東京大学情報基盤センター
INFORMATION TECHNOLOGY CENTER, THE UNIVERSITY OF TOKYO

MLPhys 学術変革領域研究(A) 学習物理学の創成
Foundation of "Machine Learning Physics"

Juliaによる4次元格子量子色力学：JuliaQCDプロジェクト

東京大学情報基盤センター学際情報科学研究部門
東京大学大学院新領域創成科学研究科物質系専攻

永井佑紀

JuliaQCD project **"JuliaQCD: Portable lattice QCD package in Julia language"**
Y. Nagai and A. Tomiya, arXiv:2409.03030

AD in QCD **"Lattice Gauge Theory via LLVM-Level Automatic Differentiation"**
Y. Nagai, A. Tomiya and H. Ohno, arXiv:2602.20516

About me

Yuki Nagai Associate Professor in the University of Tokyo

I started using Julia around 2016 when I was a visiting researcher in MIT

My interests

Condensed matter theory
Superconductivity,
Material science
Machine-learning and Physics

"Julia Programming for Numerical Computation: A one-week course" 2022

"Introduction to Numerical Calculations with Julia" 2024

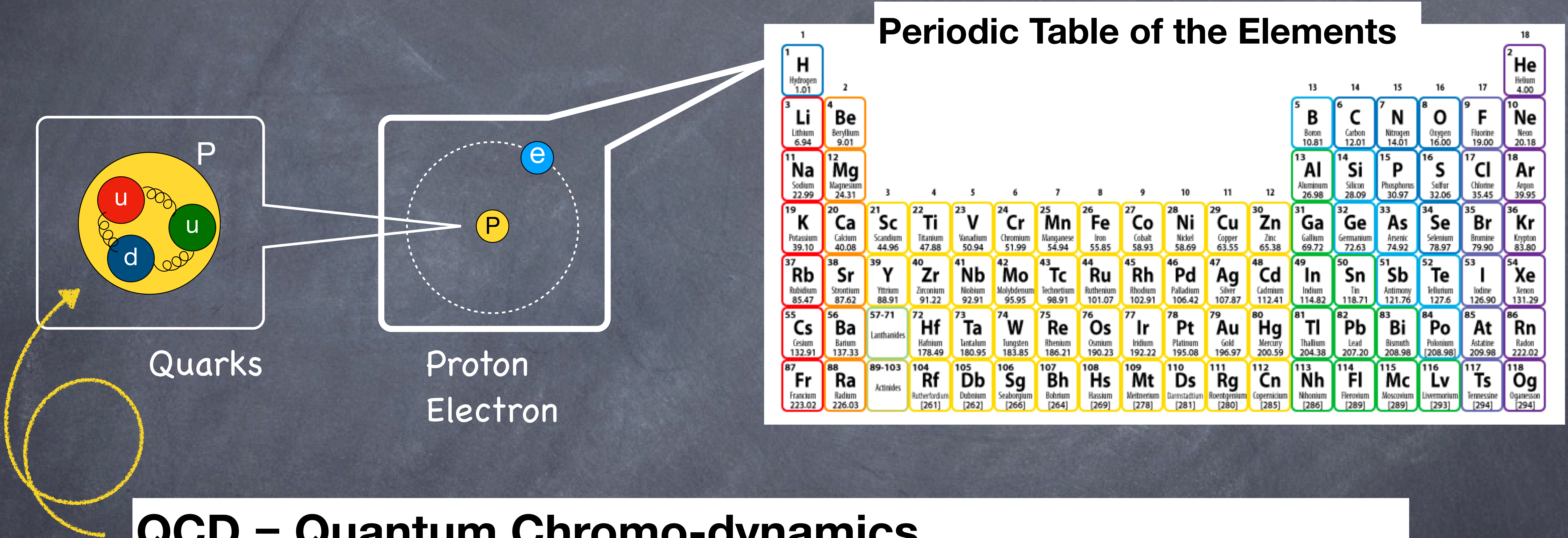
I wrote two books for numerical computing with Julia

Outline

- Introduction: Lattice QCD
- JACCによる加速
- JACC.jlその他の進展
- Differentiable Lattice QCD (AD)
- Benchmarks
- Summary

Introduction: Lattice QCD

What is QCD?



QCD = Quantum Chromo-dynamics

= A fundamental theory for particles inside of nuclei

Quantum many body, relativistic, strongly correlated

One of the hardest problem in the world

Lattice QCD and Monte Carlo

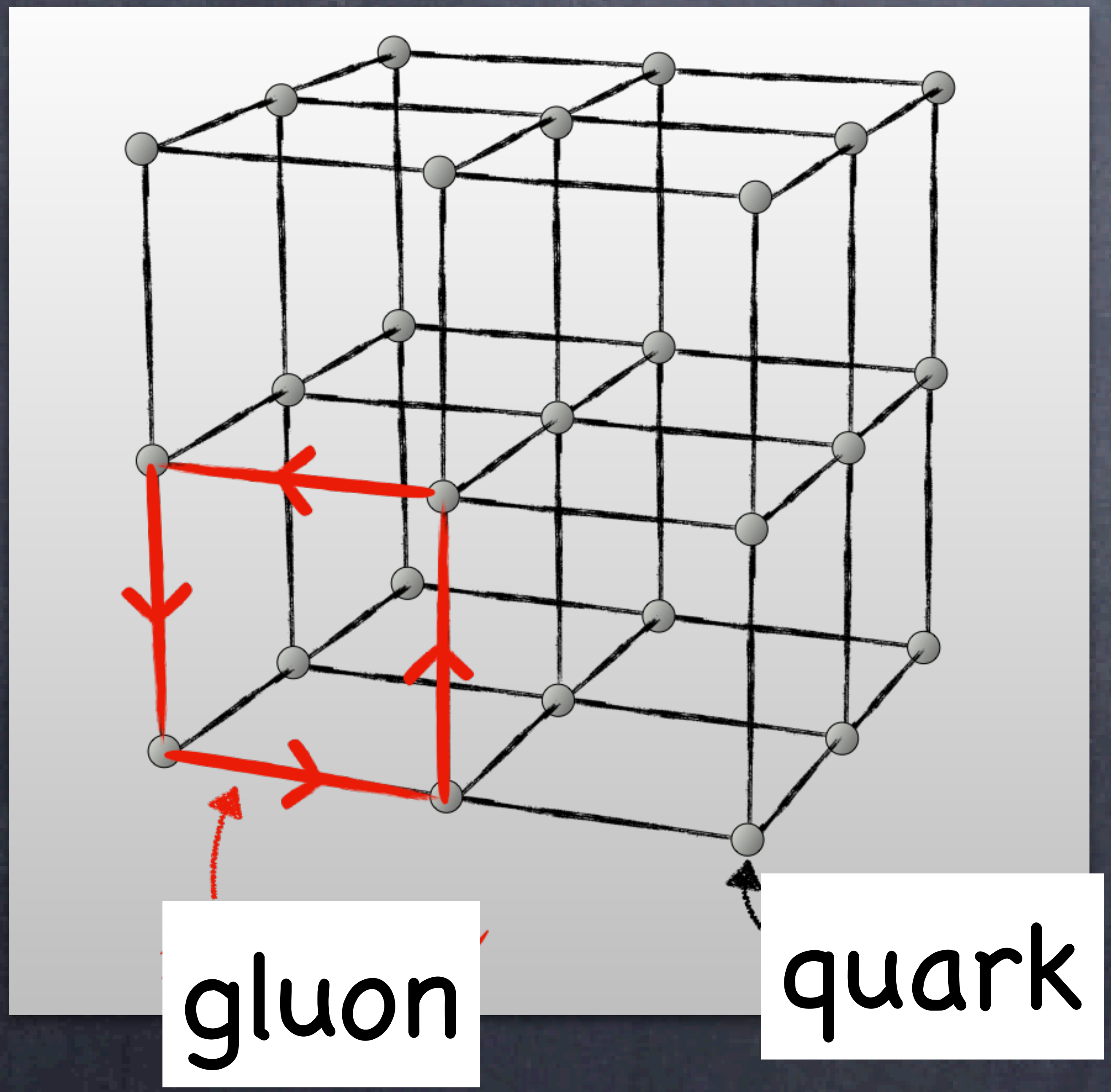
A target observable (related to observables in experiments)

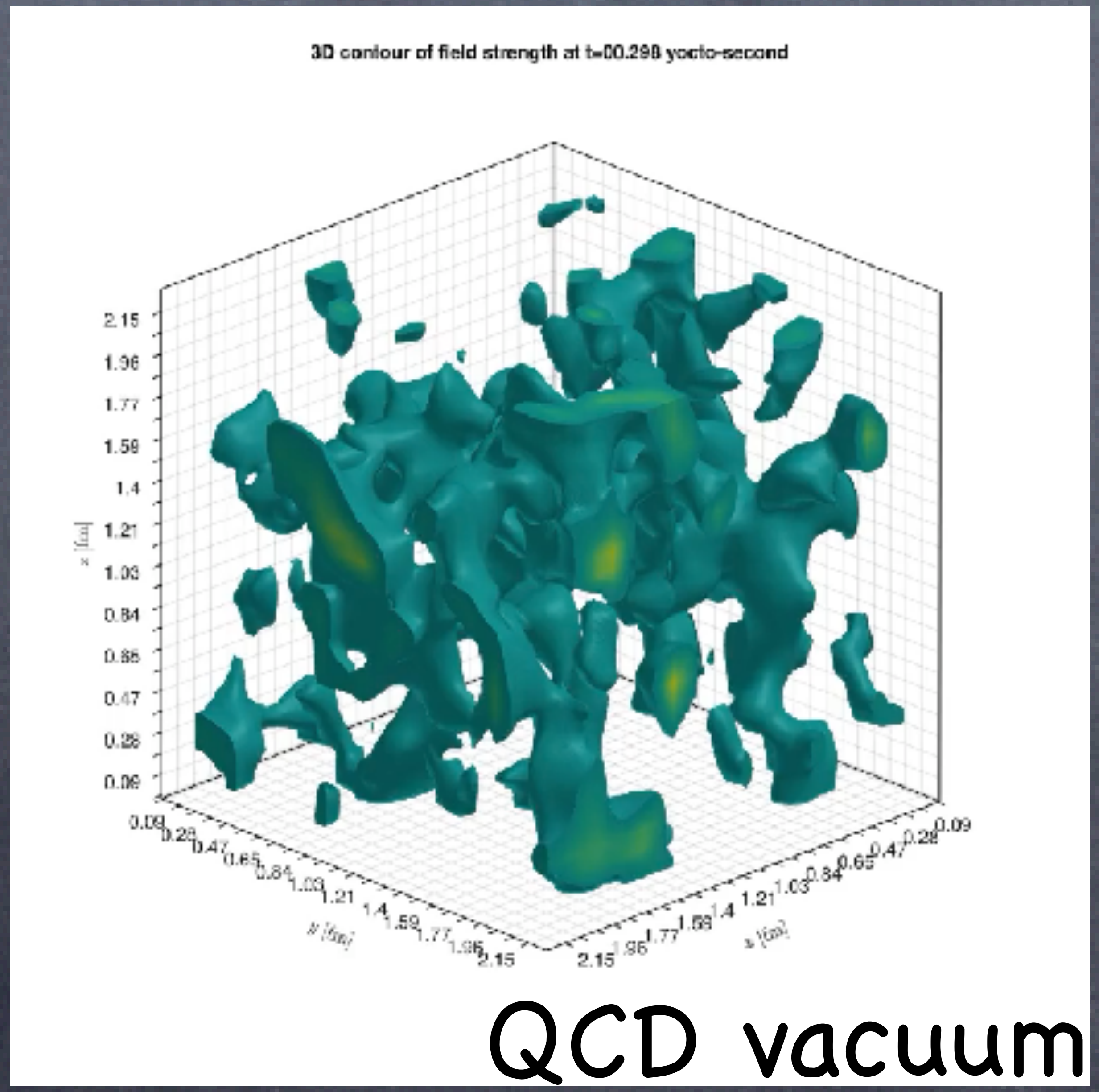
Action for lattice QCD

$$S_{\rm QCD} = S_{\rm gauge}[U] + \log\det(\not{D}[U] + m)$$

$$\langle \mathscr{O} \rangle = \frac{1}{Z}\int \mathscr{D}U e^{-S_{\rm QCD}[U]}\mathscr{O}(U)$$

e.g. $256^4 \times 4 \times 8$ dimensional integral





Generate field configurations with

$$P[U] \propto e^{-S_{\rm QCD}[U]}$$

HMC (Hamiltonian/Hybrid Monte Carlo)

$$U(\tau+\Delta\tau) = e^{i\,\Delta\tau\, P(\tau)}\, U(\tau),$$
$$P(\tau+\Delta\tau) = P(\tau) - \Delta\tau\, \mathcal{F}(\tau),$$

We solve equations of motions

What kind of calculation is needed?

Lots of inversions! Massive parallelism has been used

Anatomy of lattice QCD calculation (typical case):

Few months to few years

- **90 %** Lots of inversions with huge sparse matrices (solving linear equations (Dirac equation))
  100(times/step) x 1,000,000,000(steps) of inversions with N x N sparse matrix
  N = 20^4 x 4 x 8 x 2 = 10, 240, 000 ~ **10 Million**
- **10 %** Multiplication of 3x3 complex matrix with 20^4 x 4 times for 1 step etc

Nested loops for x=1:20, y=1:20, z=1:20, t=1:20 and internal degrees of freedoms (color=3, spinor = 4).

Conventionally, we have used C++/Fortran & massive parallelization (MPI/OpenMP/GPU, hybrid) on supercomputers

Point: **Lattice QCD is one of the most numerically expensive calculation**
Useful to understand our world & **Good benchmark** of software/hardware

Public codes for LQCD

Name (Historical order, old->new)	Language	URL	Paper
MILC code	C/C++	https://github.com/milc-qcd/milc_qcd	https://inspirehep.net/literature/321665
Lattice Tool kit	Fortran	https://github.com/tsuchim/Lattice-Tool-Kit/	NPB Proc.Suppl. 106 (2002) 1037-1039
CPS (Columbia physics system)	C/C++/Assemblar	https://github.com/RBC-UKQCD/CPS	https://arxiv.org/abs/hep-lat/0306023
Chroma	C++	https://github.com/JeffersonLab/chroma	arXiv:hep-lat/0409003
QUDA(backend)	C++/CUDA	https://github.com/lattice/quda	arXiv:1011.0024
Bridge++	C++/GPU	https://bridge.kek.jp/Lattice-code/index_e.html	J.Phys.Conf.Ser. 523 (2014) 012046
Grid	C++/GPU	https://github.com/paboyle/Grid	arXiv:1512.03487
JuliaQCD	**Julia**	**https://github.com/juliaqcd/**	**https://arxiv.org/abs/2409.03030**
SimuLATeQCD	C++/CUDA	https://github.com/LatticeQCD/SIMULATeQCD	https://arxiv.org/abs/2306.01098

and more

We made LQCD code with Julia

1. Benchmark test for *Julia itself* since LQCD is a hardest problem
2. Easy to use. **It can work on PC and supercomputers**
3. Minimize time/effort for “code development” + “execution”

Lattice QCD code for generic purpose

Open source code in Julia language

LatticeQCD.jl

Machines: Laptop/desktop/Jupyter/Supercomputers (almost everywhere)

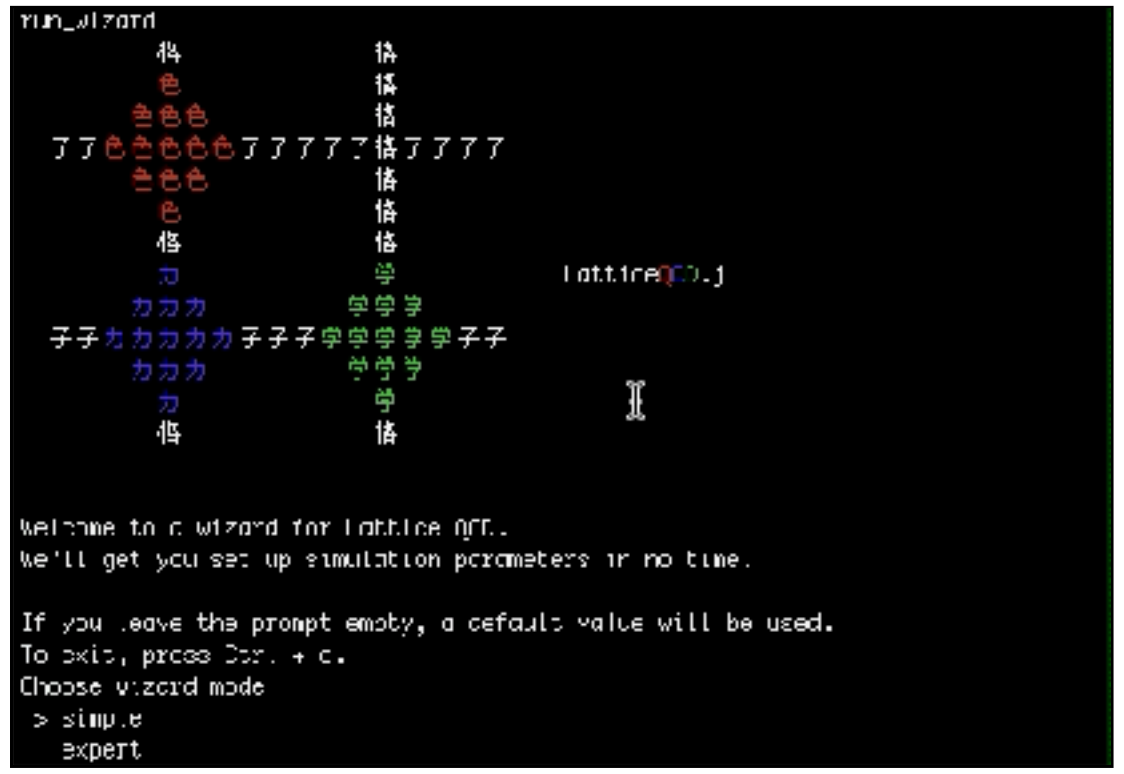
Advantage: Portability, no-explicit compile, fast, machine learning friendly

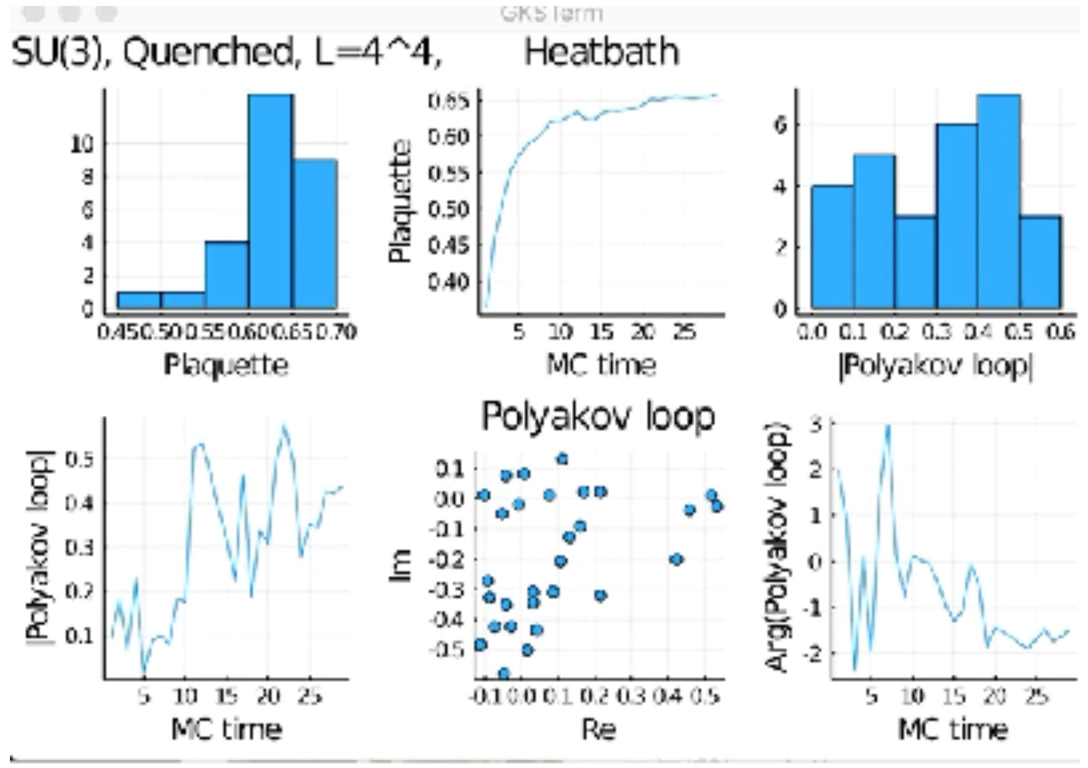
Functions: 4d, SU(Nc)-heatbath, (R)HMC, Self-learning HMC, SU(Nc) Stout, Z2 gauge,
Dynamical Staggered, Dynamical Wilson, Dynamical Domain-wall
Measurements (chiral condensate, topological charge, etc)

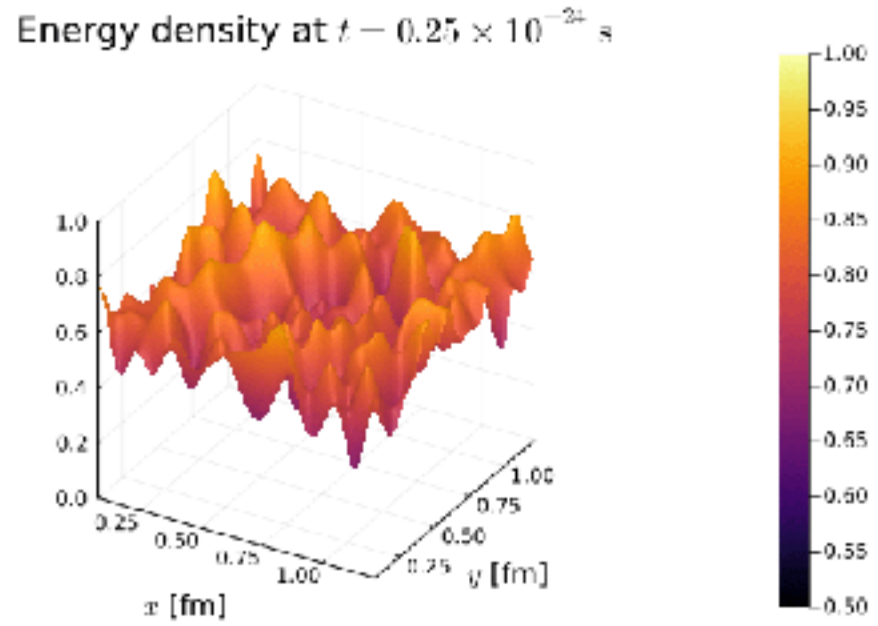
Start LQCD in **5 min**

1. Download Julia binary
2. Add the package through Julia package manager
3. Execute!

https://github.com/akio-tomiya/LatticeQCD.jl







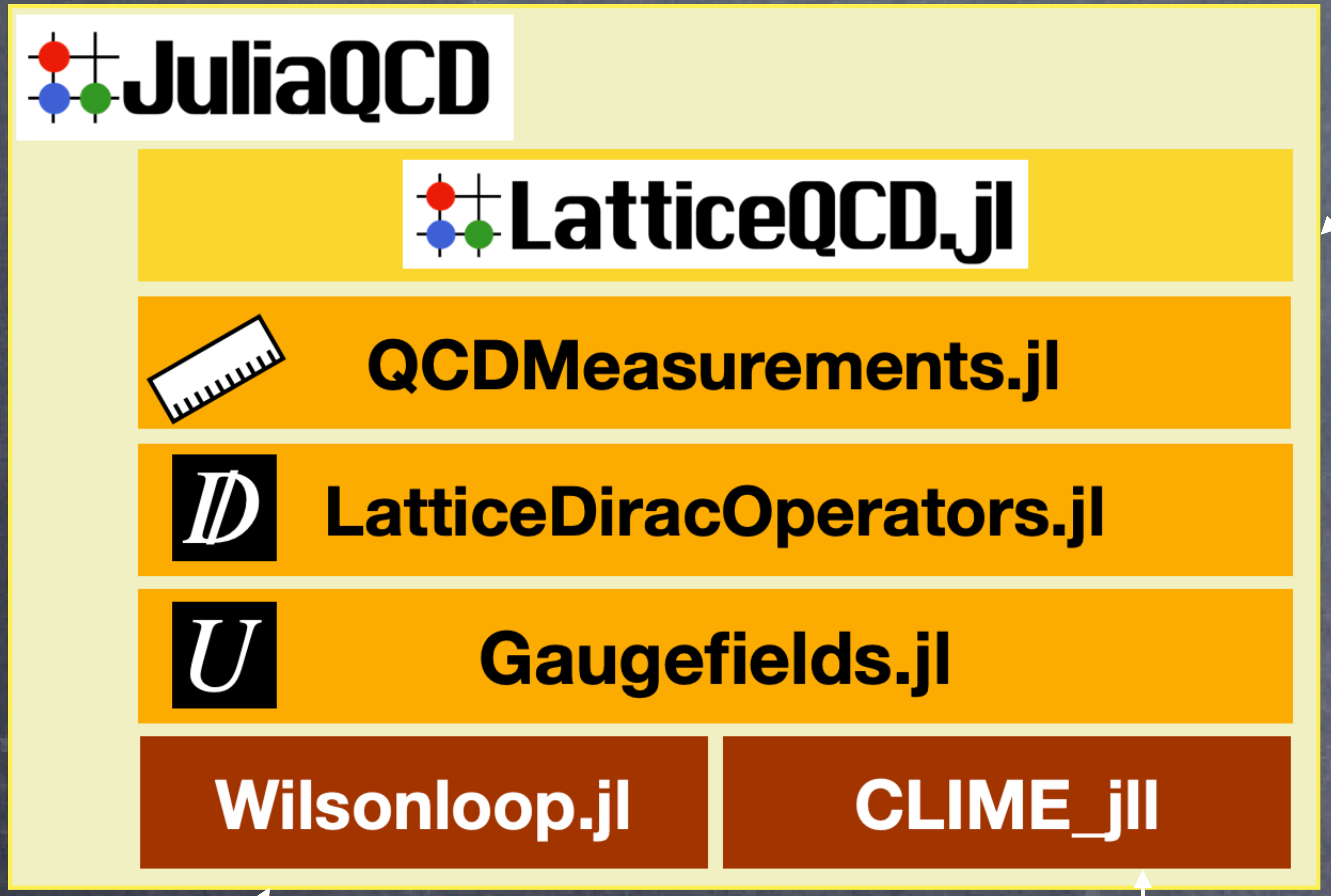
Package structure

https://github.com/JuliaQCD

LQCD code with Julia

Document: https://arxiv.org/abs/2409.03030

Dependency is automatically solved



Wrapper for LatticeDiracOperators.jl & Gaugefields.jl, QCDMeasurements.jl
- Wizard for parameter files
- HMC/RHMC for SU(Nc)
  - Stout + Wilson/Staggered/DW
- Heatbath for SU(Nc)
- Measurements
- Jupyter, Colab/**PC/Supercomputers**

etc

Measurements in LQCD (Correlator, Flow, Qtop, etc)

Fermions (+HMC), Wilson, KS, DW, MPI
PC/Supercomputers

Gauge fields (+HMC/Heatbath), MPI
SU(N) and Zn gauge
PC/Supercomputers

ILDG I/O

Symbolic operations of Wilson/Polyakov loops

Compatible with auto-grad in Flux.jl (Deep learning library)

Why do we use Julia?

Fast as Fortran, easy as Python

In LatticeQCD

Hardest part is “Quantum effects from quarks”

- **large dimensional linear equations** have to be solved, many times

N = 20^4 x 4 x 8 x 2 = 10, 240, 000 ~ **10 Million** -> Highly-optimized code is needed

The computational cost is huge -> Sometimes compiling a package becomes difficult

Portability

LatticeQCD.jl works even on Raspberry Pi Zero 2

Easy to understand

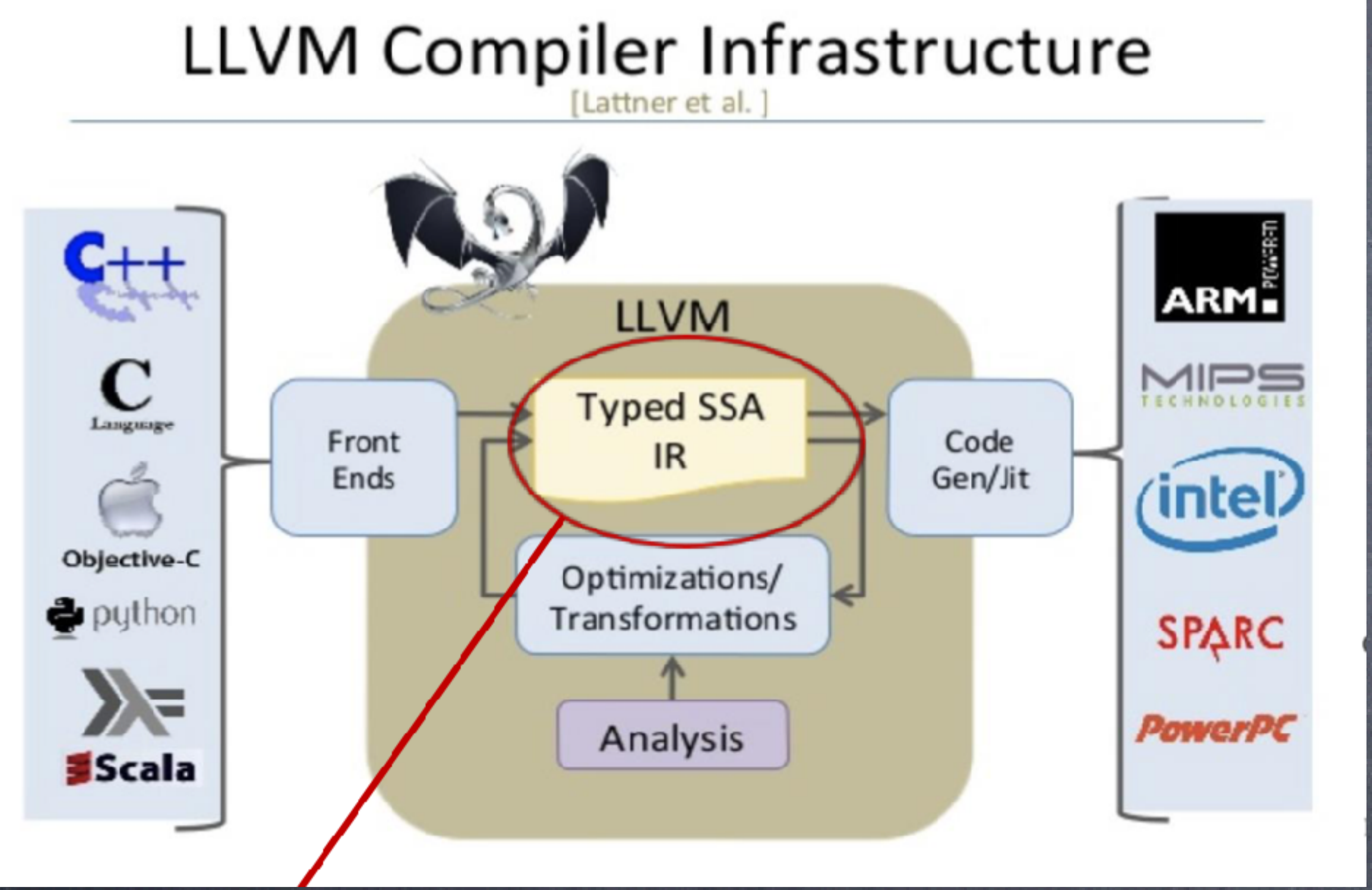
One can easily add his/her algorithms in JuliaQCD

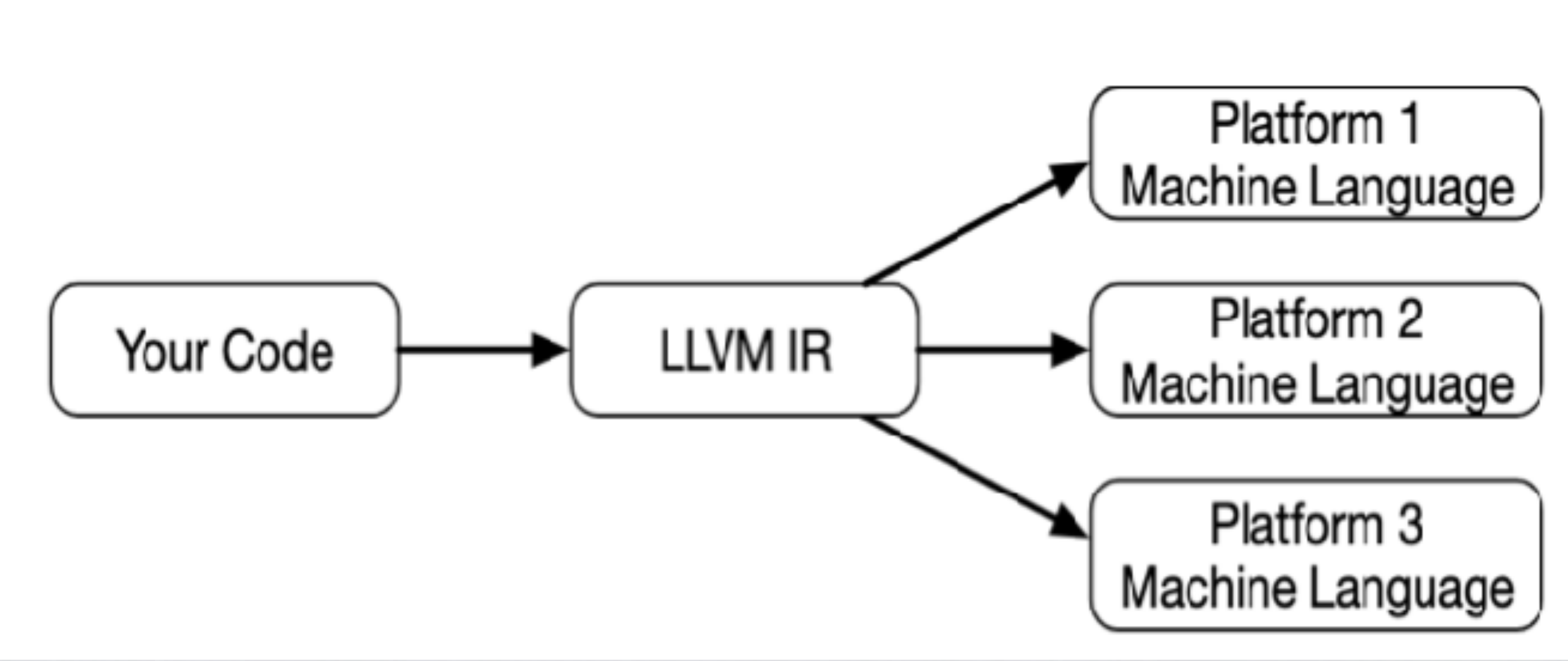
Machine-learning friendly

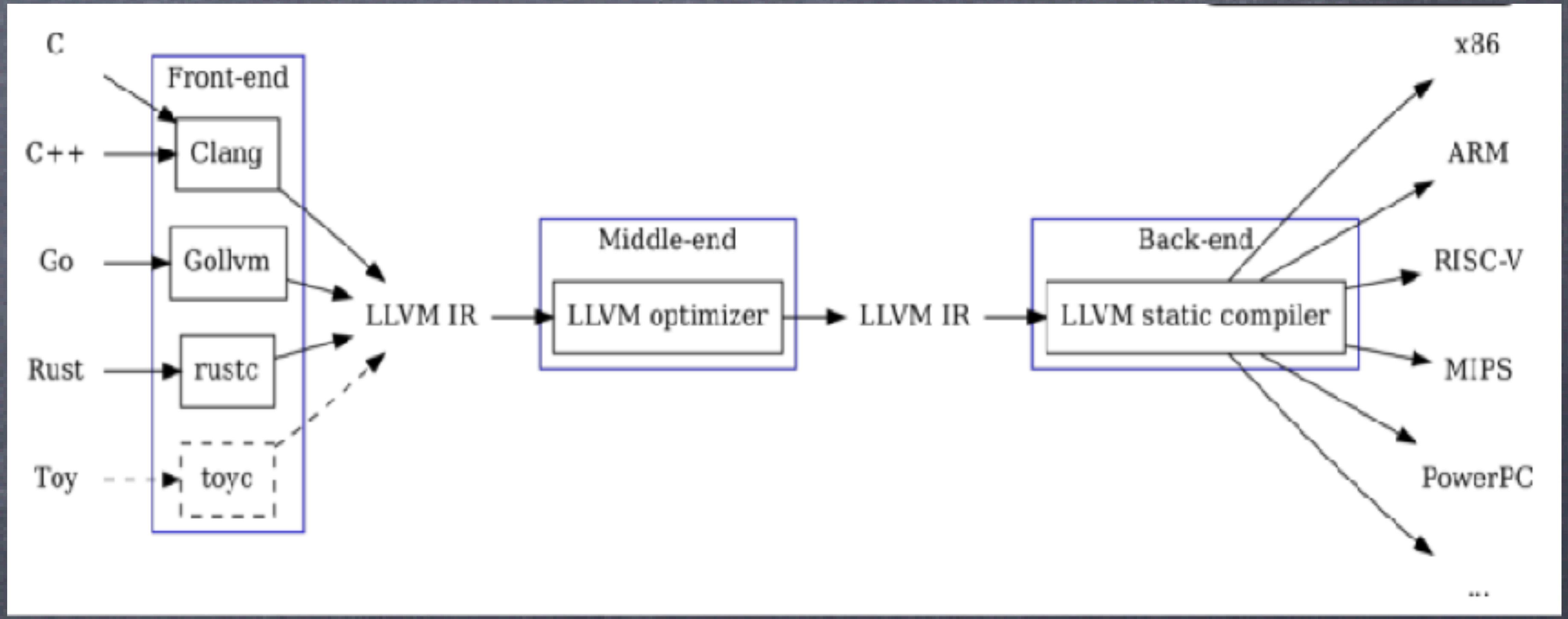
We can easily implement machine learning algorithms (like PyTorch in Python)

Julia and LLVM

Julia uses "LLVM"





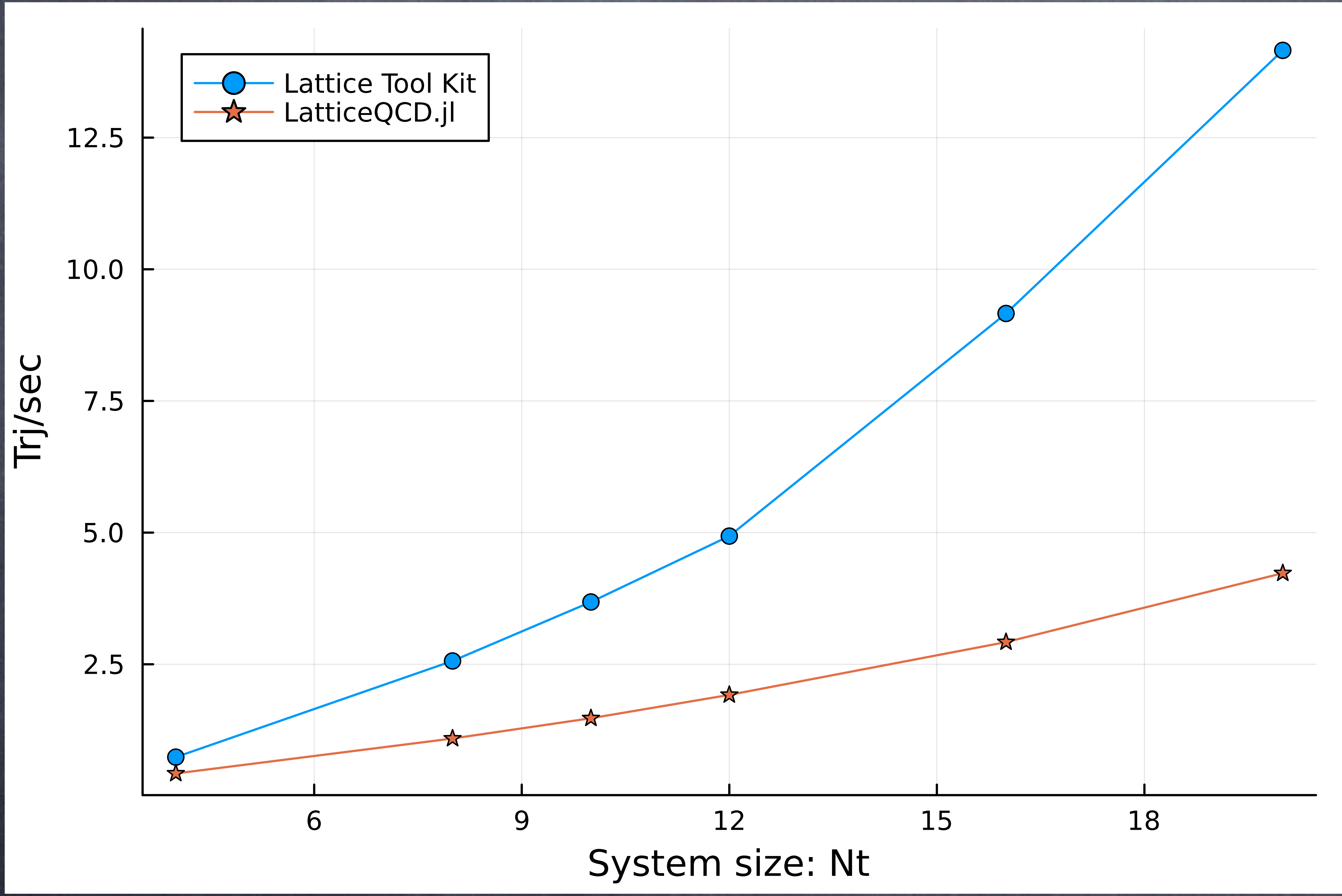


write once, run anywhere

Comparison

Single core calculation

Same algorithm, same parameter



Machine:
Mac Book Pro 14inch 2023 M2 Max,
memory 96GB
- Julia 1.10.8

```
LatticeQCD 1.3.4
LatticeDiracOperators 0.4.3
Gaugefields 0.5.1
Wilsonloop 0.1.5
```

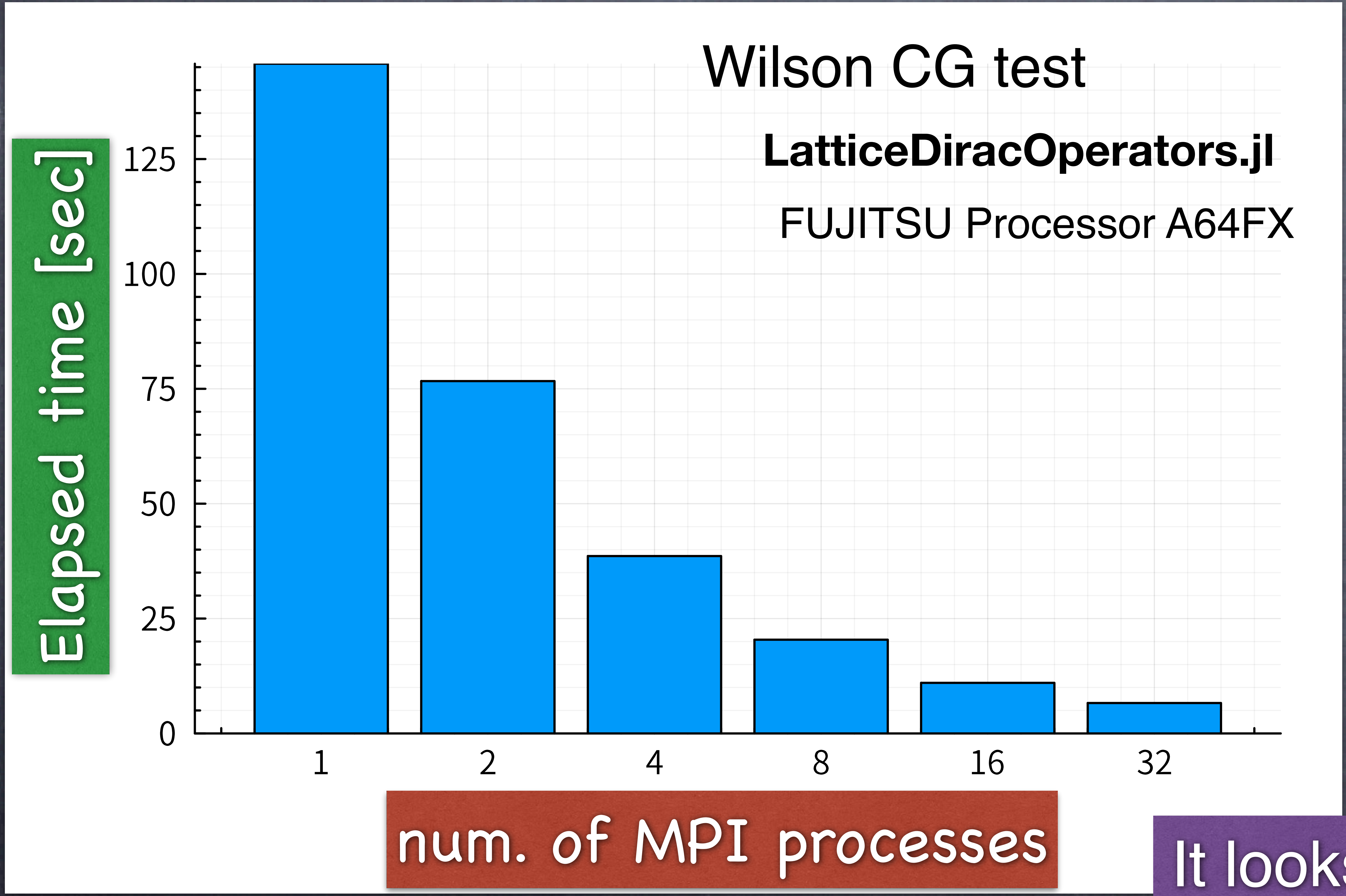
Parameters
$L=4^3 \times Nt$
Nt = 4, 8, 10, 12, 16, 20
kappa = 0.141139
beta = 5.5
Nd = 10
CG eps = 10-8

Compare with Lattice tool kit (2002)
- gfortran 14.2 (w/ -O3)
https://github.com/cometscome/Lattice-Tool-Kit

Note: we did not compare with "cutting edge" packages (MILC, grid, bridge++ etc.) written by C++

MPI performance

On supercomputer the Wisteria/BDEC-01 in the University of Tokyo



It looks scaling well

without MPI

U = Initialize_Gaugefields(NC,Nwing,NX,NY,NZ,NT,condition = "hot")

with MPI

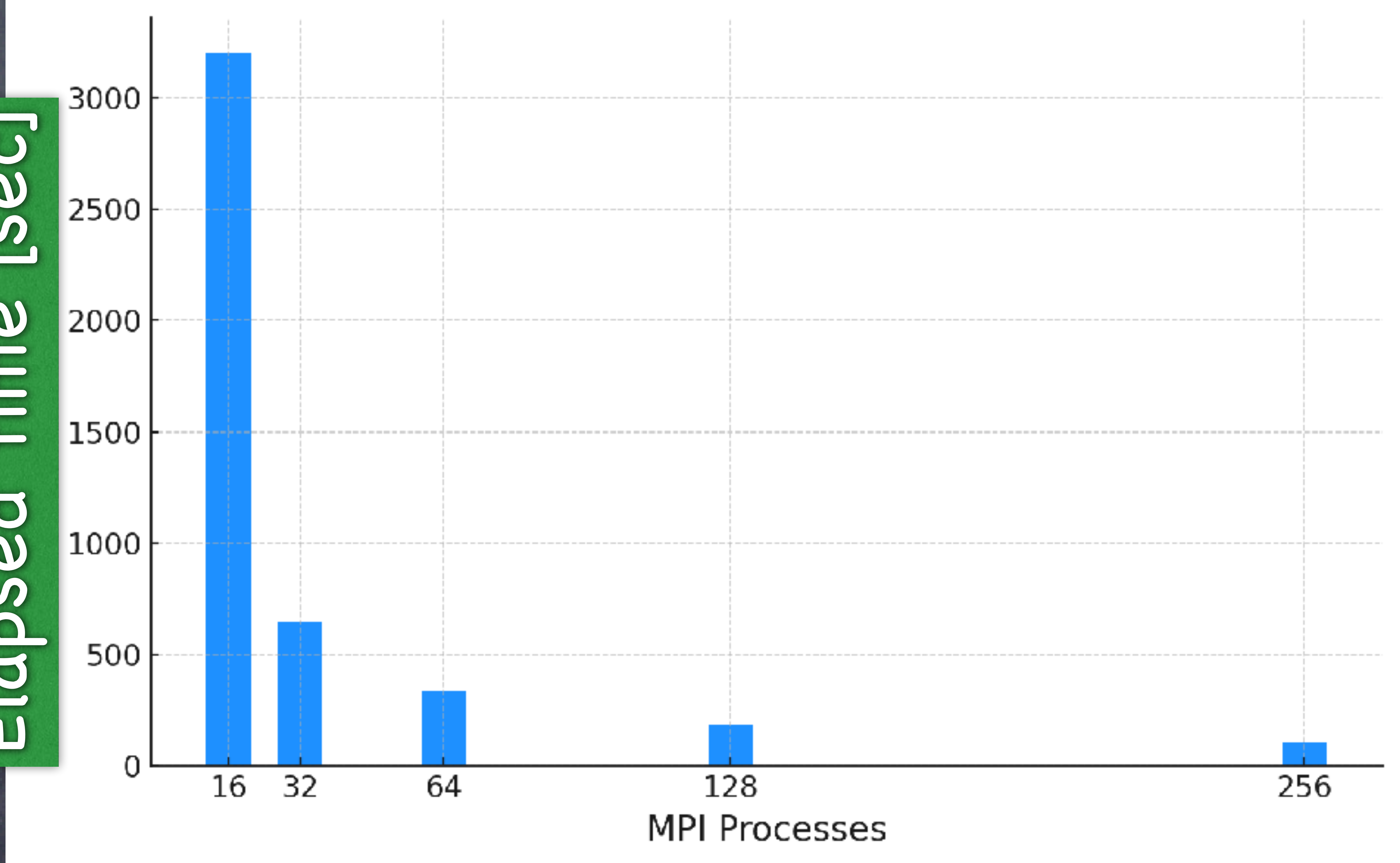
U = Initialize_Gaugefields(NC,Nwing, NX,NY,NZ,NT,condition = "hot",mpi=true,PEs = (1,1,1,2),mpiinit = false)

We can easily use MPI!

MPI performance

Fugaku: Domainwall 1/D   N = 32^4 * 4

Elapsed time [sec]



num. of MPI processes

Fugaku job: small size, node number N satisfies n = 4 * N

n	time
16	3200.326744453
32	646.440953768
64	331.800943196
128	181.141120232
256	101.730185027

We can easily use MPI!

It looks scaling well

JACCによる加速

次に何をすべきか

新しいアーキテクチャが登場するたびにコードを書き換えるのは大変

GPU化すべき？

NVIDIA用のコード（CUDA）を書く？

-> CUDA用のコードは一週間で書けた

富岳Next?

マルチGPU化すべき？

NVIDIA用のコード（CUDA）を書く？

アメリカはAMD製GPUスパコン

AMDGPU用コードを書く？

書けなくもないが…

ハードウェアに依存するとコードの保守が大変

-> JACC.jl

東京大学情報基盤センター
INFORMATION TECHNOLOGY CENTER, THE UNIVERSITY OF TOKYO

ハードウェア依存しないコーディング

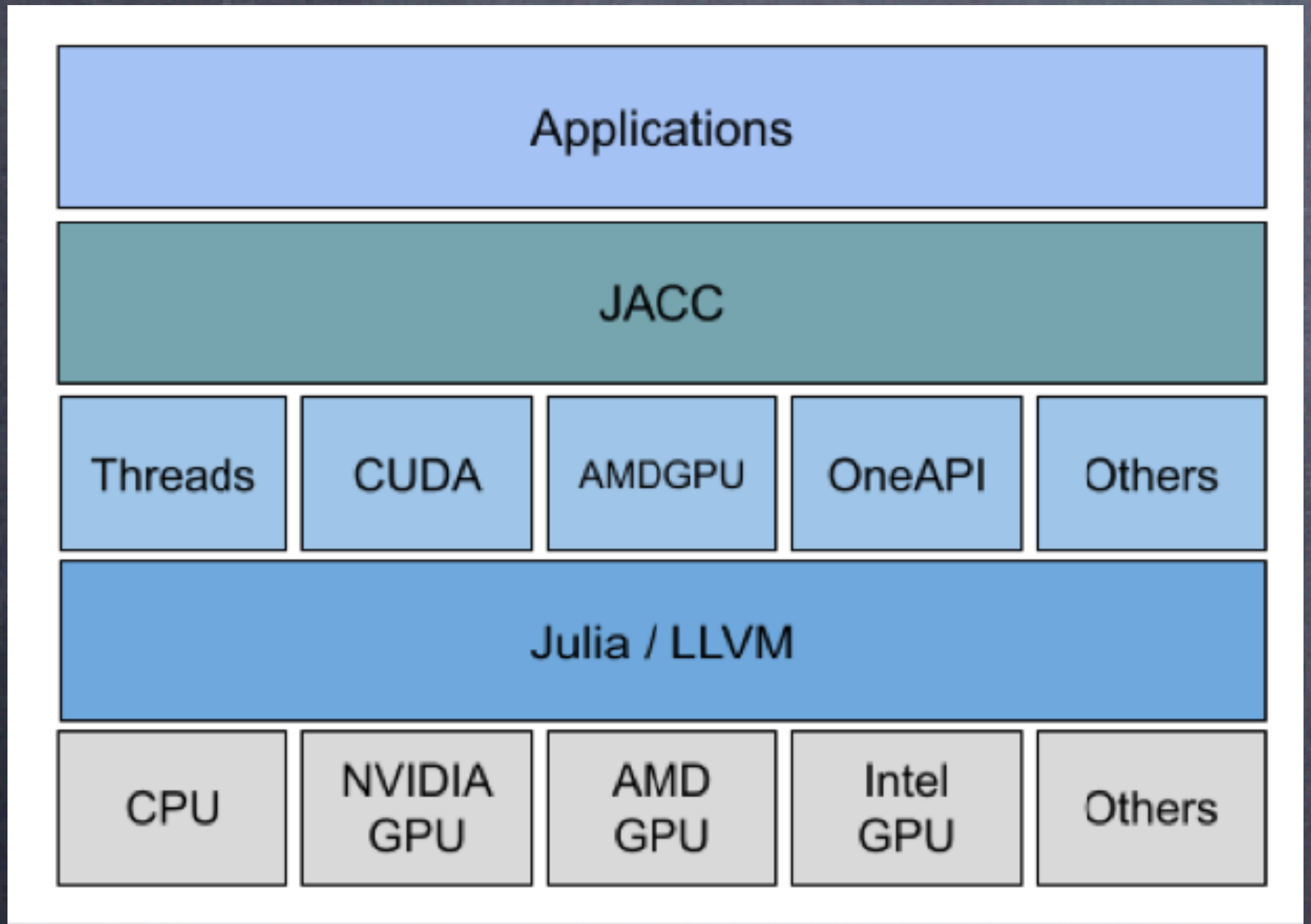
プログラムを、コードを変えずに、性能を保ったまま異なる環境で動かしたい

C++: Kokkos

OpenMP(CPUマルチスレッド), CUDA(NVIDIA GPU), HIP(AMDGPU)

Kokkosのように、ハードウェア依存しないコーディングをやりたい

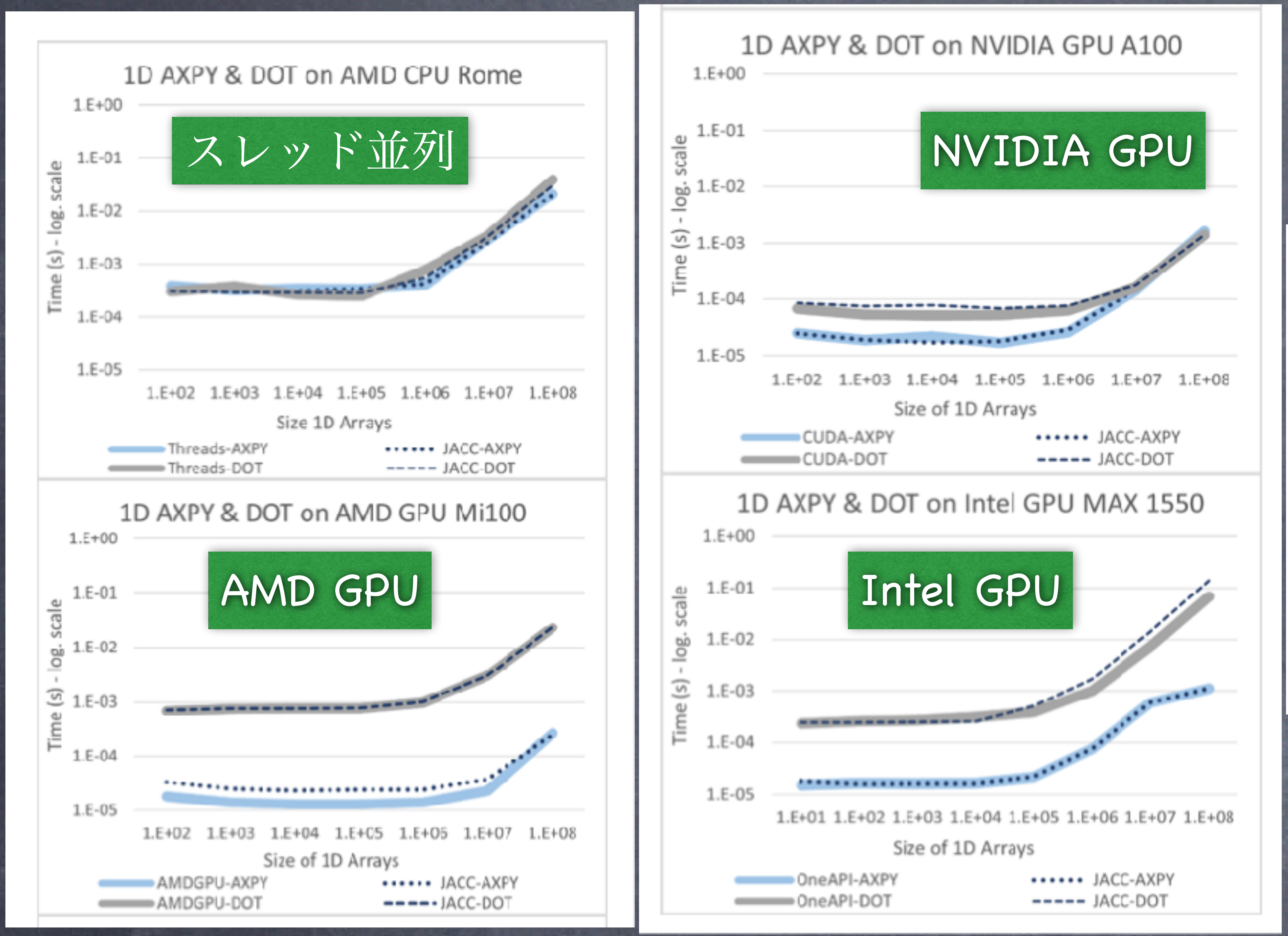
-> JACC.jl



Julia: LLVMによって、異なるCPUで実行可能

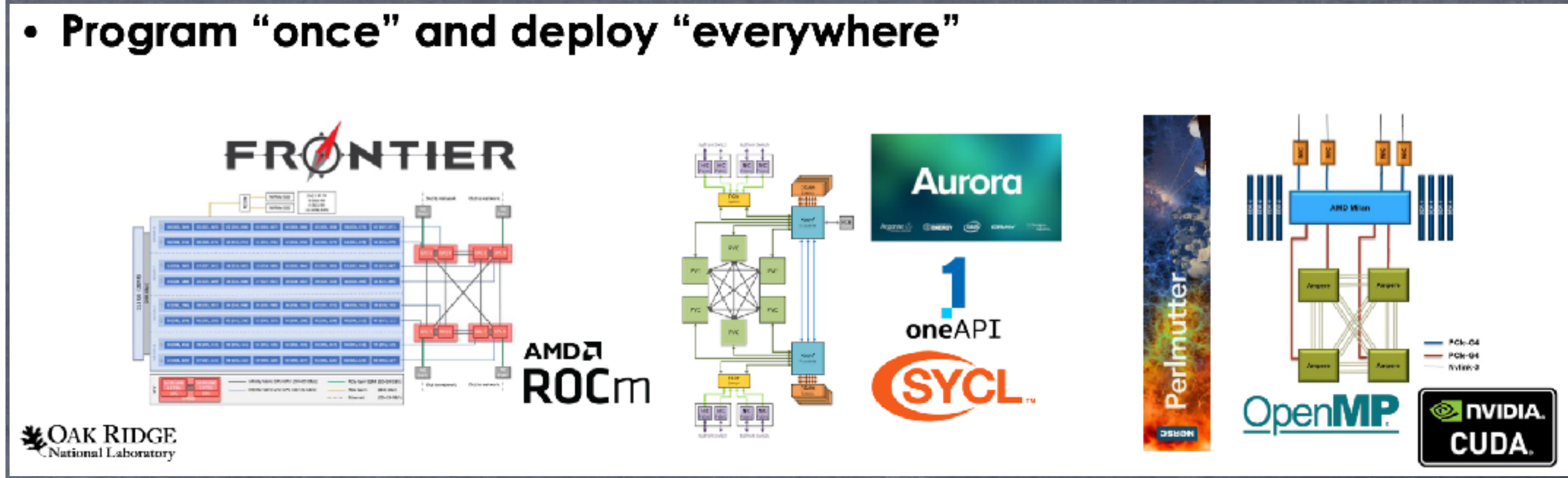
JACC.jl: 異なる加速装置(GPU)で実行可能

JACC



https://ieeexplore.ieee.org/document/10820713

JULIA FOR ACCELERATORS



https://www.cc.u-tokyo.ac.jp/events/ase/47/Teranishi.pdf

同じコードで異なる加速装置をサポート

JACC.jl

一度書けば、スレッド並列、プロセス並列(MPI)、NVIDIA GPU加速、AMD GPU加速、IntelGPU加速、全てに対応したい

-> ハイブリッド並列、マルチGPU並列にも対応したい

JACC.jlを使えばできる！

JACC.jlが提供するのは基本的には三つの関数

JACC.Array(A)　バックエンドに応じて適切に配列Aを変換する

JACC.parallel_for(N,f,a...)　定義すべき関数f(i,a...)

長さNのループを実行する

a...はa,b,c,d,...と何個変数があっても良い

JACC.parallel_reduce(N,f,a...)

長さNのループを実行し、返り値を全て足す

東京大学情報基盤センター INFORMATION TECHNOLOGY CENTER, THE UNIVERSITY OF TOKYO

Benchmarks(preliminary)

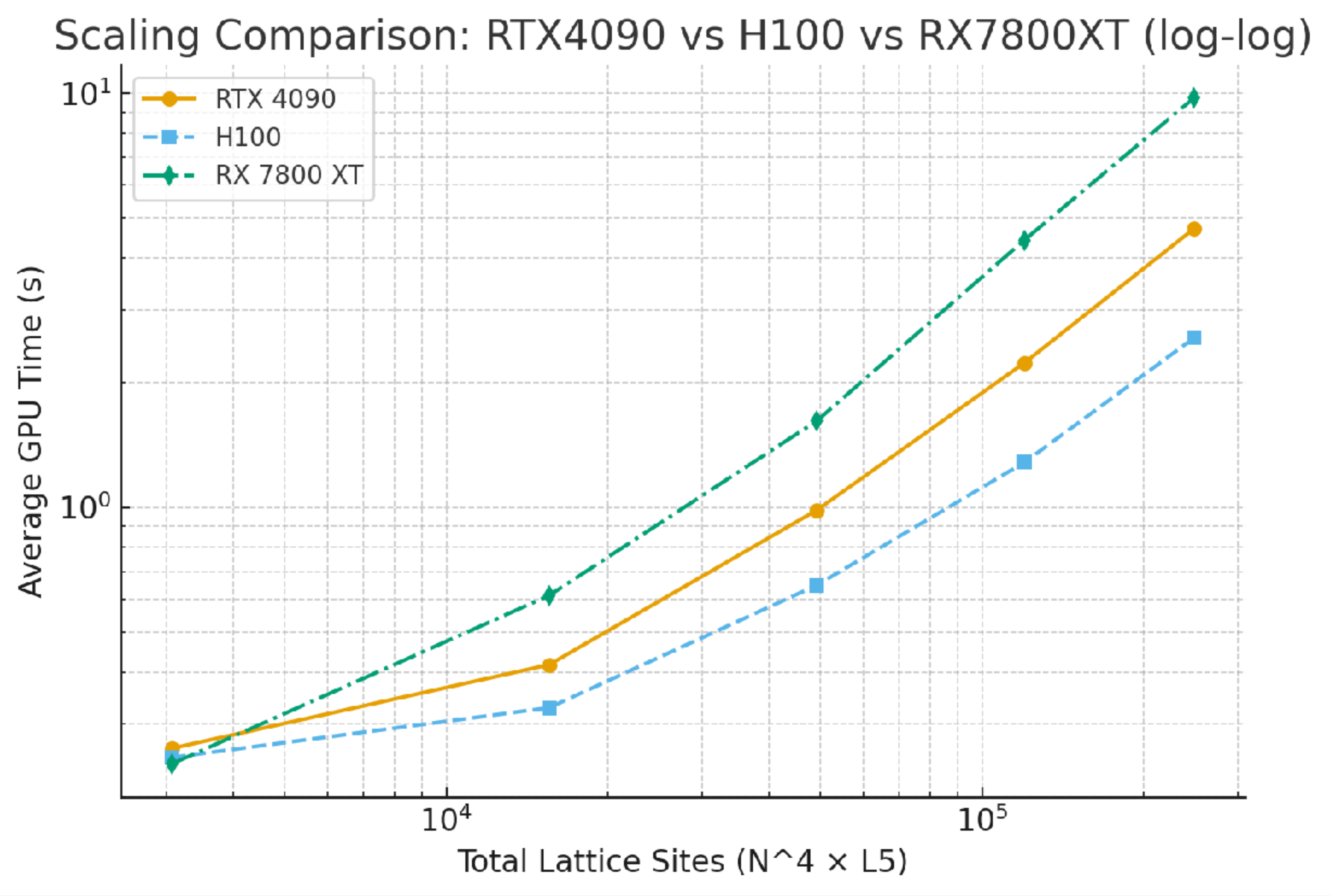
We made LatticeMatrices.jl for general D-dimensional lattice

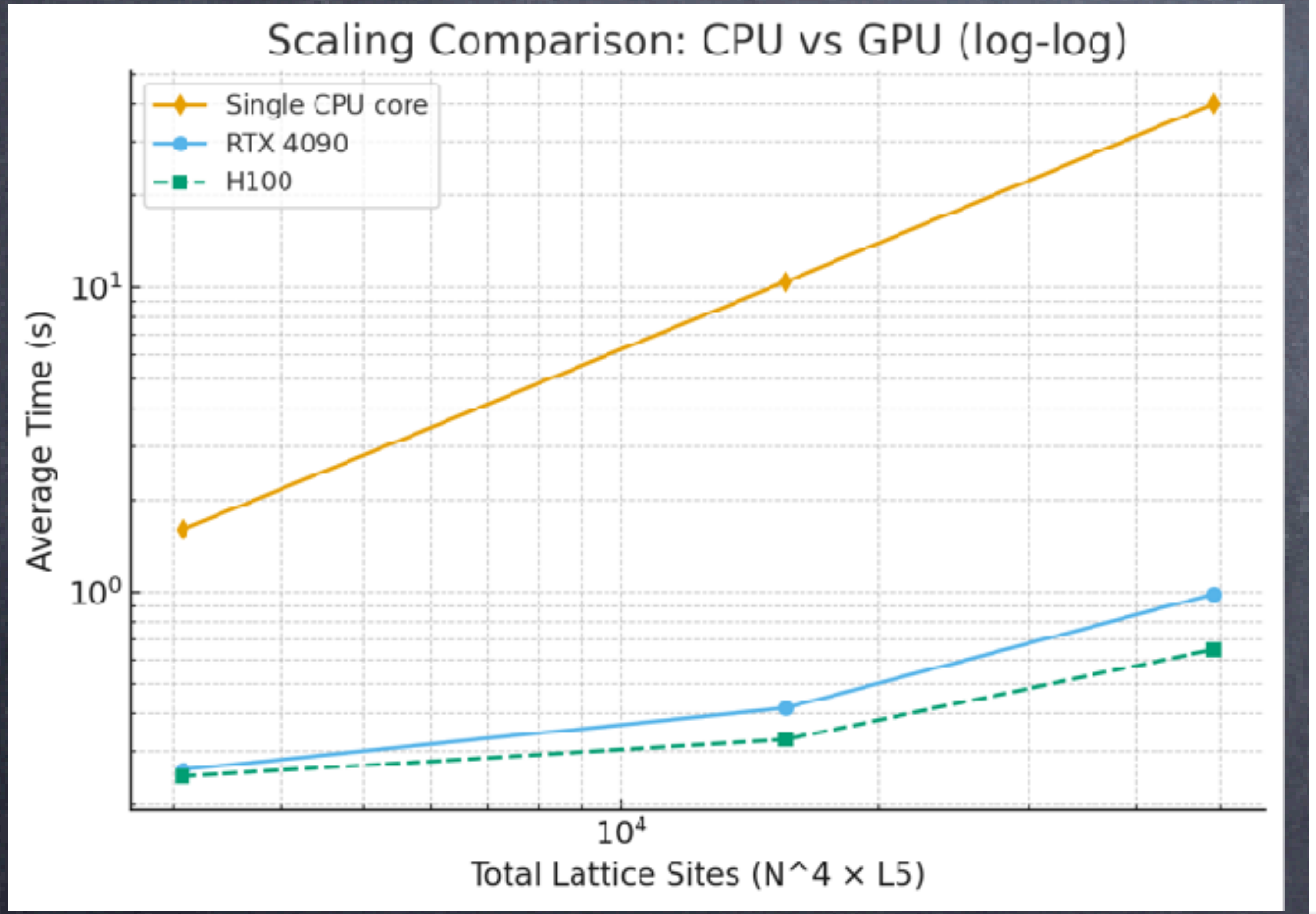
We used JACC.jl for Domainwall fermions

5D lattice (no public GPU code)

note: the computation is bottlenecked by GPU memory access

We want to improve it more...





JACC.jl

一度書けば、スレッド並列、プロセス並列(MPI)、NVIDIA GPU加速、AMD GPU加速、IntelGPU加速、全てに対応したい

-> ハイブリッド並列、マルチGPU並列にも対応したい

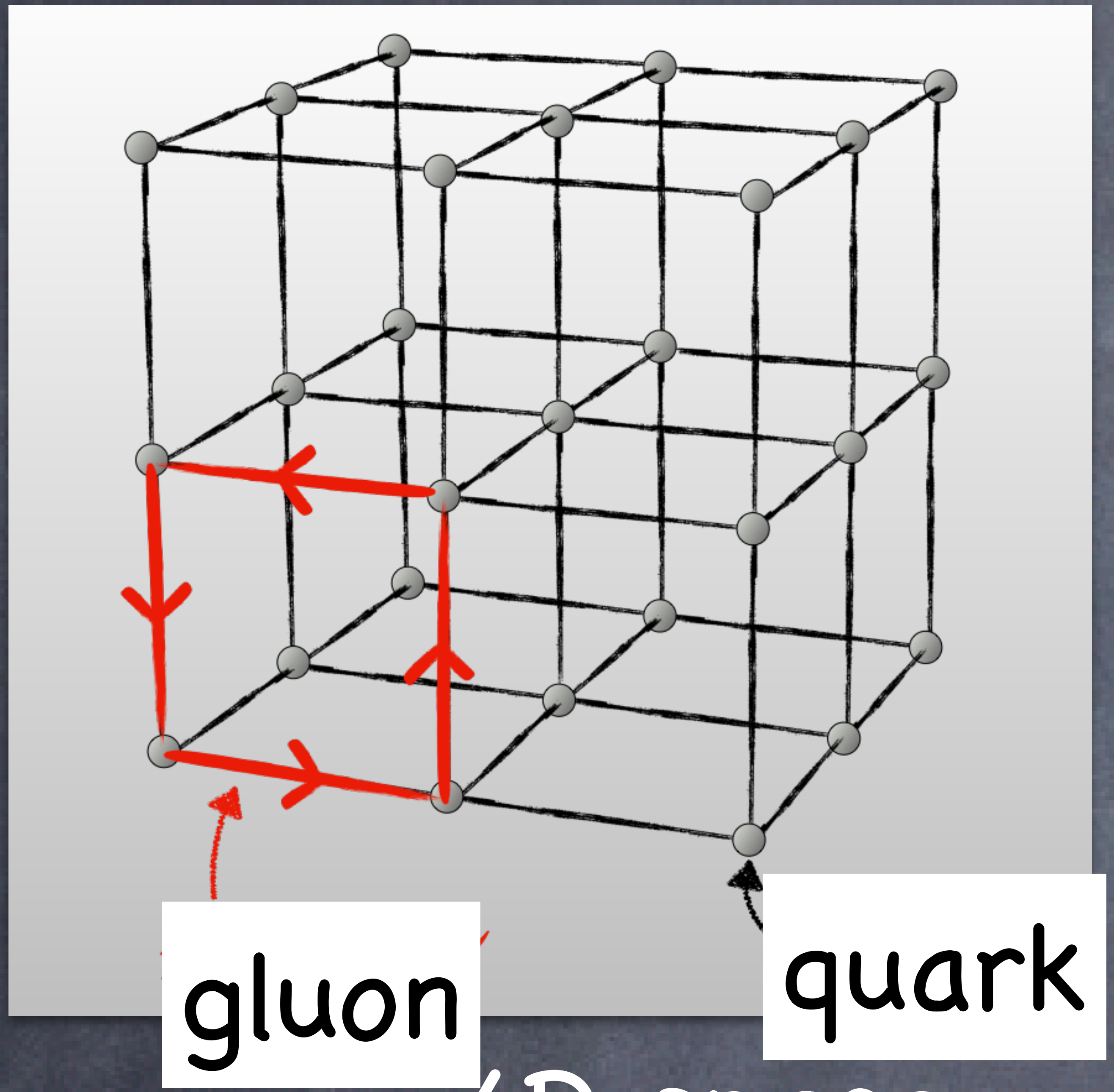
JACC.jlを使えばできる！

```
function LinearAlgebra.mul!(C::LatticeVector{4,T,AT}, A::LatticeVector{4,T,AT}, B::LatticeVector{4,T,AT}) where {T,AT}

    JACC.parallel_for(
        prod(C.PN), kernel_4Dvector_mul!, C.A, A.A, B.A, C.NC, nw, C.PN
    )
    #set_halo!(C)
end
```

```
@inline function kernel_4Dmatrix_mul!(i, C, A, B, ::Val{NC1}, ::Val{NC2}, ::Val{NC3}, ::Val{nw}, PN) where {NC1,NC2,NC3,nw}
    ix, iy, iz, it = get_4Dindex(i, PN)
    @inbounds for jc = 1:NC2
        for ic = 1:NC1
            C[ic, jc, ix+nw, iy+nw, iz+nw, it+nw] = zero(eltype(C))
        end

        for kc = 1:NC3
            b = B[kc, jc, ix+nw, iy+nw, iz+nw, it+nw]
            for ic = 1:NC1
                C[ic, jc, ix+nw, iy+nw, iz+nw, it+nw] += A[ic, kc, ix+nw, iy+nw, iz+nw, it+nw] * b# B[kc, jc, ix+nw, iy+nw, iz+nw, it+nw]
            end
        end
    end
end
```

N次元時空の上に任意の2次元配列を定義できる、LatticeMatrices.jlを開発

Everything is defined on lattice



4D space

We have to consider gluons and quarks

gluons: SU(N) matrix defined on lattice

G(N,N,Lx,Ly,Lz,Lt) : 6D array

quarks: N x NG matrix defined on lattice

F(N,NG,Lx,Ly,Lz,Lt) : 6D array

We need matrix-matrix multiplication on each lattice

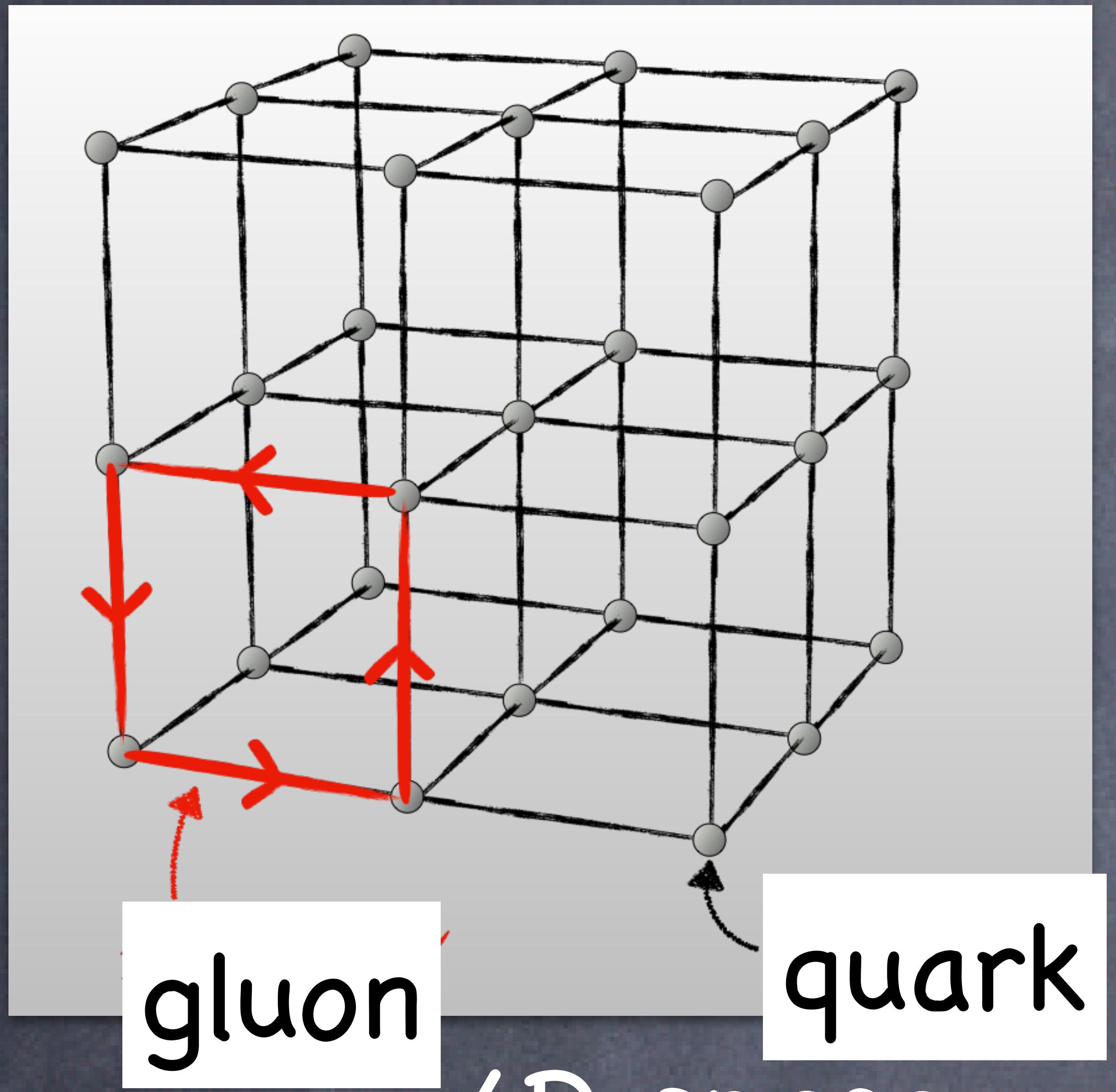
K(:,:,ix,iy,iz,it) = G1(:,:,ix,iy,iz,it)*G2(:,:,ix,iy,iz,it)

We need 4D stencil operation

H(:,:,ix,iy,iz,it) = G1(:,:,ix+1,iy,iz,it)*G2(:,:,ix-1,iy,iz,it)

-> Huge structured sparse systems

Everything is defined on lattice



4D space

We have to consider gluons and quarks

gluons: SU(N) matrix defined on lattice

G(N,N,Lx,Ly,Lz,Lt) : 6D array

quarks: N x NG matrix defined on lattice

F(N,NG,Lx,Ly,Lz,Lt) : 6D array

-> Huge structured sparse systems

CPU threads? MPI?

NVIDIA GPU? Multi-GPU?

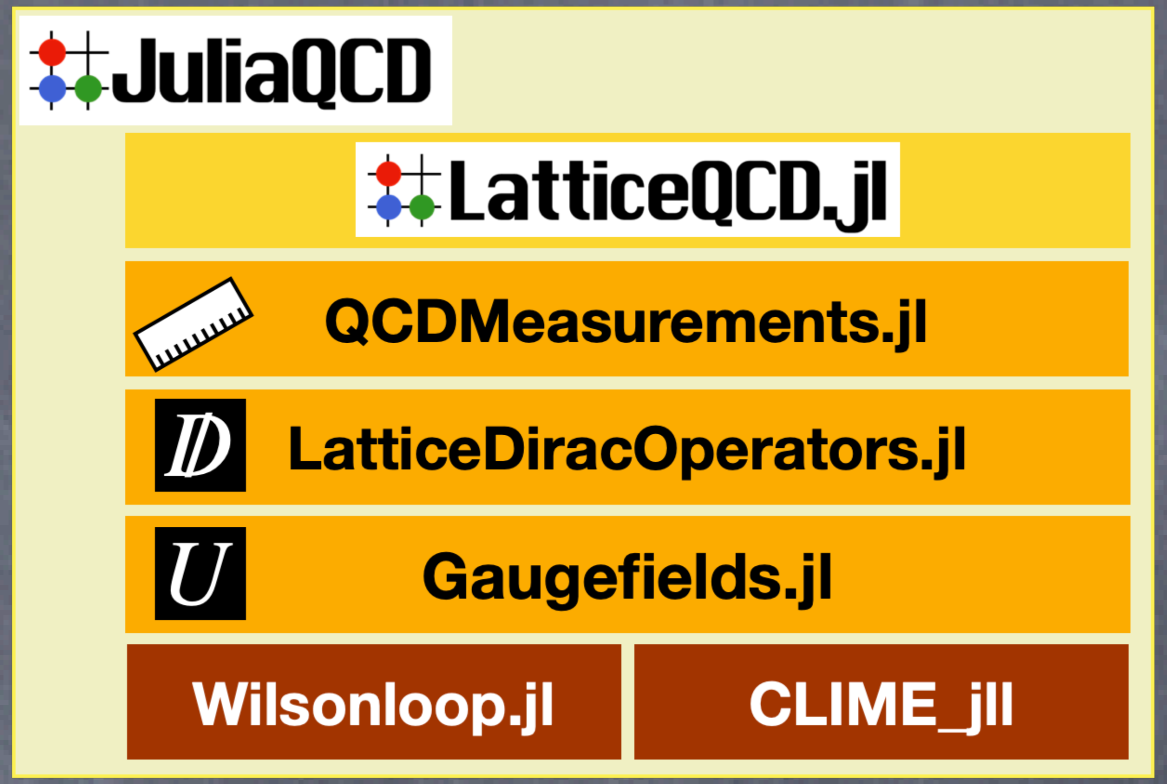
AMD GPU? Supercomputers?

Do we have to write different codes?

We want to consider all!

-> MPI+JACC.jl

LatticeMatrices.jl



A portable lattice linear algebra layer

- Backend abstraction via JACC.jl
- MPI-based domain decomposition
- Same code → CPU / NVIDIA / AMD / multi-GPU clusters
- Designed for lattice-local operations

This layer enables performance portability without code duplication.

new backend for both Gaugefields.jl and LatticeDiracOperators.jl
NxN matrices　　Nx4 matrices

New functionalities are provided via LatticeMatrices.jl
-> automatic differentiation!

JACC.jlその他の進展

JACC.jlの最近の進展

JACC.jl 1.0.0リリース

Feature \ Backend	CPU	CUDA	AMDGPU	Metal	oneAPI
CI	✅	✅	✅	✅	TBD
	x86, Arm GH Runners	RTXA4000, GTX1080	MI100	M1	TBD
Float64	✅	✅	✅	❌	✅ (if supported)
`Multi` (GPU)	N/A	✅	❌	❌	❌
`shared`	N/A	✅	✅	✅	✅
`@atomic`	✅	✅	✅	✅	✅

Multi-GPUに対応する、JACC.Multiもリリース

MPIを使わずに、GPU直接通信でマルチGPU並列を実現

KernelAbstractions.jl

JACC.jl: OpenMP的に、ユーザーはカーネルを意識せずにGPUを使える
C++のKokkosと似たような設計思想

KernelAbstractions.jl

```
@kernel function mul2_kernel(A)
  I = @index(Global)
  A[I] = 2 * A[I]
end
```

```
dev = CPU()
A = ones(1024, 1024)
ev = mul2_kernel(dev, 64)(A, ndrange=size(A))
synchronize(dev)
all(A .== 2.0)
```

```
using CUDA: CuArray
A = CuArray(ones(1024, 1024))
```

```
using AMDGPU: ROCArray
A = ROCArray(ones(1024, 1024))
```

```
using oneAPI: oneArray
A = oneArray(ones(1024, 1024))
```

```
backend = get_backend(A)
mul2_kernel(backend, 64)(A, ndrange=size(A))
synchronize(backend)
all(A .== 2.0)
```

NVIDIAでもAMDでもIntelでもどのGPUでもGPUカーネルを書ける

Differentiable Lattice QCD (AD)

"Lattice Gauge Theory via LLVM-Level Automatic Differentiation"
Y. Nagai, A. Tomiya and H. Ohno, arXiv:2602.20516

Analytic differentiation

In lattice QCD simulation, we have to do HMC (Hamiltonian/Hybrid Monte Carlo)

Molecular dynamics for gauge fields

$$U(\tau + \Delta\tau) = e^{\mathrm{i}\,\Delta\tau\, P(\tau)}\, U(\tau),$$
$$P(\tau + \Delta\tau) = P(\tau) - \Delta\tau\, \mathcal{F}(\tau),$$

variables: $U \in \mathrm{SU}(N_c)$ Lie group; $P \in \mathfrak{su}(N_c)$ Lie algebra on 4D lattice

Hamiltonian: Sg + Sf + tr(P²)/2

Sg: action for gauge fields

Sf: action for fermions

e.g. $S_f(U;\phi) = \phi^\dagger (D^\dagger D)^{-1} \phi,$

We have to solve linear equations

Conventional lattice QCD

Force is derived by hand

sometimes this is very complicated

Complicated action

-> VERY complicated forces

-> automatic differentiation!

Y. Nagai, A. Tomiya and H. Ohno, arXiv:2602.20516

Why AD is difficult in lattice QCD

Difficulties in lattice QCD

Complex valued variables on 4D space-time lattice $U \in \mathrm{SU}(N_c)$

There are many in-place operations to construct an action

PyTorch or JAX: computational graph is broken if there are in-place operations

LLVM-level automatic differentiation

Numerical evaluation of S[U] proceeds through an ordered sequence of operations

$U_0 \longrightarrow U_1 \longrightarrow \cdots \longrightarrow U_N \longrightarrow S,$ regarded as discrete time evaluation of program states

To define a reverse pass without ambiguity, we need a representation in which these successive states are distinguished

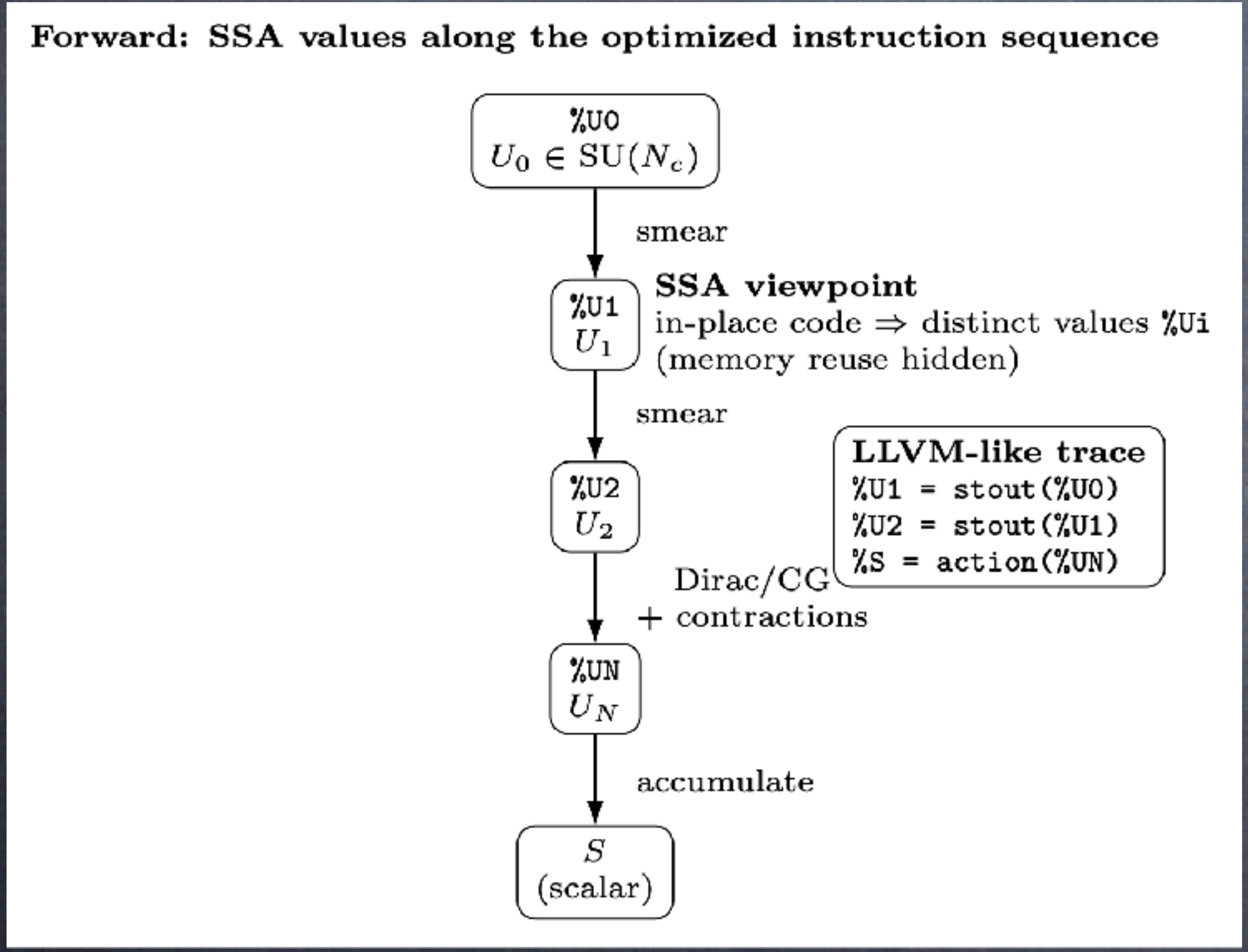
-> LLVM-IR

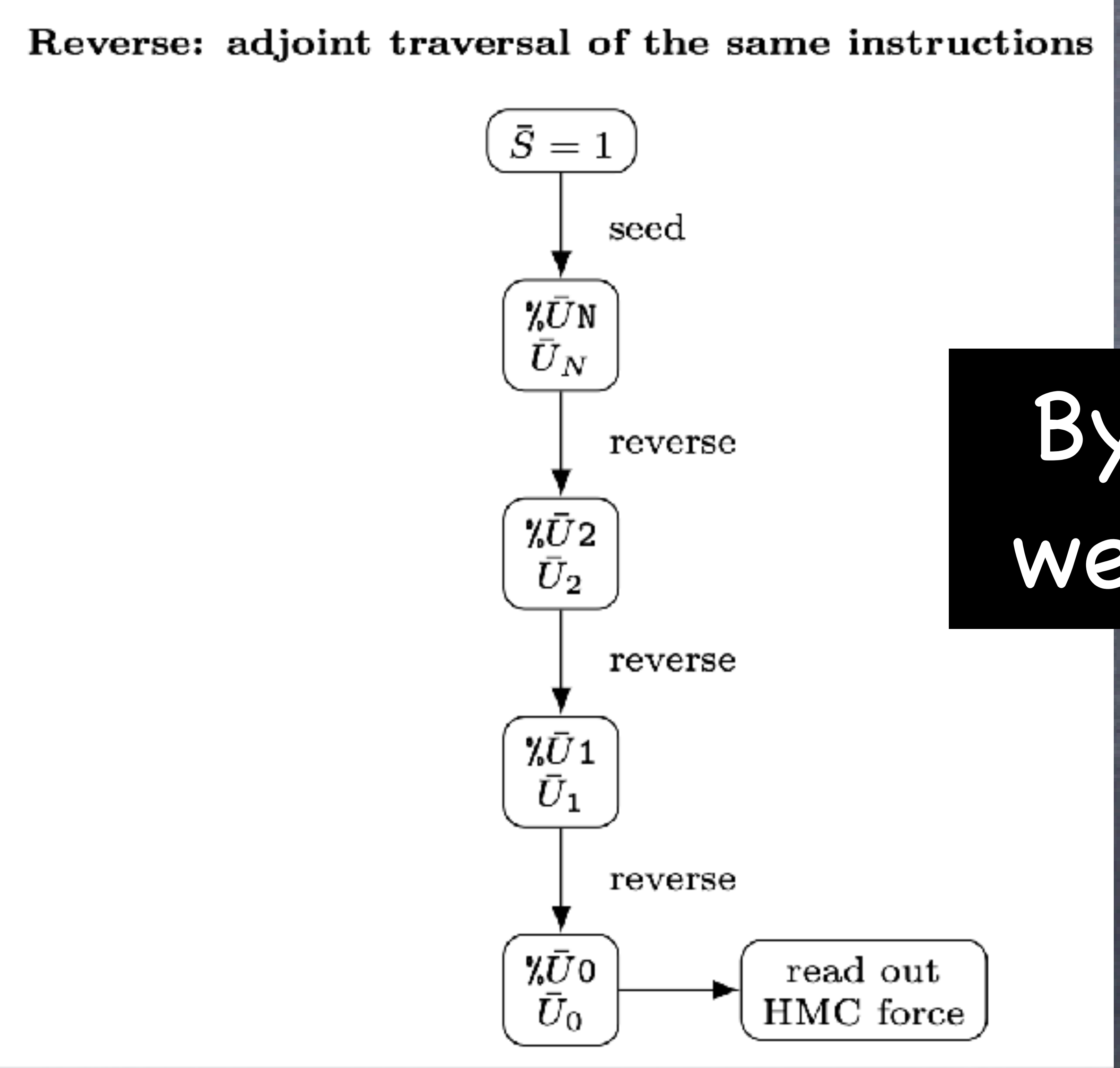
Y. Nagai, A. Tomiya and H. Ohno, arXiv:2602.20516

LLVM-level automatic differentiation

LLVM-level automatic differentiation

LLVM IR is expressed in static single assignment (SSA) form:
each intermediate value is defined exactly once.





By using LLVM-IR, we can have forces

Enzyme.jl can do it!

Y. Nagai, A. Tomiya and H. Ohno, arXiv:2602.20516

AD and custom adjoints

LatticeMatrices.jl can use multi-core or GPUs via JACC.jl

We want to use multi-core or GPUs to obtain forces

In Enzyme.jl, we can design custom Enzyme rules for reverse AD

For example

primal $A_{ij} = \sum_k B_{ik} C_{kj}$ -> parallel for loop via JACC.jl

adjoint of B $\bar{B}_{ik} \equiv \frac{\partial L}{\partial B_{ik}} = \sum_j \bar{A}_{ij} C_{kj}$ -> parallel for loop via JACC.jl

adjoint of C $\bar{C}_{ik} \equiv \frac{\partial L}{\partial C_{kj}} = \sum_i B_{ik} \bar{A}_{ij}$ -> parallel for loop via JACC.jl

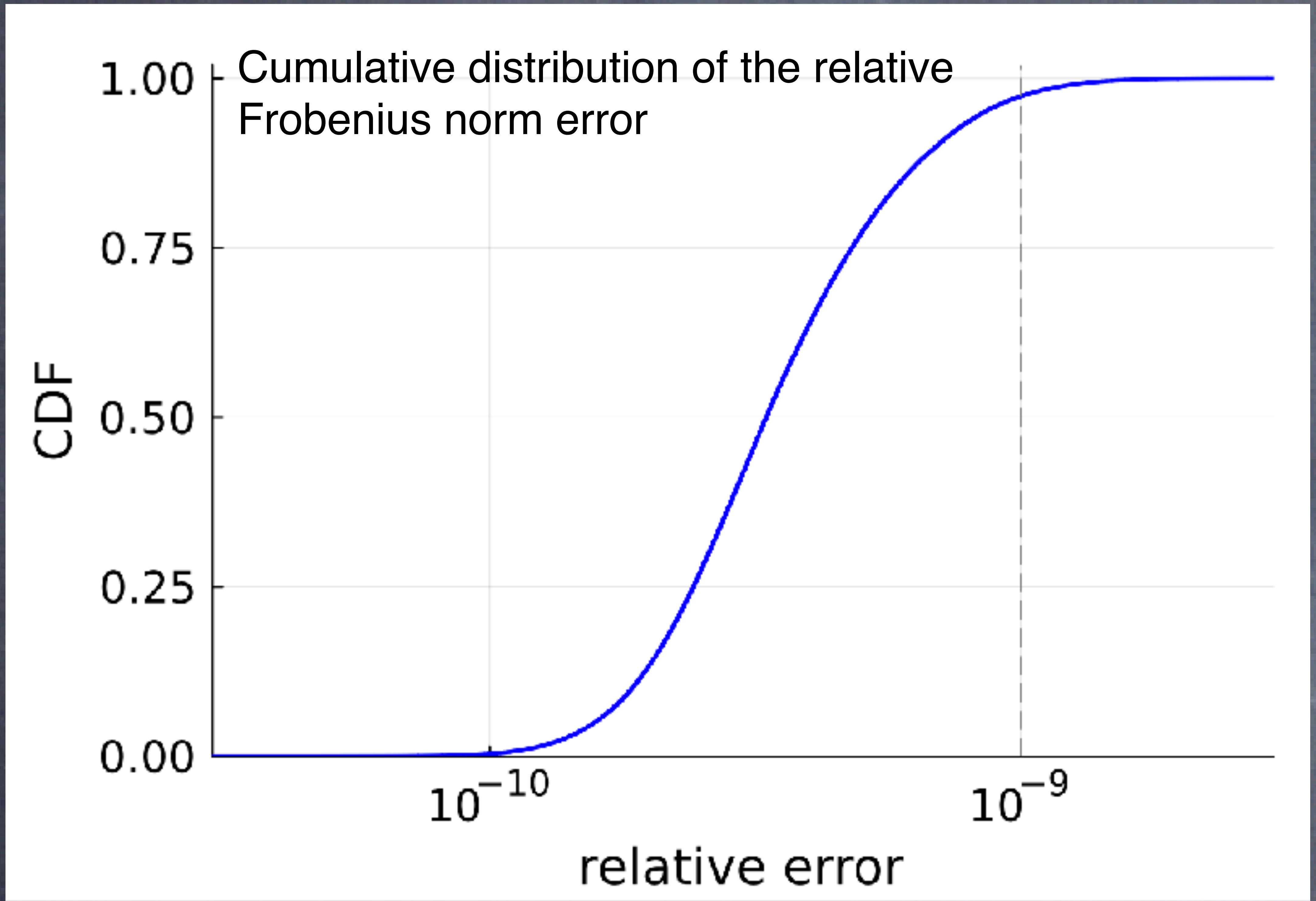
We can use JACC.jl in AD!

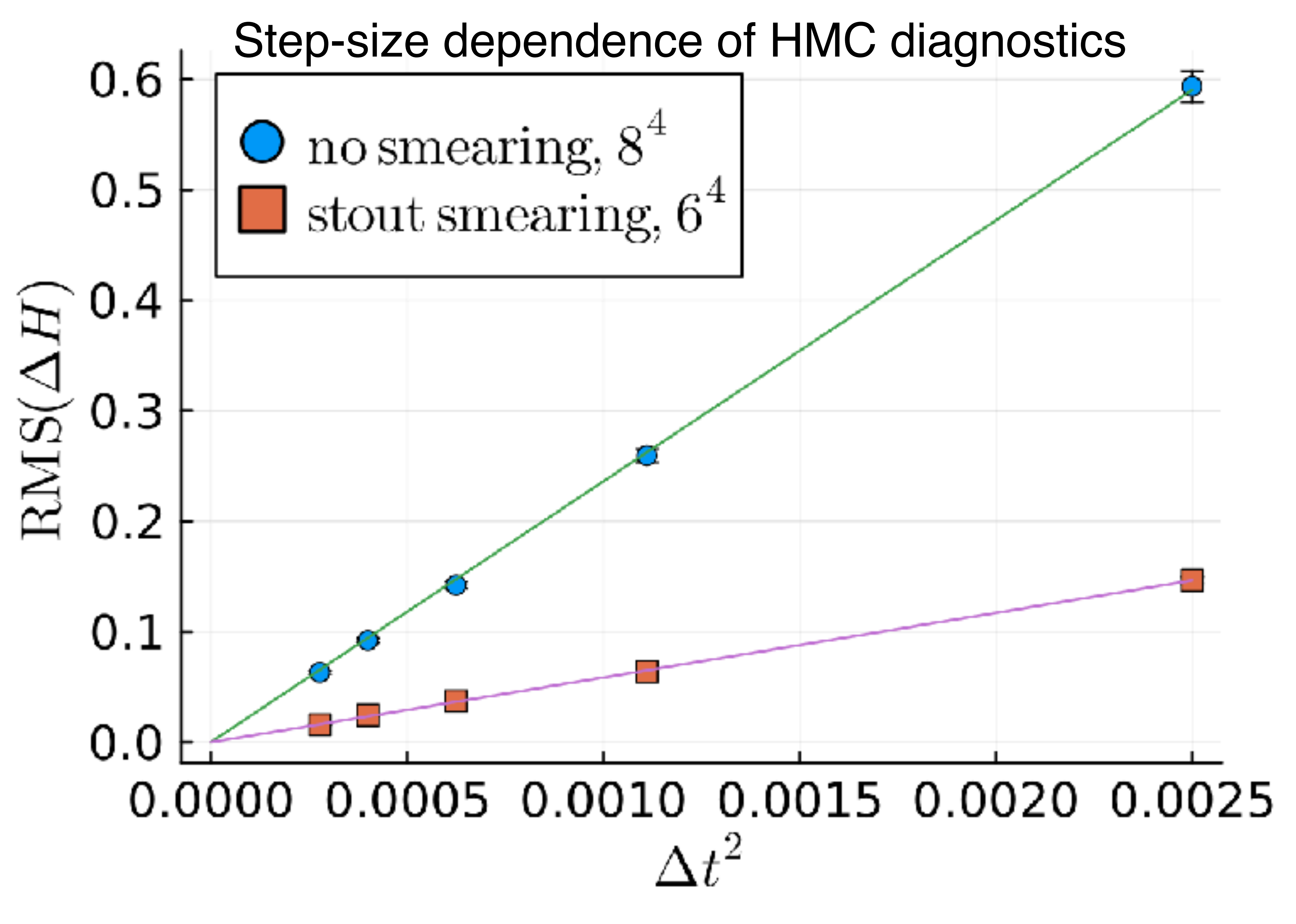
<u>Reverse kernels are parallelized through the same backend abstraction</u>

Benchmarks

Y. Nagai, A. Tomiya and H. Ohno, arXiv:2602.20516

Accuracy





LLVM-level AD is consistent with hand-written implementation

The compiler-generated adjoint faithfully reproduces the required force contributions in this nontrivial setting.

Y. Nagai, A. Tomiya and H. Ohno, arXiv:2602.20516

wall-clock time

TABLE I. Representative wall-clock time (seconds) for a single force evaluation of the Wilson fermion action on a 24^4 lattice. CPU results correspond to single-thread execution. GPU results are obtained on an NVIDIA H100 using JACC.jl.

Implementation	CPU	GPU
Hand-written	195.53	4.61
LLVM-AD	162.50	3.39

Thanks to JACC.jl, we can easily use NVIDIA H100 without changing codes

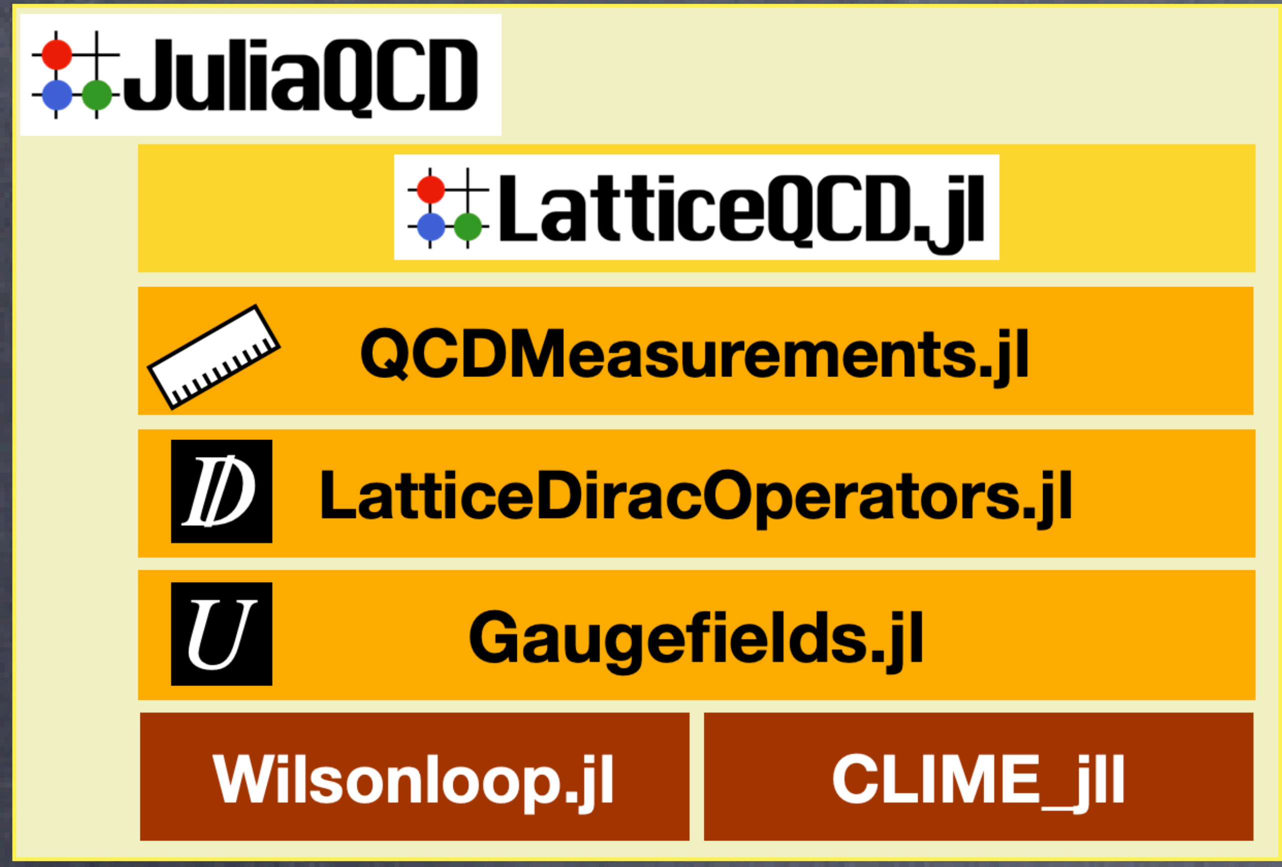
We confirmed that it works with multicore-CPU and AMD GPUs

Summary

Summary

QCD has been simulated on supercomputers

Lattice QCD is a good benchmark for software/hardware



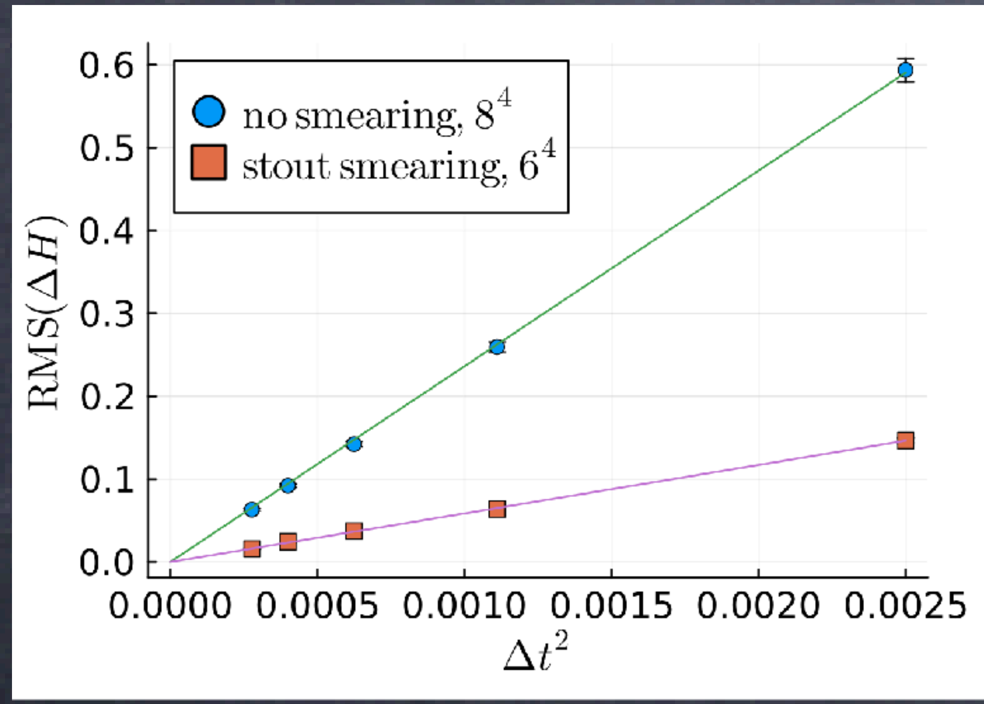
JACC.jl is good!

We made new backend LatticeMatrices.jl

We can treat gluon and quarks with GPUs

We provide LLVM-level automatic differentiation in QCD

Forces can be obtained from actions with many in-place operations



"Lattice Gauge Theory via LLVM-Level Automatic Differentiation"

Y. Nagai, A. Tomiya and H. Ohno, arXiv:2602.20516

Portable and differentiable lattice QCD is now possible through compiler-level design in Julia