

# JULIA: FROM MULTICORE TO GPU PARALLELISM

presented by Steffen Haug

# WHAT IS JULIA?

(Briefly)

1. High level programming language



1. High level programming language
2. Dynamically typed



1. High level programming language
2. Dynamically typed
3. For HPC



1. High level programming language
2. Dynamically typed
3. For HPC
4. **Novel compilation infrastructure**  
(Central on CPU and GPU)

Untyped AST specialized at runtime



1. High level programming language
2. Dynamically typed
3. For HPC
4. **Novel compilation infrastructure**  
(Central on CPU and GPU)

Untyped AST specialized at runtime

*“Looks like Python – runs like Fortran”*



# EXAMPLE

```
function square!(xs)
  for i in eachindex(xs)
    xs[i]  $\hat{=}$  2
  end
end

V = rand(10000);
@code_native syntax=:intel square(V)
```

Function that squares (mutates) a vector.



# EXAMPLE

```
function square!(xs)
  for i in eachindex(xs)
    xs[i]  $\hat{=}$  2
  end
end

V = rand(10000);
@code_native syntax=:intel square(V)
```

Not a type in sight!

# EXAMPLE

```
function square!(xs)
  for i in eachindex(xs)
    xs[i] ^= 2
  end
end

V = rand(10000);
@code_native syntax=:intel square(V)
```

When invoked with `V`, `square!` is specialized to `Vector{Float64}`.

# EXAMPLE

```
function square!(xs)
  for i in eachindex(xs)
    xs[i]  $\hat{=}$  2
  end
end

V = rand(10000);
@code_native syntax=:intel square(V)
```

`@code_native` is a “macro” that allows us to inspect code generated for a particular specialization.

# EXAMPLE

```
function square!(xs)
  for i in eachindex(xs)
    xs[i]  $\hat{=}$  2
  end
end

V = rand(10000);
@code_native syntax=:intel square(V)
```

**Simple function** – surely the assembly code is reasonably simple?





?

```

square:
    mov rax, qword ptr [rdi + 8]
    test rax, rax
    je .LBB0_7
    mov rcx, qword ptr [rdi]
    mov edx, 1
    cmp rax, 16
    jb .LBB0_5
    mov rsi, rax
    and rsi, -16
    lea rdx, [rsi + 1]
    xor edi, edi
.LBB0_3:
    vmovups xmm0, xmmword ptr [rcx + 8*rdi]
    vmovups xmm1, xmmword ptr [rcx + 8*rdi + 32]
    vmovups xmm2, xmmword ptr [rcx + 8*rdi + 64]
    vmovups xmm3, xmmword ptr [rcx + 8*rdi + 96]
    vinsertf128 ymm0, xmm0, xmmword ptr [rcx + 8*rdi + 16], 1
    vinsertf128 ymm1, xmm1, xmmword ptr [rcx + 8*rdi + 48], 1
    vinsertf128 ymm2, xmm2, xmmword ptr [rcx + 8*rdi + 80], 1
    vinsertf128 ymm3, xmm3, xmmword ptr [rcx + 8*rdi + 112], 1
    vmslpd ymm0, ymm0, ymm0
    vmslpd ymm1, ymm1, ymm1
    vmslpd ymm2, ymm2, ymm2
    vmslpd ymm3, ymm3, ymm3
    vextractf128 xmmword ptr [rcx + 8*rdi + 16], ymm0, 1
    vmovupd xmmword ptr [rcx + 8*rdi], xmm0
    vextractf128 xmmword ptr [rcx + 8*rdi + 48], ymm1, 1
    vmovupd xmmword ptr [rcx + 8*rdi + 32], xmm1
    vextractf128 xmmword ptr [rcx + 8*rdi + 80], ymm2, 1
    vmovupd xmmword ptr [rcx + 8*rdi + 64], xmm2
    vextractf128 xmmword ptr [rcx + 8*rdi + 112], ymm3, 1
    vmovupd xmmword ptr [rcx + 8*rdi + 96], xmm3
    add rdi, 16
    cmp rsi, rdi
    jne .LBB0_3
    cmp rax, rsi
    je .LBB0_7
.LBB0_5:
    dec rdx
.LBB0_6:
    vmovsd xmm0, qword ptr [rcx + 8*rdx]
    vmsltd xmm0, xmm0, xmm0
    vmovsd qword ptr [rcx + 8*rdx], xmm0
    inc rdx
    cmp rdx, rax
    jb .LBB0_6
.LBB0_7:
    vzeroupper
    ret

```



Remove some  
comments...

```

.LBB0_3:
    vmovups  xmm0, xmmword ptr [rcx + 8*rdi]
    vmovups  xmm1, xmmword ptr [rcx + 8*rdi + 32]
    vmovups  xmm2, xmmword ptr [rcx + 8*rdi + 64]
    vmovups  xmm3, xmmword ptr [rcx + 8*rdi + 96]
    vinsertf128 ymm0, ymm0, xmmword ptr [rcx + 8*rdi + 16], 1
    vinsertf128 ymm1, ymm1, xmmword ptr [rcx + 8*rdi + 48], 1
    vinsertf128 ymm2, ymm2, xmmword ptr [rcx + 8*rdi + 80], 1
    vinsertf128 ymm3, ymm3, xmmword ptr [rcx + 8*rdi + 112], 1
    vmulpd   ymm0, ymm0, ymm0
    vmulpd   ymm1, ymm1, ymm1
    vmulpd   ymm2, ymm2, ymm2
    vmulpd   ymm3, ymm3, ymm3
    vextractf128 xmmword ptr [rcx + 8*rdi + 16], ymm0, 1
    vmovupd  xmmword ptr [rcx + 8*rdi], xmm0
    vextractf128 xmmword ptr [rcx + 8*rdi + 48], ymm1, 1
    vmovupd  xmmword ptr [rcx + 8*rdi + 32], xmm1
    vextractf128 xmmword ptr [rcx + 8*rdi + 80], ymm2, 1
    vmovupd  xmmword ptr [rcx + 8*rdi + 64], xmm2
    vextractf128 xmmword ptr [rcx + 8*rdi + 112], ymm3, 1
    vmovupd  xmmword ptr [rcx + 8*rdi + 96], xmm3
    add     rdi, 16
    cmp     rsi, rdi
    jne     .LBB0_3

```

Julia has **unrolled the loop** and used **SIMD-instructions**!

```

void square(double *xs, int N) {
    for (int i = 0; i < N; i++) {
        xs[i] *= xs[i];
    }
}

```

```

.L1:
    vmovupd ymm0, YMMWORD PTR [rdi+rsi*1]
    vmovupd ymm1, YMMWORD PTR [rdi+rsi*1+0x20]
    vmovupd ymm2, YMMWORD PTR [rdi+rsi*1+0x40]
    vmovupd ymm3, YMMWORD PTR [rdi+rsi*1+0x60]
    vmulpd ymm0, ymm0, ymm0
    vmulpd ymm1, ymm1, ymm1
    vmulpd ymm2, ymm2, ymm2
    vmulpd ymm3, ymm3, ymm3
    vmovupd YMMWORD PTR [rdi+rsi*1], ymm0
    vmovupd YMMWORD PTR [rdi+rsi*1+0x20], ymm1
    vmovupd YMMWORD PTR [rdi+rsi*1+0x40], ymm2
    vmovupd YMMWORD PTR [rdi+rsi*1+0x60], ymm3
    sub     rsi, 0xffffffffffffffff80
    cmp     rdx, rsi
    jne     .L1

```

This is what Clang does.  
*(Slightly* better due to Julia-arrays being strided)



# JULIA USES LLVM



Julia can do the same optimizations we expect in C/C++

# JULIA USES LLVM



Julia can do the same optimizations we expect in C/C++

*...because it is literally the same optimizer!*

## BUT THERE ARE SOME HUGE CAVEATS

1. There is some *slight* compilation overhead at startup

However: As problem size grows, relative compilation time decreases.

## BUT THERE ARE SOME HUGE CAVEATS

1. There is some *slight* compilation overhead at startup
2. Julia is *garbage-collected* 🤔

# BUT THERE ARE SOME HUGE CAVEATS

1. There is some *slight* compilation overhead at startup
2. Julia is *garbage-collected* 🤖
3. We have no explicit control over memory  
(Allocation, copying, ...)

Accidental heap operations can really hurt!

# BUT THERE ARE SOME HUGE CAVEATS

1. There is some *slight* compilation overhead at startup
2. Julia is *garbage-collected* 🤖
3. We have no explicit control over memory  
(Allocation, copying, ...)
4. *The whole thing breaks* if Julia can't figure out the types

There are many ways to shoot yourself in the foot! 😊

```

Δx = 0.1

function riemann!(X, x)
    N = length(x)
    X[0] = 0.0
    for i in 2:N
        X[i] = X[i - 1] + Δx * X[i]
    end
end

v = rand(100000)
V = zeros(100000)
@time riemann!(V, v)

```

```

const Δx = 0.1

function riemann!(X, x)
    N = length(x)
    X[0] = 0.0
    for i in 2:N
        X[i] = X[i - 1] + Δx * X[i]
    end
end

v = rand(100000)
V = zeros(100000)
@time riemann!(V, v)

```

Two seemingly quite similar programs...

```

Δx = 0.1

function riemann!(X, x)
    N = length(x)
    X[0] = 0.0
    for i in 2:N
        X[i] = X[i - 1] + Δx * X[i]
    end
end

v = rand(100000)
V = zeros(100000)
@time riemann!(V, v)

```

0.010550 seconds  
(499.49 k allocations)

```

const Δx = 0.1

function riemann!(X, x)
    N = length(x)
    X[0] = 0.0
    for i in 2:N
        X[i] = X[i - 1] + Δx * X[i]
    end
end

v = rand(100000)
V = zeros(100000)
@time riemann!(V, v)

```

0.000175 seconds

One is  $\sim 60\times$  faster than the other!



```

Δx = 0.1

function riemann!(X, x)
    N = length(x)
    X[0] = 0.0
    for i in 2:N
        X[i] = X[i - 1] + Δx * x[i]
    end
end

v = rand(100000)
V = zeros(100000)
@time riemann!(V, v)

```

0.010550 seconds  
(499.49 k allocations)

```

const Δx = 0.1

function riemann!(X, x)
    N = length(x)
    X[0] = 0.0
    for i in 2:N
        X[i] = X[i - 1] + Δx * x[i]
    end
end

v = rand(100000)
V = zeros(100000)
@time riemann!(V, v)

```

0.000175 seconds

And 500K allocations? 🤖

```

Δx = 0.1

function riemann!(X, x)
    N = length(x)
    X[0] = 0.0
    for i in 2:N
        X[i] = X[i - 1] + Δx * x[i]
    end
end

v = rand(100000)
V = zeros(100000)
@time riemann!(V, v)

```

0.010550 seconds  
(499.49 k allocations)

```

const Δx = 0.1

function riemann!(X, x)
    N = length(x)
    X[0] = 0.0
    for i in 2:N
        X[i] = X[i - 1] + Δx * x[i]
    end
end

v = rand(100000)
V = zeros(100000)
@time riemann!(V, v)

```

0.000175 seconds

And 500K allocations? 🤖

$\Delta x$  could be reassigned  $\implies$  Julia can't assume its type!

```

Δx = 0.1

function riemann!(X, x)
    N = length(x)
    X[0] = 0.0
    for i in 2:N
        X[i] = X[i - 1] + Δx * x[i]
    end
end

v = rand(100000)
V = zeros(100000)
@time riemann!(V, v)

```

0.010550 seconds  
(499.49 k allocations)

```

const Δx = 0.1

function riemann!(X, x)
    N = length(x)
    X[0] = 0.0
    for i in 2:N
        X[i] = X[i - 1] + Δx * x[i]
    end
end

v = rand(100000)
V = zeros(100000)
@time riemann!(V, v)

```

0.000175 seconds

And 500K allocations? 🤖

Leads to runtime type checks and dynamic dispatching...

```

Δx = 0.1

function riemann!(X, x)
    N = length(x)
    X[0] = 0.0
    for i in 2:N
        X[i] = X[i - 1] + Δx * x[i]
    end
end

v = rand(100000)
V = zeros(100000)
@time riemann!(V, v)

```

0.010550 seconds  
(499.49 k allocations)

```

const Δx = 0.1

function riemann!(X, x)
    N = length(x)
    X[0] = 0.0
    for i in 2:N
        X[i] = X[i - 1] + Δx * x[i]
    end
end

v = rand(100000)
V = zeros(100000)
@time riemann!(V, v)

```

0.000175 seconds

And 500K allocations? 🤖

...and forces intermediate calculations onto the heap!

```

Δx = 0.1

function riemann!(X, x)
    N = length(x)
    X[0] = 0.0
    for i in 2:N
        X[i] = X[i - 1] + Δx * x[i]
    end
end

v = rand(100000)
V = zeros(100000)
@time riemann!(V, v)

```

0.010550 seconds  
(499.49 k allocations)

```

const Δx = 0.1

function riemann!(X, x)
    N = length(x)
    X[0] = 0.0
    for i in 2:N
        X[i] = X[i - 1] + Δx * x[i]
    end
end

v = rand(100000)
V = zeros(100000)
@time riemann!(V, v)

```

0.000175 seconds

And 500K allocations? 🤪

...and forces intermediate calculations onto the heap!



Julia *can* generate really fast code...

Julia *can* generate really fast code...

But the *lack of explicit control* requires a lot of attention.

Julia *can* generate really fast code...

But the *lack of explicit control* requires a lot of attention.

Even more so when we *parallelize* our programs.



# MULTICORE

```
using .Threads: @threads
```

```
for i in eachindex(xs)  
end
```



```
@threads for i in eachindex(xs)  
end
```

Similar API to OpenMP: *Just annotate loops with a macro.*

```
for i in eachindex(xs)
end
```



```
@threads for i in eachindex(xs)
end
```

Similar API to OpenMP: *Just annotate loops with a macro.*

#### THE NORMAL CAVEATS APPLY

1. Spawning threads involve some overhead

```
for i in eachindex(xs)
end
```



```
@threads for i in eachindex(xs)
end
```

Similar API to OpenMP: *Just annotate loops with a macro.*

#### THE NORMAL CAVEATS APPLY

1. Spawning threads involve some overhead
2. Memory is shared (Race conditions, deadlocks)

```
for i in eachindex(xs)
end
```



```
@threads for i in eachindex(xs)
end
```

Similar API to OpenMP: *Just annotate loops with a macro.*

#### THE NORMAL CAVEATS APPLY

1. Spawning threads involve some overhead
2. Memory is shared (Race conditions, deadlocks)
3. Possible speedup severely limited by core count

```
for i in eachindex(xs)
end
```



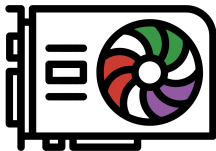
```
@threads for i in eachindex(xs)
end
```

Similar API to OpenMP: *Just annotate loops with a macro.*

#### THE NORMAL CAVEATS APPLY

1. Spawning threads involve some overhead
2. Memory is shared (Race conditions, deadlocks)
3. Possible speedup severely limited by core count
4. Domain decomposition sometimes necessary to get speedup on simple problems

# GPU PARALLELISM



# LEVEL 1: “VECTORIZED” ARRAY API

(Vectorized in the sense that you only operate on arrays, Numpy style)



# “VECTORIZED” ARRAY API

```
# In-place operations.
```

```
v = CUDA.rand(10000000)
```

```
v *= 10.0
```

```
v ^= 2
```

```
v *= sqrt.(v)
```

```
# Parallel scan.
```

```
w = CUDA.zeros(100000000)
```

```
accumulate!(max, w, v)
```

`v`, `w` of type `CuArray` that manages a device-side buffer.

# “VECTORIZED” ARRAY API

```
# In-place operations.  
v = CUDA.rand(10000000)  
v *= 10.0  
v ^= 2  
v *= sqrt.(v)  
  
# Parallel scan.  
w = CUDA.zeros(100000000)  
accumulate!(max, w, v)
```

Block- and gridsizes automatically determined.

# “VECTORIZED” ARRAY API

```
# FFT (10M-point FFT takes 200ms on 980)
```

```
using CUDA.CUFFT
```

```
 $\omega$  = fft(v)
```

```
# Kernels get fused
```

```
w *= w.^2 .+ v
```

```
# Check out what happens on the GPU
```

```
@device_code_sass w *= w.^2 .+ v
```

Bindings to CUDA libraries like cuFFT and cuBLAS  
(Also cuSPARSE, cuSOLVER, ...)

# “VECTORIZED” ARRAY API

```
# FFT (10M-point FFT takes 200ms on 980)  
using CUDA.CUFFT  
 $\omega$  = fft(v)  
  
# Kernels get fused  
w .= w.^2 .+ v  
  
# Check out what happens on the GPU  
@device_code_sass w .= w.^2 .+ v
```

GPU code generated on the fly using LLVMs PTX/SASS backend...  
(Same sort of infrastructure as specializing Julia functions)

# “VECTORIZED” ARRAY API

```
# FFT (10M-point FFT takes 200ms on 980)
```

```
using CUDA.CUFFT
```

```
 $\omega$  = fft(v)
```

```
# Kernels get fused
```

```
w  $\cdot$ = w.^2 .+ v
```

```
# Check out what happens on the GPU
```

```
@device_code_sass w  $\cdot$ = w.^2 .+ v
```

Enables kernel fusion and global optimization (!)

# “VECTORIZED” ARRAY API

```
# FFT (10M-point FFT takes 200ms on 980)
```

```
using CUDA.CUFFT
```

```
 $\omega$  = fft(v)
```

```
# Kernels get fused
```

```
w *= w.^2 .+ v
```

```
# Check out what happens on the GPU
```

```
@device_code_sass w *= w.^2 .+ v
```

Like with CPU code, you can inspect the generated GPU code.  
(SASS/PTX code is too long/ugly to put on a slide)

## LEVEL 2: JULIA ON THE GPU

# JULIA ON THE GPU

```
function cusquare!(xs)
    i = (blockIdx().x - 1) * blockDim().x + threadIdx().x
    if i < length(xs)
        xs[i] *= xs[i]
    end
    return nothing
end

Nt = 1024
Nb = ceil{Int, length(v) / Nt}
@cuda threads=Nt blocks=Nb cusquare!(v)
```

**cusquare!** is defined *as a regular Julia function!*

(It won't run if you call it; block- and thread-ID is not defined on the CPU)



# JULIA ON THE GPU

```
function cusquare!(xs)
    i = (blockIdx().x - 1) * blockDim().x + threadIdx().x
    if i < length(xs)
        xs[i] *= xs[i]
    end
    return nothing
end

Nt = 1024
Nb = ceil{Int}(length(v) / Nt)
@cuda threads=Nt blocks=Nb cusquare!(v)
```

The `@cuda` macro transforms typed [Julia IR](#) into [GPU code](#)!



# JULIA ON THE GPU

```
function cusquare!(xs)
    i = (blockIdx().x - 1) * blockDim().x + threadIdx().x
    if i < length(xs)
        xs[i] *= xs[i]
    end
    return nothing
end

Nt = 1024
Nb = ceil{Int}(length(v) / Nt)
@cuda threads=Nt blocks=Nb cusquare!(v)
```

Basically, almost *any* sensible Julia can just work on the GPU

# JULIA ON THE GPU

```
function cusquare!(xs)
    i = (blockIdx().x - 1) * blockDim().x + threadIdx().x
    if i < length(xs)
        xs[i] *= xs[i]
    end
    return nothing
end

Nt = 1024
Nb = ceil{Int}(length(v) / Nt)
@cuda threads=Nt blocks=Nb cusquare!(v)
```

However, there is no escape hatch: *The code must type check, and can't allocate*

## LEVEL 3: MULTI-GPU SYSTEMS

# LEVEL 3: MULTI-GPU SYSTEMS

`CUDA.jl` composes with `DistributedArrays.jl`

`DistributedArrays.jl` : Domain decomposition and distribution algorithms

`CUDA.jl` : Manages device-side buffers and host-device communication

Both understands Julia's abstract concept of  
an “Array”, so they work together.

*Rapid Software Prototyping for Heterogeneous and Distributed Platforms*  
Besard et. al.

- ▶ julia code on the gpu LITERALLY translating julia IR into PTX
- ▶ multi-gpu with distributedarrays
- ▶ verdict: you cant escape location location location
- ▶ julia has real potential, but it is not a free lunch
- ▶ fighting the abstraction
- ▶ what niche does julia even fill? if you understand it you dont want it, and if you dont understand it you can't use it