



GPU Programming in Julia

What, why and how?

Tim Besard

What is JuliaHub?

An exciting and robust ecosystem

APPS

PLATFORM

COMMUNITY

pumas

PHARMA
SIMULATION

Julia
SIM

PHYSICAL
SIMULATION

Cedar^{EDA}

CIRCUIT
SIMULATION

MARKETPLACE

CUSTOMER
APPS

JuliaHub

Cloud Computing Platform

Secure, validated, reproducible, scalable, easy to use.
A supercomputer at your fingertips.

julia

The open source Julia ecosystem

1 million users, over 10.000 packages
Several startups are building with Julia today

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 debuginfo=:none 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



Julia: Dynamism and Performance

Reconciled by Design ([doi:10.1145/3276490](https://doi.org/10.1145/3276490))

Why Julia for GPUs?

Julia is great for GPU programming

High-level language: higher productivity than vendor toolkits

Compiled language: enables native GPU programming

NVIDIA



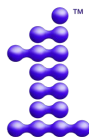
CUDA.jl

AMD



AMDGPU.jl

Intel



oneAPI.jl

Apple



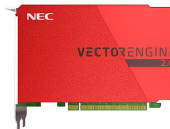
Metal.jl

GraphCore




IPUToolkit.jl

NEC



VectorEngine.jl

 Effective Extensible Programming:
Unleashing Julia on GPUs ([10.1109/TPDS.2018.2872064](https://arxiv.org/abs/10.1109/TPDS.2018.2872064))

Why Julia for GPUs?

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+ (Ventura)
- AMDGPU.jl: ROCm 5.3-5.7

```
pkg> add CUDA  
julia> using CUDA
```

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

```
CUDA runtime 12.3, artifact installation  
CUDA driver 12.3  
NVIDIA driver 545.29.6
```

```
CUDA libraries:  
- CUBLAS: 12.3.4  
- ...
```

```
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+ (Ventura)
- AMDGPU.jl: ROCm 5.3-5.7

Simplifications to get started quickly

Metal:

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

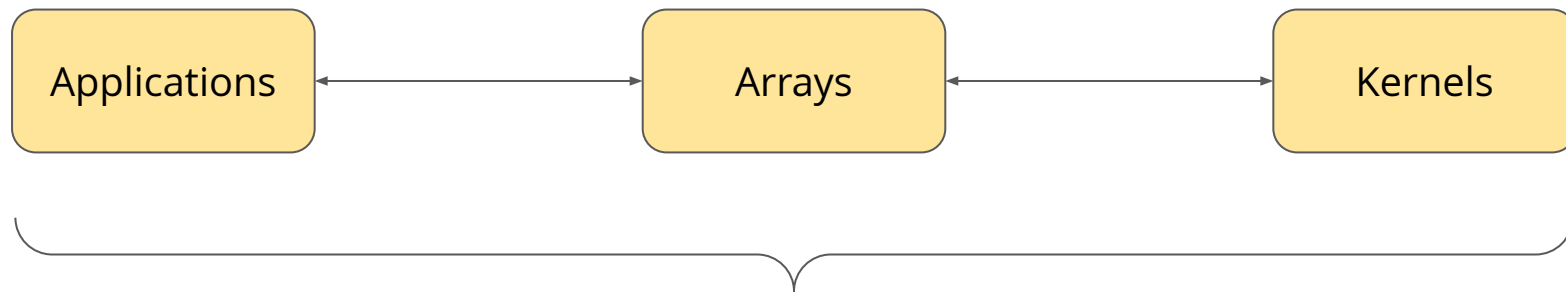
Metal.jl:

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

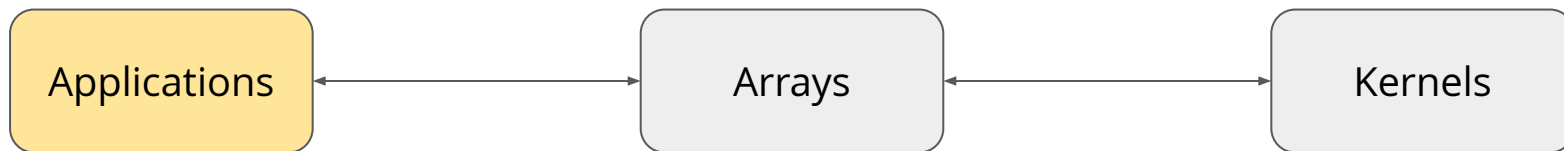
... but low-level control is still possible!

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

Multiple programming interfaces



Multiple programming interfaces

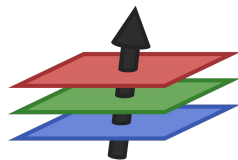


`using Flux`

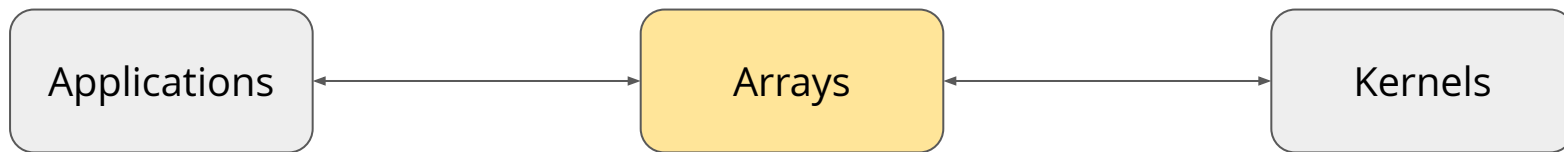
```
m = Dense(10,5) |> gpu
```

```
x = rand(10) |> gpu
```

```
m(x)
```



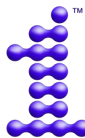
Array programming: GPUs democratized



CUDA.jl



AMDGPU.jl



oneAPI.jl



Metal.jl

```
using CUDA
```

```
a = CuArray( rand(Float32, 1024) )
```

```
sum( sin.(a) )
```

Generic code + multiple dispatch = GPU execution

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
```

```
3-element CuArray{Int64, 1}:  
 2  3  4
```

```
julia> a .+ 2b
```

```
3-element CuArray{Int64, 1}:  
 9 12 15
```

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

```
6
```

```
julia> accumulate(+, b)
```

```
3-element CuArray{Int64, 1}:  
 4  9 15
```

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

```
2
```

Array programming is powerful

using LinearAlgebra

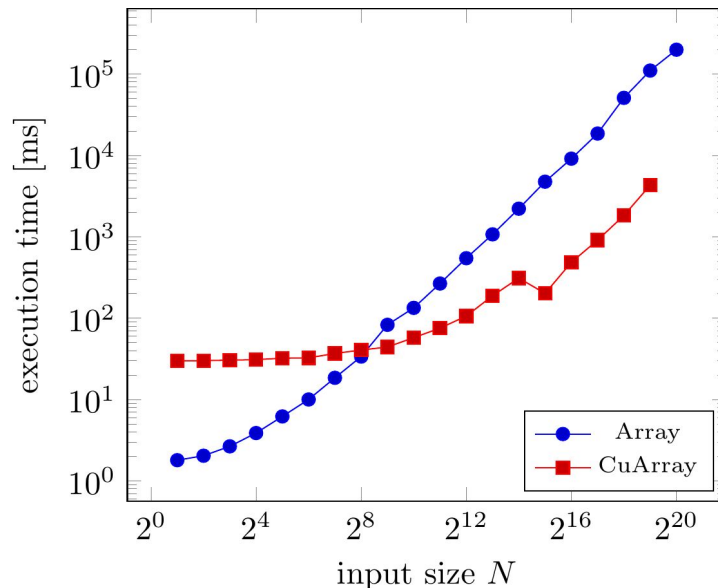
```
loss(w,b,x,y) = sum(abs2, y - (w*x .+ b)) / size(y,2)
loss∇w(w, b, x, y) = ...
lossdb(w, b, x, y) = ...
```

```
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
```

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

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



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, noint);
```

```
    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

User-friendly interfaces where possible

```
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

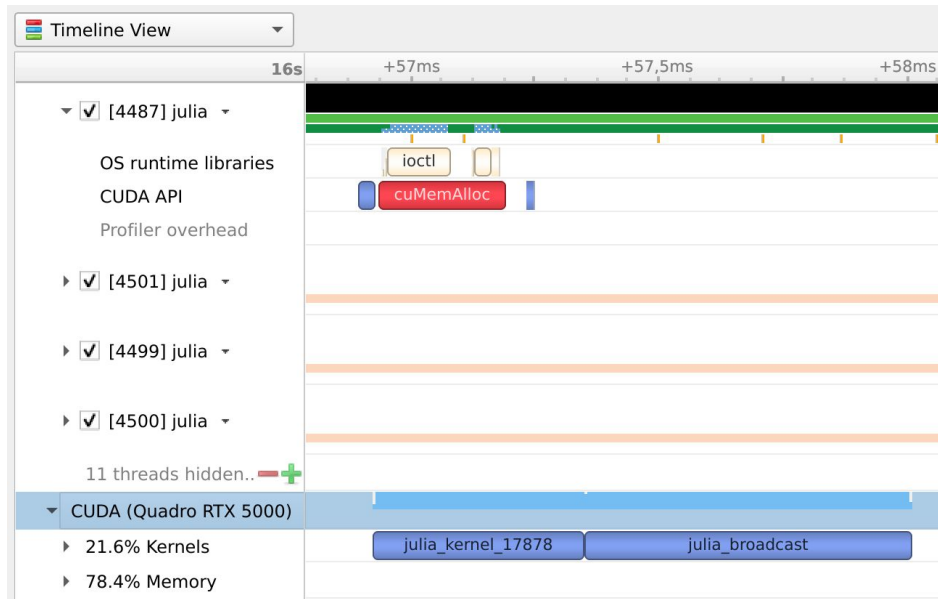
Device-side activity: GPU was busy for 422.72 μs (89.05% of the trace)

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

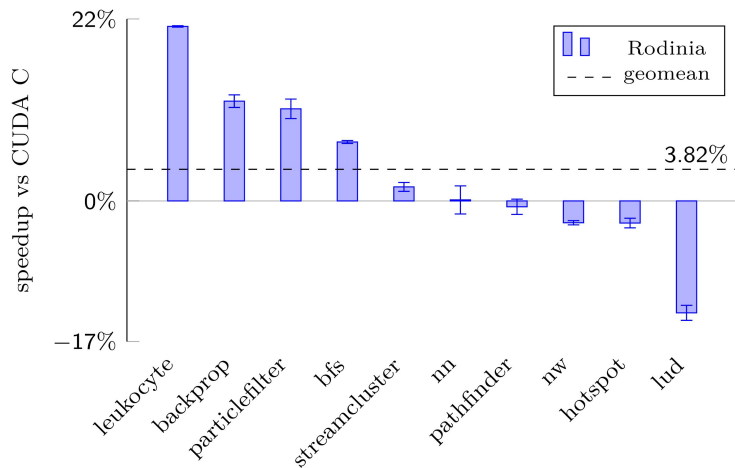
Benchmarking and profiling

Integration with vendor tools

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



Julia on GPUs performs great!



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



Effective Extensible Programming: Unleashing Julia on GPUs ([10.1109/TPDS.2018.2872064](https://arxiv.org/abs/10.1109/TPDS.2018.2872064))

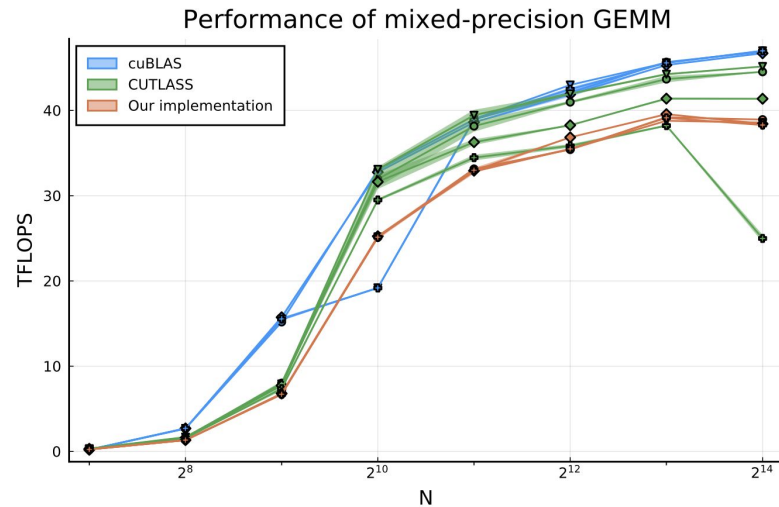
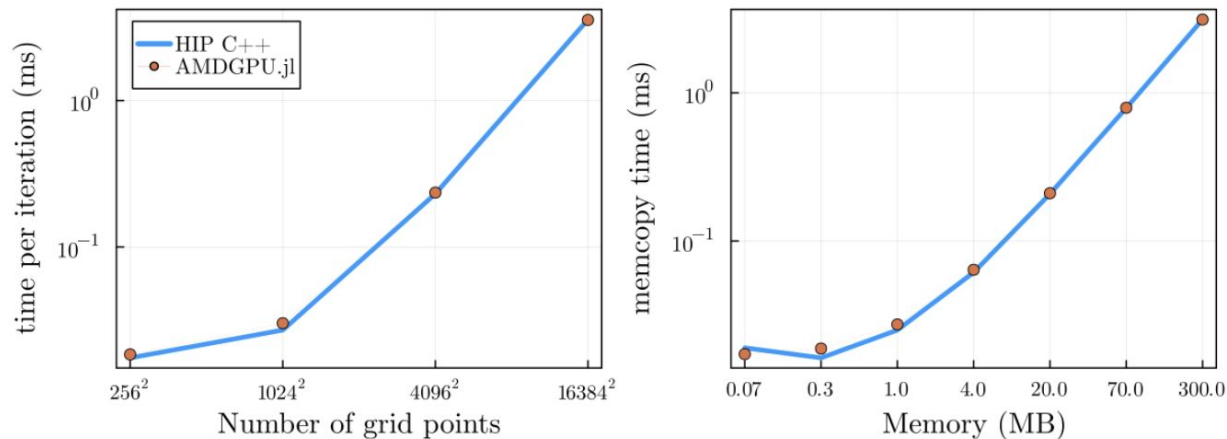


Figure 11. Performance of our mixed-precision GEMM and the state-of-the-art implementation in CUTLASS and cuBLAS.



Flexible Performant GEMM Kernels on GPUs ([10.1109/TPDS.2021.3136457](https://arxiv.org/abs/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

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](https://doi.org/10.1016/j.cma.2023.116591))

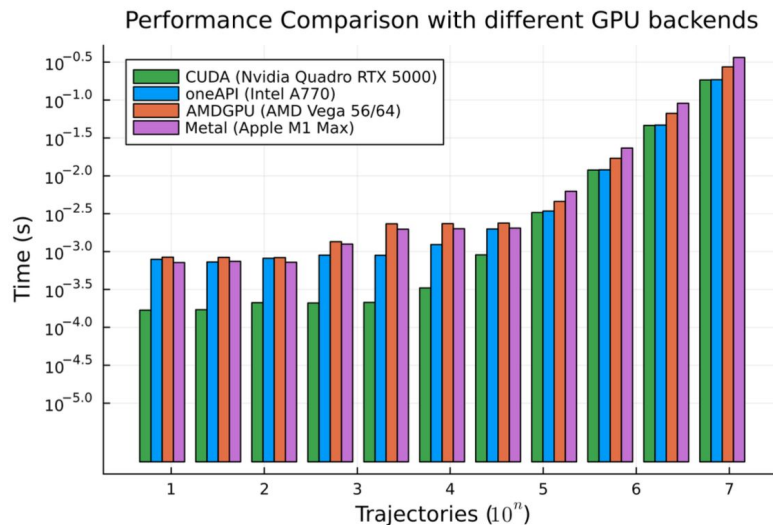


Figure 7: A comparison of ODE solve timings with fixed time-stepping, measured on different GPU platforms. We measure the time (*lower the better*) versus number of parallel solves. Here, the NVIDIA GPUs perform the best owing to the most-optimized library and matured ecosystem with JuliaGPU.

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](https://arxiv.org/abs/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](https://arxiv.org/abs/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](https://doi.org/10.1016/j.cma.2023.116591))

WaterLily.jl: GPU-accelerated fluid simulator

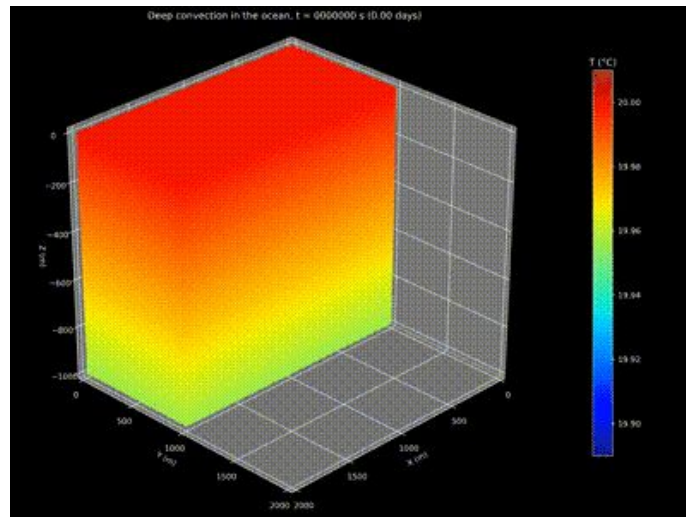


WaterLily.jl: A differentiable fluid simulator in Julia with fast heterogeneous execution
([arXiv:2304.08159](https://arxiv.org/abs/2304.08159))

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](https://arxiv.org/abs/2309.06662))



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/>