

国产超算上的格点QCD与SU(N) Yang-Mills 理论模拟软件研究进展

孙玮

中国科学院高能物理研究所

第六届中国格点量子色动力学研讨会，2026.10.8-11，深圳

QCD like vector theory
QCD
pure Yang-Mills
SU(N)

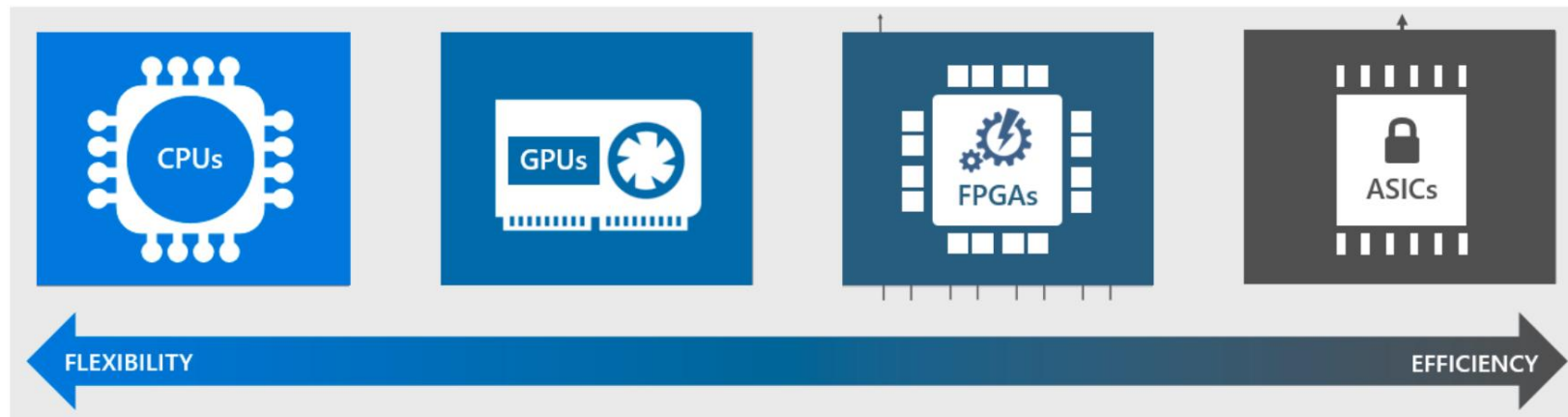
chiral gauge theory
supersymmetric theory

2D and higher dims

Lagrangian formulation
Monte Carlo method

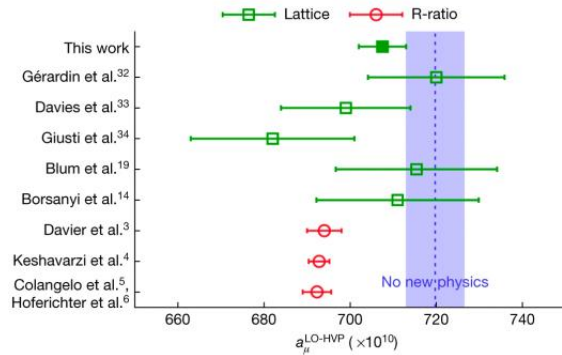


hard work on algorithm and software development



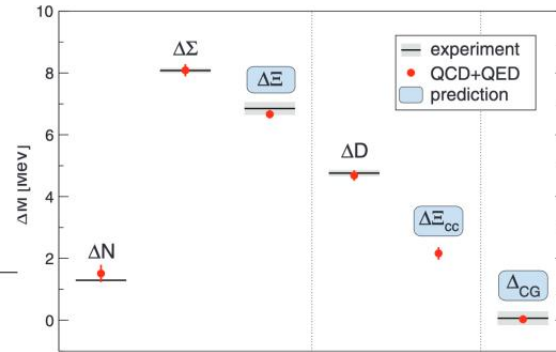
经过近五十年硬件、算法、理论的协同发展，格点场论研究取得了辉煌成就

2021



BMW, Nature 593 (2021)
第一性原理 μ 子反常磁矩计算

2015



BMW, Science 347 (2015)
第一性原理中子质子质量差计算

2007



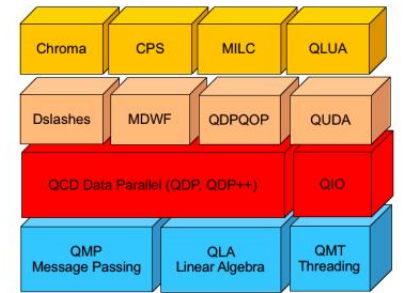
Computer Physics Communications
Volume 177, Issue 8, 15 October 2007, Pages 631-639

Lattice QCD as a video game

Gyöző I. Egri^a, Zoltán Fodor^{a b c}, Christian Hoelbling^b, Sándor D. Katz^b, Kálmán K. Szabó^b

首次使用GPU计算

2004



著名软件框架chroma发布

1974



K. Wilson发明格点规范理论

1980



M. Creutz引入Monte Carlo算法

1987



Physics Letters B

Volume 195, Issue 2, 3 September 1987, Pages 216-222

Hybrid Monte Carlo

Simon Duane¹, A.D. Kennedy, Brian J. Pendleton, Duncan Roweth

包含夸克的HMC算法

90's



**USQCD设计专用超算
QCDSP、QCDoc,
~ 1-10 TFLOPS**

LQCD 软件

美国



- 针对国产超算的尝试：神威太湖之光、曙光DCU、天河迈创等
- 尚未形成系统软件包

中国

Bridge++

日本

tmLQCD

欧洲

Grid

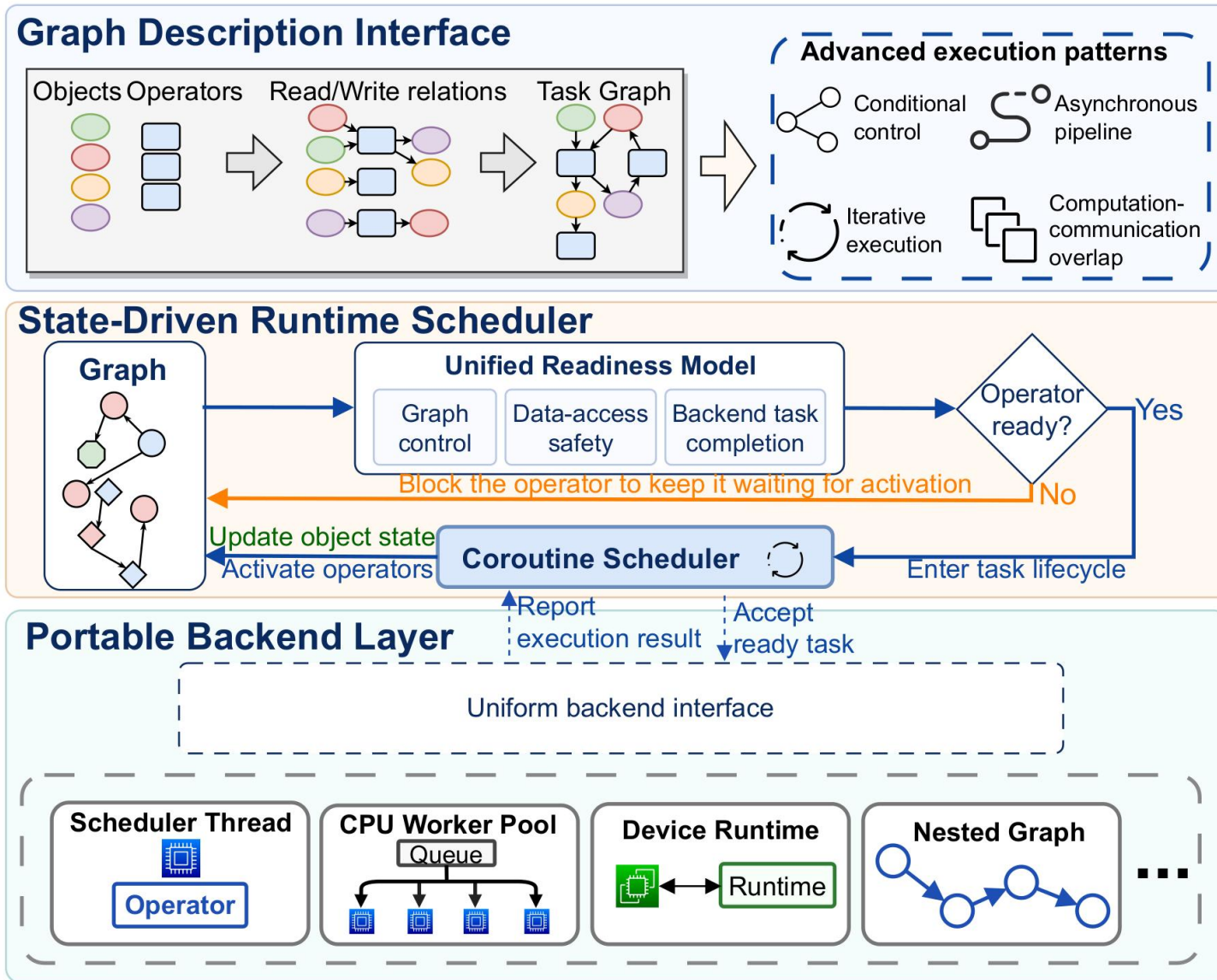
SIMULAT
eQCD

openQCD

LQCD software based on DDQ

DDQ (调度器)

- 宫明、孙玮、刘朝峰等（高能所），韩斌等（北航）
- 暂未开源
- 将计算描述为数据流图
- 形成DDQ脚本编程语言
- 代码执行依赖算子间关系, 无关编写顺序
- kernel函数用C编写，算法用DDQ脚本语言实现（当前版本）
- 实现插件式的跨硬件平台支持



Writing the graph

- ⊙ `[ out ] < op < [ in ... ]` connects outputs, the operator, and inputs
- ⊙ `load_op` pastes a library operator into the current graph
- ⊙ `=` merges two names into one object, it does not copy the buffer
- ⊙ **Scalar field**: the building block of a general field

```
# these lines may be written in any order
```

```
[c] < load_op(sci/lqcd/linalg/scalar/add) < [a b]
```

```
[psi_out] < load_op(sci/lqcd/linalg/wilson/dslash) < [conf psi]
```

Running the graph

- ⊙ The parser builds a **ring**: the operators, their objects, and a memory pool
- ⊙ **ddq_loop** runs every operator whose inputs and outputs are ready
- ⊙ A leaf body is a C function of type **task_f**, loaded from a .so by oplib
- ⊙ `ddqrun example.ddq / mpirun -np N ddqrun example.ddq`

LQCD operators on DDQ

Site types

scalar	real scalar field
cscalar	complex scalar field
vector3c	color vector
matrix3c	3×3 complex matrix
fermw	Wilson spinor, 4 spin × 3 color
gauge	four matrix3c links per site

Physics operators

wilson	D slash, M , $M^\dagger$, $M^\dagger M$
clover	field strength and clover term
inverter	CG, BiCGStab, MR
mpi_nd	halo exchange and allreduce
io	QIO gauge input

Example: scalar field operators

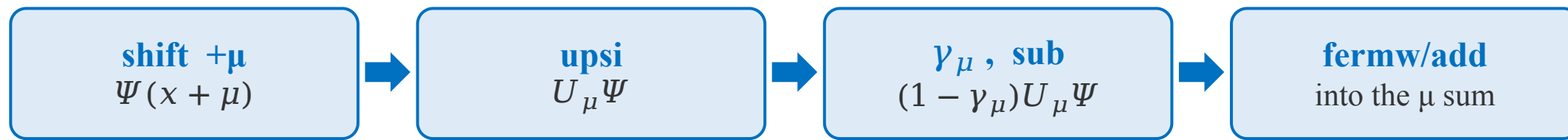
```
load_op(std/real/print) < [ s1 ] < load_op(sci/lqcd/linalg/scalar/sum) < [ p1 ]  
    < load_op(sci/lqcd/linalg/scalar/mul) < [ a b ]  
  
[ a ] < load_op(sci/lqcd/linalg/scalar/init) < [ 2.0 ]  
[ b ] < load_op(sci/lqcd/linalg/scalar/init) < [ 3.0 ]  
  
load_op(std/real/print) < [ s2 ] < load_op(sci/lqcd/linalg/scalar/sum) < [ p2 ]  
    < load_op(sci/lqcd/linalg/scalar/real_mul) < [ 1.0 b ]  
  
[ c ] < load_op(sci/lqcd/linalg/scalar/copy) < [ b ]  
load_op(std/real/print) < [ s3 ] < load_op(sci/lqcd/linalg/scalar/sum) < [ c ]
```

Wilson Dslash

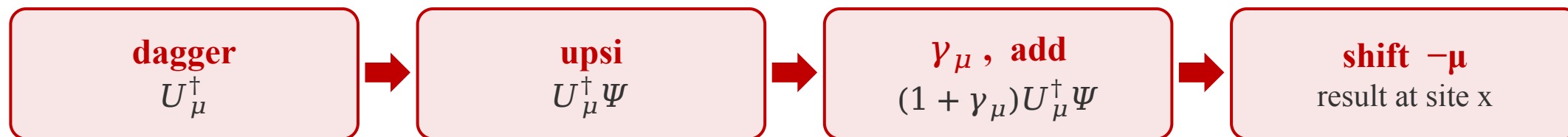
⊙ $D\Psi = \sum_{\mu} [(1 - \gamma_{\mu})U_{\mu}(x)\Psi(x + \mu) + (1 + \gamma_{\mu})U_{\mu}^{\dagger}(x - \mu)\Psi(x - \mu)]$

⊙ Each hop is a chain of **shift, upsi, γ_{μ} , dagger, add, sub**

forward



backward



Dslash in DDQ (part)

```
[ out ] < newop < [ conf psi ]
// apply Wilson Dslash operator with gauge configuration [ conf ]
// (a matrix3c field with 4 spacetime dimension)
// to a Wilson type quark field [ psi ] (4 spin, 3 color complex field)
// [ out ] is a Wilson type quark field
[ conf ] < load_op(sci/lqcd/linalg/gauge/const)
// dir 1
// forward direction
[ psip1 ] < load_op(sci/lqcd/linalg/fermw/shift) < [ psi 1 ]
[ glpsip1 ] < load_op(sci/lqcd/linalg/fermw/glmul) < [ ulpsip1 ]
           < load_op(sci/lqcd/linalg/fermw/upsi) < [ conf.1 psip1 ]
[ outp1 ] < load_op(sci/lqcd/linalg/fermw/sub) < [ ulpsip1 glpsip1 ]
// backward direction
[ ulm1dagg ] < load_op(sci/lqcd/linalg/matrix3c/dagger) < [ conf.1 ]
[ glpsim1 ] < load_op(sci/lqcd/linalg/fermw/glmul) < [ ulpsim1 ]
           < load_op(sci/lqcd/linalg/fermw/upsi) < [ ulm1dagg psi ]
[ tm1 ] < load_op(sci/lqcd/linalg/fermw/add) < [ ulpsim1 glpsim1 ]
[ outm1 ] < load_op(sci/lqcd/linalg/fermw/shift) < [ tm1 -1 ]
// sum over forward and backward
[ out1 ] < load_op(sci/lqcd/linalg/fermw/add) < [ outp1 outm1 ]
```

Example: CG inverter

```
// fermw field
[ b ] < load_op(sci/lqcd/linalg/fermw/point_slc1) < [ 0 0 0 0 ]

// make up a gauge configuration
[ conf ] < load_op(sci/lqcd/linalg/gauge/init_unit)

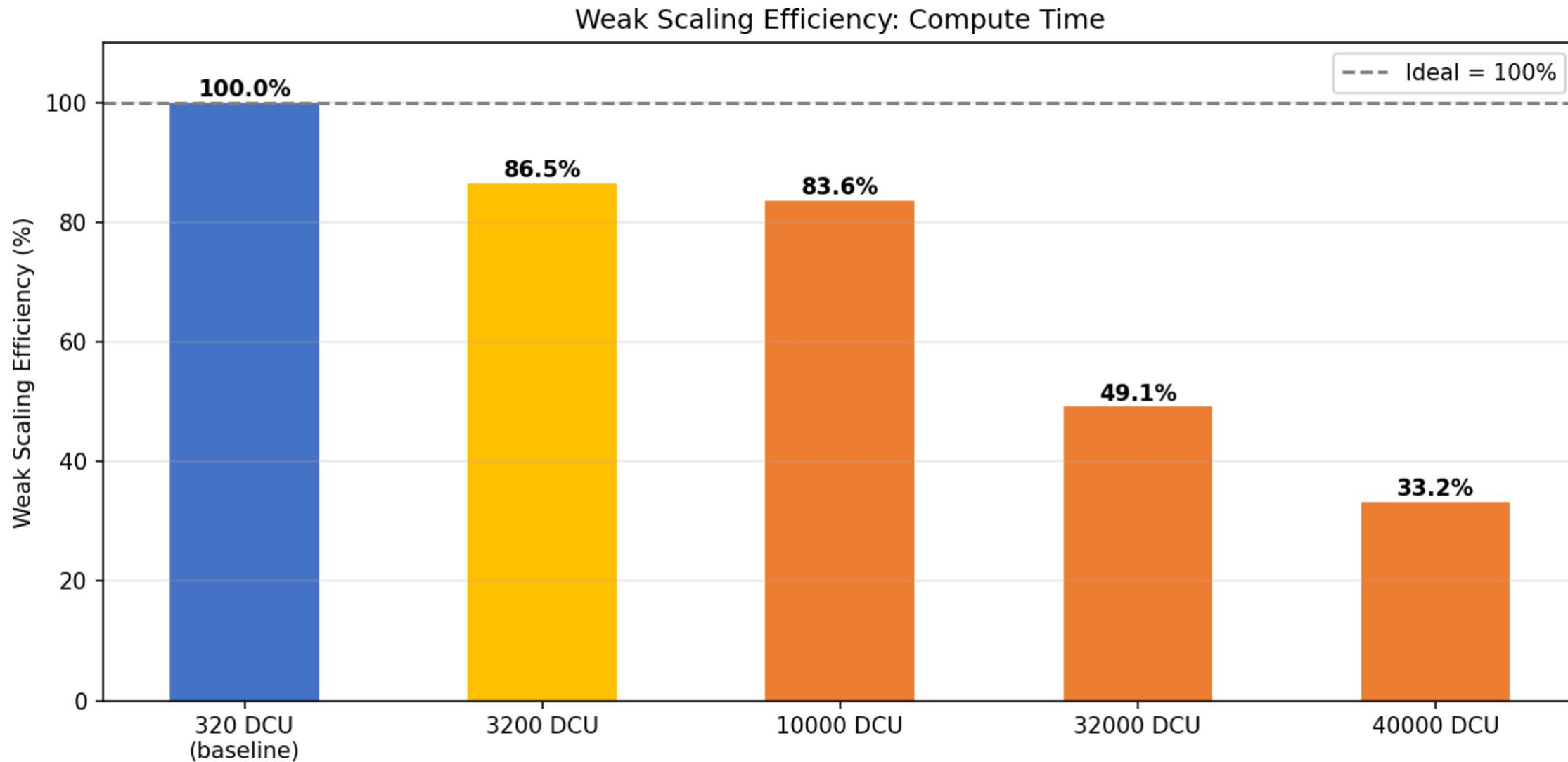
invparam.mass = 0.123
invparam.tol = 1e-10
invparam.max_iter = 100

// CG solves  $M x = b$ 
[ out ] < load_op(sci/lqcd/linalg/inverter/cg) < [ invparam conf b ]
[ outsum ] < load_op(sci/lqcd/linalg/cscalar/sum) < [ temp ]
           < load_op(sci/lqcd/linalg/fermw/sum) < [ out ]

load_op(std/print) < [ "outsum = %f + %fi\n" outsum.re outsum.im ]
```

Weak scaling on DCUs

Wilson fermion with CG inverter



SU(N) pure Yang-Mills lattice gauge theory

⊙ (anisotropic) Wilson pure gauge action

$$S_W[U] = \beta \left[\frac{1}{\xi_0} \sum_x \sum_{i < j} \left(1 - \frac{1}{N} \text{ReTr} U_{\mu\nu}(x) \right) + \xi_0 \sum_x \sum_i \left(1 - \frac{1}{N} \text{ReTr} U_{i0}(x) \right) \right]$$

⊙ Cabibbo-Marinari pseudo-heatbath (Phys. Lett. B 119 (1982))

➤ SU(2): heatbath algorithm

➤ General SU(N): no heatbath algorithm, use $\frac{N(N-1)}{2}$ SU(2) subgroup of SU(N)

➤ Calculate staple to update gauge links

$$\Sigma_\mu(x) = \sum_{\nu \neq \mu} w_{\mu\nu} [U_\nu(x + \mu) U_\mu^\dagger(x + \nu) U_\nu^\dagger(x) + U_\nu^\dagger(x + \mu - \nu) U_\mu^\dagger(x - \nu) U_\nu(x - \nu)]$$

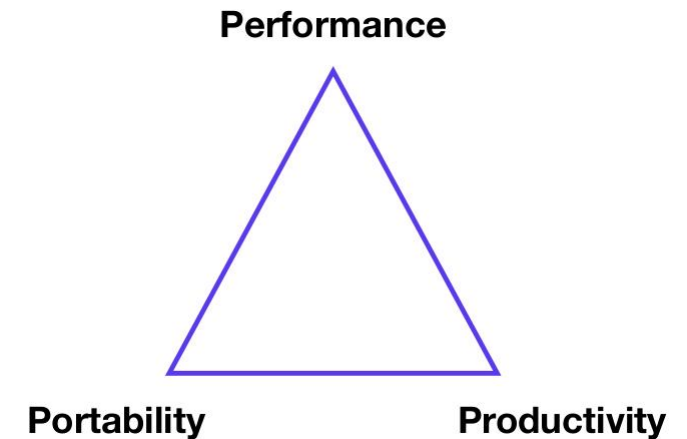
⊙ Overrelaxation

➤ Reduce autocorrelation

➤ Combined with pseudo-heatbath (cannot be used alone)

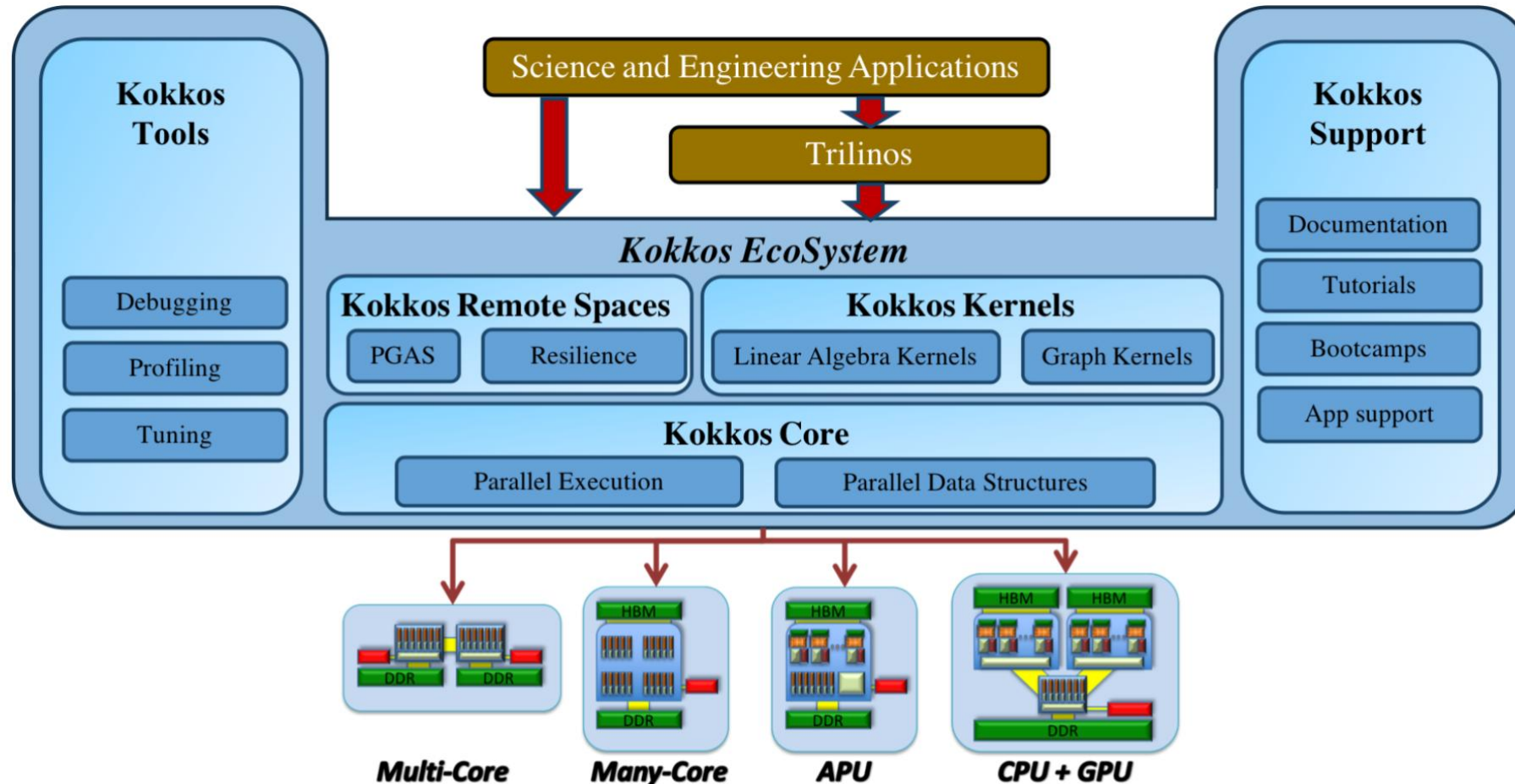
kwqft

- ⊙ Performance portable **Kokkos** implementation of Wilson pure gauge Monte Carlo simulation
- ⊙ Support **SU(N)** Yang-Mills theory in an **arbitrary** number of space-time dimensions
- ⊙ **“Write once, run everywhere”**
- ⊙ The same code runs efficiently on both CPUs and GPUs
 - CPU (using Serial or OpenMP backend)
 - **NVIDIA GPU (using CUDA backend)**
 - **AMD GPU (using HIP backend)**
 - **Intel GPU (using SYCL backend)**
- ⊙ C++20, SIMD vectorization on CPU, MPI parallel for large scale simulation
- ⊙ <https://github.com/sunwayihep/kwqft>
- ⊙ [arXiv: 2610.09766](https://arxiv.org/abs/2610.09766)



Kokkos

- ◉ Implements a programming model in C++ for writing performance portable applications
- ◉ It's a C++ library, not a new language



Kokkos View

- ⊙ A Kokkos view is a multidimensional array
- ⊙ Automatically dealloc the views through reference counting
- ⊙ Create views easily

```
const size_t L1 = 32;  
const size_t L2 = 32;  
const size_t L3 = 64;  
Kokkos::View<double***> myarray("a label", L1, L2, L3);
```

- ⊙ Views have layout: **LayoutLeft** (column-major), **LayoutRight** (row-major)
- ⊙ Views have memory space

```
Kokkos::View<double***, Kokkos::CudaSpace> myarray("a label", L1, L2, L3);
```

Kokkos parallel dispatch

- ◉ Kokkos supports `parallel_for`, `parallel_reduce`, `parallel_scan` execution patterns

Serial

```
for (int64_t i = 0; i < N; ++i) {  
    /* loop body */  
}
```

OpenMP

```
#pragma omp parallel for  
for (int64_t i = 0; i < N; ++i) {  
    /* loop body */  
}
```

Kokkos

```
parallel_for(N, [=] (const int64_t i) {  
    /* loop body */  
});
```

Skeleton of Kokkos usage

- Compile with `KOKKOS_ENABLE_OPENMP` for CPU and `KOKKOS_ENABLE_CUDA` for GPU

```
#include <Kokkos_Core.hpp>
int main(int argc, char** argv){
    Kokkos::initialize(argc,argv);
    const size_t M=N=100;
    Kokkos::View<double*> x("x",M); Kokkos::View<double*> y("y",N);
    Kokkos::View<double**> A("A",M,N);

    for(int i=0; i<M; i++) x(i)=1.0; for(int i=0; i<N; i++) y(i)=1.0;
    for(int i=0; i<M; i++)for(int j=0; j<N; j++) A(i,j)=1.0;

    double result=0;

    Kokkos::parallel_reduce("xAy", N, KOKKOS_LAMBDA(int i, double& update){
        double temp = 0;
        for(int j = 0; j < M; j++){
            temp += A(i, j) * y(j);
        }
        update += x(i) * temp;
    }, result);
    Kokkos::finalize();
}
```

Back to kwqft: various combination of build configurations

CPU:
MPI+OpenMP+SIMD

```
cmake .. -DKWQFT_NCOLORS=${NCOLORS} -DKWQFT_NDIMS=${NDIMS} \  
-DCMAKE_BUILD_TYPE=Release \  
-DKWQFT_ENABLE_OPENMP=ON \  
-DKWQFT_USE_MPI=ON \  
-DKokkos_ARCH_NATIVE=ON \  
-DKWQFT_ENABLE_HOST_SIMD=ON \  
-DKOKKOS_SOURCE_DIR="../../kokkos/kokkos" # path to Kokkos source directory
```

NVIDIA GPU:
MPI+CUDA

```
cmake .. -DKWQFT_NCOLORS=${NCOLORS} -DKWQFT_NDIMS=${NDIMS} \  
-DCMAKE_BUILD_TYPE=Release \  
-DKWQFT_ENABLE_CUDA=ON \  
-DKWQFT_USE_MPI=ON \  
-DKokkos_ARCH_AMPERE80=ON \  
-DKOKKOS_SOURCE_DIR="../../kokkos/kokkos" # path to Kokkos source directory
```

SIMD vectorization

● Arm: NEON, SVE-512

● x86: AVX2, AVX-512

● ~ 200 lines of architecture specific
SIMD code (load and store)

```
template <typename Simd, typename Real>
KOKKOS_IMPL_HOST_FORCEINLINE_FUNCTION void
loadContiguous(const Complex<Real> *p, Simd &re, Simd &im) {
    [[maybe_unused]] const Real *r = reinterpret_cast<const Real *>(p);
    #if defined(KOKKOS_ARCH_ARM_SVE)
        if constexpr (is_abi_v<Simd, double,
                        kx::simd_abi::sve_fixed_size<SVE_DOUBLES_IN_VECTOR>>) {
            const svfloat64x2_t v = svld2_f64(svptrue_b64(), r);
            re = Simd(static_cast<vls_float64_t>(svget2_f64(v, 0)));
            im = Simd(static_cast<vls_float64_t>(svget2_f64(v, 1)));
            return;
        }
        if constexpr (is_abi_v<Simd, float,
                        kx::simd_abi::sve_fixed_size<SVE_WORDS_IN_VECTOR>>) {
            const svfloat32x2_t v = svld2_f32(svptrue_b32(), r);
            re = Simd(static_cast<vls_float32_t>(svget2_f32(v, 0)));
            im = Simd(static_cast<vls_float32_t>(svget2_f32(v, 1)));
            return;
        }
    #endif
    #if defined(KOKKOS_ARCH_AVX512XEON)
        if constexpr (is_abi_v<Simd, double, kx::simd_abi::avx512_fixed_size<8>>) {
            const __m512d a = _mm512_loadu_pd(r);
            const __m512d b = _mm512_loadu_pd(r + 8);
            re = Simd(_mm512_permutex2var_pd(
                a, _mm512_setr_epi64(0, 2, 4, 6, 8, 10, 12, 14), b));
            im = Simd(_mm512_permutex2var_pd(
                a, _mm512_setr_epi64(1, 3, 5, 7, 9, 11, 13, 15), b));
            return;
        }
    }
```

Verifications: two dimensions

- The 2D SU(N) pure Yang-Mills theory is exactly solvable

- Partition function ($x = \frac{\beta}{N}$)

$$Z_N(x) = \sum_{q=-\infty}^{\infty} D_N^{(q)}(x),$$

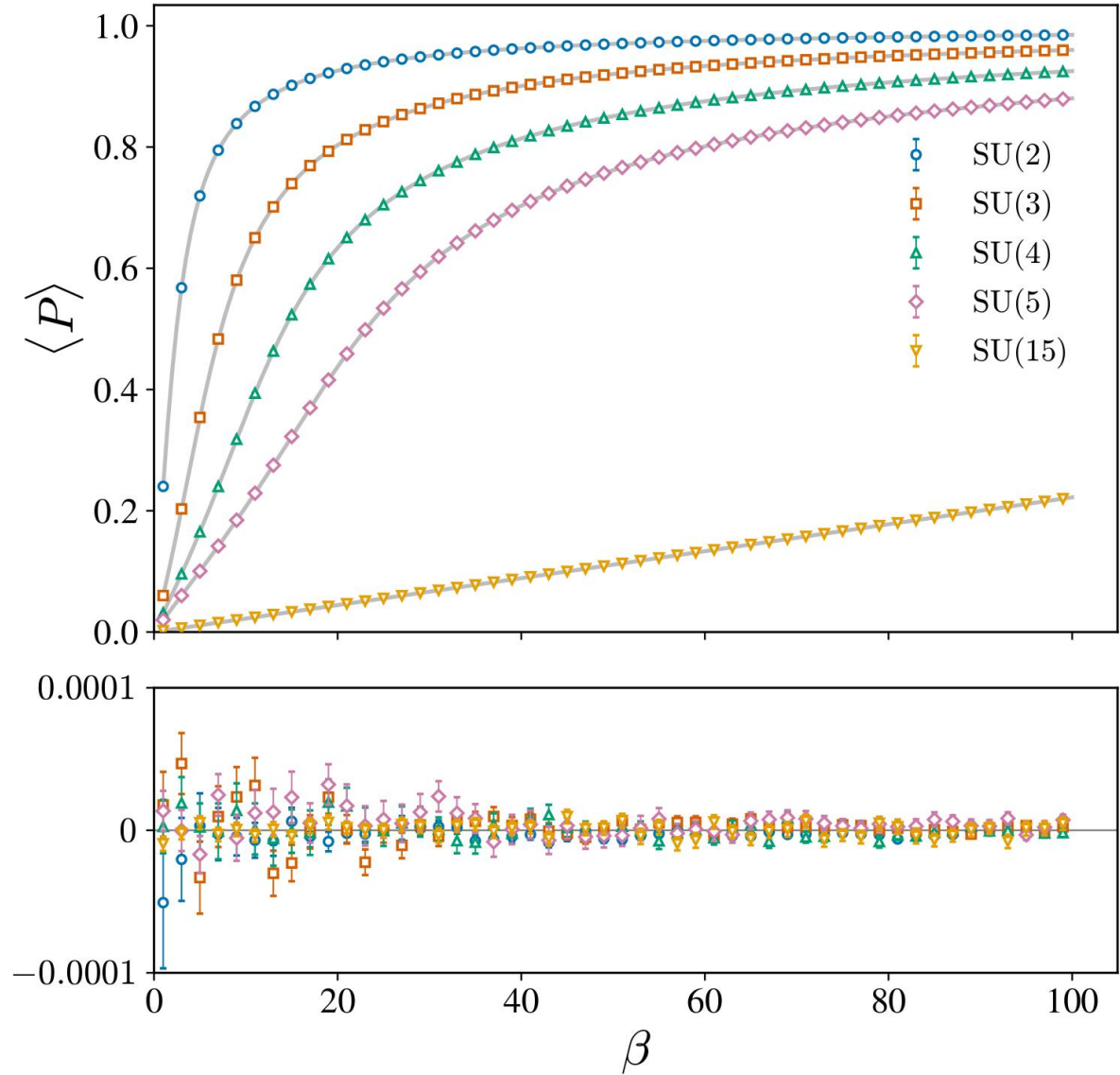
$$D_N^{(q)}(x) = \det_{0 \leq i, j < N} I_{q+i-j}(x)$$

- $I_n(x)$ is the modified Bessel function of the first kind

- Plaquette expectation value

$$\langle P_N \rangle = \frac{1}{N} \frac{d}{dx} \ln Z_N(x) = \frac{1}{N} \frac{\sum_q (D_N^{(q)}(x))'}{\sum_q D_N^{(q)}(x)},$$

- For SU(2): $\langle P_2 \rangle = I_2(\beta)/I_1(\beta)$



Verifications: three and four dimensions

2+1D

N	β	Lattice	$\langle P \rangle$, Ref. [32]	$\langle P \rangle$, kwqft
2	6	48^3	0.8247839(16)	0.8247783(40)
2	16	320^3	0.93649802(5)	0.93649802(7)
2	32	320^3	0.96851172(2)	0.96851176(4)
3	12	24^3	0.7582875(8)	0.7582951(101)
3	24	48^3	0.8849014(11)	0.8849019(14)
3	80	128^3	0.96634760(6)	0.96634748(10)
4	24	48^3	0.7742299(7)	0.7742292(27)
4	50	64^3	0.8966907(4)	0.8966914(7)
4	100	128^3	0.94923340(4)	0.94923347(11)
5	40	128^3	0.7838764(5)	0.7838765(5)
5	100	128^3	0.91791074(8)	0.91791095(14)
5	180	160^3	0.95494414(5)	0.95494409(5)
8	100	96^3	0.7714494(5)	0.7714526(5)
8	300	96^3	0.92839623(11)	0.92839588(12)
8	460	128^3	0.95368978(5)	0.95368987(5)

3+1D

N	β	Lattice	$\langle P \rangle$, Ref. [33]	$\langle P \rangle$, kwqft
2	2.4645	$32^3 \times 64$	0.644718(3)	0.644717(3)
3	6.0175	$32^3 \times 64$	0.595595(3)	0.595596(2)
4	11.0280	$24^3 \times 48$	0.578794(2)	0.578791(3)
5	17.5350	$24^3 \times 48$	0.573069(2)	0.573069(2)
6	25.4520	$24^3 \times 48$	0.568682(2)	0.568684(2)
7	34.8343	$24^3 \times 48$	0.566504(2)	0.566504(2)
17	208.45	$12^3 \times 24$	0.562729(4)	0.562733(3)

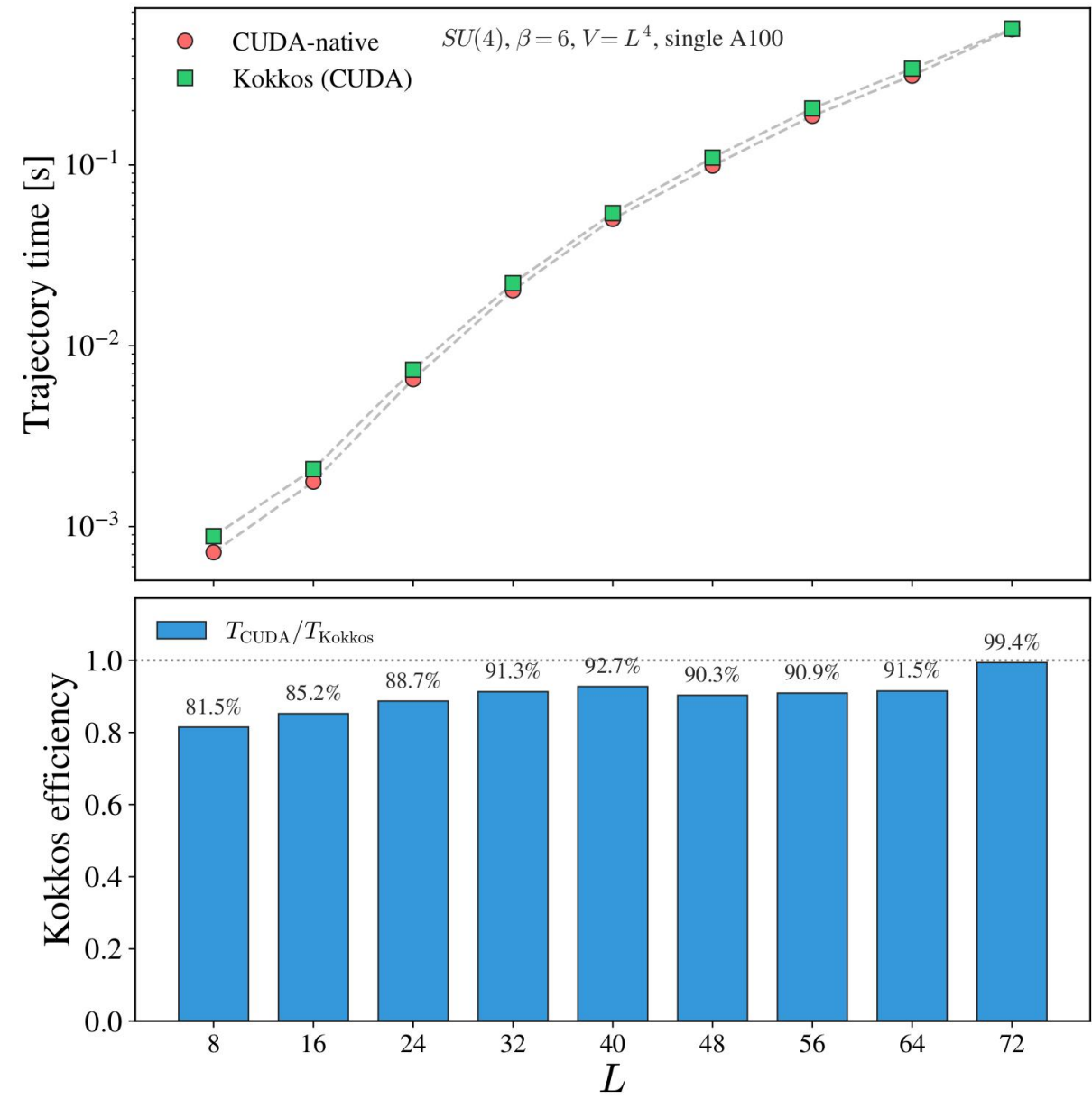
● Ref.[32]: arXiv: hep-lat/0609015

● Ref.[33]: arXiv: 1304.4437

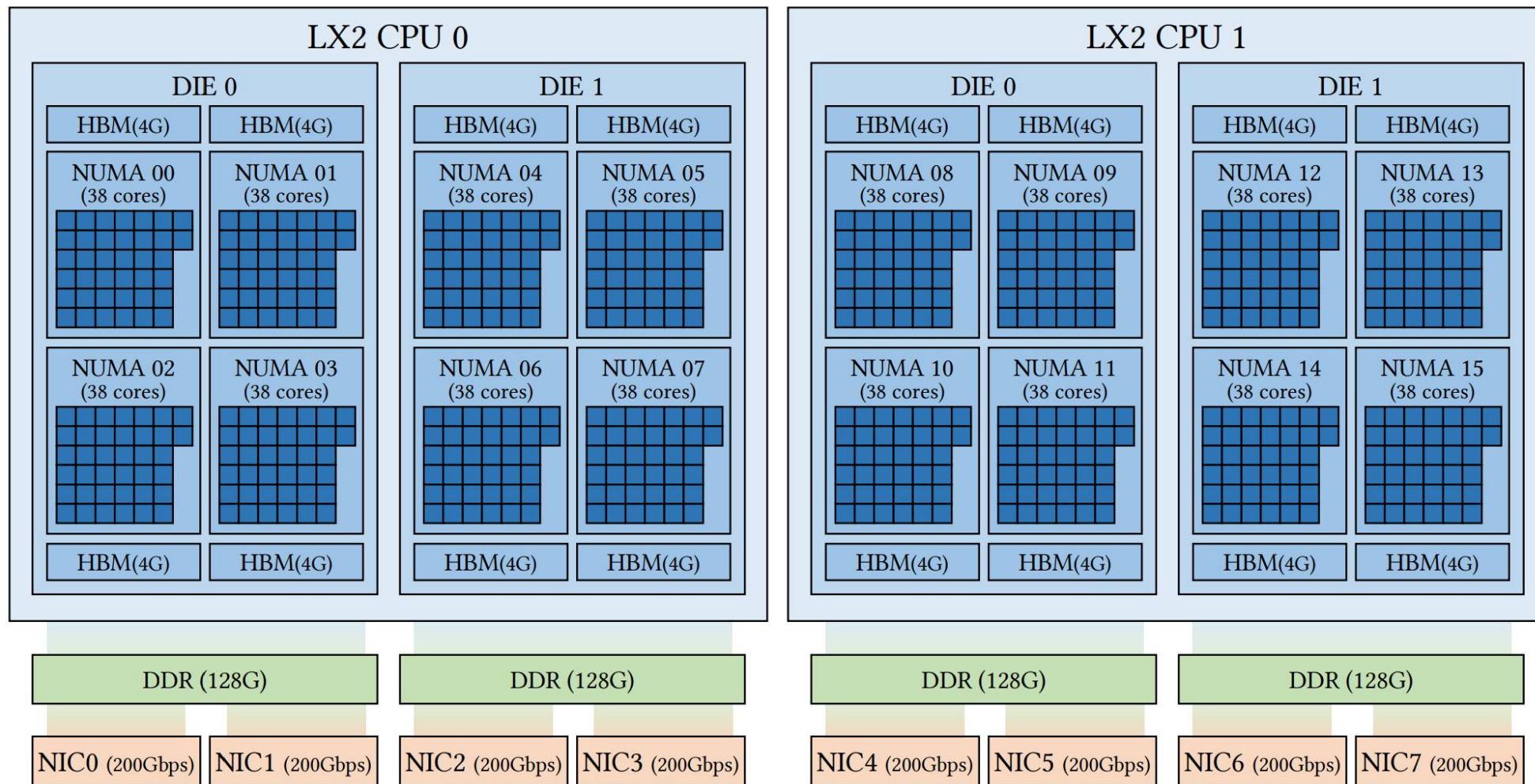
● 20000 trajectories for each case, use last 10000

● blocked jackknife errors

Performance: Kokkos CUDA backend versus native CUDA



Compute Node Architecture of the LineShine (灵晟) Supercomputer



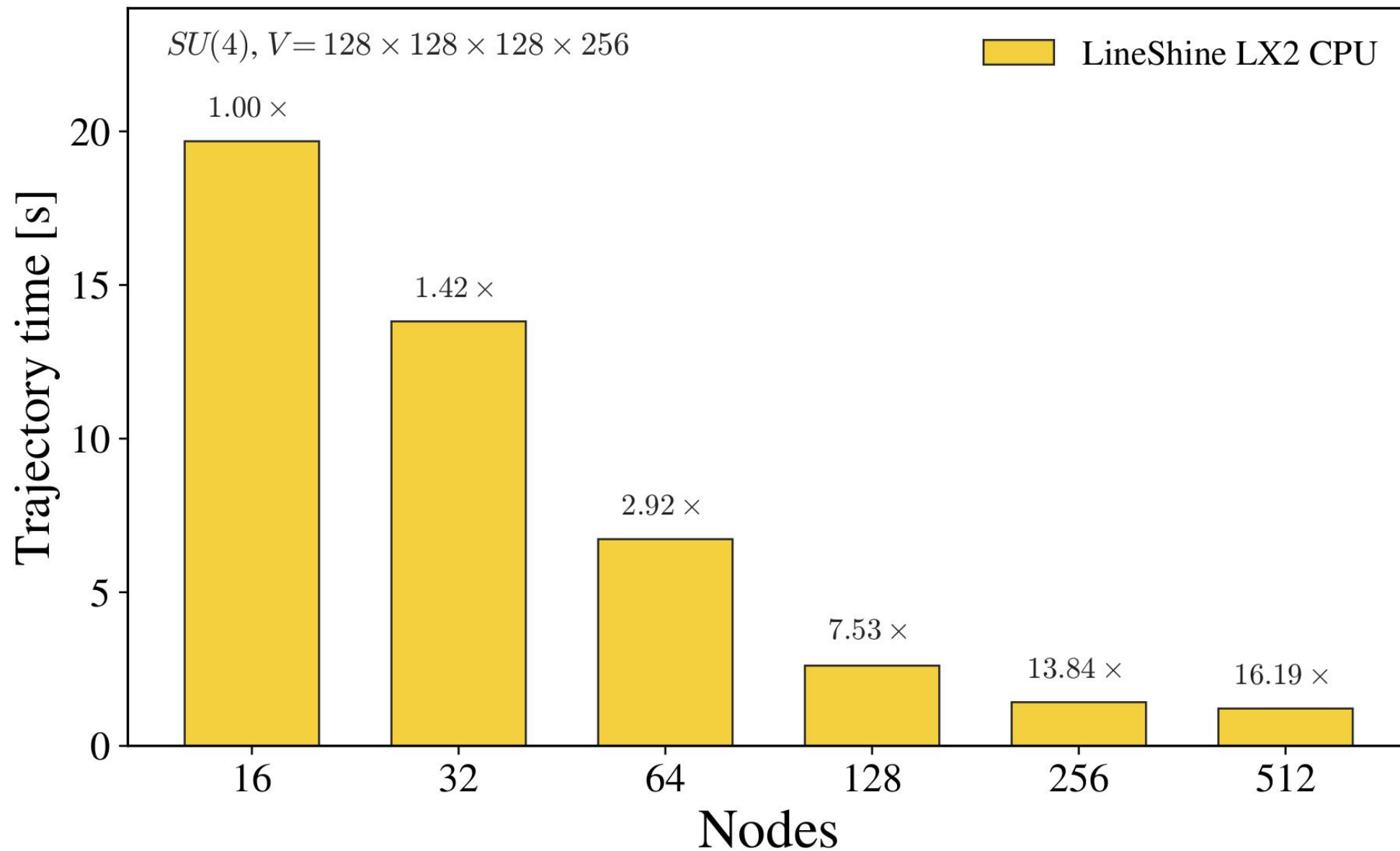
- Figure from arXiv: 2609.27482
- 304 cores (60.3 TFLOPs FP64) per LX2 CPU, 608 cores per node
- Armv9, SVE+SME, on package HBM (4TB/s)

Performance: SIMD speedup on LineShine

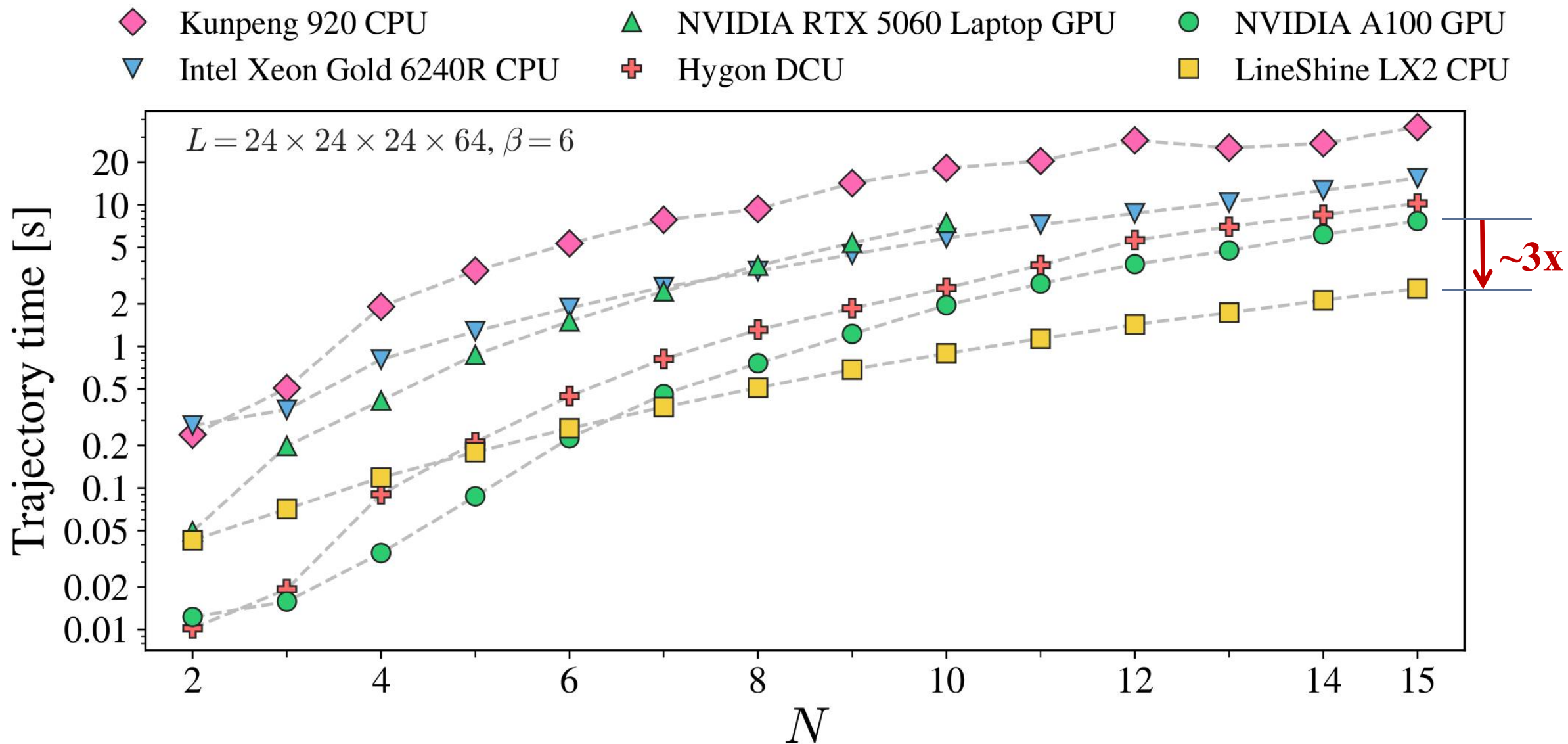
N	Scalar [s]	SIMD [s]	Speedup	N	Scalar [s]	SIMD [s]	Speedup
2	0.073	0.043	1.70	9	1.280	0.687	1.86
3	0.131	0.071	1.84	10	1.774	0.893	1.99
4	0.248	0.119	2.08	11	2.750	1.135	2.42
5	0.380	0.179	2.12	12	3.957	1.428	2.77
6	0.522	0.264	1.98	13	5.918	1.732	3.42
7	0.637	0.374	1.70	14	7.427	2.120	3.50
8	0.902	0.512	1.76	15	8.512	2.562	3.32

- ⊙ $\beta = 6.0, 24^3 \times 64$
- ⊙ single LineShine LX2 CPU
- ⊙ SIMD: SVE-512

Performance: strong scaling speedup on LineShine



Performance: trajectory time across architectures



Outlook and Future Work

◉ DDQ

➤ New-generation compiler

- DDQ script language $\rightarrow$ C programming language

- Refactor the current application level code with new language features

➤ Enhancing functionality and improving performance for large-scale computations

◉ kwqft

➤ Add fermion actions and Krylov solvers for sparse linear systems (quenched)

➤ Enable dynamical-fermion configuration generation via Hybrid Monte Carlo (HMC)