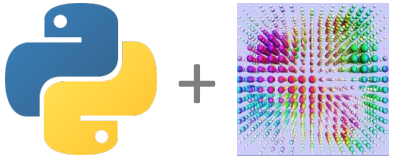
Use QUDA with Python

https://github.com/CLQCD/PyQUDA



Xiangyu Jiang · Indiana University

Lattice 2026 @ University of Maryland, College Park

BRING ON TOMORROW

July 27, 2026

Outline

Background

Motivation and related works

BRING ON TOMORROW

Background

Why a Python front-end to QUDA?

Motivation

- Easy accessibility to C/C++ libraries from Python
- Excellent GPU performance and scaling of QUDA
- Rich ecosystem of Python packages for scientific computing
  - CPU: NumPy, SciPy, …
  - GPU: CuPy, DPNP, PyTorch, …

Related works

- The community is building LQCD software in modern languages
  - Python: GPT, Lyncs-API, Qlattice
  - Julia: LatticeQCD.jl, MetaQCD.jl
  - Nim: QEX
  - …

Design

Architecture and features

BRING ON TOMORROW

Goal

Python productivity and QUDA performance

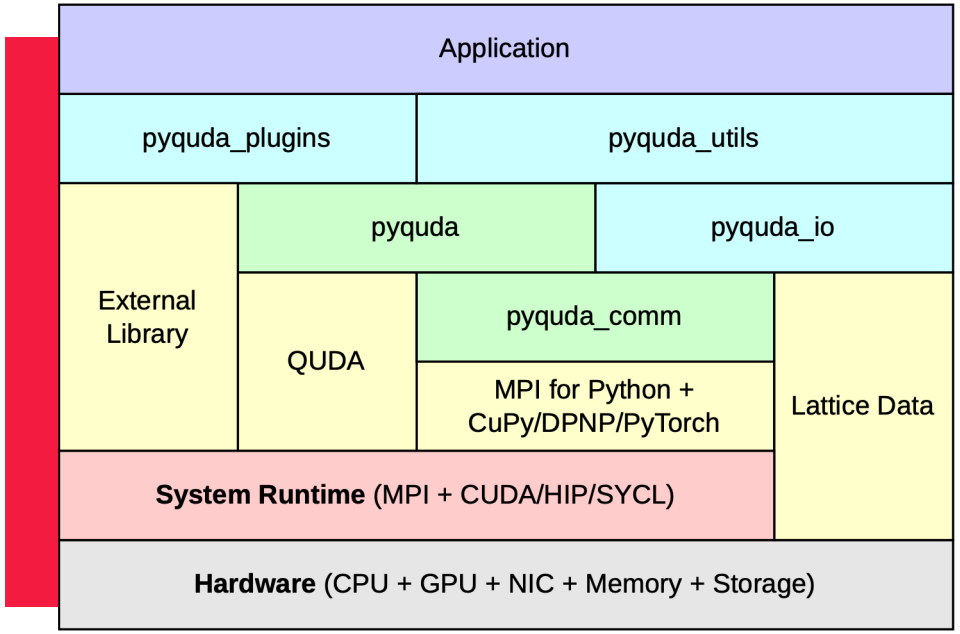
https://github.com/CLQCD/PyQUDA

Performance	Usability	Compatibility
• Performance-critical kernels from QUDA • Very thin Cython wrapper for QUDA C API • Optional GPU-accelerated array backends (CuPy, DPNP and PyTorch)	• User-friendly high-level Python interfaces • Abstraction of Dirac operators • I/O for LQCD data files in different formats • Fully type-hinted Python packages	• Supports QUDA targets: CUDA, HIP and SYCL • Basic usage with NumPy • Work with Python>=3.8

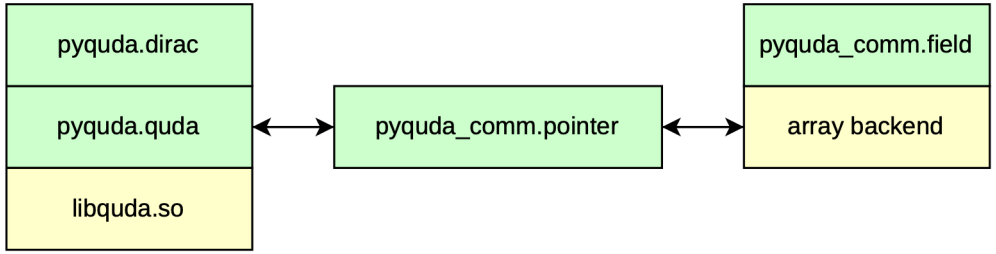
Architecture

PyQUDA project

= PyQUDA + PyQUDA-Utils



- PyQUDA provides
  - MPI and GPU initialization
  - Bindings to the QUDA C API (quda.h)
  - Bridge between Python array and QUDA
  - High-level Python interface for QUDA
- PyQUDA-Utils is built on top of PyQUDA, and provides
  - I/O functions
  - Plugin framework
  - Utilities to complete workflow



Design

PyQUDA builds core abstractions

MPI communicator

- Wraps mpi4py and provides a Cartesian-like communicator
- Reordered ranks for interoperability with other applications
- Maps each MPI rank to a GPU

Lattice field

- Stores the lattice metadata and the backend array
- Basic manipulation functions (linalg, memory management, I/O, pointers, shift)
- Exposes the pointer to the GPU memory from the backend array

Dirac operator

- Collections of QUDA conventions and parameters
- High-level Python interface for QUDA functions
- Gauge context manager

Design

PyQUDA-Utils completes the workflow

Source, phase, and MRHS

```python
from pyquda_utils import core, source, phase_v2

mom_phase_gen = phase_v2.MomentumPhase(latt_info)
mom_phase = mom_phase_gen.getPhase([0, 0, 1])

t0 = 0
b0 = [0, 0, 0, t0]

pt_source = source.propagator(latt_info, "point", b0)
sh_source = source.gaussianSmear(
    pt_source,
    gauge,
    rho=2.0,
    n_steps=20,
)

with dirac.useGauge(gauge):
    mom_propag = core.invert(
        dirac,
        "wall",
        t0,
        mom_phase,
        mrhs=12,
    )
    sh_propag = core.invertPropagator(
        dirac,
        sh_source,
        mrhs=12
    )
```

I/O for LQCD data files

Gauge	Read	Write	Checksum	Plaquette	Link Trace
Chroma QIO Lime	✓	✘	✓	N/A	N/A
ILDG Lime	✓	✘	✓	N/A	N/A
MILC	✓	✓	✓	N/A	N/A
KYU	✓	✓	N/A	N/A	N/A
χQCD	✓	✓	N/A	N/A	N/A
NERSC	✓	✓	✓	✓	✓
OpenQCD	✓	✓	N/A	✓	N/A

Propagator	Read	Write	Checksum
Chroma QIO Lime	✓	✘	✓
MILC QIO Lime	✓	✘	✓
KYU	✓	✓	N/A
χQCD	✓	✓	N/A

Other interesting features

- Gamma matrices
- Rational approximation
- 3D Laplacian eigensolver for distillation
- Heavy quark restart solver
- FFT for lattice fields
- …

Examples

And benchmarks

```
QUDA_PATH=/path/to/quda/install pip install pyquda pyquda-utils
```

Example

Propagator

1. Initialize PyQUDA and set up the lattice
2. Set up Dirac operator
3. Process and use gauge
4. Solve propagator
5. Contraction (Optional)
   - LQCDMaster: Agentic Scientific Computing for Lattice Quantum Chromodynamics Research (Gao et al., arXiv: 2607.15001)

```python
from pyquda_utils import core, io

core.init(grid_size, latt_size, resource_path="~/Lattice2026/.cache/q
latt_info = core.LatticeInfo(latt_size, t_boundary, xi_0 / nu)

dirac = core.getDirac(latt_info, mass, tol, maxiter, xi_0, coeff_r, c

gauge = io.readChromaQIOGauge(gauge_file)
gauge.stoutSmear(stout_steps, rho, dir_ignore)
with dirac.useGauge(gauge):

    propag = core.invert(dirac, "point", [0, 0, 0, 0], mrhs=12)

    # Contraction to get pion and proton correlators
```

Example

RHMC

1. Initialize PyQUDA and set up the lattice
2. Set up HMC with monomials and integrator
3. Start fresh or reload a gauge

```python
from pyquda.hmc import HMC, INT_3G1F
from pyquda.action import GaugeAction, HISQAction
from pyquda_utils import core, io
from pyquda_utils.hmc_param import (
    symanzikOneLoopGaugeLoopParam as loopParam,
    staggeredFermionRationalParam as rationalParam,
)

core.init(grid_size, resource_path="~/.cache/quda", enable_force_monit
latt_info = core.LatticeInfo(latt_size, t_boundary, anisotropy)

monomials = [
    GaugeAction(latt_info, loopParam(u_0, "hisq", 4), beta),
    HISQAction(latt_info, rationalParam((m_ud, m_s, 0.2), (2, 1, -3),
    HISQAction(latt_info, rationalParam((0.2,), (1,), 7, 9, 1e-15, 90
    HISQAction(latt_info, rationalParam((0.2,), (1,), 7, 9, 1e-15, 90
    HISQAction(latt_info, rationalParam((0.2,), (1,), 7, 9, 1e-15, 90
    HISQAction(latt_info, rationalParam((m_c,), (1,), 7, 9, 1e-15, 90
]
hmc = HMC(latt_info, monomials, INT_3G1F(n_steps))

gauge = io.readMILCGauge(f"{gauge_prefix}.{start}")
hmc.initialize(seed, gauge)
```

Example

RHMC

4. Sample momentum and pseudo fermions
5. Compute action
6. Integrate
7. Compute action again
8. Accept or restore
9. Save if necessary
10. Repeat 4 - 9

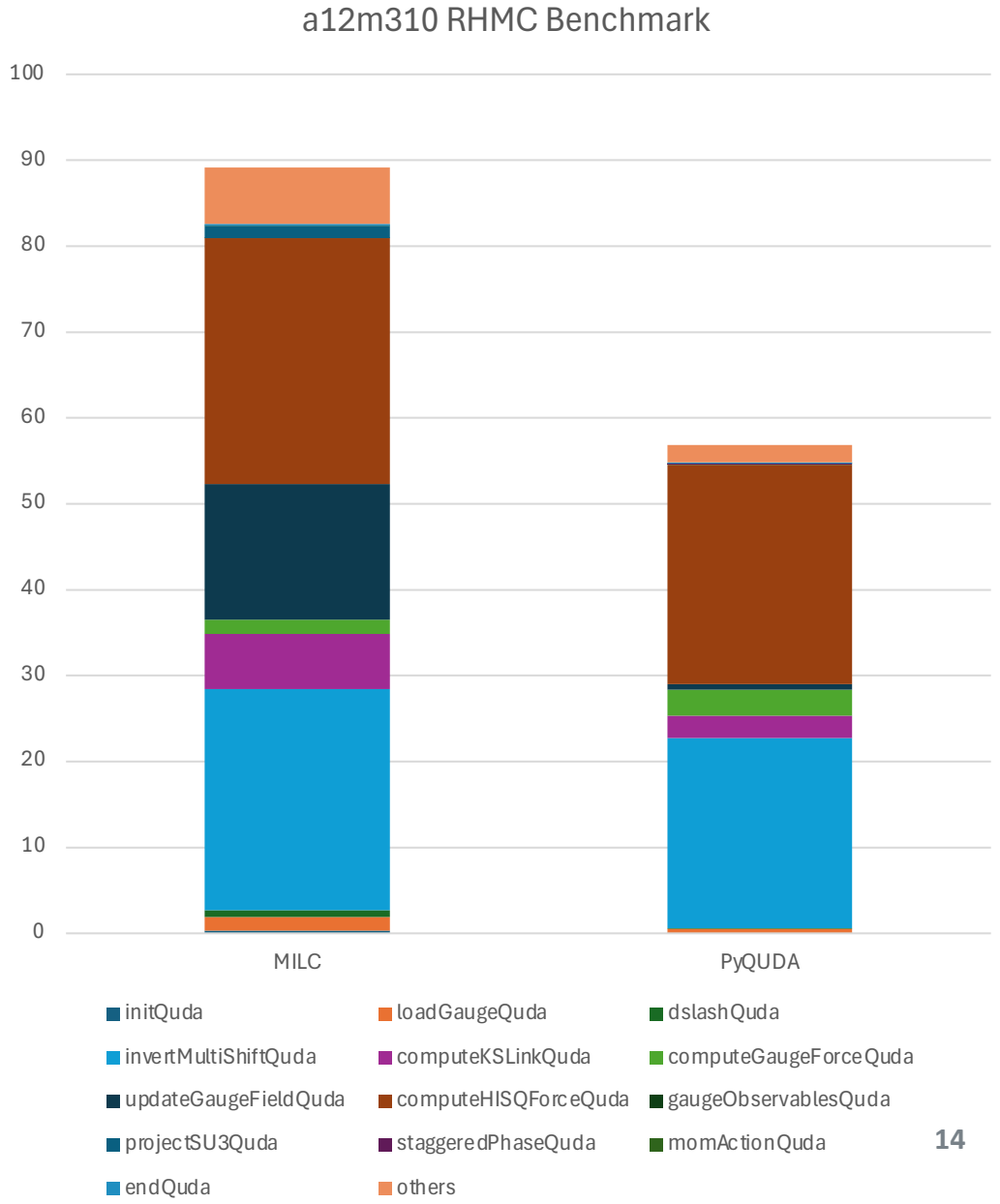
```python
for i in range(start, stop):
    hmc.gaussMom()
    hmc.samplePhi()

    kinetic_old, potential_old = hmc.momAction(), hmc.gaugeAction() +
    energy_old = kinetic_old + potential_old

    hmc.integrate(t, project_tol)

    kinetic, potential = hmc.momAction(), hmc.gaugeAction() + hmc.ferr
    energy = kinetic + potential

    accept = hmc.accept(energy - energy_old)
    if accept or i < warm:
        hmc.saveGauge(gauge)
    else:
        hmc.loadGauge(gauge)

    if (i + 1) % save == 0:
        io.writeMILCGauge(f"{gauge_prefix}.{i + 1}", gauge)
```

Benchmark

RHMC

MILC a12m310 as an example

- Benchmark is performed on 4xP100
- Compare with MILC's `su3_rhmd_hisq` double precision build, 1 trajectory
- `updateGaugeFieldQuda` dominates the final difference, which is caused by the poor host memory bandwidth on the test platform
- Difference will be negligible on modern platforms (Tested on GH200)

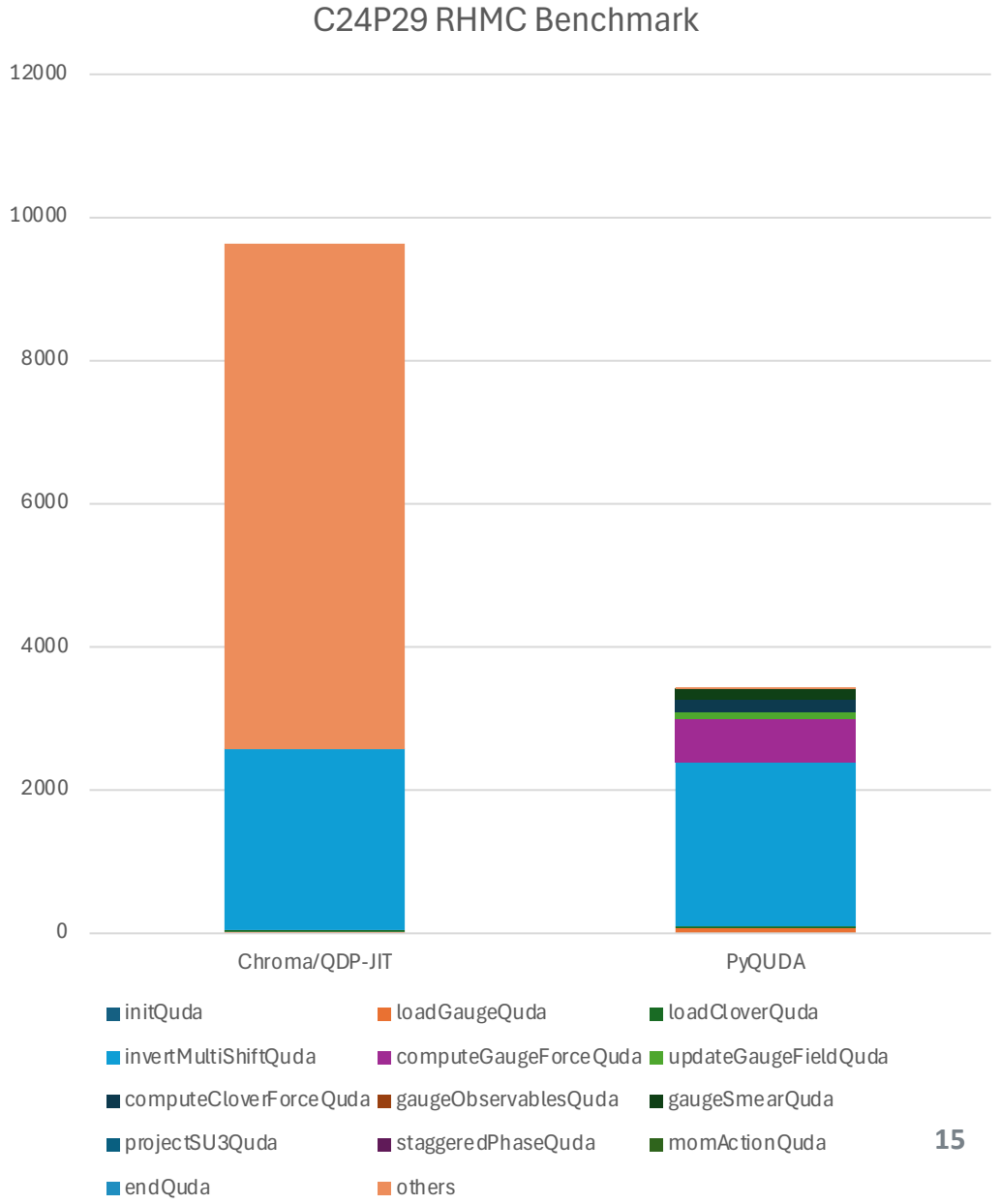


Benchmark

RHMC with stout (WIP)

CLQCD C24P29 without light quarks as an example

- Benchmark is performed on 4xP100
- Compare with Chroma/QDP-JIT's `hmc` double precision build, 100 trajectories
- Stout smearing in fermion actions require new QUDA kernel functions
- Offloading gauge and clover force computation to QUDA highly improves the overall performance
- Hasenbusch preconditioning required to implement C24P29



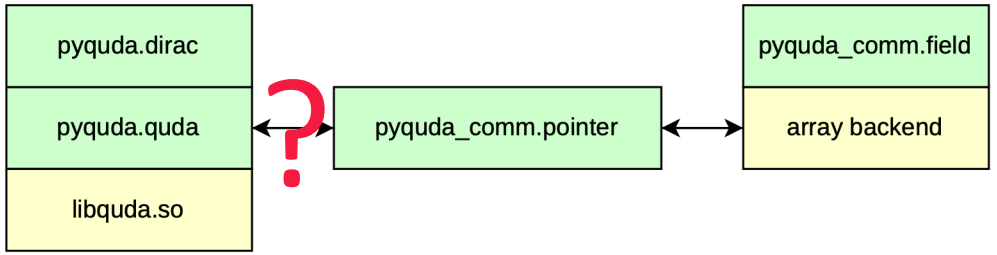
Plugins

Contraction as a plugin

Plugins

Plugin framework

https://github.com/CLQCD/contract



- Pointers do not necessarily have to be passed to QUDA C API
- C API of external libraries can be exposed through generated Cython wrappers
- Void pointers accept supported arrays, and typed pointers accept only NumPy arrays
- Struct arguments are not supported

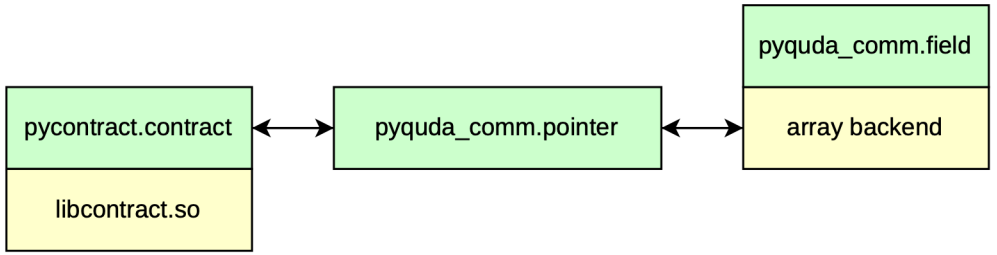
```
1  void init(int device);
2  void meson_two_point(void *correl, void *propag_i, void *propag_j, uns
3  void meson_all_source_two_point(void **correl, void *propag_i, void *p
4  void meson_all_sink_two_point(void **correl, void *propag_i, void *pro
5  void baryon_diquark(void *diquark, void *propag_i, void *propag_j, siz
6  void baryon_two_point(void *correl, void *propag_i, void *propag_j, vo
7                        unsigned long volume, int gamma_ij, int gamma_k
8  void baryon_general_two_point(void *correl, void *propag_i, void *prop
9                                BaryonContractType contract_type, size_t
10                               double _Complex *project_mn);
11 void baryon_sequential_two_point(void *propag_i, void *propag_j, void
12                                   BaryonSequentialType sequential_type,
13                                   int gamma_mn);
```

Plugins

Contraction

https://github.com/CLQCD/contract

- Achieves 30% - 80% peak GPU memory bandwidth (on CUDA) depending on the contraction type
- Implemented contraction for meson and baryon two-point correlator, baryon diquark, and baryon sequential source




```python
def mesonTwoPoint(
    propag_i: LatticePropagator,  # Forward
    propag_j: LatticePropagator,  # Backward
    gamma_ij: Gamma,  # Sink
    gamma_kl: Gamma,  # Source
):
    latt_info = propag_i.latt_info
    assert latt_info.Nd == 4 and latt_info.Ns == 4 and latt_info.Nc ==
    correl = LatticeComplex(latt_info)
    contract.meson_two_point(
        correl.data_ptr,
        propag_i.data_ptr,
        propag_j.data_ptr,
        latt_info.volume,
        gamma_ij.index,
        gamma_kl.index,
    )
    correl *= gamma_ij.factor * gamma_kl.factor
    return correl
```

Future Work

TODO list

BRING ON TOMORROW

Future Work

My TODO list

Hasenbusch preconditioning

- Required by clover
- Aux fields should be passed to QUDA directly
- Corresponding fermion action required in PyQUDA
- Should be easy since the clover force algorithm has already been implemented

Multigrid solver in RHMC

- Required by clover
- Tracking and refreshing required for multigrid solver instance in PyQUDA
- Medium, no extra kernel function required but a bit complicated

Anisotropic action in RHMC

- Required by all
- New kernel functions to handle anisotropy properly
- Medium to hard, redesign of gauge loop interface required

Thank You For Your Attention!

https://github.com/CLQCD/PyQUDA
Issues and PRs are welcome!

Xiangyu Jiang
Indiana University
xj12@iu.edu

BRING ON TOMORROW

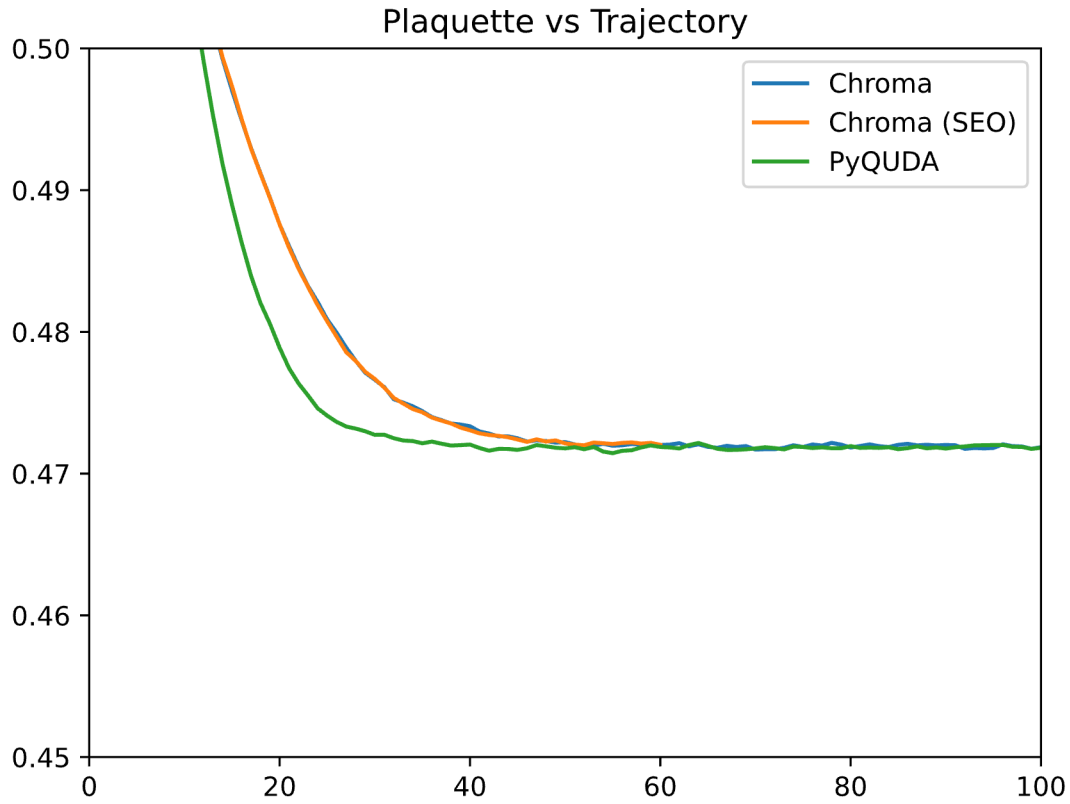
Backup Slides

Benchmark

RHMC with stout (WIP)

CLQCD C24P29 without light quarks as an example

- Thermalization sanity check
- The plaquette history from Chroma and PyQUDA converge to the same equilibrium value
- But the divergence at the beginning should be further investigated



Installation

How to install and run PyQUDA?

Prerequisites

- A QUDA installation built with (at least)
  - `QUDA_MPI=ON`, **NOT** `QUDA_QMP=ON`
  - `QUDA_INTERFACE_QDP=ON`
- A Python installation >=3.8
- (Optional) `cupy>=12` or `dpnp>=0.19` or `torch>=2`

Installation

- `export QUDA_PATH=/path/to/quda/install`
- `python3 -m pip install pyquda pyquda-utils`

Initialization in the script

- `mpirun -n 4 python test.py`

```python
from pyquda_utils import core

grid_size = [1, 1, 1, 4]
core.init(grid_size, resource_path=".cache/quda")
```

Initialization through the command line

- `mpirun -n 4 python -m pyquda -g 1 1 1 4 -p .cache/quda test_cli.py`

Contraction

Contraction by einsum

Einstein summation can be really useful when writing the tensor contraction

$$
\begin{aligned}
C_\pi(\vec{p};t,0) &= \sum_{\vec{x},\vec{y}} e^{-i\vec{p}\cdot(\vec{x}-\vec{y})} \mathrm{Tr}\left[S_l^\dagger(\vec{x},t;\vec{y},0) S_l(\vec{x},t;\vec{y},0)\right] \\
&\approx \sum_{\vec{x}} e^{-i\vec{p}\cdot\vec{x}} \mathrm{Tr}\left[S_{l,\mathrm{point}}^\dagger(\vec{x},t) S_{l,\mathrm{point}}(\vec{x},t)\right] \\
&= \sum_{\vec{x}} e^{-i\vec{p}\cdot\vec{x}} S^*_{ji,ba}(\vec{x},t) S_{ji,ba}(\vec{x},t)
\end{aligned}
$$

$$
\begin{aligned}
C_p(\vec{p};t,0) \approx \sum_{\vec{x}} & e^{-i\vec{p}\cdot\vec{x}} \epsilon^{abc}\epsilon^{a'b'c'} (C\gamma_5)_{\alpha\beta}(C\gamma_5)_{\alpha'\beta'} P^+_{\gamma'\gamma} \\
& \left[S^{aa'}_{\alpha\alpha'}(\vec{x},t) S^{bb'}_{\beta\beta'}(\vec{x},t) S^{cc'}_{\gamma\gamma'}(\vec{x},t) + S^{aa'}_{\alpha\gamma'}(\vec{x},t) S^{bb'}_{\beta\beta'}(\vec{x},t) S^{cc'}_{\gamma\alpha'}(\vec{x},t)\right]
\end{aligned}
$$

1. Explicitly write all indices when calculating correlation functions
2. Directly write the subscripts
   - Pion: “`x,xjiba,xjiba`”
   - Proton1: “`x,abc,def,ij,kl,mn,xikad,xjlbe,xnmcf`”
   - Proton2: “`x,abc,def,ij,kl,mn,ximad,xjlbe,xnkcf`”
3. Expand “`x`” with “`t`” into “`wtzyx`” (QDP order)
4. Finally we get
   - Pion: “`wtzyx,wtzyxjiba,wtzyxjiba->t`”
   - Proton1: “`wtzyx,abc,def,ij,kl,mn,wtzyxikad,wtzyxjlbe,wtzyxnmcf->t`”
   - Proton2: “`wtzyx,abc,def,ij,kl,mn,wtzyximad,wtzyxjlbe,wtzyxnkcf->t`”

Contraction

Contraction by einsum

Einstein summation can be really useful when writing the tensor contraction

$$
\begin{aligned}
C_\pi(\vec{p};t,0) &= \sum_{\vec{x},\vec{y}} e^{-i\vec{p}\cdot(\vec{x}-\vec{y})} \mathrm{Tr}\left[S_l^\dagger(\vec{x},t;\vec{y},0) S_l(\vec{x},t;\vec{y},0)\right] \\
&\approx \sum_{\vec{x}} e^{-i\vec{p}\cdot\vec{x}} \mathrm{Tr}\left[S_{l,\mathrm{point}}^\dagger(\vec{x},t) S_{l,\mathrm{point}}(\vec{x},t)\right] \\
&= \sum_{\vec{x}} e^{-i\vec{p}\cdot\vec{x}} S^*_{ji,ba}(\vec{x},t) S_{ji,ba}(\vec{x},t)
\end{aligned}
$$

$$
\begin{aligned}
C_p(\vec{p};t,0) \approx \sum_{\vec{x}} & e^{-i\vec{p}\cdot\vec{x}} \epsilon^{abc}\epsilon^{a'b'c'} (C\gamma_5)_{\alpha\beta}(C\gamma_5)_{\alpha'\beta'} P^+_{\gamma'\gamma} \\
& \left[S^{aa'}_{\alpha\alpha'}(\vec{x},t) S^{bb'}_{\beta\beta'}(\vec{x},t) S^{cc'}_{\gamma\gamma'}(\vec{x},t) + S^{aa'}_{\alpha\gamma'}(\vec{x},t) S^{bb'}_{\beta\beta'}(\vec{x},t) S^{cc'}_{\gamma\alpha'}(\vec{x},t)\right]
\end{aligned}
$$

5. Pass the subscripts string and corresponding arrays to the `einsum` function
6. Collect data from all ranks to root

```python
import cupy as cp
from pyquda_utils import core, gamma,
from opt_einsum import contract

mom_phase = phase_v2.MomentumPhase(lat

gamma_0 = gamma.gamma(0)
gamma_4 = gamma.gamma(8)
gamma_5 = gamma.gamma(15)
C = gamma.gamma(10)
P_plus = (gamma_0 + gamma_4) / 2

epsilon = cp.zeros((3, 3, 3), "<i4")
for a in range(3):
    b, c = (a + 1) % 3, (a + 2) % 3
    epsilon[a, b, c] = 1
    epsilon[a, c, b] = -1

pion = contract(
    "wtzyx,wtzyxjiba,wtzyxjiba->t",
    mom_phase.data.conj(),
    propag.data.conj(),
    propag.data,
)

```

```python
proton = (
    contract(
        "wtzyx,abc,def,ij,kl,mn,wtzyxikad,wtzyxjlbe,wtzyx
        mom_phase.data.conj(),
        epsilon,
        epsilon,
        C @ gamma_5,
        C @ gamma_5,
        P_plus,
        propag.data,
        propag.data,
        propag.data,
    )
    + contract(
        "wtzyx,abc,def,ij,kl,mn,wtzyximad,wtzyxjlbe,wtzyx
        mom_phase.data.conj(),
        epsilon,
        epsilon,
        C @ gamma_5,
        C @ gamma_5,
        P_plus,
        propag.data,
        propag.data,
        propag.data,
    )
)

```