Grid Python Toolkit (GPT)

http://github.com/lehner/gpt

Maintained by: Christoph Lehner*,†
Code co-authors: M. Bruno, D. Richtmann, S. Bürger, P. Georg, L. Jin, D. Knüttel, S. Meinel, M. Schlemmer, S. Solbrig, T. Wettig, T. Wurm

* Universität Regensburg, Fakultät für Physik; † Brookhaven National Laboratory, Physics Department

Introduction

A brief summary of GPT:

- A toolkit for lattice QCD and related theories as well as QIS (a parallel digital quantum computing simulator) and Machine Learning modules
- Python frontend, C++ backend
- Built on Grid's [1] data parallelism (MPI, OpenMP, SIMD, and SIMT)
- Initial commit Feb. 2020, 47k lines of C++/Python, >1300 commits so far, 12 contributors

Guiding principles:

- Performance portability (common framework for current and future (exascale) architectures)
- Modularity / composability (build up from modular high-performance components, several layers of composability, "composition over parametrization")

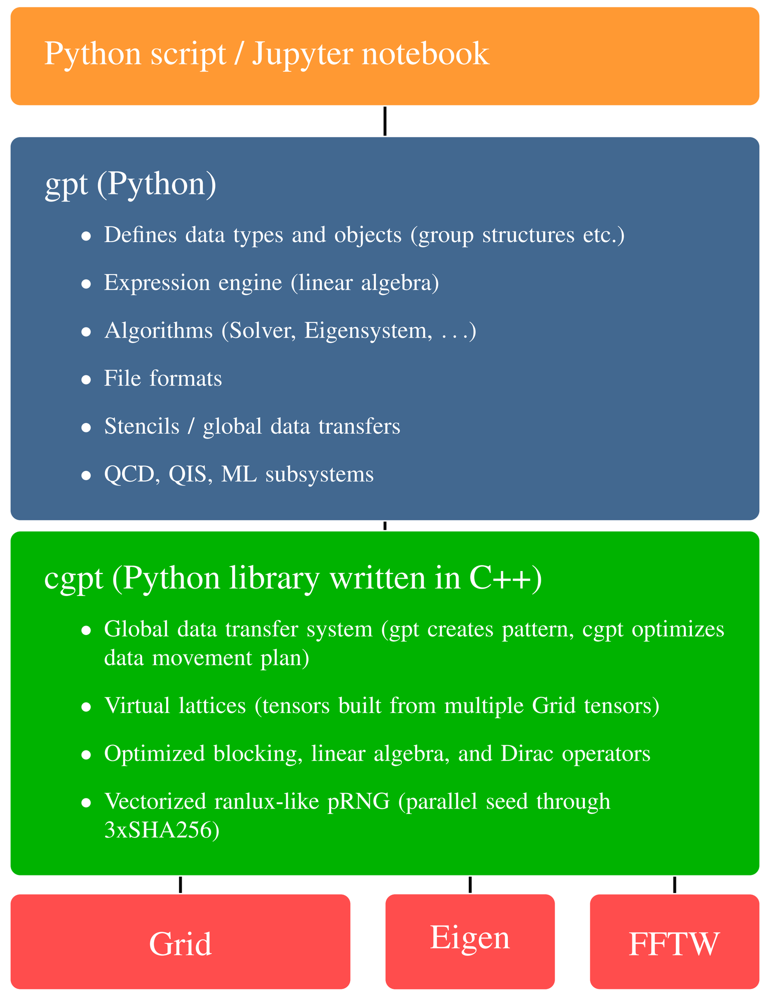

Fig. 1: GPT Library layout and its dependencies.

The QCD module

Example: Load QCD gauge configuration and test unitarity

```
In [1]: import gpt as g

        U = g.load("ckpoint_lat.IEEE64BIG.1100")

        for mu in range(4):
            g.message("SU3 - Defect: ", g.norm2(U[mu] * g.adj(U[mu]) - g.identity(U[mu])))

GPT :       1.039211 s : SU3 - Defect:  3.345168726568745e-26
GPT :       1.094156 s : SU3 - Defect:  3.3476154954606903e-26
GPT :       1.146703 s : SU3 - Defect:  3.342180010368529e-26
GPT :       1.199097 s : SU3 - Defect:  3.3423193715873574e-26
```

The expression is parsed to a tree in Python (gpt) and forwarded as abstract expression to C++ library (cgpt) for evaluation.

Example: create a pion propagator on a random gauge field

```
# double-precision 8^4 grid
grid = g.grid([8,8,8,8], g.double)

# pRNG
rng = g.random("seed text")

# random gauge field
U = g.qcd.gauge.random(grid, rng)

# Mobius domain-wall fermion
fermion = g.qcd.fermion.mobius(U, mass=0.1, M5=1.8, b=1.0, c=0.0, Ls=24,
                               boundary_phases=[1,1,1,-1])

# Short-cuts
inv = g.algorithms.inverter
pc = g.qcd.fermion.preconditioner

# even-odd-preconditioned CG solver
slv_5d = inv.preconditioned(pc.eo2_ne(), inv.cg(eps = 1e-4, maxiter = 1000))

# Abstract fermion propagator using this solver
fermion_propagator = fermion.propagator(slv_5d)

# Create point source
src = g.mspincolor(U[0].grid)
g.create.point(src, [0, 0, 0, 0])

# Solve propagator on 12 spin-color components
prop = g( fermion_propagator * src )

# Pion correlator
g.message(g.slice(g.trace(prop * g.adj(prop)), 3))
```

The following examples exhibit the modularity principle: modular code for solver configuration/near-null-space definition instead of large number of parameters.

Example: solvers are modular and can be mixed

```
# Create an coarse-grid deflated, even-odd preconditioned CG inverter
# (eig is a previously loaded multi-grid eigensystem)
sloppy_light_inverter = g.algorithms.inverter.preconditioned(
    g.qcd.fermion.preconditioner.eo1_ne(parity=g.odd),
    g.algorithms.inverter.sequence(
        g.algorithms.inverter.coarse_deflate(
            eig[1],
            eig[0],
            eig[2],
            block=200,
        ),
        g.algorithms.inverter.split(
            g.algorithms.inverter.cg({"eps": 1e-8, "maxiter": 200}),
            mpi_split=[1,1,1,1],
        ),
    ),
)
```

Example: Multi-Grid solver

```
def find_near_null_vectors(w, cgrid):
    slv = i.fgmres(eps=1e-3, maxiter=50, restartlen=25, checkres=False)(w)
    basis = g.orthonormalize(
        rng.cnormal([g.lattice(w.grid[0], w.otype[0]) for i in range(30)])
    )
    null = g.lattice(basis[0])
    null[:] = 0
    for b in basis:
        slv(b, null)
    g.qcd.fermion.coarse.split_chiral(basis)
    bm = g.block.map(cgrid, basis)
    bm.orthonormalize()
    bm.check_orthogonality()
    return basis

mg_setup_3lvl = i.multi_grid_setup(
    block_size=[[2, 2, 2, 2], [2, 1, 1, 1]], projector=find_near_null_vectors
)

wrapper_solver = i.fgmres(
    {"eps": 1e-1, "maxiter": 10, "restartlen": 5, "checkres": False}
)
smooth_solver = i.fgmres(
    {"eps": 1e-14, "maxiter": 8, "restartlen": 4, "checkres": False}
)
coarsest_solver = i.fgmres(
    {"eps": 5e-2, "maxiter": 50, "restartlen": 25, "checkres": False}
)

mg_3lvl_kcycle = i.sequence(
     i.coarse_grid(
         wrapper_solver.modified(
             prec=i.sequence(
                 i.coarse_grid(coarsest_solver, *mg_setup_3lvl[1]), smooth_solver
             )
         ),
         *mg_setup_3lvl[0],
     ),
     smooth_solver,
)
```

Features

Fermion actions:

- Domain-wall fermions: Mobius and zMobius
- Wilson-clover fermions both isotropic and anisotropic (RHQ/Fermilab actions); Open boundary conditions available

Algorithms:

- BiCGSTAB, CG, CAGCR, FGCR, FGMRES, MR solvers
- Multi-grid, split-grid, mixed-precision, and defect-correcting solver combinations
- Coarse and fine-grid deflation
- Arnoldi, implicitly restarted Lanczos, power iteration
- Chebyshev polynomials
- All-to-all vector generation
- SAP and even-odd preconditioners
- Gradient descent and non-linear CG optimizers
- Runge-Kutta integrators, Wilson flow
- Fourier acceleration
- Coulomb and Landau gauge fixing
- Domain-wall–overlap transformation and MADWF
- Symplectic integrators (leapfrog, OMF2, and OMF4)
- Markov: Metropolis, heatbath, Langevin, HMC in progress

Performance

Benchmark results are committed to https://github.com/lehner/gpt/tree/master/benchmarks/reference:

Machine	Operation	Performance	Bandwidth
Booster	$\not{D}$	12 TF/s	7.8 TB/s
Booster	ColorMatrix ×		5.2 TB/s
Booster	SpinColorMatrix ×		5.1 TB/s
Booster	SpinColorVector $\langle\cdot,\cdot\rangle$		4.8 TB/s
QPace4	$\not{D}$	0.95 TF/s	0.68 TB/s
SuperMUC-NG	$\not{D}$	0.72 TF/s	0.51 TB/s

Fig. 2: Single-node SP performance of Wilson $\not{D}$ and linear algebra on Juwels Booster (4xA100, HBM BW 1.6 TB/s per A100), Qpace4 (A64FX, HBM BW of 1 TB/s per node), and the SuperMUC-NG (Skylake 8174). The $\not{D}$ performance is inherited from Grid [1], the linear algebra performance is based on cgpt.

Production use

- Machines: Summit, Booster, QPace4, BNL KNL, Stampede2, SuperMUC-NG
- Projects: RBC/UKQCD g-2, DWF B physics projects, Wilson-Clover baryon charm physics

The domain-wall environment is fully tuned, the Wilson-Clover environment is still being optimized.

The machine learning module

Example: train simple feed-forward network

```
In [ ]: import gpt as g
        grid = g.grid([4, 4, 4], g.double)
        rng = g.random("test")

        # network and training data
        n = g.ml.network.feed_forward([g.ml.layer.nearest_neighbor(grid)] * 2)
        training_input = [rng.uniform_real(g.complex(grid)) for i in range(2)]
        training_output = [rng.uniform_real(g.complex(grid)) for i in range(2)]

        # cost functional
        c = n.cost(training_input, training_output)

        # train network
        W = n.random_weights(rng)
        gd = g.algorithms.optimize.gradient_descent
        gd(maxiter=4000, eps=1e-4, step=0.2)(c)(W, W)
```

The quantum computing module

Example: create and measure a 5-qubit bell state

```
import gpt as g
from gpt.qis.gate import *

rng = g.random("qis_test")

# initial state with 5 qubits, stored in double-precision
st = g.qis.backends.dynamic.state(rng, 5, precision=g.double)

g.message("Initial state:\n",st)

# prepare Bell-type state
st = (H(0) | CNOT(0,1) | CNOT(0,2) | CNOT(0,3) | CNOT(0,4)) * st

g.message("Bell-type state:\n",st)

# measure
st = M() * st

g.message("After single measurement:\n",st)
g.message("Classically measured bits:\n",st.classical_bit)
```

```
GPT :     197.943668 s : Initial state:
                       :   + (1+0j) |00000>
GPT :     197.949198 s : Bell-type state:
                       :   + (0.7071067811865475+0j) |00000>
                       :  + (0.7071067811865475+0j) |11111>
GPT :     197.951478 s : After single measurement:
                       :   + (1+0j) |11111>
GPT :     197.952545 s : Classically measured bits:
                       :  [1, 1, 1, 1, 1]
```

Continuous integration and Docker

CI currently has test coverage of 96%, running on each pushed commit. The docker images are automatically generated for each commit to master that passes the tests.

Quick Start

The fastest way to try GPT is to install Docker, start a Jupyter notebook server with the latest GPT version by running

```
docker run --rm -p 8888:8888 gptdev/notebook
```

and then open the shown link http://127.0.0.1:8888/?token=<token> in a browser. You should see the tutorials folder pre-installed.

Acknowledgments. I gratefully acknowledge useful discussions with participants of the weekly GPT meetings and members of the RBC/UKQCD collaborations. I am especially thankful to Peter Boyle for sharing his insights.

[1] P. Boyle, A. Yamaguchi, G. Cossu and A. Portelli, arXiv:1512.03487.