PROCEEDINGS OF SCIENCE

A Julia Code for Lattice QCD on GPUs

Guilherme Catumba,[a] **Fernando P. Panadero,**[b,*] **Carlos Pena**[b,c] **and Alberto Ramos**[a]

[a] *Instituto de Física Corpuscular (IFIC), CSIC-Universitat de València, 46071, Valencia, Spain*
[b] *Instituto de Física Teórica UAM-CSIC, c/ Nicolás Cabrera 13-15*
[c] *Department of Theoretical Physics, Universidad Autónoma de Madrid, 28049 Madrid, Spain*

E-mail: fernando.p@csic.es

We present a new GPU-based open source package to perform Lattice simulations developed in Julia. The code currently supports generation of SU(2) and SU(3) (pure gauge) configurations with different actions and boundary conditions, and is able to perform measurements of flow observables (both gluonic and fermionic) as well as different fermionic two point functions. We will show the capabilities of the package, and provide information about some measurement codes built on top of this framework.

The 41st International Symposium on Lattice Field Theory (LATTICE2024)
28 July - 3 August 2024
Liverpool, UK

*Speaker

PoS(LATTICE2024)281

1. Introduction

Lattice simulations provide a framework for first-principles computations in strongly coupled quantum field theories. Highly salable codes for HPC systems and algorithmic developments play a key role in this field, specially in the most demanding task of making precise computations in Quantum Chromodynamics (QCD).

While most lattice simulation software has been designed for CPUs, rapid development of Graphics Processing Units (GPUs) provides very efficient tools, which has led to a sustained effort both to adapt the simulation codes to support GPUs, and develop new software based directly on them. This is the main motivation for developing our code.

Julia [1] is a high-level programing languaje that provides several atractive features. The main one is just-in-time (JIT) compilation, resulting in a performance close to standard compiled languages like C while maintining the code easy to both read and develop. Multiple dispatch is also a very usefull feature for both developing and using the package, since it allows for a high degree of abstraction. Julia also provides a built-in package manager, which is fundamental for the compatibility and development of any package.

`LatticeGPU.jl` [2] is an open source package developed in Julia, designed to perform lattice simulations using GPUs. This code aims to provide a good balance between the high computational efficiency required to perform lattice simulations in reasonable time periods, and the simplicity and speed in code development required to implement and test new ideas.

The code does not support GPU parallelization yet, so no interconnect between GPUs is assumed. This still allows for some parallelization in lattice simulations, since different sections of the Monte Carlo chain, or replica runs, can be processed in parallel by different GPUs.

Some of the more relevant features of the code include the generation of quenched configurations for SU(2) and SU(3) for different actions and boundary conditions, the computation of O(a) improved Wilson fermion propagators and contractions, and the meassurement of a number of observables in both the gluonic and fermionic sectors. Some of these capabilities will be described in this work, while more details about the full content of the code will be available in an upcoming publication. Several checks have been performed both against published results [3–7], other codes [8], and consistency checks available within the package itself.

2. Code structure

In this section, we will discuss some key features available in the package. While not every detail will be covered, the goal is to provide an overview of the main tools available and the overall usability of the code.

2.1 Lattice geometry

The geometry of the lattice is encoded in the `SpaceParm` structure, needed for most functions in the package. A variable of this type can be defined via the contructor

```
lp = SpaceParm{D}(iL,sub_iL,BC,Tw_tensor),
```

where `D` is the number of dimensions, `iL` is the dimension of the lattice, `sub_iL` is the dimension of the sub-blocks, `BC` defines the boundary conditions and `Tw_tensor` is the twist-tensor [9] (which can be ommited in the constructor).
The lattice size `iL` must be a tuple of `D` integers, where the last integer will denote the time extent. The dimension of the sub-blocks `sub_iL` must be a tuple of the same type, where each component must divide the dimension of the lattice in the same direction and will define how the different fields are parallelized inside the GPU. This parallelization is illustrated in Figure 1.

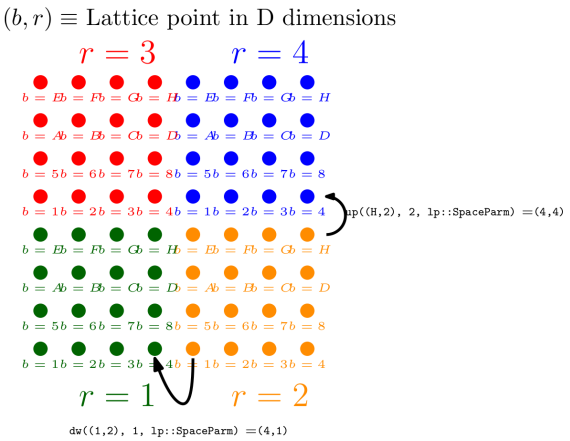

Figure 1: An example on how the lattice geometry is parallelized in the GPU. In this case, the lattice geometry is defined by `SpaceParm{2}((8,8),(4,4),BC,Tw_tensor)`. The functions `up` and `dw`, used to move with the "GPU coordinates", are also illustrated.

The available options for boundary conditions are the following:

- `BC_PERIODIC`: Periodic boundary conditions in all space-time directions.
- `BC_SF_ORBI`: Schrödinger functional (SF) boundary conditions, orbifold construction [10].
- `BC_SF_AFWB`: SF boundary conditions, Aoki-Frezzoti-Weisz choice B [11].
- `BC_OPEN`: Open boundary conditions in the time direction [12].

For the case of open boundary conditions, the effective size of the time direction will be `iL[4]-1`, while for the Periodic and both cases of Schrödinger functional boundaries, the effective time extent is `iL[4]`.

2.2 Fields and data structure

The relevant fields for a simulation will be CUDA arrays (`CuArray`) permanently stored in the GPU. The dimension and element-type of this array will depend on the field we are working with, and in most cases the elements will be custom structures of the package.

The elements of the different arrays will map into the lattice geometry according to the parallelization defined in the `SpaceParm` structure, while additional indices can label features like the direction (vector fields) or plane indexing (tensors fields). These `CuArray` can take values of several different structures, the two most relevant cases being gauge fields and fermion fields. In the case of gauge fields, one uses `vector_field` with values in a specific group. The two available choices are `SU3{T}` and `SU2{T}`, where `T` denotes the precision, usually `Float64`.

The way these stuctures store a group element is designed to minimize memory use: for `SU2` we use the Caley-Dickson representation ($g = (z_1, z_2)$, $|z_1|^2 + |z_2|^2 = 1$), while for `SU3` we store the first two rows of the matrix and recompute the third row every time it is needed, using unitarity.

For the case of (pseudo-)fermion fields, the data structure is nested in the following way: `CuArray > Spinor{NS,REP} > REP > Complex{T} > T`. The type parameter `NS` will tipically be equal to four and `REP` can take the values `SU2fund{T}` and `SU3fund{T}`.

Most functions in the package require the use of auxiliary fields, that should be allocated in the GPU beforehand. This allocation is done in the corresponding workspace structures: the one related to the gauge sector can be allocated via `ymws = YMworkspace(SU3,Float64,lp)`, while for the fermionic sector we have `dws = DiracWorkspace(SU3fund,Float64,lp)`. In both cases, replacing `SU3` for `SU2` is also supported. This workspace is designed to minimize memory allocations. While this permanent allocation together with the non-parallelization between GPUs imposes a bound on the available lattice volumes, the code can comfortably run simulations on a $L/a = 64$ volume with 40 GB of GPU memory.

It is possible to use the allocated fields in intermediate steps of a main program in order to avoid allocating too many variables, but this practice is not recommended since these fields will be overwritten by some routines of the package. The complete list of allocated fields in each workspace, as well as the methods that modify them, can be be found in the documentation.

3. Available features

3.1 Monte Carlo generation

The package supports so far the generation of quenched configurations with the Hybrid Monte Carlo algorithm. The gauge action in the bulk of the lattice can be written as

$$S_G = \beta \sum_{x,\mu>\nu} c_0 Tr(\mathbf{1} - P_{\mu\nu}) + c_1 Tr(\mathbf{1} - R_{\mu\nu}), \tag{1}$$

where $P_{\mu\nu}$ and $R_{\mu\nu}$ are respectively the oriented 1×1 and 1×2 Wislon loops. The condition $c_0 + 8c_1 = 1$ is assumed. This action parametrizes a subset of a general O(a) improved action for the gauge fields, such as Lüscher-Weisz[13] or Iwasaki[14]. The relevant structure to define the parameters of the action is the `GaugeParam` structure, and can be defined by `gp = GaugeParm{Float64}(GRP,beta,c0,(cG0,cG1),phi,lp.iL)`, where `GRP` is the gauge group, `beta` is the value of the inverse gauge coupling ($2N/g_0^2$), `c0` is defined in the action, the parameters `cG0` and `cG1` are the weights for the action in the boundaries. The boundary values of the spatial links for the SF boundary conditions are defined by `phi` $= (\phi_1, \phi_2)$ via[1] $U_i = exp(i\phi_1/lp.iL[i], i\phi_2/lp.iL[i], -i(\phi_1+\phi_2)/lp.iL[i])$.

The Hybrid Monte Carlo method is implemented in the function

```
dh,acc = HMC!(U,intsch,lp,gp,ymws).
```

The first variable is the gauge field configuration, that will be modified. The second variable defines the integration scheme for the Hamilton equations. Avaiable options are: `leapfrog(Float64, epsilon,number_of_steps)`, and the Omelyan integrators `omf2(...)` and `omf4(...)`. The last input is the Yang-Mills workspace, explained in the previous section.

The `HMC!` function will return two variables: the energy violation of the process and a boolean true if the configuration was accepted (and false if it was rejected).

[1] Note that this parametrization is only available for `SU3`.

3.2 Gauge observables

The main functions for the measurements of observables involving the Yang-Mills sector are the following

- `plaquette(U, lp::SpaceParm, gp::GaugeParm, ymws::YMworkspace)`: Returns the average value of the plaquette.
- `function gauge_action(U, lp::SpaceParm, gp::GaugeParm, ymws::YMworkspace)`: Returns the value of the gauge action.
- `function sfcoupling(U, lp::SpaceParm, gp::GaugeParm, ymws::YMworkspace)`: Measures the Schrödinger Functional coupling $dS/d\eta$ and $d^2S/d\eta dv$ [10, 15].
- `function Eoft_plaq([Eslc,] U, gp::GaugeParm, lp::SpaceParm, ymws::YMworkspace)`: Returns the value of the action density using the plaquette discretization.
- `function Eoft_clover([Eslc,] U, gp::GaugeParm, lp::SpaceParm, ymws::YMworkspace)`: Returns the value of the action density using the clover discretization.
- `function Qtop([Qslc,] U, gp::GaugeParm, lp::SpaceParm, ymws::YMworkspace)`: Returns the value of the topological charge with the clover definition.

The first argument in the last functions can be ommited; if included, it must be a vector of size `lp.iL[4]`, and the contribution of each time-slice will be written in that vector.

The gradient flow [3] has proven to be a very powerful tool for many years, specially in the Yang-Mills sector. An implementation of this construction os also available in the package.

The main structure needed for the integration of the gradient flow equations is `FlowIntr`, where the details of the integration scheme are specified. These can be defined by functions such as `wfl_euler(Float64,eps,tol)`, where `eps` is the stepsize and `tol` is the tolerance for the adaptive stepsize. This function defines the parameters for the Euler integration scheme with Wilson flow. The rest of the integration schemes can be defined by substituting "`euler`" for "`rk2`" or "`rk3`" for Runge-Kutta of order 2 and 3 respectively, and "`wfl`" for "`zfl`" for Zeuthen flow [16].

The integration of the flow equations is implemented in the function `flw(U, int::FlowIntr, ns::Int64, eps, gp, lp, ymws)`, where `ns` is the number of steps in the integration, meaning that the total integration length will be `t=ns×eps`. One can ommit `eps`, and the value of `int.eps` will be used.

An implementation of the adaptive step size integration is also available via the `flw_adapt(U, int::FlowIntr, tend, epsini, gp, lp, ymws)`, method. The argument `tend` is the total distance of integration and `epsini` is the value of the initial step for the integration and can be ommited, in which case `int.eps` will be used. In these methods, the integration step is recomputed every ten steps to keep the error of the integration below `int.tol`. In this case, the function will return two values: the number of steps in the integration and a vector with all the step sizes.

3.3 Fermionic observables

In this section we discuss the implementation of Wilson fermions in the package. The necessary parameters for fermionic meassurements are stored in the `DiracParam` structure

```
dpar = DiracParam{Float64}(REP,m0,csw,theta,mu,cF),
```

where `REP` can take the values `SU2fund` and `SU3fund`, `m0` is the bare quark mass, `csw` is the Sheikholeslami-Wohlert coefficient [17], `theta` is a 4-vector with the phase-shift for the fermions in the boundaries[2], `mu` is the twisted mass and `cF` is the improvement coefficient for the fermions with Dirichlet boundary conditions (also denoted as $\tilde{c_t}$) [10].

The Dirac operator is implemented in the functions `Dw!`, `g5Dw!` and `DwdagDw!` with the syntax `Dw!(pso,U,psi,dpar::DiracParam,dws::DiracWorkspace,lp::SpaceParm)`, where `pso` and `psi` are the output and input quark fields respectively. Note that both fields have to be allocated before the call and the output field will be modified.

Another important function is the `Csw!(dws, U, gp, lp)` function. This updates the value of the field `dws.csw`, where the Sheikholeslami-Wohlert term is stored with the clover discretization. Once this is done, this term is added every time the Dirac operator is applied.

The Conjugate-Gradient (CG) solver is implemented and allows to compute correlation functions of fermionic fields. This can be done either by inverting the Dirac operator on a specific source via the `CG!` method or by using the `propagator!` function. This function has two methods: one utilizes normally distributed random noise sources, while the other inverts a source with a specific color, spin, and position. Similar functions to compute the boundary-to-bulk propagators for the Dirichlet boundary conditions are also available, allowing to compute different correlators such as f_P and f_A for SF boundary conditions.

An implementation of the Gradient flow for fermion fields following the lines of Ref.[7] is also available in the package. The functions described for the pure gauge cause are extended to include the integration of fermion fields, thanks to the multiple dispatch feature in Julia.

Unlike in the gauge scenario, estimating certain correlation functions leads to the integration of the adjoint flow equations. The numerical instabilities apearing for the gauge fields are overcome in a similar way as in Ref [7]. An integration with adaptive step size is performed first to fix a grid in the flow time. Then, `maxnsave` intermediate gauge fields in this grid are stored, and the fermion field is backflowed through the grid using the closest stored gauge field to reconstruct the necessary gauge field at each step. The function

```
backflow(psi, U, Dt, maxnsave, gp, dpar, lp, int, ymws, dws),
```

implements this procedure, where `Dt` is the value of the final flow time, and `maxnsave` is the number of intermediate field configurations stored for the integration[3]. The field `psi` is the fermion field at flow time `Dt`, while the gauge field `U` must be the gauge field at zero flow time. One important restriction is that `int` must be any order three Runge-Kutta, and if ommited the value `wfl_rk3(Float64,0.01,1.0)` will be used. This function will modify the value of `psi` and return nothing.

3.4 Performance

The package provides performance details of most methods wothout specific benchmarking in the main program using the `TimerOutputs.jl` package. We have also benchmarked the

[2]Each value of `theta` is not the phase in each direction, but the full phase factor, so $||\texttt{theta}_i|| = 1$ should be numerically imposed.

[3]Intermediate gauge configurations are stored in the CPU memory.

performance between two devices for different processes in a $L/a = 32$ lattice: 50 integration steps of fourth order Omelyan, Gradient flow integration to $t/a^2 = 10.0$ with stepsize $\epsilon = 0.01$, backflow of a fermion field from $t/a^2 = 10.0$ and 1000 iterations of the CG. In all these cases, we obtain the expected scaling compatible with the devices specifications.

	HMC	YMGF	Bfl	Dw
A100	6.6s	80.0s	92.7s	13.9s
H100	2.8s	37.2s	46.9s	6.46s
A/H	2.34	2.15	1.97	2.15

We also compared the performance of the Nvidia A100 against CPUs of the Intel Xeon IvyBridge generation, obtaining an estimated performance of 1 GPU ~ 80-100 CPUs.

3.5 Input/Output

Reading and writting both gauge configurations and fermion propagators is supported with native formats via the functions `save_cnfg`, `read_cnfg`, `save_prop` and `read_prod`. Other standard formats are also supported for reading gauge configurations such as the CERN format used in [8].

4. Some examples

In this section, we will provide some examples to illustrate the usability of the package in some simple scenarios. We start by defining the main data structures with the parameters of the simulation and allocating the necessary fields and workspaces.

```
using LatticeGPU
lp = SpaceParm{4}((16,16,16,16),(4,4,4,4),BC_PERIODIC,(0,0,0,0,0,0))
intsch = omf4(Float64,0.01,50)
# Gauge sector
U = vector_field(SU3{Float64},lp);
ymws = YMworkspace(SU3,Float64,lp);
gp = GaugeParm{Float64}(SU3{Float64},6.0,5/3,
                                (1.0,1.0),(0.0,0.0),lp.iL)
# Quark sector
psi = scalar_field(Spinor{4,SU3fund{Float64}},lp);
dws = DiracWorkspace(SU3fund,Float64,lp);
dpar = DiracParam{Float64}(SU3fund,0.2,1.0,
                           (1.0,1.0,1.0,1.0),0.0,1.0)
```

Now we can start the simulation with a cold configuration and start generating our Monte Carlo chain, in this case of one hundred configurations. We also display the value of the plaquette and compute the quark propagator, with wich one could then compute any contraction.

```
fill!(U,one(SU3{Float64}));
for i in 1:100
    dh,acc = HMC!(U,intsch,lp,gp,ymws)
    println("## Monte Carlo step ",i)
```

```
    println("## Plaquette : ",plaquette(U, lp, gp, ymws))
    Csw!(dws,U,gp,lp)
    niter = propagator!(psi, U, dpar, dws, lp, 10000, 1.0e-13, 1)
    println("## Inversion converged after ",niter," iterations.")
end
```

5. Conclusions

In this work we presented `LatticeGPU.jl`, an open-source package for Lattice simulations on GPUs developed in Julia. The power and flexibility of Julia combined with the high efficiency of GPUs provide a great enviroment for lattice simulations. It allows the simulation package to have competitive performance while also being easily extensible, both in expanding the current capabilities and in developing new functionalities.

The code version is robust, as several results from the literature and other codes have been reproduced. Future developments in the package can be expected in different directions, from the implementation of dynamical fermions and more sophisticated solvers to the implementation of less conventional tools such as the already implemented fermion flow.

The main advantage this code can present is a good ratio of efficiency and simplicity. The use of GPUs allows for a very high performance, and the use of Julia together with the fact that a front-end use of the package does not require specific GPU knowledge, allows for this good balance.

Acknowledgments

This work is partially supported by the Spanish Research Agency (Agencia Estatal de Investigación) through the grants IFT Centro de Excelencia Severo Ochoa CEX2020-001007-S and PID2021-127526NB-I00, funded by MCIN/AEI/10.13039/501100011033.

AR acknowledge support from the Generalitat Valenciana grant CIDEGENT/2019/040, the European projects H2020-MSCA-ITN-2019//860881-HIDDeN and 101086085-ASYMMETRY, and the national projects CNS2022-136005 and PID2020-113644GB-I00.

We also acknowledge partial support from the project H2020-MSCAITN2018-813942 (EuroPLEx) and the EU Horizon 2020 research and innovation programme, STRONG2020 project, under grant agreement No. 824093. Numerical calculations have been performed on the Hydra and Ciclope installations at IFT, and local SOM clusters, funded by the MCIU with funding from the European Union NextGenerationEU (PRTR-C17.I01) and Generalitat Valenciana, ASFAE/2022/020, Artemisa, funded by the European Union ERDF and Comunitat Valenciana. We also thank A. Rago for access to the DeiC installation at SDU for test purposes. F.P.P. and C.P. thank P. Fritzsch and S. Sint for their kind hospitality and fruitful discussions.

References

[1] J. Bezanson, A. Edelman, S. Karpinski and V.B. Shah, *Julia: A fresh approach to numerical computing*, *SIAM Review* **59** (2017) 65.

[2] G. Catumba, F. Panadero, C. Pena and A. Ramos. *Lattice GPU*, URL. Documentation available at Doc.

[3] M. Lüscher, *Properties and uses of the Wilson flow in lattice QCD*, *JHEP* **08** (2010) 071 [1006.4518].

[4] O. Haan, E. Schnepf, E. Laermann, K.H. Mutter, K. Schilling and R. Sommer, *Hadron mass computations on a* $16^3 \cdot 28$ *lattice -a comparison between blocked and unblocked computations*, *Phys. Lett. B* **190** (1987) 147.

[5] XLF collaboration, *Scaling test for Wilson twisted mass QCD*, *Phys. Lett. B* **586** (2004) 432 [hep-lat/0312013].

[6] ALPHA collaboration, *The Continuum limit of the quark mass step scaling function in quenched lattice QCD*, *JHEP* **05** (2004) 001 [hep-lat/0402022].

[7] M. Lüscher, *Chiral symmetry and the Yang–Mills gradient flow*, *JHEP* **04** (2013) 123 [1302.5246].

[8] M. Lüscher. *OpenQCD*, URL.

[9] G. 't Hooft, *A Property of Electric and Magnetic Flux in Nonabelian Gauge Theories*, *Nucl. Phys. B* **153** (1979) 141.

[10] M. Lüscher, R. Narayanan, P. Weisz and U. Wolff, *The Schrödinger functional: A Renormalizable probe for nonAbelian gauge theories*, *Nucl. Phys. B* **384** (1992) 168 [hep-lat/9207009].

[11] S. Aoki, R. Frezzotti and P. Weisz, *Computation of the improvement coefficient c(SW) to one loop with improved gluon actions*, *Nucl. Phys. B* **540** (1999) 501 [hep-lat/9808007].

[12] M. Lüscher and S. Schaefer, *Lattice QCD without topology barriers*, *JHEP* **07** (2011) 036 [1105.4749].

[13] M. Lüscher and P. Weisz, *On-shell improved lattice gauge theories*, *Commun. Math. Phys.* **98** (1985) 433.

[14] Y. Iwasaki, *Renormalization Group Analysis of Lattice Theories and Improved Lattice Action. II. Four-dimensional non-Abelian SU(N) gauge model*, 1111.7054.

[15] S. Sint, *On the Schrodinger functional in QCD*, *Nucl. Phys. B* **421** (1994) 135 [hep-lat/9312079].

[16] A. Ramos and S. Sint, *Symanzik improvement of the gradient flow in lattice gauge theories*, *Eur. Phys. J. C* **76** (2016) 15 [1508.05552].

[17] B. Sheikholeslami and R. Wohlert, *Improved Continuum Limit Lattice Action for QCD with Wilson Fermions*, *Nucl. Phys. B* **259** (1985) 572.