Computer Physics Communications 300 (2024) 109164

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

Computer Programs in Physics

SIMULATeQCD: A simple multi-GPU lattice code for QCD calculations ☆

Lukas Mazur [a,*], Dennis Bollweg [b,*], David A. Clarke [c,*], Luis Altenkort [d], Olaf Kaczmarek [d,*], Rasmus Larsen [e], Hai-Tao Shu [f], Jishnu Goswami [g], Philipp Scior [b], Hauke Sandmeyer [d], Marius Neumann [d], Henrik Dick [d], Sajid Ali [d,h], Jangho Kim [i], Christian Schmidt [d], Peter Petreczky [b], Swagato Mukherjee [b,*]

(HotQCD collaboration)

[a] Paderborn Center for Parallel Computing, Paderborn University, Paderborn, Germany
[b] Physics Department, Brookhaven National Laboratory, Upton, NY, United States
[c] Department of Physics and Astronomy, University of Utah, Salt Lake City, UT, United States
[d] Fakultät für Physik, Universität Bielefeld, Bielefeld, Germany
[e] Department of Mathematics and Physics, University of Stavanger, Stavanger, Norway
[f] Institut für Theoretische Physik, Universität Regensburg, Regensburg, Germany
[g] RIKEN Center for Computational Science, Kobe 650-0047, Japan
[h] Government College University Lahore, Department of Physics, Lahore 54000, Pakistan
[i] Institute for Advanced Simulation (IAS-4), Forschungszentrum Jülich, Wilhelm-Johnen-Straße, 52428 Jülich, Germany

ARTICLE INFO

Keywords:
Lattice QCD
CUDA
HIP
GPU

ABSTRACT

The rise of exascale supercomputers has fueled competition among GPU vendors, driving lattice QCD developers to write code that supports multiple APIs. Moreover, new developments in algorithms and physics research require frequent updates to existing software. These challenges have to be balanced against constantly changing personnel. At the same time, there is a wide range of applications for HISQ fermions in QCD studies. This situation encourages the development of software featuring a HISQ action that is flexible, high-performing, open source, easy to use, and easy to adapt. In this technical paper, we explain the design strategy, provide implementation details, list available algorithms and modules, and show key performance indicators for `SIMULATeQCD`, a simple multi-GPU lattice code for large-scale QCD calculations, mainly developed and used by the HotQCD collaboration. The code is publicly available on GitHub.

Program summary
Program Title: SIMULATeQCD
CPC Library link to program files: https://doi.org/10.17632/ytgycy9nc3.1
Developer's repository link: https://github.com/LatticeQCD/SIMULATeQCD
Licensing provisions: MIT
Programming language: C++, CUDA, HIP
Nature of problem: Quantum chromodynamics (QCD) is the fundamental theory behind the strong force, one of the four fundamental forces in nature. It describes interactions, mediated by gluons, between quarks, the elementary particles that build up protons, neutrons and other hadrons. One of its unique features is asymptotic freedom: with increasing energy, the interaction strength or coupling between quarks and gluons weakens, which enables studying QCD with peturbation theory. At lower energy scales, however, the strong coupling between quarks and gluons causes perturbation theory to break down, and other tools, such as lattice QCD, need to be employed to obtain QCD predictions.
Solution method: Lattice QCD is a technique to solve QCD non-perturbatively by discretizing space and time onto a four-dimensional grid. The QCD path integral is thereby rendered finite dimensional and can be evaluated numerically via Monte Carlo methods. In this paper, we present SIMULATeQCD, a C++ code to perform lattice

☆ The review of this paper was arranged by Prof. Z. Was.

* Corresponding authors.
E-mail addresses: lukas.mazur@uni-paderborn.de (L. Mazur), dbollweg@bnl.gov (D. Bollweg), clarke.davida@gmail.com (D.A. Clarke), okacz@physik.uni-bielefeld.de (O. Kaczmarek), swagato@bnl.gov (S. Mukherjee).

https://doi.org/10.1016/j.cpc.2024.109164
Received 11 July 2023; Received in revised form 19 February 2024; Accepted 4 March 2024
Available online 8 March 2024

QCD calculations on multiple GPUs, supporting multiple APIs. SIMULATeQCD implements various Monte Carlo sampling methods ranging from sampling of purely gluonic configurations to fully dynamical quark simulations with the Highly Improved Staggered Quark (HISQ) discretization. Modules for measuring various physical observables on these samples are also available.

1. Introduction

Quantum chromodynamics (QCD) is the theory describing the strong force, which binds together quarks to form baryons and mesons. In the lattice QCD (LQCD) formulation, Euclidean space-time is discretized, providing a framework that allows the use of computational techniques to extract information about physical observables in QCD beyond the weak coupling expansion, see e.g. Ref. [1] for a review.

The lattice formulation of QCD requires the discretization of the QCD action, which leads to discretization effects, e.g. the appearance of additional, unphysical fermion states. It is possible to get rid of these unphysical states by introducing an additional term in the lattice version of the Dirac operator as proposed by Wilson [2]. But this discretization scheme, commonly known as the Wilson fermion action, breaks the chiral symmetry of continuum QCD. Another discretization scheme, known as the staggered fermion formulation [3], has four tastes of degenerate mass in the continuum limit. In absence of gauge fields, the four tastes are also degenerate at non-zero lattice spacing. The gauge interaction renders the four tastes no longer degenerate at finite lattice spacing. This is referred to as taste-breaking and is a source of large discretization errors. In practical LQCD calculations improved staggered actions are used, which are designed to reduce these effects. Among the improved staggered actions the highly improved staggered quark (HISQ) action [4] exhibits the smallest taste-breaking effects [5]. The staggered fermion formulation preserves a U(1) subgroup of the full chiral $\mathrm{SU}(4)_A$ at nonzero lattice spacing. Because of this, the quark mass in this formulation does not have an additive renormalization, unlike in the Wilson formulation. This makes the tuning of the bare parameters in this action easier, and because the smallest eigenvalue of the lattice Dirac operator is bounded from below, this formulation is computationally cheaper, as the effort required to produce configurations increases like m^{-z}, where z is at least two [6]. One recovers a single fermion species per physical flavor by taking the fourth root of the staggered fermion determinant, see discussions in Refs. [7,8]. The domain wall fermion discretization avoids unphysical fermion states and preserves a lattice version of the chiral symmetry, but it is computationally very demanding and is hence not always practical for many lattice calculations.

Since the staggered fermion formulation is computationally the least expensive approach, has small discretization effects, and preserves a subgroup of the chiral symmetry, it is the primary choice for the study of QCD phase diagram, thermodynamics, and transport properties of quark-gluon plasma in LQCD, see Refs. [9–11] for recent reviews. In particular, the use of the HISQ action is advantageous in these calculations [10].

Highly improved staggered quarks are also appropriate for situations where one wants to include a dynamical charm quark. This is relevant for example in determinations of CKM matrix elements, which are currently being improved with the use of $N_f = 2 + 1 + 1$ HISQ configurations [12], and in recent calculations of the hadronic vacuum polarization contribution to the muon's anomalous magnetic moment [13]. Other applications of the HISQ action in LQCD calculations include parton distribution functions, see e.g. Ref. [14], and determinations of the coupling constant α_s and quark masses, see e.g. Ref. [15]. Clearly, there is a wide range of application for LQCD code using the HISQ action to efficiently examine a variety of physically interesting phenomena.

Modern lattice codes tend to be written for GPUs in order to achieve the performance necessary to make lattice computations feasible. In this context it is important that such code is designed to be compatible with multiple APIs, since modern supercomputers utilize GPUs from multiple manufacturers. In particular, popular Top500 supercomputers, like Frontier, LUMI, Leonardo, or Perlmutter, utilize AMD and NVIDIA GPUs, respectively. Current lattice calculations also often demand lattice sizes so large that they can no longer be accommodated in memory by a single GPU, making multi-GPU and multi-node support a fundamental requirement.

One of the most widely used, publicly available production codes that works on multiple GPUs and features the HISQ action is MILC [16]. MILC was originally written to parallelize using the MIMD paradigm. It is quite reliable and still used, e.g., in the CKM matrix element and anomalous magnetic moment studies mentioned earlier; indeed some of our own design and parameter decisions were modeled after MILC. On the other hand, owing in part to its long history, it is challenging to adapt and optimize the native code for new supercomputers, and in fact, MILC achieves its GPU parallelization by offloading through QUDA [17,18], a lattice QCD library featuring a wide variety of optimized actions and solvers[1]. While the HISQ action already exists for GPUs in MILC through QUDA, using lattice-specific external libraries can have its drawbacks. For example developers need to maintain compatibility with the library, and there can be a steep learning curve incorporating the library and understanding its functionalities[2]. Moreover specialized applications for calculations needed for QCD thermodynamics are not present or not optimized in MILC. Other popular code bases such as Grid [22,23] and openQCD [24] do not have the HISQ action implemented and hence cannot be used for our purposes.

This motivated us to develop a self-contained, open-source code for the HISQ action that is designed with the use of multiple GPUs in mind and particularly suited, but not limited, to study QCD thermodynamics. Historically, the HotQCD collaboration has primarily used code written by the lattice group of Bielefeld University, which has evolved over time from utilizing multiple CPUs to single GPUs. Meeting some of the computational challenges described above provided a strong motivation to incorporate multi-GPU support. Moreover, writing software that meets such computational challenges effectively can be especially difficult for new developers; hence this need to extend hardware capabilities was taken as an opportunity to develop a modern, future-proof, multi-GPU framework for LQCD calculations with completely revised implementations of the basic routines [25] and structures to abstract away low-level technical details. This paper therefore provides implementation details of a new Simple, Multi-GPU Lattice code for QCD calculations, which we stylize as `SIMULATeQCD`. `SIMULATeQCD` is available on GitHub [26] and licensed under the MIT license.

`SIMULATeQCD` is a publicly available lattice code supporting the HISQ action that was specifically developed for use on multiple GPUs and for both CUDA and HIP back ends. It supports $N_f = 2 + 1$ as well N_f degenerate flavors for both zero and pure imaginary baryon chemical potential. While this code was originally created with HISQ fermions in mind, we would like to stress that this is not only a HISQ code; indeed it is already able to generate pure SU(3) configurations and measure a variety of physics observables. In addition, we have made a special effort to employ a design philosophy that encourages writing highly modularized code that takes advantage of modern `C++` features. This

[1] Still more lattice codes such as Chroma [19,20] and CPS [21] offload using QUDA and therefore should have access to the HISQ action. We single out MILC because it is the code base with which we are most familiar.

[2] This latter challenge was a reason we started a new code base in 2017 rather than branching from an existing one. In the intervening years some lattice code bases have become significantly more user-friendly.

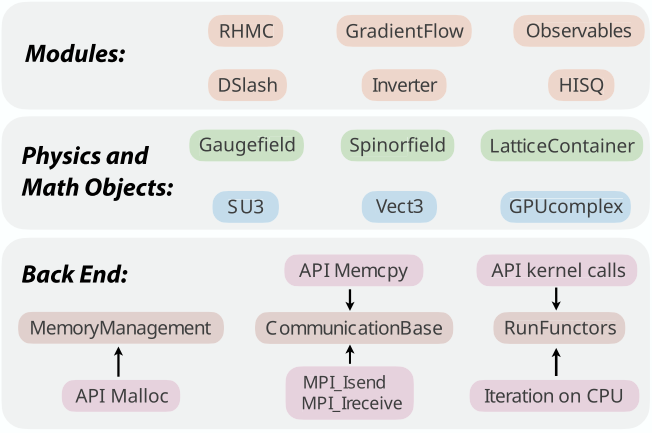

Fig. 1. Diagram illustrating the code's inheritance hierarchy by listing some example classes. Generally speaking, modules inherit from physics and math objects, which in turn inherit from the back end. Image adapted from Ref. [27].

design strategy helps expedite the implementation of additional high-performing, parallelized features, such as other action discretizations, going forward.

This paper is organized as follows: Section 2 explains our design strategy. Details of the higher-level implementations are given in Section 3, followed by the lower-level physics modules in Section 4. Finally we showcase the code's performance in Section 5 and give a brief outlook in Section 6.

2. Design strategy

In the following we discuss key ideas guiding the code's design and mention some of the tools already available for lattice calculations. In order to adequately address the challenges highlighted in Section 1, we have worked to develop code that

1. is high-performing;
2. works efficiently on multiple GPUs and nodes;
3. is flexible to changing architecture and hardware; and
4. is easy to use for lattice practitioners with intermediate C++ knowledge.

In the remainder of the paper, we refer to these Design Goals as DG1, DG2, DG3, and DG4, respectively. We try to connect various design choices to these Goals and try to evaluate our execution of the Goals when we can.

Besides support for multi-GPU via various APIs, the major difference between `SIMULATeQCD` and its predecessors lies in the clear distinction between organizational levels of the code, where much of the technical details are hidden from the highest level. This allows physicists without advanced C++ or hardware knowledge to write highly efficient code without having to understand low-level subtleties. To further help new users, and to improve clarity and reproducibility, we have extensive documentation on our GitHub. The documentation includes for instance examples how to use our modules, how to contribute to the code, and internal parameter choices for various algorithms like the RHMC.

At the highest organizational level are the modules, described in Section 3. Here, we strive to write code that closely and obviously mimics mathematical formulas or short descriptive English sentences. The modules utilize general physics and mathematics classes, which sit at an intermediate level. In turn, these classes inherit from classes of the back end, which is the lowest organizational level. An overview of our code's inheritance scheme is depicted in Fig. 1.

The folder structure of `SIMULATeQCD` is given in Fig. 2. The root directory features the `parameter` folder, which contains example files

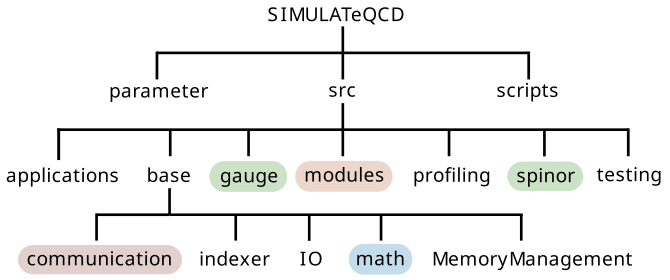

Fig. 2. Folder layout of `SIMULATeQCD`.

that can be used as input for executables. We discuss these files in Section 4.7. The `scripts` folder contains Bash and Python scripts that assist in using the code, such as scripts that run tests or estimate the memory usage. Finally the `src` folder contains the main source code, which is further partitioned as follows:

- `applications`: Ready-to-use applications for generating configurations and carrying out measurements, see Section 2.1.
- `base`: Low-level code implementation, which is discussed in detail in Section 4.
- `gauge`: Implementation of `Gaugefield` objects, described in Section 4.
- `modules`: Modules that are built up from math and physics objects, such as the classes for carrying out updates. These are described in Section 3.
- `profiling`: Ready-to-use applications to profile and check the performance of `SIMULATeQCD`, see Section 2.2.
- `spinor`: Implementation of `Spinorfield` objects, described in Section 4.
- `testing`: Ready-to-use applications for testing the code, see Section 2.2.

2.1. Applications

Due to its design strategy, `SIMULATeQCD` makes it easy for developers to create new multi-GPU applications out of already existing modules. However, many well-tested applications for LQCD calculations already exist. These applications can serve as a starting point for new code, and they have already been used extensively for production in various physics projects. Our applications include, but are not limited to, the following:

- `rhmc`: Generate 2+1 flavor HISQ gauge configurations using a rational hybrid Monte Carlo algorithm. Utilized in Refs. [28–32] for $\mu_B = 0$ and Refs. [33,34] for pure imaginary μ_B. $N_f = 3$ degenerate quarks at $\mu_B = 0$ were generated for Ref. [35]. For details see Section 3.1.
- `generateQuenched`: Generate pure gauge configurations with the Wilson gauge action by applying heat bath and over-relaxation updates. Utilized in Refs. [36–38]. For details see Section 3.1.
- `gradientFlow`: Integrate the gradient flow equation using the Wilson or Zeuthen action with fixed or adaptive step sizes and measure various gauge constructs at each step. Utilized in Refs. [36–40].
- `gaugeFixing`: Fix the gauge of a configuration to the Coulomb or Landau gauge. Can optionally measure various gauge constructs at the end. Utilized in Refs. [41–43].
- `wilsonLinesCorrelator`: Measure the trace of the product of two Wilson lines going in opposite time directions separated by a distance r. Utilized in Refs. [42,43].
- `sublatticeUpdates`: Measure the color-electric correlators and energy-momentum-tensor correlators using the multilevel algorithm.

2.2. Testing and profiling

SIMULATeQCD is constantly being expanded and improved by multiple physicists working in multiple places of the code, and it is of utmost importance that physics results and performance remain stable under these changes. To help check and protect against bugs, we have a large suite of tests as well as some profilers. We only allow code to be merged into the main branch when it is confirmed that all executables compile and all tests pass. Changes to lower level code are asked to further demonstrate no significant loss of performance, for which we use the profilers. When a new feature is implemented, we require the implementer to set up a testing script. Testing strategies vary depending on the feature, but they include comparisons to old code; comparisons to analytic results; performing a calculation in another, independent way; and link-by-link checks against trusted configurations.

3. Physics modules

A typical lattice calculation is divided into two main parts: First gauge configurations are generated, then physical observables are measured on the configurations. Many observables are extracted from correlation functions. Sometimes the observables are extremely noisy, and hence it is necessary to employ various noise-reduction techniques. SIMULATeQCD has modules to accomplish all of these tasks, which we discuss in the present section.

LQCD uses a finite region of discretized 4-d space-time with spatial extensions N_σ and temporal extension N_τ, such that the total number of sites is $N_\sigma^3 \times N_\tau$. Sites are separated by the lattice spacing a. These quantities are related to the physical volume by $V = (aN_\tau)^3$ and temperature by $T = 1/aN_\tau$. Each site has coordinates $(x, y, z, t) \in \mathbb{Z}^4$ with $x, y, z \in \{0, ..., N_\sigma - 1\}$ and $t \in \{0, ..., N_\tau - 1\}$, and each direction is represented by $\mu \in \{0, 1, 2, 3\}$. The gluon fields $U_\mu(x) \in \mathrm{SU}(3)$ are the links, and staggered fermion (spinor) field objects $\chi^c(x) \in \mathbb{C}^3$ [3], where $c \in \{0, 1, 2\}$, rest on the sites. Links are implemented straightforwardly as 3×3 complex matrices and the spinor fields as complex 3-vectors. These fundamental variables are stored in `Gaugefield` and `Spinorfield` objects, which contain arrays of links and spinors, respectively. `Gaugefield` objects have periodic boundary conditions (BCs) while `Spinorfield` objects have periodic spatial BCs and an anti-periodic time BC.

3.1. Configuration generation

Gauge configurations are generated via Markov Chain Monte Carlo using Metropolis-type algorithms [44]. To generate the required random numbers we use a hybrid Tausworthe generator [45,46].

Currently, SIMULATeQCD can generate pure SU(3) configurations and HISQ[3] configurations with $N_f = N_l + N_s$ for N_l light and N_s strange dynamical quarks. Non-integer numbers of flavors are supported as well. To enable simulations of four or more degenerate flavors, the fermion determinant was split up into N_{pf} pseudofermion fields, since the rational approximation requires $N_i < 4$:

$$\prod_{i=1}^{N_{\mathrm{pf}}} \left(\frac{\det\left(\not{D}_{l,i} + m_l\right)}{\det\left(\not{D}_{s,i} + m_s\right)} \right)^{\frac{N_l}{4N_{\mathrm{pf}}}} \prod_{j=1}^{N_{\mathrm{pf}}} \left(\det\left(\not{D}_{s,j} + m_s\right)\right)^{\frac{N_l+N_s}{4N_{\mathrm{pf}}}} . \tag{1}$$

Note that N_l and N_s are input parameters of the rational approximation, while only N_{pf} is an input parameter of SIMULATeQCD. An extension of the code to support different numbers of pseudofermions in the light and strange mass would be straightforward to implement. By default, simulations run with $N_{\mathrm{pf}} = 1$. Also by default, dynamical quark configurations are generated at $\mu_B = 0$, but it is also possible to simulate at pure imaginary μ_B [47].

[3] Our link variable treatment is given in our documentation on GitHub.

For pure SU(3) we have implemented the standard Wilson gauge action [2], with efficient sampling through heat bath (HB) [48–50] and over-relaxation (OR) [51,52] updates. For dynamical fermions, we use the HISQ action [4] and generate configurations using a Rational Hybrid Monte Carlo (RHMC) algorithm [53,54]. To carry out matrix inversions we use a conjugate gradient algorithm, for which multiple right-hand sides [55] (MRHS), multiple shifts [56], and mixed precision implementations are available to improve performance. The RHMC uses a three-scale integrator [57], which naturally profits from the Hasenbusch trick [58]. By default, the integration uses a leapfrog, but an Omelyan (2^{nd} order minimal norm) integrator for the largest scale is also available [59]. The value of the parameter in the force cut-off that we use in the force filter is $\delta = 10^{-5}$ [60]. This value may need to be tuned when one wants to perform QCD thermodynamics calculations very close to the chiral limit.

3.2. Inverters

For matrix inversions, as needed for example in the RHMC algorithm, we implement multiple types of the Conjugate Gradient (CG) algorithm. The CG inverter solves the linear equation $Ax = b$ where A is a symmetric, positive definite matrix, x is the solution vector, and b is the input vector. In our practical applications, A is almost always the $M^\dagger M$ operator where M is the fermion matrix. To separate our inverter implementation of this specific case however, our implementations contain `invert` methods that take an abstract base class `LinearOperator`, as well as two vector field templates representing x and b. The `LinearOperator` class' only member function is the virtual `applyMdaggM` function that defines the interface for the matrix vector product Ay that will be used in the inversion. The two main classes that are used in our production setting are `ConjugateGradient` and `AdvancedMultiShiftCG` with the former offering a standard as well as multi right-hand-side version of the CG algorithm which solves the linear equation $Ax_k = b_k$ with multiple input vectors b_k. The `AdvancedMultiShiftCG` class provides the multi-shift CG algorithm that we use in our RHMC algorithm. It solves the linear equation $\left(A + \sigma_k\right) x_k = b$ for various shifts σ_k. It exploits the shift invariance of the Krylov subspace constructed in the conjugate gradient iteration so that the expensive matrix vector product, here the `applyMdaggM` call, only needs to be performed on the smallest shift σ_0 and solutions for higher shifts σ_i, $i > 0$, are obtained along the way at the cost of few additional AXPY operations [56].

3.3. Measurement of observables

Implementations for physical observables consisting of gauge constructs include: the Polyakov loop, Polyakov loop correlators in the singlet, average, and octet channels, the chiral condensate, the topological charge, topological charge density, its correlator, the Weinberg operator calculated from the field strength tensor and $\mathcal{O}(a^2)$- and $\mathcal{O}(a^4)$-improved field strength tensor [61], the plaquette, the clover, the color-electric and color-magnetic correlator, and energy-momentum tensor correlators.

Hadronic correlation functions along any direction are implemented using HISQ fermions (point sources) to compute, for example, (screening) masses of mesons in various channels. We also support measurement of Taylor coefficients for the expansion of $\log \mathcal{Z}_{\mathrm{QCD}}$ in μ_B/T.

Some observables, such as certain Polyakov loop correlators, must be measured in special gauges. To this end, Coulomb and Landau gauge fixing are implemented via over-relaxation [62] updating, with over-relaxation parameter $\omega = 1.3$.

3.4. Noise reduction

For applications where one needs to attenuate short-distance gauge fluctuations, we have implemented the gradient flow [63] with Wil-

son and Symanzik-improved actions [64] (Zeuthen flow). The numerical integration is carried out using a third-order Runge-Kutta method for Lie groups [65,66] with fixed or adaptive step sizes. The blocking method [39] can be a good supplement to the gradient flow method to substantially improve the signal-to-noise ratio of bosonic correlators or other correlators with disconnected contributions. It is implemented by breaking two planes at different time slices into bins and computing bin-bin correlators.

We have also implemented hypercubic blocking (HYP) smearing [67], which uses hypercubic fat links to minimize the largest fluctuations of the plaquette by maximizing the smallest plaquettes.

The multi-level algorithm [68,69] is implemented by incorporating sublattice updates and observable calculation. The lattice in the temporal direction is divided uniformly into sublattices. Within each sublattice, HB and OR updates are performed in parallel several times followed by a measurement of the observable. At the moment the Polyakov loop, color-electric correlators, and energy-momentum tensor correlators can be measured using this algorithm.

3.5. General calculation of all-to-all correlators

One of the most common types of observable needed in lattice computations is the correlator. Generically a correlator is an operator

$$\text{correlator} = \langle f(A, B; r) \rangle \Big|_D, \tag{2}$$

where A and B are some operators evaluated at space-time positions x and y belonging to the domain D, $r = |x - y|$, and f is an arbitrary function of A and B. In order to calculate this quantity, one has to find all possible pairs of sites at a given r. To save the user from having to solve this problem for their own correlator of interest, in pursuit of DG4, we have implemented a general framework.

The `CorrelatorTools` class handles these general calculations. The fundamental method is the `correlateAt` method

```
correlateAt<f>("domain",A,B).
```
(3)

This correlates arrays `A` and `B` of arbitrary type according to the *correlator archetype* `f`. The site pairing scheme is controlled by `domain`. This returns an array of $\langle f(A, B) \rangle$ indexed by r^2. At the moment, this framework is only available for single GPU use.

4. Low-level implementation

The back end of `SIMULATeQCD` operates close to the hardware level to achieve high performance (DG1), for example through dedicated classes for memory management and communication between host and devices, as lattice applications are usually bound by bandwidth. The four-dimensional lattice also needs to be translated to one-dimensional memory, which is handled by an indexing class. Furthermore, there is a class that maps sets of lattice sites to GPU threads, such that operations on multiple sites are run in parallel. The big advantage of this class is that the operations on each site can be written as high-level functors that do not require any knowledge about the specifics of GPU programming. The back end of the code is finally completed with classes that manage file input/output and logging.

4.1. Allocating and deallocating memory

The memory demands for lattice calculations can increase with increasing lattice size in a nontrivial way, depending on the algorithm being used. Thus for complex applications, it can be difficult to keep track of all dynamically allocated memory on both host and device, especially since GPU memory allocation and deallocation has to be handled via the appropriate API, depending on the hardware. Additionally, to avoid memory leaks, memory should be automatically deallocated when appropriate. Finally, memory (de)allocation can take a non-negligible amount of time, so one can gain a performance boost by allowing large chunks of memory to be shared.

Addressing these issues in a way that is straightforward for lattice practitioners without detailed knowledge of HPC touches on all four Design Goals. Therefore, we developed the centralized `MemoryManagement` class, which manages and knows about all instances of dynamically allocated memory. This is done through the `gMemory` member class, which can hold the raw pointer to dynamically allocated memory along with API-independent wrappers for memory allocation and deallocation. `gMemory` objects are pointed to by our custom smart pointer, the `gMemoryPtr`. A `gMemoryPtr` is labeled by a `SmartName`, a descriptive string chosen by the user that by default prevents that two pointers point to the same memory. However, the memory can be shared by choosing a `SmartName` beginning with the string `SHARED`. Each `gMemoryPtr` object also informs the `MemoryManagement` when it is created and destroyed.

The `MemoryManagement` is never explicitly instantiated, as all necessary methods are static. This includes those needed to create new dynamic memory, destroy it, and report to the user which dynamic memory exists, along with its size and `SmartName`.

4.2. Communication

To address DG2, `SIMULATeQCD` splits a lattice into multiple sublattices, with partitioning possible along any direction. Each sublattice is given to a single GPU, and we call this sublattice the *bulk*. In addition, the GPU holds a copy of the outermost borders of neighboring sublattices, which we call the *halo*. This halo is necessary because many measurement and update processes are stencil operations, which means that a calculation performed at the boundary of the sublattice may need information from a neighboring sublattice. Hence, boundary information from all sublattices must be copied into their neighbors' halos. A schematic drawing of the exchange of halo information between different sublattices is shown in Fig. 3 (top).

Communication between CPU processes is facilitated through MPI, which also enables GPU communication. For both NVIDIA and AMD GPUs, intranode communication is managed using P2P `memcpy` calls, provided by the respective vendor libraries (CUDA and HIP). For internode communication, we utilize GPU-aware MPI when available on the cluster (commonly referred to as CUDA-aware MPI on NVIDIA HPC clusters). On clusters where GPU-aware MPI is not available, we resort to using a normal version of MPI, and hence, GPU-GPU internode communication occurs through the CPUs. An illustration of different communication paths for two nodes is provided in Fig. 3 (bottom). Since halos extend the boundary area around a sublattice, they may use a significant amount of memory. Additionally, the P2P and GPU-aware MPI implementations require additional communication buffers on the GPU compared to the GPU-unaware MPI implementation. Therefore, if communication is not a limiting factor for a specific application, it might be beneficial to turn off GPU-aware MPI/P2P to save GPU memory and use the GPU-unaware MPI communication instead. Hence the code also provides options to choose which communication path should be used (DG1).

Halo communication proceeds by first copying halo information contiguously into a buffer. This requires translating from the sublattice's indexing scheme to the buffer's indexing scheme. The `HaloOffsetInfo` class provides offsets for different halo segments (stripe halo, corner halo, etc). These offsets and the buffer base pointer are used to place the halo data at the correct position in the buffer. An example for the corner halo would be:

$$\text{corner buffer pointer} = \text{buffer base pointer} + \text{corner halo offset}. \tag{4}$$

The `SiteComm` class is the lowest-level class from which all objects that need to communicate across sublattices, such as the `Gaugefield` introduced in the next section, inherit. It uses the

```
template<class floatT, bool onDevice, size_t HaloDepth, CompressionType comp>
struct plaquetteKernel
{
  gaugeAccessor<floatT, comp> gAcc;

  plaquetteKernel(Gaugefield<floatT, onDevice, HaloDepth, comp> &gauge) : gAcc(gauge.getAccessor()) {}

  __device__ __host__ floatT operator()(gSite site){

    floatT result = 0;
    for (int nu = 1; nu < 4; nu++) {
      for (int mu = 0; mu < nu; mu++) {
        SU3<floatT> tmp = gAcc.template getLinkPath<All, HaloDepth>(site, nu, mu, Back(nu));
        result += tr_d(gAcc.template getLinkPath<All, HaloDepth>(site, Back(mu)), tmp);
      }
    }
    return result;
  }
};
```

Listing 1: An example functor, `PlaquetteKernel`, utilizing functor syntax. It is templated to allow for arbitrary precision `floatT`, to run on GPU with `onDevice==True`, for arbitrary `HaloDepth`, and to allow the `Gaugefield` to use arbitrary `CompressionType`. This functor takes a `Gaugefield` object as argument, whose elements are accessed in memory through the `gaugeAcessor` `gAcc`, set to point to the `Gaugefield` accessor in the initializer list. The argument of `operator()` indicates that this functor will be iterated over `gSite` objects. We indicate with `All` that we run over both even and odd parity sites. The method `getLinkPath` multiplies all links starting at `site`, following a path in the specified directions. We compute the real part of the trace with `tr_d`. Due to our functor syntax, all lattice splitting, communication, and distribution to GPU threads is done behind the scenes.

```
const int halodepth = 0;
const bool useGPU = true;
double plaq;

typedef GIndexer<All, halodepth> GInd;

Gaugefield<double, useGPU, halodepth> gauge

LatticeContainer<useGPU, double> latContainer
latContainer.adjustSize(GInd::getLatData().vol4);

latContainer.template iterateOverBulk<All, halodepth>(plaquetteKernel<floatT, HaloDepth>(gauge));
latContainer.reduce(plaq, GInd::getLatData().vol4);

plaq /= (GInd::getLatData().globalLattice().mult()*18);
```

Listing 2: Using an iterator to calculate the plaquette with the functor given in Listing 1. We instantiate our `Gaugefield` object and a `LatticeContainer`, which is needed for the reduction across threads and nodes. Our iterator, `iterateOverBulk` will assign each `gSite` in the bulk of a sublattice to a GPU thread; hence it needs to have $N^3_{\sigma,\mathrm{sub}} \times N_{\tau,\mathrm{sub}} =$ `vol4` elements. The reduction happens over the global lattice, so in the last time we normalize by $N^3_\sigma \times N_\tau$, three colors, and six plaquettes per site in four dimensions.

`MemoryManagement` class to allocate memory for the buffer[4], uses the `HaloOffsetInfo` to translate the local index to the buffer index, copies information into the buffer, and finally uses the `CommunicationBase` to carry out the exchange[5]. This chain of dependencies is illustrated in Fig. 4.

Additionally, we enhance performance (DG1) by allowing the code to perform certain computations concurrently with communication, such as copying halo buffers into the bulk, whenever feasible.

4.3. Indexing and fields

Our site indexing is done in lexicographic order, but for many purposes, it is convenient to characterize sites with an even/odd parity[6]. For applications where this is the case, it is more efficient to organize sites in memory such that all even sites are in the first half of the memory and all odd sites are in the second half. Therefore we convert the 4-d coordinate of a site to the 1-d memory index as

[4] In this context there are a few different kinds of send/receive buffers for the halo, depending on the communication scheme, e.g. GPU-aware MPI or P2P. To help manage this we also have a `HaloSegmentInfo` class, which adds the halo offset to the pointer for the corresponding buffer.

[5] More precisely, we use the `CommunicationBase` in case MPI or GPU-aware MPI is used for communication. In the case of P2P, we just call CUDA/HIP `memcpy` to carry out the communication.

[6] Sometimes referred to as a checkerboard with red/black sites.

```cpp
template<typename Accessor, typename Functor, typename CalcReadInd, typename CalcWriteInd>
__global__ void performFunctor(Accessor res, Functor op, CalcReadInd calcReadInd,
                               CalcWriteInd calcWriteInd, const size_t size_x)

{
    size_t i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i >= size_x) {
        return;
    }
#ifdef USE_CUDA
    auto site = calcReadInd(blockDim, blockIdx, threadIdx);
#elif defined USE_HIP
    auto site = calcReadInd(dim3(blockDim), GetUint3(dim3(blockIdx)), GetUint3(dim3(threadIdx)));
#endif
    res.setElement(calcWriteInd(site), op(site));
}


template<bool onDevice, class Accessor>
template<unsigned BlockSize, typename CalcReadInd, typename CalcWriteInd, typename Functor>
void RunFunctors<onDevice, Accessor>::iterateFunctor(Functor op, CalcReadInd calcReadInd,
                                                     CalcWriteInd calcWriteInd, const size_t elems_x)
{
    dim3 blockDim;

    blockDim.x = BlockSize;

    const dim3 gridDim = static_cast<int> (ceilf(static_cast<float> (elems_x)
                / static_cast<float> (blockDim.x)));

    if (onDevice) {
#ifdef USE_CUDA
        performFunctor<< gridDim, blockDim, 0, stream >> (getAccessor(), op, calcReadInd,
                                                          calcWriteInd, elems_x);
#elif defined USE_HIP
        hipLaunchKernelGGL(performFunctor, dim3(gridDim), dim3(blockDim), 0, stream ,
                           getAccessor(), op, calcReadInd, calcWriteInd, elems_x);
#endif
    } else {
        /// ...CPU implementation
    }
}
```

Listing 3: Sketch of low-level functor and iterator implementation. API-dependent constructions are wrapped inside `performFunctor` and `iterateFunctor` methods. The user can decide at compile time whether to use the CUDA or HIP API, which is implemented here using `ifdef`. These methods are templated to allow memory access to an arbitrary return type `res` using memory accessor `Accessor`. The operation `op` is at this level completely unspecified; it will be defined at a higher level through a functor, for example the plaquette functor shown in Listing 1. The types `CalcReadInd` and `CalcWriteInd` respectively translate from GPU thread index to input memory index and from input to output memory index. The `site` object in `performFunctor` is set to `auto` to allow use with arbitrary indexing object, for example the `gSite` and `gSiteMu` objects defined in Section 4.3.

$$\begin{aligned}\text{index} = &\frac{1}{2}\left(x + yN_\sigma + zN_\sigma^2 + tN_\sigma^3\right) \\ &+ \frac{1}{2}N_\sigma^3 N_\tau\,(x+y+z+t) \bmod 2.\end{aligned} \tag{5}$$

Building up from this, links are indexed by

$$\begin{aligned}\text{index} = &\frac{1}{2}\left(x + yN_\sigma + zN_\sigma^2 + tN_\sigma^3\right) \\ &+ \frac{1}{2}N_\sigma^3 N_\tau\,(x+y+z+t) \bmod 2 + \mu N_\sigma^3 N_\tau.\end{aligned} \tag{6}$$

When using multiple GPUs, similar formulae hold for the sublattices, except that the extensions are replaced by $N_\sigma \to N_{\sigma,\text{sub}}$ and $N_\tau \to N_{\tau,\text{sub}}$. Hence we distinguish between the local index and global index. Moreover, as explained in the previous section and as depicted in Fig. 3, information about neighboring fields is stored in a halo surrounding the bulk. Therefore besides bulk indexing, each sub-lattice has so-called "full" indices that include the halos. This scheme is computed by

$$\begin{aligned}\text{index}_{\texttt{Full}} = &\frac{1}{2}\left(x + yN_x + zN_xN_y + tN_xN_yN_z\right) \\ &+ \frac{1}{2}N_xN_yN_zN_t\,(x+y+z+t) \bmod 2,\end{aligned} \tag{7}$$

while the links are indexed by

$$\begin{aligned}\text{index}_{\texttt{Full}} = &\frac{1}{2}\left(x + yN_x + zN_xN_y + tN_xN_yN_z\right) \\ &+ \frac{1}{2}N_xN_yN_zN_t(x+y+z+t) \bmod 2 \\ &+ \mu N_xN_yN_zN_t,\end{aligned} \tag{8}$$

with

```cpp
enum Operation { add, subtract, mult, divide };

template<typename typeLHS, typename typeRHS, Operation op, typename testLHS=void, typename testRHS=void>
struct GeneralOperator {};

template<typename typeLHS, typename typeRHS>
auto general_add(const typeLHS &lhs, const typeRHS &rhs) {
    return GeneralOperator<typeLHS, typeRHS, add>(lhs, rhs);
}

// Class + Class: Using Multiple type specialization with SFINAE:
template<typename typeLHS, typename typeRHS>
struct GeneralOperator<
        typeLHS, // Left hand side object
        typeRHS, // Right hand side object
        add, // Type of operator (add mult divide subtract)
        std::enable_if_t<custom_is_class<typeLHS>::value>, // SFINAE to check if typeLHS is a class
        std::enable_if_t<custom_is_class<typeRHS>::value> // SFINAE to check if typeRHS is a class
> {
    // We need to get the type of the return value of getAccessor(). This is what is stored here.
    const decltype(std::declval<typeLHS>().getAccessor()) _lhs;
    const decltype(std::declval<typeRHS>().getAccessor()) _rhs;

    // Constructor copies the accessors
    GeneralOperator(const typeLHS &lhs, const typeRHS &rhs) :
            _lhs(lhs.getAccessor()), _rhs(rhs.getAccessor()) {}

    // GeneralOperator is its own accessor
    GeneralOperator< typeLHS, typeRHS, add,
            std::enable_if_t<custom_is_class<typeLHS>::value>,
            std::enable_if_t<custom_is_class<typeRHS>::value> >
    getAccessor() const { return *this; }

    // Call the operator: Do the operation element wise.
    // This is what is called in another operator or in a kernel which runs the operation.
    template<typename Index>
    inline __host__ __device__ auto operator()(const Index i) const {
        auto rhs = _rhs(i);
        auto lhs = _lhs(i);
        return lhs + rhs;
    }
};
```

Listing 4: An excerpt from the `GeneralOperator` implementation showing the specialization of `GeneralOperator` to add two "Class" objects.

$$N_i = N_\sigma + H_i, \qquad i \in x, y, z,$$
$$N_t = N_\tau + H_t,$$

where H_i, H_t are the halo depths in different directions.

In order to abstract these difficulties away from the user, we have implemented `gSite` objects. A `gSite` object holds all information about a site, including its coordinates and index in the local, global, and full context. All methods for calculating indices, coordinates, and movement along the lattice are contained in the `GIndexer` class. With DG4 in mind, we make it easy for the user to control indexing by passing a `Layout` template parameter, which can be set to `Even`, `Odd`, or `All`. Similarly, the `gSiteMu` class holds all information about link indexing.

4.4. Functor syntax

Much of the effort in this code goes into abstracting away highly complex parallelization, which depends on the API, whether GPUs or CPUs are used, the number of processes, the node layout being used, and so on. Accomplishing this for the general case is crucial to DG4; hence we have implemented a system where one can iterate an arbitrary operation that depends on arbitrary arguments over an arbitrary set of coordinates, taking care of the parallelization behind the scenes.

One common task in lattice calculations is to perform the same calculation on each lattice site, which involves link variables at or close to that site, for example, computing the plaquette,

$$U^{\square}_{\mu\nu}(x) = U_\mu(x) U_\nu(x + a\hat{\mu}) U^\dagger_\mu(x + a\hat{\nu}) U^\dagger_\nu(x). \tag{9}$$

An example *kernel* computing the plaquette is shown in Listing 1. The code for the kernel is wrapped in a *functor*, i.e. a `struct` with its `operator()` overloaded, which in this case takes a `gSite` object as argument. The functor is passed to a function which we call an *iterator*. This function iterates over all required lattice sites and calls the functor on each one, thereby distributing the calls on the available computing resources. Basic iterators are provided by the `RunFunctors` class which implements iteration over the bulk, the full lattice including halos, as well as a looping iterator that can iterate over the lattice and an additional index. Physics classes like the `Gaugefield` and `Spinorfield` that derive from `RunFunctors` use these to create specialized iterators to iterate for example over all links of a `Gaugefield` in a specific direction or a subset of RHS for our multi-RHS `Spinorfield`. An example usage of this is given in Listing 2, where the `iterateOverBulk` iterator is called using the functor `plaquetteKernel` as a template parameter. By using functors as template parameters for iterators, we can conveniently iterate arbitrary

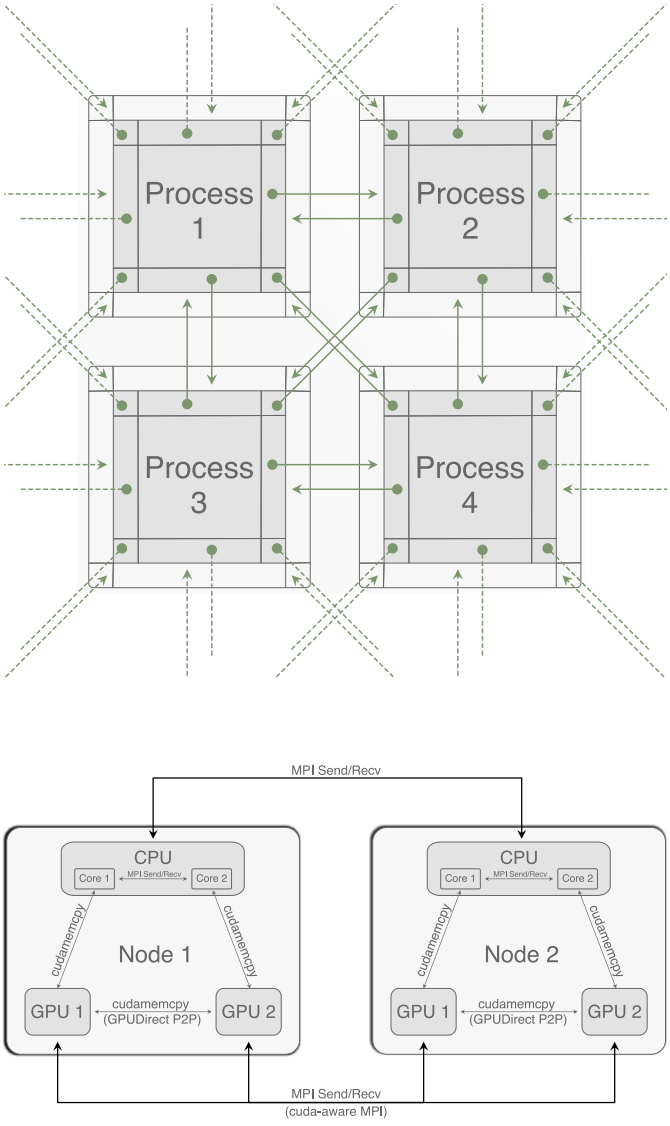

Fig. 3. *Top*: Schematic halo exchange for four processes in two dimensions, each process containing a sublattice. The bulk is indicated by the dark gray squares, and the halo is indicated by light grey squares. Copies of sites located near the borders of the bulk, set off here by solid grey lines, are stored in the halos. Information starts at solid green circles and is injected into the halos following the arrows. Dotted lines indicate injections following periodic boundary conditions. *Bottom*: Illustration of the topology of a fictitious HPC system which consists of two nodes with two (NVIDIA) GPUs per node. The arrows represent all possible communication paths. Images taken from Ref. [25]. (Color online.)

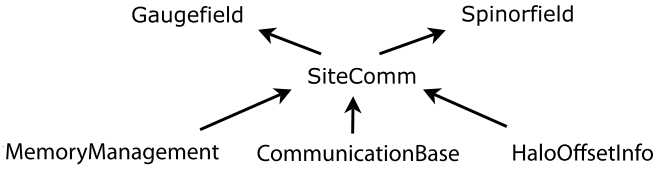

Fig. 4. Dependence structure of the `Gaugefield` and `Spinorfield` on communication classes.

calculations over sites, without the need to write specialized code each time.

Iterators such as `iterateOverBulk`, which calls the `struct`'s `operator()` at bulk sites, inherit from the class `RunFunctors`, which contains the lowest-level methods that iterate functors over the desired target set. In Listing 3 we provide sketches of the most salient methods in the `RunFunctors` class, namely `performFunctor`, which applies a functor to a `gSite` or `gSiteMu` object, and `iterateFunctor`, which carries out `performFunctor` using CUDA or HIP.

If we are not using GPUs and just evaluate the computation kernel on multiple CPUs, then the iterator method iterates sequentially over the sites/links of a sublattice using `for`-loops, with one sublattice per CPU core in parallel. But if we use GPUs, then there are no `for`-loops. Instead, the computation of each iteration of these nested `for`-loops will also be parallelized on threads of the GPUs. For example with the plaquette computation,

$$\text{number of GPU threads} = \frac{N_\sigma^3 \times N_\tau}{\text{number of GPUs}} \tag{10}$$

are spawned, where each thread is doing the work corresponding to a certain site.

As mentioned in Section 4.3, we index sites and links through `gSite` and `gSiteMu` objects, respectively; hence one usually passes either a `gSite` or `gSiteMu` object to the functor. The `runFunctors` class also contains several `CalcGSite` methods that tell the iterator how to translate from these objects to GPU thread indices.

4.5. Expression templates

To facilitate an efficient but intuitive composition of math expressions involving basic physics objects such as spinorfields and gaugefields (DG4), we implement general math operations such as additions, subtractions, multiplications and divisions via expression templates [70].

We define a set of functions such as `general_add` and `general_mult` that, instead of performing the corresponding math operation, return a functor `GeneralOperator` that holds the information to carry out the desired calculation. The execution of this calculation is triggered by the copy assignment operator (=) of the basic physics object, which passes the functor to an `iterateFunctor` method.

Using the "Substitution failure is not an error" (SFINAE) principle, we have specialized the `GeneralOperator` struct for different type combinations such as "Class + Class" and "Class + scalar". In this context, "Class" means an object that has a `getAccessor`-method and operator() defined. This object could be a `Gaugefield`, a `Spinorfield` or another `GeneralOperator`. Nesting of expressions such as $a \cdot b + c \cdot d$, where a, b, c, d are physics objects, and combinations of operators and scalars such as $(a \cdot b) \cdot 2.0$ are also implemented. This highly templated, low-level code allows users to write high-level physics code involving `Spinorfield`s, `Gaugefield`s, etc. that closely resembles familiar mathematical equations while also avoiding superfluous evaluations of temporary expressions.

Listing 4 shows an excerpt of the `GeneralOperator` implementation detailing the specialization of additions of two class objects.

4.6. LatticeContainer and reduction

In lattice QCD computations, intermediate results per lattice site are typically calculated. These intermediate results are then reduced over the entire lattice, such as by summing over the entire lattice. When a lattice is divided among multiple processes, the procedure becomes slightly more complex:

1. Intermediate results are computed in parallel on all sites of all sublattices.
2. Each process performs a reduction into a single quantity.
3. A final reduction is performed across all processes into a single quantity, which is then distributed over all processes if necessary.

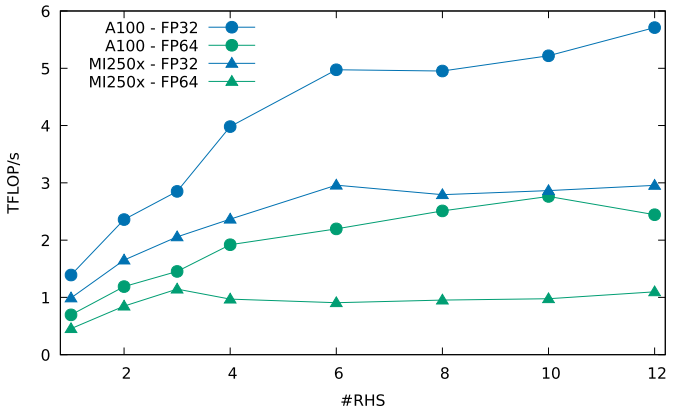

Fig. 5. Performance of the HISQ Dirac operator with varying number of RHS vectors on a 32^4 lattice on a single A100 GPU and single MI250x GCD.

In order to address DG4, this common task has been combined into a single reduction method which is part of the `LatticeContainer` class. This class is a container of dimension $N_\sigma^3 \times N_\tau$ which elements can be of any datatype and is distributed over the processes similarly to the `Gaugefield` or `Spinorfield`. By calling that reduction method, these elements are automatically reduced across all GPUs.

In addition to a standard sum over all elements, the `LatticeContainer` class also provides a method to reduce only the time slices of the lattice, resulting in a vector of length N_τ. This is particularly useful for computing correlation functions in the time direction. Furthermore, there is the stacked reduction, which can be used to reduce the intermediate results of a stacked spinor. Internally, all these reduction operations are performed using MPI's collective communication calls in combination with CUB (NVIDIA) and hipCUB (AMD).

4.7. Parameter handling and IO

Parameter files hold input parameters such as quark masses and couplings, as well as information like the lattice dimension, random number seed, etc. These are implemented through our `LatticeParameters` class. More specialized parameter classes, for example the parameter class for the RHMC, inherit from this. Default parameter files are in the `parameter` folder mentioned in Section 2, but one can also pass a custom parameter file as argument to any executable.

`SIMULATeQCD` supports the International Lattice Data Grid (ILDG) format for reading and writing the gauge configurations [71]. The gauge configuration files for ILDG format are packaged together with a corresponding XML metadata (QCDml) file [72]. To assist with forming a QCDml file, one can pass optional, ILDG-compliant parameters to the `LatticeParameters` class that are printed to the standard output. These metadata can then be captured by custom scripts. In addition to ILDG format, `SIMULATeQCD` also has support for MILC [16], NERSC, and openQCD [24] configuration formats, three other commonly used formats in the lattice community.

5. Performance

In this section we showcase our code's performance, giving quantitative insight on our ability to achieve DG1.

When generating HISQ configurations for typical parameters, about 60% of our RHMC run time is spent inverting the Dirac matrix via conjugate gradient, and in it, applying the $\not{D}$ operator to a vector is the most performance-critical kernel. Hence this and related kernels are important benchmarks.

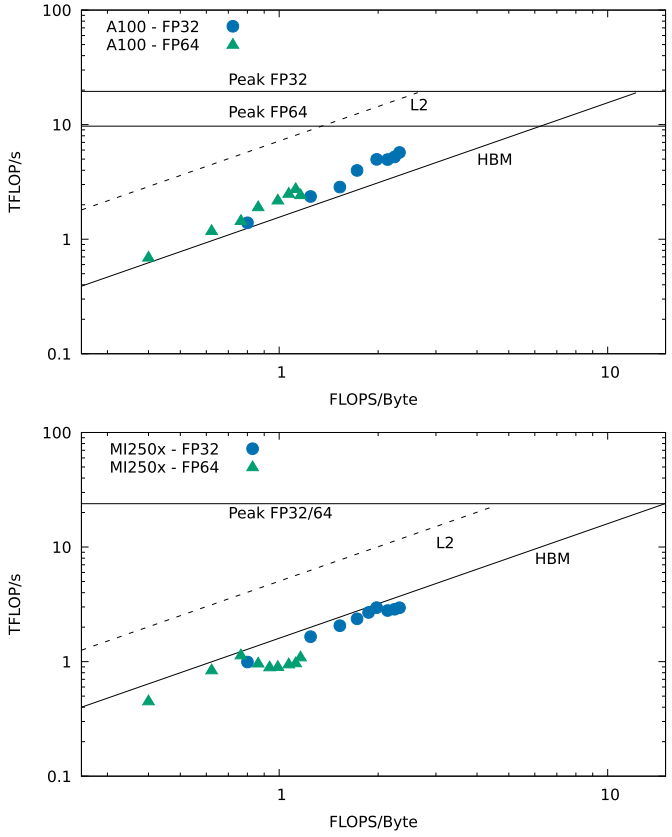

Fig. 6. Roofline plots of the HISQ Dirac operator with varying number of RHS vectors on a 32^4 lattice on a single A100 GPU and single MI250x GCD.

5.1. HISQ Dslash performance model

To understand the HISQ Dslash's performance, we will use a simple roofline model. The action of the kernel on a vector ϕ_x at a given lattice site x takes the form

$$\not{D}\phi_x = \sum_{\mu=0}^{4} \Big[\left(X_{x,\mu}\phi_{x+\hat{\mu}} - X^\dagger_{x-\hat{\mu},\mu}\phi_{x-\hat{\mu}} \right) + \left(W_{x+3\hat{\mu}}\phi_{x+3\hat{\mu}} - W^\dagger_{x-3\hat{\mu},\mu}\phi_{x-3\hat{\mu}} \right) \Big],$$

where X and W are smeared gaugefields and we ignore constants and the staggered phases for convenience. In each direction, the kernel performs four complex three-dimensional matrix-vector multiplications as well as three vector additions, which results in 1146 floating point operations (FLOP) per site x. Meanwhile, the kernel has to read 288 floats for the link matrices X and W, 96 floats for the vectors at $x \pm \hat{\mu}$ and $x \pm 3\hat{\mu}$ and write 6 floats at site x. This results in an arithmetic intensity of

$$\mathrm{FLOP/BYTE}(\not{D}) = \frac{1146\,\mathrm{FLOP/site}}{(288+96+6)\mathrm{words/site}},$$

which translates to arithmetic intensities of about 0.73 FLOP/Byte in single precision and 0.36 FLOP/Byte in double precision. As the ratio of theoretical peak FLOP/s to memory bandwidth of modern GPUs is around $\mathcal{O}(10)$ FLOP/Byte, the HISQ Dslash's performance is heavily bound by the devices memory bandwidth. This can be improved in situations where the kernel can be applied to multiple right-hand-side (RHS) vectors at once. The loaded gaugefields X and W can then be re-used, resulting in an arithmetic intensity that scales with the number of RHS vectors like

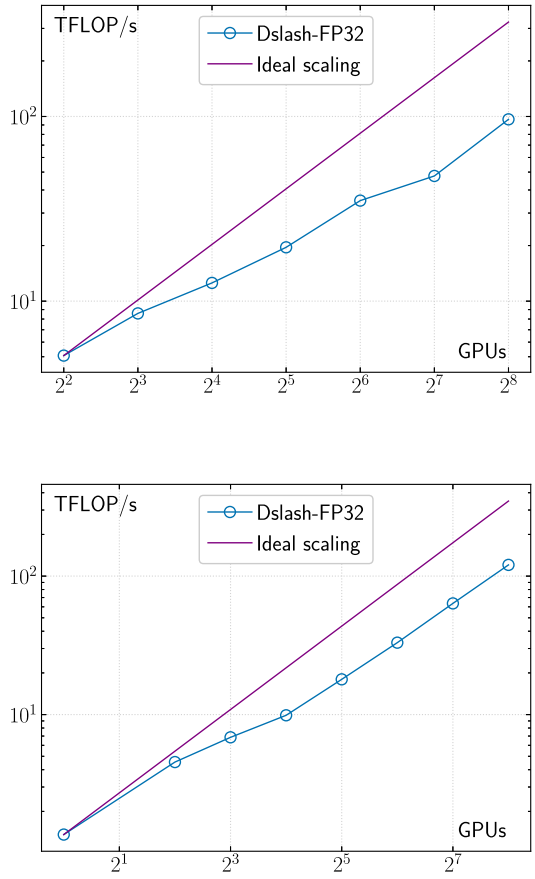

Fig. 7. Scaling of HISQ Dirac operator with a single RHS on Perlmutter. *Top*: Strong scaling for a 96^4 lattice. *Bottom*: Weak scaling for a 32^4 local lattice.

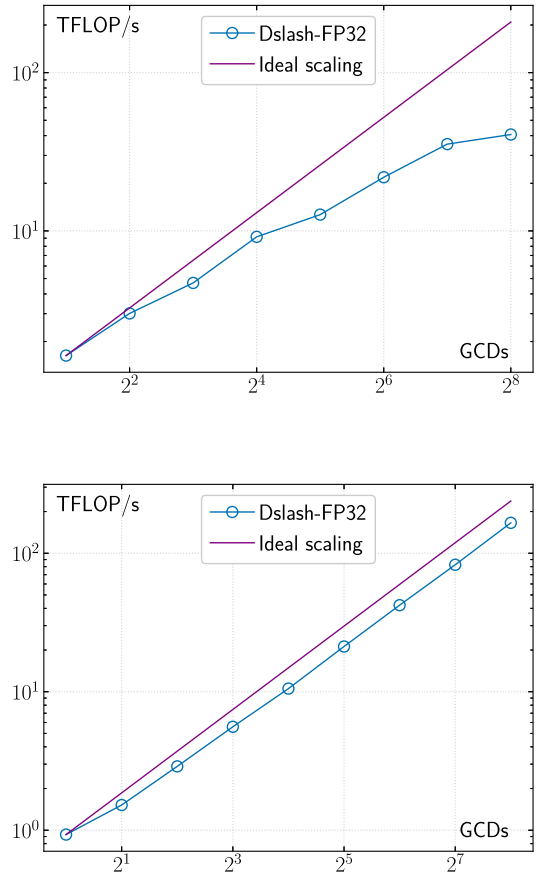

Fig. 8. Scaling of HISQ Dirac operator with single RHS on Frontier. *Top*: Strong scaling for a 96^4 lattice. *Bottom*: Weak scaling for a 32^4 local lattice.

$$\text{FLOP/BYTE}(\not{D})_{n-RHS} = \frac{n \cdot 1146\,\text{FLOP/site}}{(288 + n \cdot (96+6))\text{words/site}}.$$

Our implementation offers two methods to accommodate applying the Dslash kernel to multiple RHS vectors: an inner loop over N_{inner} RHS vectors inside each thread of the kernel so that data re-use via registers or L1 cache is possible as well as an outer loop over N_{outer} RHS vectors as an additional thread-block dimension in which case re-use can only happen via the L2 cache. Both can be combined so that the kernel is applied to $N_{\text{inner}} \times N_{\text{outer}}$ RHS vectors. With the above arithmetic intensity at hand, we can benchmark our HISQ Dslash implementation and compare the achieved performance against the performance rooflines stemming from the GPUs peak FLOP/s as well as memory bandwidths, which we show in the following two subsections.

5.2. Benchmarks on A100 GPUs

The performance of our multi-RHS Dslash implementation for varying number of RHS vectors on a single A100 GPU is shown in Fig. 5. For each number of RHS vectors, we tested all possible combinations of N_{inner} and N_{outer} and show the best performing combination in the plot.

We see significant performance improvements from Gaugefield re-use with increasing number of RHS vectors up to 6 RHS vectors and a further slight increase in performance up to 12 RHS vectors in the single precision case. For double precision, we see a more gradual increase in performance up to 10 RHS vectors. For both precision, we found the best performance when keeping N_{inner} and N_{outer} of similar size but avoid $N_{\text{inner}} > 6$ as that leads to register spilling.

In Fig. 6 we compare our kernels performance to rooflines dictated by the A100's theoretical peak FLOP/s as well as HBM DRAM and L2 cache bandwidths. We find that for both FP32 and PF64 arithmetic and all number of RHS vectors, we surpass the performance ceiling imposed by the A100's HBM bandwidth of 1.55 TB/s. Closer inspection of the kernels performance using NVIDIA's Nsight compute profiler shows that we saturate about 85% of the theoretical peak HBM bandwidth but efficient cache usage lifts our kernels effective bandwidth above that limit.

In Fig. 7 we show strong- and weak-scaling benchmarks of our HISQ Dslash on Perlmutter, which has 4 NVIDIA A100 GPUs per node. Nodes are interconnected via HPE Slingshot 11 fabric and have 4 NICs providing 25 GB/s bandwidth each. We find good weak scaling on this system up to 256 GPUs. A slight decrease in speedup can be seen as soon as node-to-node communication starts. Strong scaling benchmarks with a 96^4 global lattice size start deviating from ideal scaling earlier. As the local lattice sizes and thus the compute workloads of the individual GPUs keep getting smaller, hiding the HISQ Dslash's communication becomes increasingly difficult.

5.3. Preliminary benchmarks on MI250X GPUs

The performance of our Dslash implementation on a single MI250x GCD for varying number of RHS vectors is shown in Fig. 5 with the corresponding roofline plot in the bottom half of Fig. 6. For single precision arithmetic, we can see an increase in performance up to 6 RHS vectors after which the performance plateaus, whereas for double precision arithmetic, performance increases only up to 3 RHS vectors and then flattens. In contrast to our A100 benchmarks, we only observe a significant performance gain from using multiple RHS when we iterate over the vectors inside the inner loop, i.e. keeping $N_{\text{outer}} = 1$. We suspect that this is due to the MI250x's much smaller L2 cache of 16 MB compared to the A100's 40 MB L2 cache. In the roofline plot, we can see that we stay consistently below the HBM bandwidth im-

Table 1
Data for Fig. 5, showing the performance in TFLOP/s of the HISQ Dirac operator with varying number of RHS vectors on a 32^4 lattice on a single A100 GPU and a single MI250x GCD.

RHS	A100		MI250x	
	FP32	FP64	FP32	FP64
1	1.36	0.70	0.93	0.45
2	2.36	1.19	1.65	0.85
3	2.85	1.45	2.06	1.14
4	3.98	1.92	2.36	0.97
6	4.97	2.20	2.96	0.91
8	4.95	2.51	2.79	0.95
10	5.22	2.76	2.86	0.98
12	5.71	2.44	2.95	1.10

Table 2
Data for Fig. 7, showing the strong and weak scaling of the HISQ Dirac operator on Perlmutter with a 96^4 global lattice and 32^4 local lattice, respectively.

Strong scaling		Weak scaling	
GPUs	TFLOP/s	GPUs	TFLOP/s
1	-	1	1.36
4	5.07	4	4.54
8	8.58	8	6.85
16	12.55	16	9.88
32	19.56	32	17.96
64	35.02	64	33.06
128	47.62	128	63.48
256	96.47	256	120.44

Table 3
Data for Fig. 8, showing the strong and week scaling of the HISQ Dirac operator on Frontier with a 96^4 global lattice and 32^4 local lattice, respectively.

Strong scaling		Weak scaling	
GCDs	TFLOP/s	GCDs	TFLOP/s
1	-	1	0.93
2	1.63	2	1.52
4	3.01	4	2.89
8	4.69	8	5.58
16	9.17	16	10.53
32	12.67	32	21.20
64	21.82	64	42.15
128	35.34	128	82.50
256	40.63	256	165.72

posed performance limit. In Fig. 8 we show HISQ Dslash benchmarks on Frontier. This system is configured with 4 AMD MI250X per node. Each MI250X is comprised of 2 Graphic Compute Dies (GCDs); i.e. there are 8 GCDs per node. GCDs inside an MI250X are connected via Infinity Fabric with 200 GB/s bi-directional bandwidth, and the 4 MI250X cards on a node are connected via Infinity Fabric with bi-directional bandwidths between 50-100 GB/s depending on their arrangement. Nodes are connected via four HPE Slingshot NICs each providing 25 GB/s of bandwidth. The weak scaling benchmark with a 32^4 local lattice volume shows close to ideal scaling all the way up to 256 GCDs. Going from 8 GCDs to 16 GCDs shows no significant drop in speedup although node-to-node communications kick in. The strong scaling benchmark scales well up until 16 GCDs. After that point, communication can no longer be hidden effectively and the speedup decreases. First tests on LUMI-G produced comparable single GCD performance as that in Frontier. Performance data are summarized in Tables 1–3.

6. Outlook

We presented `SIMULATeQCD`, a multi-GPU, multi-node lattice code that allows simulation of dynamical fermions using the HISQ action and works for both NVIDIA and AMD back ends. We chose the HISQ action since it is widely used for state-of-the-art, high-statistics QCD studies. In writing `SIMULATeQCD`, we sought to write clean code that is highly modularized, so that anyone with modest knowledge of C++ can get started writing production code right away. We believe this is an important and sometimes overlooked characteristic of code in a lattice context, where personnel and hardware are constantly changing. At the same time, we did our best to ensure good performance by writing code close to the hardware that is both mindful of and flexible to the computing system. We leveraged functor syntax to balance both of these needs. `SIMULATeQCD` has many production-ready applications, which are listed in Section 2.1. We gave some details on the implementation of our physics modules in Section 3 for easy and transparent reference in the future.

What exists in `SIMULATeQCD` presently mostly includes code that has been relevant to HotQCD projects. One of the most important characteristics of `SIMULATeQCD` is that it is relatively straightforward to implement new algorithms; hence wherever there is desire and manpower, our code can be extended to suit the needs of a new project in an uncomplicated, undemanding way. One of the obvious avenues of extension for `SIMULATeQCD` on the physics side includes implementation of other 4-d actions[7], such as the Wilson action for fermions. While the code already works for $N_f = 2 + 1$ and degenerate N_f systems, some work is still needed to include a dynamical charm quark. A relatively new feature allows measurement of Taylor coefficients of the pressure, however this still needs porting a deflated solver algorithm to be suitable for efficient use with lighter quarks. Finally, the code is already relatively integrated with ILDG, and this should ideally be continued in parallel with the ILDG revival.

Turning to low-level implementation, machines such as Aurora are utilizing Intel GPUs, so it will be valuable to introduce also a Sycl back end. Along this vein, while most modules require a GPU back end, everything can also be compiled and run on CPUs only[8] by adding a few macros, which so far has been done for a few selected modules.

CRediT authorship contribution statement

Lukas Mazur: Writing – review & editing, Validation, Software, Project administration, Methodology, Conceptualization. **Dennis Bollweg:** Writing – review & editing, Validation, Software. **David A. Clarke:** Writing – review & editing, Writing – original draft, Validation, Software. **Luis Altenkort:** Writing – review & editing, Validation, Software. **Olaf Kaczmarek:** Supervision, Project administration, Funding acquisition, Conceptualization. **Rasmus Larsen:** Software. **Hai-Tao Shu:** Software. **Jishnu Goswami:** Software. **Philipp Scior:** Validation, Software. **Hauke Sandmeyer:** Software, Conceptualization. **Marius Neumann:** Software. **Henrik Dick:** Software. **Sajid Ali:** Software. **Jangho Kim:** Software. **Christian Schmidt:** Supervision, Software, Funding acquisition. **Peter Petreczky:** Writing – review & editing. **Swagato Mukherjee:** Supervision, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

[7] Domain wall fermions will require an overhaul of the existing indexer, or the implementation of a new one.

[8] It should be noted that at the moment, we have not yet implemented vectorized CPU code.

Data availability

The code is available on github and zenodo. Performance data are included in this document.

Acknowledgements

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Nuclear Physics through Contract No. DE-SC0012704, and within the framework of Scientific Discovery through Advance Computing (SciDAC) award Fundamental Nuclear Physics at the Exascale and Beyond.

This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract No. DE-AC05-00OR22725.

This research used awards of computer time provided by the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231.

We acknowledge EuroHPC JU for awarding this project access to LUMI-G at CSC Finland and the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time through the John von Neumann Institute for Computing (NIC) on the GCS Supercomputer JUWELS at Jülich Supercomputing Centre (JSC). We thank EuroHPC JU for their support within the extraordinary support program (ESP).

This work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) Proj. No. 315477589-TRR 211 and the European Union under Grant Agreement No. H2020-MSCAITN-2018-813942. This work was partly performed in the framework of the PUNCH4NFDI consortium supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 460248186 (PUNCH4NFDI).

J.K. was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) through the funds provided to the Sino-German Collaborative Research Center TRR110 “Symmetries and the Emergence of Structure in QCD” (DFG Project-ID 196253076 - TRR 110).

The authors gratefully acknowledge computing time provided to them on the high-performance computers Noctua2 at the NHR Center PC2. These are funded by the Federal Ministry of Education and Research and the state governments participating on the basis of the resolutions of the GWK for the national highperformance computing at universities (www.nhr-verein.de/unsere-partner).

We thank the Bielefeld HPC.NRW team for their support and M. Klappenbach for his hard work maintaining the Bielefeld cluster. We thank H. Simma for useful technical discussions relating to the ILDG format. Thanks to A. R. Gannon for the design of Figs. 2 and 4. Finally we extend a big thanks to G. Curell for implementing a container.

References

[1] A. Bazavov, et al., Nonperturbative QCD simulations with 2+1 flavors of improved staggered quarks, Rev. Mod. Phys. 82 (2010) 1349–1417, https://doi.org/10.1103/RevModPhys.82.1349, arXiv:0903.3598.

[2] K.G. Wilson, Confinement of quarks, Phys. Rev. D 10 (1974) 2445–2459, https://doi.org/10.1103/PhysRevD.10.2445.

[3] J.B. Kogut, L. Susskind, Hamiltonian formulation of Wilson’s lattice gauge theories, Phys. Rev. D 11 (1975) 395–408, https://doi.org/10.1103/PhysRevD.11.395.

[4] E. Follana, et al., HPQCD, UKQCD collaboration, Highly improved staggered quarks on the lattice, with applications to charm physics, Phys. Rev. D 75 (2007) 054502, https://doi.org/10.1103/PhysRevD.75.054502, arXiv:hep-lat/0610092.

[5] A. Bazavov, et al., The chiral and deconfinement aspects of the QCD transition, Phys. Rev. D 85 (2012) 054503, https://doi.org/10.1103/PhysRevD.85.054503, arXiv:1111.1710.

[6] K. Orginos, Innovations in lattice QCD algorithms, J. Phys. Conf. Ser. 46 (2006) 132–141, https://doi.org/10.1088/1742-6596/46/1/018, https://iopscience.iop.org/article/10.1088/1742-6596/46/1/018.

[7] A.S. Kronfeld, Lattice gauge theory with staggered fermions: how, where, and why (not), PoS LATTICE2007 (2007) 016, https://doi.org/10.22323/1.042.0016, arXiv:0711.0699.

[8] C. Bernard, M. Golterman, Y. Shamir, Effective field theories for QCD with rooted staggered fermions, Phys. Rev. D 77 (2008) 074505, https://doi.org/10.1103/PhysRevD.77.074505, arXiv:0712.2560.

[9] H.-T. Ding, F. Karsch, S. Mukherjee, Thermodynamics of strong-interaction matter from lattice QCD, Int. J. Mod. Phys. E 24 (10) (2015) 1530007, https://doi.org/10.1142/S0218301315300076, arXiv:1504.05274.

[10] C. Schmidt, S. Sharma, The phase structure of QCD, J. Phys. G 44 (10) (2017) 104002, https://doi.org/10.1088/1361-6471/aa824a, arXiv:1701.04707.

[11] J.N. Guenther, Overview of the QCD phase diagram: recent progress from the lattice, Eur. Phys. J. A 57 (4) (2021) 136, https://doi.org/10.1140/epja/s10050-021-00354-6, arXiv:2010.15503.

[12] A. Bazavov, et al., $B_s \to K\ell\nu$ decay from lattice QCD, Phys. Rev. D 100 (3) (2019) 034501, https://doi.org/10.1103/PhysRevD.100.034501, arXiv:1901.02561.

[13] C.T.H. Davies, et al., Windows on the hadronic vacuum polarisation contribution to the muon anomalous magnetic moment, arXiv:2207.04765.

[14] Z. Fan, W. Good, H.-W. Lin, Gluon parton distribution of the nucleon from (2+1+1)-flavor lattice QCD in the physical-continuum limit, Phys. Rev. D 108 (1) (2023) 014508, https://doi.org/10.1103/PhysRevD.108.014508, arXiv:2210.09985.

[15] B. Chakraborty, C.T.H. Davies, B. Galloway, P. Knecht, J. Koponen, G.C. Donald, R.J. Dowdall, G.P. Lepage, C. McNeile, High-precision quark masses and QCD coupling from $n_f = 4$ lattice QCD, Phys. Rev. D 91 (5) (2015) 054508, https://doi.org/10.1103/PhysRevD.91.054508, arXiv:1408.4169.

[16] MILC collaboration code for lattice qcd calculations, https://github.com/milc-qcd/milc_qcd.

[17] M.A. Clark, R. Babich, K. Barros, R.C. Brower, C. Rebbi, Solving lattice QCD systems of equations using mixed precision solvers on GPUs, Comput. Phys. Commun. 181 (2010) 1517–1528, https://doi.org/10.1016/j.cpc.2010.05.002, arXiv:0911.3191.

[18] M.A. Clark, et al., lattice/quda: Quda v1.1.0, https://doi.org/10.5281/zenodo.5610079, Oct. 2021.

[19] R.G. Edwards, B. Joo, The Chroma software system for lattice QCD, Nucl. Phys. B, Proc. Suppl. 140 (2005) 832, https://doi.org/10.1016/j.nuclphysbps.2004.11.254, arXiv:hep-lat/0409003.

[20] The Chroma software system for lattice QCD, https://github.com/JeffersonLab/chroma/tree/master.

[21] USQCD: US lattice quantum chromodynamics, https://usqcd-software.github.io/CPS.html.

[22] P.A. Boyle, G. Cossu, A. Yamaguchi, A. Portelli, Grid: a next generation data parallel C++ QCD library, PoS LATTICE2015 (2016) 023, https://doi.org/10.22323/1.251.0023.

[23] Grid: data parallel C++ mathematical object library, https://github.com/paboyle/Grid.

[24] openQCD simulation programs for lattice QCD, https://luscher.web.cern.ch/luscher/openQCD/.

[25] L. Mazur, Topological Aspects in Lattice QCD, Ph.D. thesis, Bielefeld U., 2021, https://doi.org/10.4119/unibi/2956493.

[26] `SIMULATeQCD` public code repository, https://github.com/LatticeQCD/SIMULATeQCD, https://doi.org/10.5281/zenodo.10363968.

[27] D. Bollweg, L. Altenkort, D.A. Clarke, O. Kaczmarek, L. Mazur, C. Schmidt, P. Scior, H.-T. Shu, HotQCD on multi-GPU systems, PoS LATTICE2021 (2022) 196, https://doi.org/10.22323/1.396.0196, arXiv:2111.10354.

[28] D.A. Clarke, O. Kaczmarek, F. Karsch, A. Lahiri, M. Sarkar, Sensitivity of the Polyakov loop and related observables to chiral symmetry restoration, Phys. Rev. D 103 (1) (2021) L011501, https://doi.org/10.1103/PhysRevD.103.L011501, arXiv:2008.11678.

[29] A. Bazavov, et al., Skewness, kurtosis, and the fifth and sixth order cumulants of net baryon-number distributions from lattice QCD confront high-statistics STAR data, Phys. Rev. D 101 (7) (2020) 074502, https://doi.org/10.1103/PhysRevD.101.074502, arXiv:2001.08530.

[30] D. Bollweg, J. Goswami, O. Kaczmarek, F. Karsch, S. Mukherjee, P. Petreczky, C. Schmidt, P. Scior, Second order cumulants of conserved charge fluctuations revisited: vanishing chemical potentials, Phys. Rev. D 104 (7) (2021) 074512, https://doi.org/10.1103/PhysRevD.104.074512, arXiv:2107.10011.

[31] D. Bollweg, J. Goswami, O. Kaczmarek, F. Karsch, S. Mukherjee, P. Petreczky, C. Schmidt, P. Scior, Taylor expansions and Padé approximants for cumulants of conserved charge fluctuations at nonvanishing chemical potentials, Phys. Rev. D 105 (7) (2022) 074511, https://doi.org/10.1103/PhysRevD.105.074511, arXiv:2202.09184.

[32] D. Bollweg, D.A. Clarke, J. Goswami, O. Kaczmarek, F. Karsch, S. Mukherjee, P. Petreczky, C. Schmidt, S. Sharma, Equation of state and speed of sound of (2+1)-flavor QCD in strangeness-neutral matter at non-vanishing net baryon-number density, arXiv:2212.09043.

[33] P. Dimopoulos, L. Dini, F. Di Renzo, J. Goswami, G. Nicotra, C. Schmidt, S. Singh, K. Zambello, F. Ziesché, Contribution to understanding the phase structure of strong interaction matter: Lee-Yang edge singularities from lattice QCD, Phys. Rev. D 105 (3) (2022) 034513, https://doi.org/10.1103/PhysRevD.105.034513, arXiv:2110.15933.

[34] F. Cuteri, J. Goswami, F. Karsch, A. Lahiri, M. Neumann, O. Philipsen, C. Schmidt, A. Sciarra, Toward the chiral phase transition in the Roberge-Weiss plane, Phys. Rev.

D 106 (1) (2022) 014510, https://doi.org/10.1103/PhysRevD.106.014510, arXiv:2205.12707.
[35] L. Dini, P. Hegde, F. Karsch, A. Lahiri, C. Schmidt, S. Sharma, Chiral phase transition in three-flavor QCD from lattice QCD, Phys. Rev. D 105 (3) (2022) 034510, https://doi.org/10.1103/PhysRevD.105.034510, arXiv:2111.12599.
[36] L. Altenkort, A.M. Eller, O. Kaczmarek, L. Mazur, G.D. Moore, H.-T. Shu, Heavy quark momentum diffusion from the lattice using gradient flow, Phys. Rev. D 103 (1) (2021) 014511, https://doi.org/10.1103/PhysRevD.103.014511, arXiv:2009.13553.
[37] L. Altenkort, A.M. Eller, O. Kaczmarek, L. Mazur, G.D. Moore, H.-T. Shu, Sphaleron rate from Euclidean lattice correlators: an exploration, Phys. Rev. D 103 (2021) 114513, https://doi.org/10.1103/PhysRevD.103.114513, https://link.aps.org/doi/10.1103/PhysRevD.103.114513.
[38] L. Altenkort, A.M. Eller, A. Francis, O. Kaczmarek, L. Mazur, G.D. Moore, H.-T. Shu, Viscosity of pure-glue qcd from the lattice, https://doi.org/10.48550/ARXIV.2211.08230, https://arxiv.org/abs/2211.08230.
[39] L. Altenkort, A.M. Eller, O. Kaczmarek, L. Mazur, G.D. Moore, H.T. Shu, Lattice QCD noise reduction for bosonic correlators through blocking, Phys. Rev. D 105 (2022) 094505, https://doi.org/10.1103/PhysRevD.105.094505, arXiv:2112.02282.
[40] L. Altenkort, O. Kaczmarek, R. Larsen, S. Mukherjee, P. Petreczky, H.-T. Shu, S. Stendebach, Heavy quark diffusion from 2+1 flavor lattice QCD, arXiv:2302.08501.
[41] D.A. Clarke, O. Kaczmarek, F. Karsch, A. Lahiri, Polyakov loop susceptibility and correlators in the chiral limit, PoS LATTICE2019 (2020) 194, https://doi.org/10.22323/1.363.0194, arXiv:1911.07668.
[42] G. Parkar, D. Bala, O. Kaczmarek, R. Larsen, S. Mukherjee, P. Petreczky, A. Rothkopf, J.H. Weber, Static quark anti-quark interactions at non-zero temperature from lattice QCD, EPJ Web Conf. 274 (2022) 04006, https://doi.org/10.1051/epjconf/202227404006, arXiv:2211.12937.
[43] G. Parkar, O. Kaczmarek, R. Larsen, S. Mukherjee, P. Petreczky, A. Rothkopf, J.H. Weber, Complex potential at T > 0 from fine lattices, PoS LATTICE2022 (2023) 188, https://doi.org/10.22323/1.430.0188.
[44] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, E. Teller, Equation of state calculations by fast computing machines, J. Chem. Phys. 21 (6) (1953) 1087–1092, https://doi.org/10.1063/1.1699114, http://aip.scitation.org/doi/10.1063/1.1699114.
[45] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, Numerical recipes: the art of scientific computing, 2nd edition, 1992.
[46] P. L'Ecuyer, Maximally equidistributed combined Tausworthe generators, Math. Comput. 65 (213) (1996) 203–213, https://doi.org/10.1090/S0025-5718-96-00696-5, https://www.ams.org/mcom/1996-65-213/S0025-5718-96-00696-5/.
[47] P. Hasenfratz, F. Karsch, Chemical potential on the lattice, Phys. Lett. B 125 (1983) 308–310, https://doi.org/10.1016/0370-2693(83)91290-X.
[48] N. Cabibbo, E. Marinari, A new method for updating SU(N) matrices in computer simulations of gauge theories, Phys. Lett. B 119 (4–6) (1982) 387–390, https://doi.org/10.1016/0370-2693(82)90696-7.
[49] K. Fabricius, O. Haan, Heat bath method for the twisted Eguchi-Kawai model, Phys. Lett. B 143 (4–6) (1984) 459–462, https://doi.org/10.1016/0370-2693(84)91502-8, http://linkinghub.elsevier.com/retrieve/pii/0370269384915028.
[50] A.D. Kennedy, B.J. Pendleton, Improved heatbath method for Monte Carlo calculations in lattice gauge theories, Phys. Lett. B 156 (5–6) (1985) 393–399, https://doi.org/10.1016/0370-2693(85)91632-6.
[51] S.L. Adler, Over-relaxation method for the Monte Carlo evaluation of the partition function for multiquadratic actions, Phys. Rev. D 23 (12) (1981) 2901–2904, https://doi.org/10.1103/PhysRevD.23.2901.
[52] M. Creutz, Overrelaxation and Monte Carlo simulation, Phys. Rev. D 36 (2) (1987) 515–519, https://doi.org/10.1103/PhysRevD.36.515.
[53] A.D. Kennedy, I. Horvath, S. Sint, A new exact method for dynamical fermion computations with nonlocal actions, Nucl. Phys. B, Proc. Suppl. 73 (1999) 834–836, https://doi.org/10.1016/S0920-5632(99)85217-7, arXiv:hep-lat/9809092.
[54] M.A. Clark, A.D. Kennedy, The RHMC algorithm for two flavors of dynamical staggered fermions, Nucl. Phys. B, Proc. Suppl. 129 (2004) 850–852, https://doi.org/10.1016/S0920-5632(03)02732-4, arXiv:hep-lat/0309084.
[55] S. Mukherjee, O. Kaczmarek, C. Schmidt, P. Steinbrecher, M. Wagner, HISQ inverter on Intel® Xeon Phi^{TM} and NVIDIA® GPUs, PoS LATTICE2014 (2015) 044, https://doi.org/10.22323/1.214.0044, arXiv:1409.1510.
[56] B. Jegerlehner, Krylov space solvers for shifted linear systems, arXiv:hep-lat/9612014.
[57] J.C. Sexton, D.H. Weingarten, Hamiltonian evolution for the hybrid Monte Carlo algorithm, Nucl. Phys. B 380 (1992) 665–677, https://doi.org/10.1016/0550-3213(92)90263-B.
[58] M. Hasenbusch, Speeding up the hybrid Monte Carlo algorithm for dynamical fermions, Phys. Lett. B 519 (2001) 177–182, https://doi.org/10.1016/S0370-2693(01)01102-9, arXiv:hep-lat/0107019.
[59] I. Omelyan, I. Mryglod, R. Folk, Symplectic analytically integrable decomposition algorithms: classification, derivation, and application to molecular dynamics, quantum and celestial mechanics simulations, Comput. Phys. Commun. 151 (3) (2003) 272–314, https://doi.org/10.1016/S0010-4655(02)00754-3, https://www.sciencedirect.com/science/article/pii/S0010465502007543.
[60] A. Bazavov, et al., Scaling studies of QCD with the dynamical HISQ action, Phys. Rev. D 82 (2010) 074501, https://doi.org/10.1103/PhysRevD.82.074501, arXiv:1004.0342.
[61] S.O. Bilson-Thompson, D.B. Leinweber, A.G. Williams, Highly improved lattice field-strength tensor, Ann. Phys. 304 (1) (2003) 1–21, https://doi.org/10.1016/S0003-4916(03)00009-5, https://linkinghub.elsevier.com/retrieve/pii/S0003491603000095.
[62] J.E. Mandula, M. Ogilvie, Efficient gauge fixing via overrelaxation, Phys. Lett. B 248 (1–2) (1990) 156–158, https://doi.org/10.1016/0370-2693(90)90031-Z.
[63] M. Lüscher, Properties and uses of the Wilson flow in lattice QCD, J. High Energy Phys. 08 (2010) 071, https://doi.org/10.1007/JHEP08(2010)071, arXiv:1006.4518, Erratum: J. High Energy Phys. 03 (2014) 092.
[64] A. Ramos, S. Sint, Symanzik improvement of the gradient flow in lattice gauge theories, Eur. Phys. J. C 76 (1) (2016) 15, https://doi.org/10.1140/epjc/s10052-015-3831-9, arXiv:1508.05552.
[65] H. Munthe-Kaas, Runge-Kutta methods on Lie groups, BIT Numer. Math. 38 (1) (1998) 92–111, https://doi.org/10.1007/BF02510919, http://link.springer.com/10.1007/BF02510919.
[66] E. Celledoni, A. Marthinsen, B. Owren, Commutator-free Lie group methods, Future Gener. Comput. Syst. 19 (3) (2003) 341–352, https://doi.org/10.1016/S0167-739X(02)00161-9, https://linkinghub.elsevier.com/retrieve/pii/S0167739X02001619.
[67] A. Hasenfratz, F. Knechtli, Flavor symmetry and the static potential with hypercubic blocking, Phys. Rev. D 64 (2001) 034504, https://doi.org/10.1103/PhysRevD.64.034504, arXiv:hep-lat/0103029.
[68] M. Luscher, P. Weisz, Locality and exponential error reduction in numerical lattice gauge theory, J. High Energy Phys. 09 (2001) 010, https://doi.org/10.1088/1126-6708/2001/09/010, arXiv:hep-lat/0108014.
[69] H.B. Meyer, Locality and statistical error reduction on correlation functions, J. High Energy Phys. 01 (2003) 048, https://doi.org/10.1088/1126-6708/2003/01/048, arXiv:hep-lat/0209145.
[70] T. Veldhuizen, Expression templates, C++ Rep. 7 (5) (1995) 26–31, https://web.archive.org/web/20050210090012/, http://osl.iu.edu/~tveldhui/papers/Expression-Templates/exprtmpl.html.
[71] M.G. Beckett, P. Coddington, B. Joó, C.M. Maynard, D. Pleiter, O. Tatebe, T. Yoshie, Building the international lattice data grid, Comput. Phys. Commun. 182 (6) (2011) 1208–1214, https://doi.org/10.1016/j.cpc.2011.01.027, https://linkinghub.elsevier.com/retrieve/pii/S0010465511000476.
[72] C. Maynard, D. Pleiter, QCDml: first milestones for building an international lattice data grid, Nucl. Phys. B, Proc. Suppl. 140 (2005) 213–221, https://doi.org/10.1016/j.nuclphysbps.2004.11.116, https://linkinghub.elsevier.com/retrieve/pii/S0920563204006814.