Effectively using GPUs with Julia

Tim Besard (@maleadt)

What you should know

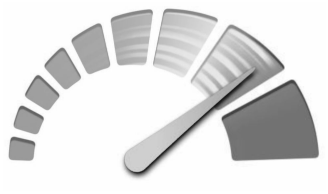
- High-level programming language with low-level performance
- Solves “two language problem”, but requires proficiency

- Hardware accelerator for massively parallel applications
- Throughput oriented: hard to program

Why not both?

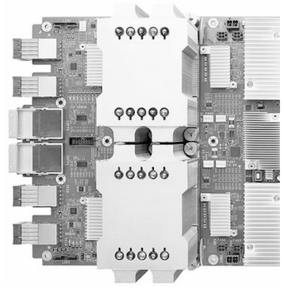
- High-level programming without GPU experience
- Low-level programming for high-performance and flexibility

Choice of hardware



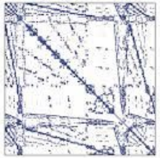
CLArrays.jl

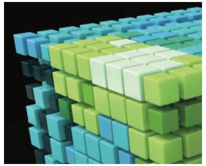
XLA.jl



NVIDIA

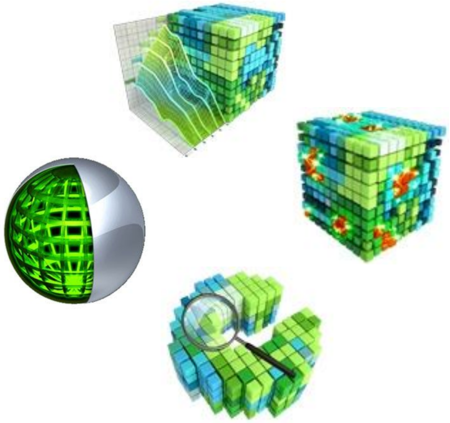
CUDA®





cuDNN

LLVM Compiler Infrastructure



How to train your GPU: 10.000 foot view

Fortran

Thrust

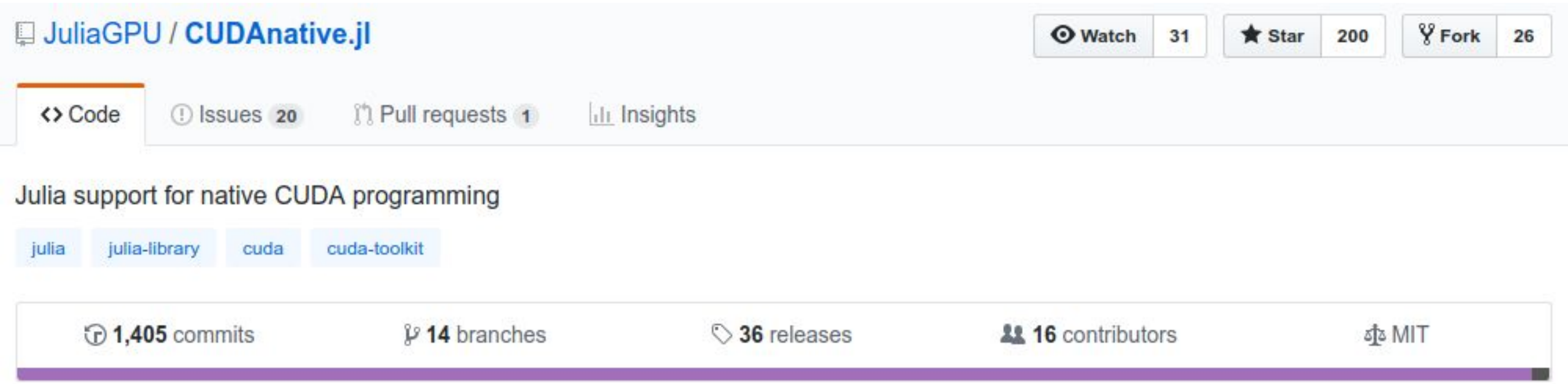
OpenACC

CUDA
C/C++

Application → Data-parallel operations → CUDA kernels →

julia

`AbstractArrays` CuArrays.jl CUDAnative.jl


JuliaGPU / CUDAnative.jl
Watch 31
Star 200
Fork 26
Code
Issues 20
Pull requests 1
Insights
Julia support for native CUDA programming
julia
julia-library
cuda
cuda-toolkit
1,405 commits
14 branches
36 releases
16 contributors
MIT

Hello GPU!

```
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
```

Code is specialized

```
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
```

```
julia> @device_code_typed @cuda say(42)
 1 ─ %1 = CUDAnative.threadIdx_x()::UInt32
 │   %2 = (Core.zext_int)(Core.Int64, %1)::Int64
 │   %3 = (Base.add_int)(%2, 1)::Int64
 │
 │   %4 = (CUDAnative.cuprintf)("...",
 │                              %3, num)::Int32
 └──      return
) => Nothing
```

Code is compiled

```
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
```

```
julia> @device_code_llvm @cuda say(42)
define void @say(i64) {
entry:
 %1 = call i32 @llvm.nvvm.read.ptx.sreg.tid.x()
 %addconv = add nuw nsw i32 %1, 1
 %2 = zext i32 %addconv to i64

 %3 = alloca %printf_args.0
 %4 = bitcast %printf_args.0* %3 to i8*
 %5 = getelementptr inbounds %printf_args.0,
      %printf_args.0* %3, i64 0, i32 0
 store i64 %2, i64* %5
 %6 = getelementptr inbounds %printf_args.0,
      %printf_args.0* %3, i64 0, i32 1
 store i64 %0, i64* %6

 %7 = call i32 @vprintf(i8* ..., i8* %4)
 ret void
}
```

Code is compiled

```
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
```

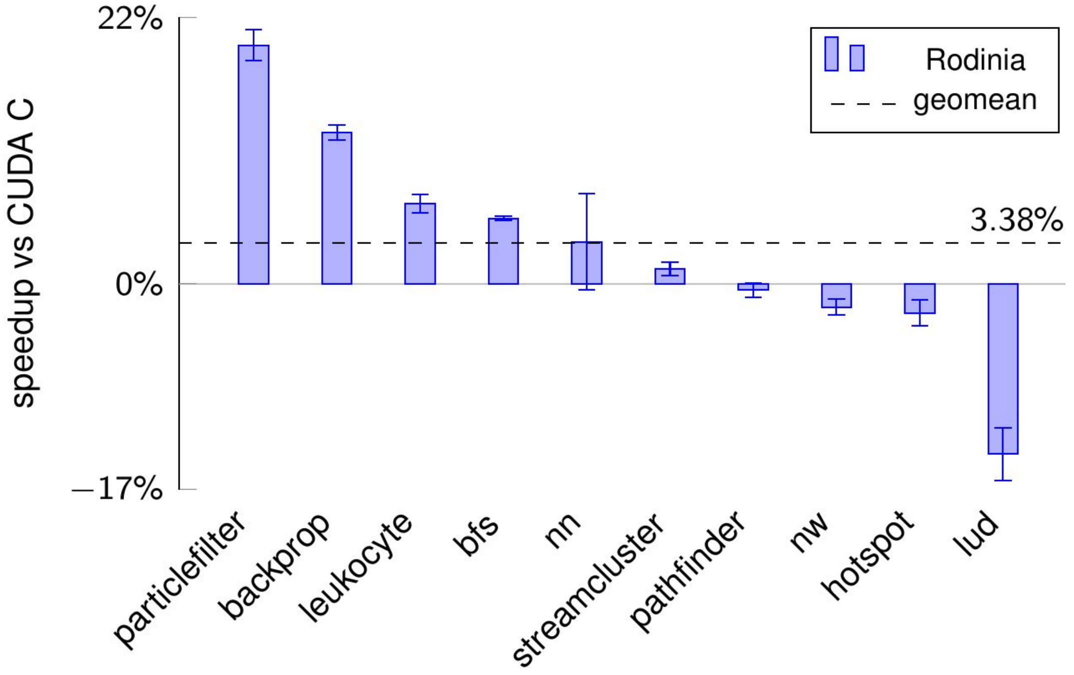
```
julia> @device_code_sass @cuda say(42)
.say:
  S2R R1, SR_TID.X;
  IADD32I R1, R1, 0x1;

  MOV R2, c[0x0][0x44];
  IADD32I R2, R2, -0x10;
  MOV R8, c[0x0][0x140];
  MOV R9, c[0x0][0x144];
  MOV R3, RZ;
  MOV R7, RZ;
  MOV32I R4, 32@lo(__unnamed_1);
  STL.64 [R2+0x8], R8;
  LOP.OR R6, R2, c[0x0][0x24];
  MOV32I R5, 32@hi(__unnamed_1);
  STL.64 [R2], R1;

  JCAL `(vprintf);
  MOV RZ, RZ;
  EXIT;
```

Effective Extensible Programming: Unleashing Julia on GPUs (arXiv:1712.03112)

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
```

dot syntax

```
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
```

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
```

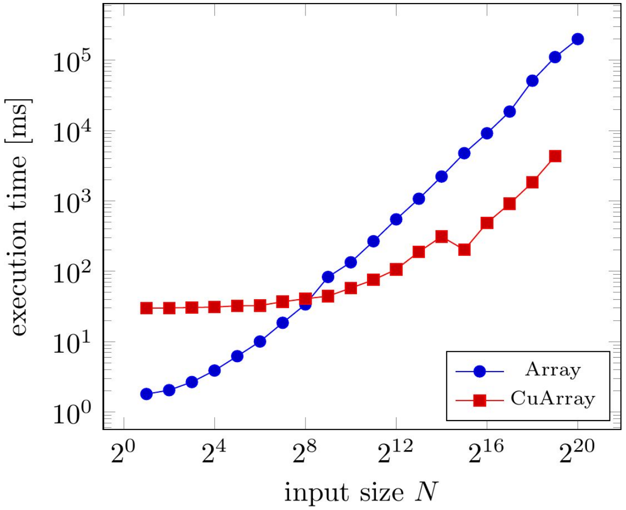
Effective GPU Programming

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

```
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
```




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

Fin.

cuArrays vs CUDANative

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

```
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 soooooow

```
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
...
```

```
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)
```

Strenghts

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](https://arxiv.org/abs/1810.08297))

But Wait… There's MORE!

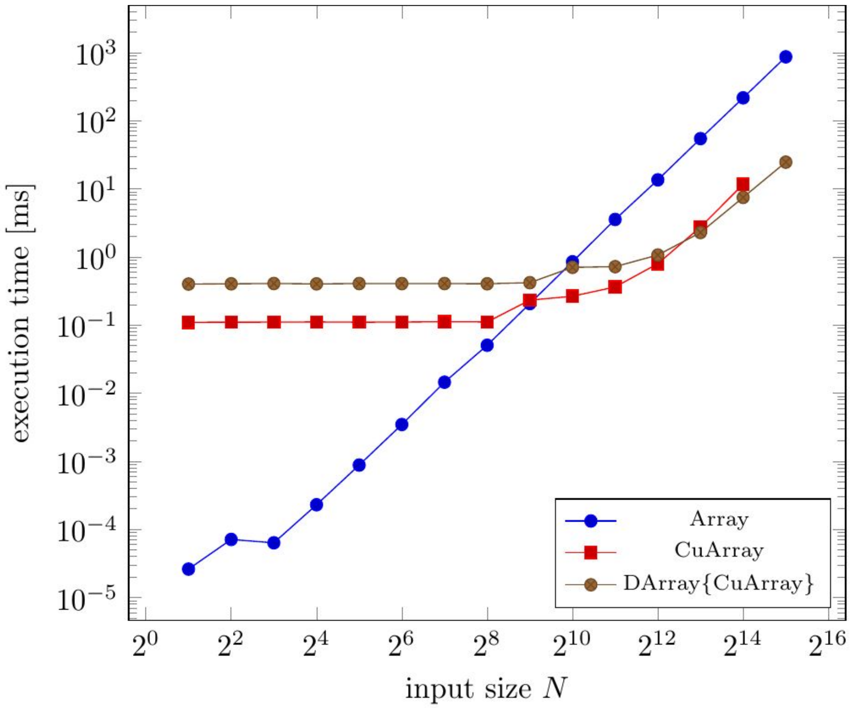
Composability: reuse of infrastructure

WORK IN PROGRESS

```
julia> A = rand(4096,4096)
4096×4096 Array{Float64,2}
```

JuliaParallel / **DistributedArrays.jl**

```
julia> using Distributed, CUDAdrv, CUDAnative
julia> addprocs(length(CUDAdrv.devices()))
julia> remotecall_wait(CUDAnative.device!, p, d)
       for (p,d) in zip(workers(), devices())

julia> using DistributedArrays
julia> dA = distribute(A)
4096×4096 DArray{Float64,2,Array{Float64,2}}

julia> using CuArrays
julia> dgA = map_localparts(CuArray, dA)
4096×4096 DArray{Float64,2,CuArray{Float64,2}}

julia> dgA * dgA
```



Rapid Software Prototyping for Heterogeneous and Distributed Platforms

Optimizability: it's julia all the way down

```julia
function seed!(duals::AbstractArray{Dual{T,V,N}}, x, seed::Partials{N,V} where {T,V,N}
    for i in eachindex(duals)
        duals[i] = Dual{T,V,N}(x[i], seed)
    end
    return duals
end

function ForwardDiff.seed!(duals::AbstractArray{Dual{T,V,N}}, x, seed::Partials{N,V} where {T,V,N}
    duals .= Dual{T,V,N}.(x, Base.RefValue(seed))
    return duals
end

function ForwardDiff.seed!(duals::CuArray{Dual{T,V,N}}, x, seed::Partials{N,V} where {T,V,N}
    function kernel(duals, x, seed)
        i = threadIdx().x
        duals[i] = Dual{T,V,N}(x[i], seed)
        return
    end
    @cuda threads=length(duals) kernel(duals, x, seed)
    return duals
end
```

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)

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

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

Accurate measurements of possible short-running code

Memory allocation behavior

Application performance metrics

Tools

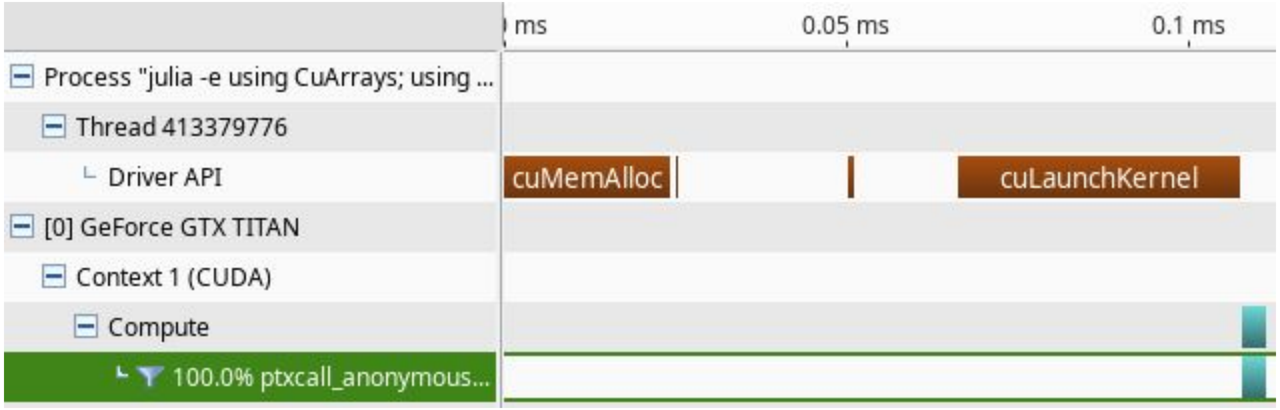
3. Profiling

```
$ 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

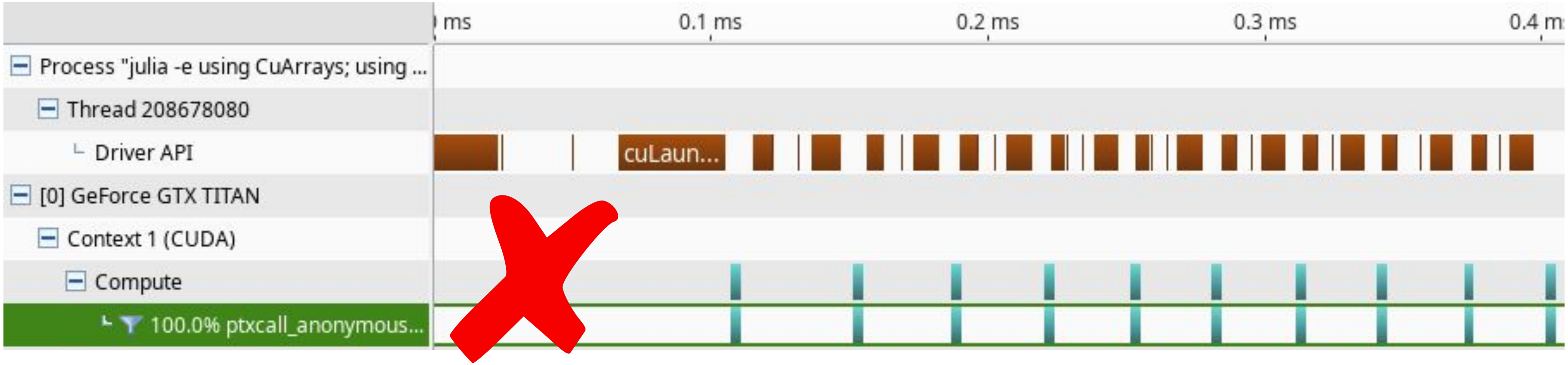
```
$ nvvp julia

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



Tools

3. Profiling

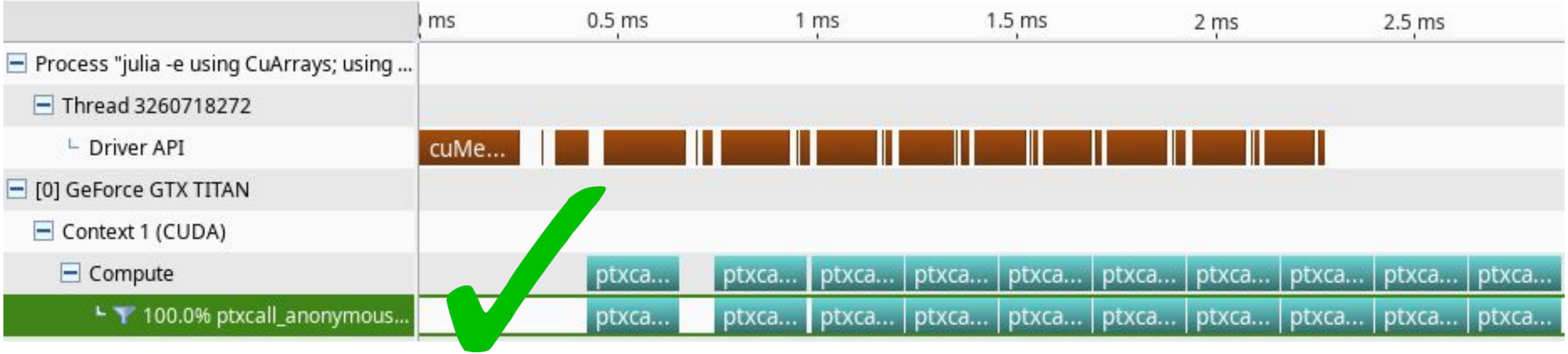
```
$ nvvp julia
julia> identity.(CuArray{Float32}(undef, 1024))
```



Tools

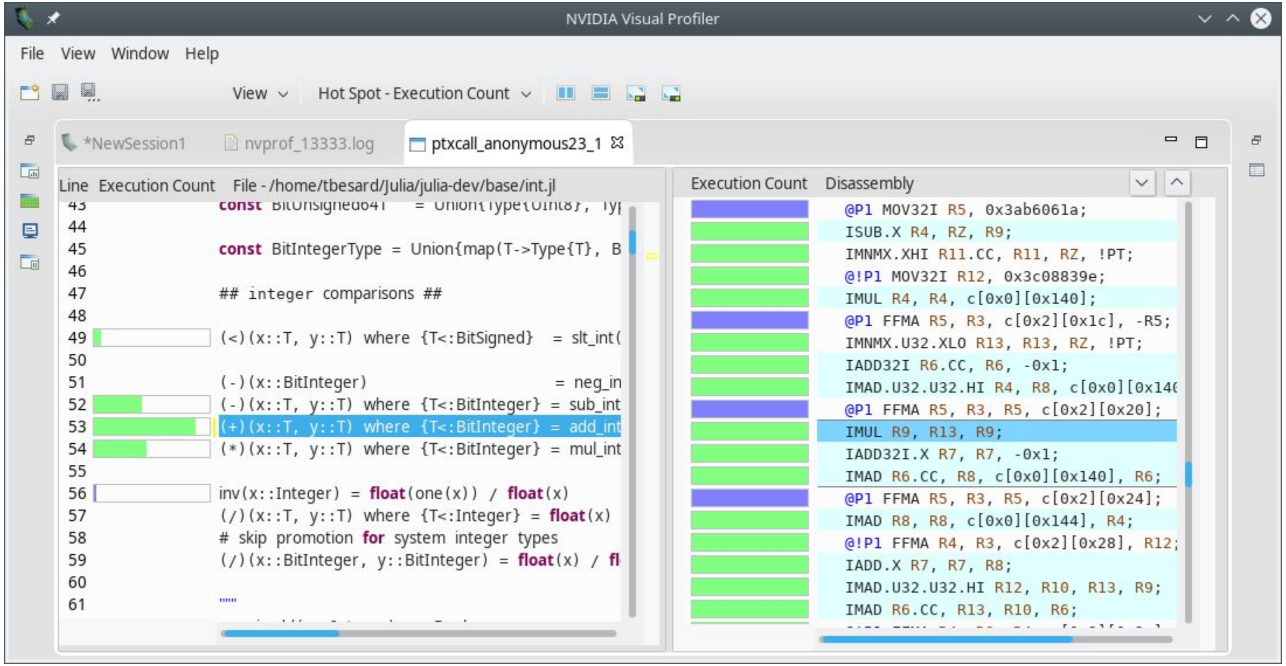
3. Profiling

```
$ nvvp julia

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



Tools

3. Profiling



Conclusion

- Great tools for **single-language GPU programming**
  - CuArrays.jl: high-level and productive
  - CUDAnative.jl: low-level performance

- Strengths: optimizability & composability
- Weaknesses: run into GPU limitations

- Tools
- Community: Slack and Discourse

Effectively using GPUs with Julia

Tim Besard (@maleadt)

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