Lecture 5: GPU programming

Valentin Churavy (@vchuravy)

Levels of Parallelism in Hardware

1. Instruction-Level Parallelism
2. Data-Level Parallelism
   a. SIMD/Vector
   b. GPUs
3. Thread-Level Parallelism

Instruction-Level Parallelism

Assembly-esque Julia syntax

```julia
function f(A, x)
    i = length(A)

    @label Loop
    a = A[i]     # Load
    c = a + x    # Add
    A[i] = c     # Store
    i = i - 1    # Decrement
    i > 0 && @goto Loop

    return A
end
```

RISC-V Assembly

```
...
Loop: fld    f0, 0(x1)
      fadd.d f4, f0, f2
      fsd    f4, 0(x1)
      addi   x1, x1, -8
      bnez   x1, Loop
...
```

What are the data-dependencies in the loop?

Pipeline Scheduling: Latency

Naive scheduling

```
@label Loop
a = A[i]                 # Cycle 1
# Stall                  # Cycle 2
c = a + x                # Cycle 3
# Stall                  # Cycle 4
# Stall                  # Cycle 5
A[i] = c                 # Cycle 6
i = i - 1                # Cycle 7
i > 0 && @goto Loop      # Cycle 8
```

- Load latency: 1 cycle
- Float arithmetic latency: 2 cycle
- Integer arithmetic latency: 0 cycle

After reordering

```
@label Loop
a = A[i]                 # Cycle 1
i = i - 1                # Cycle 2
c = a + x                # Cycle 3
# Stall                  # Cycle 4
# Stall                  # Cycle 5
A[i+1] = c               # Cycle 6
i > 0 && @goto Loop      # Cycle 7
```

- How many cycles are overhead: 2
- How many cycles are stalls: 2
- How many cycles are actually work: 3

Note: `A[i+1]` is free since it can be precomputed relative to `A[i]`

Unrolling

Goal: Reduce overhead relative to work

- Replicate loop body
- Rename variables
- Change stride and exit condition
- Add tail loop that deals with non-multiples of the unroll factor

Unroll factor: 4

```
@label Loop
a = A[i]
c = a + x
A[i] = c
a1 = A[i-1]
c1 = a1 + x
A[i-1] = c1
a2 = A[i-2]
c2 = a2 + x
A[i-2] = c2
a3 = A[i-3]
c3 = a3 + x
A[i-3] = c3
i = i - 4
i > 4 && @goto Loop
```

- Do we still have stalls?: Yes
- How many stall cycles: 12
- How many cycles are overhead: 2
- How many cycles are actually work: 12

Note: `A[i-1]` is free since it can be precomputed relative to `A[i]`

Loop unrolling and re-ordering

Loop with stalls annotated

```
@label Loop
a = A[i]
# Stall
c = a + x
# Stall
# Stall
A[i] = c
a1 = A[i-1]
# Stall
c1 = a1 + x
# Stall
# Stall
A[i-1] = c1
a2 = A[i-2]
# Stall
c2 = a2 + x
# Stall
# Stall
A[i-2] = c2
a3 = A[i-3]
# Stall
c3 = a3 + x
# Stall
# Stall
A[i-3] = c3
i = i - 4
i > 4 && @goto Loop
```

Re-ordered loop

```
@label Loop
a  = A[i]
a1 = A[i-1]
a2 = A[i-2]
a3 = A[i-3]
c  = a  + x
c1 = a1 + x
c2 = a2 + x
c3 = a3 + x
A[i]   = c
A[i-1] = c1
A[i-2] = c2
A[i-3] = c3
i = i - 4
i > 4 && @goto Loop
```

- How many stalls? 0
- How many overhead cycles: 2
- How many cycles are actually work: 12

Instruction-Level Parallelism

What has your processor done for you recently?

As programmers we have the mental model that a processor takes your instructions in **linear** order

In reality they exploit instruction-level parallelism and have deep cache hierarchies

Ways to exploit instruction-level parallelism:

- Superscalar / Multiple issue
- Out-of-order execution / Speculation
- Prediction

Data-parallelism

Instead of writing serial code and expect the processor to pick up our slack, we could also write explicit parallel code, this can be especially beneficial for inherently data-parallel codes

- Explicit Vector Programming
- GPU Programming

What are data-parallel programs?

- `map`/`broadcast`
- Matrix multiply

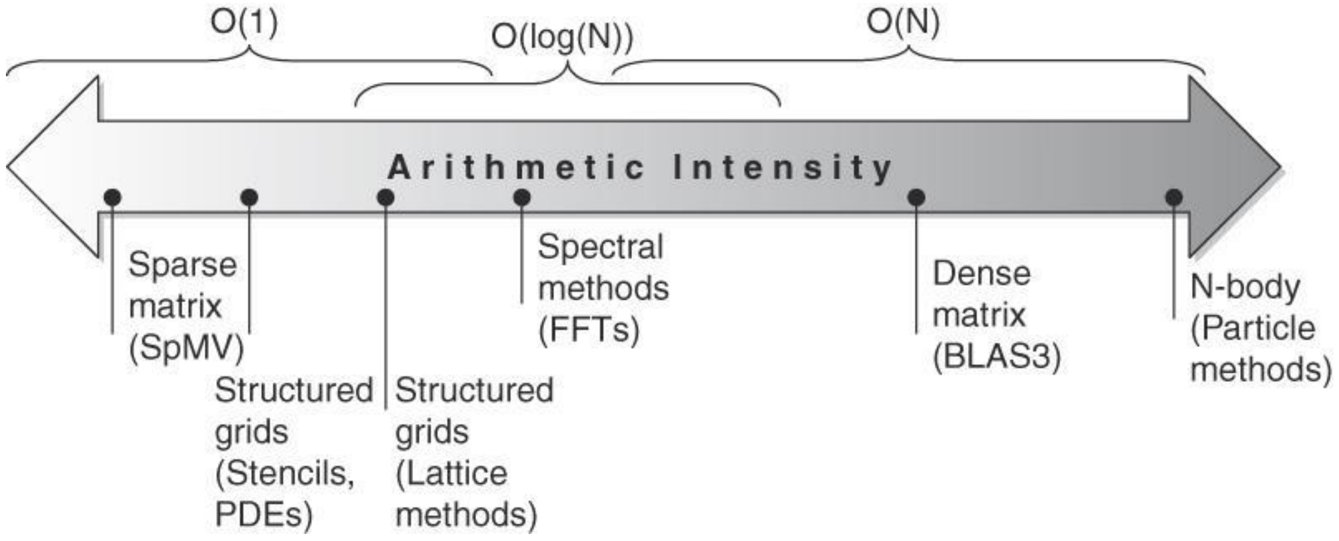


Figure 4.10 Arithmetic intensity, specified as the number of floating-point operations to run the program divided by the number of bytes accessed in main memory [Williams et al. 2009]. Some kernels have an arithmetic intensity that scales with problem size, such as dense matrix, but there are many kernels with arithmetic intensities independent of problem size.

SIMD (Explicitly vectorized)

```
using SIMD
A = rand(Float64, 64)
T = Vec{4, Float64}
x = 1.0

for i in 1:4:length(A)
    a = vload(T, A, i)
    c = a + x
    vstore(c, A, i)
end
```

- Stalls are only per instruction, and not per element
- Reduced overhead
  - 3 instructions processing 4 elements
  - 2 overhead instructions
  - Overhead is amortized across 4 elements.

- We can remove stalls similar to what we did for the serial code
  - Pipelining
  - Interleaving and Unrolling
- Latencies will be higher

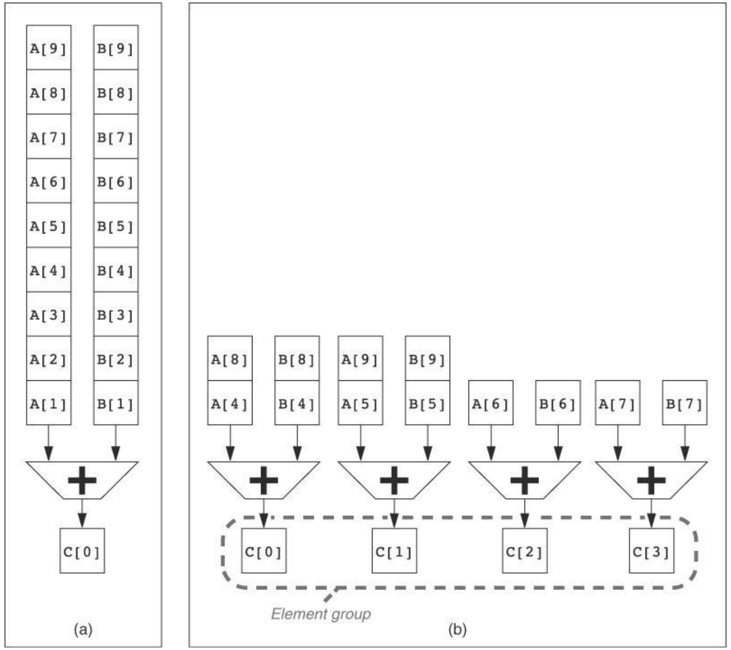


Figure 4.4 Using multiple functional units to improve the performance of a single vector add instruction, C = A + B. The vector processor (a) on the left has a single add pipeline and can complete one addition per cycle. The vector processor (b) on the right has four add pipelines and can complete four additions per cycle. The elements within a single vector add instruction are interleaved across the four pipelines. The set of elements that move through the pipelines together is termed an *element group*. (Reproduced with permission from Asanovic [1998].)

Vector gather and scatter

`vload` loads a block of memory.

How do we work with sparse data and indirect indexing?

- Index vector of the same SIMD length
- Gather to load data
- Scatter to write data

Less efficient than vload

```
idx = Vec((1, 3, 2, 4))
v = vgather(A, idx)
vscatter(v, A, idx)
```

How do we handle code that branches

```julia
A = rand(Int64, 64)
for i in 1:length(A)
    a = A[i]
    if a % 2 == 0
        A[i] = -a
    end
end
```

Data or index dependent controlflow

```julia
using SIMD
A = rand(Int64, 64)
T = Vec{4, Int64}

for i in 1:4:length(A)
    a = vload(T, A, i)
    mask = a % 2 == 0        # calculate mask
    b = -a                   # If branch
    c = vifelse(mask, b, a)  # merge results
    vstore(c, A, i)
end
```

Hardware solution:

- Vector predication (NEC Aurora VX)
- Masked load and store (SIMD Intel CPU)

Compiler solution:

- ISPC
- OpenCL/CUDA

GPU (implicit vectorization)

```
pkg> add CUDAnative
julia> using CUDAnative
julia> function say(num)
        @cuprintf("Thread %ld says: %ld\n",
                  threadIdx().x, num)
        return
      end
julia> @cuda threads=4 say(42)
Thread 1 says: 42
Thread 2 says: 42
Thread 3 says: 42
Thread 4 says: 42
```

- Lane index:
  `threadIdx().x`
- Number of lanes:
  `threads=4`
- Call of a kernel:
  `@cuda`
- Native vector width:
  32

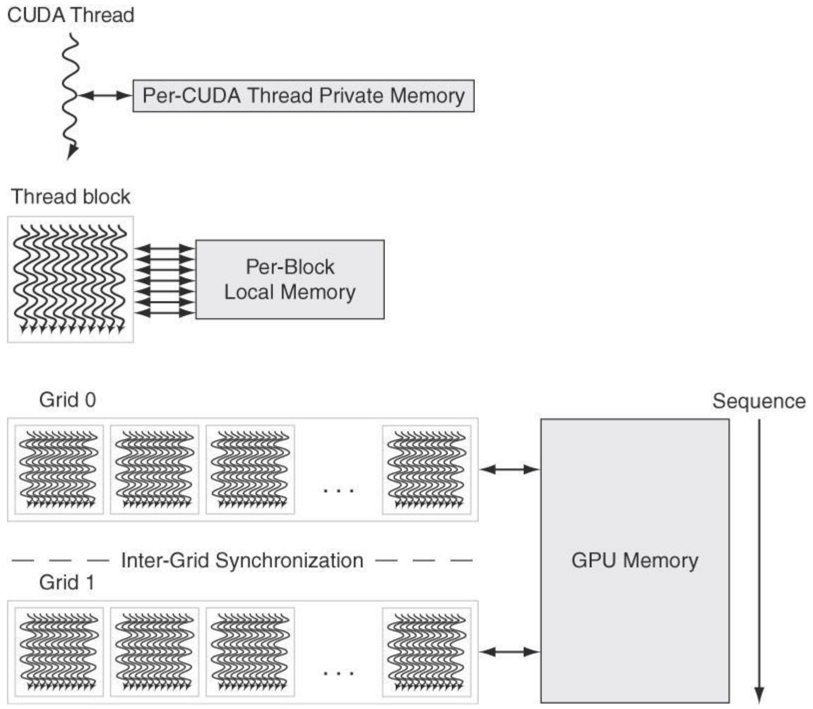


Figure 4.18 GPU Memory structures. GPU Memory is shared by all Grids (vectorized loops), Local Memory is shared by all threads of SIMD instructions within a thread block (body of a vectorized loop), and Private Memory is private to a single CUDA Thread.

Multiple layers of parallelism

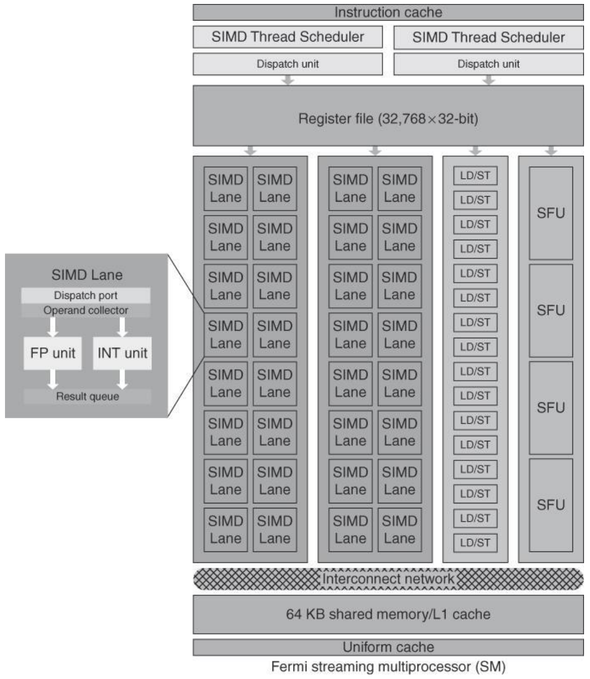
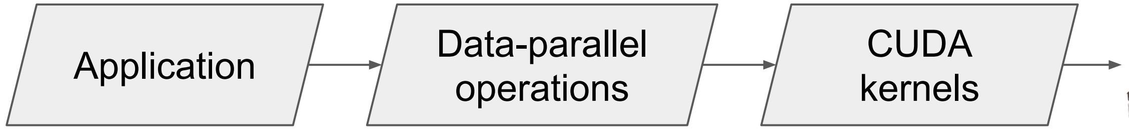


Figure 4.20 Block diagram of the multithreaded SIMD Processor of a Fermi GPU. Each SIMD Lane has a pipelined floating-point unit, a pipelined integer unit, some logic for dispatching instructions and operands to these units, and a queue for holding results. The four Special Function units (SFUs) calculate functions such as square roots, reciprocals, sines, and cosines.

GPU programming in Julia

With material from Tim Besard

How to train your GPU: 10.000 foot view

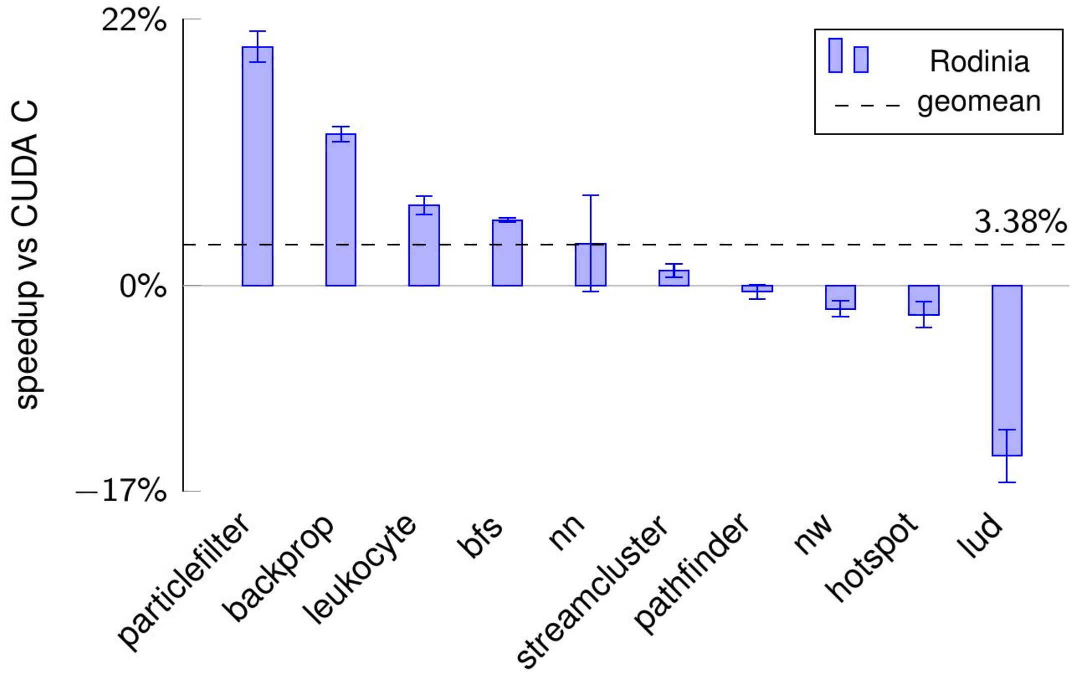


AbstractArrays CuArrays.jl CUDAnative.jl

Why should you care?

1) Performance

2) Powerful abstractions



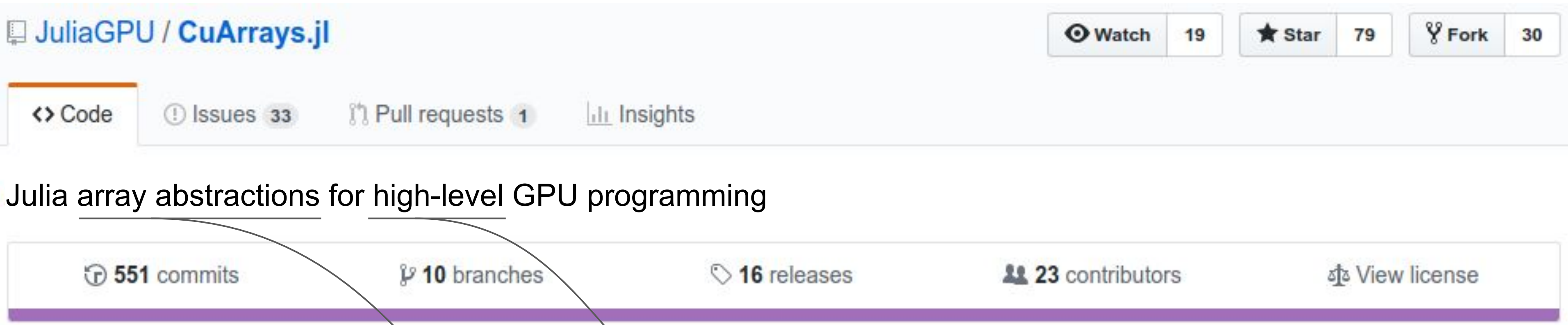
Show me what you got

```
julia> function say(f)
        i = threadIdx().x
        @cuprintf("Thread %ld says: %ld\n",
                  i, f(i))
        return
      end

julia> @cuda say(x->x+1)
Thread 1 says: 2
```

Show me what you got

```
julia> a = CuArray([1., 2., 3.])

julia> function apply(op, a)
        i = threadIdx().x
        a[i] = op(a[i])
        return
      end

julia> @cuda threads=length(a) map(x->x^2, a)

julia> a
3-element CuArray{Float32,1}:
1.0
4.0
9.0
```

```
julia> @device_code_ptx @cuda apply(x->x^2, a)
apply(.param .b8 a[16])
{
        ld.param.u64    %rd1, [a+8];
        mov.u32         %r1, %tid.x;

        // index calculation
        mul.wide.u32    %rd2, %r1, 4;
        add.s64         %rd3, %rd1, %rd2;
        cvta.to.global.u64      %rd4, %rd3;

        ld.global.f32   %f1, [%rd4];
        mul.f32         %f2, %f1, %f1;
        st.global.f32   [%rd4], %f2;

        ret;
}
```



JuliaGPU / **CuArrays.jl**

Watch 19 | Star 79 | Fork 30

Code | Issues 33 | Pull requests 1 | Insights

Julia array abstractions for high-level GPU programming

551 commits | 10 branches | 16 releases | 23 contributors | View license

No GPU programming experience

Data-parallel programming model

≡ Thrust ArrayFire

≢ OpenACC CUDA C/C++

Not just another array library

```
julia> a = CuArray([1,2,3])
3-element CuArray{Int64,1}:
1
2
3
```

```
julia> function apply(op, a)
        i = threadIdx().x
        a[i] = op(a[i])
       end
julia> @cuda threads=length(a) apply(op, a)

julia> map(op, a)

julia> reduce(binop, a)
6

julia> broadcast(+, [1], [2 2], [3 3; 3 3])
2×2 CuArray{Int64,2}:
6  6
6  6

julia> [1] .+ [2 2] .+ [3 3; 3 3]
2×2 CuArray{Int64,2}:
6  6
6  6
```

dot syntax

Not just another array library

```
julia> a = CuArray([1f0, 2f0, 3f0])
3-element CuArray{Float32,1}:
1.0
2.0
3.0

julia> f(x) = 3x^2 + 5x + 2

julia> a .= f.(2 .* a.^2 .+ 6 .* a.^3 .- sqrt.(a))
3-element CuArray{Float32,1}:
  184.0
 9213.753
96231.72
```

⇒

Single kernel!

- Fully specialized
- Highly optimized
- Great performance

Vendor libraries

```
julia> a = CuArray{Float32}(undef, (2,2));
```

CURAND

```
julia> rand!(a)
2×2 CuArray{Float32,2}:
0.73055   0.843176
0.939997  0.61159
```

CUBLAS

```
julia> a * a
2×2 CuArray{Float32,2}:
 1.32629  1.13166
 1.26161  1.16663
```

CUSOLVER

```
julia> LinearAlgebra.qr!(a)
CuQR{Float32,CuArray{Float32,2}}
with factors Q and R:
Float32[-0.613648 -0.78958; -0.78958 0.613648]
Float32[-1.1905 -1.00031; 0.0 -0.290454]
```

CUFFT

```
julia> CUFFT.plan_fft(a) * a
2-element CuArray{Complex{Float32},1}:
 -1.99196+0.0im   0.589576+0.0im
 -2.38968+0.0im  -0.969958+0.0im
```

CUDNN

```
julia> softmax(real(ans))
2×2 CuArray{Float32,2}:
 0.15712  0.32963
 0.84288  0.67037
```

CUSPARSE

```
julia> sparse(a)
2×2 CuSparseMatrixCSR{Float32,Int32}
with 4 stored entries:
  [1, 1]  =  -1.1905
  [2, 1]  =  0.489313
  [1, 2]  =  -1.00031
  [2, 2]  =  -0.290454
```

Effective GPU Programming

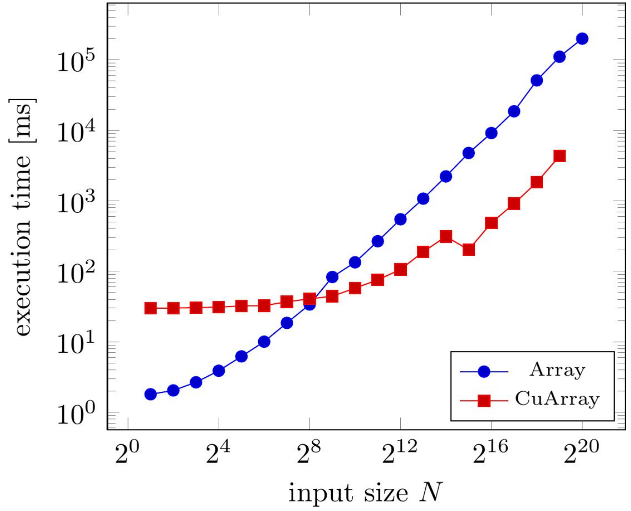
How do you *actually* use this stuff?

Types and gradients, including Forward.gradient

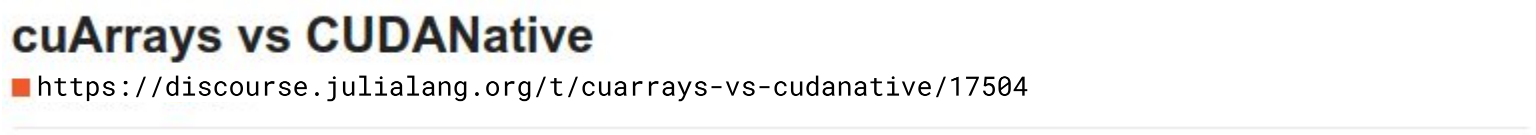
🟩 https://discourse.julialang.org/t/types-and-gradients-including-forward-gradient/946

```julia
using LinearAlgebra

loss(w,b,x,y) = sum(abs2, y - (w*x .+ b)) / size(y,2)
loss∇w(w, b, x, y) = ...
lossdb(w, b, x, y) = ...

function train(w, b, x, y ; lr=.1)
   w -= lmul!(lr, loss∇w(w, b, x, y))
   b -= lr * lossdb(w, b, x, y)
   return w, b
end

n = 100
p = 10
x = randn(n,p)'
y = sum(x[1:5,:]; dims=1) .+ randn(n)'*0.1
w = 0.0001*randn(1,p)
b = 0.0

for i=1:50
   w, b = train(w, b, x, y)
end
```

```julia
x = CuArray(x)
y = CuArray(y)
w = CuArray(w)
```



Fin.



cuArrays vs CUDANative

https://discourse.julialang.org/t/cuarrays-vs-cudanative/17504

```julia
function diff_y(a, b)
   s = size(a)
   for j = 1:s[2]
       for i = 1:s[1]
           @inbounds a[i,j] = b[i,j+1] - b[i,j]
       end
   end
end

N = 64
nx = N^2
ny = N
a = ones(Float32,nx,ny-1)
b = ones(Float32,nx,ny)

julia> using BenchmarkTools
julia> @btime diff_y($a,$b);
 39.599 μs (0 allocations: 0 bytes)

julia> @btime diff_y($(CuArray(a)),$(CuArray(b)));
 4.499 s (3354624 allocations: 165.38 MiB)
```

Performance killers

1. Scalar iteration is slooooow

```julia
function diff_y(a, b)
   s = size(a)
   for j = 1:s[2]
       for i = 1:s[1]
           @inbounds a[i,j] = b[i,j+1] - b[i,j]
       end
   end
end

julia> CuArrays.allowscalar(false)

julia> diff_y(CuArray(a), CuArray(b))
ERROR: scalar getindex is disallowed
Stacktrace:
...
[5] getindex at ./abstractarray.jl
[6] diff_y(::CuArray, ::CuArray)
    at ./REPL[109]:5
...
```

```julia
function diff_y(a, b)
   s = size(a)
   for j = 1:s[2]

       @inbounds a[:,j] .= b[:,j+1] - b[:,j]

   end
end

julia> @btime diff_y($(CuArray(a)),$(CuArray(b)));
 2.503 ms (16884 allocations: 661.50 KiB)
```

Performance killers

2. Avoid multiple kernels

```julia
function diff_y(a, b)



   a .= @views b[:, 2:end] .- b[:, 1:end-1]


end

julia> @btime diff_y($(CuArray(a)),$(CuArray(b)));
 39.057 μs (40 allocations: 2.08 KiB)

julia> @btime diff_y($a,$b);
 39.599 μs (0 allocations: 0 bytes)
```

```julia
function diff_y(a, b)
   s = size(a)
   for j = 1:s[2]

       @inbounds a[:,j] .= b[:,j+1] - b[:,j]

   end
end

julia> @btime diff_y($(CuArray(a)),$(CuArray(b)));
 2.503 ms (16884 allocations: 661.50 KiB)
```

Performance killers

3. Bump the problem size

```
julia> N = 256

julia> @btime diff_y($(CuArray(a)),$(CuArray(b)));
 1.494 ms (40 allocations: 2.08 KiB)

julia> @btime diff_y($a,$b);
 11.719 ms (2 allocations: 128 bytes)
```

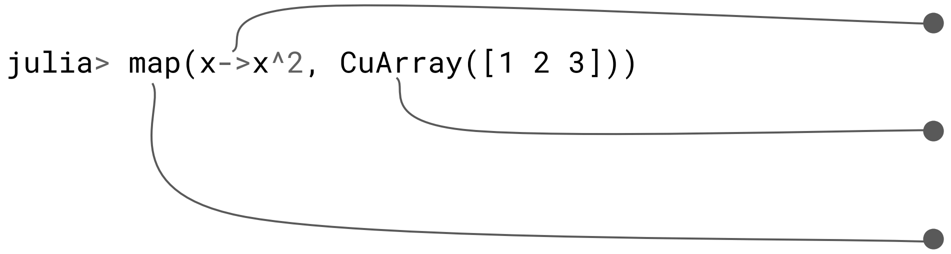
4. Keep data on the GPU

```
julia> @btime diff_y(CuArray($a),CuArray($b));
 72.050 ms (93 allocations: 255.50 MiB)
```

Strengths

1. Single, productive programming language
2. Platform-independent, generic code
3. High-level, zero-cost abstractions
4. Great performance potential
5. **Composability**
6. **Optimizability**

Composability



```
julia> map(x->x^2, CuArray([1 2 3]))
```

Separation of concerns

- *what* is computed
- *where* does it happen
- *how* is it computed

CUDAnative.jl	2383 LOC
GPUArrays.jl	1468 LOC
CuArrays.jl 859	LOC (without libraries)

Composability: reuse of libraries

```
loss(w,b,x,y) = sum(abs2, y - (w*x .+ b)) / size(y,2)

julia> loss(w,b,x,y)
4.222961132618434

julia> loss∇w(w, b, x, y)
1×10 CuArray{Float64,2}:
-1.365  -1.961  -1.14  -2.023  -1.981  -0.2993  -0.2667  -0.07669  -1.038  -0.1823

using ForwardDiff
loss∇w(w, b, x, y) = ForwardDiff.gradient(w -> loss(w, b, x, y), w)

julia> @which mul!(w, w, x)
mul!(...) in CuArrays.CUBLAS at src/blas/highlevel.jl

julia> @which mul!(w, w, ForwardDiff.Dual.(x))
mul!(...) in CuArrays at src/generic_matmul.jl
```

Dynamic Automatic Differentiation of GPU Broadcast Kernels (arXiv:1810.08297)

Optimizability: it's julia all the way down

1. Rewrite using array abstractions
   `using CuArrays` + generic code
2. Avoid GPU antipatterns
3. Specialize with broadcast expressions
4. Specialize with GPU kernels

Use the Tools

Tools

1. Reflection and introspection

```
julia> using CUDAnative

julia> @device_code_llvm curand(2) .+ 2

define void @ptxcall_anonymous(...) {
  …
}
```

```
@device_code_{lowered,typed,warntype,llvm,ptx,sass}
```

```
julia> ENV["JULIA_DEBUG"] = "CUDAnative"

julia> curand(2) .+ 2;
┌ Debug: Compiled getfield(GPUArrays, ...)() to PTX 3.5.0 for SM 3.5.0 using 8 registers.
│ Memory usage: 0 bytes local, 0 bytes shared, 0 bytes constant
└ @ CUDAnative ~/Julia/CUDAnative/src/execution.jl
```

Tools

2. Performance measurements

```
julia> const x = CuArray{Float32}(undef, 1024)

julia> using BenchmarkTools
julia> @benchmark CuArrays.@sync(identity.($x))
BenchmarkTools.Trial:
 memory estimate:  1.34 KiB
 allocs estimate:  33
 --------------
 minimum time:     13.824 μs (0.00% GC)
 median time:      16.361 μs (0.00% GC)
 mean time:        16.489 μs (0.00% GC)
 maximum time:     401.689 μs (0.00% GC)
 --------------
 samples:          10000
 evals/sample:     1
```

Tools

2. Performance measurements

```
julia> const x = CuArray{Float32}(undef, 1024)

julia> using BenchmarkTools
julia> @benchmark CuArrays.@sync(identity.($x))
BenchmarkTools.Trial:
 minimum time:     13.824 μs (0.00% GC)
 maximum time:     401.689 μs (0.00% GC)

julia> CuArrays.@time CuArrays.@sync identity.(x);
 0.000378 seconds (57 CPU allocations: 1.938 KiB)
                  (1 GPU allocation: 4.000 KiB)
```

Tools

2. Performance measurements

```
julia> const x = CuArray{Float32}(undef, 1024)

julia> using BenchmarkTools
julia> @benchmark CuArrays.@sync(identity.($x))
BenchmarkTools.Trial:
 minimum time:     13.824 μs (0.00% GC)
 maximum time:     401.689 μs (0.00% GC)
```

Accurate measurements of possible short-running code

```
julia> CuArrays.@time CuArrays.@sync identity.(x);
 0.000378 seconds (57 CPU allocations: 1.938 KiB)
                  (1 GPU allocation: 4.000 KiB)
```

Memory allocation behavior

```
julia> using CUDAdrv
julia> CUDAdrv.@elapsed identity.(x)
5.888f-6
```

Application performance metrics

Tools

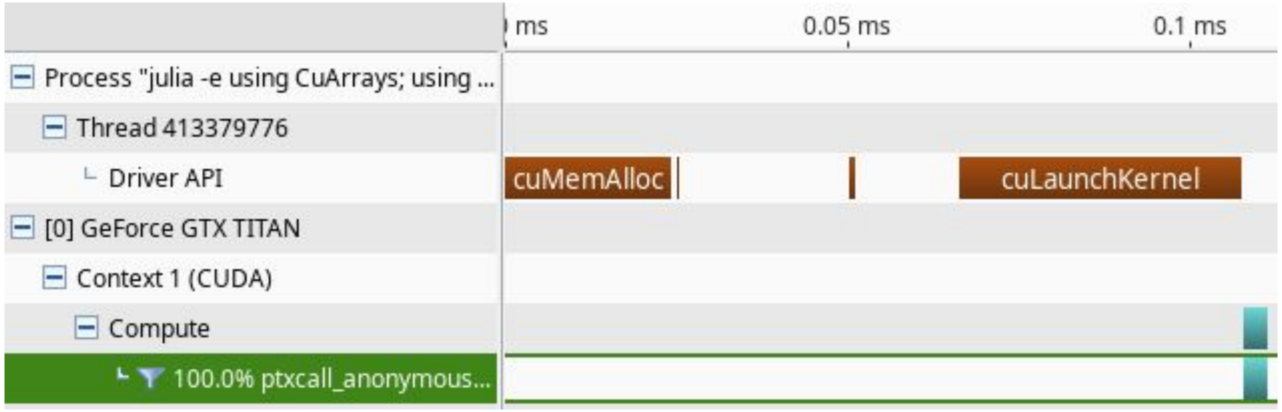
3. Profiling

```
$ nvprof --profile-from-start off julia

julia> const x = CuArray{Float32}(undef, 1024)
julia> identity.(x)

julia> CUDAdrv.@profile begin
         identity.(x)
       end

julia> exit()
==22272== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  3.5520us         1  3.5520us  3.5520us  3.5520us  ptxcall_anonymous
      API calls:   61.70%  39.212us         1  39.212us  39.212us  39.212us  cuLaunchKernel
                   37.36%  23.745us         1  23.745us  23.745us  23.745us  cuMemAlloc
                    0.93%     592ns         2     296ns     222ns     370ns  cuCtxGetCurrent
```

Tools

3. Profiling

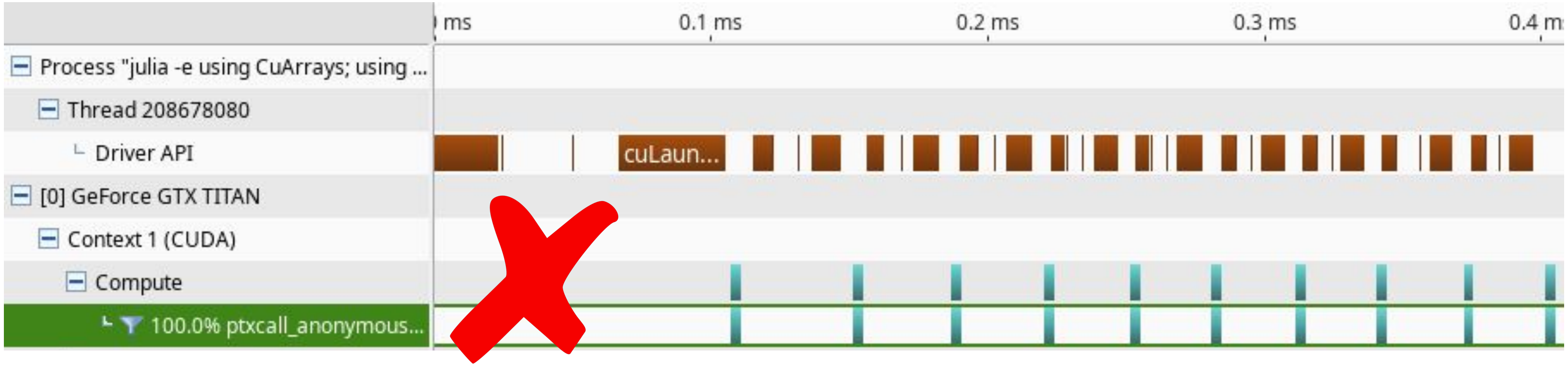
```
$ nvvp julia
julia> identity.(CuArray{Float32}(undef, 1024))
```



Tools

3. Profiling

```
$ nvvp julia
julia> identity.(CuArray{Float32}(undef, 1024))
```

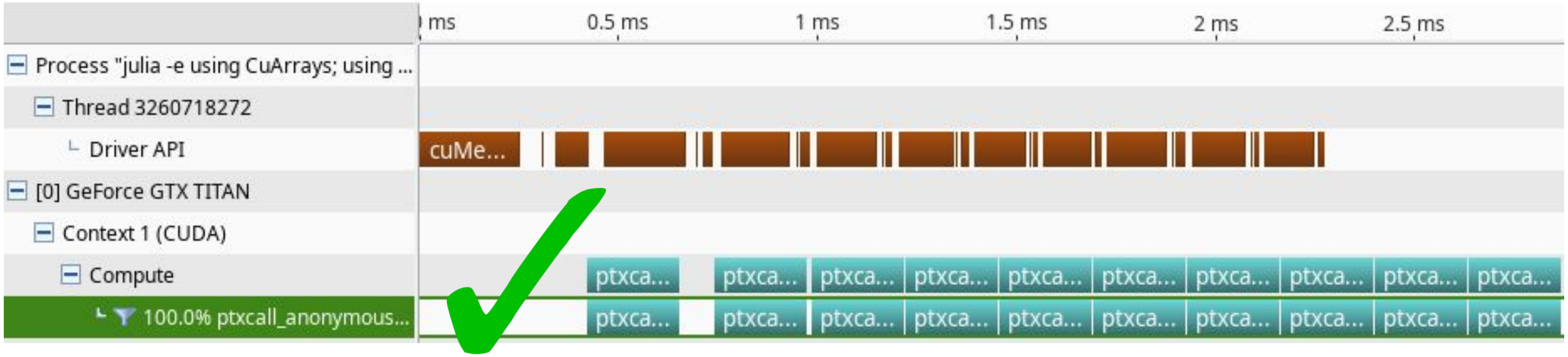


Tools

3. Profiling

```
$ nvvp julia
```

```
julia> sin.(CuArray{Float32}(undef, 1024, 1024))
```



Effectively using GPUs with Julia

Tim Besard (@maleadt)

http://julialang.slack.com/
https://discourse.julialang.org/c/domain/gpu