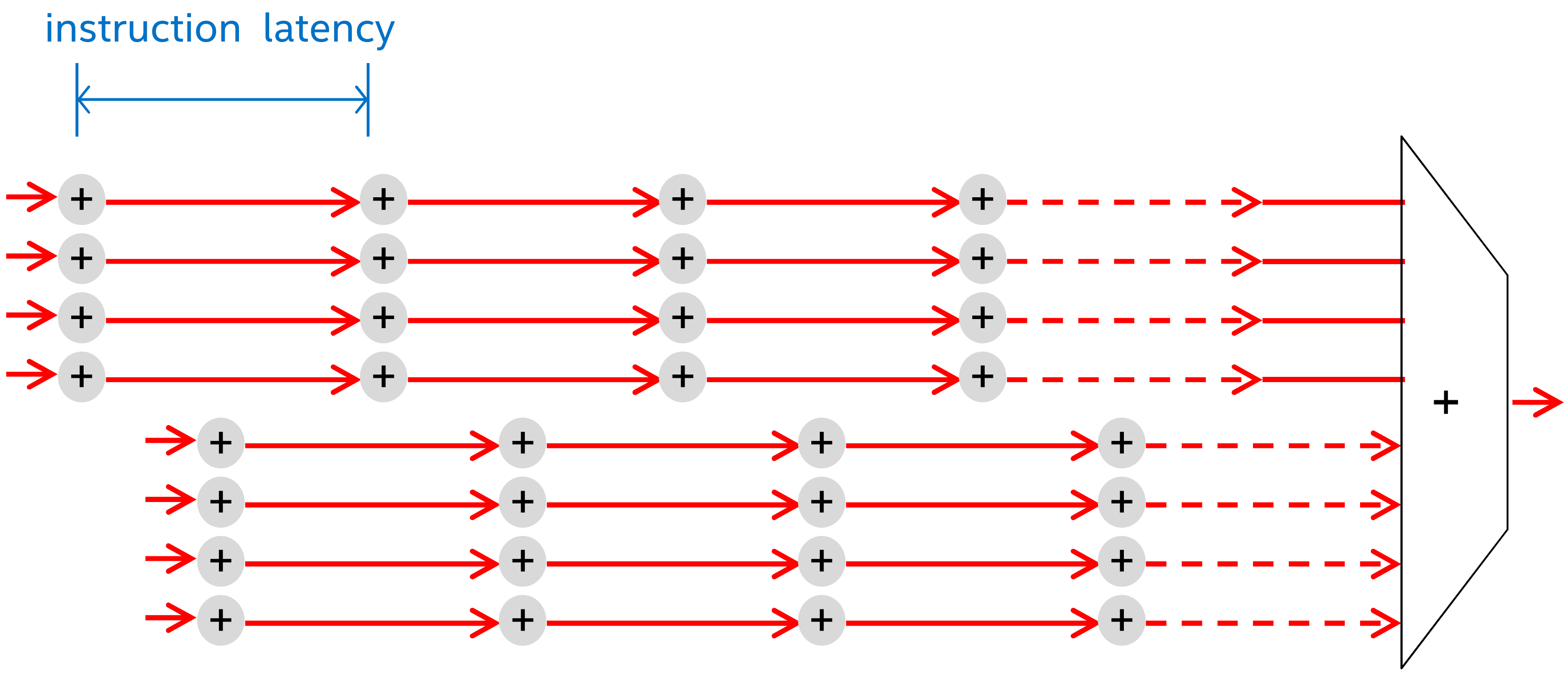
- floating-point min/max

Occasional Speedup Surprise

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

- On hardware with vector size of 8!

Instruction Level Parallelism



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

Vectorization Recommendations (Julia with LLVM 3.3)

No cross-iteration dependencies

Trip count must be obvious

Loop body must be straight-line code

Subscripts should be unit-stride

Works best with Float32

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

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

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 (learn to write type-stable code)
- Unit-stride subscripts
- Float32 works best

Implicit vectorization

- Just a few arrays accessed
- No floating-point reductions

Explicit vectorization

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

Future: LLVM 3.5

Vectorizer still limited to single basic block

- But often generates better code

Enables Intel® Advanced Vector Extensions 2 (Intel® AVX2)

- Fused multiply-add
  - Issue: requires “unsafe algebra” to enable.

Future Possibilities

SLPVectorizer (PR#6271)

- Vectorizes tuple math
- Slows down compilation

Vectorize loops with bounds checks

- Exploit reordering permissiveness to vectorize or hoist bounds checks

Vectorize loop bodies that are not straight-line code

- C/C++/Fortran compilers do this.
- SIMD extensions with masking (e.g. Intel® AVX-512) make this worthwhile
- Requires major LLVM hacking

Allow forward lexical dependencies

Diagnostics for why loop does not vectorize

Summary

Vectorization is optimization that speeds up *some* routines

- **Transposes** evaluation order

Implicit vectorization happens sometimes.

Explicit vectorization requires `@simd`

Current limitations that might be removed in future:

- Straight-line loop body
- Requires **`@inbounds`**
- Unit stride accesses to arrays
- Works best with Float32

There's much room for future improvement

Legal Disclaimer & Optimization Notice

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

intel®