What Can Julia Bring to Dense Linear Algebra and Scientific Computation?

Rabab Alomairy

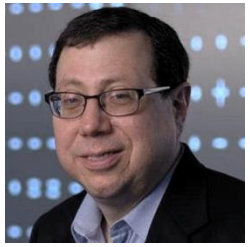
Massachusetts Institute of Technology

9th May 2025

Outline

- Motivation
- Overview of Julia For HPC
  - Kernel Programming
  - KernelAbstractions.jl
  - GPUArrays.jl
  - Dagger as Asynchronous Many-Task Runtime (AMT)
- Julia for Dense Linear Algebra
  - SVD Computations
  - Julia LAPACK/BLAS Operations
  - Julia Unified TRMM/TRSM Recursive Subdivision
- Conclusion and Future Directions

Team



Alan Edelman
JuliaLab PI

Rabab Alomairy
Postdoc

Julian Samaroo
RSE

Evelyne Ringoot
PhD student

Felipe Tome
Visiting researcher

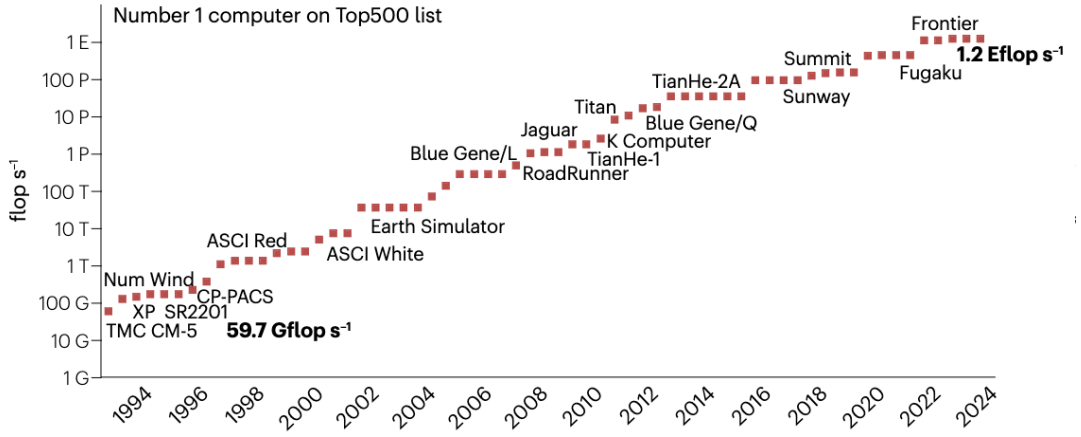
Vicki Carrica
UROP student

Maxwell Onyango
UROP student

Sophie Xuan
UROP student

Exascale introduces significant complexities

- Three decades of top-ranked systems [1]

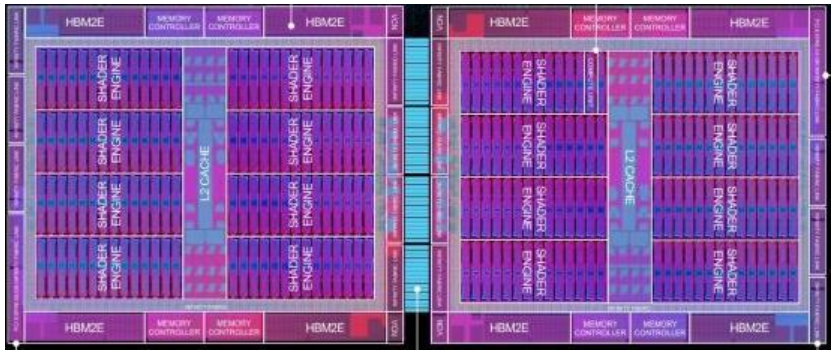


- Challenges: concurrency, memory hierarchies, and heterogeneity

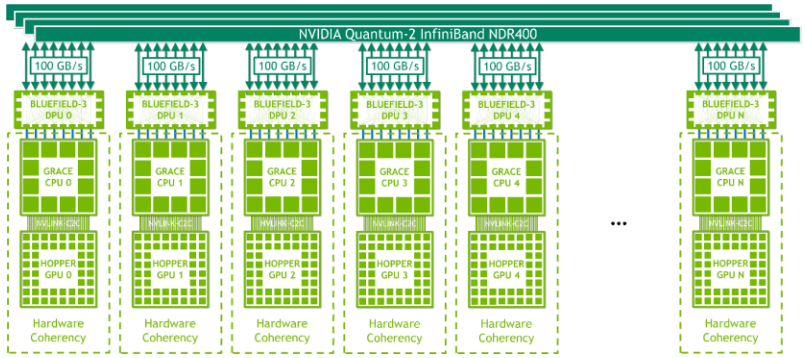
- Algorithms, tools, libraries, languages need to adapt to accordingly

- Half-precision executes at nearly 1 Pflop/s

AMD Instinct MI300X Accelerator



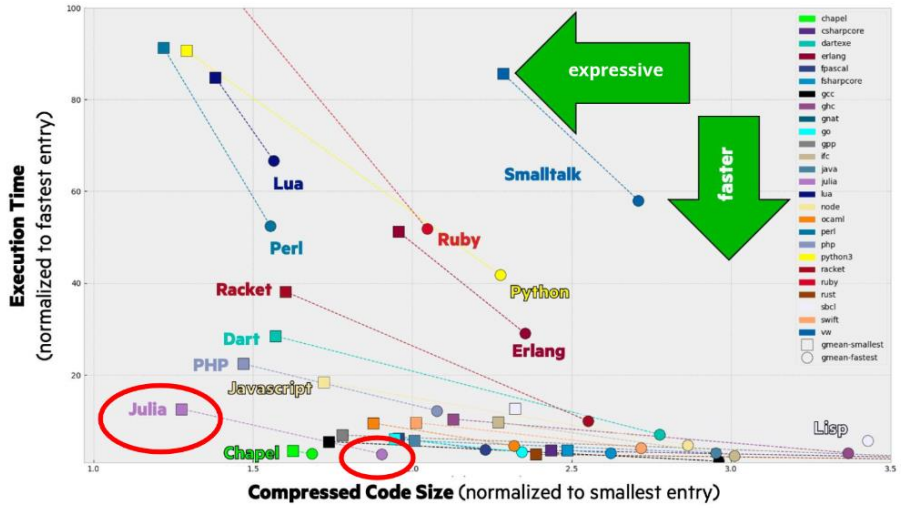
NVIDIA HGX Grace Hopper



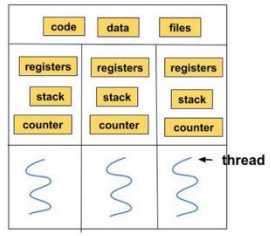
[1] Dongarra, Jack, and David Keyes. "The co-evolution of computational physics and high-performance computing." *Nature Reviews Physics* (2024).

julia: Driving Adaptability

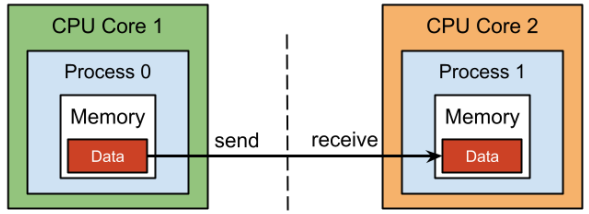
- Julia is easy, expressive, and robust
- Julia leverages the LLVM compiler
- Julia supports dynamic type inference
- Julia's multiple dispatch makes it composable
- Julia supports various parallel computing paradigms
- Julia's GPU ecosystem is very rich

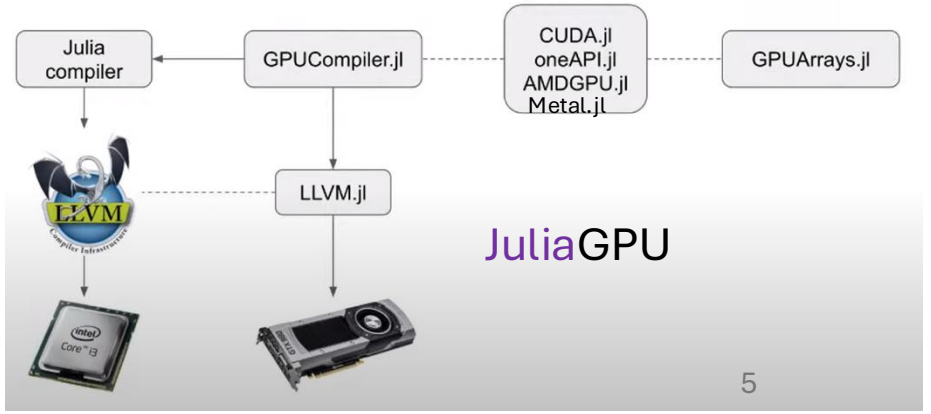


Julia Multithreading



Julia Distributed

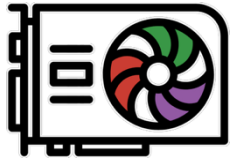




Overview of Julia For HPC

Performance, Portability, and Productivity

Julia for HPC

Accelerators 	Shared Memory	Distributed Computing
 CUDA.jl  AMDGPU.jl  OneAPI.jl  Metal.jl	**Threads.jl** Static and dynamic threads	**DistributedNext.jl** Create and control multiple Julia remote processes.
GPUArrays.jl Easy array programming Reusable GPU array functionality for Julia's various GPU backends.	**Floops.jl** Generate a fast generic sequential and parallel for iteration	**MPI.jl** Julia interface to the Message Passing Interface (MPI)
KernelAbstractions.jl GPU-like kernels targeting CPU and different execution backends		**Dagger.jl** Julia native asynchronous many-task runtime

Vendor-Specific Kernel Programming

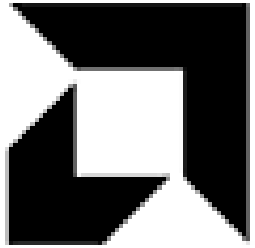
CUDA

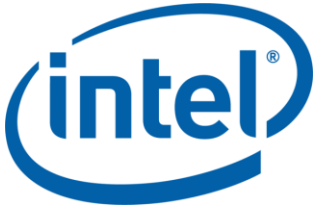
HIP

SYCL

Metal

```
id<MTLBuffer> a = [device newBufferWithLength:512*4];
memcpy(a.contents, ..., 512*4);
id<MTLBuffer> b = [device newBufferWithLength:512*4];
memcpy(b.contents, ..., 512*4);
id<MTLBuffer> c = [device newBufferWithLength:512*4];
kernel void vadd(device float *a,
                 device float *b,
                 device float *c,
                 uint i [[thread_position_in_grid]]) {
    c[i] = a[i] + b[i];
}
```





Vendor-Specific Kernel Programming in Julia

CUDA.jl

AMDGPU.jl

OneAPI.jl

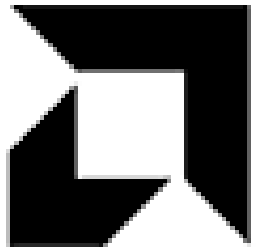
Metal.jl

```
using Metal

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

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

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

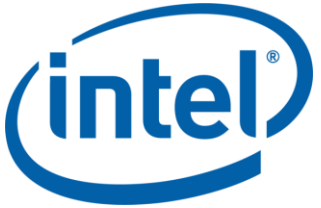
Julia Packages

CUDA.jl

AMDGPU.jl



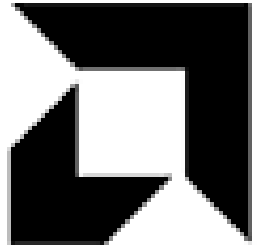
OneAPI.jl

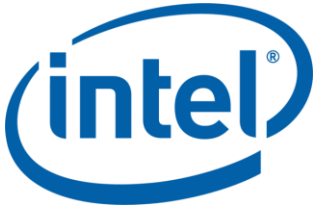


Metal.jl

KernelAbstractions.jl: Macro-based Kernel API

```julia
# Choose your backend: CUDA, AMDGPU, oneAPI, Metal
using KernelAbstractions
using CUDA  # or: AMDGPU, oneAPI, Metal
const backend = CUDABackend()  # or: ROCBackend(), OneAPIBackend(),
MetalBackend()

N = 512
a = KernelAbstractions.allocate(backend, rand(Float32, N))
b = KernelAbstractions.allocate(backend, rand(Float32, N))
c = KernelAbstractions.allocate(backend, Float32, N)

@kernel function vadd_kernel(a, b, c)
    i = @index(Global)
    c[i] = a[i] + b[i]
end

# Launch with N threads (1D)
kernel_instance = vadd_kernel(backend, ndrange = N)
kernel_instance(a, b, c; ndrange = N)
```





GPUArrays.jl



CUDA.jl AMDGPU.jl OneAPI.jl Metal.jl

Generic type agnostic code, except importing packages or allocating arrays.

```
using Metal
T= Float32
a = MtlArray(rand(T, 32,32))
b = sum(a)
Array(b)
```

Compatible with Julia's array functions

```
view(a, 2:4, 2:4)
sin.(a)
a*2
a'
```

Vendor library integration

rand.jl

```
rand!(a)
```

cuRAND, rocRAND, ...

LinearAlgebra.jl

```
a*a
qr!(a)
```

cuBlas, rocBlas, ...
cuSOLVER, rocSOLVER, ...

AbstractFFTs.jl

```
plan_fft(a) * a
```

cuFFT, rocFFT, ...

NNlib.jl

```
softmax(a)
```

cuDNN, hipDNN, ...

Julia native support

Compiled to native code

```
map(a) do x
       x+1
end
```

Kernel support for broadcasting and operation fusion

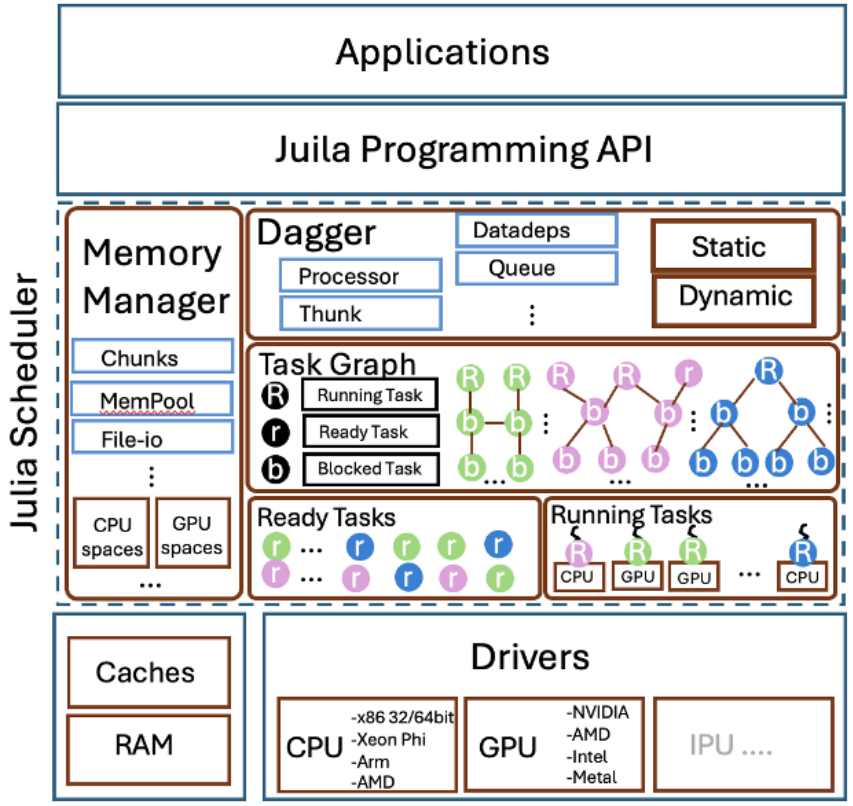
```
a .+ 2b
```

More functionality, flexibility, and portability

```
reduce(+, a)
accumulate(+, a; dims=2)
findfirst(isequal(2), a)
```

Dagger as Asynchronous Many-Task Runtime (AMT)

- Julia is a productive environment for designing native task-based asynchronous runtime systems
- Dagger manages tasks across heterogeneous hardware
  - CPU, GPU, ...etc
- Dagger builds a task execution flow (DAG)
- Dagger handles memory spaces for CPUs and GPUs
- Uses DArray as a basic block for managing data movement
- Uses DTask objects to schedule tasks on hardware at runtime, based on resource availability



Julia for Dense Linear Algebra

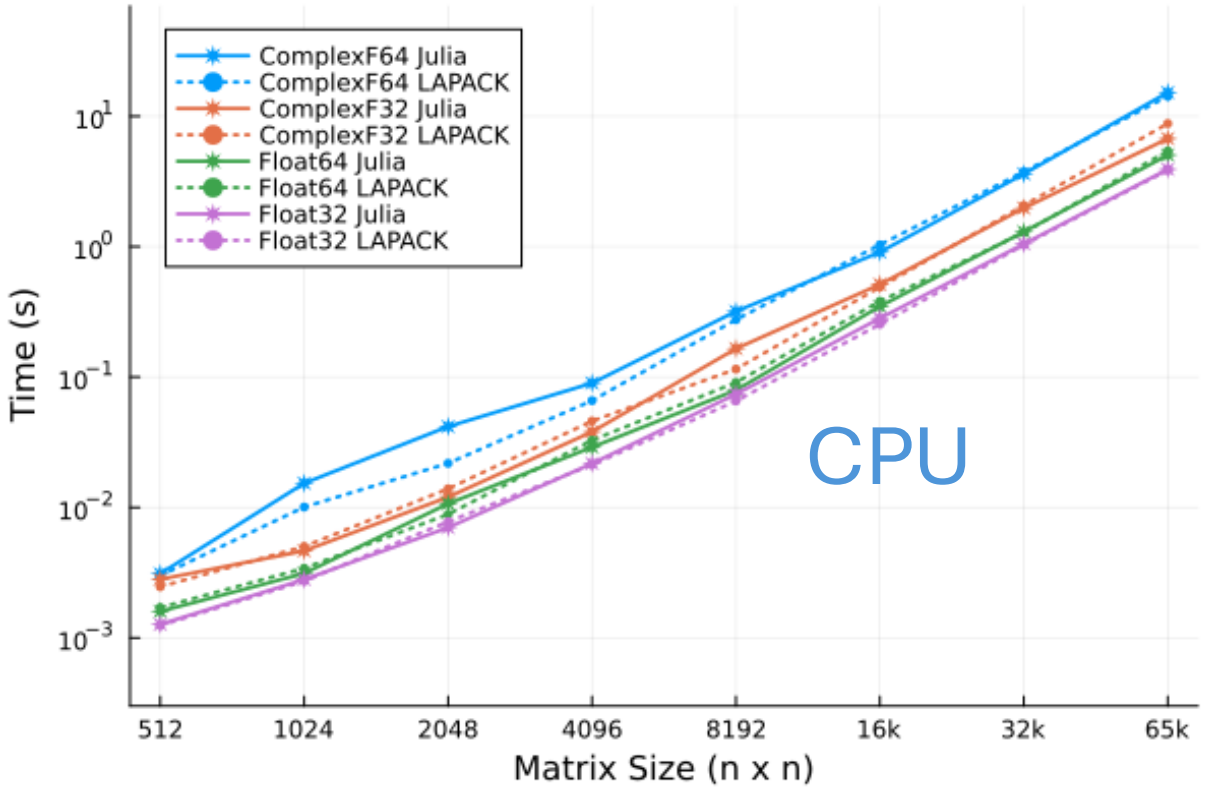
One Kernel to Rule Them All

Julia LAPACK/BLAS Operations using GPUArrays.jl

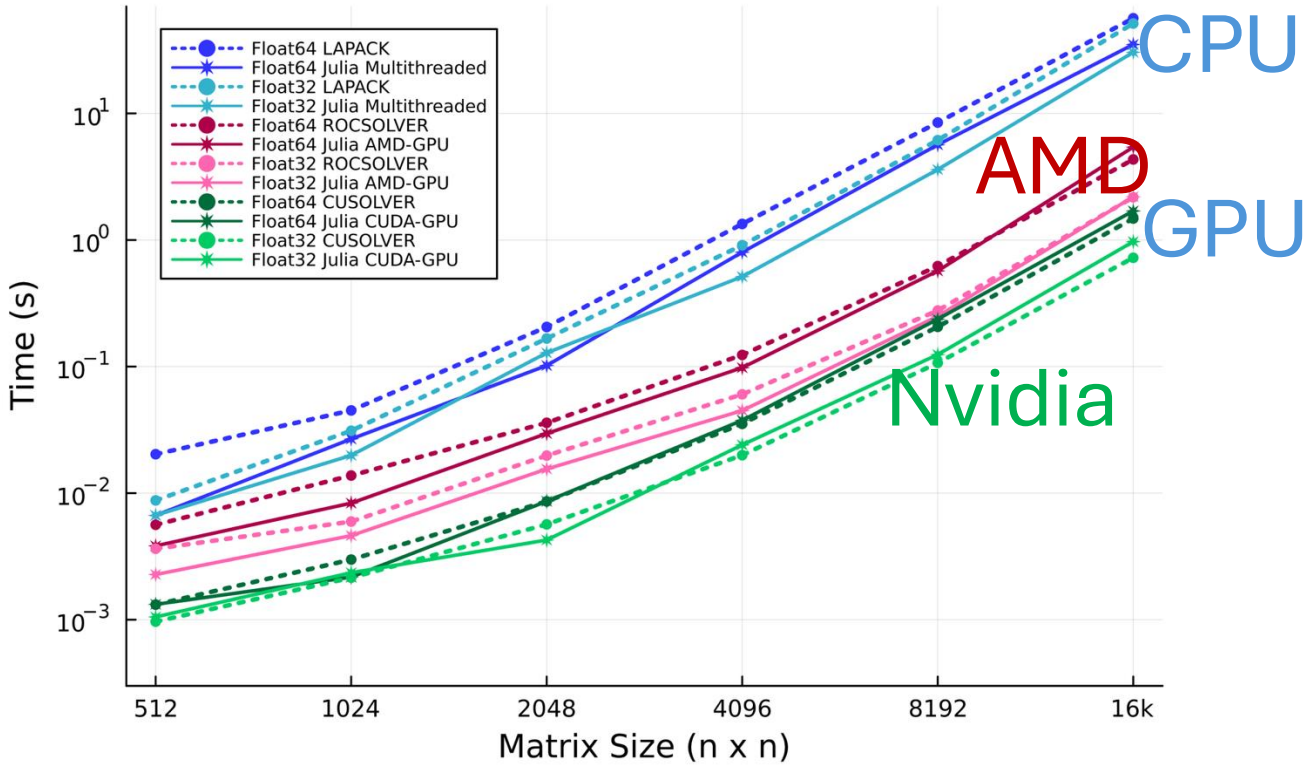
Larfb: $A + AYTY^T = AQ$

urmqr: $A + YTY^T A = QA$

Data type agnostic: performance across data types



Hardware agnostic: Performance across CPU-GPU



Intel IceLacke with A100, Intel CascadeLake with A100, AMD Milan with MI100

(Xuan, Alomairy, et.al., 2024)
HPEC24: Best paper award

SVD Computations using JuliaGPU

- SVD based on Two-stage QR factorization

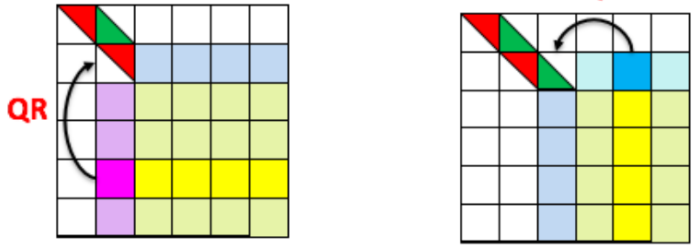
Algorithm 1 SVD stage 1: reduction to band-diagonal form [44]

1: **for** Diagonal tile $k : 1 \rightarrow$ N **do**
2: GEQRT($\text{Tile}_{k,k}$) {Calculate Householder QR}
3: UNMQR(Row_k) {Apply Householder QR}
4: **for** Row l below k : k+1 $\rightarrow$ N **do**
5: TSQRT($\text{Tile}_{k,k}$, $\text{Tile}_{l,k}$) {Calculate Householder QR}
6: TSMQR(Row_k, Row_l) {Apply Householder QR}
7: **end for**
8: GEQRT($\text{Tile}^T_{k,k+1}$) {Calculate Householder QR}
9: UNMQRT(Col^T_{k+1}) {Apply Householder QR}
10: **for** Col l right of $k+1$: k+2 $\rightarrow$ N **do**
11: TSQRT($\text{Tile}^T_{k,k+1}$, $\text{Tile}^T_{k,l}$) {Calculate Householder QR}
12: TSMQR(Col^T_{k+1}, Col^T_l) {Apply Householder QR}
13: **end for**
14: **end for**

QR

LQ

SVD

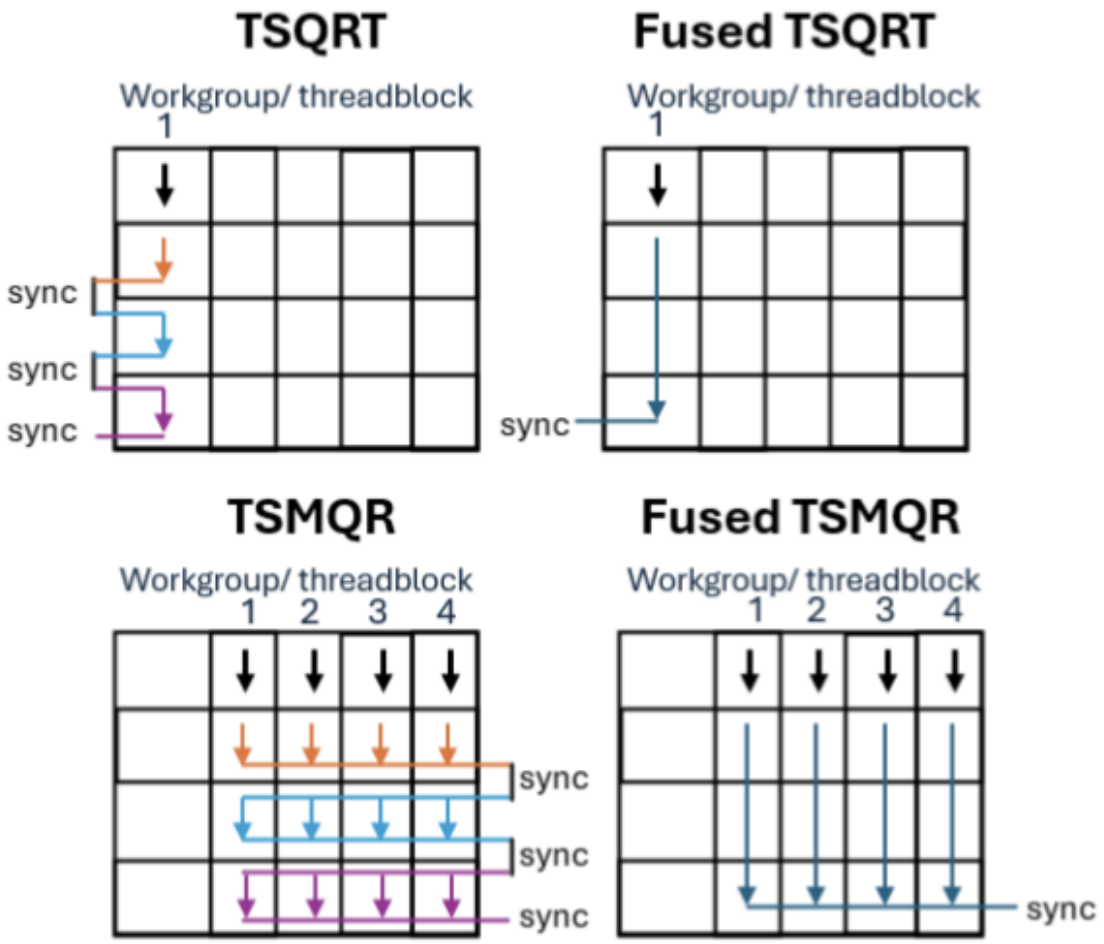


(Kabir,e.a., 2017)

Multiple Dispatch
and
Lazy Transpose Operator

(Ringoot, Alomairy, et.al., 2025)
Submitted to ICPP

Fused TSQRT and TSMQR



(Ringoot, Alomairy, et.al., 2025)
Submitted to ICPP

Fused TSMQR using KernelAbstractions.jl

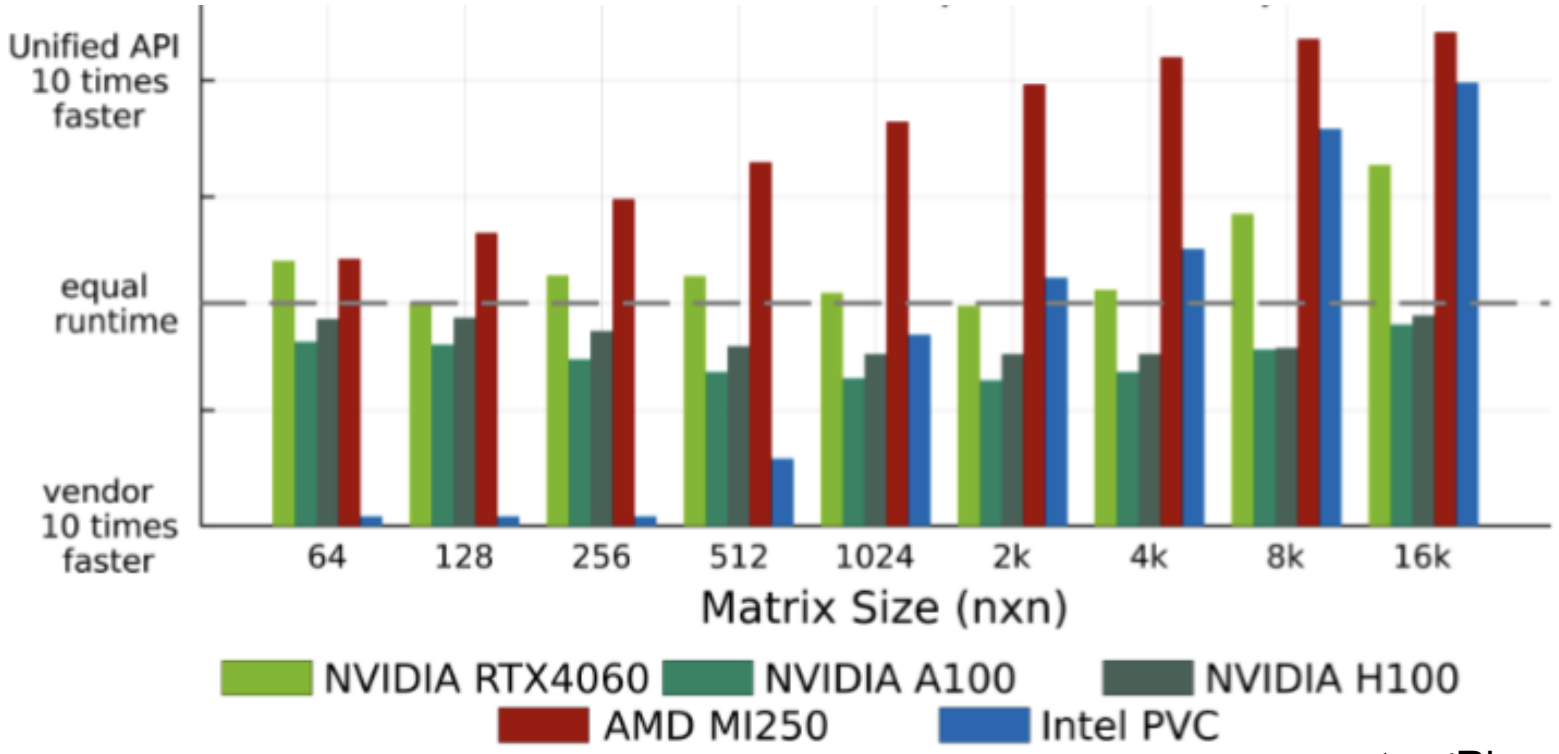
```
@kernel function TSMQR_fused_kernel!(A, B, @Const(M),
                                     @Const(Tau), nbrows::Int)
    g = @index(Group, Linear)
    i = @index(Local, Linear)
    colB = @private eltype(A) (TILESIZE)
    colA = @private eltype(A) (TILESIZE)
    colM= @localmem eltype(A) (TILESIZE)
    colTau = @localmem eltype(A) (TILESIZE)
    @unroll for l in 1:TILESIZE
        colA[l] = A[...]
    end
    for row in 1:nbtiles
        @unroll for l in 1:TILESIZE
            colB[l] = B[...]
        end
        @unroll for j in 0:(TILESIZE/MULSIZE)-1
            colTau[j*MULSIZE+i]=tau[...]
        end
 for k in 1:TILESIZE
            @unroll for j in 0:(TILESIZE/MULSIZE)-1
                colM[j*MULSIZE+i]=M[...]
            end
```

```
@synchronize
            tmp_sum= zero(eltype(A))
            @unroll for l in 1:TILESIZE
                tmp_sum += colM[l] * colB[l]
            end
            tmp_sum= (tmp_sum+colA[k])* colTau[k]
            colA[k] -= tmp_sum
            @unroll for l in 1:TILESIZE
                colB[l] -= tmp_sum * colM[l]
            end
            @synchronize
        end
        @unroll for l in 1:TILESIZE
            B[...] = colB[l]
        end
    end
    @unroll for l in 1:TILESIZE
        A[...] = colA[l]
    end
end
```

(Ringoot, Alomairy, et.al., 2025)
Submitted to ICPP

Performance Compared to Vendor-Optimized Libraries

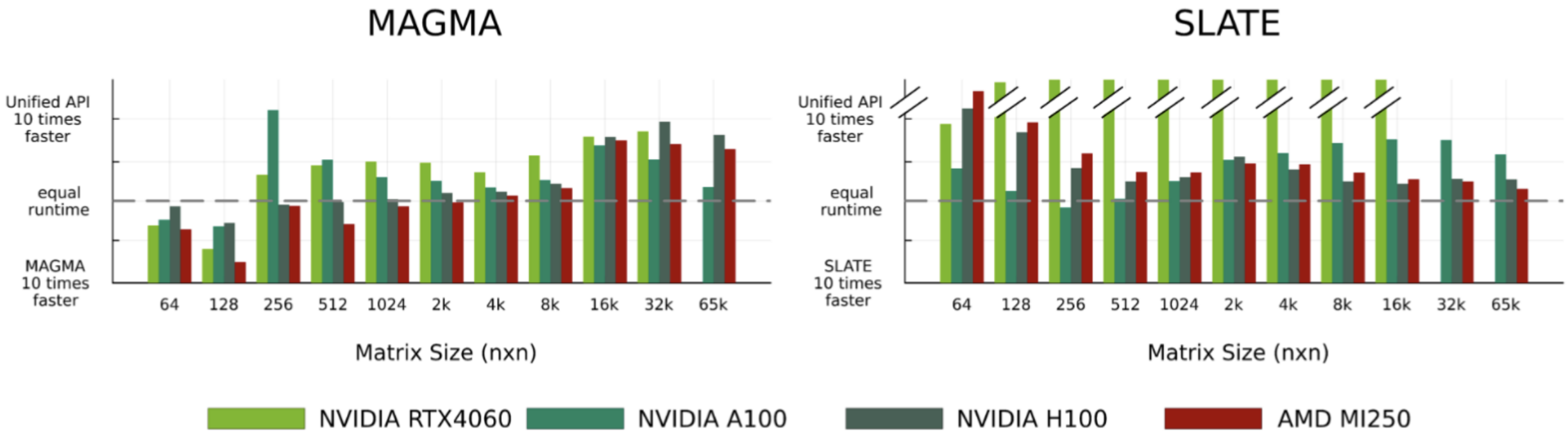
- Performance of singuler value computation
- Ratio of Julia unified implementation to cuSLOVER, rocSOLVER, and oneMKL



(Ringoot, Alomairy, et.al., 2025)
Submitted to ICPP

Performance Compared to State-of-the-Art Libraries

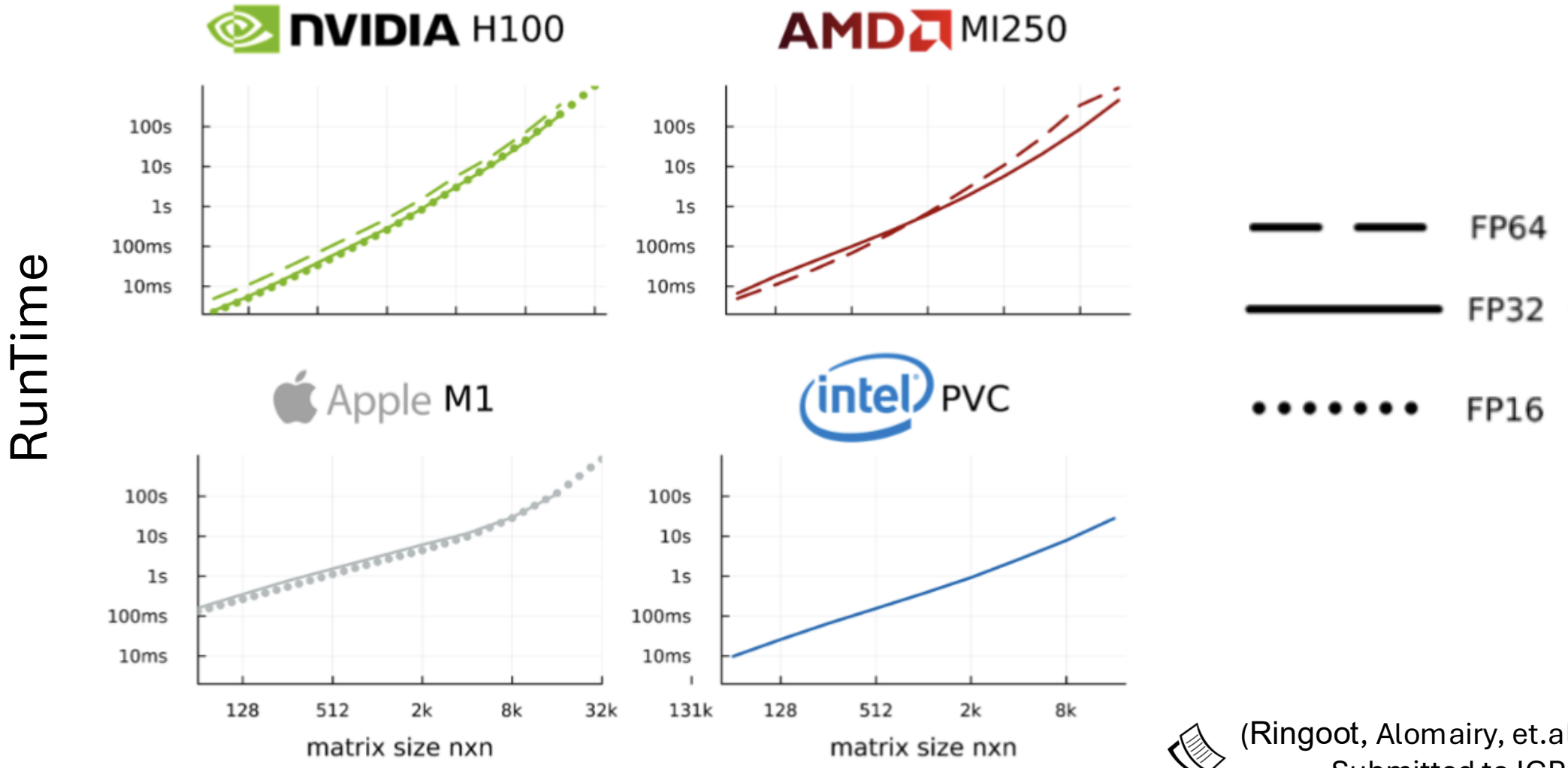
- Performance of singuler value computation
- Ratio of Julia unified implementation to MAGMA and SLATE



(Ringoot, Alomairy, et.al., 2025)
Submitted to ICPP

Hardware and Type Agnostic

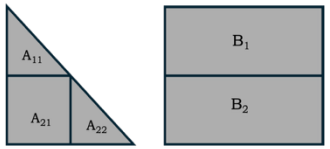
- Potability



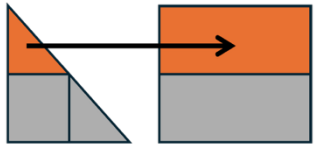
(Ringoot, Alomairy, et.al., 2025)
Submitted to ICPP

Julia Unified TRMM/TRSM Recursive Subdivision

TRMM/TRSM



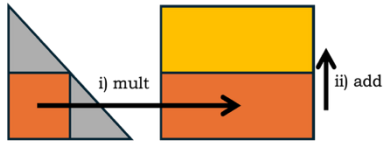
1. Partition A and B



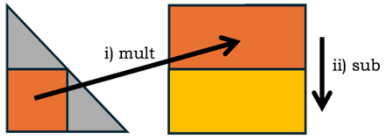
2. Recursive TRMM: $C_1 = A_{11} \times B_1$

Recursive TRSM: $A_{11} \times X_1 = B_1$

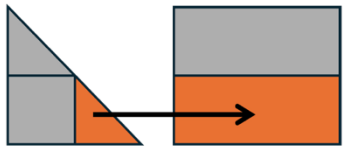
3. GEMM



TRMM: $C_1 = A_{21} \times B_2 + C_1$



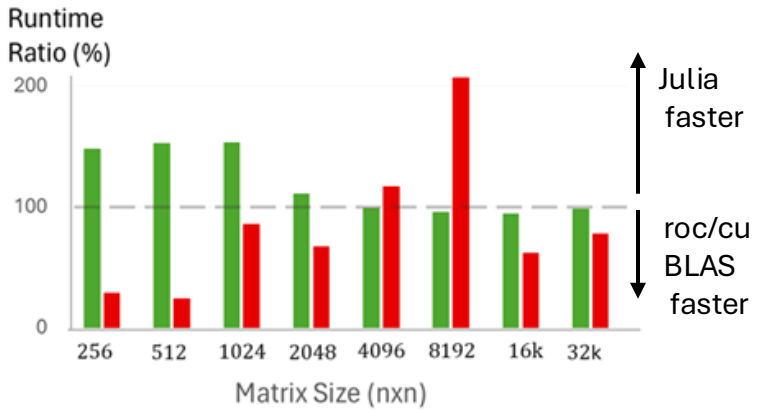
TRSM: $B_2 = B_2 - A_{21} \times B_1$



4. Recursive TRMM: $C_2 = A_{22} \times B_2$

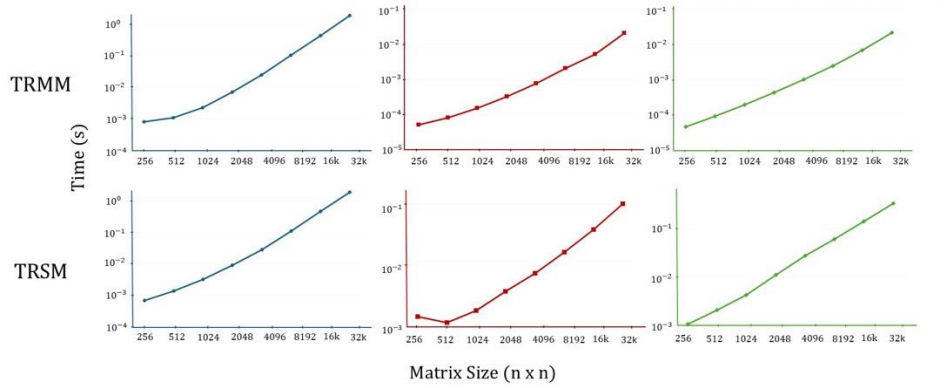
Recursive TRSM: $A_{22} \times X_2 = B_2$

cuBLAS/rocBLAS vs. Julia



AMD NVIDIA

Apple AMD NVIDIA



(Carrera, Max, Alomairy,et.al, WAMTA, 2025)
(Original Algorithm Charara, et.al., CCPE, 2017)

Consolidated APIs

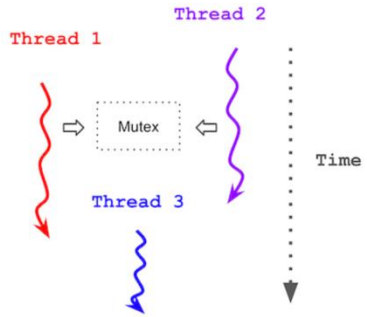
```
1. A = rand(T, 20480, 20480)
2. DA = Distribute(A, AutoBlocks())::DArray
3. cholesky!(A)
4. cholesky!(DA)
5. A.U ≈ DA.U
```

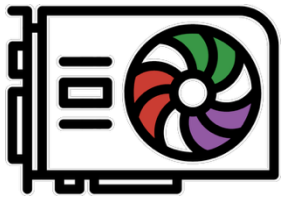
DArray

LinearAlgebra.jl

Dagger.jl

Multi-threaded

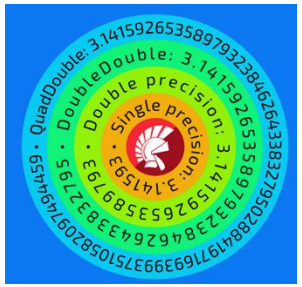




GPUs

Distributed

Multiprecision




```
 1. function cholesky!(A::DArray{T,2}) where T
 2.
 3.     Ac = A.chunks
 4.     mt, nt = size(Ac)
 5.
 6.     Dagger.spawn_datadeps() do
 7.         for k in range(1, mt)
 8.             Dagger.@spawn LAPACK.potrf!('U', InOut(Ac[k, k]))
 9.             for n in range(k+1, nt)
10.                 Dagger.@spawn BLAS.trsm!('L', uplo, trans, 'N', one(T), In(Ac[k, k]), InOut(Ac[k, n]))
11.             end
12.             for m in range(k+1, mt)
13.                 Dagger.@spawn BLAS.syrk!(uplo, 'T', -one(T), In(Ac[k, m]), one(T), InOut(Ac[m, m]))
14.                 for n in range(m+1, nt)
15.                     Dagger.@spawn BLAS.gemm!(trans, 'N', -one(T), In(Ac[k, m]), In(Ac[k, n]), one(T), InOut(Ac[m, n]))
16.                 end
17.             end
18.         end
19.     end
20.
21.     return UpperTriangular(A)
22. end
```

Dagger DataDeps

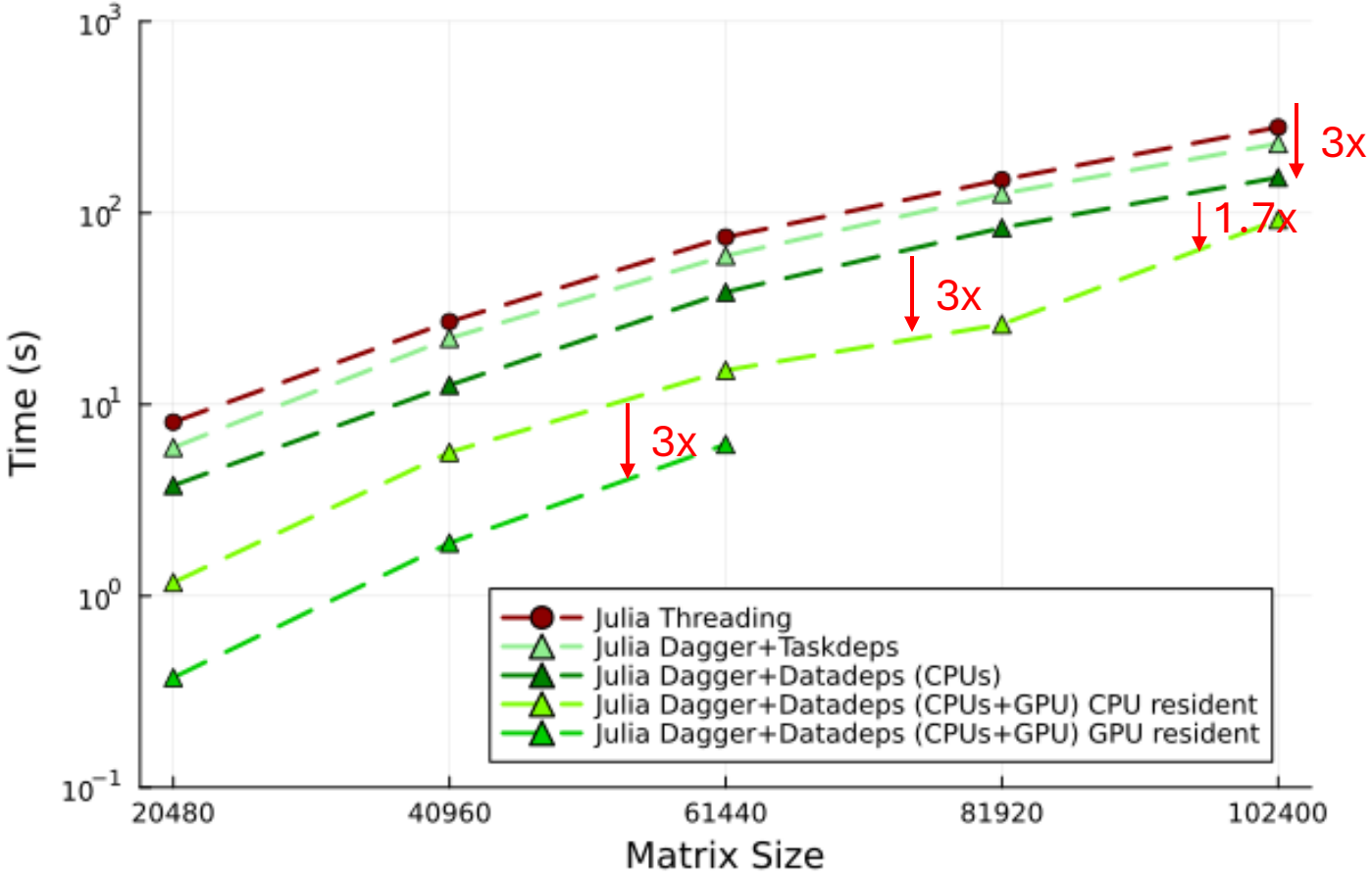
DTask

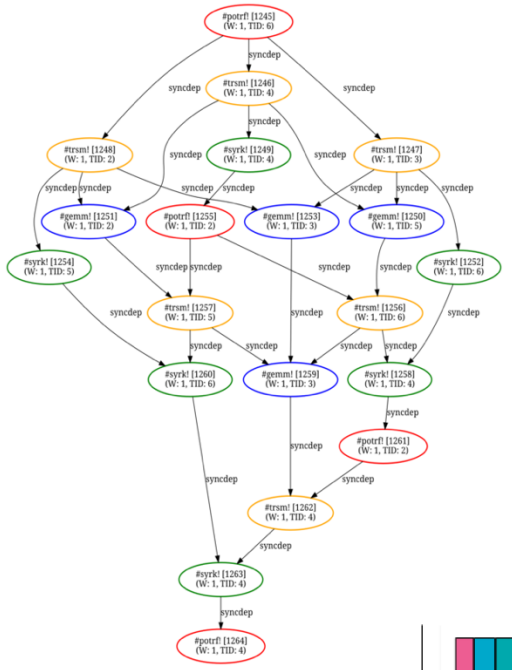
✓ Productivity

(Alomairy, et.al., HPEC, 2024)

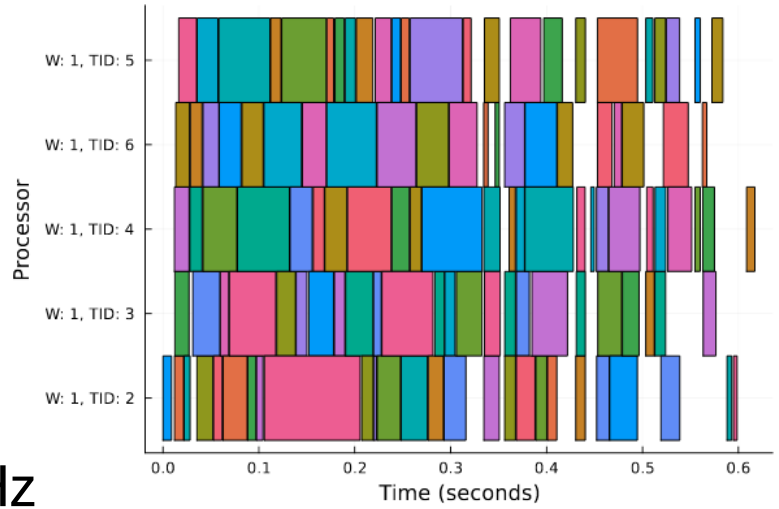
Tile Cholesky Factorization

- Comparing Julia Parallel Paradigms





Dagger Datadeps

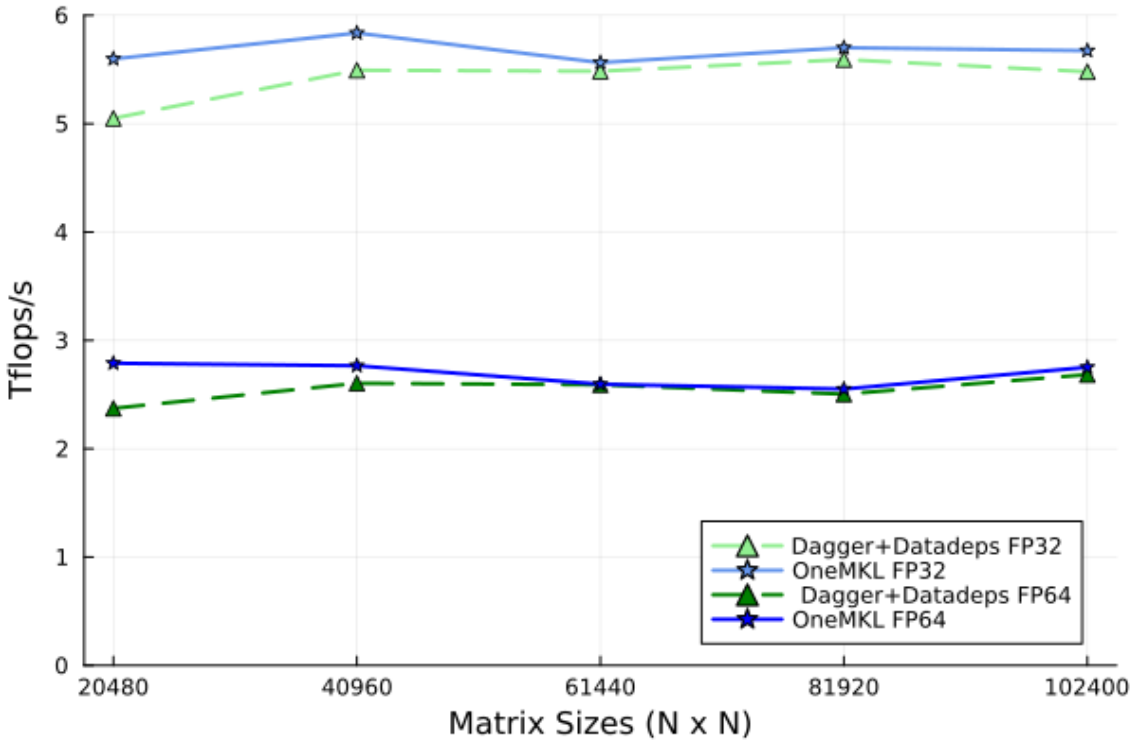


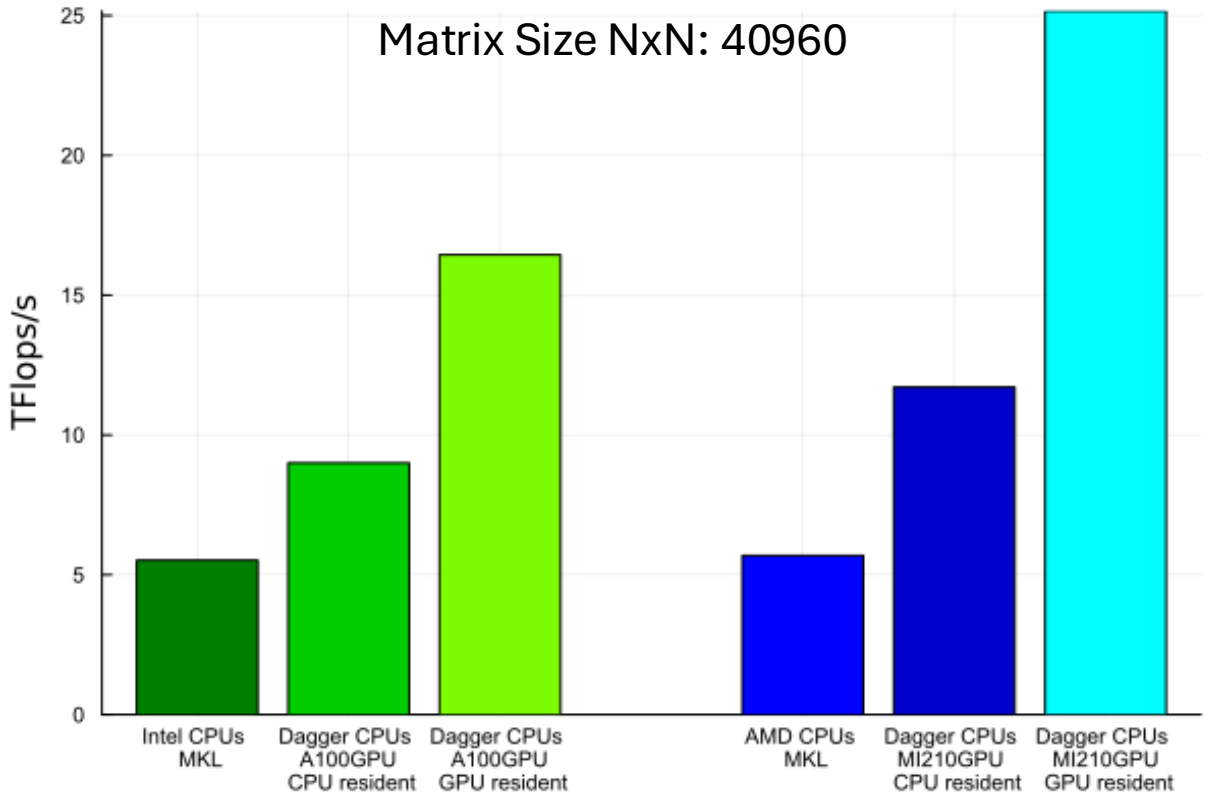
Dual-socket 28-core Intel(R) Xeon(R) Gold 6330 CPU @ 2.00GHz with 1TB of main memory and A100 NVIDIA GPU with 40G HBM

(Alomairy, et.al., HPEC, 2024)

Tile Matrix-Matrix Multiplication

✓ Portability





Dual-socket 28-core Intel(R) Xeon(R) Gold 6330 CPU @ 2.00GHz with 1TB of main memory

NVIDIA A100 GPU

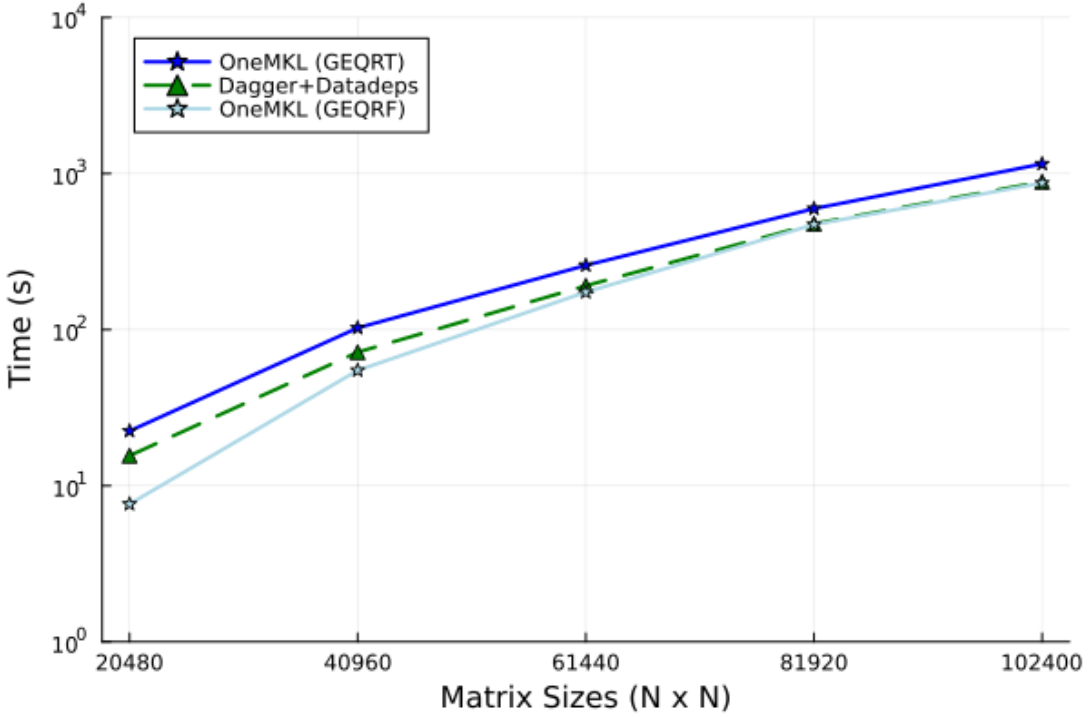
AMD MI210 GPU

(Alomairy, et.al., HPEC, 2024)

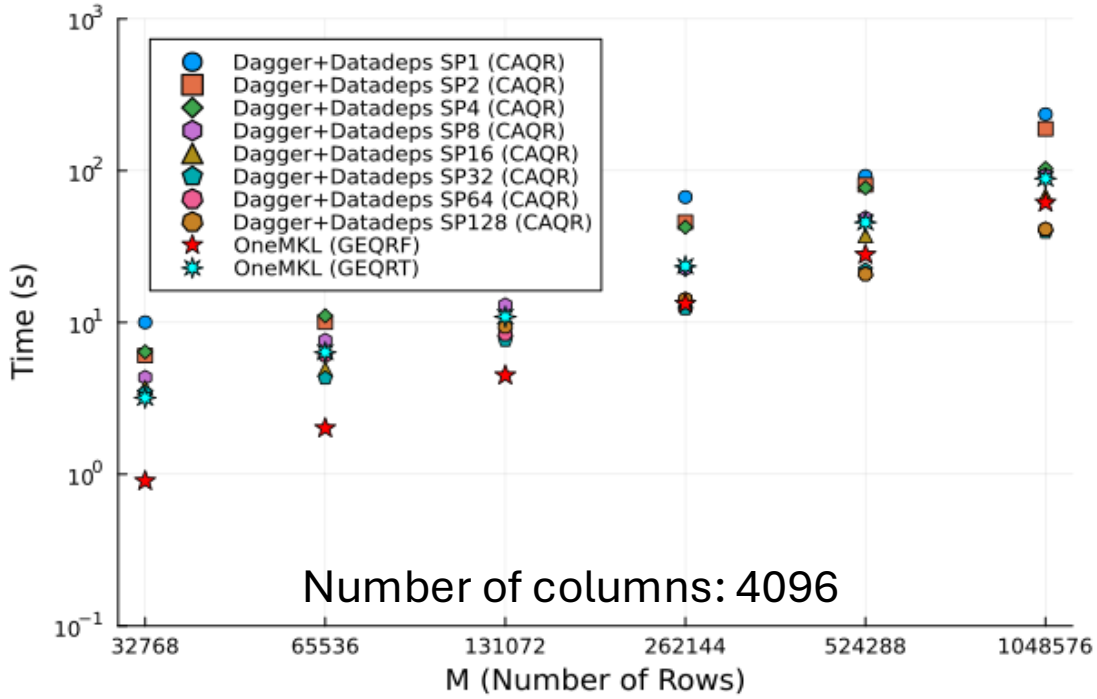
Tile QR Factorization

✓ Performance

- Performance of square matrices



- Performance of tall and skinny matrices
- Semi-parallel communication avoiding QR



1.5x-2x

Dual-socket 28-core Intel(R) Xeon(R Gold 6330 CPU @ 2.00GHz with 1TB of main memory

(Alomairy, et.al., HPEC, 2024)
(Original Algorithm, Hadri, IPDPS, 2010)

References

- Dongarra, Jack, and David Keyes. "*The co-evolution of computational physics and high-performance computing*." Nature Reviews Physics (2024).
- Cedric Augonnet, Samuel Thibault, and et al. StarPU: " *a Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures* ". Concurrency and Computation: Practice and Experience (2011).
- George Bosilca, Aurelien Bouteiller, and et al. " *PaRSEC: Exploiting Heterogeneity to Enhance Scalability* ". Computing in Science & Engineering (2013).
- James Demmel, Laura Grigori, and et al. "*Communication-Optimal Parallel and Sequential QR and LU Factorizations*". SIAM Journal on Scientific Computing (2012).
- Mark Gates, Jakub Kurzak, and et al. " *SLATE: Design of a Modern Distributed and Accelerated Linear Algebra Library* ". In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (2019).
- Mose Giordano, Milan Klower, and et al. *" Productivity Meets Performance: Julia on A64FX* ". IEEE International Conference on Cluster Computing(2022).
- Bilal Hadri, Hatem Ltaief , and et al. " *Tile QR Factorization with Parallel Panel Processing for Multicore Architectures* ". 2010 IEEE International Symposium on Parallel & Amp; Distributed Processing (2010).
- Michael P. Robson, Ronak Buch, and et al. " *Runtime Coordinated Heterogeneous Tasks in Charm++* ". In Proceedings of the International Workshop on Extreme Scale Programming Models and Middleware (2016).
- Asim YarKhan, Jakub Kurzak, and et al. " *Porting the PLASMA Numerical Library to the OpenMP Standard* ". International Journal of Parallel Programming (2017).

Acknowledgment

- King Abdullah University of Science and Technology (KAUST)
- National Science Foundation (NSF)
- Advanced Research Projects Agency-Energy (ARPA-E)
- U.S. Department of Energy (DOE)
- U.S. Air Force Artificial Intelligence Accelerator (AFRL)
- Defense Advanced Research Projects Agency (DARPA)
- Sao Paulo Research Foundation (FAPESP)

Thanks!!