JuliaHub

GPU Programming in Julia

What, why and how?

Tim Besard

What are GPUs?

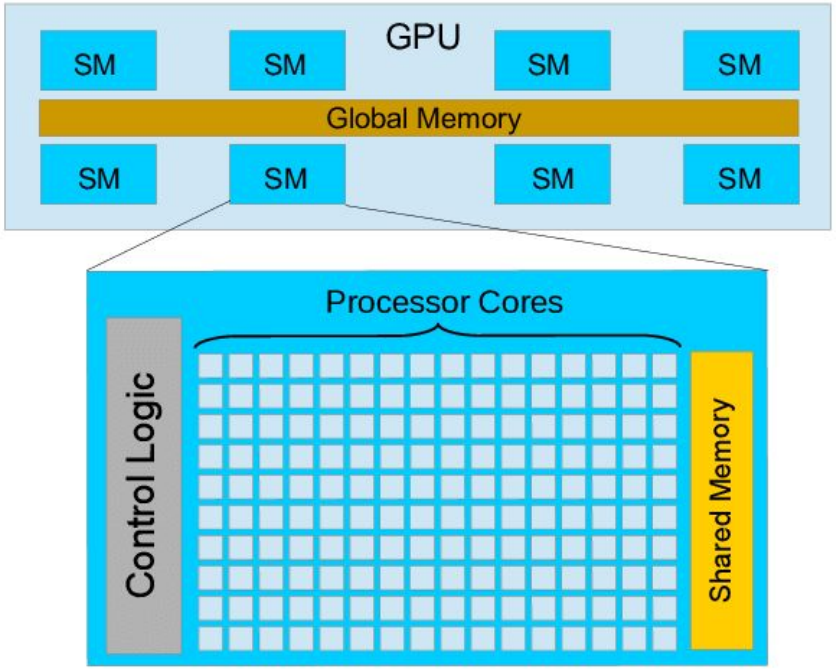
GPUs are massively parallel accelerators

Consist of multiple processors (SMs)

- Simple control logic & cache
- Large register file and lots of ALUs

CPU: few but powerful, independent threads
GPU: many lightweight, cooperative threads

High-bandwidth memory bus to feed threads

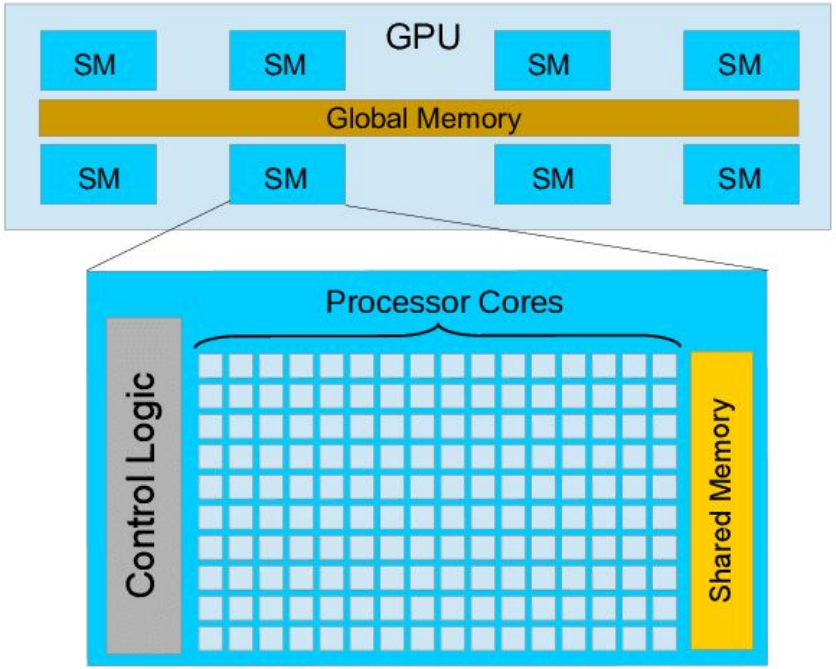


What are GPUs?

GPUs are massively parallel accelerators

Programming a GPU requires:

- Effective algorithms: use SM-local memory, minimize global memory traffic, lockstep execution
- Efficient code: high-quality, statically-typed, predictable machine code



Why Julia?

High-level programming language designed for performance

```
julia> data = (1, rand())
(1, 0.5326182923218289)

julia> sum(data)
1.532618292321829
```

Why Julia?

High-level programming language
designed for performance

```
julia> data = (1, rand())
(1, 0.5326182923218289)

julia> sum(data)
1.532618292321829

julia> @code_llvm sum(data)
define double @julia_sum({ i64, double }* nocapture nonnull readonly align 8 dereferenceable(16) %0) #0 {
  %1 = getelementptr inbounds { i64, double }, { i64, double }* %0, i64 0, i32 0
  %2 = getelementptr inbounds { i64, double }, { i64, double }* %0, i64 0, i32 1
  %3 = load i64, i64* %1, align 8
  %4 = sitofp i64 %3 to double
  %5 = load double, double* %2, align 8
  %6 = fadd double %5, %4
  ret double %6
}
```

Why Julia?

High-level programming language
designed for performance

```
julia> data = (1, rand())
(1, 0.5326182923218289)

julia> sum(data)
1.532618292321829

julia> @code_native sum(data)
julia_sum:
       ldp    d0, d1, [x0]
       scvtf  d0, d0
       fadd   d0, d1, d0
       ldr    d1, [x0, #16]
       fadd   d0, d0, d1
       ret
```

- Machine-native types
- Type inference
- Multiple dispatch
- Specialization and devirtualization
- LLVM-based JIT compiler

Julia: Dynamism and Performance Reconciled by Design (doi:10.1145/3276490)

Why Julia for GPU programming?

High-level programming language <u>designed for performance</u>

```
julia> data = (1, rand())
(1, 0.5326182923218289)

julia> sum(data)
1.532618292321829

julia> @code_ptx sum(data)
.func julia_sum(.param .b64 data)
        ld.param.u64 %rd1, [data];
        ld.u64       %rd2, [%rd1];
        cvt.rn.f64.s64      %fd1, %rd2;
        ld.f64       %fd2, [%rd1+8];
        add.f64      %fd3, %fd2, %fd1;
        st.param.f64 [retval+0], %fd3;
```

- Machine-native types
- Type inference
- Multiple dispatch
- Specialization and devirtualization
- LLVM-based JIT compiler

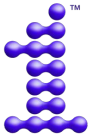
Effective Extensible Programming: Unleashing Julia on GPUs (10.1109/TPDS.2018.2872064)

Julia for GPU programming

Julia is great for GPU programming

High-level language: higher productivity than vendor toolkits

Compiled language: enables native GPU programming

NVIDIA	AMD	Intel	Apple		GraphCore	NEC
						
CUDA.jl	**AMDGPU.jl**	**oneAPI.jl**	**Metal.jl**		**IPUToolkit.jl**	**VectorEngine.jl**

Julia for GPU programming

Key features of the JuliaGPU stack

1. User friendly, but still flexible
2. Multiple programming interfaces
3. Ability to write portable GPU applications

User-friendly installation and set-up

Minimize dependencies through artifacts

- CUDA.jl: NVIDIA driver
- oneAPI.jl: Linux 6.2+
- Metal.jl: macOS 13+
- AMDGPU.jl: ROCm 5.3+

```
pkg> add CUDA
julia> using CUDA

julia> CUDA.versioninfo()
Downloading artifact: ...

CUDA runtime 12.6, artifact installation
CUDA driver 12.6
NVIDIA driver 560.35.3

CUDA libraries:
- CUBLAS: 12.6.1
- ...

1 device:
  0: NVIDIA RTX 6000 Ada Generation
     (sm_89, 47.288 GiB / 47.988 GiB available)
```

User-friendly installation and set-up

Minimize dependencies through artifacts

- CUDA.jl: NVIDIA driver
- oneAPI.jl: Linux 6.2+
- Metal.jl: macOS 13+
- AMDGPU.jl: ROCm 5.3+

```
pkg> add oneAPI
julia> using oneAPI

julia> oneAPI.versioninfo()

Binary dependencies:
- NEO: 24.26.30049+0
- ...

1 driver:
- 00000000-0000-0000-17f9-ff2401037561
  (v1.3.30049, API v1.3.0)

1 device:
- Intel(R) Arc(TM) A770 Graphics
```

User-friendly installation and set-up

Minimize dependencies through artifacts

- CUDA.jl: NVIDIA driver
- oneAPI.jl: Linux 6.2+
- Metal.jl: macOS 13+
- AMDGPU.jl: ROCm 5.3+

```
pkg> add Metal
julia> using Metal

julia> Metal.versioninfo()

macOS 15.0.0, Darwin 24.0.0

Toolchain:
- Julia: 1.10.5
- LLVM: 15.0.7

Julia packages:
- Metal.jl: 1.3.0
- ...

1 device:
- Apple M3 Pro (80.000 KiB allocated)
```

User-friendly installation and set-up

Minimize dependencies through artifacts

- CUDA.jl: NVIDIA driver
- oneAPI.jl: Linux 6.2+
- Metal.jl: macOS 13+
- AMDGPU.jl: ROCm 5.3+

```
pkg> add OpenCL#master pocl_jll
julia> using OpenCL, pocl_jll

julia> OpenCL.versioninfo()

OpenCL.jl version 0.10.0

Toolchain:
 - Julia v1.10.5
 - OpenCL_jll v2024.5.8+1

Available platforms: 1
 - Portable Computing Language
   OpenCL 3.0, PoCL 6.0
   · cpu (fp64, il)
```

User-friendly installation and set-up

Minimize dependencies through artifacts

- CUDA.jl: NVIDIA driver
- oneAPI.jl: Linux 6.2+
- Metal.jl: macOS 13+
- AMDGPU.jl: ROCm 5.3+

Simplifications to get started quickly

Metal:

```
id<MTLDevice> device =
      MTLCreateSystemDefaultDevice();
id<MTLCommandQueue> commandQueue =
[device newCommandQueue];
...
```

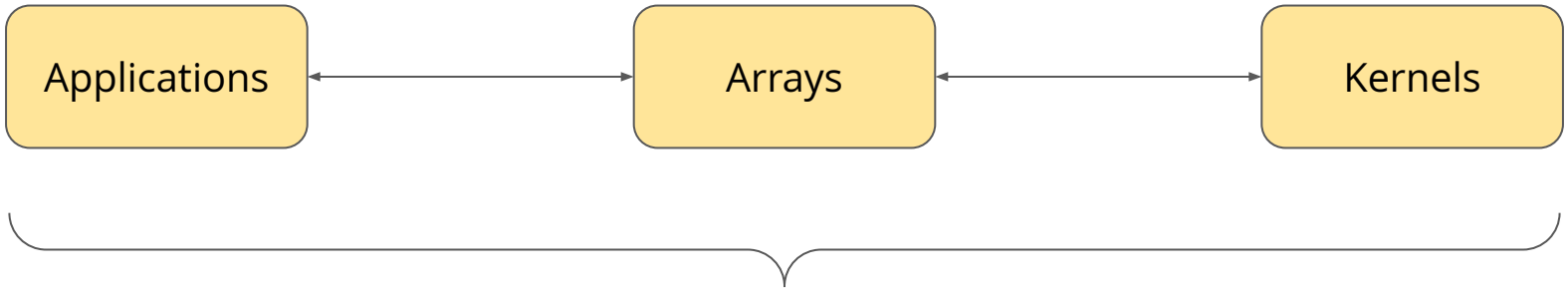
Metal.jl:

```
julia> @metal my_kernel()
```

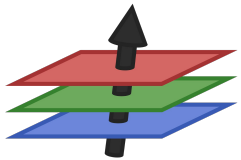
... but low-level control is still possible!

```
julia> dev = MTL.MTLCreateSystemDefaultDevice()
julia> MTLCommandQueue(dev)
```

Multiple programming interfaces



Multiple programming interfaces

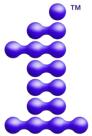


```
using Flux

m = Dense(10,5) |> gpu

x = rand(10) |> gpu

m(x)
```

Array programming: GPUs democratized

CuArray

ROCArray



oneArray

MtlArray

```
using CUDA

a = CuArray( rand(Float32, 1024) )

sum( sin.(a) )
```

Generic code + multiple dispatch = GPU execution

Array programming: GPUs democratized

Goal: Compatibility with Julia's CPU array type

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

julia> view(a, 2:2)
1-element CuArray{Int64, 1}:
2

julia> sum(a)
6

julia> a * 2
1×3 CuArray{Int64, 2}:
2  4  6
```

```
julia> a'
3×1 Adjoint{Int64, CuArray{Int64, 2}}:
1
2
3

julia> a * CuArray([1,2,3])
1-element CuArray{Int64, 1}:
14
```

Array programming: GPUs democratized

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

Random.jl

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

LinearAlgebra.jl

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

SparseArrays.jl

```
julia> sparse(a)
2×2 CuSparseMatrixCSR{Float32, Int32}
with 3 stored entries:
  [1, 1]  =  -1.1905
  [2, 1]  =  0.489313
  [1, 2]  =  -1.00031
```

AbstractFFTs.jl

```
julia> 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
```

NNlib.jl

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

Array programming: The JIT is crucial!

Higher-order abstractions obviate kernel programming

```
julia> a = CuArray([1 2 3])
julia> b = CuArray([4 5 6])

julia> map(a) do x
         x + 1
       end
1×3 CuArray{Int64, 2}:
2  3  4

julia> a .+ 2b
1×3 CuArray{Int64, 2}:
9  12  15
```

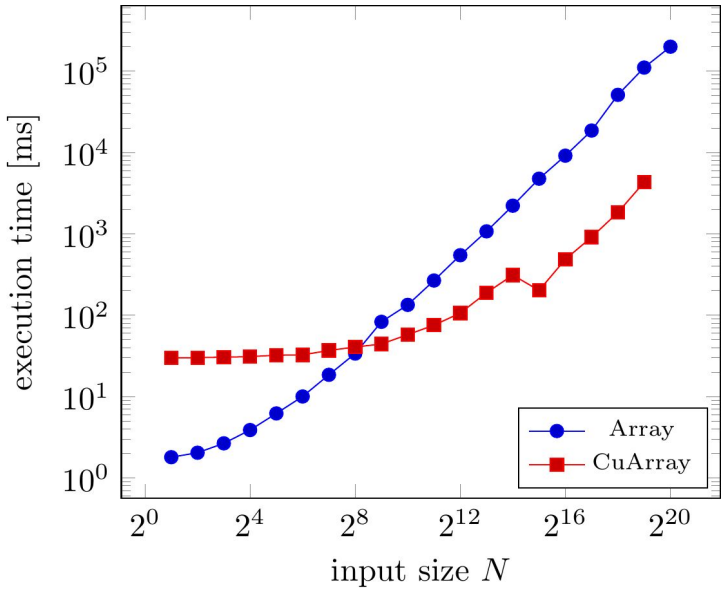
```
julia> reduce(+, a)
6

julia> accumulate(+, b; dims=2)
1×3 CuArray{Int64, 2}:
4  9  15

julia> findfirst(isequal(2), a)
CartesianIndex(1, 2)
```

Array programming is powerful

```
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
```

Array programming is powerful

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

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



Rapid Software Prototyping for Heterogeneous and Distributed Platforms (10.1016/j.advengsoft.2019.02.002)

Kernel programming: Performance & flexibility

```
using CUDA

N = 512
a = CuArray(rand(N))
b = CuArray(rand(N))
c = similar(a)

function vadd(a, b, c)
   i = threadIdx().x
   c[i] = a[i] + b[i]
   return
end

@cuda threads=N vadd(a, b, c)
```

Simplified host code

Similar device code

```
const int N = 512; size_t nb = N * nbof(float);
cudaMalloc((void **)&a, nb);
cudaMemcpy(a, ..., nb, cudaMemcpyHostToDevice);
cudaMalloc((void **)&b, nb);
cudaMemcpy(b, ..., nb, cudaMemcpyHostToDevice);
cudaMalloc((void **)&c, nb);

__global__ void vadd(float *a, float *b, float *c)
{
   int i = threadIdx.x;
   c[i] = a[i] + b[i];
}


vadd<<<512>>>(a, b, c);
```

Kernel programming: Performance & flexibility

```
using Metal

N = 512
a = MtlArray(rand(N))
b = MtlArray(rand(N))
c = similar(a)

function vadd_kernel(a, b, c)
   i = thread_position_in_grid_1d()
   c[i] = a[i] + b[i]
   return
end

@metal threads=N vadd_kernel(a, b, c)
```

Changes from the vendor toolkits to provide a consistent interface

```
kernel void
vadd_kernel(const device float* a,
            const device float* b,
            device float* c,
            uint i [[thread_position_in_grid]]) {
   c[i] = a[i] + b[i];
}
```

Kernel programming: Performance & flexibility

```
using oneAPI

N = 512
a = oneArray(rand(N))
b = oneArray(rand(N))
c = similar(a)

function vadd_kernel(a, b, c)
   i = get_global_id()
   c[i] = a[i] + b[i]
   return
end

@oneapi items=N vadd_kernel(a, b, c)
```

Changes from the vendor toolkits to provide a consistent interface

```
queue.submit([&](handler &h) {
 accessor a(a_buf, h, read_only);
 accessor b(b_buf, h, read_only);
 accessor c(c_buf, h, write_only, noinit);

 h.parallel_for(num_items, [=](auto i) {
   c[i] = a[i] + b[i];
 });
});
```

Kernel programming: Performance & flexibility

Kernel features for GPU kernel development

- Indexing intrinsics
- Shared memory
- Subgroup operations
- Atomic memory

CUDA.jl is the most mature back-end

- Dynamic parallelism
- Cooperative groups
- Tensor cores
- …

The JIT in action

Powerful code introspection utilities

```
@device_code_ptx @cuda threads=N vadd_kernel(a, b, c)
```

```
.visible .entry vadd_kernel(.param .b8 vadd_param_0[16], .param .b8 vadd_param_1[40],
                            .param .b8 vadd_param_2[40], .param .b8 vadd_param_3[40]) {
 ld.param.u64 %rd1, [vadd_param_1];
 ld.param.u64 %rd2, [vadd_param_2];
 ld.param.u64 %rd3, [vadd_param_3];
 mov.u32 %r1, %tid.x;
 mul.wide.u32 %rd4, %r1, 4;
 add.s64 %rd5, %rd1, %rd4;
 ld.global.f32 %f1, [%rd5];
 add.s64 %rd6, %rd2, %rd4;
 ld.global.f32 %f2, [%rd6];
 add.f32 %f3, %f1, %f2;
 add.s64 %rd7, %rd3, %rd4;
 st.global.f32 [%rd7], %f3;
 ret;
}
```

The JIT in action

Powerful code introspection utilities

```
@device_code_agx @metal threads=N vadd_kernel(a, b, c)
```

```
vadd_kernel._agc.main.constant_program:
 device_load      0, i32, xy, r0_r1, u0_u1, 0, signed, lsl 1
 wait             0
 uniform_store    2, i16, xy, 0, r0l_r0h, 12
 uniform_store    2, i16, xy, 0, r1l_r1h, 14
 device_load      0, i32, xy, r0_r1, u2_u3, 0, signed, lsl 1
 wait             0
 uniform_store    2, i16, xy, 0, r0l_r0h, 16
 uniform_store    2, i16, xy, 0, r1l_r1h, 18
 device_load      0, i32, xy, r0_r1, u4_u5, 0, signed, lsl 1
 wait             0
 uniform_store    2, i16, xy, 0, r0l_r0h, 20
 uniform_store    2, i16, xy, 0, r1l_r1h, 22
 stop
```

```
vadd_kernel._agc.main:
 get_sr           r1, sr80 (thread_position_in_grid.x)
 device_load      0, i32, x, r0, u6_u7, r1, signed
 device_load      0, i32, x, r2, u8_u9, r1, signed
 wait             0
 fadd32           r0, r0.discard, r2.discard
 device_store     0, i32, x, r0, u10_u11, r1, signed, 0
 stop
```

The JIT in action

Powerful code introspection utilities

```
@device_code_spirv @oneapi threads=N vadd_kernel(a, b, c)

OpEntryPoint Kernel %vadd "vadd_kernel"

%20 = OpLoad %v3ulong %__spirv_BuiltInGlobalInvocationId
%21 = OpCompositeExtract %ulong %20 0

%27 = OpInBoundsPtrAccessChain %_ptr_CrossWorkgroup_float %arg0 %21
%28 = OpLoad %float %27
%31 = OpInBoundsPtrAccessChain %_ptr_CrossWorkgroup_float %arg1 %21
%32 = OpLoad %float %31
%33 = OpFAdd %float %28 %32
%36 = OpInBoundsPtrAccessChain %_ptr_CrossWorkgroup_float %arg2 %21
OpStore %36 %33

OpReturn
OpFunctionEnd
```

Benchmarking and profiling

Convenient benchmarking utilities

```
julia> a = CuArray(1:9_999_999)

julia> @time CUDA.@sync a .+ reverse(a);
 0.000551 seconds (37 allocations: 1.562 KiB)

julia> CUDA.@time a .+ reverse(a);
 0.000517 seconds (37 CPU allocations: 1.562 KiB)
                  (2 GPU allocations: 152.588 MiB, 2.17% memmgmt time)
```

Benchmarking and profiling

Convenient benchmarking utilities

```
julia> a = CuArray(1:9_999_999)

julia> using BenchmarkTools
julia> @benchmark CUDA.@sync $a .+ reverse($a)
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  466.216 µs …   2.339 ms  ┊ GC (min … max): 0.00% … 15.88%
 Time  (median):     480.915 µs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   486.750 µs ± 102.462 µs  ┊ GC (mean ± σ):  0.23% ±  0.87%
```


```
 Memory estimate: 1.52 KiB, allocs estimate: 35.
```

Benchmarking and profiling

Support for vendor-specific profiling interfaces

```
julia> CUDA.@profile a .+ reverse(a)
Profiler ran for 474.69 µs, capturing 19 events.

Host-side activity: calling CUDA APIs took 143.05 µs (30.14% of the trace)
```

Time (%)	Total time	Time distribution	Name
24.61%	116.83 µs	58.41 µs ± 74.85 (5.48...111.34)	cuMemAllocFromPoolAsync
4.97%	23.6 µs	11.8 µs ± 5.9 (7.63...15.97)	cuLaunchKernel

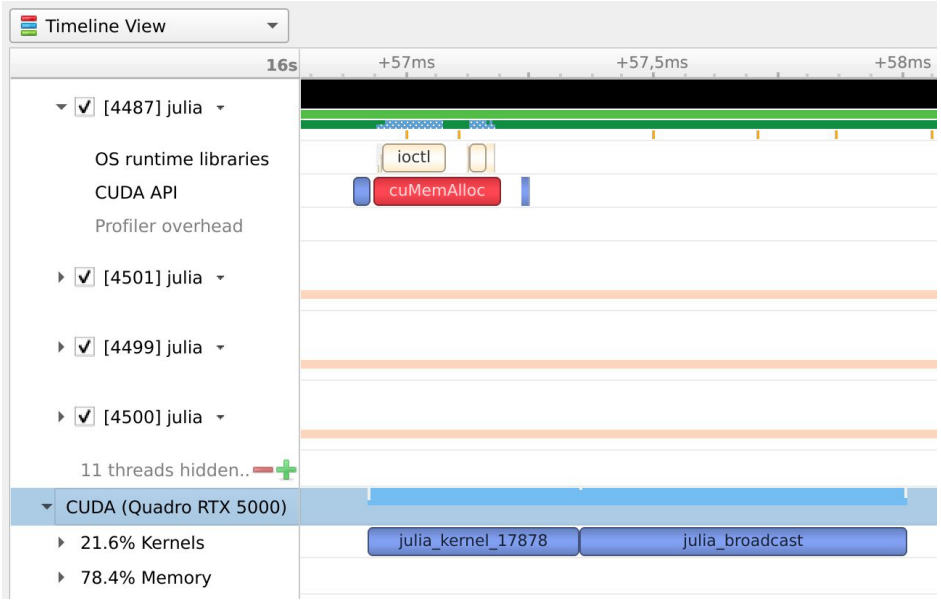
```
Device-side activity: GPU was busy for 422.72 µs (89.05% of the trace)
```

Time (%)	Total time	Name
56.65%	268.94 µs	broadcast_kernel
32.40%	153.78 µs	reverse_kernel

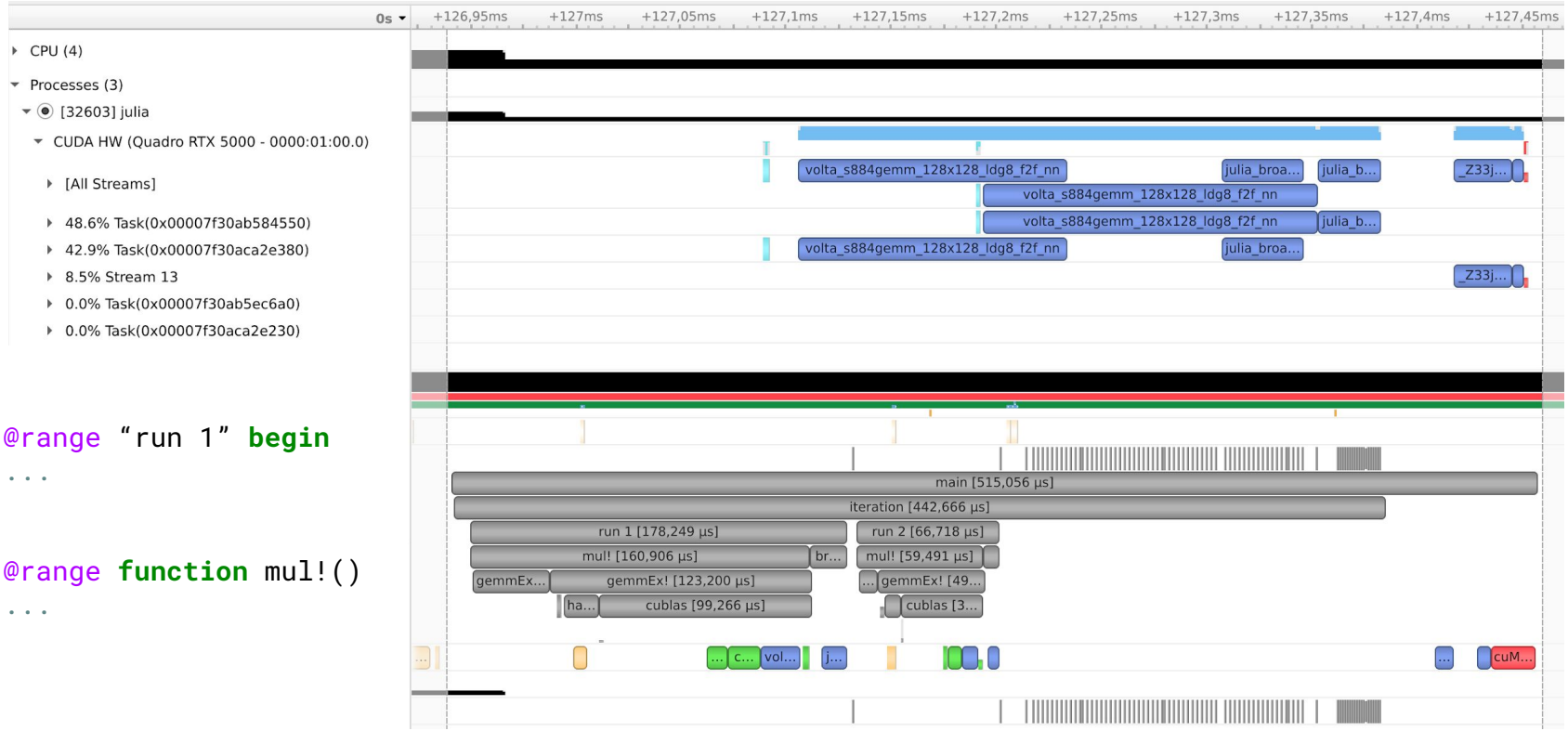
Benchmarking and profiling

Support for vendor-specific profiling interfaces

- CUDA.jl: NSight Systems & Compute
- Metal.jl: XCode & Instruments
- AMDGPU.jl: rocprof
- oneAPI.jl: VTune



Benchmarking and profiling

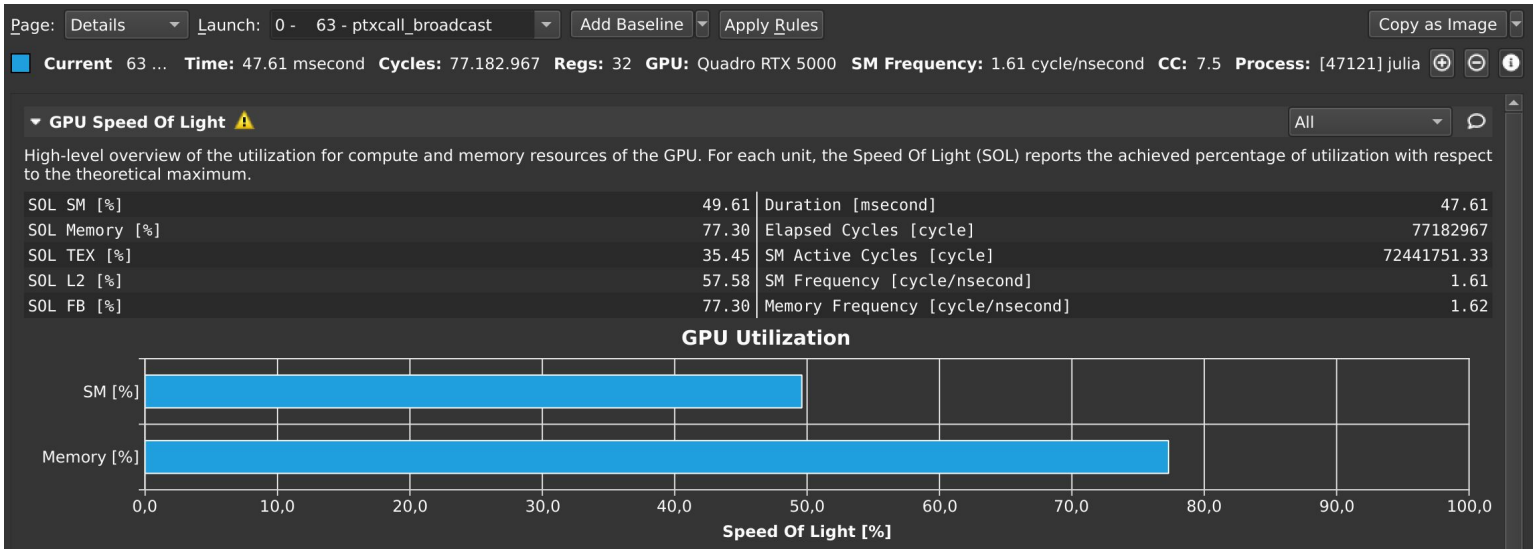



```
NVTX.@range “run 1” begin
    # ...
end

NVTX.@range function mul!()
    # ...
end
```

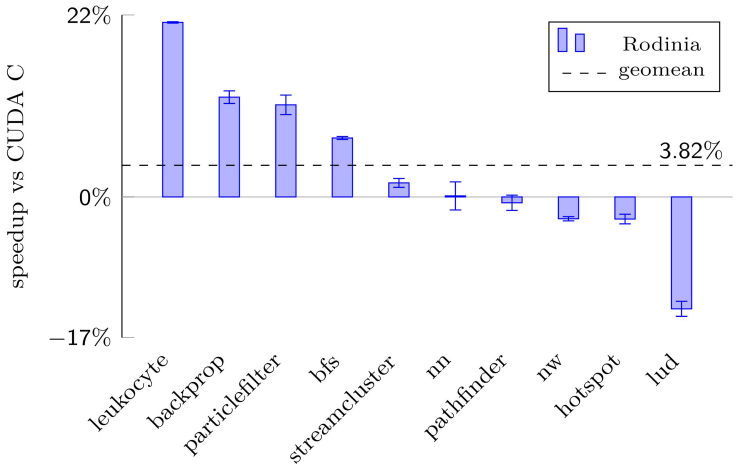
Benchmarking and profiling

```
$ nv-nsight-cu-cli --mode=launch julia
julia> CUDA.@profile external=true a .+ reverse(a)
julia>
```

```
$ nv-nsight-cu
```



Julia on GPUs performs great!



Performance of the Rodinia benchmark suite implemented in Julia, compared to CUDA C.

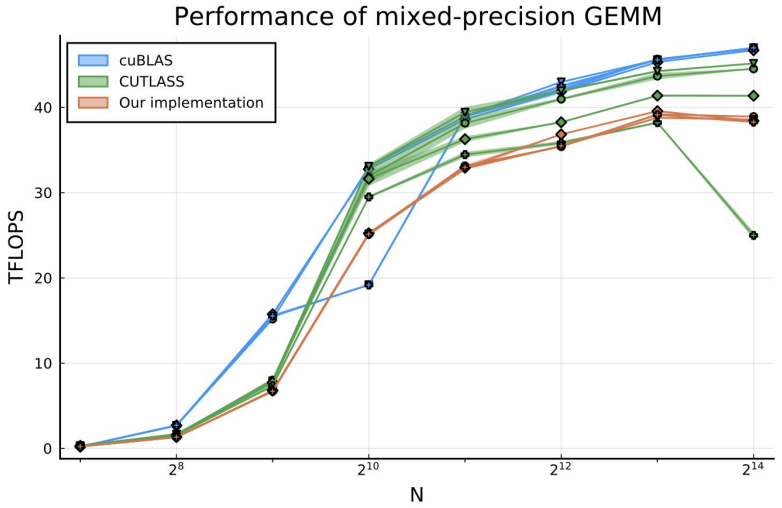
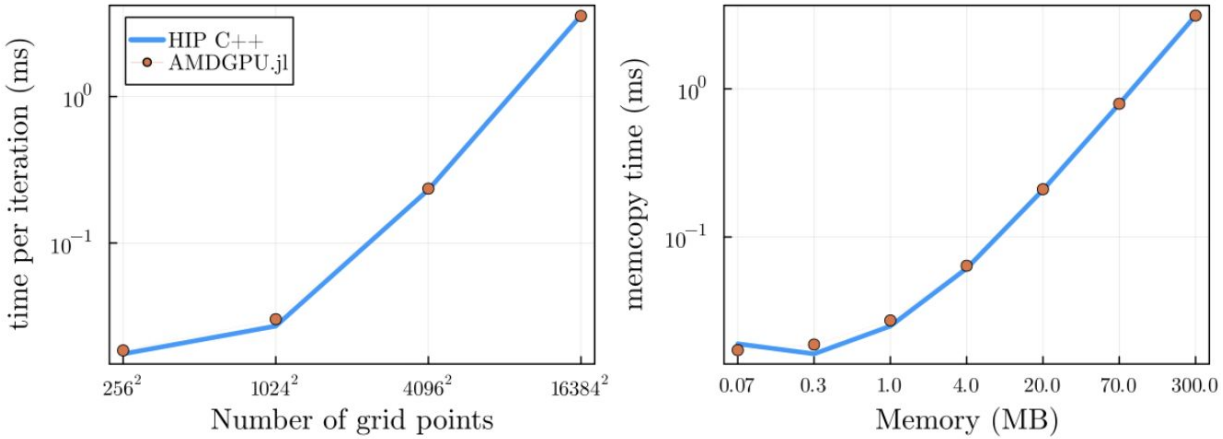


Figure 11. Performance of our mixed-precision GEMM and the state-of-the-art implementation in CUTLASS and cuBLAS.

Effective Extensible Programming: Unleashing Julia on GPUs (10.1109/TPDS.2018.2872064)

Flexible Performant GEMM Kernels on GPUs (10.1109/TPDS.2021.3136457)

Julia on GPUs performs great!



Preliminary performance of a memcopy and 2D diffusion kernel implemented in Julia with AMDGPU.jl and executed on a MI250x GPU.

What’s about portability?

Array programming: (mostly) portable by nature

What's about portability?

Array programming: (mostly) portable by nature

Kernel programming: use KernelAbstractions.jl

```
@kernel function memcpy_kernel!(A, @Const(B))
   I = @index(Global)
   @inbounds A[I] = B[I]
end

function memcpy!(A, B)
   backend = get_backend(A)

   kernel = memcpy_kernel!(backend)
   kernel(A, B, ndrange=length(A))
end
```

Works with: CUDA.jl, AMDGPU.jl, Metal.jl oneAPI.jl, and CPU!

Supported operations:

- Indexing: `@index`, `@groupsize`
- Memory: `@localmem`, `@private`
- Execution: `@synchronize`, `@uniform`
- Atomics: `Atomix.@atomic`
- Output: `@print`

What’s about portability?

DiffEqGPU.jl: GPU-accelerated ODE solvers

Automated Translation and Accelerated Solving of Differential Equations on Multiple GPU Platforms (10.1016/j.cma.2023.116591)

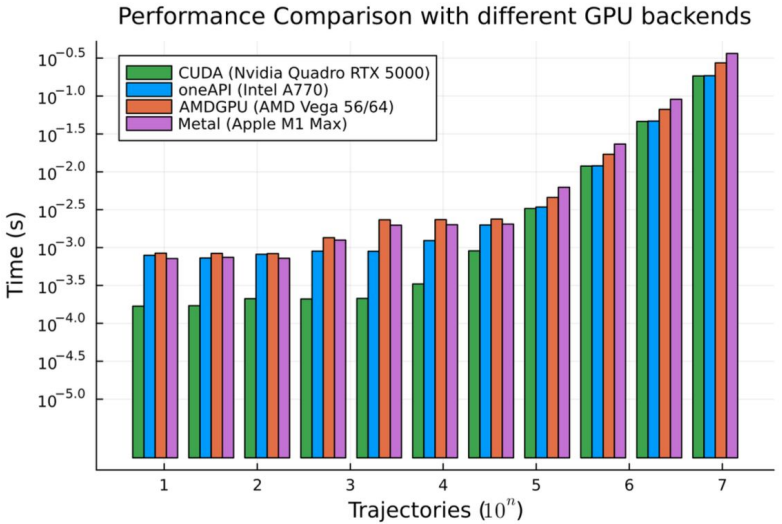


Figure 7: A comparison of ODE solve timings with fixed time-stepping, measured on different GPU platforms. We measure the time *(lower the better)* versus number of parallel solves. Here, the NVIDIA GPUs perform the best owing to the most-optimized library and matured ecosystem with JuliaGPU.

What’s about portability?

DiffEqGPU.jl: GPU-accelerated ODE solvers

Automated Translation and Accelerated Solving of Differential Equations on Multiple GPU Platforms (10.1016/j.cma.2023.116591)

WaterLily.jl: GPU-accelerated fluid simulator

WaterLily.jl: A differentiable fluid simulator in Julia with fast heterogeneous execution (arXiv:2304.08159)

weymouth Apr 2023

The speed-up with GPUs enables impressive 3D flow simulations on a PC. This jelly fish simulation runs and visualizes real-time on my laptop:

What’s about portability?

DiffEqGPU.jl: GPU-accelerated ODE solvers

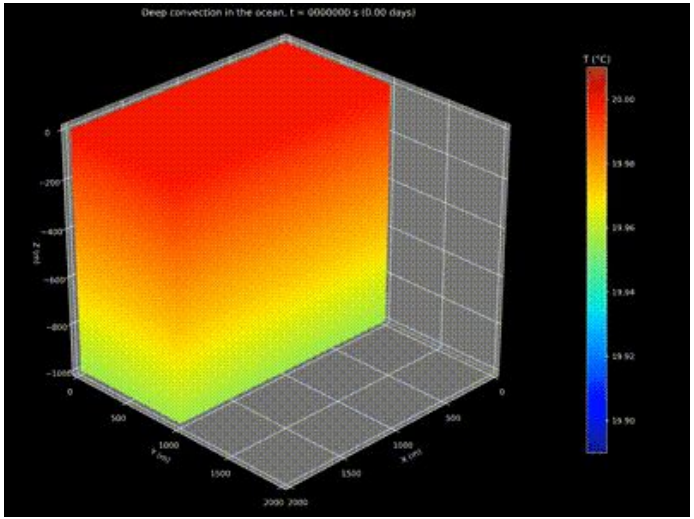
Automated Translation and Accelerated Solving of Differential Equations on Multiple GPU Platforms (10.1016/j.cma.2023.116591)

WaterLily.jl: GPU-accelerated fluid simulator

WaterLily.jl: A differentiable fluid simulator in Julia with fast heterogeneous execution (arXiv:2304.08159)

Oceananigans.jl: GPU-accelerated ocean simulator

Oceananigans.jl: A model that achieves breakthrough resolution, memory and energy efficiency in global ocean simulations (arXiv:2309.06662)



JuliaHub

Case study

How do we *actually* use all this?

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(A_i - B_i)^2}$$

```
rmse(A, B) = sqrt(sum((A-B).^2) / length(A))

N = 2048
T = Float32

A = rand(T, N, N)
B = rand(T, N, N)

using BenchmarkTools
@btime rmse(A, B)
# 3.590 ms (4 allocations: 32.00 MiB)
```

```
using CUDA
CUDA.allowscalar(false)

dA = CuArray(A)
dB = CuArray(B)


@btime rmse(dA, dB)
# 57.030 µs (119 allocations: 5.78 KiB)
```

Identifying optimization opportunities

Start by profiling the application

```
julia> CUDA.@profile rmse(dA, dB)
Profiler ran for 157.36 µs, capturing 226 events.
Host-side activity: calling CUDA APIs took 80.59 µs (51.21% of the trace)
```

Time (%)	Total time	Calls	Time distribution	Name
12.73%	20.03 µs	3	6.68 µs ± 1.09 (5.72 ‥ 7.87)	**cuLaunchKernel**
11.82%	18.6 µs	1		**cudaLaunchKernel**
8.94%	14.07 µs	1		cuMemcpyDtoHAsync
4.09%	6.44 µs	4	1.61 µs ± 0.23 (1.43 ‥ 1.91)	cuMemAllocFromPoolAsync
0.45%	715.26 ns	2	357.63 ns ± 168.59 (238.42 ‥ 476.84)	cuStreamSynchronize

```
Device-side activity: GPU was busy for 41.48 µs (26.36% of the trace)
```

Time (%)	Total time	Calls	Name
9.09%	14.31 µs	1	**partial_mapreduce_grid(identity, add_sum, Float32, CartesianInd**
7.42%	11.68 µs	1	**void geam_kernel<int, float, false, false, false, false, 6, 5,**
7.42%	11.68 µs	1	**_35(CuKernelContext, CuDeviceArray<Float32, 2, 1>, Broadcasted<**
1.52%	2.38 µs	1	partial_mapreduce_grid(identity, add_sum, Float32, CartesianInd
0.91%	1.43 µs	1	[copy device to pageable memory]

Optimizing array operations for the GPU

- Pre-allocating outputs
- Prefer in-place operations
- Fuse operations together

Optimizing array operations for the GPU

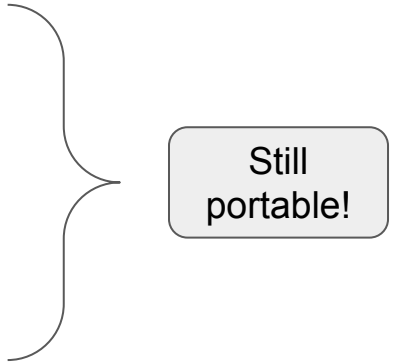
- Pre-allocating outputs
- Prefer in-place operations
- **Fuse operations together**

```
rmse(A, B) = sqrt(sum((A-B).^2) / length(A))
@btime rmse(A, B)     # 3.590 ms (4 allocations: 32.00 MiB)
@btime rmse(dA, dB)   # 57.030 µs

rmse(A, B) = sqrt(sum((A.-B).^2) / length(A))
@btime rmse(A, B)     # 1.796 ms (3 allocations: 16.00 MiB)
@btime rmse(dA, dB)   # 42.650 µs
```

Optimizing array operations for the GPU

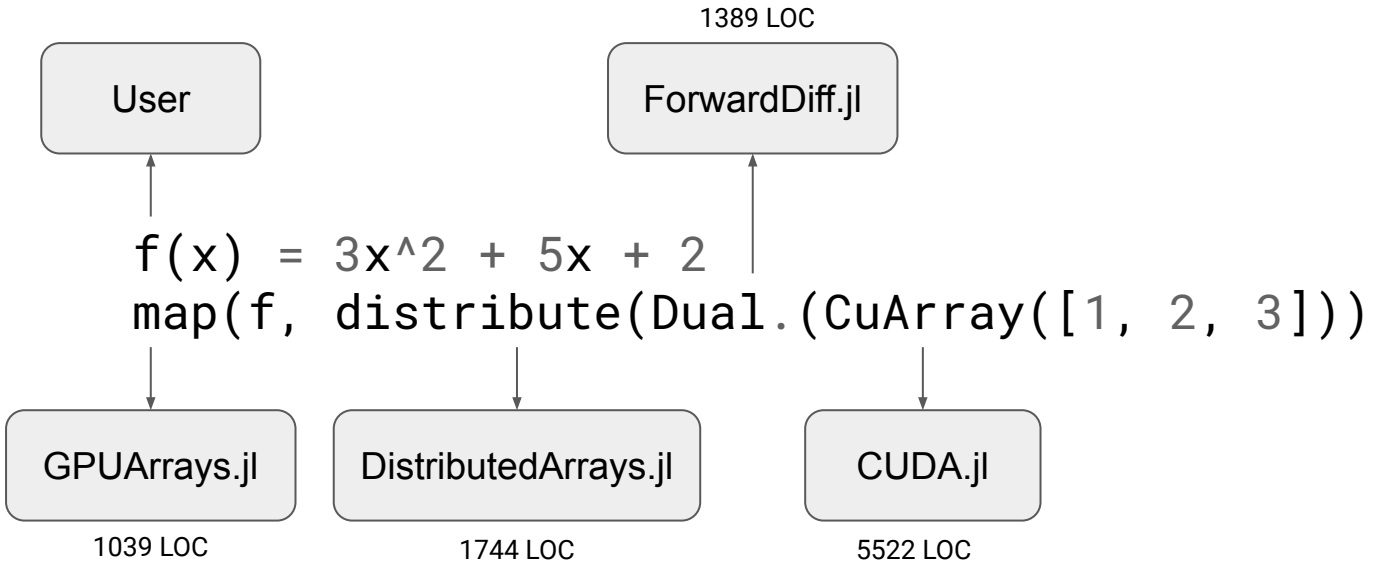
- Pre-allocating outputs
- Prefer in-place operations
- **Fuse operations together**

```
rmse(A, B) = sqrt(sum((A-B).^2) / length(A))
@btime rmse(A, B)     # 3.590 ms (4 allocations: 32.00 MiB)
@btime rmse(dA, dB)   # 57.030 µs

function rmse(A, B)
   bc = Base.broadcasted(A, B) do a, b
       (a - b) ^ 2
   end
   sqrt(sum(bc) / length(A))
end
@btime rmse(A, B)     # 7.768 ms
@btime rmse(dA, dB)   # 34.120 µs
```



Pro tip: Write generic array code!

Generic code can be reused and composed.



The Unreasonable Effectiveness of Multiple Dispatch (youtu.be/kc9HwsxE1OY)

Optimizing array operations for the GPU

Fusing array operations reduces kernel launches

```
julia> CUDA.@profile rmse(dA, dB)
Profiler ran for 111.82 µs, capturing 153 events.
Host-side activity: calling CUDA APIs took 56.74 µs (50.75% of the trace)
```

Time (%)	Total time	Calls	Time distribution	Name
22.81%	25.51 µs	2	12.76 µs ± 6.91 (7.87 ‥ 17.64)	**cuLaunchKernel**
11.73%	13.11 µs	1		cuMemcpyDtoHAsync
4.05%	4.53 µs	2	2.26 µs ± 0.84 (1.67 ‥ 2.86)	cuMemAllocFromPoolAsync
1.07%	1.19 µs	2	596.05 ns ± 168.59 (476.84 ‥ 715.26)	cuStreamSynchronize

```
Device-side activity: GPU was busy for 18.84 µs (16.84% of the trace)
```

Time (%)	Total time	Calls	Name
14.07%	15.74 µs	1	**partial_mapreduce_grid(identity, add_sum, Float32, CartesianInd** …
1.71%	1.91 µs	1	partial_mapreduce_grid(identity, add_sum, Float32, CartesianInd …
1.07%	1.19 µs	1	[copy device to pageable memory] …

Let’s write a kernel

Host: Allocate memory
Launch kernel
Fetch result

```
function rmse(A, B)
    # validate inputs, allocate output
    @assert size(A) == size(B)
    C  = similar(A, 1)
    C .= 0

    # launch grid
    threads = 512
    blocks = cld(length(A), threads)
    @cuda threads blocks rmse_kernel(C, A, B)

    return sqrt(C[] / length(A))
end
```

Device: Execute kernel

```
function rmse_kernel(C, A, B)
    i = (blockIdx().x-1) * blockDim().x + threadIdx().x

    if i <= length(A)
        a = A[i]
        b = B[i]
        CUDA.@atomic C[] += (a-b)^2
    end

    return
end

@btime rmse(dA, dB)   # 5.779 ms
```

Let’s write a kernel

Writing fast GPU kernels is tricky!

```
@btime baseline(dA, dB)     # 34.120 µs
@btime rmse(dA, dB)         # 5.779 ms
```

- Minimize global memory traffic

```
@btime rmse(dA, dB)    # 115.188 µs
```

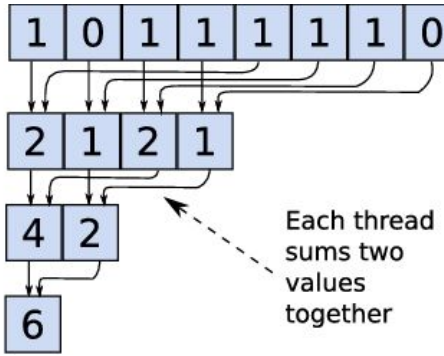
```
function rmse_kernel(C, A, B)
    i = (blockIdx().x-1) * blockDim().x + threadIdx().x
    stride = gridDim().x * blockDim().x

    # every thread processes multiple elements
    val = zero(eltype(C))
    while i <= length(A)
        a = A[i]
        b = B[i]
        val += (a-b)^2

        i += stride
    end
    CUDA.@atomic C[] += val

    return
end
```

Let’s write a kernel

Writing fast GPU kernels is tricky!

```
@btime baseline(dA, dB)     # 34.120 µs
@btime rmse(dA, dB)         # 5.779 ms
```

- Minimize global memory traffic
  ```
  @btime rmse(dA, dB)    # 115.188 µs
  ```
- Use local memory hierarchies
  ```
  @btime rmse(dA, dB)    # 29.159 µs
  ```




```
# initialize shared memory
shared = CuStaticSharedArray(eltype(C), 1024)
shared[thread] = val

# perform a parallel reduction
stride = 1
while stride < threads
    if mod1(thread, 2*stride) == 1
        shared[thread] += shared[thread+stride]
    end
    sync_threads()
    stride *= 2
end

# load the final value on the first thread
if thread == 1
    val = shared[thread]
    CUDA.@atomic C[] += val
end
```

Let’s write a *portable* kernel

```
function rmse(A, B)
   @assert size(A) == size(B)
   C = similar(A, 1)
   C .= 0

   backend = KernelAbstractions.get_backend(C)
   kernel = rmse_kernel(backend, 512)
   kernel(C, A, B; ndrange=length(A))

   return sqrt(Array(C)[] / length(A))
end

@btime rmse(dA, dB)    # 92.809 µs
```

```
@kernel function rmse_kernel(C, A, B)
   i = @index(Global, Linear)

   # KA.jl doesn't (nicely) support grid-stride loops,
   # so we only process a single item per thread.
   a = A[i]
   b = B[i]
   val = (a-b)^2

   # initialize shared memory
   thread = @index(Local, Linear)
   threads = @groupsize()[1]
   shared = @localmem eltype(C) (512,)
   shared[thread] = val

   # perform a parallel reduction

   # load the final value on the first thread
end
```

Conclusion

Julia is great for GPU programming

Arrays to get started, kernels when needed

Portability and performance at each level of abstraction

Try it out!
https://juliagpu.org/