Optimizing a Multiple Right-hand Side Dslash Kernel for Intel Knights Corner

Aaron Walden [*1], Sabbir Khan[†1], Bálint Joó[‡2], Desh Ranjan[§1], and Mohammad Zubair[¶1]

[1] *Department of Computer Science, Old Dominion University, Norfolk, VA USA 23529*
[2] *Thomas Jefferson National Accelerator Facility, Newport News, VA USA 23606*

July 31, 2016

Abstract

There is a significant interest in the computational physics community to perform lattice quantum chromodynamics (LQCD) simulations, which can run into the trillions of operations. LQCD computations solve a sparse linear system using a Wilson Dslash kernel, which has an arithmetic intensity of 0.88-2.29. This makes Dslash memory bandwidth-bound on most architectures, including Intel Xeon Phi Knights Corner (KNC). Most research optimizing the Dslash operator has been focused on single right-hand side (SRHS) linear solvers. There is a class of LQCD computations which aims to solve systems with multiple right-hand sides (MRHS), presenting additional opportunities for data reuse and vectorization. We present two approaches to MRHS Dslash: a vector register blocking approach and one using the software package QPhiX with a custom code generator for low-level intrinsics. We observed significant speedups using our approaches, with sustained performance of over 700 GFLOPS (single precision) in one instance. We achieved up to 29% of theoretical peak performance compared to a maximum of 13% obtained by the previous SRHS method using QPhiX.

Keywords: LQCD, optimization, performance, Wilson-Dslash, code generator, parallel programming, vectorization, Xeon Phi Knights Corner

Preprint Numbers: JLAB-IT-16-01

[*]awalden@cs.odu.edu
[†]skhan@cs.odu.edu
[‡]bjoo@jlab.org
[§]dranjan@cs.odu.edu
[¶]zubair@cs.odu.edu

1 Introduction

Lattice quantum chromodynamics (LQCD) is a uniquely important computational technique for the simulation of the strong nuclear force, which governs quark and gluon interaction in the nucleon. LQCD is to date the only non-perturbative, model-independent, quantum field theory in use for the calculation of quark-gluon interactions. LQCD simulations are thus needed for areas of research at the frontiers of physics, including understanding of the allowed states and structure of hadronic and nuclear matter. To facilitate numerical computation, LQCD discretizes space-time as a 4-dimensional hypercubic lattice. To simulate larger lattices with shorter lattice spacing, ever-increasing computing power is required. The computational core of LQCD with Wilson fermions is the Wilson Dslash operator (henceforth Dslash), a nearest neighbor stencil operator summing matrix-vector multiplications over lattice points, whose performance is bandwidth-bound on most architectures [7]. Reportedly, up to 90% of LQCD running time may be spent applying Dslash [6]. Clearly, optimization of Dslash is paramount in the performance of LQCD simulations.

We approach the optimization of Dslash by designing two different kernels for Intel Xeon Phi Knights Corner (KNC). Significant research has been devoted to exploring KNC's potential to drive LQCD simulations [5], [7], [14], [10], [8]. The bulk of this research in the area of Dslash has involved single right-hand side (SRHS) solvers, though [10] and [12] use multiple right-hand sides (MRHS). We describe Dslash kernels applied to MRHS in parallel on a single node. For our approaches, we have written kernels which use 8 and 16 right-hand sides (RHS). The intuition behind a MRHS approach is that each RHS can make use of the same gauge field configuration (see Section 2.1), which can increase the arithmetic intensity of the Dslash operator from 0.92 (SRHS) to 1.47 (16 RHS) in an otherwise unoptimized scenario.

In the first of our two approaches, we hand code a kernel using KNC vector *intrinsics* (see Section 2.2) which uses a register blocking (RegBlk) technique to minimize the pressure register spills would put on the L1 cache. We reduce register pressure by specifying a certain order to the matrix-vector multiplications and by holding accumulated sums in vector registers. This is straightforward in a kernel with 8 RHS, but requires some tricks to eliminate spills in a 16 RHS kernel, due to the limited number of vector registers. We also explain our approach to vectorization for KNC's powerful vector processing unit (VPU). In contrast to RegBlk, vectorization is simple for 16 RHS and challenging for 8 RHS.

Our second approach optimizes a kernel using the *QPhiX* LQCD framework [4] and its custom code generator [3], which generates SIMD intrinsics for modern architectures. The goal of QPhiX is to provide a high level module which handles threading, cache-blocking, and MPI communication and a module which provides an abstraction into which the code generator can plug SIMD intrinsics for various architectures. QPhiX and its code generator also provide multiple configurations (blocking, vector length, precision) and approaches to

vectorization. We modified the code generator to provide a new configuration option for 16 RHS and to support prefetching on KNC. We also modified QPhiX to support MRHS (for example, a new memory layout is necessary) and our own site traversal strategies (see Section 6). To implement 8 RHS, we modified the 16 RHS code produced by the code generator.

The remainder of this paper is structured as follows. In section 2 we describe the Dslash operator at a high level as well as the target hardware, KNC. In section 3 we detail our RegBlk implementations. In section 4 we describe the QPhiX and code generator (QPhiX-CoGen) approach. Finally, we discuss our results and compare them to a SRHS QPhiX-CoGen kernel.

2 Background

2.1 Dslash

LQCD's Dslash operator is applied over a finite space-time discretized as a 4-dimensional hypercubic lattice. One may imagine the lattice as a set of linked points. In LQCD, quark fields are represented by lattice points and gluon fields by the links. The lattice has some length in a direction $\mu \in \{x, y, z, t\}, L_\mu$. The number of sites on a lattice is given by $VOL = L_x \times L_y \times L_z \times L_t$. Each site (with coordinates $<x, y, z, t>$) has 2 neighbors for each direction, forward and backward, which correspond to the positive and negative directions on that axis. For neighbor site determination, the lattice has periodic boundary conditions.

We can think of Ψ and $\mathbf{U}$ as the input to Dslash. Ψ defines ψ for every site, a 4×3 matrix of complex numbers called a *spinor*. $\mathbf{U}$ defines U for every site, a 3×3 matrix of complex numbers called a gauge field or gauge matrix. Since U are members of the group $SU(3)$, they can be stored in several representations. A particular trick is to store only two rows of a 3×3 unitary matrix representation, and to reconstruct the 3rd row by appealing to unitarity (i.e. that $\det(U) = 1$) from the complex conjugate of the vector product of the first two rows.

Dslash computes χ for all of the even or odd (based on sum of coordinates) sites on the lattice. χ is also a spinor of the same dimensions as ψ, and in a full solver will be used as the input to another iteration of the computation.

We can employ a spin projection trick, reducing $\psi \in \mathbb{C}^{4\times3}$ to $\psi' \in \mathbb{C}^{2\times3}$, which increases the arithmetic intensity of the operator. Applying Dslash to a site will calculate χ by summing $U\psi'$ for each neighbor of the site in question into what we'll call χ^u, the upper sum. Simultaneously, the lower sum χ^l will be computed as a permutation of the result of $U\psi'$ for each neighbor. These two sums together form χ. For more information about Dslash, please see [9].

2.2 Intel Xeon Phi Knights Corner

Knights Corner is a line of many-core PCIe coprocessor cards in the Intel Xeon Phi family. KNC cards are massively parallel chips with high memory bandwidth suited for scientific computing applications. They feature up to 61 cores running at up to 1.238 GHz. A key feature of KNC is its VPU. KNC boasts 512-bit vector registers, capable of SIMD operations

on 16 single precision numbers simultaneously. Individual cores can support up to 4 hardware threads, each with a full context of registers, including vector registers, of which there are 32. Intel provides a set of C-style functions called intrinsics, which act directly on vector registers in an assembly-like way. For further KNC details, please see [1].

3 Multiple Right-hand Side Performance Model

The approach of solving for MRHS is an established technique in LQCD research [10], [13]. We create a new operator with lower bandwidth needs than N applications of the original operator. Application of Dslash to a single site performs 1320 FLOPs. Because we assume the operator is bandwidth-bound [7], we can analyze the expected speedup for different numbers of right-hand sides if we take the performance to be equal to: $\frac{\text{FLOPs}}{\text{byte}} \times$ bandwidth. Then, we need only divide the MRHS FLOPs/byte by the SRHS FLOPs/byte to compute speedup. We begin by defining variables. N is the number of right hand sides. Let G be the number of components in a gauge matrix and let $F = \texttt{sizeof(float)}$. In this model, we will not consider architectural details like cache line size. For each site's Dslash computation, we need to load $8GF$ bytes. Let S be the number of components in a spinor (24). Let us define a SRHS *neighbor spinor reuse factor* R_1 and similarly R_N for MRHS. This is the number of neighbor spinors already present in cache when processed. For each site in SRHS, then, we need to load $(8 - R_1)$ neighbor spinors plus one unavoidable spinor to write the output. In total, this is $SF((8 - R_1) + 1)$ bytes in spinors. For MRHS, we replace R_1 with R_N and multiply the neighbor spinors and FLOPs by N. That gives us MRHS FLOPs/byte of $\frac{1320N}{8GF+NSF((8-R_N)+1)}$ and SRHS FLOPs/byte of $\frac{1320}{8GF+SF((8-R_1)+1)}$. Then, to compute speedup, we divide MRHS by SRHS:

$$\text{speedup} = \frac{8NGF + NSF(9 - R_1)}{8GF + NSF(9 - R_N)}$$

To visualize our speedup as values of N and R_N vary, we can assume values of the other variables. We can take R_1 to be 7, which is borne out in practice for at least lattices up to 32^4, though it requires substantial effort to achieve [14]. Single precision ($F = 4$), a spinor size of $S = 24$, and 12-compression of U meaning $G = 12$ (see Section 6), yields the graph of speedup versus N shown in Figure 1. A separate curve is shown for each value of parameter $R_N \in \{0, 1, \ldots, 7\}$, which is not necessarily restricted to integers. Notice we require $R_N > 3$ to achieve any speedup, which will limit our performance gains for lattices of medium to large size, as spinor data scales by N and cache size remains fixed.

4 Register Blocking Approaches

8 RHS. We must compute $U\psi'$ for a single U and 8 different ψ'. This suggests a simple vectorization – broadcast a component of U to fill a register then fill another register with whatever will be multiplied by that component of U. That will be a row of ψ'. Happily,

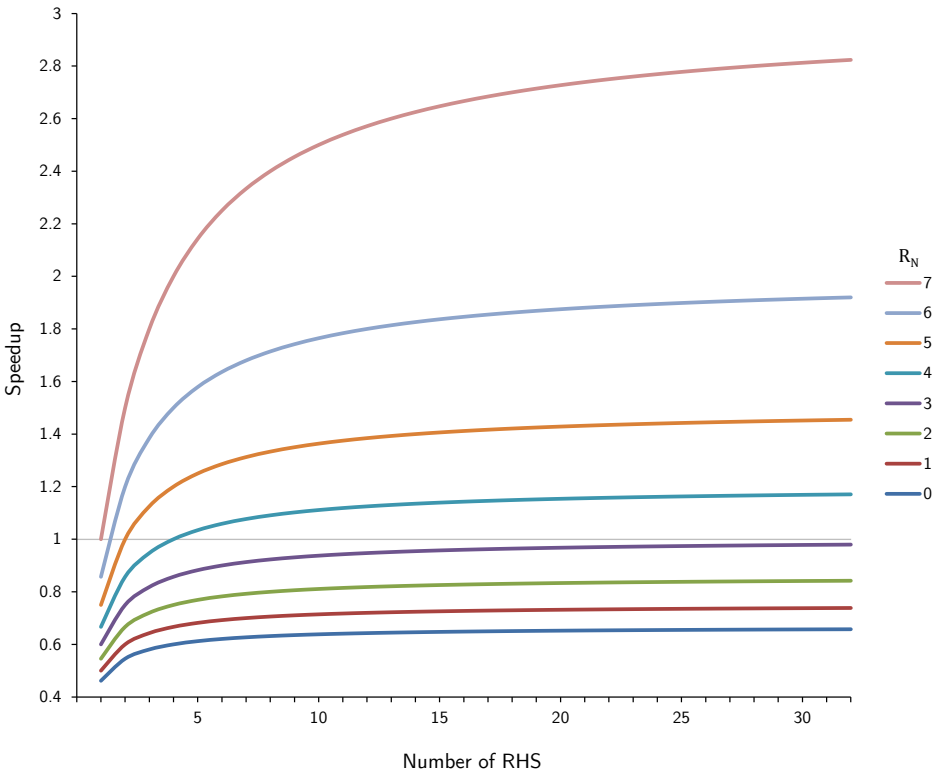

Figure 1: MRHS scaling for values of N and R_N.

because there are 2 (real or imaginary) components in a row of ψ' and 8 RHS, we can fill a vector register with a row of ψ' by treating the real and imaginary components separately. We cannot store rows of ψ' in memory directly, however, as they are a projection (see Section 2.1) of some ψ, meaning the components of ψ' are formed by computing the sum or difference of two components of ψ. Thus, we must store components of ψ in pairs of 8 (for each RHS) which interact with other pairs to form a row of the projected ψ'. With such a layout, we fully vectorize both the projections and the matrix-vector multiplications.

In the projection of ψ, when row 0 interacts with row 2, row 1 interacts with row 3. For a given μ, real components always interact with real or imaginary components and vice versa. All interactions occur intra-column. The direction of μ only changes signs. Bearing in mind these restrictions and the fact that KNC allows permutation across 256-bit lanes, it is clear we can pair components by column and realness, and these pairs and the general data layout are given in Figure 2.

For 8 RHS, the upper and lower sums occupy 12 vector registers (24 components, 2 components per register). The projected matrix occupies 6 registers. We can project a single component of U at a time, multiplying by the rows of ψ' and storing the result in two accumulator registers, one for the real and imaginary components of the row of the result. We then proceed across a row of U, computing the dot product of the first row and column of U

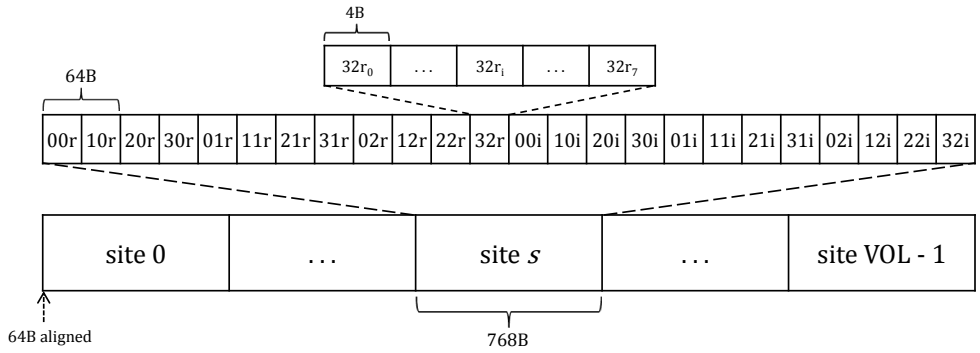

Figure 2: Layout of 8 RHS RegBlk in memory.

and ψ', respectively. Once a component of $U\psi'$ has been computed, it can be added directly to the upper sum and some component of the lower sum. his requires only an additional 3 registers, bringing the total to 21, well short of the limit of 32.

16 RHS. KNC's vector length is 64 bytes or 16 4-byte floats. This simplifies the vectorization scheme for 16 RHS. Each component of ψ fills one vector register, so there is no need to worry about how different components will interact during projection and multiplication. Thus, all of the difficulties of the 8 RHS scheme vanish, and we simply place each component of ψ in its own register.

16 RHS register blocking requires a different approach than the one we use for 8 RHS. We must necessarily keep the upper and lower sums in registers (to avoid spills) and this requires 24, leaving only 8 for calculations. Since our 8 RHS algorithm requires loading all projections, that would require $24 + 12$ registers for 16 RHS, so we must use a different approach.

The problem lies in the temporary accumulation of sums. What we can do to solve this is propagate the sign changes required by the lower sum down to the lower level multiplies, changing `fmadds` to `fnmadds` where appropriate. Then we no longer require any intermediate accumulation registers. We can add directly to the upper and lower sums when computing $U\psi'$. We offer a practical example to aid in understanding. For each direction, we add to χ^u_{00r} (upper sum, component 00r) the real part of the complex dot product of the first row and column of U and ψ', respectively.

$$\chi^u_{00r} \leftarrow \chi^u_{00r} + u_{00r}\psi'_{00r} - u_{00i}\psi'_{00i} + u_{01r}\psi'_{10r} - u_{01i}\psi'_{10i} + u_{02r}\psi'_{20r} - u_{02i}\psi'_{20i}$$

This is straightforward. Let us assume we are computing for the first direction backward. Then, we subtract the same dot product used to add to χ^u_{00r} from $01i$ of the lower:

$$\chi^l_{01i} \leftarrow \chi^l_{01i} + -(u_{00r}\psi'_{00r} - u_{00i}\psi'_{00i} + u_{01r}\psi'_{10r} - u_{01i}\psi'_{10i} + u_{02r}\psi'_{20r} - u_{02i}\psi'_{20i})$$

This is how we go about *unrolling* the multiplication to eschew intermediate sums. We simply compute the dot product twice, using `fmadd` and `fnmadd` where appropriate to account for the sign changes. Unrolling allows us to carry out the computation using only 5 registers (a row of ψ' and a component of U) in addition to the sums.

This approach results in approximately 23% more cycles spent on vector arithmetic instructions compared to the 8 RHS approach, but these may be hidden behind load latencies.

5 QPhiX and Code Generator based Approach

QPhiX and its code generator provide an approach to vectorization over multiple sites, but no MRHS option. We have modified both QPhiX and its code generator to compute the Dslash operator using 16 RHS. To do so, we modified QPhiX's memory layout to add an extra dimension for each site, over which we vectorize. We also customized the threaded loop over sites in order to experiment with our own site traversal strategies (see Section 6).

In typical SRHS Wilson Dslash, vectorization is done over sites. Thus, each vector register holds multiple sites' worth of data. The unmodified code generator generates intrinsics according to this requirement. We have modified the code generator for our MRHS implementation to generate intrinsics to vectorize our code on the number of RHS. We also modified the broadcast of elements of U to only broadcast from a single site's U. Finally, we modified the code generator to produce prefetch instructions of gauge field data for the current site and spinor data for the current and next sites.

For 8 RHS, the approach (pairing, memory layout, etc.) is the same as RegBlk, the only difference is that the code generator based approach lacks register blocking.

6 Results

Experimental Setup. To optimize our different approaches to Dslash implementation, we test every combination of the following options: number of RHS (8/16), lattice size ($8^4/16^4/24^4/32^4$), software prefetching (L1/L2/both/none), thread interleaving (interleaving/default), gauge compression (12/16/default), and cache-controlling traversal (CCT/default).

We discuss additional optimization experiments in [2]. In total, we experiment with over 192 different parameter combinations. In this section we will briefly describe the novel experimental techniques. Please see the referenced thesis for full details.

Thread interleaving divides a chunk of sites among the threads of a single core instead of allocating $\frac{1}{\text{threads\_per\_core}}$ of that chunk of contiguous sites to a single thread (the default allocation). In the former chunk, threads *step over* one another in an interleaved pattern. For example, with 4 threads per core, thread 0 would process sites 0, 4, etc., thread 1 would process sites 1, 5, and so on. See Figure 3. *Compression* refers to the size of stored gauge matrices, as mentioned in Section 2.1. *Cache-controlling traversal* (CCT) is a method of lattice traversal (order sites are processed by threads) that aims to increase the effective size of L2 by performing controlled evictions using the `_mm_clevict` intrinsic. By traversing slices of the t dimension one a time, we can use controlled evictions to *make room* for new data by explicitly evicting data which we know will not be reused. LRU evictions may result in eviction of data which could be reused by Dslash. Our t slice traversal makes it possible to perform controlled evictions in this way.

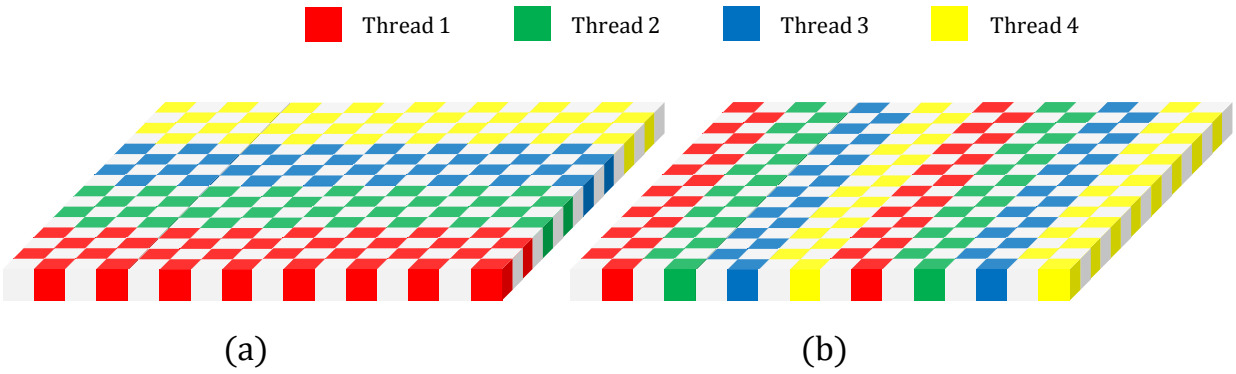

Figure 3: **(a)** Default chunking. Threads process their chunks lexicographically. **(b)** Interleaved traversal. Threads in a core alternate sites in a lexicographical manner. To divide the sites thusly, take the plane shown to be the entire lattice.

All kernels were compiled by the Intel C++ compiler version 16.0.0 and run in native mode on a Xeon Phi 7120P card using 60 cores and 4 threads per core. All experiments were run in single precision. Unless otherwise noted, results are given in GFLOPS.

6.1 Optimization Results

We begin by discussing the results of our optimization experiments on our kernel implementations. For a full treatment of these results for RegBlk, please see [2].

Prefetching. As expected, both L1 and L2 prefetching increase performance in almost all cases. L1 prefetches increase speeds by roughly 20%. L2 prefetches increase speeds by 40–60% with higher increases for larger lattices. The exception is that L2 prefetches decrease speed by 15% in the case of lattices of volume 8^4 for both 8 and 16 RHS, but only for our RegBlk approach and only when 2 MB memory pages are enabled. At 8^4, for a given core, all processed sites fit in L2, eliminating the need for these prefetches. Why there is no such performance change for QPhiX+CoGen is unknown.

Interleaving. Thread interleaving results show consistency for number of RHS but are inconsistent across lattice volumes. Results show a strong increase for 8^4, strong decrease for 16^4, little change for 24^4, and strong increase for 32^4. The number of RHS does not strongly influence this pattern. Results are amplified for our RegBlk approach. We hypothesize that interleaving results are sensitive to the access pattern of site data, which is consistent for a given lattice volume.

Compression. Gauge compression results are as expected. In most cases, compression to 16 numbers gives superior performance due to increased arithmetic intensity. The reason that 12-compression fails is that backward U are not stored contiguously. U are associated with the forward links of one site and those matrices are stored contiguously in memory, indexed by the linearized index of the site. Backward U are loaded as the (Hermitian conjugate of

the) forward link of the backward neighbor. These U are thus stored in 4 noncontiguous locations. Because 12 floats is smaller than the cache line size of KNC (64B), an unused 16B are loaded for every backward 12-compressed U. This results in the same amount of data being loaded for 12 and 16 compression, but 12-compression performs extra integer operations. For very small lattices (8 RHS 8^4), the default (uncompressed) option is superior because there is an excess of memory bandwidth.

Cache-controlling traversal. Explicit evictions were of no use in either approach. However, for our RegBlk kernel, CCT without evictions (essentially the blocking scheme of [11]) increases speeds for lattices of volumes 24^4 and 32^4 for both 8 and 16 RHS. A stronger result is observed for 32^4. For smaller lattices, the increased number of thread synchronization barriers overshadows any gains from CCT. We observe a synergistic increase in speed for 32^4 when also employing interleaved traversal. CCT combined with interleaving increases speed by 32% higher than CCT and interleaving if we consider their effects additively. This makes intuitive sense when considering that the access pattern for CCT+interleaving differs from using either alone. Though we observe improvement using CCT for 24^4, a different combination of results is superior. We do not observe the synergistic effect of interleaving and CCT for 24^4, which is consistent with interleaving's ineffectiveness for 24^4. Again, in the QPhiX+CoGen approach, we see a similar pattern of results but they are dampened.

Table 1 shows the (condensed) results of our optimization experiments on our kernels. For the full results of all 192+ combinations of experimental parameters, see [2]. The relative difference in performance for 8 RHS versus 16 RHS is what we would expect due to the higher arithmetic intensity of 16 RHS.

	RegBlk				QPhiX+CoGen			
	8 RHS		16 RHS		8 RHS		16 RHS	
VOL	Def	Opt	Def	Opt	Def	Opt	Def	Opt
8^4	579	651	640	**708**	306	343	385	411
16^4	405	419	463	**473**	399	440	392	425
24^4	300	337	326	**375**	302	346	289	320
32^4	255	346	235	**387**	263	301	243	304

Table 1: Optimization results (GFLOPS). Highest results in bold. *Def* refers to the unoptimized base RegBlk or QPhiX+CoGen MRHS implementation. *Opt* refers to the highest result achieved using some combination of our optimization techniques.

We should note here that we performed an additional experiment to verify the soundness of our 16 RHS RegBlk approach. We compared our unrolled approach to a modification using accumulator registers in place of extra dot products, ignoring spills. Our unrolled approach performed some 25% better or more in all test cases.

6.2 Results Comparison

RegBlk vs. QPhiX+CoGen. In all cases, the best results are obtained by our 16 RHS RegBlk approach. For lattices of size 8^4 and 16^4, QPhiX+CoGen 16 RHS performs nearly as well as the RegBlk approach, if we discount several factors. In the RegBlk approach, the OpenMP thread spawn is placed outside of the iteration loop, but QPhiX+CoGen pays the cost of spawning threads on every iteration, which accounts for approximately 150 GFLOPS lost at 8^4. The remaining difference is accounted for by the L2 prefetching issue from the previous section. For 32^4, RegBlk gains a significant amount of performance from thread interleaving and cache-controlling traversal. QPhiX+CoGen does not show nearly the same level of performance gain from these options. Because the implementations only differ meaningfully in the inclusion of register blocking, we surmise the differences observed, especially in situations with higher cache use (CCT+interleaving at 32^4), are due to increased cache pressure caused by register spilling, which is avoided by RegBlk.

VOL	MRHS	SRHS	Speedup
8^4	708	–	–
16^4	473	251	1.88
24^4	375	255	1.47
32^4	387	315	1.23

Table 2: Comparison of highest results by lattice size, MRHS vs. SRHS (GFLOPS).

MRHS vs. SRHS. In Table 2 we compare the best results from our MRHS kernels to the results of a SRHS kernel using unmodified QPhiX+CoGen which has been tested on the same hardware setup. We observe a set of speedups consistent with the performance model we introduced in Section 3. In this model, speedup is dependent on the parameter R_N, the MRHS spinor reuse factor, which is not possible to measure directly. We note that at maximum reuse (which we estimate to be 7 out of 8 neighbors reused), the achievable speedup is approximately 2.8. Though we do not have a result for SRHS 8^4, if we estimate that number at 251, the speedup for MRHS at 8^4 would be 2.8. This follows from the lattice size: at this size, R_N is very high because nearly the entire lattice fits into L2. For further evidence validating our model, consider 16 RHS 16^4, which has neighbor (read) data equal to SRHS 32^4. Considering that the write data also scales with N, we would expect a somewhat smaller reuse factor for 16 RHS 16^4. Given the speedup of 1.88, we calculate a reuse factor of approximately 6 for 16 RHS 16^4, which is very close to the 7 we assume for SRHS 32^4.

As the amount of MRHS spinor data scales with N, the reuse factor and speedup drop quickly. Looking at Table 1, we see that without optimization, speedup for 16 RHS 32^4 is below 1.0. Measuring bandwidth at 140 GB/s, we can calculate, using our model, that our optimizations for RegBlk 16 RHS 32^4 must have increased R_N by a factor of 3.8 in order to achieve the speedup we did over the default RegBlk 16 RHS implementation.

7 Conclusions and Future Work

We have presented the optimization of a single precision Dslash kernel for KNC using a dual approach which included a register blocking hand-coded kernel and a kernel customized from QPhiX and its code generator. We achieved 29% of peak performance on our target architecture, KNC, compared to 13% achieved by the previous SRHS kernel. We observed speedups of 23% and greater in all tested regimes, showing that our kernel is effective on real world problem sizes. We have shown with a direct comparison that register blocking for KNC's VPU may be a critical component of high-performance kernels, as the non-blocked approach showed dampened ability to gain speedup from advanced cache-blocking techniques that are required to achieve the kind of spinor reuse necessary for MRHS implementations to be worthwhile [2].

In our future work, we will continue to optimize our highest performing kernel. We plan to investigate controlled spilling of registers to test the feasibility of implementing a RegBlk approach similar to that of 8 RHS for 16 RHS by using strategic stores to L1 to avoid having to compute extra dot products. We will attempt to increase data reuse in our kernel by testing more advanced lattice traversal techniques. In the area of QPhiX+CoGen, we will give the code generator the ability to generate 8 RHS code as it currently does 16 RHS. After optimization is complete, we will integrate our kernel into a full multi-node LQCD solver.

8 Acknowledgments

This work was partially supported by a grant from Jefferson Lab. Aaron Walden and Sabbir Khan were also partially supported by the Old Dominion University Modeling and Simulation Fellowship Program and gratefully acknowledge this support. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Nuclear Physics under contract DE-AC05-06OR23177.

References

[1] Intel® Xeon Phi™ coprocessor: Software developers guide. Technical report, Intel Corporation, March 2014.

[2] Aaron Walden. An optimized multiple right-hand side dslash kernel for Intel® Xeon Phi™. Master's thesis, Old Dominion University, Norfolk, VA, 2016.

[3] B. Joó et. al. Code generator for the QPhiX library, Wilson fermions.

[4] B. Joó et. al. QPhiX: QCD for Intel Xeon Phi and Xeon processors.

[5] A. Diavastos, G. Stylianou, and G. Koutsou. Exploring parallelism on the Intel® Xeon Phi™ with lattice-QCD kernels.

[6] R. Gupta. Introduction to lattice QCD. arXiv:hep-lat/9807028.

[7] S. Heybrock, B. Joó, D. D. Kalamkar, M. Smelyanskiy, K. Vaidyanathan, T. Wettig, and P. Dubey. Lattice QCD with domain decomposition on Intel® Xeon Phi™ co-processors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 69–80, Piscataway, NJ, USA, 2014. IEEE Press.

[8] B. Joó, D. D. Kalamkar, K. Vaidyanathan, M. Smelyanskiy, K. Pamnany, V. W. Lee, P. Dubey, and W. Watson. Lattice QCD on Intel® Xeon Phi™ coprocessors. In J. M. Kunkel, T. Ludwig, and H. W. Meuer, editors, *Supercomputing*, pages 40–54, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[9] B. Joó, M. Smelyanskiy, D. D. Kalamkar, and K. Vaidyanathan. Chapter 9 - wilson dslash kernel from lattice qcd optimization. In J. Reinders and J. Jeffers, editors, *High Performance Parallelism Pearls Volume Two: Multicore and Many-core Programming Approaches*, volume 2, pages 139 – 170. Morgan Kaufmann, Boston, MA, USA, 2015.

[10] O. Kaczmarek, C. Schmidt, P. Steinbrecher, S. Mukherjee, and M. Wagner. HISQ inverter on Intel Xeon Phi and NVIDIA GPUs. *CoRR*, abs/1409.1510, 2014.

[11] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5D blocking optimization for stencil computations on modern CPUs and GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–13, Washington, DC, USA, 2010. IEEE Computer Society.

[12] D. Richtmann, S. Heybrock, and T. Wettig. Multiple right-hand-side setup for the DD-αAMG. In *Proceedings of the 33rd International Symposium on Lattice Field Theory*, July 2015.

[13] T. Sakurai, H. Tadano, and Y. Kuramashi. Application of block Krylov subspace algorithms to the Wilson–Dirac equation with multiple right-hand sides in lattice QCD. *Computer Physics Communications*, 181(1):113–117, 2010.

[14] M. Smelyanskiy, K. Vaidyanathan, J. Choi, B. Joó, J. Chhugani, M. A. Clark, and P. Dubey. High-performance lattice QCD for multi-core based parallel systems using a cache-friendly hybrid threaded-MPI approach. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–10, Nov 2011.