Pushing Memory Bandwidth Limitations Through Efficient Implementations of Block-Krylov Space Solvers on GPUs

M. A. Clark
NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95054, USA

Alexei Strelchenko
Fermi National Accelerator Laboratory, Batavia, IL 60510-5011, USA

Alejandro Vaquero
Department of Physics and Astronomy, University of Utah, Salt Lake City, UT 84112, USA

Mathias Wagner∗
NVIDIA GmbH, Adenauer Str. 20A4, 52146 Würselen, Germany

Evan Weinberg
Department of Physics, Boston University, Boston, MA 02215, USA

Abstract
Lattice quantum chromodynamics simulations in nuclear physics have benefited from a tremendous number of algorithmic advances such as multigrid and eigenvector deflation. These improve the time to solution but do not alleviate the intrinsic memory-bandwidth constraints of the matrix-vector operation dominating iterative solvers. Batching this operation for multiple vectors and exploiting cache and register blocking can yield a super-linear speed up. Block-Krylov solvers can naturally take advantage of such batched matrix-vector operations, further reducing the iterations to solution by sharing the Krylov space between solves. However, practical implementations typically suffer from the quadratic scaling in the number of vector-vector operations. Using the QUDA library, we present an implementation of a block-CG solver on NVIDIA GPUs which reduces the memory-bandwidth complexity of vector-vector operations from quadratic to linear. We present results for the HISQ discretization, showing a 5× speedup compared to highly-optimized independent Krylov solves on NVIDIA’s SaturnV cluster.

Keywords: Block solver, GPU
PACS: 12.38.Gc, 02.60.Dc, 02.70.-c

1. Introduction
Two trends in HPC architectures present particular challenges to scientific software. The wider and wider architectures, in particular GPUs and Intel Xeon Phi processors, require an increasing amount of parallelism that cannot always be extracted from algorithms that had been considered state-of-the-art only a few years ago. Furthermore, the available memory bandwidth presents itself as the limiting factor in most scientific calculations. Hence, in addition to extracting parallelism exploiting data locality is key to achieve high performance. This has triggered a lot of algorithmic research, and the revisiting of older ideas can provide algorithmic speedup and high efficiency on current architectures even if these ideas had been discarded as less efficient decades ago.

In this paper we will focus on lattice quantum chromodynamics (LQCD), the numerical simulations of the theory of the strong force that underpins nuclear and particle physics. In the past few years, LQCD applications have been continually under intensive development to address computationally demanding scientific problems in nuclear and high energy physics. These algorithmic improvements and the advances in computing architectures now allow predictions with errors comparable or even below experimental results obtained at large particle colliders like the LHC at CERN. To control the statistical error, a typical computational scenario requires the solution of many

∗Corresponding author
Email addresses: mclark@nvidia.com (M. A. Clark), astrel@fnal.gov (Alexei Strelchenko), alexvaq@physics.utah.edu (Alejandro Vaquero), mathiasw@nvidia.com (Mathias Wagner), weinbe2@bu.edu (Evan Weinberg)
systems of linear equations with the same matrix but different right hand sides (rhs). One example of this is the calculation of the fermion magnetic moment, which can be computed within the standard model to high accuracy. The Dirac equation predicts \( \mu = \frac{ge}{(2m)S} \), where \( \mu \) is the magnetic moment, \( e \) is the fermion electric charge, \( m \) is its mass, \( S \) is the spin and \( g \) is the Landé-\( g \)-factor. For free fermions \( g = 2 \), but when we introduce electro-weak and strong force interactions, \( g \) receives perturbative and non-perturbative corrections, which are gathered in the anomalous magnetic moment, defined as \( a_\mu = \frac{\mu - \frac{e\gamma^5}{2}}{\frac{e\gamma^5}{2}} \), which measures how far we are from the free fermion case. The particular case of the muon is extremely interesting, because there is a 3\( \sigma \) difference between the theoretical and experimental determinations of \( a_\mu \). It is of the utmost importance to improve the precision of both determinations, for this difference can be an indicative of new fundamental physics. The largest part of the error in the theoretical determination of \( a_\mu \) comes from non-perturbative QCD contributions. The LQCD calculation of this contribution is essentially computing the trace of the inverse of a large sparse matrix, the exact calculation of which is unfeasible. Therefore the calculation relies on stochastic estimators \([1]\), involving solving thousands of rhs per linear system for many different linear systems. The result is that solving linear systems accounts for \( \sim 90\% \) of the total computational time.

There are multiple approaches to reduce this time: the low eigenvectors of the linear system can be computed and projected out, deflating the linear system, and accelerating the convergence rate, see for example \([2, 3]\). While deflation is effective, its applicability can be limited to smaller problem sizes given that the number of eigenvectors required for effective deflation scales with the problem size. Another method is adaptive multigrid \([4, 5]\) and the related inexact deflation \([6]\), which are mathematically optimal (no condition number dependence and constant iteration count with volume). However, multigrid is not applicable to all linear systems in lattice QCD (although this is a very active area of research).

A third approach, and the one taken in this work, is the Block-Krylov solver method. This method reduces the iteration count by processing blocks of right hand sides simultaneously, augmenting the Krylov space of each rhs by information from the other rhs in the block. Unlike deflation and multigrid methods, there is no setup phase that needs to be amortized. The use of blocks not only naturally generates more parallelism but also allows the exploitation of data locality by simultaneous application of a matrix-vector operator and specialized linear algebra routines. This not only makes Block-Krylov methods very suitable algorithms for current and future HPC systems, but also give a multiplicative speedup from the implementation in addition to the algorithmic speedup if implemented efficiently.

GPUs provide the perfect setting for demonstrating the block-solver algorithm. Present GPUs typically feature thousands of floating point units coupled to a very wide and fast memory bus, and, as common to many-core processors, a high algorithmic arithmetic intensity is required to reach a reasonable percentage of peak performance. In this work we utilize the QUDA library \([7]\) to implement a high-efficiency block CG solver for LQCD that is able to utilize the growing imbalance between floating-point peak performance and memory bandwidth. This work is cross cutting since this is a problem that affects HPC in general and the exploration of algorithms that can continue to achieve meaningful speedup as we trend towards the exascale and beyond will only continue to have increased relevance.

Our implementation of a block CG solver is, to the best of our knowledge, the first efficient implementation\(^1\) of a reliable block solver at scale on GPUs. Previous implementations either ran on CPU \([8, 9]\) or didn’t provide scaling results \([10, 11, 12]\). We take advantage of mixed precision to mitigate memory-bandwidth limitations and – as a novel development – implement a first generalization of the idea of reliable updates \([13]\) to block solvers, a necessary step to correct for round-off errors due to the use of mixed precision. We employ block BLAS operations to saturate memory bandwidth and exploit data locality for both streaming and reduction BLAS operations, allowing us to take advantage of the extra compute power offered by modern nodes. Last, we achieve good strong scaling both by an improved implementation of a multi-right-hand-side-stencil operation and by a new interface which allows us to overlap communications inherent to stencil operations with independent computation.

This paper is organized as follows: in \(\S2\) we highlight previous work in this area, in \(\S3\) we give an overview of the LQCD computations, and we give an overview of the block CG solver in \(\S4\). We introduce the QUDA library in \(\S5\) and describe in the detail the optimization techniques employed in our block CG implementation \(\S6\). We show weak scaling performance curves from one and two NVIDIA Quadro GP100 GPUs, and more significantly, strong-scaling performance curves from the NVIDIA SaturnV DGX-1 supercomputer in \(\S7\); we discuss future implications in \(\S8\) before finally concluding with \(\S9\).

\(\S2\). Previous Work

The first use of GPUs for LQCD was reported more than a decade ago \([14]\), before the advent of compute APIs, necessitating the use of graphics APIs. Since then LQCD has been at the forefront of the adoption of GPUs in HPC. Notable publications include multi-GPU parallelization \([15, 16, 17]\), the use of additive-Schwarz preconditioning to improve strong scaling \([18]\), implementation of multi-grid solvers \([19]\), software-managed cache-blocking strategies \([20]\), and JIT-compilation to enable the

---

\(^1\)Efficient in the sense that we obtain a non-negligible speedup compared to a highly optimized baseline on our target architecture.
offload of the entire underlying data-parallel framework of the Chroma [21] application without any high-level source changes [22].

Block-Krylov-space methods go back to the 1980s [23] for CG and Bi-CG solvers. By solving multiple rhs in parallel they combine the benefit of increased parallelism and data locality with an reduced iteration count compared to a non-block method. The naive implementation was found to suffer from stability issues and the subsequent revisions which introduce explicit re-orthogonalization [24] have significantly improved their general usability. However, the stability improvements come at the cost of implementation efficiency and can become prohibitively expensive outweighing any benefit over non-block solvers. In LQCD, block methods have not been widely adopted. The use of block BiCGSTAB has been reported in [25, 26, 27]. Birk and Frommer generalized block methods also for the use in shifted systems [28, 29], which dominate the runtime in Monte Carlo ensemble generation in LQCD.

To our best knowledge block-Krylov methods have not been reported to be efficiently implemented for LQCD on GPUs. The benefit of using the block idea for the matrix-vector operation to multiple rhs has been reported on GPU and Intel Xeon Phi [30, 31], however there it was combined with a regular CG solver resulting in only a pseudo-block algorithm.

3. Lattice Quantum Chromodynamics

3.1. Computational Method

LQCD calculations are typically Monte-Carlo evaluations of Euclidean-time path integrals. A sequence of configurations of the gauge fields $U$ is generated in a process known as configuration generation. The gauge configurations are importance-sampled with respect to the lattice action and represent a snapshot of the QCD vacuum. Once the field configurations have been generated, one moves on to the second stage of the calculation, known as analysis. In this phase, observables of interest, e.g., $g - 2$ in the present case, are evaluated on the gauge configurations in the ensemble, and the results are then averaged appropriately, to form ensemble averaged quantities. The analysis phase can be task parallelized over the available configurations in an ensemble and is thus extremely suitable for capacity-level work on clusters, though calculations on the largest ensembles can also make highly effective use of capability-sized partitions of leadership supercomputers. These calculations rely heavily on computational resources, with LQCD workloads accounting for a significant fraction of supercomputing cycles consumed worldwide.

3.2. Dirac PDE Discretization

The fundamental interactions of QCD are encoded in the quark-gluon interaction differential operator known as the Dirac operator. As is common in PDE solvers, the derivatives are replaced by finite differences. Thus on the lattice, the Dirac operator becomes a large sparse matrix, $M$, and the calculation of quark physics is essentially reduced to many solutions to systems of linear equations given by

$$Mx = b.$$ \hspace{1cm} (1)

A small handful of discretizations are in common use, differing in their theoretical properties. Here we focus on the so-called staggered form [32], which is a central-difference discretization; where the infamous fermion doublers (which arise due to the instability in the central difference approximation) are removed through “staggering” the underlying fermionic (spin) degrees of freedom onto neighboring lattice sites. This essentially reduces the number of spin degrees of freedom per site from four to one, which reduces the computational burden significantly. This transformation, however, comes at the expense of increased discretization errors, and breaks the so-called quark-flavor symmetry. To reduce these discretization errors, the gauge field that connects nearest neighboring sites on the lattice is smeared, which essentially is a local averaging of the field. There are many prescriptions for this averaging; and here we employ the popular HISQ procedure [33]. When acting in a vector space that is the tensor product of a four-dimensional discretized Euclidean space-time (the lattice) and color space, the HISQ stencil operator is given by

$$M_{x,x'} = -\frac{1}{2} \sum_{\mu=1}^{4} (\tilde{U}_x^\mu \delta_{x+\hat{\mu},x'} + \tilde{U}_x^{\mu \dagger} \delta_{x-\hat{\mu},x'} + \tilde{U}_x^{\mu \dagger} \delta_{x+3\hat{\mu},x'} + \tilde{U}_x^{\mu \dagger} \delta_{x-3\hat{\mu},x'}) + m\delta_{x,x'}.$$

(2)

Here $\delta_{x,y}$ is the Kronecker delta; $\tilde{U}$ (\(\tilde{U}\)) is the “fat” (“long”) matrix field that connects lattice sites that are one hop (three hops) apart. These are fields of 3 x 3 matrices acting in color space that live between the spacetime sites (and hence are referred to as link matrices); and $m$ is the quark mass parameter. The indices $x$ and $x'$ are spacetime indices (the color indices have been suppressed for brevity). This matrix acts on a vector consisting of a complex-valued three-component color-vector for each point in spacetime. When represented as a sparse matrix, there are 16 off-diagonal matrices, representing the 1-hop and 3-hop link matrices. Given the stencil coefficients are dependent on the underlying gauge field, the coefficients vary for each configuration in the Monte Carlo ensemble.

For the current problem sizes of interest, the linear system has a rank that ranges from $V \approx 10^4$ for a small lattice volume (32$^4$) up to $V \approx 10^{10}$ for the largest (144$^3 \times 288$). Furthermore, the linear system is typically very ill conditioned: the highest eigenvalue is of order $O(1)$, but the lowest one can be as low as $O(1/V)$ (dependent on the quark mass parameter). The standard workhorse algorithm employed is the CG algorithm, where the normal operator $A = M^T M$ is used since $M$ itself is not Hermi-
4. Block CG Solver

The block-CG algorithm was first published in [23]. The idea is to apply the CG solver [34] to a block of $N$ rhs. In a broad sense, scalars in the CG algorithm become $N \times N$ matrices and for $N = 1$ the algorithm reduces to the regular CG.

In infinite precision the block-CG algorithm will converge in at most $L/N$ iterations where for the $L$ denotes the length of the vectors, or the rank of the matrix $M$. While the algorithm maintains a simple link to regular CG, it has issues with numerical stability and will break down if the residual matrix $(R^\dagger R)$ becomes singular. There have been various attempts to remedy this, including dynamically monitoring the rank and removing any linear dependent columns [35]. A more suitable approach was introduced in [24], where a set of retooled block-CG methods are discussed. The common idea is to use a QR decomposition in [24], where a set of retooled block-CG methods are discussed. The common idea is to use a QR decomposition to enforce the full rank of the $P$ matrices and for $N = 1$ the algorithm reduces to the regular CG.

An important observation is that any block algorithm will naively show quadratic scaling in the number of BLAS operations from CG become BLAS-3 operations. More specifically, vector-vector operations and reductions. E.g., the BLAS-1 operations from CG become BLAS-3 operations. Moreover, the factorization required for stability increases the cost per iteration. We will address these concerns when we discuss our implementation in §6.2.

An important observation is that any block algorithm will naively show quadratic scaling in the number of BLAS-1 operations from CG become BLAS-3 operations. More specifically, vector-vector operations and reductions. E.g., the BLAS-1 operations from CG become BLAS-3 operations. Moreover, the factorization required for stability increases the cost per iteration. We will address these concerns when we discuss our implementation in §6.2.

5. The QUDA Library

QUDA (QCD on CUDA) is a library that aims to accelerate LQCD computations through offload of the most time-consuming components of an LQCD application to NVIDIA GPUs. It is a package of optimized CUDA C++ kernels and wrapper code, providing a variety of optimized linear solvers, as well as other performance critical routines required for LQCD calculations. All algorithms can be run distributed on a cluster of GPUs, using MPI to facilitate inter-GPU communication. On systems with multiple GPUs that are directly connected with either NVLink or PCIe, QUDA will take full advantage utilizing direct DMA copies between GPUs to minimize the communication overhead. Similarly, on systems that support GPU Direct RDMA, where the NIC can read/write directly to the GPU’s memory space, QUDA will utilize this to improve multi-node scaling. It has been designed to be easy to interface to existing code bases, and in this work we exploit this interface to use the popular LQCD application MILC [36] as a driver. The QUDA library has attracted a diverse developer community and is being used in production at U.S. national laboratories, as well as in locations in Europe and India. The latest development version is always available in a publicly-accessible source code repository [37].

The general strategy is to assign a single GPU thread to each lattice site. Each thread is then responsible for all memory traffic and operations required to update that site on the lattice given the stencil operator. Since the computation always takes place on a regular grid, a matrix-free approach is used. Maximum memory bandwidth is obtained by reordering the vector and link fields to achieve memory coalescing, e.g., using structures of float2 or float4 arrays, and using the texture cache where appropriate. Memory-traffic reduction is employed where possible to overcome the relatively low arithmetic intensity of the Dirac matrix-vector operations, which would otherwise limit performance. The primary strategy employed here is the use of low-precision data storage, utilizing single precision or a custom 16-bit fixed-point storage format (hereon referred to as “half precision”) together with mixed-precision linear solvers to achieve high speed with no loss in accuracy [1].

The library has been designed to allow for maximum flexibility with respect to algorithm parameter space and maximum performance. For example, all lattice objects (fields) maintain their own precision and data ordering as a dynamic variable. This allows for run-time policy tuning of algorithms; these parameters are then bound at kernel launch time when the appropriate C++ template is instantiated corresponding to these parameters.

In fact the use of this autotuning is a key to achieving high performance: all kernel launch parameters (block, grid, and shared memory size) are autotuned upon the first call and cached for subsequent reuse. In the work presented here, we extended the autotuner extensively as an algorithm-policy tuner, discussed in §6.2 and §6.4.

---

An important observation is that any block algorithm will naively show quadratic scaling in the number of BLAS-1 operations from CG become BLAS-3 operations. Moreover, the factorization required for stability increases the cost per iteration. We will address these concerns when we discuss our implementation in §6.2.

---

An important observation is that any block algorithm will naively show quadratic scaling in the number of BLAS-1 operations from CG become BLAS-3 operations. Moreover, the factorization required for stability increases the cost per iteration. We will address these concerns when we discuss our implementation in §6.2.

---

An important observation is that any block algorithm will naively show quadratic scaling in the number of BLAS-1 operations from CG become BLAS-3 operations. Moreover, the factorization required for stability increases the cost per iteration. We will address these concerns when we discuss our implementation in §6.2.

---

An important observation is that any block algorithm will naively show quadratic scaling in the number of BLAS-1 operations from CG become BLAS-3 operations. Moreover, the factorization required for stability increases the cost per iteration. We will address these concerns when we discuss our implementation in §6.2.

---

An important observation is that any block algorithm will naively show quadratic scaling in the number of BLAS-1 operations from CG become BLAS-3 operations. Moreover, the factorization required for stability increases the cost per iteration. We will address these concerns when we discuss our implementation in §6.2.
6. Efficient implementation of BlockCGrQ in QUDA

Our implementation of blockCGrQ integrates many developments to overcome the issues discussed in [1] and [11]. We will address these as we step through algorithm [1] and discuss the details of our efficient implementation. We may expect a reduced time to solution from blockCGrQ due to the reduced iteration count. An additional benefit can be achieved by applying the matrix-vector operation to multiple rhs in parallel, exploiting data locality in this operation; see [6.2]. This is countered, however, by the quadratic increase in the number of BLAS-1 vector-vector operations, which generalize to BLAS-3 matrix-matrix operations, as well as an additional cost due to the factorization employed on lines [4] and [11].

We make use of a thin QR factorization [38]. By strict mathematical definition, the QR factorization of a rectangular matrix would be given as \( \tilde{Q}L \times L \tilde{R}L \times N = M^{L \times N} \), where \( \tilde{Q} \) is dense with orthonormal columns and \( \tilde{R} \) is upper right triangular. A dense matrix of dimension \( L \times L \) is prohibitive in LQCD as \( L \) is typically of \( O(10^{5}-9) \). However, the matrix \( \tilde{R} \), being upper right triangular, has zeros in the bottom \((L - N) \times N \) components. Thus, the right \((L - N) \times N \) columns of \( \tilde{Q} \) are irrelevant and can be dropped. This leads to the thin QR factorization given in algorithm [2], which of note contains structured BLAS operations.

Algorithm 2 thin QR: Decompose \( Q^{L \times N}R^{N \times N} = M^{L \times N} \)

```plaintext
1: procedure THINQR(M^{L \times N})
2: \( H^{N \times N} = (M^{L \times N})^{T}M^{L \times N} \)
3: \( (R^{N \times N})^{T}R^{N \times N} = H^{N \times N} \) \( \triangleright \) Cholesky decomp
4: \( Q^{L \times N} = M^{L \times N}(R^{N \times N})^{-1} \)
5: end procedure
```

With this exposition, we can now decompose the operations in our implementation of algorithm [1] into four classes: matrix-vector operations in lines [3] and [9], vector-vector streaming BLAS operations in lines [8] and [10], vector-vector reduction BLAS operations in lines [3] and [9], and small dense matrix operations. The latter three kinds of operations are also embedded in the thin QR decompositions.

A key ingredient for the implementation is generalizing our lattice fields from a single vector of length \( L \) to an efficient data structure of size \( L \times N \). We refer to these \( L \times N \) matrices as composite fields with details discussed in [6.1]. We will discuss how to exploit data locality to reuse \( A \) in the matrix-vector operation to obtain a speedup per rhs in [6.2] and mitigate the naïve quadratic scaling of the vector-vector BLAS operations to achieve an overall multiplicative speedup in [6.3] 6.4.

For all small \( N \times N \) matrices we exploit the Eigen library [39] for the Cholesky decomposition of \( R \). Given that \( N \leq 64 \), these matrices are comparably small and all operations are negligible in the overall runtime.

**Mixed-precision implementation**

For our mixed-precision implementation we rely on “reliable updates”, referring to a class of residual correction methods described, e.g., in [13, 40]. A general residual correction method periodically updates the iterated residual with the true residual from the iterated solution. This corrects for numerical round-off effects and, by extension, possible instability in Krylov methods. In the case of mixed-precision solves, this correction is often essential to monitor convergence in a meaningful fashion. We based our implementation of reliable updates on the implementation given in [7], with important deviations presented in algorithm [3]. We believe that this is the first implementation of a residual correction method to improve the convergence of a block-Krylov method.

Algorithm 3 Reliable updates

```plaintext
1: procedure RELIABLEUPDATE(X^{L \times N}, \tilde{Q}^{L \times N}, S^{N \times N})
2: \( Y^{L \times N} = Y^{L \times N} + X^{L \times N} \) \( \triangleright \) caxpy 5d
3: \( X^{L \times N} = 0 \)
4: \( R^{L \times N} = B^{L \times N} - A^{L \times L}Y^{L \times N} \) \( \triangleright \) Block Mat-Vec, caxpy 5d
5: \( \tilde{Q}^{L \times N}C^{N \times N} = R^{L \times N} \)
6: \( S^{N \times N} = C^{N \times N}(C_{dd}^{N \times N})^{-1} \)
7: end procedure
```

Like in [7], we employ a persistent fine precision accumulator for the iterated solution, denoted by the block vector \( Y^{L \times N} \). The block vector \( X^{L \times N} \) only contains the iterated update to the solution between reliable updates, which, in contrast to [7], is maintained at full precision. The block vectors \( P^{L \times N} \) and \( Q^{L \times N} \) are maintained in lower precision. We perform a reliable update when the maximum relative iterated residual decreases by a factor of \( \delta \). We use the relative residual as opposed to the absolute residual to perform a meaningful comparison between the different rhs vectors. The iterated residual can be computed after \( C^{N \times N} \) is updated on line [12] of algorithm [1] by the identity \( |\tilde{R}|_{L \times N}^{2} = \sum_{j}|C_{ij}^{N \times N}|^{2} \).

When a reliable update is triggered, the current iterated solution update \( X^{L \times N} \) is accumulated into full solution \( Y^{L \times N} \) and the true residual is computed in full precision, as implemented in [7]. As required by blockCGrQ, we perform a thin QR factorization of the true residual. The update of \( S^{N \times N} \) on line [6] of algorithm [3] also forces the preservation of the identity given in equation (6.1) of [24]. The new \( S^{N \times N} \) enters the update of \( P^{N \times N} \) on line [8] of algorithm [1] as well. This is essential to maintain one of the defining properties of blockCGrQ, \( (\tilde{Q}_{dd}^{L \times N})^{T}P^{L \times N} = S^{N \times N}, \) noted below figure 6.1 of [24], which is intimately connected to preserving the Krylov search space through the reliable update.
6.1. Composite Fields

In order to provide both ease of expressibility and high efficiency for the block-solver implementation, we extended QUDA to facilitate the application of the required linear algebra components on both sets of vector fields and individual vector fields. To this end we enhanced QUDA’s vector-field container ColorSpinorField, with a new attribute: whether it is a composite field or not. When it is a composite field, the container contains an STL vector of component ColorSpinorFields, whose memory refers to contiguous subsets of the parent composite. The composite field can thus be viewed as a single higher-dimensional object while providing accessor methods to expose individual components or the entire STL vector. This functionality of being able to switch between block and scalar methods while using the same data structure was critical in algorithm prototyping, debugging, and performance analysis.

6.2. Block-optimized Matrix-vector Product

The efficient application of the stencil operator, equation \ref{eq:blockStencil}, is critical in a performance-optimized iterative linear solver. The stencil connects lattice sites that are one and three hops away in all dimensions in both positive and negative directions relative to the center site. To apply the stencil, we load the neighboring sites (6 numbers per site), the link matrix between the neighboring sites (18 numbers per matrix), perform the link-matrix times vector multiplication and accumulate to the new resulting center value. With 16 neighbors, this means each application of the operator requires 1158 flops and 390 words of memory traffic per lattice site, for an arithmetic intensity of 0.74 in single precision. The Pascal-generation GPUs used in this study can achieve a STREAM bandwidth of 575 GB/s, giving a naive performance upper bound of 435 GFLOPS in single precision, compared to a peak of 10.1 TGFLOPS, so we can see that any naive implementation of the stencil would be memory-bandwidth bound.

In this specific algorithm, we apply this stencil to multiple vector fields simultaneously, and as such we have implemented a block-optimized stencil operator which allows us to severely reduce the contribution of link matrices to the overall memory traffic. In the usual single-rhs stencil operator, a volume of threads are launched, each assigned to a grid point. Threads are partitioned into a grid of thread blocks, where each thread block runs on a given streaming multiprocessor (SM), roughly equivalent to a CPU core. This provides enough thread-level parallelism to saturate the GPU memory bus.

When augmenting the GPU kernel to apply the stencil to multiple sites, we utilize the Cartesian thread indexing that GPUs expose: we map the y-dimension thread to the rhs index, keeping the x-dimension thread mapping to grid points unchanged. Since all threads in the same thread block share the same L1 cache, threads with a common grid index in the same thread block (e.g., different rhs index) are able to exploit temporal locality to share the link matrix load. The optimal x-y shape of the thread block is autotuned: this effectively optimizes for the competing localities (link-matrix element reuse versus vector-field neighbor reuse) and L1 vs L2 reuse (larger blocks will result in more L1 cache hits versus L2 cache hits).

In Figure 1 we plot the performance results from the block-optimized stencil. The solid lines represent the performance of the cache-tiled implementation, the dashed lines the cache-and-register-tiling implementation and the dotted line is a roofline model of the performance. Tests were performed on a Quadro GP100 GPU.

![Figure 1: The performance in GFLOPS of the stencil operator as a function of number of rhs vectors. The solid lines represent the cache-tiling implementation, the dashed lines the cache-and-register-tiling implementation and the dotted line is a roofline model of the performance. Tests were performed on a Quadro GP100 GPU.](image-url)
The multi-GPU parallelization of stencil operators in QUDA has been reported elsewhere [15,18]. The essential strategy is to overlap the communication of the halo region with the computation on the interior, with a final halo update kernel required to produce the complete answer. The ability to hide the communication scales with the surface-to-volume ratio, hence strong scaling is often communication limited. To maximize performance we made use of two key technologies:

- Peer-to-peer communication: direct communication between GPUs within the same node
- GPU Direct RDMA: direct communication between the GPUs and the NICs maximizing the inter-node communication rate

On a sufficiently distributed system, the communication of the halo region can take much longer than the computation on the interior. To help mitigate this, we have implemented an interface to the stencil kernel, such that an algorithm can schedule additional computation during the halo exchange that is independent of the stencil application. This is an important optimization for a wide class of algorithms, but in particular for the block-Krylov application. This is an important optimization for a wide class of algorithms, but in particular for the block-Krylov

6.3. Streaming Multi-BLAS

For the block linear algebra operations we distinguish between block-AXPY-type operations and block-DOT-type operations. They differ conceptually, and in implementation, since the latter require global reductions. We refer to the former as Streaming Multi-BLAS, which we focus here, and the latter as Reduction Multi-BLAS, which we delay until 6.4.

For the discussion here we chose a casaxy operation as the representative example. In blockCGGrQ it is used in the form $Y^{L\times N} = Y^{L\times N} + X^{L\times M} a^{M\times N}$, or at the element level,

$$y_{k,j} = y_{k,j} + \sum_{i=0}^{N-1} x_{k,i} a_{i,j}.$$  

Note that in the case of blockCGGrQ we only need the case $N = M$, and as such $a^{N\times N}$ is a square matrix. We will assume $M = N$, the number of right hand sides, unless noted otherwise.

Equation 3 can be implemented as separate casaxy applications for each component of $a^{N\times N}$. This implementation incurs a quadratic scaling of memory traffic and floating-point operations with $N$. Due to the low arithmetic intensity, the operation is severely limited by memory bandwidth and becomes prohibitively expensive as the number of right hand sides increases. A block casaxy takes advantage of the memory reuse inherent in equation 3 by re-expressing the set of vector-vector operations as a single inner product. For a given component $k$, block casaxy loads all $N$ components of $X^{L\times N}$ and $Y^{L\times N}$, performs the $N^2$ multiplications and additions, then saves all $N$ components of $Y^{L\times N}$ back to memory. The quadratic scaling of memory traffic becomes a linear scaling. Since the required flops remains well below the theoretical peak of modern GPUs, the quadratic cost of the algorithm shifts to lower latency cache load instructions which can take advantage of the spatial locality inherent to cache.

We have developed a general framework to perform such block-BLAS operations. We distribute the update of a single block vector constituent $k$ over $N$ threads on the GPU by assigning a unique constituent $y_{k,j}$ to each thread. Each thread is tasked with updating $y_{k,j}$ with the contributions from each $x_{k,i}$. Similar to the block-matrix-vector product in [6,2] we use CUDA thread blocks to ensure that for a given $k$, all $j = 0, \ldots, N$ are scheduled to the same SM. This guarantees that all $x_{k,j}$ reside in L1 cache and can be shared between all $N$ threads, exploiting spatial locality.

We take extensive advantage of C++ templating which allows the compiler to perform additional optimizations. We template over all possible combinations of the block sizes of $N$ and $M$ up to a sufficient maximum, enabling the compiler to perform loop unrolling. In addition, we template over all possible precisions for $L^{L\times M}$ and all equal or lower precisions for $X^{L\times N}$. This optimization is essential for efficient mixed-precision block-Krylov methods. Furthermore we take advantage of QUDA’s autotuner in the block-BLAS framework. Finally, we note that while the block casaxy is essentially a CGEMM operation included with every standard BLAS library, the disparity between the sizes of $N$ and $L$ ($L \gg N$) are such that no out-of-the-box BLAS library would perform well for this use case, requiring us to implement our own framework to this end.

Figure 2 shows that the cost of the block-casaxy operation scales linearly for a wide range of the number of rhs. The eventual deviation from linear scaling is due to the less-penalizing cache traffic. This is consistent with figure 3 where we see that the performance of the block-casaxy operations also scales linearly in $N$. Note that due to finite cache sizes there is a turnover point in $N$ where
it is advantageous to recursively divide the block BLAS into tiles, e.g., to split a $24 \times 24$ block operation into four $12 \times 12$ block operations. We refer to a recursed segment of the BLAS operation as a “tile”, and its size as the “tile size”.

There are two places in algorithm 1, lines 4 and 8, where we can take advantage of recursively divided block BLAS operations and perform additional optimizations. In each case, we take advantage of the triangular structure of the small, dense matrix $A$ and skip computing empty tiles. The benefit of this optimization scales quadratically in $N$ over tilesize. However, we observe that this optimization is only beneficial simultaneous with when tiling itself becomes beneficial.

6.4. Reduction Multi-BLAS

For the discussion of block reductions, we will consider the representative case of a complex dot product, or $cDotProduct$, $K^{N\times M} = (X^{L\times N})^\dagger Y^{L\times M}$. At the component level it is defined as

$$k_{ij} = \sum_{k=0}^{L-1} x_{k,i}^* y_{k,j}. \quad (4)$$

In the following we again limit our discussion to the case $N = M$ and assume that the resulting matrix $K^{N\times N}$ is Hermitian, which is always true in blockCGrQ.

At a basic level, our software implementation of block reduction operations is similar to the implementation of Streaming Multi-BLAS operations. The goal is again to achieve an implementation of Reduction Multi-BLAS operations that obeys a linear scaling in memory traffic as opposed to a quadratic scaling. We template over both the left and right block vectors, allowing the compiler to generate efficient code for every combination of left and right block sizes and apply the autotuner separately to each combination. Each thread is responsible for the multiplication of a single component $y_{k,j}$ with every component of $x_{k,i}$. We utilize the CUB library for an efficient thread block and inter-block reduction per site. We still recursively divide block reduction operations and take advantage of the known structure of the resulting dense matrices. Finally, on distributed systems, we only perform the global MPI reduction once all inner tiles have completed. We emphasize that an efficient implementation of Reduction Multi-BLAS requires a more careful recursion strategy when tiling.

We see in figure 2 that reductions scale linearly in cost for a much narrower regime of $N$ than the block $caxpy$ operation. This is consistent with what is seen in figure 3. Block reductions quickly saturate cache bandwidth as a function of $N$ and tiling thus becomes essential even for modest numbers of right hand sides. On top of tuning the internal block reduction, we tune the tile size for recursive reductions to be mindful of two different potential optimizations via a policy class. First, we perform distinct tunings for when the left and right block vectors are equal versus when they are not equal. When the two block vectors are equal, the subset of reductions on the “block-diagonal” inner tiles feature memory-load reuse. Second, we perform distinct tunings for when the output dense matrix is Hermitian versus when it is not Hermitian. In blockCGrQ, the output dense matrices are always Hermitian, thus we only perform a block reduction on inner tiles (without loss of generality) above the block diagonal. We then reconstruct the inner tiles below the block diagonal by taking advantage of the overall Hermiticity of the matrix. Smaller tile sizes further reduce the total number of redundant calculations. For this reason, it may be beneficial to use a smaller tile size despite an overall increase in memory streaming. This is automatically optimized by policy-class tuning.
from the block-matrix-vector operation and the reduced iteration count from blockCGrQ.

7. Results

7.1. Methodology

To test the algorithm in the context of the target measurement of the disconnected part of the contribution to the muon $g - 2$, we used development versions of the MILC code and QUDA. We compiled using GCC v4.8.5 and the CUDA-toolkit 8.0. The Small and Medium data sets were processed on a workstation equipped with two Quadro GP100 GPUs using MVAPICH2 2.2. The Large and X-Large data sets were run on the NVIDIA SaturnV cluster: each node consists of a DGX-1 server, with eight Tesla P100 GPUs with the NVLink intra-node fabric and a quad EDR Infiniband interconnect between nodes. OpenMPI 1.10.6 was used as the communications layer. We used gauge configurations from an ongoing $g - 2$ project, as shown in the following table.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
\textbf{Label} & $L_x$ & $L_y$ & $a$ (fm) & $m_\pi$ (MeV) \\
\hline
Small & 16 & 48 & 0.15 & $\approx 265$ \\
Medium & 32 & 48 & 0.15 & $\approx 133$ \\
Large & 48 & 64 & 0.12 & $\approx 133$ \\
X-Large & 64 & 192 & 0.06 & $\approx 297$ \\
\hline
\end{tabular}
\caption{Lattice configurations and their physical parameters; the pion mass $m_\pi$ roughly corresponds to the quark mass parameter that appears in equation \ref{eq:2}.}
\end{table}

The goal of our tests is to determine if the optimizations explored in \cite{10} bear fruit when combined in blockCGrQ. As a model test case, we consider stochastic trace estimation for the HISQ stencil given in equation \ref{eq:2}. This is a relevant measurement for the $g - 2$ project and is also a scenario which necessarily requires a large number of inversions. In this case, we have a total of 192 rhs per test: 24 noise sources, each partitioned into 8 disjoint subsets. For the 1–2 GPU tests on the Small and Medium datasets, we utilize the Truncated Solver Method (TSM) \cite{12}, which consists of a predictor-corrector scheme, partitioning the trace of the inverse matrix into two sweeps: first a cheaper bulk prediction is calculated, at the expense of introducing a bias that must be corrected in a subsequent high precision (fine) computation. The sloppy relative residual was set to $5 \times 10^{-9}$, whereas the fine solves used $2 \times 10^{-9}$. For simplicity, in our strong scaling tests, we only consider fine solves to a relative tolerance of $2 \times 10^{-12}$.

The primary quantity of interest is the total time to iteratively invert $M$ against all 192 rhs. Our baseline is the current CG implementation in QUDA where we sequentially invert on all rhs. For comparison, we perform blockCGrQ in blocks of rhs of size 8, 16, 24, 32, 48, and 64, leading to 24, 12, 8, 6, 4, and 3 applications of blockCGrQ, respectively. We skip some cases as applicable due to insufficient memory or unreasonably small local volumes. We consider both the case of full double precision and a mixed
double-single solve to explore the viability, stability, and potential benefits of block reliable updates. We choose to trigger a reliable update when the relative residual drops by a factor of 10. Double-half mixed-precision solves will be a point of future work as noted in [8].

7.2. Results: Single Node

In the single-node case, given the optimizations discussed above, our implementation of blockCGrQ should generally display a monotonic decrease in the complete time to solution for all 192 rhs. An appreciable increase in the time to solution would indicate a remnant quadratic scaling in memory streaming, or, in the case of mixed precision, a breakdown of stability. For smaller block sizes, this is in part due to the reduced total iteration counts due to the sharing of the Krylov space inherent to blockCGrQ, but more importantly because we can better saturate memory bandwidth. For larger block sizes, after fully saturating memory bandwidth, the benefit is near solely from the reduced total iteration count.

This behavior is reflected in figure 5, the total time to solution; figure 6, the average TFLOPS per second; and the green (lowest) and blue (second highest) curves in figure 7, the total number of iterations. The relative reduction in iteration count for a double-precision case is emphasized in figure 8. We see that the total time to solution does indeed monotonically decrease, reaching a $7 \times$ speedup for double precision and $4 \times$ speedup for mixed-precision, although the benefits become less drastic for a larger block size. The drastic speedup for smaller block sizes is largely due to a better utilization of memory bandwidth, as the iteration count does not greatly decrease. After memory bandwidth has saturated, roughly correlated to a saturation in the TFLOPS per second, the benefit comes from a significant reduction in the number of iterations.

Figure 5: Time to solution for the smaller volumes as a function of the number of RHS. For $32^3 \times 48$, 2 GPU, the time to solution is essentially consistent between double and mixed-precision for 48 and 64 right hand sides.

Figure 6: Performance in TFlops/s for the larger volumes as a function of the number of RHS.

We note that the pure-double-precision solve saturates before the double-single-mixed-precision solve: this reflects both a better utilization of streaming memory bandwidth. For intermediate block sizes, this also reflects a stability of the mixed-precision solve. For larger block sizes, the double-precision solve and the mixed-precision solve take a consistent amount of time (48 and 64). This coincides with a slight breakdown in the stability of the mixed-precision solve, where the reduced time per iteration is offset by the relative increase in iteration count. Regardless, it is still slower than the mixed-precision solve at a lower block size (32). It is thus important to appropriately tune the block size; bigger is not necessarily better.

7.3. Results: Strong Scaling

In considering the strong-scaling regime of our implementation of blockCGrQ, we ran on the NVIDIA SaturnV cluster. The key features to highlight are that we are able to better utilize the GPUs by overlapping computation with network communications during the matrix-vector operation. In addition, we can reduce the overall number of MPI reductions by simultaneously reducing over multiple rhs as per utilizing blockCGrQ. There is also the subtle benefit that, in cases where the local volume per GPU becomes small, we can still saturate streaming memory bandwidth by sufficiently increasing the block size.

It is difficult to decouple all three of these benefits. However, we can generally comment on the achieved speedup by again considering figure 9, the total time to solution; figure 10, the average TFLOPS per second; and the yellow (second lowest) and red (highest) curves in figure 7, the total number of iterations. The relative reduction in iteration count for a double-precision case is emphasized in figure 8. We emphasize that, despite corresponding to a larger overall volume, it is not surprising that the X-Large volume (yellow) took far less iterations than the Large volume (red)—the HISQ stencil in the X-Large case is better...
conditioned than the Large case in one-to-one correspondence with the larger $m_\pi$ in table 1. It is more important to consider the relative behavior along the curves.

Our data shows that we achieve an impressive improvement over CG in the strong-scaling regime for a smaller blocksize across both data sets and both precision studies. This is, in some sense, reflective of both the single-node optimizations and the additional strong-scaling benefits. In the case of the X-Large measurement set, the TFLOPS per second monotonically increases with both the number of rhs and the number of GPUs. This may be from a combination of there being increasing work to overlap with communications and additional memory bandwidth to saturate with smaller local volumes (especially in the mixed-precision case). This suggests that, given the resources, the algorithm would continue to strong scale well. The relatively low iterations count corresponds to a relative well-conditioning of the HISQ stencil on the X-Large data set: an increased blocksize always leads to a reduction in the number of iterations.

We see a smaller benefit for the Large measurement set, where in some cases the TFLOPS per second saturates. This could be due to a saturation of memory bandwidth on a local node combined with network latency being hidden behind computation, which in some regards indicates we can strong scale ideally. For very large block sizes there is an increase in the time to solution in the mixed precision case: this is likely due to a decreased stability in the mixed precision case. However, in all cases, there is an ideal number of rhs where we can minimize the total time to solution, and for our X-Large data set, that ideal pushes the maximum blocksize and strong-scaling limit we can
achieve with our current tests. In the end the benefits translate into a 2-3× speedup factor for the mixed double-single precision solver, and a 4-5× speedup factor for double precision, compared to the baseline.

8. Future Work

Our initial implementation of blockCGrQ overcomes many of the traditional issues which stifled the development and use of block-Krylov methods. There are several low-hanging fruits that can extend the application of block-CGrQ in its current form that may be of interest. For example, deflation methods, or any other method which improves the initial guess to a linear system, can be trivially combined with block-Krylov methods.

There are more advanced algorithmic pursuits that can be considered. As the Lanczos method is closely related to CG, there is an analogous block-Lanczos method that is related to block CG. At this time, we do not know if a Lanczos relation can be easily recovered from the modifications made in blockCGrQ, but if possible it may allow for the development of a block analogy to the EigCG \cite{2} which generates and iteratively improves a deflation space along the course of a Krylov solve without any additional stencil applications.

There are further improvements possible within the scope of our current work. One novel development in this work was the implementation of reliable updates \cite{13} in a block-Krylov method. While this was successful in the case of stabilizing a full double-precision solve as well as a mixed-precision double-single solve with a modestly sized number of rhs, our implementation of reliable updates broke down for double-single with a large number of rhs and completely broke down for a double-half mixed-precision solve. There are more advanced reliable-update methods in the literature for non-block-Krylov methods, which could potentially be generalized for block-Krylov methods. A stable implementation with half precision would significantly speed up portions of blockCGrQ that are still memory bandwidth bound. It would also speed up the matrix-vector portions of the algorithm because the HISQ stencil could now be applied in low precision. Last, it would reduce the overall memory footprint which can be the limiting factor when running at low-node count.

As a last note, other non-block-Krylov methods have block-Krylov extensions. The technology we have developed for an efficient implementation of blockCGrQ, i.e. block-optimized stencil operations and multi-BLAS operations, are immediately applicable to other block-Krylov methods. We expect, however, that it will be essential to first develop better reliable-update methods for block-CGrQ before exploring other solvers.

9. Conclusions

In this paper we revisited the blockCGrQ solver proposed in \cite{23}. We present an implementation that, by exploiting data locality for reductions and streaming multi-BLAS operation, reduces the naïve quadratic scaling of memory traffic to almost linear scaling. Using a block-optimized matrix-vector operation we obtain a speedup that multiplicatively combines with the reduced iteration count from the blockCGrQ algorithm. With its increased parallelism it is an algorithm very well suited also for future exascale machines where we expect the gap between peak floating point operations per seconds and memory bandwidth will continue to increase. We have also laid the groundwork and demonstrated the algorithm working using mixed precision, a further important building block for efficient algorithms. In our real world LQCD use case we demonstrated a speedup of 5×, notably not requiring any overhead from setup and also not requiring large storage for precalculated inputs such as eigenvectors. The availability of the building blocks for an efficient implementation of blockCGrQ can be employed with new algorithms, in order to improve lattice field theory computational methods to allow new physics research. Precision results in LQCD are already competing in accuracy with the best experimental results, but in a lot of areas computational demands are still the limiting factor.

Acknowledgement

The work by A.S. was done for Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the U.S. Department of Energy, Office of Science, Office of High Energy Physics. A.V. was supported by the U.S. National Science Foundation under grant PHY14-14614. E.W. was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. The authors are grateful for the ORNL-sponsored 2017 GPU Hackathon hosted at the Juelich Supercomputing Center where significant development work was accomplished.

References

\begin{thebibliography}{9}
\bibitem{1} K. Bitar, A. D. Kennedy, R. Horsley, S. Meyer, P. Rossi, The QCD finite temperature transition and hybrid monte carlo, Nucl. Phys. B313 (1989) 348–376. doi:10.1016/0550-3213(89)90323-4
\bibitem{2} A. Stathopoulos, K. Orginos, Computing and deflecting eigenvalues while solving multiple right-hand side linear systems with an application to quantum chromodynamics. SIAM Journal on Scientific Computing 32 (1) (2010) 439–462. arXiv:http://dx.doi.org/10.1137/080725532, doi:10.1137/080725532 URL: http://dx.doi.org/10.1137/080725532
\bibitem{3} W. M. Wilcox; Deflation methods in fermion inverters, PoS LAT2007 (2007) 025. arXiv:0710.1813
\end{thebibliography}
[4] J. Brannick, R. C. Brower, M. A. Clark, J. C. Osborn, C. Rebbi, Adaptive Multigrid Algorithm for Lattice QCD, Phys. Rev. Lett. 100 (2008) 041601. arXiv:0707.4018 doi:10.1103/PhysRevLett.100.041601

[5] R. Babich, et al., Adaptive multigrid algorithm for the lattice Wilson-Dirac operator, Phys. Rev. Lett. 105 (2010) 201602. arXiv:1005.3043 doi:10.1103/PhysRevLett.105.201602

[6] M. Lüscher, Local coherence and deflation of the low quark modes in lattice QCD, JHEP 07 (2007) 081. arXiv:0706.2298 doi:10.1088/1126-6708/2007/07/081

[7] 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. arXiv:0911.3191 doi:10.1016/j.cpc.2010.05.002

[8] H. Calandra, S. Gratton, J. Langou, X. Pinel, X. Vasseur, Flexible variants of block restarted gmr methods with application to geophysical SIAM Journal on Scientific Computing 34 (2) (2012) A714–A736. arXiv:https://doi.org/10.1137/10082364X URL https://doi.org/10.1137/10082364X

[9] P. Jolivet, P.-H. Tournier, Block iterative methods and recycling for improved scalability of linear solvers in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '16, IEEE Press, Piscataway, NJ, USA, 2016, pp. 17–17:14. URL http://dl.acm.org/citation.cfm?id=3014904.3014927

[10] E. Bavier, M. Hoemmen, S. Rajamanickam, H. Thornquist, Amesos2 and belos: Direct and iterative solvers for large sparse linear systems, Sci. Program. 20 (3) (2012) 241–255. doi:10.1156/2012/2543875 URL http://dx.doi.org/10.1155/2012/243875

[11] H. Ji, M. Sooskonik, Y.Li, [An implementation of block conjugate gradient algorithm on cpu-gpu processors] in: Proceedings of the 1st International Workshop on Hardware-Software Co-Design for High Performance Computing, Co-HPC '14, IEEE Press, Piscataway, NJ, USA, 2014, pp. 72–77. doi:10.1109/Co-HPC.2014.10 URL http://dx.doi.org/10.1109/Co-HPC.2014.10

[12] H. Ji, Y.Li, A breakdown-free block conjugate gradient method, BIT Numerical Mathematics 57 (2) (2017) 379–403. doi:10.1007/s10543-016-0631-2 URL http://dx.doi.org/10.1007/s10543-016-0631-2

[13] H. A. van der Vorst, Q. Ye, Residual replacement strategy in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis SC ’16, IEEE Computer Society, Washington, DC, USA, 2010, pp. 1–11. doi:http://dx.doi.org/10.1109/SC.2010.40 URL http://dl.acm.org/citation.cfm?id=3014904.3014995

[14] R. Babich, M. A. Clark, B. Joó, J. Brannick, R. C. Brower, Accelerating lattice QCD on gpus using fine-grained parallelization, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC'14, IEEE Press, Piscataway, NJ, USA, 2016, pp. 68:1–68:12. URL http://dl.acm.org/citation.cfm?id=3014904.3014995

[15] R. Babich, M. A. Clark, High-efficiency Lattice QCD computations on the Fermi architecture, Innovative Parallel Computing (InPar). 2012. doi:10.1109/InPar.2012.6339591

[16] R. G. Edwards, B. Joó, The Chroma software system for lattice QCD, Nucl. Phys. Proc. Suppl. 140 (2005) 582. arXiv:hep-lat/0409003 doi:10.1016/j.nuclphysbps.2004.11.254

[17] F. T. Winter, M. A. Clark, R. G. Edwards, B. Joó, A Framework for Lattice QCD Calculations on GPUs. 2014. arXiv:1408.5926 doi:10.1109/IPDPS.2014.112 URL http://inspirehep.net/record/1312379/files/arXiv:1408.5925.pdf

[18] D. P. O’Leary, The block conjugate gradient algorithm and related methods, Linear algebra and its applications 29 (1980) 293–322.

[19] C. Dubrulle, Retooling the method of block conjugate gradients, ETNA. Electronic Transactions on Numerical Analysis [electronic only] 12 (2001) 216–233.

[20] A. Dubrulle, Retooling the method of block conjugate gradients, Tech. Rep. 0611022 arXiv:arXiv:hep-lat/0311004

[21] R. G. Edwards, B. Joó, A. Strelchenko, M. Cheng, A. Gambhir, R. C. Brower, Accelerating lattice QCD multigrid on gpus using fine-grained parallelization, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC’16, IEEE Press, Piscataway, NJ, USA, 2016, pp. 68:1–68:12.

[22] O. Kaczmarek, C. Schmidt, P. Steinbrecher, M. Wagner, HISQ inverter for Multi-GPU Calculations on GPUs, 2014. arXiv:1403.0359 doi:10.1109/SC.2014.201 URL http://www.sciencedirect.com/science/article/pii/S0010465509002859

[23] H. Tadano, Modified Block BiCGSTAB for Lattice QCD, Comput. Phys. Commun. 181 (2010) 883. arXiv:0907.3261 doi:10.1016/j.cpc.2009.12.025

[24] Y. Nakanuma, K. I. Ishikawa, Y. Kuramashi, T. Sakurai, H. Tadano, Modified Block BiCGSTAB for Lattice QCD, Comput. Phys. Commun. 181 (2010) 34–37. arXiv:1104.0739 doi:10.1016/j.cpc.2011.08.010

[25] S. Birk, A. Frommer, A CG Method for Multiple Right Hand Sides and Multiple Shifts in Lattice QCD Calculations, PoS LATTICE2011 (2011) 027. arXiv:1005.0359

[26] S. Birk, A. Frommer, A deflated conjugate gradient method for multiple right hand sides and multiple shifts, Numer. Algorithms 67 (3) (2014) 507–529. doi:10.1007/s11075-013-9805-9 URL http://dx.doi.org/10.1007/s11075-013-9805-9

[27] O. Kaczmarek, C. Schmidt, P. Steinbrecher, M. Wagner, Conjugate gradient solvers on Intel Xeon Phi and NVIDIA GPUs in: Proceedings, GPU Computing in High-Energy Physics (GPUHEP2014): Pisa, Italy, September 10-12, 2014, 2015, pp. 157–162. arXiv:1411.4439 doi:10.3204/DESY-PDC-2014-06/28 URL http://inspirehep.net/record/1328508/files/arXiv:1411.4439.pdf

[28] S. Mukherjee, O. Kaczmarek, C. Schmidt, P. Steinbrecher, M. Wagner, HISQ inverter on Intel® Xeon Phi™ and NVIDIA® GPUs, PoS LATTICE2014 (2015) 044. arXiv:1509.1510

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

[30] E. Follana, Q. Mason, C. Davies, K. Hornbostel, P. Lepage, H. Trottier, Further improvements to staggered quarks, Nucl. Phys. Proc. Suppl. 129 (2004) 447–449, [arXiv:1408.5925]. doi:10.1016/j.nuclphysbps.2004.11.254
[34] M. R. Hestenes, E. Stiefel, Methods of Conjugate Gradients for Solving Linear Systems, Journal of Research of the National Bureau of Standards 49 (6) (1952) 409–436.

[35] A. A. Nikishin, A. Y. Yeremin, Variable block cg algorithms for solving large sparse symmetric positive definite linear systems on parallel computers, I: General iterative scheme, SIAM Journal on Matrix Analysis and Applications 16 (4) (1995) 1135–1153. arXiv:http://dx.doi.org/10.1137/0895479893247679
doi:10.1137/0895479893247679
URL http://dx.doi.org/10.1137/0895479893247679

[36] C. Bernard, et al., The MILC Code, http://www.physics.utah.edu/~detar/milc/milcv7.pdf (2010).

[37] http://lattice.github.com/quda (2017).

[38] G. H. Golub, C. F. Van Loan, Matrix Computations 3rd Edition, Johns Hopkins Studies in the Mathematical Sciences, 1996.
URL http://www.loc.gov/catdir/bios/jhu052/96014291.html;http://www.loc.gov/catdir/description/jhu051/96014291.html

[39] http://eigen.tuxfamily.org (2017).

[40] G. L. G. Sleijpen, H. A. van der Vorst, Reliable updated residuals in hybrid bi-cg methods, Computing 56 (2) (1996) 141–163. doi:10.1007/BF02309342
URL http://dx.doi.org/10.1007/BF02309342

[41] D. Merrill, CUR, https://nvlabs.github.io/cub (2016).

[42] G. S. Bali, S. Collins, A. Schafer, Effective noise reduction techniques for disconnected loops in lattice qcd, Comput. Phys. Commun. 181 (2010) 1570–1583. arXiv:0910.3970 doi:10.1016/j.cpc.2010.05.008