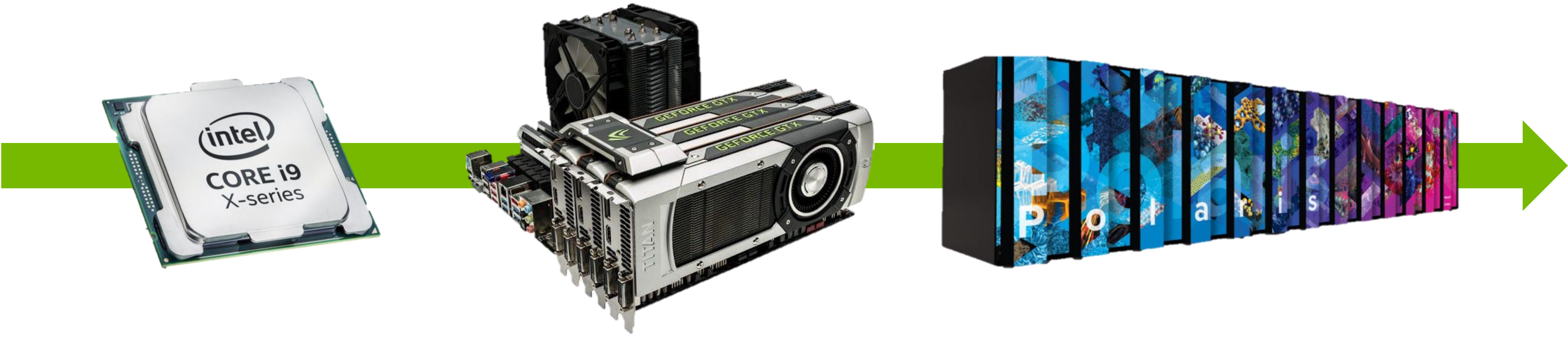
cuNumeric.jl : Automating Distributed Numerical Computing

Ethan Meitz[1], **David Krasowska**[2], Wonchan Lee[3], Pat McCormick[4]

[1]Carnegie Mellon University
[2]Northwestern University
[3]NVIDIA
[4]Los Alamos National Laboratory

The Goal: Scale with Zero Code Changes

- "Easy" to implement the correct physics in a high-level language like Python, Julia, or MATLAB
- Time consuming to modify code to scale across multiple CPUs/GPUs
  - Need to learn and debug new technologies like OpenMPI, CUDA etc.



Code that runs on a single CPU core should also be able to run on multiple cores, multiple GPUs and across multiple nodes

cuNumeric.jl: Distributed Code with Minimal Effort

- Core Type: **NDArray**
  - Drop-in replacement for Base.Array but can represent data across multiple devices
  - Key Differences: Operations are broadcast by default, slices are always views, avoid scalar indexing

```
(3.8.0) (base) david@dubliner:~/julia-con/slides-examples/diff$ cat diff_grayscott.txt
1,2d0
< using cuNumeric
<
81,84c79,82
<     u = cuNumeric.ones(dims)
<     v = cuNumeric.zeros(dims)
<     u_new = cuNumeric.zeros(dims)
<     v_new = cuNumeric.zeros(dims)
---
>     u = ones(dims)
>     v = zeros(dims)
>     u_new = zeros(dims)
>     v_new = zeros(dims)
86,87c84,85
<     u[1:150, 1:150] = cuNumeric.random(FT, (150, 150))
<     v[1:150, 1:150] = cuNumeric.random(FT, (150, 150))
---
>     u[1:150, 1:150] = random(FT, (150, 150))
>     v[1:150, 1:150] = random(FT, (150, 150))
(3.8.0) (base) david@dubliner:~/julia-con/slides-examples/diff$
```

7 LOC changed for 2D Gray Scott Reaction-Diffusion simulation

Legate Enables Composable and Distributed Libraries

By providing data and task management abstractions, this enables the efficient implementation of complex library APIs.

NumPy | SciPy | pandas | scikit learn | dmlc XGBoost | HDF | . . .

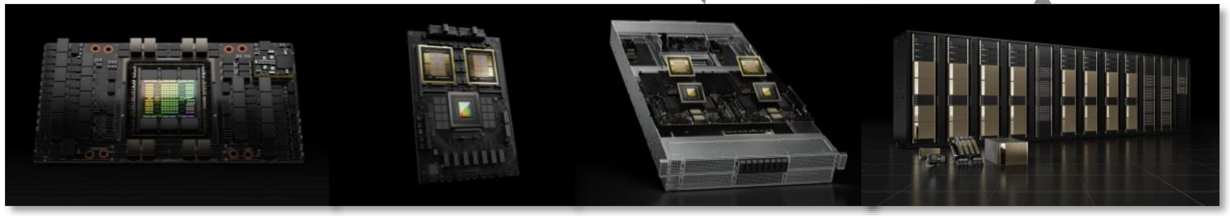
cuPyNumeric	Legate Sparse	Legate Dataframe	Legate RAFT	Legate Boost	Legate IO

Legate Framework

Legate Core - Productivity & Composability Layer

Legion - Implicit Parallelism Layer

Realm - Runtime for Scalable and Portable Execution



Single GPU | Mixed CPU/GPU | Single-Node Multi-GPU | Multi-Node Multi-GPU Cloud & Supercomputer

Wouldn't this be great in Julia?

https://developer.nvidia.com/legate
https://legion.stanford.edu/

How Legate Works

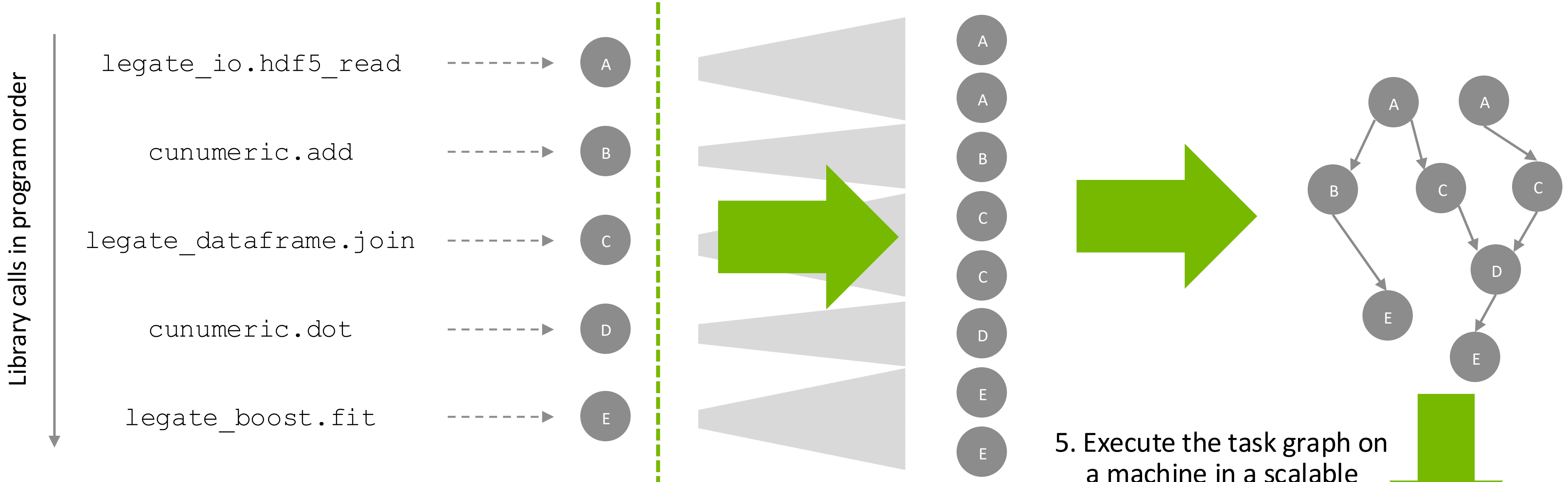
Implicit parallelism via “scale-free” tasking

1. Legate program makes API calls

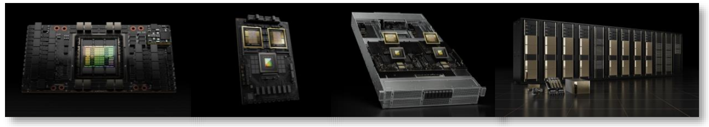
2. Legate libraries issue **“scale-free” tasks**

3. Convert each scale-free task to parallel tasks

4. Analyze data dependencies and constructs a task graph



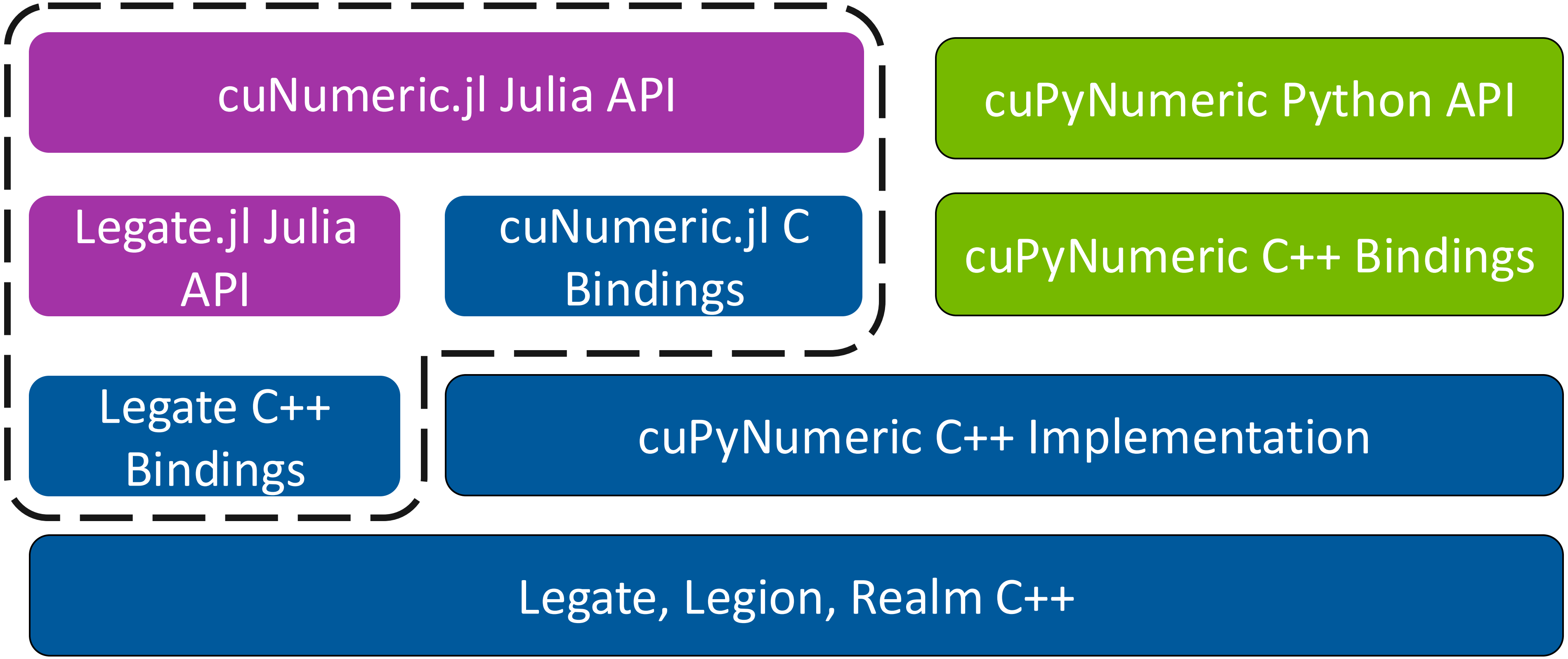
5. Execute the task graph on a machine in a scalable manner



Legate Framework

Legate libraries are **free of any explicit parallelization or synchronization/data movement**, making them **composable and transparently scalable** by construction

Software Stack



Generates parallel tasks

Determines data dependencies and orders task execution

Executes tasks on target hardware

New Contributions

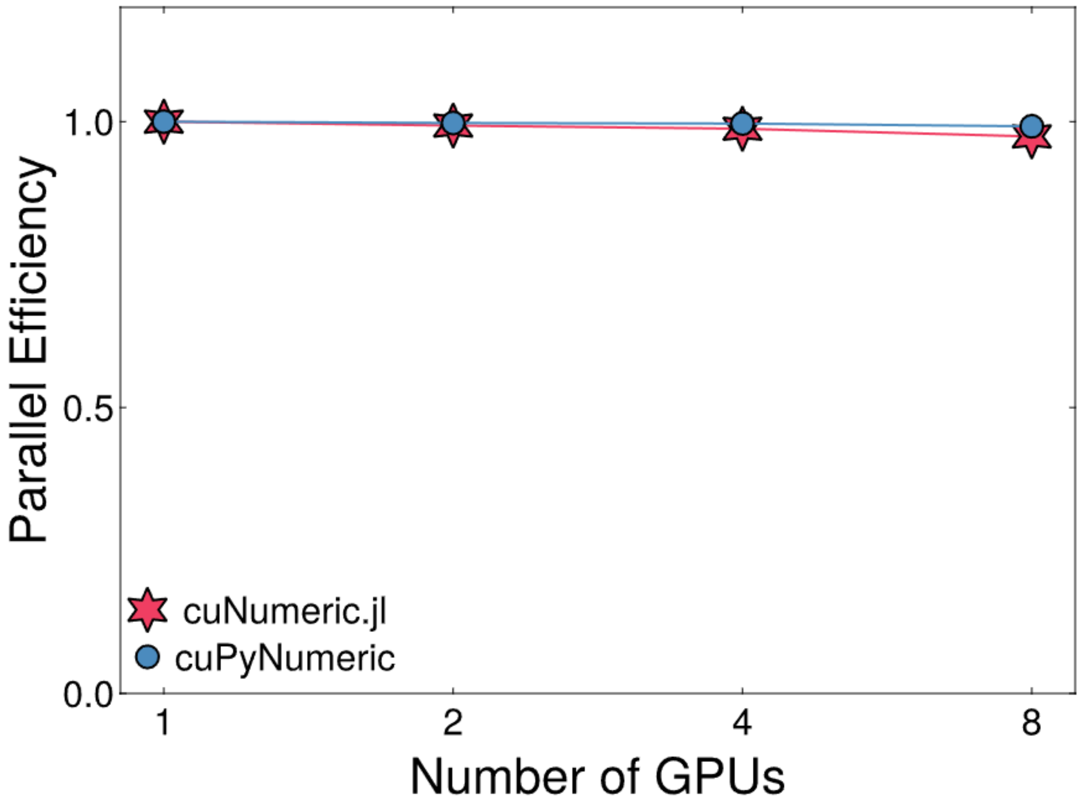
Monte Carlo Integration

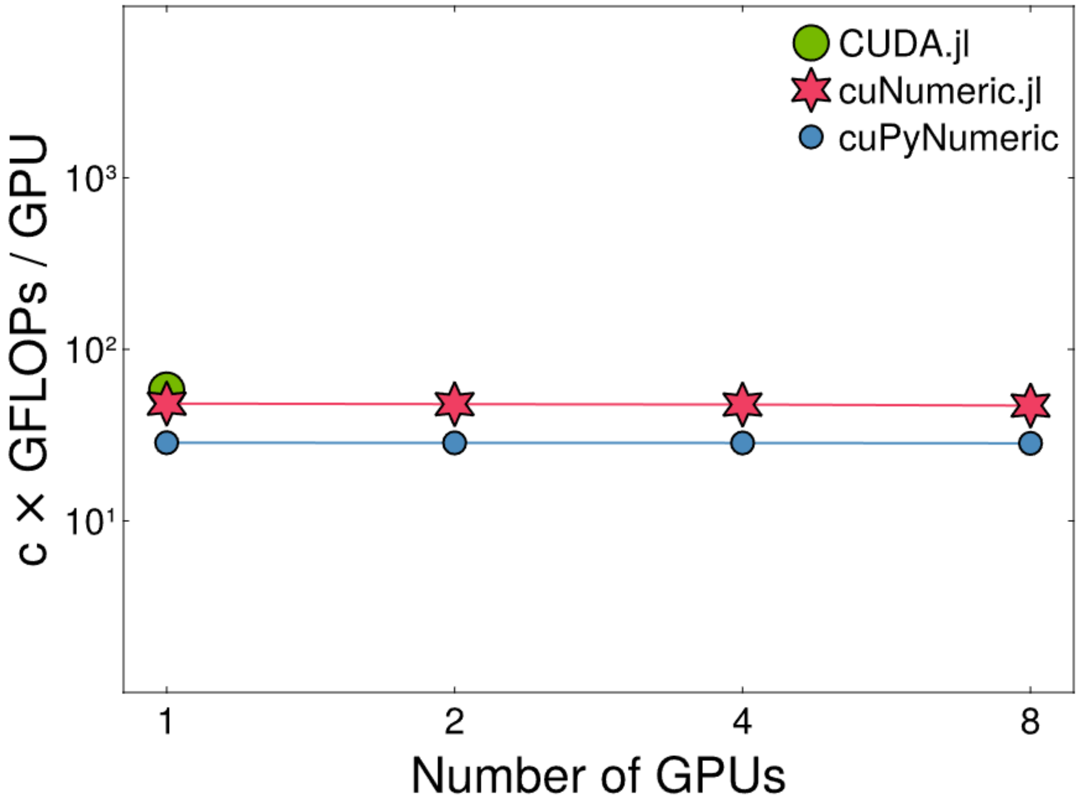
- Embarrassingly Parallel → Should scale perfectly
- cuNumeric.jl directly calls C++, cuPyNumeric passes through several layers of Python before C++

Syntax:

```
integrand = (x) -> exp(-square(x))
N = 1_000_000
x_max = 5.0
domain = [-x_max, x_max]
Ω = domain[2] - domain[1]
samples = Ω*cuNumeric.rand(NDArray, N) - x_max
estimate = (Ω/N) * sum(integrand(samples))
```

Benchmark (8x A100):





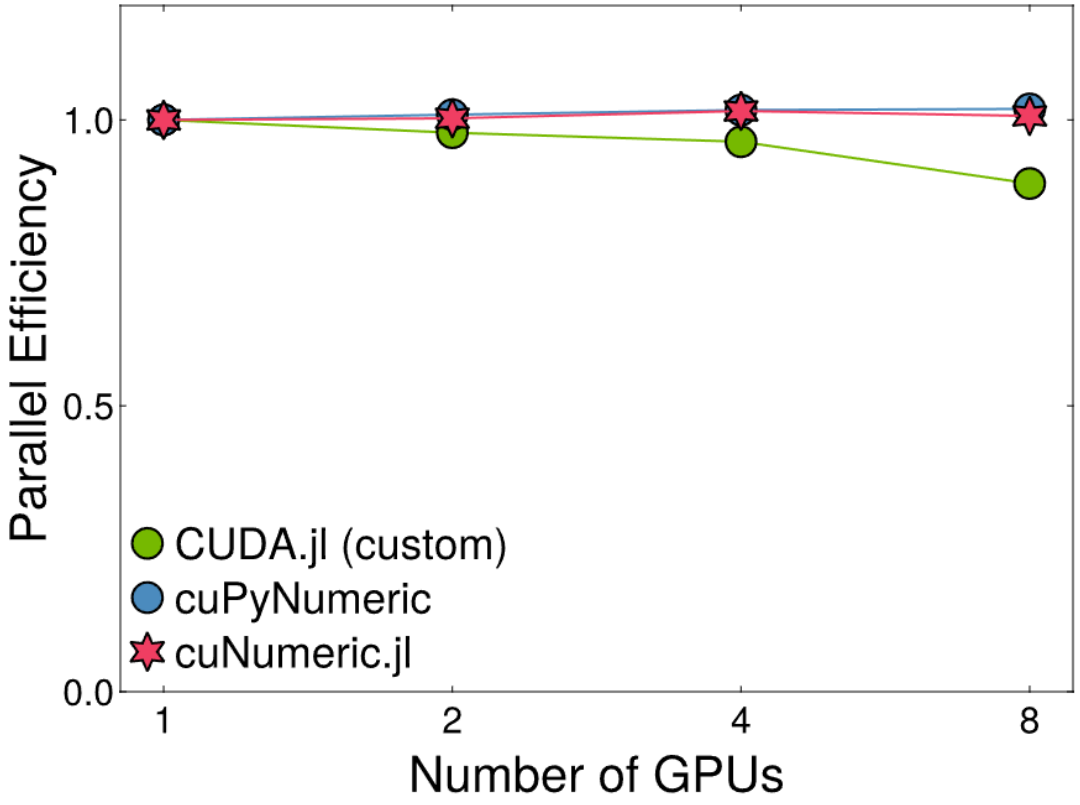
*Actual FLOPs unknown, off by constant factor (c < 1.0)

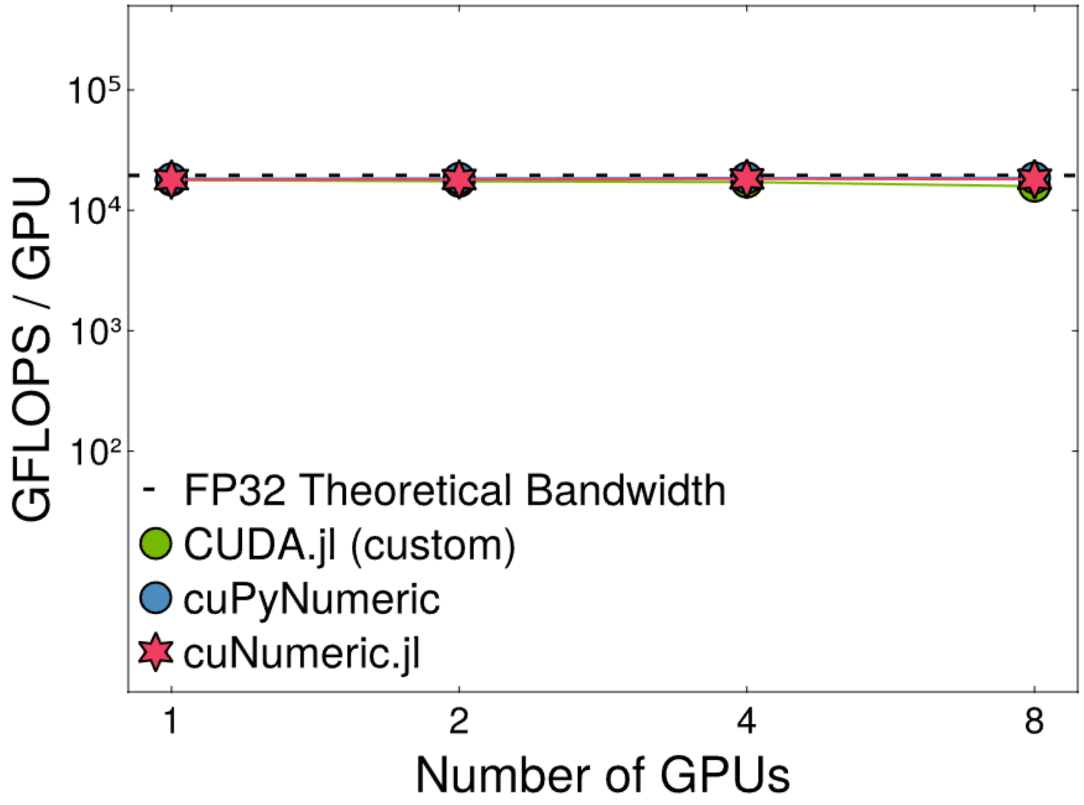
Matrix Multiplication

Syntax*:

```
N = 10
A = cuNumeric.rand(Float32, N, N)
B = cuNumeric.rand(Float32, N, N)
C = cuNumeric.zeros(Float32, N, N)
mul!(C, A, B)
```

Benchmark (8x A100):





*Float32 random number generation not supported yet, requires type cast

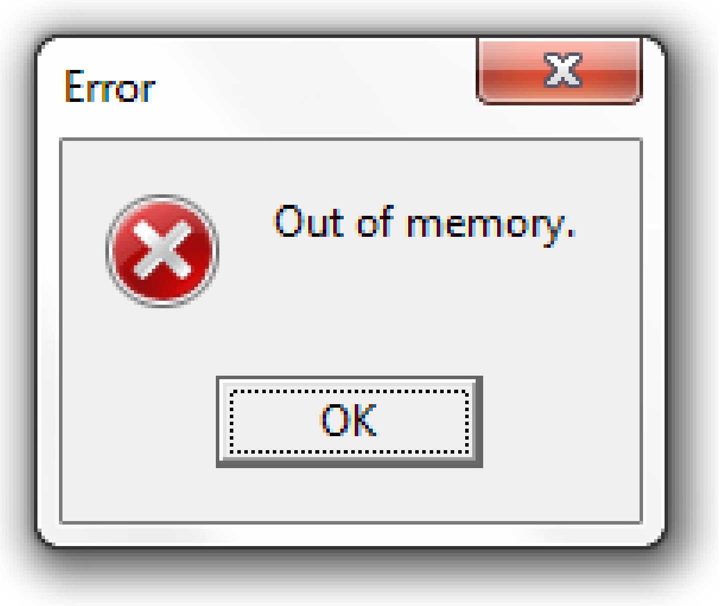
Gray Scott Reaction Diffusion (2D)

Syntax:

```julia
N = 100
u = cuNumeric.ones((N, N))
v = cuNumeric.ones((N, N))
for i in iters
    F_u = ((u[2:(end - 1), 2:(end - 1)] .* (v[2:(end - 1), 2:(end - 1)]
        .* v[2:(end - 1), 2:(end - 1)])) - (f+k)*v[2:(end - 1), 2:(end - 1)])
    #.... more physics
    #.... update u and v
    GC.gc()
end
```

Manual GC allows Gray Scott to run

Benchmark (8x A100):



Each object, including slices, are treated as a temporary and remain uncollected.
Why are they not collected before OOM?

Foreign Memory is not automatically garbage collected

- GC sees foreign memory as a pointer → 8 bytes
- GC relies on the heap size (of Julia objects) to decide when GC is invoked

```julia
using CUDA, Profile

arr = CUDA.ones(Float32, 10_000_000)
Profile.take_heap_snapshot()
Base.summarysize(arr)
```

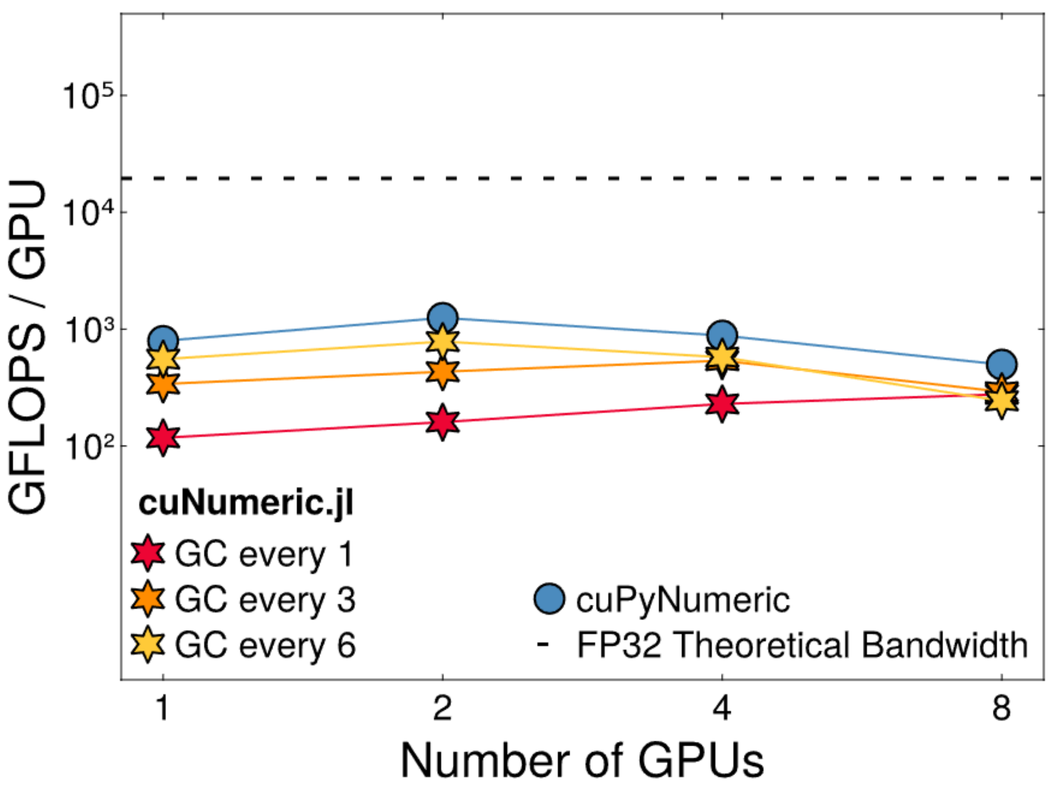
~40MB

Julia sees GPU data as 184 bytes

Constructor	Retained Size
▼ CUDA.CuArray{Float32, 1, CUDA.Devi	0.2 kB
▼ data :: GPUArrays.DataRef{CUDA.M	0.1 kB
▼ rc :: GPUArrays.RefCounted{CUD	0.1 kB
▼ obj :: CUDA.Managed{CUDA.Dev	0.1 kB
stream :: CUDA.CuStream @4	0.0 kB
count :: Base.Threads.Atomic	0.0 kB
finalizer :: typeof(CUDA.poo	0.0 kB

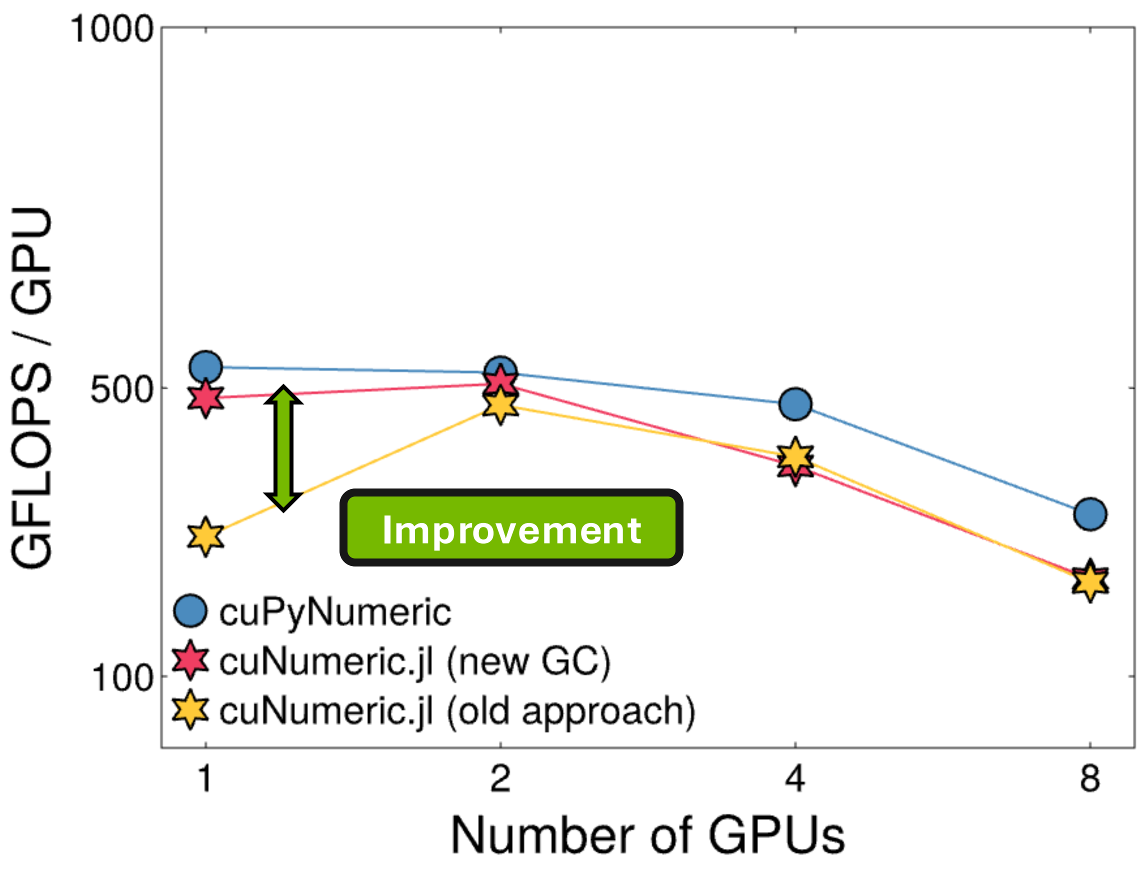
184 bytes in heap snapshot too

Garbage Collection heavily impacts performance



- All existing GPU backends in Julia:
  - Calculate array memory footprint in the constructor
  - Track GPU memory allocations
  - Manually calls GC based on a memory pressure heuristic
  - try-catch on allocation to avoid OOM
- NDArray Properties:
  - Deferred execution model conceals Julia object's physical size at creation
  - Runtime cannot recover from failed mapping allocation

Custom heuristic GC improves performance




```
N = 100
u = cuNumeric.ones((N, N))
v = cuNumeric.ones((N, N))
for i in iters
    F_u = ((u[2:(end - 1), 2:(end - 1)] .* (v[2:(end - 1), 2:(end - 1)]
        .* v[2:(end - 1), 2:(end - 1)])) - (f+k)*v[2:(end - 1), 2:(end - 1)])
    #.... more physics
    #.... update u and v
end
```

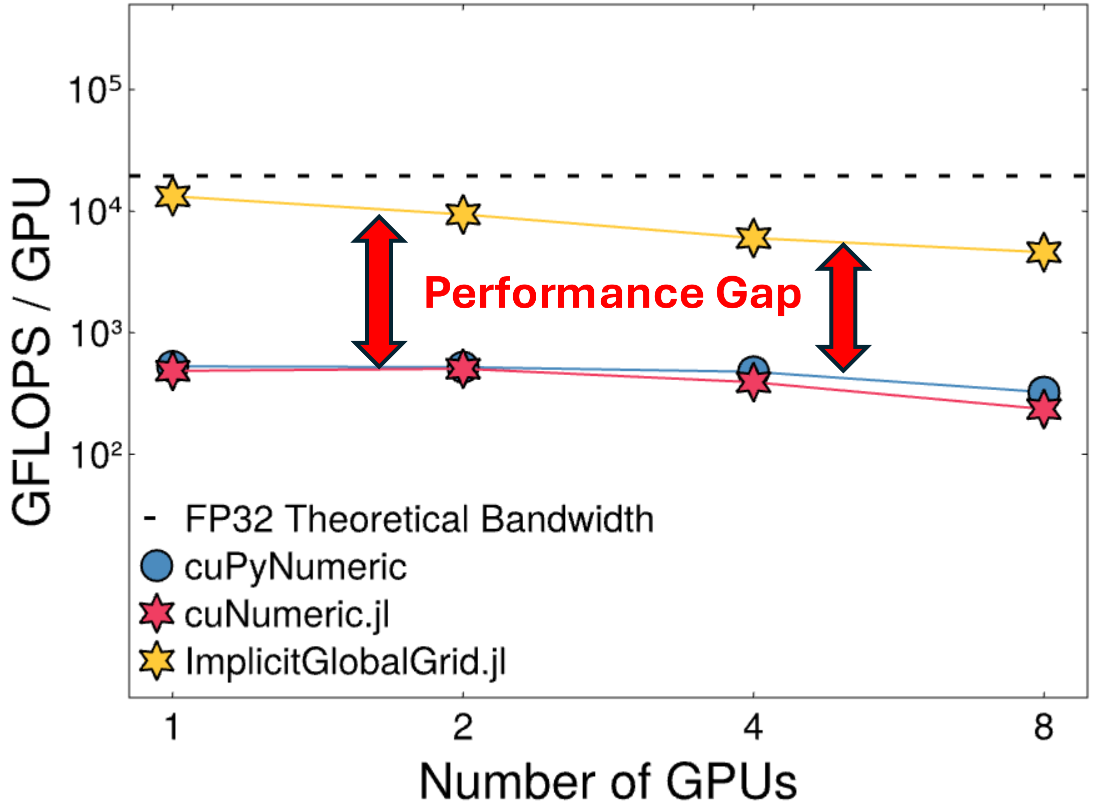
Automatic GC allows Gray Scott to run

- We are still exploring solutions to achieve better scalability

Unfused Operations Bottleneck Performance

ImplicitGlobalGrid.jl:

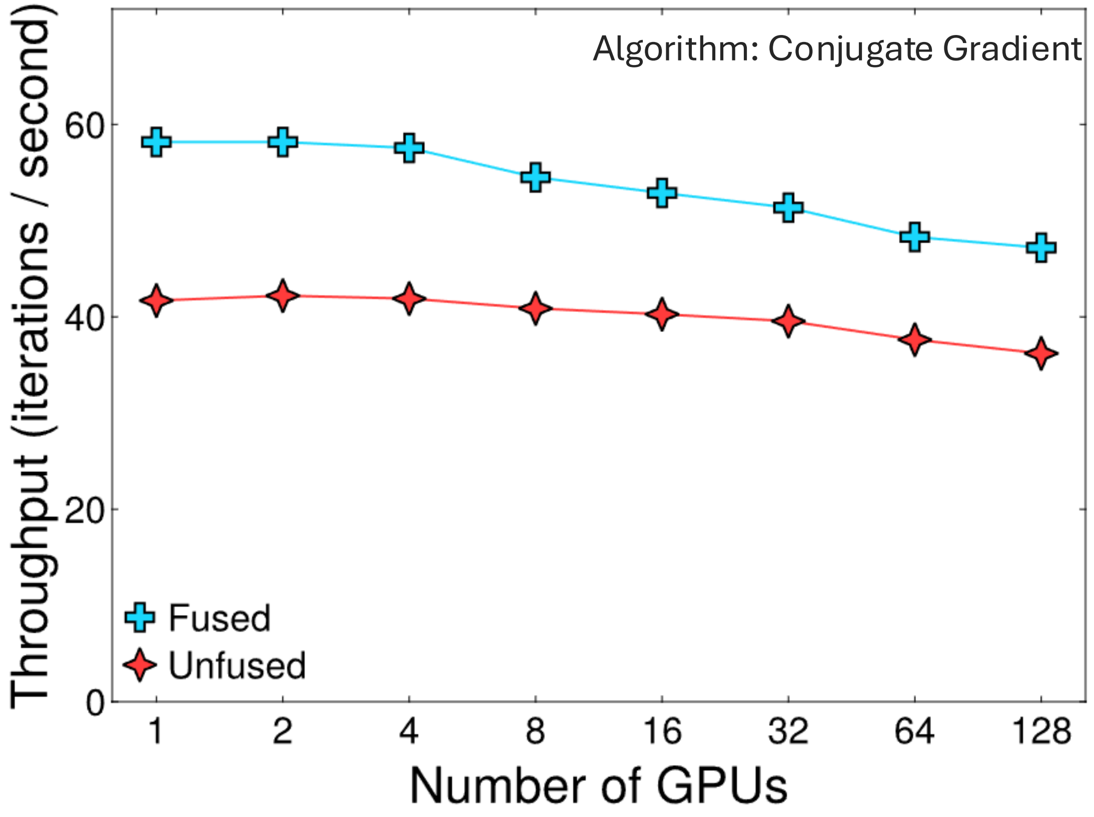
- Stencil-based PDE solver
- Multi-node CPU and GPU (NVIDIA & AMD)
- Minimal code changes



Räss, L., Omlin, S., & Podladchikov, Y. Y. *JuliaCon* 2019.

Kernel Fusion:

- Under development in cuPyNumeric, but shown to provide 2x speed-up on average and up to 10x
- With CUDA.jl and Legate.jl we propose to generate fused, distributed kernels



Rohan Yadav, https://doi.org/10.1145/3669940.3707216, *ASPLOS* 2025

Multi-GPU CUDA Kernels with cuNumeric.jl

- To enable kernel fusion, we need to be able to run CUDA.jl kernels through Legate.jl

```julia
using cuNumeric
using CUDA

function kernel_add(a, b, c, N)
    i = (blockIdx().x - 1i32) * blockDim().x + threadIdx().x
    if i <= N
        @inbounds c[i] = a[i] + b[i]
    end
    return nothing
end


N = 1024
threads = 256
blocks = cld(N, threads)

a = cuNumeric.full(N, 1.0f0)
b = cuNumeric.full(N, 2.0f0)
c = cuNumeric.ones(Float32, N)

task = cuNumeric.@cuda_task kernel_add(a, b, c, UInt32(1))

cuNumeric.@launch task=task threads=threads blocks=blocks \
                  inputs=(a, b) outputs=c scalars=UInt32(N)
```

CUDA.jl kernel

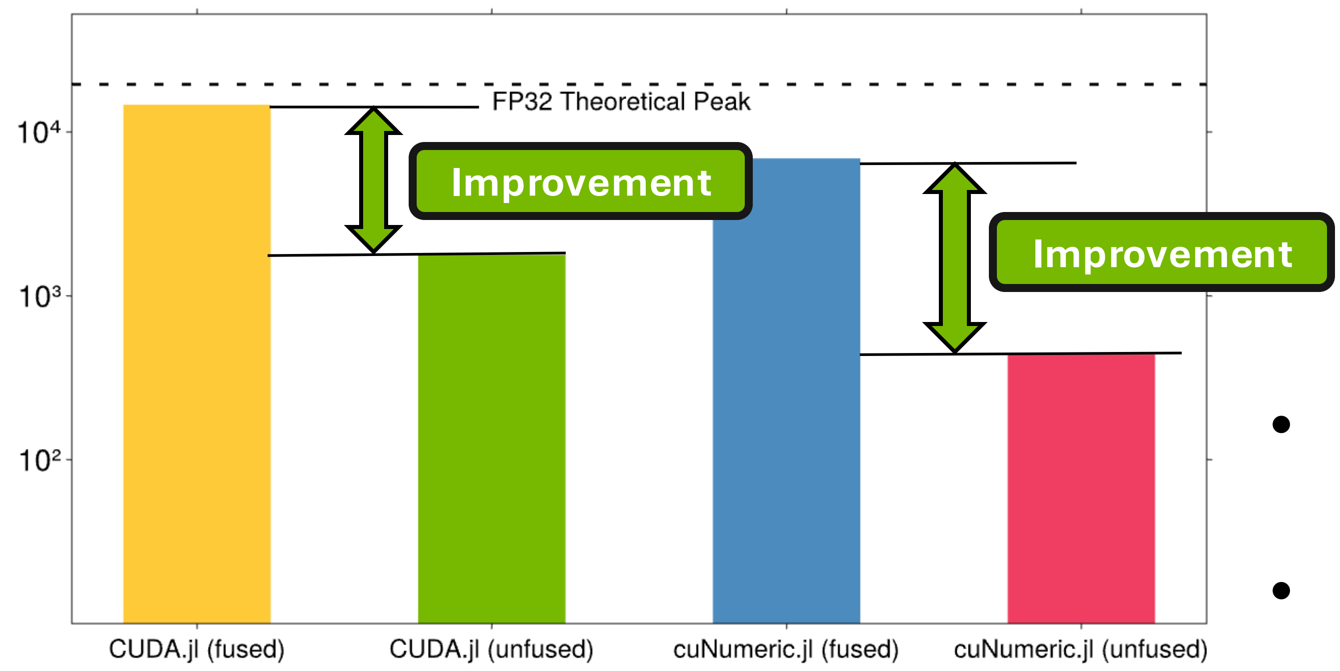
Initialize NDArrays

Compile

Launch

Gray Scott mini 1D with custom CUDA.jl kernels

Benchmark (1x A30x):



Syntax:

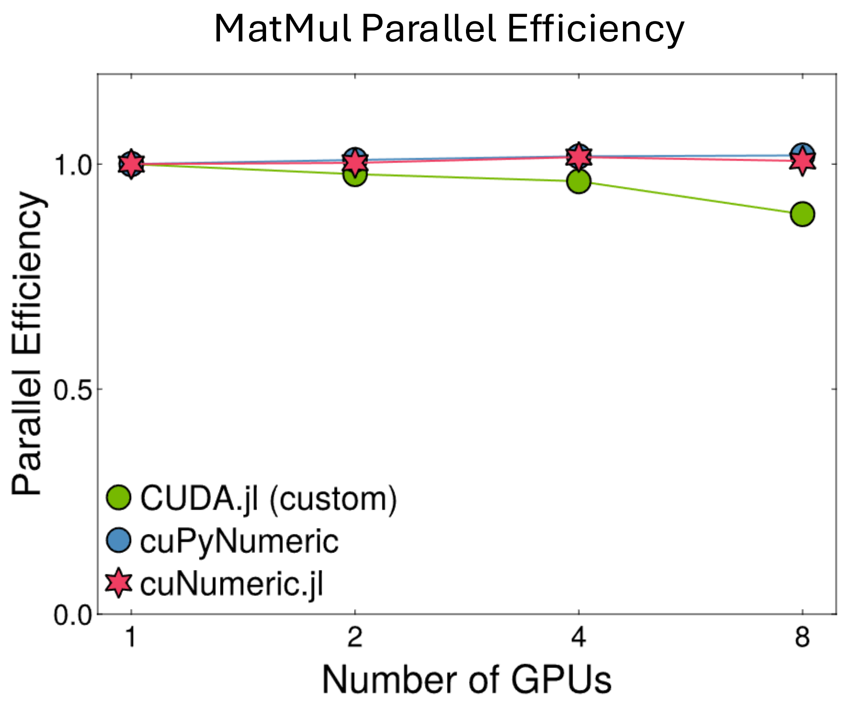
```
function fused_kernel(u, v, F_u, F_v, N::UInt32, f::Float32, k::Float32)
    i = (blockIdx().x - 1i32) * blockDim().x + threadIdx().x
    if i <= (N*N-2)
        @inbounds begin
            u_ij = u[i + 1]
            v_ij = v[i + 1]
            v_sq = v_ij * v_ij
            F_u[i] = (-u_ij * v_sq) + f*(1.0f0 - u_ij)
            F_v[i] = (u_ij * v_sq) - (f + k)*v_ij
        end
    end

    return nothing
end
```

- Operator invocation overhead limits performance scalability
- Ongoing backend enhancements for multi-dimensional workloads and multi-GPU execution
- Backend overhead is amortized as GPU count increases

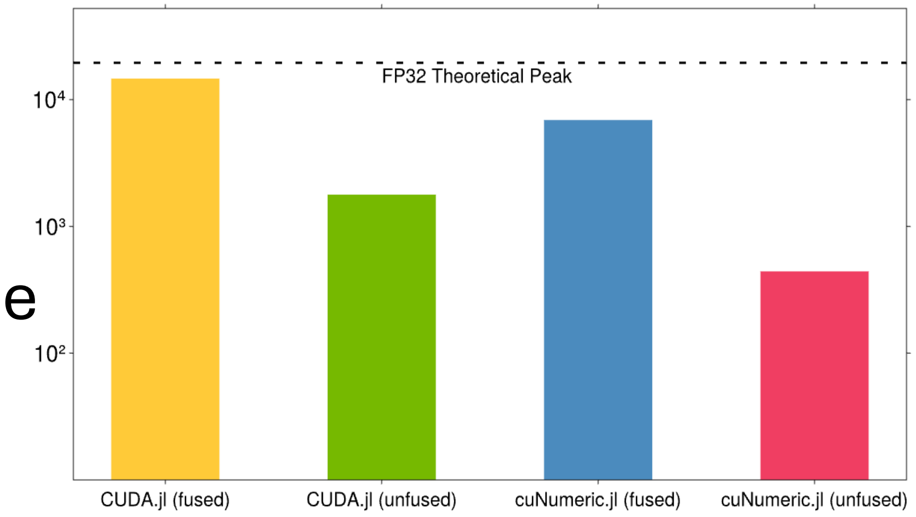
Conclusions and Future Work

- Minimal code changes for scaling across large heterogeneous distributed systems
- Good weak scaling efficiency on diverse applications
- Ability to register custom CUDA kernels



Next Steps

1. Support a wider range of custom CUDA kernels
2. Improve robustness and accessibility of package installation
3. Enhance integration with Julia Abstraction interface + Legate
4. Benchmark on multi-node systems
5. Better GC heuristics



Aiming for September* beta release

Ethan Meitz emeitz@andrew.cmu.edu

David Krasowska krasow@u.northwestern.edu

Wonchan Lee wonchanl@nvidia.com

Pat McCormick pat@lanl.gov

*this is an estimate; we will register the package upon beta launch

Check out our repo
https://github.com/JuliaLegate/cuNumeric.jl/

This material is based upon work supported by the U.S. Department of Energy, National Nuclear Security Administration, Office of Science, Office of Advanced Scientific Computing Research, Department of Energy Computational Science Graduate Fellowship under Award Number(s) DE-SC0024386.