Spring 2016

An Optimized Multiple Right-Hand Side Dslash Kernel for Intel Xeon Phi

Aaron Walden
Old Dominion University

AN OPTIMIZED MULTIPLE RIGHT-HAND SIDE DSLASH KERNEL FOR INTEL® XEON PHI™

by

Aaron Walden
B.S. August 2014, Old Dominion University

A Thesis Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
May 2016

Approved by:

Mohammad Zubair (Director)

Desh Ranjan (Co-Director)

Bálint Joó (Member)

Michele Weigle (Member)

ABSTRACT

AN OPTIMIZED MULTIPLE RIGHT-HAND SIDE DSLASH KERNEL FOR INTEL® XEON PHI™

Aaron Walden
Old Dominion University, 2016
Director: Dr. Mohammad Zubair

Lattice quantum chromodynamics (LQCD) stands unique as the only computationally tractable, non-perturbative, and model-independent quantum field theory of the strong nuclear force. The computational core of LQCD is the Wilson Dslash operator, a nearest neighbor stencil operator summing matrix-vector multiplications over lattice points, whose performance is bandwidth-bound on most architectures. Reportedly, up to 90% of LQCD running time may be spent computing Dslash. In recent years, efforts have been made by researchers to optimize LQCD calculations for floating point coprocessor cards such as GPUs and Intel Xeon Phi Knights Corner (KNC), which boast powerful vector processing units. Most of these efforts in the area of Dslash have focused on single right-hand side solvers. This thesis will present two optimized Dslash kernels which simplify vectorization using multiple right-hand sides and traverse lattices using novel methods. The speedups resulting from these approaches will be explored in the context of KNC's architecture.

For those to whom nothing has yet been dedicated.

ACKNOWLEDGMENTS

This work is funded in part by a grant from Jefferson Lab and a Modeling and Simulation fellowship from Old Dominion University. I must also acknowledge Jefferson Lab for allowing me access to their computational resources.

TABLE OF CONTENTS

Page

APPENDICES

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

CHAPTER I

INTRODUCTION

Lattice quantum chromodynamics (LQCD) stands unique as the only computationally tractable, non-perturbative, and model-independent quantum field theory of the strong nuclear force. LQCD simulations are thus needed for areas of research at the frontiers of physics, including color confinement, exploration of the early Universe, and physics beyond the Standard Model. LQCD achieves tractability by discretizing our familiar space-time as a 4-dimensional hypercubic lattice. To simulate larger lattices with shorter spacing, ever-increasing computing power is required. The computational core of LQCD with Wilson fermions is the Wilson Dslash operator, a nearest neighbor stencil operator summing matrix-vector multiplications over lattice points, whose performance is bandwidth-bound on most architectures [1]. Reportedly, up to 90% of LQCD running time may be spent applying Dslash [2]. LQCD's computational intensity is such that it has inspired the design of supercomputers [3] and accordingly, a significant fraction of supercomputing cycles are devoted to LQCD simulation [4]. As of November 2015, 4 out of the 10 top supercomputers in the world [5] employ coprocessor cards, which offer resource-efficient floating point arithmetic and high memory bandwidth in comparison with general-purpose CPUs. Intel® Xeon Phi™ Knights Corner (KNC) is a line of such cards, and 2 of the top 10 supercomputers are equipped with thousands of KNC coprocessors [6], [7]. Significant research has been devoted to exploring KNC's potential to drive LQCD simulations [4], [1], [8], [9], [10]. The bulk of this research in the area of Wilson Dslash has involved single right-hand side (SRHS) solvers. In contrast, this thesis describes a C++ Wilson Dslash kernel applied to multiple right-hand sides (MRHS) in parallel on a single node. We employ vector register blocking and bandwidth optimization via improved lattice traversal to achieve significant speedups over previous implementations.

I.1 PROBLEM AND APPROACH

To maximize single-node performance, we must design an algorithm which fully utilizes hardware capabilities while satisfying the constraints of LQCD theory. In the case of KNC, the arithmetic intensity of Wilson Dslash is such that unvectorized code cannot reach anywhere near an adequate level of performance. This means we must devise a method of computation using KNC's 512-bit wide vector registers. In single right-hand side cases, vectorization is typically achieved by loading multiple, neighboring lattice points together with a *structure of arrays* approach [8]. This thesis describes two different approaches to vectorization using multiple right-hand sides instead of the multiple site method.

We must issue enough load instructions to saturate KNC's considerable memory bandwidth. This is known to require explicit software prefetching [11], and our approach in this thesis is to experiment with all manner of prefetching in combination with our other experimental parameters.

Finally, we must traverse the lattice in such a way as to maximize cache reuse of neighbor data, thereby minimizing memory bandwidth requirements. In this thesis we propose, implement, and evaluate three different approaches to lattice traversal. We experiment with these schemes over several different problem sizes and in combination with different implementation options and even other traversal schemes, when possible.

I.2 CONTRIBUTIONS

We make the following contributions with this research:

- Description and implementation of two different MRHS vectorization schemes for LQCD's Wilson Dslash operator on Xeon Phi™ Knights Corner

- Description and implementation of three different (two novel) lattice traversal schemes

- Description and implementation of several other program options designed to increase the performance of the Wilson Dslash operator on Xeon Phi™ Knights Corner

- Experimentation which explores the behavior of every possible combination of program options we have devised for real world single node problem sizes

- Explanation of the results we observe

I.3 THESIS ORGANIZATION

The remainder of this thesis is organized as follows. Chapter II introduces details of LQCD relevant to calculation of the Dslash operator. It also discusses the architecture of KNC relating to program optimization and outlines a theoretical performance model for our approach. Finally, Chapter II gives an overview of the state of the art of LQCD optimization. Chapter III explains what we refer to as the *base implementation*, which includes how and why the program is written as it is and introduces the experiments we conduct unrelated to lattice traversal. Chapter IV explains the default and novel ways of traversing lattice sites with which we experiment. Included is some discussion of the intuition behind and expected performance of our proposed methods. Chapter V discusses our experimental setup and observations before providing our best explanations of these results. Finally, we compare our results to similar Dslash implementations. Whenever the term *Dslash* is used without qualification in this thesis, we refer to the Wilson formulation.

CHAPTER II

BACKGROUND AND STATE OF THE ART

A full explanation of LQCD is well beyond the scope of this thesis. We instead narrow our discussion to the mechanics of the Dslash operator insofar as they constrain the design of our algorithm. We begin the chapter by detailing the lattice over which Dslash is applied. We explain the size and character of the input and output data. We then explain in detail the Dslash operator itself. We give a numerical model for the performance of MRHS Dslash relative to single RHS. We explain the architecture of Xeon Phi™ Knights Corner as it relates to our optimization efforts. Finally, we give an overview of the state of the art of Dslash optimization.

LQCD exposition in this chapter based on [2] and [8] unless otherwise noted.

II.1 LATTICE

The substrate in which Dslash application takes place is the aforementioned discretized 4-dimensional hypercubic lattice from which LQCD takes its name. QCD is a gauge theory of strong force interactions between quarks and gluons. One may imagine the lattice as a series of linked points. In LQCD, quark fields are represented by lattice points and gluon fields by the links. Lattice points are separated by some distance a (it is also possible to have different spacings in the same lattice). This parameter a represents the discretization, in some sense; as a approaches zero, LQCD approaches continuum QCD, which is not discretized. Computational physicists use the lattice discretization to numerically integrate path integrals using a Monte Carlo method. This introduces statistical errors in addition to the systematic errors introduced by nonzero lattice spacing. The need to achieve a desired precision in the presence of these errors is one motivation for the optimization of LQCD computations.

II.1.1 Sites

The number of sites on a lattice is given by $VOL = L_x L_y L_z L_t$, where L_μ is the number of points on the lattice in a particular direction. In this thesis, we refer to a dimension (x, y, z, t) generically by μ. A site is defined by its coordinates in the 4 directions $<x, y, z, t>$. To facilitate linear memory storage and traversal, we linearize lattice point coordinates with the following formula:

$$s = \frac{L_x}{2}\left(y + L_y\left(z + L_z t\right)\right) + \frac{x}{2} + \frac{VOL}{2}\left(\left(x + y + z + y\right) \mathbin{\&} 1\right)$$

Where & is the bitwise AND operator, which is used to test whether or not the coordinate sum is even or odd. Sites have 2 neighbors in each direction, for a total of 8. An important property of the lattice is that it is considered to *wrap around* in the sense of neighbors. Consider a lattice with L_x equal to 8. Then, the positive and negative x-direction neighbors of site $<7, 0, 0, 0>$ are $<0, 0, 0, 0>$ and $<6, 0, 0, 0>$, respectively.

II.1.2 Checkerboarding

To facilitate some mathematical tricks, we may divide the lattice into even and odd sites, where an even site means the sum of the direction indices is divisible by 2 (so, site $<0, 0, 0, 0>$ would be even). This colors the lattice in a checkerboard pattern, where an adjacent site is always of the opposite color. In this thesis, we illustrate checkerboarding by coloring sites red and black or gray and white.

II.2 SPINORS

An entity of primary interest in QCD is the Dirac equation, $M\psi = \chi$, which gives the propagation of quarks in a gluon field. Here, M is the Wilson-Fermion matrix, and ψ and χ are *spinors* $\in \mathbb{C}^{3\times4}$ which represent color (3 indices) and spin (4 indices). As these are complex matrices, there are a total of 24 degrees of freedom. For the purposes of our Dslash calculation, we can view ψ (along with gauge matrices) as input data and χ as output data. Each lattice point (site) is assigned its own $\psi, \chi \in \mathbb{C}^{3\times4}$. In this thesis, we sometimes use ψ and χ to refer to the entire collection of ψ_s,χ_s over a lattice, where ψ_s,χ_s refer to the spinors for some site s. Thus, when we refer to *neighbors* or *neighbor data* we refer to spinors (and usually to ψ).

II.3 GAUGE FIELDS

Returning to the Dirac equation, $M\psi = \chi$, we focus on the Wilson-Fermion matrix M, which gives the interactions between quarks and gluons. M is given by:

$$M = (N_d + m) - \frac{1}{2}D$$

Where N_d is the number of dimensions (4) and m is a parameter related to quark mass. The Wilson-Dslash term, D, is given by:

$$D = \sum_{\mu=1}^{N_d} ((1 - \gamma_\mu) \otimes U_s^\mu \delta_{s+\hat{\mu},s'} + (1 + \gamma_\mu) \otimes U_{s-\hat{\mu}}^{\mu\dagger} \delta_{s-\hat{\mu},s'}) \qquad (1)$$

We'll return to this equation to explain D in Section II.4. For now, we need only understand that the gluonic gauge fields formulate D, and consequently M, and are ultimately multiplied by ψ to form our output, χ.

The gauge fields U are SU(3) complex matrices $\in \mathbb{C}^{3\times3}$ which means they have several important properties. U is unitary, and its Hermitian conjugate is its inverse ($UU^\dagger = I$). The columns and rows of U form an orthonormal basis of $\mathbb{C}^3$. Of special interest to optimization of Dslash is the latter property. It allows us to compute the 3rd row or column of U given the first 2. This gauge *compression* allows us to trade bandwidth for FLOPs, which can be exploited to our advantage.

We can think of a complete set of gauge matrices over a lattice as a configuration. A property of LQCD computations is that it is perfectly useful to compute $M\psi = \chi$ for the same M but different ψ. This is the basis of our multiple right-hand side approach. We can compute several χ simultaneously, using the same U, which increases the arithmetic intensity of our computation, saving precious memory bandwidth.

There is a different gauge matrix for each site and forward direction, meaning the link along the positive direction of an (x, y, z, or t) axis. The gauge matrix for a backward link of site s is the Hermitian conjugate of the matrix for the forward link of the neighbor of s in that direction. Figure 1 shows an example planar slice of the lattice which should make this clear. In the figure, we show only the data relevant to applying Dslash to site s.

II.4 DSLASH OPERATOR

Recall from Equation 1 the Wilson-Dslash term, D. Here $\delta_{i,j}$ is the Kronecker-delta function and the indices i and j refer to sites:

$$\delta_{i,j} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

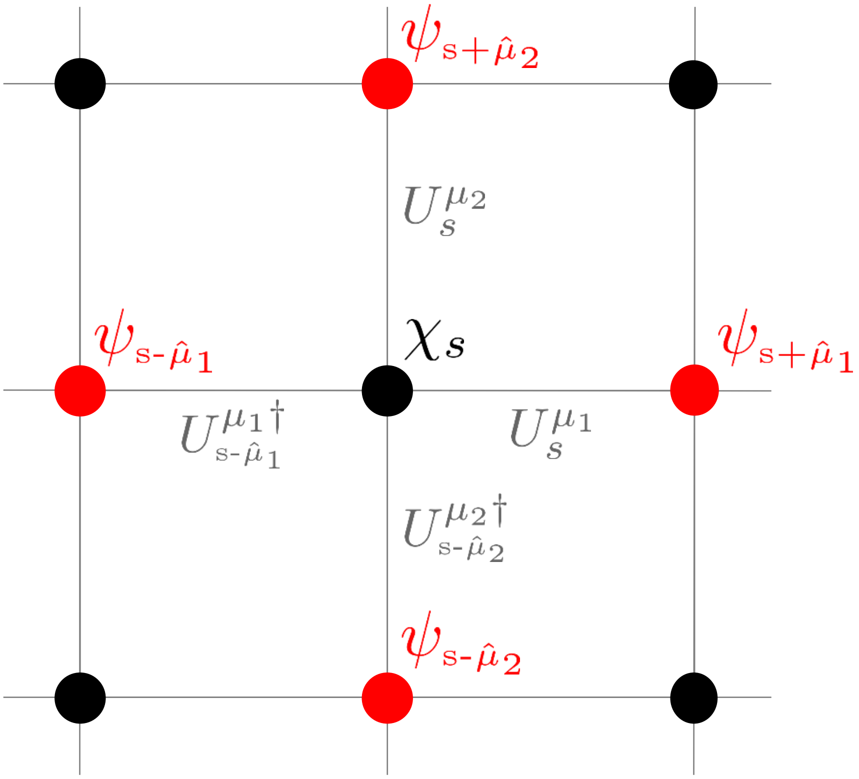


FIG. 1: A planar slice of the lattice in two arbitrary directions, μ_1 and μ_2. Here $s \pm \hat{\mu}$ refers to the neighbor site of s in the positive or negative direction μ. χ is shown for site s instead of ψ because these are the data of interest for computing Dslash for site s.

D is thus a large and sparse matrix. $(1 \pm \gamma_\mu)$ are special projector operators which act only upon spin indices. After a 4-spinor has been projected, it has only two independent spin degrees of freedom, which are combinations of the original 4 spins. We can take advantage of this fact by constructing the two independent degrees of freedom for a given projector, multiplying only these with the gauge matrix U, and then reconstructing the 4 spin components of the product. In other words, we can say $P_\mu^\pm = R_\mu^\pm Q_\mu^\pm$ where Q is referred to as the *spin projection* operation and R is called *reconstruction*. Since R and Q act trivially on color indices we can then write

$$U^\mu P_\mu^\pm \psi = U^\mu R_\mu^\pm Q_\mu^\pm \psi = R_\mu^\pm U^\mu Q_\mu^\pm \psi \tag{2}$$

Implementing the projector operators in this way that halves the number of matrix-vector multiplications needed.

We compute the reduced then reconstructed matrix product $RUQ\psi$ and sum the results over all $\pm\mu$ (8 directions, in the case we describe in this thesis). A high-level pseudocode description of our algorithm appears below. In this code, $\hat{R}$ refers to a version of the reconstruction operator above which acts on a half-spinor, the result of $UQ\psi$. In the remainder of the thesis, we will sometimes refer to $Q\psi$ as ψ'.

Algorithm 1 *WilsonDslash*

```
for all sites $s$ do
    $\chi_u \leftarrow 0$
    $\chi_l \leftarrow 0$
    for $\mu \leftarrow 1$ to $N_d$ do
        /* Compute forward direction */
        $v_a \leftarrow U^{\mu}_{s} Q^{+}_{\mu} \psi_{s+\hat{\mu}}$
        $\chi_u \leftarrow \chi_u + v_a$
        $\chi_l \leftarrow \chi_l + \hat{R}^{+}_{\mu} v_a$

        /* Compute backward direction */
        $v_a \leftarrow U^{\mu\dagger}_{s-\hat{\mu}} Q^{-}_{\mu} \psi_{s-\hat{\mu}}$
        $\chi_u \leftarrow \chi_u + v_a$
        $\chi_l \leftarrow \chi_l + \hat{R}^{-}_{\mu} v_a$
    end for
    $\chi_s = [\chi_u, \chi_l]$
end for
```

Computational physicists refer to *Dslash plus* and *Dslash minus.* In this thesis, we implement and refer to Dslash plus. Dslash minus is the Hermitian conjugate of Dslash plus and so they differ only in sign flips and matrix transpositions. In terms of arithmetic operations and memory access patterns, they are identical, and so our Dslash plus implementation results are applicable to a Dslash minus implementation created from our Dslash plus code.

II.5 MULTIPLE RIGHT-HAND SIDE PERFORMANCE MODEL

The approach of computing multiple right-hand sides is an established technique in LQCD research [12], [9]. 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 [1], we can analyze the expected speedup for different numbers of right-hand sides if we take the performance to be equal to:

$$\frac{\text{FLOPs}}{\text{byte}} \times \text{bandwidth}$$

Then, we need only divide the MRHS FLOPs/byte by the single right-hand side FLOPs/byte to compute speedup. We begin by defining variables:

$$F = \texttt{sizeof(float)}$$

$$S \text{ is the number of entires of } \psi \in \mathbb{C}^{3\times 4} \text{ (24)}$$

$$U \text{ is the number of entries of U} \in \mathbb{C}^{3\times 3} \text{ (18, 16, or 12)}$$

$$R_1 \text{ is the reuse factor for SRHS neighbor spinors}$$

$$R_N \text{ is the reuse factor for MRHS neighbor spinors}$$

$$N \text{ is the number of right-hand sides}$$

$$\text{SRHS\_FLOPPB} = \frac{1320}{8UF + (8 - R_1)SF + SF}$$

$$\text{MRHS\_FLOPPB} = \frac{1320N}{8UF + N((8 - R_N)SF + SF)}$$

$$\text{speedup} = \frac{MRHS\_FLOPPB}{SRHS\_FLOPPB}$$

$$= \frac{N(8UF + (8 - R_1)SF + SF)}{8UF + N((8 - R_N)SF + SF)}$$

If we take F to be 4, U to be 12 (assuming full compression, see Section III.7), S to be 24, and R_1 to be 7, said to be borne out in practice [13], we have:

$$\text{speedup} = \frac{576N}{384 + N(96(8 - R_N) + 96)}$$

In Figure 2, we plot the speedup equation for values of N on the x-axis and speedup on the y-axis with a curve for each value of $R_N \in \{0, 1, \ldots, 7\}$. We note that the higher the reuse factor, the greater the discrepancy between 8 and 16 RHS. We additionally note that R_N must be > 3 to expect any speedup, which may limit our

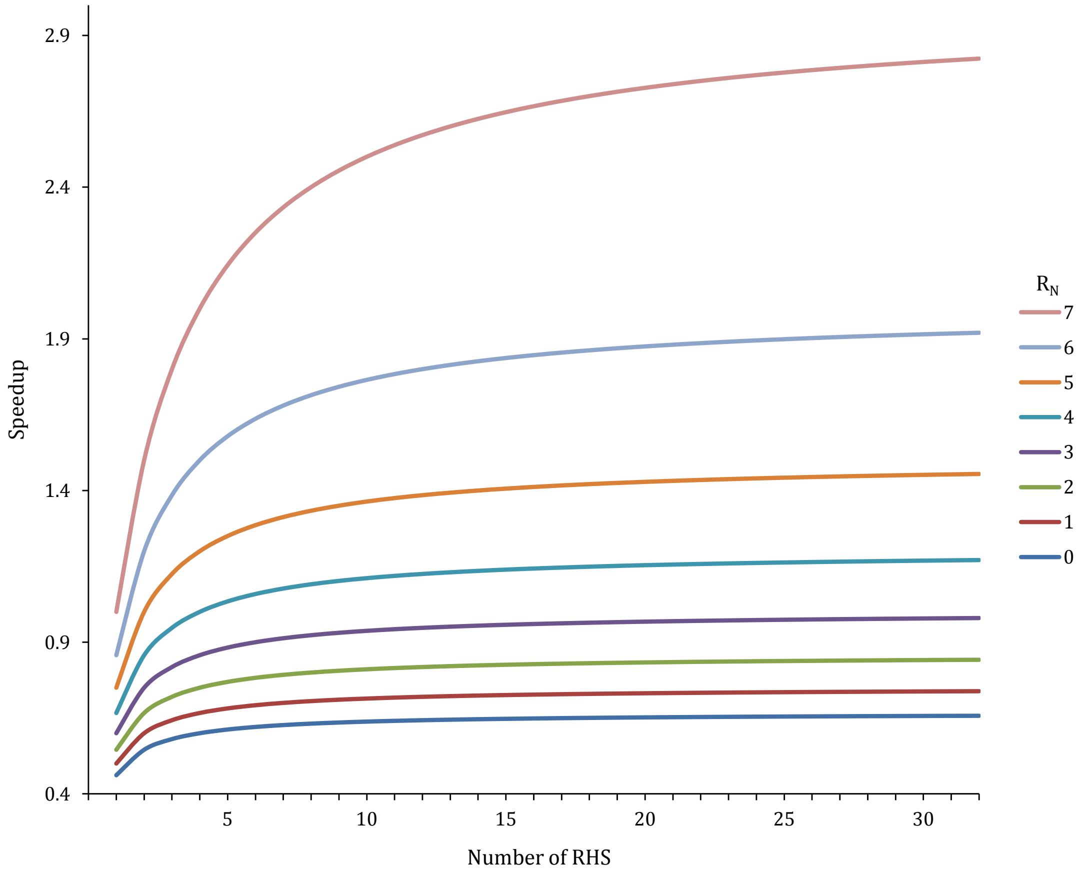


FIG. 2: MRHS scaling for values of N (number RHS) and R_N (neighbor reuse factor).

performance gains for lattices of medium to large size, where reuse may be difficult to achieve with spinor data scaling by N and a fixed size cache.

This performance model is based on [14].

II.6 XEON PHI™ KNIGHTS CORNER

Knights Corner is a (relatively) nascent line of many-core PCIe coprocessor cards in the Intel® Xeon Phi™ family. KNC cards are analogous to modern GPUs in the sense that they are massively parallel chips with high memory bandwidth suited for scientific computing applications. KNC differs from GPUs in that its many-core integrated architecture (MIC) consists of x86 compatible in-order cores with speeds of 1 to 1.238 GHz. One of its most touted features is the ability to run code written for general purpose x86 CPUs. In practice, however, code must be specifically crafted to take advantage of KNC's unique architecture. This combination of features sows a field which has become ripe for experimentation. Information in this section is taken from [15] unless otherwise noted.

Knights Corner chips feature up to 61 cores running at the aforementioned speeds. Each core possesses 32KB 8-way set associative L1 instruction and data caches and a 512KB unified L2 cache. Each L2 cache is joined through a bidirectional ring interconnect system, pictured in Figure 3. L2 caches achieve global coherence through connection to a series of 64 tag directories. They are capable of cache-to-cache transfers which bypass main memory. Each cache line is 64B. Evictions from cache occur according to the least recently used line. Notably, KNC is capable of explicit evictions from cache using the `_mm_clevict` intrinsic.

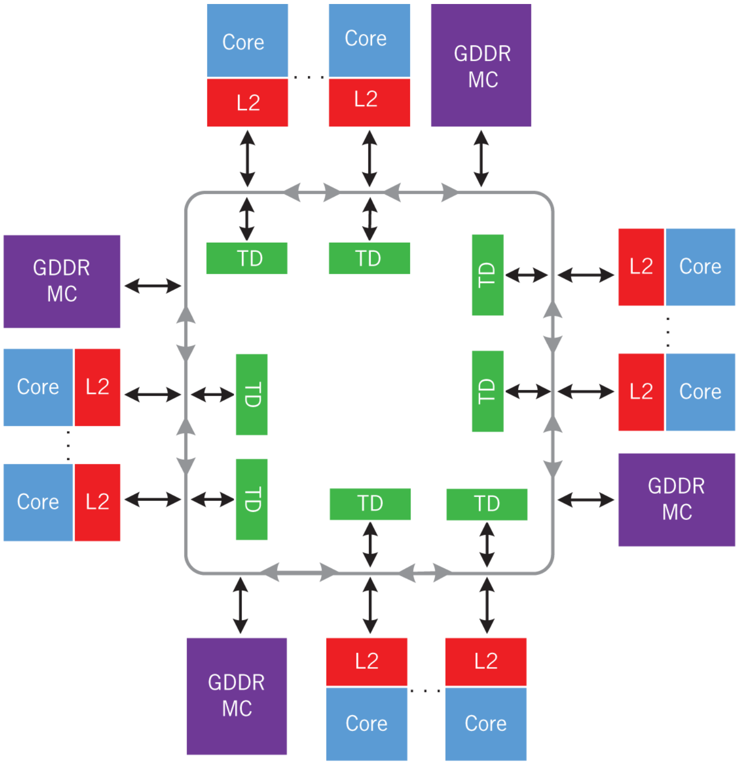


FIG. 3: Xeon Phi™ Knights Corner architectural overview.

A key feature of KNC is its vector processing unit (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. The primary motivation of our MRHS scheme is to fully exploit KNC's VPU in a way minimally restrictive to experimenters. To facilitate this, Intel® provides a set of C style functions they call intrinsics, which act directly on vector registers in an assembly-like way. KNC can only issue one vector instruction every 2 cycles. Thus, at least 2 threads are required to achieve full performance. On top of this, arithmetic instructions have a latency of 4 cycles, which means the full number of hardware threads is necessary to hide all latency where a kernel is arithmetically intense.

KNC coprocessors are equipped with up to 16 GB GDDR5 SDRAM. With 8 memory controllers capable of 5.5 GT/s , KNC has a peak theoretical memory bandwidth of 352 GB/s. As our operator is bandwidth-bound, memory performance is of great interest to us. In [11], researchers report read bandwidth up to 164 GB/s and write bandwidth up to 76 GB/s for a total observed bandwidth of 240 GB/s. Our own results echo these. It should be noted, however, that Dslash is much more reliant on read bandwidth and our performance will be constrained by that lower value. Additionally, KNC provides *streaming store* instructions which bypass cache when writing to memory. KNC is capable of hardware and software prefetches to increase memory bandwidth utilization. To our advantage, software prefetch instructions may be issued alongside other instructions as a consequence of KNC's instruction pipeline.

KNC coprocessors run an embedded Linux μOS which communicates with the host and runs native applications. When running an application of the card, it is conventional to leave a single core idle to handle these OS functions, lest program performance be affected.

Finally, applications for KNC may be run in either *offload* or *native* mode. Offload mode refers to applications which are run by a conventional CPU and specific sections of code are *offloaded* onto the card. We are primarily concerned with native applications which run directly on the card. Our focus is to optimize a very specific KNC kernel.

II.7 STATE OF THE ART

The state of the art of LQCD optimization is diverse. Kernels have been variously optimized for supercomputers, server CPUs (Xeon™, Opteron™), and coprocessors. We examine the approaches to each in turn.

In [16], the authors compose an LQCD solver for the BlueGene/L supercomputer. They acknowledge that the bulk of computation takes place when applying Dslash, and optimize accordingly. They hand-optimize Dslash, paying special attention to

BlueGene/L's hardware features, such as fused multiply-add instructions. They optimize the memory layout so that accesses are sequential. Some attention is given to site traversal, but the authors are more concerned with shared neighbor communication between nodes, something beyond the scope of this work. Overall, their approach is rather similar to our own, which is not surprising, given that the critical kernel is the same and that memory bandwidth growth is still being outpaced by growth of CPU speeds.

The authors of [17] use a different approach. They create a library which writes optimized assembly code for different supercomputing platforms, including the BlueGene/L. The authors highlight the fact that general-purpose C++ compilers do not optimally exploit the large numbers of registers available to RISC chips. Our own code relies heavily on register blocking for performance.

In a similar vein, the authors of [18] have written a code generator for their own C++ QCD software package, QPhiX [19]. The code generator outputs optimized vector intrinsics for modern architectures (AVX, AVX2, AVX512, SSE, KNC, etc) which execute the Dslash kernel and other parts of the LQCD solvers. These intrinsics are hidden under a layer of abstraction and plugged into QPhiX, which handles higher level functions like parallelization and cache-blocking. [20] experiments with QPhiX-codegen and verifies its strong performance on both CPUs and Xeon Phi™ KNC. Our own code is essentially a handwritten version of the KNC kernel generated by QPhiX-codegen.

In [8], the authors describe the implementation of an optimized Dslash for Intel® Xeon™ nodes. What is of interest is the *3.5D blocking* scheme it employs, the source of which is an earlier paper [21]. After implementing our cache-controlling traversal (see Section IV.2), we came to realize that the blocking strategy is the same as that used in [8], [21]. We were beaten to the punch by at least 6 years, apparently. Our own idea is still novel in that it attempts to implement the cache-controlling evictions, which, perhaps not coincidentally, emulate the *perfect LRU* which is assumed in [8].

There are at least several existing MRHS approaches in the area of LQCD solvers. [12] and [22] are a series of papers by several Japanese authors exploring a new method which is mainly a mathematical tweak to a known solver. The MRHS concerns they express are how the MRHSs affect the convergence of solutions. Though interesting, it is mostly unrelated to our own work. A more pertinent MRHS paper is [9]. In this paper the authors directly compare MRHS implementations of their Highly Improved

Staggered operator kernel for KNC and NVIDIA™ GPUs. They implement kernels for 1 to 8 right hand sides, employing *site fusion* grouping strategies when the number of RHS is smaller than the vector length (which is to say, always, in this case). They discuss register strategies and it seems likely that their 8 RHS approach is similar to our own. Devising schemes to implement 1–8 RHS must have taken considerable effort, as the vectorization is far from obvious. They achieve results similar to our naive traversal for 8 RHS, though we must bear in mind that this is a different kernel with a different arithmetic intensity. What's interesting about this paper are the GPU results. Without any complex blocking strategies, a K40 can reach 450 GFLOPS computing the operator on very large lattices (e.g. $48^3 \times 12$). This is perhaps not all that surprising, given the K40's much higher bandwidth.

There are numerous GPU implementations of Dslash and related LQCD solvers. Some take advantage of half precision calculations that result in double and single precision solution accuracy [23]. Though this is intriguing, such full solvers are beyond the scope of this thesis.

II.8 SUMMARY

In this chapter, we discussed the mechanics of the Dslash operator insofar as they constrain the design of our algorithm. We described the lattice over which Dslash is applied. We explained the size and character of the input and output data. We explained in detail the Dslash operator itself. We gave a numerical model for the performance of MRHS Dslash relative to single RHS. We explained the architecture of Xeon Phi™ Knights Corner as it relates to our optimization efforts. Finally, we gave an overview of the state of the art of LQCD optimization.

CHAPTER III

BASE IMPLEMENTATION

In this chapter we will discuss implementation unrelated to lattice traversal, including all options tested. We discuss in depth our vectorization schemes for 8 and 16 RHS, including our adaptation of these algorithms for KNC's hardware.

We should note that the implementation described herein is strictly for single precision, otherwise the descriptions for 8 and 16 RHS would not be apt.

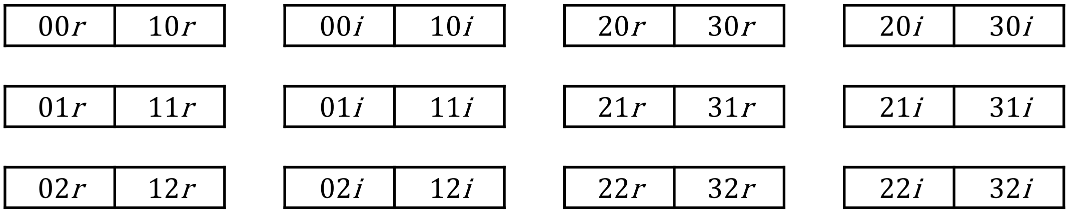
III.1 MULTIPLE RIGHT-HAND SIDE VECTORIZATION

In this section we discuss our scheme for exploiting KNC's vector processing unit. Recall from section Section II.6 that KNC's vectors are 512 bits in length, enough to hold 16 4-byte floats. Consequently, vectorization using 16 RHS is simple. Optimal vectorization using 8 RHS, however, presents several challenges, our solutions to which we discuss in Subsection III.1.1. We follow that with a discussion of the implementation of 16 RHS.

III.1.1 8 RHS

A multiple RHS approach is the computation of the Dslash operator for many independent input ψ simultaneously. Each operator acts upon a separate ψ, but shares the same gauge fields, or links, U (see Chapter II). This approach suggests a simple vectorization – namely, when a component of some ψ is used in computation, the vector lanes will be filled by the component in question belonging to each RHS. This leaves half of the vector register empty, unfortunately. To fill the remaining lanes, the obvious solution is to load them with a different component of ψ, as individual spinors (ψ) are made up of an even number of components. The problem is how to pair these components in such a way that extraneous operations are minimized and correctness is preserved.

The desired characteristics for such an 8 RHS vectorization are as follows:



$00r$	$10r$	$00i$	$10i$	$20r$	$30r$	$20i$	$30i$
$01r$	$11r$	$01i$	$11i$	$21r$	$31r$	$21i$	$31i$
$02r$	$12r$	$02i$	$12i$	$22r$	$32r$	$22i$	$32i$

FIG. 4: 8 RHS vectorization layout. Each component represents 8 4-byte floats, each corresponding to a different RHS.

1. Paired components should interact with other paired components at the same time.

   If a vector register v_0 contains the two components $[00r|10r]$ (i.e. there are 8 copies of 00r in the first 8 32-bit lanes, one from each RHS) and vector v_1 contains $[20i|30i]$, then, if the computation calls for $00r + 20i$, it should also call for $10r + 30i$ to avoid wasted operations.

2. Each component should appear in at most a single pairing.

   For example, if v_0 contains $[00r|10r]$, v_1 should not contain $[00r|20r]$.

3. When projected (Section II.4), the resulting pair should form a row of the projected matrix.

   Say, for example, a projection is calculated as follows: $00r_{proj} = 00r + 20i$ and $01r_{proj} = 10r + 30i$. Then, we must have vectors (say, v_0 and v_1) containing $[00r|10r]$ and $[20i|30i]$ or $[00r|30i]$ and $[20i|10r]$ so that $v_{proj} = v_0 + v_1$. This characteristic is necessary because computation of $U\psi'$ will require these components to be multiplied by the same entries of U.

Recall from Section II.4 that by transforming input ψ by a projection operator, we can (nearly) halve the number of arithmetic operations necessary to compute Dslash. Each direction of Dslash requires different combinations of ψ's components when projecting. See Appendix A for a full listing of the projection combinations.

Luckily, there exists a layout that satisfies the three desirable characteristics specified in this section. There is a minor caveat, however. Projection operations require a permutation or sign flip of half of a vector register. KNC provides vector intrinsics to permute across 256-bit lanes in this way. The layout is given in Figure 4.

III.1.2 16 RHS

KNC's vector length is 64 bytes or 16 4-byte floats. This completely 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.

III.2 REGISTER BLOCKING

KNC vector intrinsic instructions provide a level of assembly-like control over KNC's vector registers, which allows us to accumulate the upper and lower sums in registers before storing, ensuring there are no unnecessary writes to memory. We refer to this as a register blocking approach.

Recall from Section II.6 that KNC provides a set of 32 vector registers per hardware thread. A single hardware thread will apply Dslash to a single lattice point at a time, so we assume 32 vector registers will be available to our algorithm. In the following sections, we discuss how this constraint affects the design of our algorithm for the two different RHS implementations.

III.2.1 8 RHS Algorithm

For 8 RHS, the upper and lower sums occupy 12 vector registers (24 components, 2 components per register). The projected matrix occupies 6 registers. Components of the projection sharing a vector register are column-wise adjacent. Thus, they will multiply the same components of U when computing $U\psi'$. Then, we must broadcast each component of U to fill an entire vector register (recall from Section II.3 that gauge fields can be reused by every lattice). To put every component of U in a vector register, then, would require 18 registers, which brings the total over 32. Recall from Section II.4 that the lower sum components are some permutation of upper sum components. In the simplest case, we need to maintain a separate register which accumulates each addition to the upper sum (from each direction) so we can manipulate it before adding it to the appropriate lower sum component. We have (at least) these two options, then (with v representing a vector register, r and i indicating real and imaginary components, respectively):

- 1. Load all projection components (v_{proj} as appropriate).
  2. Moving row-wise (within a row), broadcast a component of U into v_u.
  3. Multiply $v_u v_{proj}$ for appropriate v_{proj} and accumulate in v_{a_r} or v_{a_i}.
  4. When finished with the row of U, add v_{a_r} and v_{a_i} to upper and lower sums appropriately.

  This approach requires $1\,(u) + 6$ (projections) $+2$ (accumulators) $= 9$ registers.

- 1. Load a row of projection components.
  2. Moving column-wise, broadcast a component of U into v_u.
  3. Multiply $v_u v_{proj}$ for appropriate v_{proj} and accumulate in some v_a (because we move column-wise in U, we need to keep an accumulator for every component of the upper sum).
  4. When finished with all U, add all v_a to upper and lower sums appropriately.

  This approach requires $1\,(u) + 2$ (projections) $+6$ (accumulators) $= 9$ registers.

We chose the first option for no reason in particular. We did not implement the second as it necessitates rewriting the entire program by hand. Algorithm 2 illustrates our chosen 8 RHS Dslash application for a single direction at the register level. In this pseudocode, v_X represents some subset of a thread's set of vector registers and v_X^i refers to some specific register in that set. Letters r and i always refer to the real and imaginary components, respectively, of some matrix. The function load_proj(. . .) loads all 6 projections into v_p and the details are unnecessary here.

Algorithm 2 *8RHSDIRSUM*

```
$v^p \leftarrow$ load_proj$(\dots)$
for $\rho \leftarrow 1$ to $3$ do                                   ▷ for each row of $U$
    for $c \leftarrow 1$ to $3$ do                                  ▷ for each col of $U$
        $v_u \leftarrow u^{\rho,c,r}$
        $v_a^r \leftarrow v_a^r + v_u v_p^{c,r}$
        $v_a^i \leftarrow v_a^i + v_u v_p^{c,i}$
        $v_u \leftarrow u^{\rho,c,i}$
        $v_a^r \leftarrow v_a^r - v_u v_p^{c,i}$
        $v_a^i \leftarrow v_a^i + v_u v_p^{c,r}$
    end for
    $v_{\chi_u}^{\rho,r} \leftarrow v_{\chi_u}^{\rho,r} + v_a^r$
    $v_{\chi_u}^{\rho,i} \leftarrow v_{\chi_u}^{\rho,i} + v_a^i$
    $v_{\chi_l}^{\hat{R}(\rho,\mu,r)} \leftarrow v_{\chi_l}^{\hat{R}(\mu,d,r)} + \tilde{R}(\rho,\mu,d,r)v_a^r$     ▷ $\mu \in \{\text{all directions}\}$
    $v_{\chi_l}^{\hat{R}(\rho,\mu,i)} \leftarrow v_{\chi_l}^{\hat{R}(\mu,d,i)} + \tilde{R}(\rho,\mu,d,i)v_a^i$     ▷ $d \in \{\text{forward, back}\}$
end for
function $\hat{R}(\rho,\mu,h)$                                      ▷ $h \in \{r,i\}$
    return appropriate $\chi_l$ index based on parameters
end function
function $\tilde{R}(\rho,\mu,d,h)$                                  ▷ $h \in \{r,i\}$
    return 1 or $-1$ based on parameters
end function
```

III.2.2 16 RHS Algorithm

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 and this requires 24, leaving only 8 for calculations. Looking at the simple options we used for 8 RHS, we can see that neither of these will work. For the first we require 12 registers to hold all the projection components and for the second we require 12 registers for temporary sum accumulation.

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

Algorithm 3 *16RHSDIRSUM*

```
for $c_u \leftarrow 1$ to 3 do                                        ▷ for each col of $U$
    $v_u \leftarrow$ load_u_row(…)
    for $c_\psi \leftarrow 1$ to 2 do                                 ▷ for each col of $Q\psi$
        for $h_\psi \leftarrow 1$ to 2 do                             ▷ $h \in \{r, i\}$
            $v_\psi \leftarrow$ proj$(Q\psi, \mu, d, c_u, c_\psi, h_\psi)$
            for all $v_u^{\rho,c_u,h_u}$ do                           ▷ 6 nums in col $c_u$, $\rho =$ row
                $v_{\chi_u}^{\rho,c_\psi,H(h_u,h_\psi)} \leftarrow v_{\chi_u}^{\rho,c_\psi,H(h_u,h_\psi)} + v_u^{\rho,c_u,h_u} v_\psi \hat{S}(h_u, h_\psi)$
                $i \leftarrow \hat{R}(\mu, \rho, c_\psi, H(h_u, h_\psi))$   ▷ saving space on page
                $v_{\chi_l}^{i} \leftarrow v_{\chi_l}^{i} + v_u^{\rho,c_u,h_u} v_\psi \tilde{R}(\mu, d, \rho, c_\psi, H(h_u, h_\psi))$
            end for
        end for
    end for
end for
function $\hat{S}(h_u, h_\psi)$
    if $h_u = i$ AND $h_\psi = i$ then
        return -1
    else
        return 1
    end if
end function
function $H(h_u, h_\psi)$
    if $h_u \neq h_\psi$ then
        return i
    else
        return r
    end if
end function
function $\hat{R}(\mu, \rho, c_\psi, H(h_u, h_\psi))$
    return appropriate $\chi_l$ index based on parameters
end function
function $\tilde{R}(\mu, d, \rho, c_\psi, H(h_u, h_\psi))$
    return 1 or $-1$ based on parameters
end function
```

require any intermediate accumulation registers. We can add directly to the upper and lower sums when computing $U\psi'$. Algorithm 3 illustrates the looping used to keep register use down to 31. The pseudocode is Byzantine in its indexing (it would be 4 pages long without it), so we offer a practical example to aid in understanding. For each direction, we add to χ^u_{00r} the real part of the complex dot product of the first row and column of U and $Q\psi$ (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 $00r$ of the accumulated sum from $01i$ of the lower, so:

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

This is how we go about *unrolling* the multiplication so we don't need intermediate sums. We can just compute the dot product twice, using `fmadd` and `fnmadd` where appropriate to account for the sign changes.

This approach results in extra instructions compared to the 8 RHS approach, but we can find no way to avoid this, given only 32 registers. Table 1 gives a breakdown of the intrinsic counts for the two approaches.

Intrinsic	8 RHS	16 RHS
`fmadd_ps`	168	698
`sub_ps`	52	48
`fnmadd_ps`	48	472
`mul_ps`	50	0
`add_ps`	124	49
`permutevar`	49(×1.5)	0
Total	515.5	1267

TABLE 1: Intrinsic counts for 8 RHS and 16 RHS.

If we double the total number of intrinsics for 8 RHS, the two approaches do the same amount of work. We also consider that `permutevar` instructions have 6 cycle latency, 50% more than arithmetic instructions [11]. In that case, 16 RHS performs 23% more *cycles' worth* of intrinsics, in some sense.

III.3 MEMORY LAYOUT

In this section, we describe how the values of U and ψ are stored in physical memory. Because KNC loads vector registers with 64-byte aligned chunks, our vectorization layout dictates the memory layout. As explained in Section II.1.1, each lattice point has a linearized index calculated from its coordinates. In the cases of both U and ψ, we store the data for each site in a one-dimensional array indexed by these linearized values. We access values in this array by casting its base address to a multi-dimensional array.

III.3.1 ψ

We require that values of ψ are grouped into 64-byte chunks in the order that they are expected to appear in vector registers (see Figure 4). Accordingly, we organize ψ first by site, then by paired components, with 8 different RHS values for each component. An illustration of the layout appears in Figure 5.

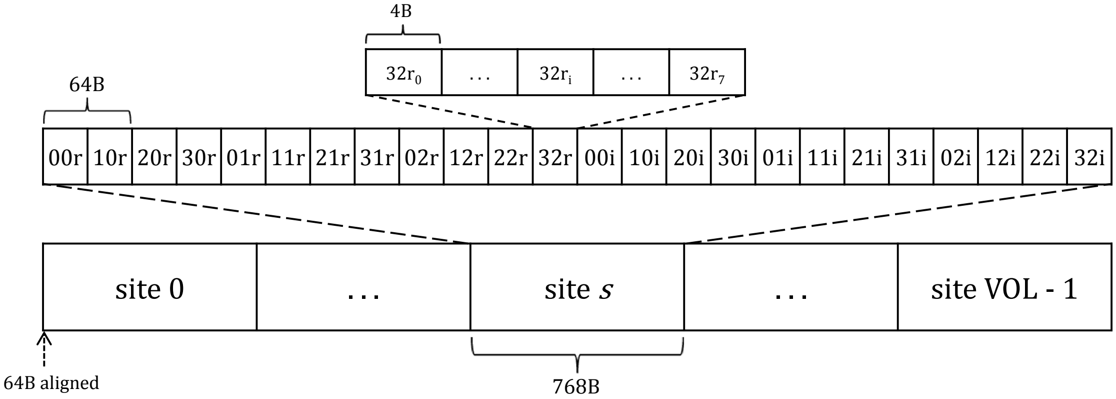

4B
$32r_0$
...
$32r_i$
...
$32r_7$
64B
00r 10r 20r 30r 01r 11r 21r 31r 02r 12r 22r 32r 00i 10i 20i 30i 01i 11i 21i 31i 02i 12i 22i 32i
site 0
...
site s
...
site VOL - 1
64B aligned
768B

FIG. 5: 8 RHS ψ in physical memory.

16 RHS ψ is stored analogously to 8, except that each component holds 16 floats, and the order of the components (per site) is arbitrary.

III.3.2 χ

χ is stored similarly to ψ in both the cases of 8 and 16 RHS, with one caveat. For 8 RHS, χ is not stored in the special pairs as it is in ψ, but in pairs of components that share the same row and type (real/imaginary). Thus, $[00r|01r]$ would be one such pair and together occupy 64 contiguous bytes of memory. This is a consequence of the algorithm used to calculate χ (see Algorithm 1). In a full solver, it would be necessary to store ψ and χ in the same layout, as χ will become a future input. We can slightly modify our algorithm to achieve this by using the KNC intrinsic `_mm512_mask_blend_ps` to restore the original pairings shown in Figure 4. As we hold all results in registers already, this adjustment should not be costly.

III.3.3 U

We only require that values of any single $U \in \mathbb{C}^{3\times 3}$ are stored contiguously. Thus, we store four gauge matrices at every index of U. These correspond to the forward links for the indexed site (see Section II.3). The storage scheme is much like that of ψ in Figure 5, but with four gauge matrices for every site index. U does not change with the number of RHS. Since values of U are broadcast to fill vector registers, their ordering in memory is arbitrary.

III.4 INDEX CALCULATION

To apply Dslash for some site, we need its linear site index to access ψ in memory and its coordinates to determine neighboring sites. We linearize neighbor coordinates (see Subsection II.1.1) to access neighboring ψ. These calculations can be expensive. If, however, the computation is memory bandwidth bound, the cost may be hidden. To test this hypothesis, we add a *shift table* option to the program. The shift table contains precomputed neighbor indices and, in the worst case, occupies less space than a single cache line, so its effect on memory bandwidth saturation should be minimal. In our experiments, we observe the effect of using the shift table instead of computing neighbor indices on the fly.

III.5 SYNCHRONIZATION BARRIERS

We parallelize the computation using the OpenMP library, binding OpenMP threads to hardware threads. We divide the linearized sites among the threads according to some scheme, we call this a lattice traversal. The default style, which we refer to as *default chunking*, of traversal uses a `#pragma omp parallel for` directive and loops over all lattice sites. OpenMP will break the sites into chunks and distribute them in numerical order to the different threads. For example, if there were 100 sites and 20 threads, thread 0 would process the first 5 sites, thread 1 the next 5, and so on. There will be a default OpenMP synchronization barrier after the `for` loop over sites.

Our experimental styles of traversal launch all threads with a `#pragma omp parallel` and manually assign a range of sites to apply Dslash over. Using this style, we explicitly call a synchronization barrier after a thread has called Dslash for its allotment of sites. This method of synchronization allows us to make use of a special barrier written for KNC (`Barrier_mic.h` [24]) to test its effects on synchronization latency. However, we have to spend FLOPs explicitly calculating site indices.

III.6 PREFETCHING

Achieving maximum memory throughput on KNC requires prefetching [11]. We cannot reliably know what sites are being processed by other threads at a given moment. A thread only has definite knowledge of what sites it will load, so we must prefetch using this information. We choose to prefetch into L1 ψ for the current direction before issuing loads and arithmetic intrinsics. If fetched any earlier, the data might spill from L1, as four threads share an L1 of only 32 KB.

A thread also knows what site it will process on its next call to Dslash. We can L2 prefetch all the data needed by the next call far in advance, as the size of L2 is 512 KB, enough to hold nearly 700 ψ for 8 RHS (ignoring that L2 is unified instruction and data).

We experiment with prefetching by selectively disabling the software prefetching of specific neighbors and by disabling L1 or L2 prefetching altogether. Hardware prefetching is left at its default setting.

III.7 GAUGE FIELD COMPRESSION

We can take advantage of the fact that the third column of any U is the cross product of the first two columns. Instead of storing all 18 values, we need at most 12 and can compute the rest. In the case of a bandwidth-bound computation, this seems advantageous.

We experiment with gauge matrices of sizes 18, 12, and 16. We refer to the act of reducing gauge matrices to hold 12 and 16 values as 12-compression and 16-compression, respectively. Though 16-compression would seem at first superfluous, it creates gauge matrices the size of a cache line. This may positively affect performance because backward neighbor U are not stored contiguously (though they can be if storing 8 U per site). Figure 6 illustrates physical memory with and without compression.

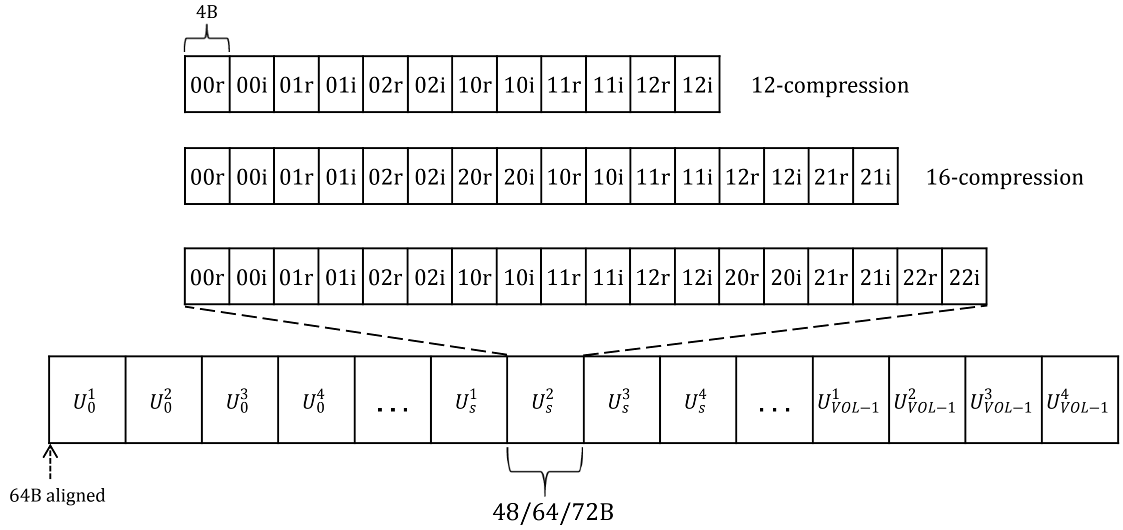

4B
00r 00i 01r 01i 02r 02i 10r 10i 11r 11i 12r 12i
12-compression
00r 00i 01r 01i 02r 02i 20r 20i 10r 10i 11r 11i 12r 12i 21r 21i
16-compression
00r 00i 01r 01i 02r 02i 10r 10i 11r 11i 12r 12i 20r 20i 21r 21i 22r 22i
U_0^1 U_0^2 U_0^3 U_0^4 ... U_s^1 U_s^2 U_s^3 U_s^4 ... U_{VOL-1}^1 U_{VOL-1}^2 U_{VOL-1}^3 U_{VOL-1}^4
64B aligned
48/64/72B

FIG. 6: U in physical memory with and without compression.

III.8 PAGE SIZE

For good performance, it can be necessary to enable 2MB memory pages on the coprocessor itself. For applications such as ours which execute in native mode, this must be done manually, either by adding specific code or through a library [25]. For the majority of (our) cases, the page size makes little difference, but it is necessary in one instance (see Chapter V).

III.9 SUMMARY

In this chapter, we detailed our implementation strategy for 8 and 16 RHS. We explained the challenges of filling vector registers completely presented by 8 RHS and how we overcame them. We described how we used a register blocking approach to store intermediate sums and avoid extra memory writes. We then discussed the challenges register blocking presents for a 16 RHS implementation due to the limited number of registers available to hardware threads and how we overcame them. We contrasted our 8 and 16 RHS approaches, which differ as a consequence of KNC's architectural characteristics. We detailed our approach to memory layout in context of the needs of our algorithm. Finally, we introduced several experimental aspects of our implementation: index calculation, synchronization barriers, prefetching, compression, and TLB page size.

CHAPTER IV

LATTICE TRAVERSAL EXPERIMENTS

In this chapter, we present experimental strategies for lattice traversal, which is the order in which threads process sites. Our general motivation when designing traversals is to minimize the number of ψ which are *reloaded*. To *reload* some ψ means to, during a single iteration, load ψ from main memory after having already loaded it from memory and evicted it from cache. In the typical case of a large lattice and small cache, reuse of ψ will be very low without a clever traversal strategy (we know this from our own experiments).

IV.1 LEXICOGRAPHICAL TRAVERSAL

We call the processing of sites in numerical order of their linearized site index (see Subsection II.1.1) lexicographical traversal. Assuming we are computing Dslash over even ($(x+y+z+t) \mid 2$) and odd sites separately and that even sites are processed first, then, beginning with site $<0,0,0,0>$, we traverse along the x direction, processing even values of x. After exhausting x, we increment y and traverse along x again, beginning at $<0,1,0,0>$ and processing odd values of x. When we reach the highest value of y, we increment z and the process repeats. Figure 7 illustrates the pattern of lexicographical traversal.

IV.2 CACHE-CONTROLLING TRAVERSAL

To improve upon naive lexicographical traversal, we propose a method to minimize main memory accesses by controlling the contents of L2 cache. KNC cores are capable of L2 cache-to-cache transfers, allowing cores to treat non-local L2 caches as a pseudo third level of cache. Among KNC's intrinsics is a function, `_mm_clevict`, which evicts a block from both levels of cache. Using this instruction, threads *make room* for their site's neighbors in cache by evicting sites that will not be loaded again in the near future. Though the latency of cache-to-cache transfers and memory-to-cache transfers is nearly the same (memory is reportedly 17% slower [11]), our hypothesis is that memory bandwidth is the limiting factor of Dslash performance for

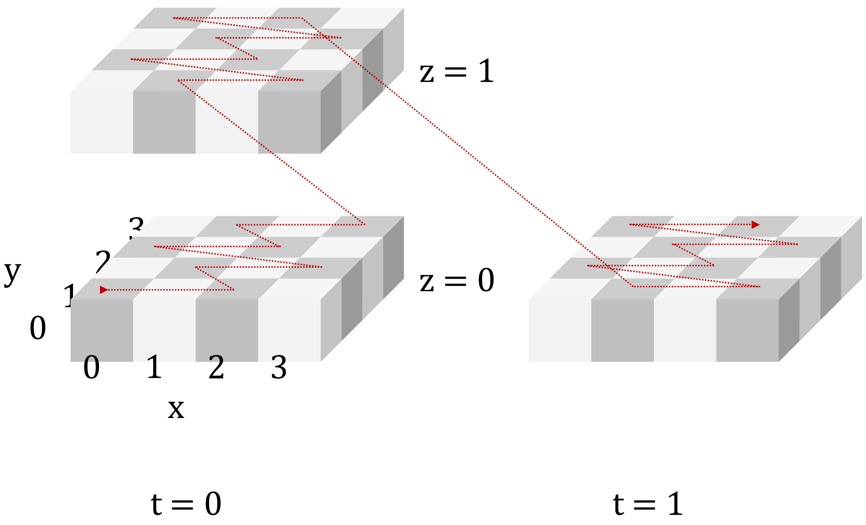


FIG. 7: An example of lexicographical site traversal of even sites beginning with site 0.

large lattices, so diverting memory transfers to cache may increase overall throughput.

To explain cache-controlling traversal, we think of the lattice as a series of *slices* in the time dimension. Each *time slice* is of size $L_x L_y L_z$ and there are L_t of them. We process each cube in order of increasing t, evicting sites from $t-2$ (wrapping around as necessary) as we go. To accomplish this, we synchronize the threads with a barrier after each slice, then divide the cube between the threads as in default chunking (see Section III.5). For example, assume L2 will hold three slices' worth of ψ, neglecting U, which can be loaded non-temporally. After processing the sites at t, t, $t+1$, and $t-1$ will reside in L2. Threads then synchronize and begin to process $t+1$. Before site $<i,j,k,t+1>$ loads a forward time neighbor from $t+2$, which will cause an uncontrolled eviction, we evict ψ from $<i,j,k,t-1>$, and so do not risk evicting a neighbor needed to process the sites of $t+1$.

Because thread scheduling is essentially random, the least recently used ψ cannot reliably be determined. Thus, when processing a site at t, it is possible to evict a site from t itself, $t-1$ or $t-2$. Cache-controlling traversal ensures the site is evicted from $t-2$. Edges evicted in this way will be reused, unlike most sites we evict, but they would need to be reloaded anyway once we reach the opposite end of the lattice.

For a direct performance comparison, let us consider a synchronized traversal along t as described in the preceding paragraphs but without deliberate evictions. We hypothesize that the difference in performance will depend on the amount of memory bandwidth saved by increasing cache-to-cache transfers and thus decreasing main memory traffic. As stated above, an uncontrolled eviction may remove a block containing data from t, $t-1$, or $t-2$. We consider evictions from $t-2$ desirable and others undesirable. We can conduct a simulation of our hypothetical traversal and determine the rate at which undesirable evictions are made. Table 2 shows the relationship between lattice size and rate of undesirable evictions. Interestingly, the rate changes with lattice volume, but not in a wholly predictable way. This result may be worth further inquiry.

Lattice Volume	% Undesirable Evictions
8^4	31.5
16^4	16.3
24^4	10.6
32^4	12.7

TABLE 2: Percentage of undesirable evictions in simulations traversing lattices of various volumes.

For our cache-controlling traversal to pay off, the amount of bandwidth saved must increase effective memory throughput enough to overcome the cost of `_mm_clevict`, additional synchronization barriers, and the index calculation for the evicted site. Unfortunately for our scheme, the percentage of evictions which are undesirable decreases significantly for larger lattices.

IV.3 INTERLEAVING

By default, OpenMP divides the lattice into congruent chunks of lexicographically contiguous sites and assigns one chunk to each thread, which is traversed lexicographically. Recall that each core supports four hardware threads. If the lattice is large enough, even two adjacent threads will never share a neighbor. In other words, by the time thread 1 loads a site with a neighbor in chunk 2 (traversed by thread 2), that site will already have been evicted from L2. Figure 8 contrasts the two traversal strategies. Note that although the figure may suggest that interleaved sites are traversed in y and then x, this is not the case. Each thread still traverses in order of increasing x, y, z, then t.

We estimate that, for large lattices (e.g. 32^4), interleaved traversal will result in increased cache hit rates, as the size of the cache is effectively quadrupled compared to the default case. Simulations predict a gain in L2 hit rate of over 10% for 32^4.

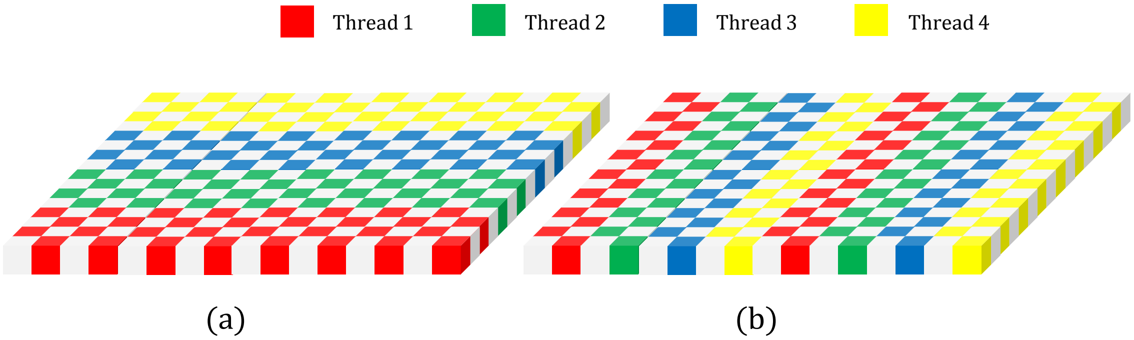


FIG. 8: **(a)** Default chunking. Threads process their chunks lexicographically. **(b)** Interleaved traversal. Threads alternate sites, but they are still traversed lexicographically.

IV.4 SUMMARY

In this chapter, we described three proposals for lattice traversal. We described a lexicographically chunked traversal equivalent to a `#pragma omp parallel for` loop. We then proposed a novel traversal technique designed to maximize the efficiency of KNC's interconnected L2 caches. Finally, we described a traversal which interleaves threads bound to the same core to take advantage of their shared cache.

CHAPTER V

RESULTS AND DISCUSSION

In this chapter, we describe the experimental procedure used to test our kernel and discuss the results thereof. For aesthetic purposes, numerical values appearing in this section have been rounded to the nearest integer toward zero.

V.1 EXPERIMENTAL SETUP

To evaluate our different approaches to Dslash implementation, we test every combination of the following options (refer to Chapter III for descriptions):

- Number of RHS (8/16)
- Lattice size ($8^4/16^4/24^4/32^4$)
- Shift table (ST/default)
- Thread interleaving (interleaving/default)
- Compression (12/16/default)
- Cache-controlling traversal (CCT/default)

These 192 different combinations make up the main experiment which generates the majority of our analysis. Before performing these tests, we evaluate `Barrier_mic` versus the OpenMP synchronization barrier and test the effects of prefetching on the various lattice sizes. We proceed in this way because the results are quite clear in these two cases and it allows us to reduce the number of results we present to a manageable number.

We run each program for a number of iterations based on lattice size (5000/500/100/50 for $8^4/16^4/24^4/32^4$, respectively), recording the run time of the iterations (excluding the setup of ψ and U), and divide the number of FLOPs performed ($1320 \times \#\text{RHS} \times \frac{VOL}{2}$) by this time. The first few iterations are thrown out, which, to the best of our knowledge, is the convention when timing for performance. We perform a number of such trials based on lattice size (20/15/10/5) and take the average to be our result. One may wonder why we do not simply perform more iterations and this is because we observe a variance in the GFLOPS which is inexplicably unchanged by adding iterations to the trials. We use Intel® Vtune™ to measure memory bandwidth.

Our code is compiled by the Intel® C++ compiler version 16.0.0 and run in native mode, directly on the KNC card. In addition to the experimental options, every test program uses the following optional switches (required switches e.g. `-mmic -openmp` are omitted):

```
-O3 -no-opt-prefetch -fno-alias -mcmodel=large
```

The machine on which we test is equipped with a Xeon Phi™ KNC 7120P which has a theoretical peak performance of 2.4 teraFLOPS (single precision) and 352 GB/sec memory bandwidth. Xeon Phi™ KNC 5110P read bandwidth is reported to be 164 GB/s by the STREAM benchmark [11], and our own testing confirms this result. The 7120P's bandwidth should be within 10% of the 5110P, as Intel® gives their max memory bandwidth as 352 and 320, respectively.

We run all tests with 4 threads per core to minimize the latency of 4 cycle arithmetic instructions and 60 cores, leaving the last to handle OS functions. We find that in all tested cases these settings produce the best results.

We ensure correctness by first calculating a checksum for a particular lattice size using QDP++ [26]. We compare this sum to our own output.

V.2 RESULTS

V.2.1 Prefetching

We begin our presentation of the experimental results by discussing L1 and L2 prefetching.

The results here are clear, save a single caveat we discuss below. L1 prefetching improves performance without exception, and dramatically so in the case of 8^4. L2 prefetching improves performance dramatically for the bandwidth-bound larger lattices of 24^4 and 32^4. L2 prefetching is not required in the case of 8^4, as the entire lattice is L2 resident. There is a caveat to this, however. Disabling L2 prefetching improves performance for 8^4 lattices only when 2MB pages are enabled (see Section

VOL	8^4		16^4		24^4		32^4	
RHS	8	16	8	16	8	16	8	16
L1P	651	708	419	473	337	375	346	387
No L1P	571	565	362	389	298	325	311	329
L2P	582	618	419	473	337	375	346	388
No L2P	651	708	300	327	228	233	218	244

TABLE 3: L1 and L2 prefetch experimental results. Values are in GFLOPS. The results compared here are the fastest case for each lattice size (see Table 5).

III.8). If huge pages are not enabled, L2 prefetching strongly improves performance for 8^4, but overall performance is significantly lower.

Experiments that follow can be assumed to use the optimal prefetching strategy based on lattice size.

V.2.2 Barriers

The test of synchronization barriers yields another clear result.

VOL	8^4		16^4		24^4		32^4	
RHS	8	16	8	16	8	16	8	16
`Barrier_mic`	651	708	419	473	337	375	346	387
OpenMP	520	630	409	467	323	371	338	383
difference	131	78	10	6	14	4	8	4

TABLE 4: Comparison (in GLFOPS) of the effects of two different barriers on results. `Barrier_mic` is a specialized barrier designed for KNC. OpenMP is the default OpenMP synchronization barrier. Difference is positive when `Barrier_mic` is higher.

The results here are as one would expect. Barrier length is sufficiently short that choice of barrier has little effect on results for lattices larger than 8^4, but a profound effect on lattices of that size, as the barrier time is large relative to iteration time. Similarly, the effect of a slower barrier is doubly negative for 8 RHS, as the amount of work done between barriers is half that of 16 RHS. `Barrier_mic` is intended to be a *fast* barrier for KNC and it succeeds in this capacity.

Experiments that follow can be assumed to use `Barrier_mic` exclusively.

V.2.3 Main Results

With barriers and prefetching out of the way, we present the main results in

Table 5. Build options appear on the left side of the table. There are 4 categories of build option: shift table, interleaving, compression, and cache-controlling traversal. Presence of an *S* indicates the shift table was used. Letter *i* indicates interleaving was used. A *12* indicates 12-compression and a *16* 16-compression. *C* indicates cache-controlling traversal. Absence of an indicator for any category implies the default value (see Chapter III). The maximum values for each lattice size and RHS value are indicated in bold and in the final row of the table.

In the following sections we attempt to explain the observed results for each category.

VOL	8^4		16^4		24^4		32^4	
RHS	8	16	8	16	8	16	8	16
Si12C	155	226	313	352	325	341	317	357
Si12	521	638	342	368	315	354	284	292
Si16C	170	243	332	372	335	354	338	379
Si16	614	**708**	357	390	328	**375**	294	305
SiC	160	232	334	367	327	343	314	365
Si	**651**	694	339	373	315	357	275	288
S12C	135	178	303	357	312	331	290	276
S12	495	605	383	439	314	327	258	246
S16C	149	191	336	374	**337**	349	314	284
S16	568	670	411	464	336	340	278	252
SC	142	181	329	374	310	341	283	279
S	600	656	398	462	307	328	248	239
i12C	152	225	301	348	284	317	314	355
i12	495	608	334	367	293	337	285	290
i16C	167	247	334	369	314	328	**346**	**387**
i16	581	703	363	393	319	354	299	303
iC	164	236	332	365	307	327	323	373
i	625	671	343	378	307	348	281	295
12C	134	176	298	349	283	315	288	273
12	466	577	377	434	290	320	262	243
16C	146	194	335	380	310	323	318	285
16	549	668	**419**	**473**	315	331	281	255
C	141	184	332	371	297	329	293	279
default	579	640	405	463	300	326	255	235
max	651	708	419	473	337	375	346	387

TABLE 5: Main experimental results. The option key is as follows: *S*=shift table, *i*=interleaving, *12*=12-compression, *16*=16-compression, *C*=cache-controlling traversal. Absence of a key category implies a default value (see Chapter III).

V.2.4 Compression

The effects of compression are easily understood, for the most part. 16-compression ensures gauge matrices are the size of cache lines, so never is extra data loaded. Use of uncompressed and 12-compressed gauge matrices always results in superfluous memory traffic due to the layout of U. There are 4 forward links associated with the site being processed. These are stored contiguously. Backward links are indexed by the backward neighbor, which is different for each direction. This results in extra traffic for every load of a backward link. In the uncompressed case, this results in 32 bytes of extra data loaded from the forward links and 56 bytes per backward link, for a total of 200 bytes per site. This adds 10% memory traffic overhead (for 16 RHS). In the 12-compressed case, we're required to load an entire cache line for each access, so while this does not hurt for the forward links (because 12×4 is a multiple of 16, the number of floats in a cache line), it costs 16 bytes per backward link, for a total of 64 bytes. This means the total amount of data loaded in the 12-compressed case is equal to the 16-compressed case, but 12-compression introduces additional arithmetic instructions in the calculation of the extra 4 gauge matrix entries, which is why 12-compression fails to beat 16-compression in any case we tested. This effect can be mitigated by storing back-link U contiguously with forward U in the order of access, but this doubles the amount of data stored and is not the approach that we chose.

In the case of 8^4 we see a different pattern. Uncompressed wins for 8 RHS and 16-compressed wins for 16 RHS. In the case of 8 RHS, this is seemingly because the lattice is cache-bound and we have no reason to increase the arithmetic intensity. In the case of 16 RHS, we surmise the reason for 16-compression's victory is that the lattice in this case is very close to busting L2, and the 10% overhead of uncompressed gauge matrices is enough to push it over, causing the drop in performance observed for lack of compression.

VOL	8^4		16^4		24^4		32^4	
RHS	8	16	8	16	8	16	8	16
12	466	577	377	434	290	320	262	243
16	549	668	419	473	315	331	281	255
default	579	640	405	463	300	326	255	235

TABLE 6: Compression comparison (GFLOPS).

V.2.5 Interleaving

There are consistent patterns observed across our tests of thread interleaving. The patterns are consistent for a particular lattice, however there is no clear trend in the data based on lattice size. Table 7 shows the average change in GFLOPS by lattice size and traversal strategy. Because CCT changes access patterns, there is an influence on the performance of interleaving (which also changes access patterns). Use of the average difference is appropriate here because the trends are consistent across lattice sizes and traversal strategies. There are no swings concealed by considering the average.

VOL	8^4		16^4		24^4		32^4	
RHS	8	16	8	16	8	16	8	16
C	20	50	2	-5	7	3	27	90
default	38	34	-53	-77	2	25	22	50

TABLE 7: Average change in GFLOPS for interleaved threads versus non-interleaved by lattice size and traversal strategy. Here, default refers to all non-CCT results.

Based on the results in Table 7, we conclude that the effect of interleaving is based on access pattern, but not necessarily predictable, given a lattice and traversal strategy. The idea behind interleaved traversal is to increase locality by having threads that share a cache work in the same area, where spatial neighbors are shared. Why this strategy should be so effective for small and large lattices, but ineffective to detrimental for medium-sized lattices, is not clear. Additionally, we see that CCT has a pronounced effect on the results of interleaving, lending more credence to the idea that interleaving results are based on access pattern. We also observe a pattern where interleaving has a stronger influence, positive and negative, on 16 RHS. Again, this behavior is mysterious. Further discussion is devoted to the confluence of interleaving and CCT in Subsection V.2.8.

V.2.6 Shift Table

Use of a shift table for neighbor indices should increase memory traffic and provide greater speedup to lattices which are not bandwidth-bound. We find this to be the case with one exception, 24^4. There is a rather large bump in performance for lattices of size 24^4 for unknown reasons. The effect is increased by CCT. Why this should be the case is not clear.

Use of a shift table does not significantly alter speeds for 16^4 and 32^4, which is expected. In the case of 8^4, the shift table predictably increases speed modestly as we desire lowered arithmetic intensity in this cache-bound region. Shift table usage does nothing for 8^4 CCT, but speeds there are presumably flattened by barriers.

VOL	8^4		16^4		24^4		32^4	
RHS	8	16	8	16	8	16	8	16
C	1	-1	2	2	25	20	-4	-2
default	25	17	-1	-2	15	10	-4	0

TABLE 8: Average change in GFLOPS for shift table versus on-the-fly index calculation by lattice size and traversal strategy. Here, default refers to all non-CCT results.

V.2.7 Number of RHS

We expect 16 RHS to outperform 8 RHS in every case. This is what we find, but the situation is not so simple. We compare the two implementations, splitting the table into interleaved and non-interleaved traversal in Table 9.

VOL	8^4	16^4	24^4	32^4		8^4	16^4	24^4	32^4
Si12C	71	39	16	40	S12C	43	54	19	-14
Si12	117	26	39	8	S12	110	56	13	-12
Si16C	73	40	19	41	S16C	42	38	12	-30
Si16	94	33	47	11	S16	102	53	4	-26
SiC	72	33	16	51	SC	39	45	31	-4
Si	43	34	42	13	S	56	64	21	-9
i12C	73	47	33	41	12C	42	51	32	-15
i12	113	33	44	5	12	111	57	30	-19
i16C	80	35	14	41	16C	48	45	13	-33
i16	122	30	35	4	16	119	54	16	-26
iC	72	33	20	50	C	43	39	32	-14
i	46	35	41	14	default	61	58	26	-20
avg	81	34	30	26	avg	68	51	20	-18

TABLE 9: Difference between 16 RHS and 8 RHS (GFLOPS). A positive value indicates a higher speed for 16 RHS. The left table shows interleaved traversals. The right, default.

For a lattice of size 8^4, barrier length dominates and 16 RHS outperforms 8 RHS accordingly. At this size, the entire lattice fits into L2, so reuse is not a strong factor, as 8 RHS 8^4 is already busting L1. We see a sharp reduction in 16 RHS advantage after 8^4. Barrier length matters little even at 16^4 (see Subsection V.2.2), but reuse factor is dropping for 16 RHS as lattices grow in size. Reuse is so low at 32^4 non-interleaved that 8 RHS outperforms 16 in every case. Recall that even if reuse were similar, 8 RHS has an advantage in instruction efficiency (see Subsection III.2.2). However, interleaving increases reuse for 32^4 so powerfully that the performance of 16 RHS overcomes the deficit and even surpasses 8 RHS (though not by much, except in the case of CCT, which is discussed in depth in Subsection V.2.8).

V.2.8 Cache-controlling Traversal

The most important result we have to discuss is that of cache-controlling traversal. Recall (from Section IV.2) that our intention with CCT was to explicitly ensure that sites evicted from cache would be those not needed by the sites currently being processed. To do this, it was necessary to both explicitly control the contents of cache by using an eviction instruction, `_mm_clevict`, and to implicitly control do so by forcing threads to work together on sequential subsections of the lattice.

It turns out that only implicit control is necessary to increase performance. The first tests of CCT showed a huge performance gain for 32^4. We realized that we had in fact introduced two different variables, the explicit and implicit control. Explicit evictions cannot be done without sequential sublattice traversal, so the only test to perform was disabling the explicit evictions and leaving the controlled traversal. Doing so further increased performance (by 10 GFLOPS). The evictions were only wasting CPU cycles.

Table 10 shows the average change in speed by lattice size and number of RHS. Results are consistent other than 24^4 8 RHS, where there are a few small gains but mostly small losses.

VOL	8^4		16^4		24^4		32^4	
RHS	8	16	8	16	8	16	8	16
C	-410	-443	-49	-52	0	-8	36	54

TABLE 10: Average change in GFLOPS when employing cache-controlling traversal.

For small lattices, CCT's extra barriers cripple performance. At 16^4, it's an open question whether or not the barriers or the change in access patterns (or both) degrade performance. We performed a test, including a single barrier at half volume for a lattice of 16^4 16 RHS, which similarly degraded performance by approximately

50 GFLOPS. This is a strong indication that 16^4 16 RHS is very sensitive to access pattern.

For 24^4, little change in speed occurs when using CCT. Performance for 32^4, however, is dramatically increased, more so when combined with thread interleaving, even eliciting a synergistic response. Interleaving increases 32^4 16 RHS by 60 GFLOPS and CCT alone increases performance by 44 GFLOPS. Together, performance is increased by 152 GFLOPS. Because the only variable that could be changing here is reuse, we must conclude that interleaving and CCT together greatly increase reuse by altering the site access pattern. This makes intuitive sense, as CCT forces threads to work *closer* to each other than they would otherwise, increasing the potential for neighbor reuse via cache-to-cache transfers and standard cache behavior. We can also view CCT as resynchronizing threads 32 times per iteration. For a larger lattice, the chance for interleaved threads to desynchronize due to the random nature of thread scheduling is increased. CCT puts a bound on how far out of synchronization threads can fall.

We are lucky, in some sense, that the performance gains seen by CCT are not dependent on `_mm_clevict` because this instruction is (for Xeon Phi™) exclusive to KNC and nowhere to be found in the next generation of KNC's intrinsic set (AVX/2/512). In light of the failure of explicit eviction, we should perhaps rename CCT to reuse-controlling traversal or synchronization-controlling traversal, but we prefer the ring of the original name.

V.3 PERFORMANCE COMPARISON

It is difficult to directly compare our work to most of the published kernels in Section II.7 because their kernels and solvers are generally of different arithmetic intensity ([9], for example). We can, however, compare directly to soon-to-be published results from an optimized a 16 RHS Dslash kernel using QPhiX and its code generator [27]. We can also compare directly to the results of a single RHS Dslash [13].

We observe speedups of at least 20% over the best previous performance in all cases. QPhiX 16 RHS does not implement register blocking, which gives the compiler control of how register spilling occurs (our implementation should not spill). For larger lattices, we presume this to account for the difference in performance between the two implementations, which are otherwise very similar at the KNC intrinsic level.

VOL	Our 16 RHS	QPhiX + CG 16 RHS	Single RHS
8^4	**708**	411	low
16^4	**473**	402	251
24^4	**375**	306	255
32^4	**387**	282	315

TABLE 11: Results comparison (GFLOPS).

We see such a large performance difference at 8^4 because our code places the OpenMP thread spawn outside of the iteration loop, so the QPhiX+CG implementation pays the cost of the thread spawn for every iteration.

V.4 SUMMARY

In this chapter we described our experimental setup and results before offering our interpretation thereof. We first presented experiments in which we determined the optimal use of prefetching and synchronization barriers. We then presented the results of a series of 192 different experiments using our programs. We described the effects of gauge compression on various lattices and explained the behavior we observed. We determined that compression into a size equal to a cache line is ideal unless all U are stored contiguously, which necessitates double the data stored for U. We described the effects of interleaving and their seemingly unpredictable nature. We presented the differences between the results of 8 RHS and 16 and noted that they were essentially as predicted. We described the results of our cache-controlling traversal. We discovered that the evictions were unnecessary and actually a hindrance to performance. We described how CCT and interleaving used together synergize to greatly increase data reuse. Finally, we compared our performance to similar Dslash implementations and showed that our methods resulted in strong speedups in all tested cases.

CHAPTER VI

CONCLUSION AND FUTURE WORK

LQCD simulations remain a computation of profound importance. Precision improvements to LQCD calculations that will require exascale computing have been identified [28]. Thus, the need for more efficient algorithms is clear. Performance improvements like those achieved in this thesis will only increase in significance as the gap between CPU and memory speeds continues to grow. In this thesis we have contributed the following:

- Implemented two novel register blocking algorithms for 8 and 16 RHS Dslash kernels
- Achieved speedups of 1.2 to 1.87 in real world single node lattice sizes
- Showed that with 4 U stored per site, gauge compression should be to chip cache line size
- Reported the effects of numerous variables on KNC Dslash performance
- While exploring these variables, we raised a number of interesting questions (see future work below)

In the future, we plan to address the following:

- An expanded implementation of CCT which includes traversal and eviction based on cache size
- CCT techniques may fail for lattices with relatively large spatial dimensions. For example, $48^3 \times 16$ is not much larger than 32^4, but performance is reduced. We can retool CCT to take advantage of a short time dimension.
- Expand the kernel beyond a single node
- Explore new traversal experiments. For example: two chunks allocated to adjacent (in number) cores might traverse in opposite directions and *meet* in the middle. Another example is a more advanced form of CCT which aims to avoid reloading of ψ (almost) entirely

- CCT has hinted at a lower bound for cache size needed to completely avoid reloads of ψ, in the future we intend to prove this bound analytically

The following questions were raised during our experiments and we plan to answer them:

- Why does the CCT percentage of undesirable evictions change based on lattice size?
- What interaction is there between 2MB pages and prefetching for 8^4?
- If it is true that access pattern is the determining factor which explains many of our performance results, how exactly do these patterns change performance?
- Why is thread interleaving detrimental for lattices of size 16^4 specifically?
- Why is shift table use surprisingly effective for 24^4?
- Why does CCT not benefit 24^4, which should be bandwidth-bound and in need of extra reuse from locality?

REFERENCES

[1] 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*, ser. SC ’14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 69–80. [Online]. Available: http://dx.doi.org/10.1109/SC.2014.11

[2] R. Gupta, “Introduction to Lattice QCD,” arXiv:hep-lat/9807028. [Online]. Available: http://arxiv.org/abs/hep-lat/9807028

[3] Lawrence Livermore National Laboratory, “Quark Theory and Today’s Supercomputers: It’s a Match,” *Science & Technology Review*, pp. 11–16, January/February 2008.

[4] A. Diavastos, G. Stylianou, and G. Koutsou, “Exploring Parallelism on the Intel® Xeon Phi™ with Lattice-QCD Kernels.” [Online]. Available: http://clusterware.cyi.ac.cy/data/paper.pdf

[5] top500.org. November 2015. [Online]. Available: http://top500.org/lists/2015/11/

[6] S. Chan, J. Cheung, E. Wu, H. Wang, C. Liu, X. Zhu, S. Peng, R. Luo, and T. W. Lam, “MICA: A fast short-read aligner that takes full advantage of Intel Many Integrated Core Architecture (MIC),” *CoRR*, vol. abs/1402.4876, 2014. [Online]. Available: http://arxiv.org/abs/1402.4876

[7] V. Noormofidi, S. R. Atlas, and H. Duan, “Performance Analysis of an Astrophysical Simulation Code on the Intel Xeon Phi Architecture,” *CoRR*, vol. abs/1510.02163, 2015. [Online]. Available: http://arxiv.org/abs/1510.02163

[8] 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*, Nov 2011, pp. 1–10.

[9] O. Kaczmarek, C. Schmidt, P. Steinbrecher, S. Mukherjee, and M. Wagner, “HISQ inverter on Intel Xeon Phi and NVIDIA GPUs,” *CoRR*, vol. abs/1409.1510, 2014. [Online]. Available: http://arxiv.org/abs/1409.1510

[10] 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 *Supercomputing*, J. M. Kunkel, T. Ludwig, and H. W. Meuer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 40–54.

[11] J. Fang, A. L. Varbanescu, H. J. Sips, L. Zhang, Y. Che, and C. Xu, “An Empirical Study of Intel Xeon Phi,” *CoRR*, vol. abs/1310.5842, 2013. [Online]. Available: http://arxiv.org/abs/1310.5842

[12] 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*, vol. 181, no. 1, pp. 113–117, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010465509002859

[13] B. Joó, Personal Communication.

[14] ——, Unpublished Notes.

[15] “Intel® Xeon Phi™ Coprocessor: Software Developers Guide,” Intel Corporation, Tech. Rep., March 2014.

[16] P. Vranas, G. Bhanot, M. Blumrich, D. Chen, A. Gara, P. Heidelberger, V. Salapura, and J. C. Sexton, “The BlueGene/L Supercomputer and Quantum ChromoDynamics,” in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC ’06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1188455.1188507

[17] P. A. Boyle, “The BAGEL Assembler Generation Library,” *Computer Physics Communications*, vol. 180, no. 12, pp. 2739–2748, 2009.

[18] B. Joó et. al, “Code Generator for the QPhiX library, Wilson Fermions.” [Online]. Available: https://github.com/JeffersonLab/qphix-codegen

[19] ——, “QPhiX: QCD for Intel Xeon Phi and Xeon Processors.” [Online]. Available: https://github.com/JeffersonLab/qphix

[20] M. Schröck, S. Simula, and A. Strelchenko, "Accelerating Twisted Mass LQCD with QPhiX," in *Proceedings of the 33rd International Symposium on Lattice Field Theory*, July 2015. [Online]. Available: http://arxiv.org/abs/1510.08879

[21] 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*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–13. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.2

[22] T. Sakurai, H. Tadano, and Y. Kuramashi, "Application of preconditioned block BiCGGR to the Wilson–Dirac equation with multiple right-hand sides in lattice QCD," arXiv:0907.3261, 2009. [Online]. Available: http://arxiv.org/pdf/0907.3261v1

[23] M. A. Clark, R. Babich, K. Barros, R. C. Brower, and C. Rebbi, "Solving Lattice QCD systems of equations using mixed precision solvers on GPUs," *Computer Physics Communications*, vol. 181, pp. 1517–1528, 2010.

[24] B. Joó et. al, "Barrier_mic.h." [Online]. Available: https://github.com/JeffersonLab/qphix/blob/master/include/qphix/Barrier_mic.h

[25] "How to Use Huge Pages to Improve Application Performance on Intel® Xeon Phi™ Coprocessor," Intel Corporation, Tech. Rep., 2012. [Online]. Available: https://software.intel.com/sites/default/files/Large_pages_mic_0.pdf

[26] usqcd software, "The QDP++ Framework for Lattice QCD." [Online]. Available: http://usqcd-software.github.io/qdpxx/

[27] S. Khan, Personal Communication.

[28] P. Mackenzie, "Lattice QCD and the Standard Model in the Exascale Era," in *SciDAC 2010*, July 2010, http://computing.ornl.gov/workshops/scidac2010/presentations/p_mackenzie.pdf.

APPENDIX A

INTERACTING COMPONENTS

μ			
1	2	3	4
00r 30i	00r 30r	00r 20i	00r 20r
00i 30r	00i 30i	00i 20r	00i 20i
10r 20i	10r 20r	10r 30i	10r 30r
10i 20r	10i 20i	10i 30r	10i 30i
01r 31i	01r 31r	01r 21i	01r 21r
01i 31r	01i 31i	01i 21r	01i 21i
11r 21i	11r 21r	11r 31i	11r 31r
11i 21r	11i 21i	11i 31r	11i 31i
02r 32i	02r 32r	02r 22i	02r 22r
02i 32r	02i 32i	02i 22r	02i 22i
12r 22i	12r 22r	12r 32i	12r 32r
12i 22r	12i 22i	12i 32r	12i 32i

TABLE 12: Components of ψ which interact during projection, which is relevant to 8 RHS pairing (see Subsection III.1.1)

APPENDIX B

COMPILATION VARIABLES

Variable	Range	Description (with implicit *if defined*)
BARR	–	Synchronize with `Barrier_mic.h`
CCT	–	Enables cache-controlling traversal
COMPRESS12	–	Enables 12-compression, mutually exclusive with COMPRESS16
COMPRESS16	–	Enables 16-compression
HUGEPAGES	–	Enables 2MB memory pages
ITER	INT+	Dslash will be applied to the entire lattice ITER number of times
ITT	–	Enables Vtune™ resume/pause around timing
L1PREFETCH	–	Disables L1 prefetch if set to 0
L2PREFETCH	0-255	Bitwise enable L2 prefetch for directions
LX, LY, LZ, LT	INT+	Set lattice dimensions
NEI0	–	Set neighbor indices to 0. Used for testing maximum performance
NOCHECKOUT	–	Do not display checksum after execution
NOFLIP	–	Disable permutes for testing (8 RHS)
QUICKSET	–	Disable initialization of ψ, χ, and U
SEED1	INT+	Set seed for ψ init
SEED2	INT+	Set seed for U init
SHIFTTABLE	–	Enables shift table, disabling index calc
SKIP	INT+	Determines how many iterations to perform without timing
THREADSITEREPORT	–	Output a list of sites processed by threads in numerical order separated by a newline
TIMEOUT	–	Output iteration time elapsed in seconds with `setprecision(12)`

TABLE 13: Compilation variables (C preprocessor). A range of – indicates a variable which can only be either defined or undefined. A range of INT+ indicates a variable may be set to any positive integer (without considering overflow).

APPENDIX C

8 RHS EXAMPLE CODE

```
  v1 = _mm512_load_ps( &((((float(*)[12][16])&psi[0])[fsite1])[6]) );
  v13 = _mm512_load_ps( &((((float(*)[12][16])&psi[0])[fsite1])[7]) );
#ifndef NOFLIP
  v1 = _mm512_add_ps(v1, pm(flip(v13)) );
#else
  v1 = _mm512_add_ps(v1,v13);
#endif
  v7 = _mm512_extload_ps(&(UCAST(uc[site][1]))[0][0][1], _MM_UPCONV_PS_NONE,_MM_BROADCAST_1X16,_MM_HINT_NONE);
  v14 = _mm512_mul_ps(v7, v1);

  v0 = _mm512_load_ps( &((((float(*)[12][16])&psi[0])[fsite1])[0]) );
  v13 = _mm512_load_ps( &((((float(*)[12][16])&psi[0])[fsite1])[1]) );
#ifndef NOFLIP
  v0 = _mm512_add_ps(v0, pm(flip(v13)) );
#else
  v0 = _mm512_add_ps(v0,v13);
#endif
  v15 = _mm512_mul_ps(v7, v0);

  v6 = _mm512_extload_ps(&(UCAST(uc[site][1]))[0][0][0], _MM_UPCONV_PS_NONE,_MM_BROADCAST_1X16,_MM_HINT_NONE);
  v14 = _mm512_fmsub_ps(v6, v0, v14);
  v15 = _mm512_fmadd_ps(v6, v1, v15);

  v2 = _mm512_load_ps( &((((float(*)[12][16])&psi[0])[fsite1])[2]) );
  v13 = _mm512_load_ps( &((((float(*)[12][16])&psi[0])[fsite1])[3]) );
#ifndef NOFLIP
  v2 = _mm512_add_ps(v2, pm(flip(v13)) );
#else
  v2 = _mm512_add_ps(v2,v13);
```

```
#endif

  v8 = _mm512_extload_ps(&(UCAST(uc[site][1]))[1][0][0], _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HINT_NONE);
  v14 = _mm512_fmadd_ps(v8, v2, v14);

  v3 = _mm512_load_ps( &((((float(*)[12][16])&psi[0])[fsite1])[8]) );
  v13 = _mm512_load_ps( &((((float(*)[12][16])&psi[0])[fsite1])[9]) );
#ifndef NOFLIP
  v3 = _mm512_add_ps(v3, pm(flip(v13)) );
#else
  v3 = _mm512_add_ps(v3,v13);
#endif
  v15 = _mm512_fmadd_ps(v8, v3, v15);

  v9 = _mm512_extload_ps(&(UCAST(uc[site][1]))[1][0][1], _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HINT_NONE);
  v14 = _mm512_fnmadd_ps(v9, v3, v14);
  v15 = _mm512_fmadd_ps(v9, v2, v15);

  v4 = _mm512_load_ps( &((((float(*)[12][16])&psi[0])[fsite1])[4]) );
  v13 = _mm512_load_ps( &((((float(*)[12][16])&psi[0])[fsite1])[5]) );
#ifndef NOFLIP
  v4 = _mm512_add_ps(v4, pm(flip(v13)) );
#else
  v4 = _mm512_add_ps(v4,v13);
#endif

#ifdef COMPRESS16
  v10 = _mm512_extload_ps(&(UCAST(uc[site][1]))[0][3][0], _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HINT_NONE);
#elif defined(COMPRESS12)
  v10 = _mm512_set1_ps( pgud1f[0][1][RE]*pgud1f[1][2][RE] - pgud1f[0][1][IM]*pgud1f[1][2][IM] - (pgud1f[0][2][RE]*pgud1f[1][1][RE] - pgud1f[0][2][IM]*pgud1f[1][1][IM]) );
#else
  v10 = _mm512_extload_ps(&(UCAST(uc[site][1]))[2][0][0], _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HINT_NONE);
#endif
```

```
  v14 = _mm512_fmadd_ps(v10, v4, v14);

  v5 = _mm512_load_ps( &((((float(*)[12][16])&psi[0])[fsite1])[10]) );
  v13 = _mm512_load_ps( &((((float(*)[12][16])&psi[0])[fsite1])[11]) );
#ifndef NOFLIP
  v5 = _mm512_add_ps(v5, pm(flip(v13)) );
#else
  v5 = _mm512_add_ps(v5,v13);
#endif
  v15 = _mm512_fmadd_ps(v10, v5, v15);

#ifdef COMPRESS16
  v11 = _mm512_extload_ps(&(UCAST(uc[site][1]))[0][3][1], _MM_UPCONV_PS_NONE,_MM_BROADCAST_1X16,_MM_HINT_NONE);
#elif defined(COMPRESS12)
  v11 = _mm512_set1_ps( -(pgud1f[0][1][RE]*pgud1f[1][2][IM] + pgud1f[0][1][IM]*pgud1f[1][2][RE]) + (pgud1f[0][2][RE]*pgud1f[1][1][IM] + pgud1f[0][2][IM]*pgud1f[1][1][RE]) );
#else
  v11 = _mm512_extload_ps(&(UCAST(uc[site][1]))[2][0][1], _MM_UPCONV_PS_NONE,_MM_BROADCAST_1X16,_MM_HINT_NONE);
#endif
  v14 = _mm512_fnmadd_ps(v11, v5, v14);
  v15 = _mm512_fmadd_ps(v11, v4, v15);

  v16 = _mm512_add_ps(v16, v14); //upper 00r | 01r
#ifndef NOFLIP
  v22 = _mm512_add_ps(v22, mp(flip(v14)) );
#else
  v22 = _mm512_add_ps(v22,v14);
#endif

  v17 = _mm512_add_ps(v17, v15);
#ifndef NOFLIP
  v23 = _mm512_add_ps(v23, mp(flip(v15)) );
#else
  v23 = _mm512_add_ps(v23,v15);
#endif

```

```
  v7 = _mm512_extload_ps(&(UCAST(uc[site][1]))[0][1][1],
     _MM_UPCONV_PS_NONE,_MM_BROADCAST_1X16,_MM_HINT_NONE);
  v14 = _mm512_mul_ps(v7, v1);
  v15 = _mm512_mul_ps(v7, v0);
  v6 = _mm512_extload_ps(&(UCAST(uc[site][1]))[0][1][0],
     _MM_UPCONV_PS_NONE,_MM_BROADCAST_1X16,_MM_HINT_NONE);
  v14 = _mm512_fmsub_ps(v6, v0, v14);
  v15 = _mm512_fmadd_ps(v6, v1, v15);
  v8 = _mm512_extload_ps(&(UCAST(uc[site][1]))[1][1][0],
     _MM_UPCONV_PS_NONE,_MM_BROADCAST_1X16,_MM_HINT_NONE);
  v14 = _mm512_fmadd_ps(v8, v2, v14);
  v15 = _mm512_fmadd_ps(v8, v3, v15);
  v9 = _mm512_extload_ps(&(UCAST(uc[site][1]))[1][1][1],
     _MM_UPCONV_PS_NONE,_MM_BROADCAST_1X16,_MM_HINT_NONE);
  v14 = _mm512_fnmadd_ps(v9, v3, v14);
  v15 = _mm512_fmadd_ps(v9, v2, v15);
#ifdef COMPRESS16
  v10 = _mm512_extload_ps(&(UCAST(uc[site][1]))[1][3][0],
     _MM_UPCONV_PS_NONE,_MM_BROADCAST_1X16,_MM_HINT_NONE);
#elif defined(COMPRESS12)
  v10 = _mm512_set1_ps( pgud1f[1][0][RE]*pgud1f[0][2][RE] - pgud1f
     [1][0][IM]*pgud1f[0][2][IM] - (pgud1f[0][0][RE]*pgud1f[1][2][RE]
      - pgud1f[0][0][IM]*pgud1f[1][2][IM]) );
#else
  v10 = _mm512_extload_ps(&(UCAST(uc[site][1]))[2][1][0],
     _MM_UPCONV_PS_NONE,_MM_BROADCAST_1X16,_MM_HINT_NONE);
#endif
  v14 = _mm512_fmadd_ps(v10, v4, v14);
  v15 = _mm512_fmadd_ps(v10, v5, v15);
#ifdef COMPRESS16
  v11 = _mm512_extload_ps(&(UCAST(uc[site][1]))[1][3][1],
     _MM_UPCONV_PS_NONE,_MM_BROADCAST_1X16,_MM_HINT_NONE);
#elif defined(COMPRESS12)
  v11 = _mm512_set1_ps( -(pgud1f[1][0][RE]*pgud1f[0][2][IM] + pgud1f
     [1][0][IM]*pgud1f[0][2][RE]) + (pgud1f[0][0][RE]*pgud1f[1][2][IM
     ] + pgud1f[0][0][IM]*pgud1f[1][2][RE]) );
#else
  v11 = _mm512_extload_ps(&(UCAST(uc[site][1]))[2][1][1],
     _MM_UPCONV_PS_NONE,_MM_BROADCAST_1X16,_MM_HINT_NONE);
#endif
  v14 = _mm512_fnmadd_ps(v11, v5, v14);
  v15 = _mm512_fmadd_ps(v11, v4, v15);

  v18 = _mm512_add_ps(v18, v14);
#ifndef NOFLIP
  v24 = _mm512_add_ps(v24, mp(flip(v14)) );
#else
  v24 = _mm512_add_ps(v24,v14);
#endif
  v19 = _mm512_add_ps(v19, v15);
#ifndef NOFLIP
  v25 = _mm512_add_ps(v25, mp(flip(v15)) );
#else
  v25 = _mm512_add_ps(v25,v15);
#endif

  v7 = _mm512_extload_ps(&(UCAST(uc[site][1]))[0][2][1], _MM_UPCONV_PS_NONE,_MM_BROADCAST_1X16,_MM_HINT_NONE);
  v14 = _mm512_mul_ps(v7, v1);
  v15 = _mm512_mul_ps(v7, v0);
  v6 = _mm512_extload_ps(&(UCAST(uc[site][1]))[0][2][0], _MM_UPCONV_PS_NONE,_MM_BROADCAST_1X16,_MM_HINT_NONE);
  v14 = _mm512_fmsub_ps(v6, v0, v14);
  v15 = _mm512_fmadd_ps(v6, v1, v15);
  v8 = _mm512_extload_ps(&(UCAST(uc[site][1]))[1][2][0], _MM_UPCONV_PS_NONE,_MM_BROADCAST_1X16,_MM_HINT_NONE);
  v14 = _mm512_fmadd_ps(v8, v2, v14);
  v15 = _mm512_fmadd_ps(v8, v3, v15);
  v9 = _mm512_extload_ps(&(UCAST(uc[site][1]))[1][2][1], _MM_UPCONV_PS_NONE,_MM_BROADCAST_1X16,_MM_HINT_NONE);
  v14 = _mm512_fnmadd_ps(v9, v3, v14);
  v15 = _mm512_fmadd_ps(v9, v2, v15);
#if defined(COMPRESS16) || defined(COMPRESS12)
  v10 =_mm512_set1_ps( pgud1f[0][0][RE]*pgud1f[1][1][RE] - pgud1f[0][0][IM]*pgud1f[1][1][IM] - (pgud1f[0][1][RE]*pgud1f[1][0][RE] - pgud1f[0][1][IM]*pgud1f[1][0][IM]) );
#else
  v10 = _mm512_extload_ps(&(UCAST(uc[site][1]))[2][2][0], _MM_UPCONV_PS_NONE,_MM_BROADCAST_1X16,_MM_HINT_NONE);
#endif
  v14 = _mm512_fmadd_ps(v10, v4, v14);
  v15 = _mm512_fmadd_ps(v10, v5, v15);
```

```
#if defined(COMPRESS16) || defined(COMPRESS12)
  v11 = _mm512_set1_ps( -(pgud1f[0][0][RE]*pgud1f[1][1][IM] + pgud1f
      [0][0][IM]*pgud1f[1][1][RE]) + pgud1f[0][1][RE]*pgud1f[1][0][IM]
       + pgud1f[0][1][IM]*pgud1f[1][0][RE] );
#else
  v11 = _mm512_extload_ps(&(UCAST(uc[site][1]))[2][2][1],
      _MM_UPCONV_PS_NONE,_MM_BROADCAST_1X16,_MM_HINT_NONE);
#endif
  v14 = _mm512_fnmadd_ps(v11, v5, v14);
  v15 = _mm512_fmadd_ps(v11, v4, v15);
  v20 =  _mm512_add_ps(v20, v14);
#ifndef NOFLIP
  v26 = _mm512_add_ps(v26, mp(flip(v14)) );
#else
  v26 = _mm512_add_ps(v26,v14);
#endif

  v21 = _mm512_add_ps(v21, v15);
#ifndef NOFLIP
  v27 = _mm512_add_ps(v27, mp(flip(v15)) );
#else
  v27 = _mm512_add_ps(v27,v15);
#endif
```

Listing C.1: 8 RHS code for direction 1 forward. The variables are unfortunately named, but think of v0–v5 as holding projections, v6–v11 U, v12–v15 temporary, v16–v21 upper sum, v22–27 lower sum.

APPENDIX D

16 RHS EXAMPLE CODE

```
#undef thesite
#define thesite fsite1
#define r00 _mm512_load_ps( &((((float(*)[24][16])&psi[0])[thesite])[0]) )
#define r01 _mm512_load_ps( &((((float(*)[24][16])&psi[0])[thesite])[1]) )
#define r02 _mm512_load_ps( &((((float(*)[24][16])&psi[0])[thesite])[2]) )
#define r10 _mm512_load_ps( &((((float(*)[24][16])&psi[0])[thesite])[3]) )
#define r11 _mm512_load_ps( &((((float(*)[24][16])&psi[0])[thesite])[4]) )
#define r12 _mm512_load_ps( &((((float(*)[24][16])&psi[0])[thesite])[5]) )
#define r20 _mm512_load_ps( &((((float(*)[24][16])&psi[0])[thesite])[6]) )
#define r21 _mm512_load_ps( &((((float(*)[24][16])&psi[0])[thesite])[7]) )
#define r22 _mm512_load_ps( &((((float(*)[24][16])&psi[0])[thesite])[8]) )
#define r30 _mm512_load_ps( &((((float(*)[24][16])&psi[0])[thesite])[9]) )
#define r31 _mm512_load_ps( &((((float(*)[24][16])&psi[0])[thesite])[10]) )
#define r32 _mm512_load_ps( &((((float(*)[24][16])&psi[0])[thesite])[11]) )
#define i00 _mm512_load_ps( &((((float(*)[24][16])&psi[0])[thesite])[12]) )
#define i01 _mm512_load_ps( &((((float(*)[24][16])&psi[0])[thesite])[13]) )
#define i02 _mm512_load_ps( &((((float(*)[24][16])&psi[0])[thesite])[14]) )
#define i10 _mm512_load_ps( &((((float(*)[24][16])&psi[0])[thesite])[15]) )
```

```
  #define i11 _mm512_load_ps( &((((float(*)[24][16])&psi[0])[thesite])[16]) )
  #define i12 _mm512_load_ps( &((((float(*)[24][16])&psi[0])[thesite])[17]) )
  #define i20 _mm512_load_ps( &((((float(*)[24][16])&psi[0])[thesite])[18]) )
  #define i21 _mm512_load_ps( &((((float(*)[24][16])&psi[0])[thesite])[19]) )
  #define i22 _mm512_load_ps( &((((float(*)[24][16])&psi[0])[thesite])[20]) )
  #define i30 _mm512_load_ps( &((((float(*)[24][16])&psi[0])[thesite])[21]) )
  #define i31 _mm512_load_ps( &((((float(*)[24][16])&psi[0])[thesite])[22]) )
  #define i32 _mm512_load_ps( &((((float(*)[24][16])&psi[0])[thesite])[23]) )
  //proj_r00 first
  proj = _mm512_add_ps(r00, r30);
  u_r0 = _mm512_extload_ps(&(UCAST(uc[site][1]))[0][0][0], _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HILT_NONE);
  upper_r00 = _mm512_fmadd_ps(proj,u_r0,upper_r00);
  lower_r01 = _mm512_fmadd_ps(proj,u_r0,lower_r01);
  u_i0 = _mm512_extload_ps(&(UCAST(uc[site][1]))[0][0][1], _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HILT_NONE);
  upper_i00 = _mm512_fmadd_ps(proj,u_i0,upper_i00);
  lower_i01 = _mm512_fmadd_ps(proj,u_i0,lower_i01);
  u_r1 = _mm512_extload_ps(&(UCAST(uc[site][1]))[0][1][0], _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HILT_NONE);
  upper_r10 = _mm512_fmadd_ps(proj,u_r1,upper_r10);
  lower_r11 = _mm512_fmadd_ps(proj,u_r1,lower_r11);
  u_i1 = _mm512_extload_ps(&(UCAST(uc[site][1]))[0][1][1], _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HILT_NONE);
  upper_i10 = _mm512_fmadd_ps(proj,u_i1,upper_i10);
  lower_i11 = _mm512_fmadd_ps(proj,u_i1,lower_i11);
  u_r2 = _mm512_extload_ps(&(UCAST(uc[site][1]))[0][2][0], _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HILT_NONE);
  upper_r20 = _mm512_fmadd_ps(proj,u_r2,upper_r20);
  lower_r21 = _mm512_fmadd_ps(proj,u_r2,lower_r21);
  u_i2 = _mm512_extload_ps(&(UCAST(uc[site][1]))[0][2][1], _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HILT_NONE);
  upper_i20 = _mm512_fmadd_ps(proj,u_i2,upper_i20);
```

```
    lower_i21 = _mm512_fmadd_ps(proj, u_i2, lower_i21);

    //proj_i00
    proj = _mm512_add_ps( i00, i30);
    upper_r00 = _mm512_fnmadd_ps(proj, u_i0, upper_r00);
    lower_r01 = _mm512_fnmadd_ps(proj, u_i0, lower_r01);

    upper_i00 = _mm512_fmadd_ps(proj, u_r0, upper_i00);
    lower_i01 = _mm512_fmadd_ps(proj, u_r0, lower_i01);

    upper_r10 = _mm512_fnmadd_ps(proj, u_i1, upper_r10);
    lower_r11 = _mm512_fnmadd_ps(proj, u_i1, lower_r11);

    upper_i10 = _mm512_fmadd_ps(proj, u_r1, upper_i10);
    lower_i11 = _mm512_fmadd_ps(proj, u_r1, lower_i11);

    upper_r20 = _mm512_fnmadd_ps(proj, u_i2, upper_r20);
    lower_r21 = _mm512_fnmadd_ps(proj, u_i2, lower_r21);

    upper_i20 = _mm512_fmadd_ps(proj, u_r2, upper_i20);
    lower_i21 = _mm512_fmadd_ps(proj, u_r2, lower_i21);

    //proj_r01
    proj = _mm512_sub_ps( r10, r20   );
    upper_r01 = _mm512_fmadd_ps(proj, u_r0, upper_r01);
    lower_r00 = _mm512_fnmadd_ps(proj, u_r0, lower_r00);

    upper_i01 = _mm512_fmadd_ps(proj, u_i0, upper_i01);
    lower_i00 = _mm512_fnmadd_ps(proj, u_i0, lower_i00);

    upper_r11 = _mm512_fmadd_ps(proj, u_r1, upper_r11);
    lower_r10 = _mm512_fnmadd_ps(proj, u_r1, lower_r10);

    upper_i11 = _mm512_fmadd_ps(proj, u_i1, upper_i11);
    lower_i10 = _mm512_fnmadd_ps(proj, u_i1, lower_i10);

    upper_r21 = _mm512_fmadd_ps(proj, u_r2, upper_r21);
    lower_r20 = _mm512_fnmadd_ps(proj, u_r2, lower_r20);

    upper_i21 = _mm512_fmadd_ps(proj, u_i2, upper_i21);
    lower_i20 = _mm512_fnmadd_ps(proj, u_i2, lower_i20);

    //proj_i01
    proj = _mm512_sub_ps(i10, i20);
    upper_r01 = _mm512_fnmadd_ps(proj, u_i0, upper_r01);
    lower_r00 = _mm512_fmadd_ps(proj, u_i0, lower_r00);

    upper_i01 = _mm512_fmadd_ps(proj, u_r0, upper_i01);
    lower_i00 = _mm512_fnmadd_ps(proj, u_r0, lower_i00);

    upper_r11 = _mm512_fnmadd_ps(proj, u_i1, upper_r11);
    lower_r10 = _mm512_fmadd_ps(proj, u_i1, lower_r10);

    upper_i11 = _mm512_fmadd_ps(proj, u_r1, upper_i11);
    lower_i10 = _mm512_fnmadd_ps(proj, u_r1, lower_i10);

    upper_r21 = _mm512_fnmadd_ps(proj, u_i2, upper_r21);
    lower_r20 = _mm512_fmadd_ps(proj, u_i2, lower_r20);

    upper_i21 = _mm512_fmadd_ps(proj, u_r2, upper_i21);
    lower_i20 = _mm512_fnmadd_ps(proj, u_r2, lower_i20);

    //proj_r10
    proj = _mm512_add_ps(r01, r31);
    u_r0 = _mm512_extload_ps(&(UCAST(uc[site][1]))[1][0][0],
        _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HILT_NONE);
    upper_r00 = _mm512_fmadd_ps(proj, u_r0, upper_r00);
    lower_r01 = _mm512_fmadd_ps(proj, u_r0, lower_r01);
    u_i0 = _mm512_extload_ps(&(UCAST(uc[site][1]))[1][0][1],
        _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HILT_NONE);
    upper_i00 = _mm512_fmadd_ps(proj, u_i0, upper_i00);
    lower_i01 = _mm512_fmadd_ps(proj, u_i0, lower_i01);
    u_r1 = _mm512_extload_ps(&(UCAST(uc[site][1]))[1][1][0],
        _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HILT_NONE);
    upper_r10 = _mm512_fmadd_ps(proj, u_r1, upper_r10);
    lower_r11 = _mm512_fmadd_ps(proj, u_r1, lower_r11);
    u_i1 = _mm512_extload_ps(&(UCAST(uc[site][1]))[1][1][1],
        _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HILT_NONE);
    upper_i10 = _mm512_fmadd_ps(proj, u_i1, upper_i10);
    lower_i11 = _mm512_fmadd_ps(proj, u_i1, lower_i11);
    u_r2 = _mm512_extload_ps(&(UCAST(uc[site][1]))[1][2][0],
        _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HILT_NONE);
    upper_r20 = _mm512_fmadd_ps(proj, u_r2, upper_r20);
    lower_r21 = _mm512_fmadd_ps(proj, u_r2, lower_r21);
    u_i2 = _mm512_extload_ps(&(UCAST(uc[site][1]))[1][2][1],
        _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HILT_NONE);
    upper_i20 = _mm512_fmadd_ps(proj, u_i2, upper_i20);
    lower_i21 = _mm512_fmadd_ps(proj, u_i2, lower_i21);

    //proj_i10
    proj = _mm512_add_ps(i01,  i31);
    upper_r00 = _mm512_fnmadd_ps(proj, u_i0, upper_r00);
    lower_r01 = _mm512_fnmadd_ps(proj, u_i0, lower_r01);

    upper_i00 = _mm512_fmadd_ps(proj, u_r0, upper_i00);
    lower_i01 = _mm512_fmadd_ps(proj, u_r0, lower_i01);

    upper_r10 = _mm512_fnmadd_ps(proj, u_i1, upper_r10);
    lower_r11 = _mm512_fnmadd_ps(proj, u_i1, lower_r11);

    upper_i10 = _mm512_fmadd_ps(proj, u_r1, upper_i10);
    lower_i11 = _mm512_fmadd_ps(proj, u_r1, lower_i11);

    upper_r20 = _mm512_fnmadd_ps(proj, u_i2, upper_r20);
    lower_r21 = _mm512_fnmadd_ps(proj, u_i2, lower_r21);

    upper_i20 = _mm512_fmadd_ps(proj, u_r2, upper_i20);
    lower_i21 = _mm512_fmadd_ps(proj, u_r2, lower_i21);

    //proj_r11
    proj = _mm512_sub_ps(r11,  r21);
    upper_r01 = _mm512_fmadd_ps(proj, u_r0, upper_r01);
    lower_r00 = _mm512_fnmadd_ps(proj, u_r0, lower_r00);

    upper_i01 = _mm512_fmadd_ps(proj, u_i0, upper_i01);
    lower_i00 = _mm512_fnmadd_ps(proj, u_i0, lower_i00);

    upper_r11 = _mm512_fmadd_ps(proj, u_r1, upper_r11);
    lower_r10 = _mm512_fnmadd_ps(proj, u_r1, lower_r10);

    upper_i11 = _mm512_fmadd_ps(proj, u_i1, upper_i11);
    lower_i10 = _mm512_fnmadd_ps(proj, u_i1, lower_i10);

    upper_r21 = _mm512_fmadd_ps(proj, u_r2, upper_r21);
    lower_r20 = _mm512_fnmadd_ps(proj, u_r2, lower_r20);

    upper_i21 = _mm512_fmadd_ps(proj, u_i2, upper_i21);
    lower_i20 = _mm512_fnmadd_ps(proj, u_i2, lower_i20);

    //proj_i11
    proj = _mm512_sub_ps(i11, i21);
    upper_r01 = _mm512_fnmadd_ps(proj, u_i0, upper_r01);
    lower_r00 = _mm512_fmadd_ps(proj, u_i0, lower_r00);

    upper_i01 = _mm512_fmadd_ps(proj, u_r0, upper_i01);
    lower_i00 = _mm512_fnmadd_ps(proj, u_r0, lower_i00);

    upper_r11 = _mm512_fnmadd_ps(proj, u_i1, upper_r11);
    lower_r10 = _mm512_fmadd_ps(proj, u_i1, lower_r10);

    upper_i11 = _mm512_fmadd_ps(proj, u_r1, upper_i11);
    lower_i10 = _mm512_fnmadd_ps(proj, u_r1, lower_i10);

    upper_r21 = _mm512_fnmadd_ps(proj, u_i2, upper_r21);
    lower_r20 = _mm512_fmadd_ps(proj, u_i2, lower_r20);

    upper_i21 = _mm512_fmadd_ps(proj, u_r2, upper_i21);
    lower_i20 = _mm512_fnmadd_ps(proj, u_r2, lower_i20);

    //proj_r20
    proj = _mm512_add_ps(r02, r32);
#ifdef COMPRESS16
    u_r0 = _mm512_extload_ps(&(UCAST(uc[site][1]))[0][3][0],
        _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HILT_NONE);
#elif defined(COMPRESS12)
    u_r0 = _mm512_set1_ps( pgud1f[0][1][RE]*pgud1f[1][2][RE] - pgud1f
        [0][1][IM]*pgud1f[1][2][IM] - (pgud1f[0][2][RE]*pgud1f[1][1][RE]
         - pgud1f[0][2][IM]*pgud1f[1][1][IM]) );
#else
    u_r0 = _mm512_extload_ps(&(UCAST(uc[site][1]))[2][0][0],
        _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HILT_NONE);
#endif
    upper_r00 = _mm512_fmadd_ps(proj, u_r0, upper_r00);
    lower_r01 = _mm512_fmadd_ps(proj, u_r0, lower_r01);
```

```
#ifdef COMPRESS16
  u_i0 = _mm512_extload_ps(&(UCAST(uc[site][1]))[0][3][1], _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HILT_NONE);
#elif defined(COMPRESS12)
  u_i0 = _mm512_set1_ps( -(pgud1f[0][1][RE]*pgud1f[1][2][IM] + pgud1f[0][1][IM]*pgud1f[1][2][RE]) + (pgud1f[0][2][RE]*pgud1f[1][1][IM] + pgud1f[0][2][IM]*pgud1f[1][1][RE]) );
#else
  u_i0 = _mm512_extload_ps(&(UCAST(uc[site][1]))[2][0][1], _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HILT_NONE);
#endif
  upper_i00 = _mm512_fmadd_ps(proj, u_i0, upper_i00);
  lower_i01 = _mm512_fmadd_ps(proj, u_i0, lower_i01);
#ifdef COMPRESS16
  u_r1 = _mm512_extload_ps(&(UCAST(uc[site][1]))[1][3][0], _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HILT_NONE);
#elif defined(COMPRESS12)
  u_r1 = _mm512_set1_ps( pgud1f[1][0][RE]*pgud1f[0][2][RE] - pgud1f[1][0][IM]*pgud1f[0][2][IM] - (pgud1f[0][0][RE]*pgud1f[1][2][RE] - pgud1f[0][0][IM]*pgud1f[1][2][IM]) );
#else
  u_r1 = _mm512_extload_ps(&(UCAST(uc[site][1]))[2][1][0], _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HILT_NONE);
#endif
  upper_r10 = _mm512_fmadd_ps(proj, u_r1, upper_r10);
  lower_r11 = _mm512_fmadd_ps(proj, u_r1, lower_r11);
#ifdef COMPRESS16
  u_i1 = _mm512_extload_ps(&(UCAST(uc[site][1]))[1][3][1], _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HILT_NONE);
#elif defined(COMPRESS12)
  u_i1 = _mm512_set1_ps( -(pgud1f[1][0][RE]*pgud1f[0][2][IM] + pgud1f[1][0][IM]*pgud1f[0][2][RE]) + (pgud1f[0][0][RE]*pgud1f[1][2][IM] + pgud1f[0][0][IM]*pgud1f[1][2][RE]) );
#else
  u_i1 = _mm512_extload_ps(&(UCAST(uc[site][1]))[2][1][1], _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HILT_NONE);
#endif
  upper_i10 = _mm512_fmadd_ps(proj, u_i1, upper_i10);
  lower_i11 = _mm512_fmadd_ps(proj, u_i1, lower_i11);
#if defined(COMPRESS16) || defined(COMPRESS12)
```

```
  u_r2 =_mm512_set1_ps( pgud1f[0][0][RE]*pgud1f[1][1][RE] - pgud1f[0][0][IM]*pgud1f[1][1][IM] - (pgud1f[0][1][RE]*pgud1f[1][0][RE] - pgud1f[0][1][IM]*pgud1f[1][0][IM]) );
#else
  u_r2 = _mm512_extload_ps(&(UCAST(uc[site][1]))[2][2][0], _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HILT_NONE);
#endif
  upper_r20 = _mm512_fmadd_ps(proj, u_r2, upper_r20);
  lower_r21 = _mm512_fmadd_ps(proj, u_r2, lower_r21);
#if defined(COMPRESS16) || defined(COMPRESS12)
  u_i2 = _mm512_set1_ps( -(pgud1f[0][0][RE]*pgud1f[1][1][IM] + pgud1f[0][0][IM]*pgud1f[1][1][RE]) + pgud1f[0][1][RE]*pgud1f[1][0][IM] + pgud1f[0][1][IM]*pgud1f[1][0][RE] );
#else
  u_i2 = _mm512_extload_ps(&(UCAST(uc[site][1]))[2][2][1], _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HILT_NONE);
#endif
  upper_i20 = _mm512_fmadd_ps(proj, u_i2, upper_i20);
  lower_i21 = _mm512_fmadd_ps(proj, u_i2, lower_i21);

  //proj_i20
  proj = _mm512_add_ps(i02, i32);
  upper_r00 = _mm512_fnmadd_ps(proj, u_i0, upper_r00);
  lower_r01 = _mm512_fnmadd_ps(proj, u_i0, lower_r01);

  upper_i00 = _mm512_fmadd_ps(proj, u_r0, upper_i00);
  lower_i01 = _mm512_fmadd_ps(proj, u_r0, lower_i01);

  upper_r10 = _mm512_fnmadd_ps(proj, u_i1, upper_r10);
  lower_r11 = _mm512_fnmadd_ps(proj, u_i1, lower_r11);

  upper_i10 = _mm512_fmadd_ps(proj, u_r1, upper_i10);
  lower_i11 = _mm512_fmadd_ps(proj, u_r1, lower_i11);

  upper_r20 = _mm512_fnmadd_ps(proj, u_i2, upper_r20);
  lower_r21 = _mm512_fnmadd_ps(proj, u_i2, lower_r21);

  upper_i20 = _mm512_fmadd_ps(proj, u_r2, upper_i20);
  lower_i21 = _mm512_fmadd_ps(proj, u_r2, lower_i21);

  //proj_r21
```

```
    proj = _mm512_sub_ps(r12, r22);
    upper_r01 = _mm512_fmadd_ps(proj,u_r0,upper_r01);
    lower_r00 = _mm512_fnmadd_ps(proj,u_r0,lower_r00);

    upper_i01 = _mm512_fmadd_ps(proj,u_i0,upper_i01);
    lower_i00 = _mm512_fnmadd_ps(proj,u_i0,lower_i00);

    upper_r11 = _mm512_fmadd_ps(proj,u_r1,upper_r11);
    lower_r10 = _mm512_fnmadd_ps(proj,u_r1,lower_r10);

    upper_i11 = _mm512_fmadd_ps(proj,u_i1,upper_i11);
    lower_i10 = _mm512_fnmadd_ps(proj,u_i1,lower_i10);

    upper_r21 = _mm512_fmadd_ps(proj,u_r2,upper_r21);
    lower_r20 = _mm512_fnmadd_ps(proj,u_r2,lower_r20);

    upper_i21 = _mm512_fmadd_ps(proj,u_i2,upper_i21);
    lower_i20 = _mm512_fnmadd_ps(proj,u_i2,lower_i20);

    //proj_i21
    proj = _mm512_sub_ps(i12, i22);
    upper_r01 = _mm512_fnmadd_ps(proj,u_i0,upper_r01);
    lower_r00 = _mm512_fmadd_ps(proj,u_i0,lower_r00);

    upper_i01 = _mm512_fmadd_ps(proj,u_r0,upper_i01);
    lower_i00 = _mm512_fnmadd_ps(proj,u_r0,lower_i00);

    upper_r11 = _mm512_fnmadd_ps(proj,u_i1,upper_r11);
    lower_r10 = _mm512_fmadd_ps(proj,u_i1,lower_r10);

    upper_i11 = _mm512_fmadd_ps(proj,u_r1,upper_i11);
    lower_i10 = _mm512_fnmadd_ps(proj,u_r1,lower_i10);

    upper_r21 = _mm512_fnmadd_ps(proj,u_i2,upper_r21);
    lower_r20 = _mm512_fmadd_ps(proj,u_i2,lower_r20);

    upper_i21 = _mm512_fmadd_ps(proj,u_r2,upper_i21);
    lower_i20 = _mm512_fnmadd_ps(proj,u_r2,lower_i20);
```

Listing D.1: 16 RHS code for direction 1 forward. As the second code, it's better organized.

VITA

Aaron Walden
Department of Computer Science
Old Dominion University
Norfolk, VA 23529

EDUCATION

M.S. in Computer Science, Old Dominion University, 2016
B.S. in Computer Science, Old Dominion University, 2014
A.S. in Computer Science, Tidewater Community College, 2012
A.A.S. in Electronics Engineering Technology, Tidewater Community College, 2012

EMPLOYMENT

2014–present	**Graduate Research Assistant**
	Old Dominion University, Norfolk, VA.
	Topic: Multiple right-hand side Dslash for Xeon Phi™
2013–2014	**NSF-funded Research Experience for Undergraduates**
	Old Dominion University, Norfolk, VA.
	Topic: ALERT, Wireless Sensor Re-tasking
	http://www.cs.odu.edu/~inets/Public/ALERT

PUBLICATIONS

A complete list is available at http://www.cs.odu.edu/~awalden/pubs

Permalinks

Email: waldenac@gmail.com

Typeset using LaTeX.