CHAPTER

WILSON DSLASH KERNEL FROM LATTICE QCD OPTIMIZATION

9

Bálint Joó*, Mikhail Smelyanskiy†, Dhiraj D. Kalamkar‡, Karthikeyan Vaidyanathan‡

*U.S. Department of Energy, Thomas Jefferson National Accelerator Facility (Jefferson Lab), USA** *Parallel Computing Lab, Intel Corporation, California, USA† Parallel Computing Lab, Intel Corporation, India‡*

Quantum chromodynamics (QCD) is the theory of the strong nuclear force, one of the fundamental forces of nature making up the Standard Model of particle interactions. QCD is responsible for binding quarks together into protons and neutrons, which in turn make up the nuclei of atoms. Lattice quantum chromodynamics, (lattice QCD or LQCD) is a version of QCD suitable for use on computers. It is the only model independent approach for carrying out nonperturbative calculations in QCD, and it is used primarily in calculations for theoretical nuclear and high energy physics. Lattice QCD simulations of quarks and gluons were one of the original Grand Challenge problems in High Performance Computing, and lattice QCD codes are typically implemented early on new high performance computing systems. Lattice QCD calculations use significant proportions of supercomputer time in the United States and worldwide. As an example, in 2014 lattice QCD calculations were responsible for consuming 13% of the computer resources at the U.S. Department of Energy, National Energy Research Scientific Computing Center (NERSC).

In this chapter, we will explore a key kernel of LQCD calculations, known as the Wilson-Dslash kernel. This kernel is essentially a finite difference operator, similar in spirit to stencils. We will outline our optimizations of this operator for the Intel Xeon Phi coprocessor following the methods described first in our paper "Lattice QCD on Intel Xeon Phi coprocessors" (see the "For more information" section at the end of this chapter for the specific information). Our approach is to take the reader on a journey, starting from an existing production implementation of this operator which already features thread-level parallelism, and yet still does not achieve very high performance when deployed on the Intel Xeon Phi coprocessor. By comparing performances with a simple performance model, we will proceed to show how improving cache reuse and memory bandwidth utilization along with vectorization can improve the performance of this code by more than a factor of 8× over the original implementation running on Intel Xeon Phi coprocessor. What is more, when we feed the optimizations back to regular Intel Xeon processors, we observe a 2.6× performance improvement over the best configuration of the original code on that platform too.

Throughout the chapter we will make reference to the supplied Code package, which contains both our initial implementation of Wilson-Dslash as well as the optimized one, along with some dependency libraries. While we concentrate primarily on these two Dslash implementations and a code generator, the Code package features many directories which we set up to allow the code to be built in different configurations, utilizing different optimizations. Throughout the chapter we will always highlight the

directory in the Code package where one should work in order to follow our discussion. We note upfront, that our work on the Intel Xeon Phi coprocessor architecture was done exclusively in *native-mode* rather than using heterogeneous, offload techniques. For build instructions in the Code package, we urge the reader to follow the supplied README file.

THE WILSON-DSLASH KERNEL

In this section, we will describe the Wilson-Dslash kernel and consider some of its computational properties. Before going into the nitty gritty let us say a few words about the LQCD setup. The idea is that instead of the usual continuum we represent four-dimensional space-time as a four-dimensional (hyper)cubic lattice. Each lattice point has coordinates $x = (n_x,n_y,n_z,n_t)$ where the components are just the coordinates in the usual X, Y, Z, and T directions. The quark fields are defined on the lattice sites and are known as *spinors*, denoted as $\psi(x)$. For current purposes $\psi(x)$ can be thought of as a set of complex numbers with two additional indices which are called "color" and "spin." So a quark field at site x is denoted $\psi^a_\alpha(x)$ with $a = \{0,1,2\}$ being the color indices, and $\alpha = \{0,1,2,3\}$ being the spin indices. In general, Latin indices will indicate color, and Greek-letter indices will indicate spin. In addition to the quark fields, one also has fields for the gluons known as *gauge fields*. These are ascribed to directed links between lattice points and for this reason they are often referred to as *gluon links* or *link matrices*. Typically they are denoted as $U^{ab}_\mu(x)$ where a and b are color indices again, but μ is a *direction* index. For $\mu = \{0,1,2,3\}$ we understand that $U^{ab}_\mu(x)$ is the gluon field on the link *emanating* from site x, in the forward direction in X, Y, Z, and T directions, respectively. As one can guess from the Latin indices, on each link U is a 3×3 complex matrix in *color*, with no spin index. Further, U is more than just a 3×3 matrix, it is also a member of the group $SU(3)$, which means that it is *unitary*, so that the Hermitian Conjugate of any given link matrix (the transpose of the matrix with each element complex conjugated) is its inverse: $(U^{a,b})^\dagger = (U^{b,a})^* = U^{-1}$ and each matrix has determinant $\det(U^{a,b}_\mu(x)) = 1$.

With these definitions in mind, we can write down the Wilson-Dslash operator D, acting on a quark spinor ψ as:

$$D^{a,b}_{\alpha,\beta}(x,y)\psi^b_\beta(y) = \sum_{\mu=0}^{3} U^{a,b}_\mu(x) P^{-\mu}_{\alpha,\beta}\psi^b_\beta(x+\hat{\mu}) + U^{\dagger\ \ a,b}(x-\hat{\mu}) P^{+\mu}_{\alpha,\beta}\psi^b_\beta(x-\hat{\mu}), \tag{9.1}$$

where in the equation above, *repeated indices are summed over*, and the $P^{\pm\mu}$ are *projection matrices* acting on *spin indices*. We can also see that the operation is like a stencil: for each output point x, we use the spinors from the neighboring points in the forward and backward μ directions: $\psi(x+\hat{\mu})$ and $\psi(x-\hat{\mu})$; and the sum involves all four directions. Each spinor from a neighboring site is multiplied by the link connecting the central site to it. So $\psi(x+\hat{\mu})$ is multiplied by $U_\mu(x)$ and $\psi(x-\hat{\mu})$ is multiplied by $U^\dagger(x-\hat{\mu})$, which—because of the Hermitian conjugation—should be interpreted as the "inverse" of the link pointing from $x-\hat{\mu}$ to x, that is, the link pointing from x to $x-\hat{\mu}$.

The multiplication by the U links is done only for the color indices, and must be repeated for each of the four spin indices. It may help to think of the spinor as a four-component vector, where each component is itself a three-component complex vector (also known as a color-vector, or color 3-vector). Likewise, the multiplication by $P^{\pm\mu}$ is done only for spin-indices, and must be repeated for each of the three color indices of ψ. In this case, and when working with spin-indices, it may be helpful to consider the spinor as an object with three elements, each of which is a four-component vector (spin-vector).

One final trick to consider is due to the nature of the projectors: $P^{\pm\mu} = (1 \pm \gamma_\mu)$. The γ_μ are 4×4 complex matrices acting only on spins, and they are sparse. We choose them to be in a particular representation which we show below:

$$\gamma_0 = \begin{pmatrix} 0 & 0 & 0 & i \\ 0 & 0 & i & 0 \\ 0 & -i & 0 & 0 \\ -i & 0 & 0 & 0 \end{pmatrix} \quad \gamma_1 = \begin{pmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{pmatrix}$$

$$\gamma_2 = \begin{pmatrix} 0 & 0 & i & 0 \\ 0 & 0 & 0 & -i \\ -i & 0 & 0 & 0 \\ 0 & i & 0 & 0 \end{pmatrix} \quad \gamma_3 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

The projectors possess a particular property which we illustrate below for the case for P^{-0}. Let us consider the spinor ψ^a_α on a single site, for just one color component, say $a = 0$ for simplicity (and of course this is repeated for all color components). We can write $\psi_\alpha = (\psi_0,\psi_1,\psi_2,\psi_3)^T$. Then

$$P^{-0}\psi = \begin{pmatrix} 1 & 0 & 0 & -i \\ 0 & 1 & -i & 0 \\ 0 & i & 1 & 0 \\ i & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \psi_0 \\ \psi_1 \\ \psi_2 \\ \psi_3 \end{pmatrix} = \begin{pmatrix} \psi_0 - i\psi_3 \\ \psi_1 - i\psi_2 \\ i\psi_1 + \psi_2 \\ i\psi_0 + \psi_3 \end{pmatrix} = \begin{pmatrix} h_0 \\ h_1 \\ -ih_1 \\ -ih_0 \end{pmatrix}, \tag{9.2}$$

where we defined $h_0 = \psi_0 - i\psi_3$ and $h_1 = \psi_1 - i\psi_2$. In other words, the lower two spin components of the result (to which we will refer as r_2 and r_3) are the same as the upper two, but multiplied by $-i$. This property is general, and applies to all of the projectors, although the details do vary with the projector in question. The action of finding the right h_0 and h_1 is known as *projection* and the placement into the lower components and the multiplies by i, etc. is called *reconstruction*. We show the details of how to perform these operations for all the projectors, with our particular choice of γ matrices in the table shown in Figure 9.1. The practical use of this property is that when we multiply the projected spinors by the U link matrices, we do not need to repeat the matrix multiplication for each of the four spin indices of ψ. Instead, we need to multiply only the two color vectors h_0 and h_1 resulting from the

$(\pm,\mu)$	h_0	h_1	r_2	r_3
$(-,0)$	$\psi_0 - i\psi_3$	$\psi_1 - i\psi_2$	ih_1	ih_0
$(-,1)$	$\psi_0 + \psi_3$	$\psi_1 - \psi_2$	$-h_1$	h_0
$(-,2)$	$\psi_0 - i\psi_2$	$\psi_1 + i\psi_3$	ih_0	$-ih_1$
$(-,3)$	$\psi_0 - \psi_2$	$\psi_1 - \psi_3$	$-h_0$	$-h_1$
$(+,0)$	$\psi_0 + i\psi_3$	$\psi_1 + i\psi_2$	$-ih_1$	$-ih_0$
$(+,1)$	$\psi_0 - \psi_3$	$\psi_1 + \psi_2$	h_1	$-h_0$
$(+,2)$	$\psi_0 + i\psi_2$	$\psi_1 - i\psi_3$	$-ih_0$	ih_1
$(+,3)$	$\psi_0 + \psi_2$	$\psi_1 + \psi_3$	h_0	h_1

FIGURE 9.1

How to compute h_1 and h_2 for the projectors $P^{\pm\mu}$, and how to reconstruct the lower spin components r_2 and r_3 of the projected result from h_1 and h_2.

projection of ψ with $P^{\pm\mu}$. These form the top two components to be accumulated in the result. We can recover the lower two components from them by the reconstruction step.

This leads us to the *pseudocode* for computing the application of the Wilson-Dslash operator to a lattice spinor shown in Figure 9.2. We assume that each array element holds complex numbers. We have a loop over the four dimensions and within each, we loop over the forward and backward

```
// Pseudocode for Wilson Dslash Operator
// array elements are assumed to be complex
// Input arrays: psi[x][color][spin]
//               U[x][mu][color][color]
//
// Output array: result[x][color][spin]
// Temporary arrays: h[color][2], and uh[color][2]
//
forall sites x {

  // Zero result for this site
  for(int spin=0; spin < 4; spin++) {
   for(int color=0; color < 3; color++) {
     result[x][color][spin] = 0;
    }
  }


  for(dim=0; dir < 4; dim++) {
    for(forw_back=0; forw_back < 2; forw_back++) {
       int neighbor;
       if ( forw_back == 0 ) {
         neighbor = forward_neighbor_index(x,dim);
      }
      else {
        neighbor = back_neighbor_index(x,dim);
     }

     for(int color=0; color<3; color++) {
       h[color][0] = project_h0(dim, forw_back, psi[neighbor][color]);
       h[color][1] = project_h1(dim, forw_back, psi[neighbor][color]);
     }

     if( forw_back == 0 ) {
       // Forward links: Multiply h by U
       uh[:][0] = mat_mult(U[dim][x], h[:][0]);
       uh[:][1] = mat_mult(U[dim][x], h[:][1]);
     }
     else {
       // Back links: Multiply h by U^\dagger
       uh[:][0] = mat_adj_mult(U[dim][neighbor], h[:][0]);
       uh[:][1] = mat_adj_mult(U[dim][neighbor], h[:][1]);
     }

    // Reconstruct bottom 2 indices and accumulate
    for(int color=0; color < 3; color++) {
       result[x][color][0] += uh[color][0];
       result[x][color][1] += uh[color][1];
       result[x][color][2] = reconstruct_r2(dim,forw_back,uh[color]);
       result[x][color][3] = reconstruct_r3(dim,forw_back,uh[color]);
    }
   } // forw_back loop
  } // loop over dim
} // loop over sites
```

FIGURE 9.2

Pseudocode for implementing the Wilson-Dslash operator.

directions. In each direction, we compute the projections h_0, h_1 for the neighboring site. Then, depending on whether they are the forward or backward neighbors, we multiply h_0 and h_1 either by U or $U^\dagger$ as appropriate to form uh_0 and uh_1. We accumulate uh_0 and uh_1 into the top two spin components of the result, and the appropriate reconstructions into the lower components of the result.

PERFORMANCE EXPECTATIONS

Let us now consider some basic performance expectations for this operation. First, we will work out the naive arithmetic intensity, in terms of the number of floating point operations (FLOP-s) needed versus the minimum useful amount of data movement. In this discussion, sign flips and multiplies by i which are basically just interchanging real and imaginary complex components and potentially flipping signs, will not be considered as floating point operations. We will also refer generically as *additions* to both additions and subtractions.

Each projection operation reads in all 4×3 (spin $\times$ color) components of a spinor, which corresponds to 12 complex numbers or 24 floating point numbers. Each projection operation of the spinor is 2 complex additions per color component, that is, 6 complex additions or 12 floating point additions.

We can consider the matrix-vector products as three complex scalar (dot) product operations. Each dot product is between a row of the matrix, and the vector with which it is multiplied. This has a floating point cost of three complex multiplies, and two complex additions. A complex multiply is made of six floating point operations. Hence, the dot product is $3 \times 6 + 4 = 22$ FLOP-s where the four comes from the two complex additions. The whole matrix vector operation is then $3 \times 22 = 66$ FLOP-s. For each neighbor, one needs to repeat this twice (for h_0 and h_1) which gives: 132 FLOP-s. In terms of memory traffic, one also needs to load the U matrix, which is $3 \times 3 = 9$ complex numbers or 18 real numbers.

While no FLOP-s are needed for the reconstruction, one does need to sum all the results. To sum the eight neighbors one needs seven spinor additions. Each of these is $4 \times 3 = 12$ complex additions, or 24 real additions. Finally, one needs to write out the result to memory, which involves saving 12 complex numbers or 24 floating point numbers.

In summary, the minimum memory traffic per output lattice site is

$$d = 8 \times 24 + 8 \times 18 + 24 = 360 \ \ \text{floating point numbers}, \tag{9.3}$$

where the first term is for the eight neighbor spinors (read), the second term is for the eight link-matrices (read), and the last term is for the writing of the result. In single precision, this comes to $d_s = 4d = 1440$ bytes, in double it is $d_d = 8d = 2880$ bytes, where the factors for 4 and 8 are the sizes of `float` and `double` in bytes, respectively.

The arithmetic involved for the same site is:

$$F = 8 \times 12 + 8 \times 132 + 7 \times 24 = 1320 \ \ \text{FLOP} - \text{s}, \tag{9.4}$$

where the three terms can be identified with the cost of eight projections, eight sets of matrix multiplies and the accumulation of the results as discussed before. This gives us a *naive arithmetic intensity* of $I_s = F/d_s = 0.92$ FLOP/byte in single precision or $I_d = F/d_d = 0.46$ FLOP/byte in double precision.

In practice, the Intel Xeon Phi coprocessor, can sustain about 150 GB/s of memory bandwidth and the 60 cores can provide approximately 2022 single precision GFLOPS. The practical single precision FLOP/byte balance point of the Intel Xeon Phi coprocessor model 5110P is thus approximately 13.5 FLOP/byte,

while in double precision it is around 6.74 FLOP/byte.[1] Hence, this naive performance model predicts that for problems which do not fit into caches, Wilson-Dslash will always be memory bound on the Intel Xeon Phi coprocessor architecture.

REFINEMENTS TO THE MODEL

Up until now, we have not considered hardware in the above discussions, however, by taking into account some hardware features, we can arrive at a more sophisticated model. While the number of *useful* FLOPS remains unchanged, we can refine the memory traffic part of the calculation. In particular, Intel Xeon Phi coprocessors feature 32 kb of L1 cache and 512 kb of L2 data-cache per core. The L2 caches appear as a unified coherent cache to the programmer through a tag directory mechanism, while Intel Xeon processors have large unified L3 caches. One can then consider a more refined model which we present below:

$$F = \frac{1320}{\frac{8G}{B_r} + \frac{(8-R)S + rS}{B_r} + \frac{S}{B_w}}, \tag{9.5}$$

where now F is the performance in GFLOPS, G is the size of the gauge matrices in bytes, S is the size of the spinors in bytes, B_r and B_w are the read-bandwidth and write bandwidths in GB/s between the memory and the *lowest level of cache*, respectively, and R is the number of neighbors per site which are cached and can be re-used without main memory traffic. The factor r is in place to allow us to consider *read-for-write* when writing. If one can use *streaming stores* which bypass the cache on write, then we can set $r = 0$. If we need to read a piece of memory to cache before writing it we can set $r = 1$. This model assumes that the cache is infinitely faster than accessing main memory. While no explicit assumption is made about the size of the cache, one can in principle attempt to capture cache-size effects by tuning the reuse factor R.

We note that we assume no reuse for the gauge fields, since the Dslash typically operates on a checkerboarded lattice as shown in Figure 9.3. When processing a point of a given color (e.g., black), one needs the links pointing to it from its backward neighbor sites of the opposite color (red in this example), and the links pointing forward from the site in question. When processing the next point of the original color, we must remember that this is not an immediate neighbor of the first one, since checkerboarding does not allow two sites of the same color to be neighbors, and hence there must be at least one site of the other color between the two under consideration, which is the backward neighbor of the second site. The second site will use the links of this intermediate site instead of those of the first site.

Additional tricks—compression

Our naive performance indicates that the problem is memory bandwidth bound, with an arithmetic intensity of around 0.92 FLOP/byte in single precision. One way to increase the arithmetic intensity is to consider gauge field compression to reduce memory traffic (reduce the size of G), and using the

[1]The peak GDDR bandwidth is quoted as 320 GB/s; however, we have never been able to reach more than around 170 GB/s in practice, hence quoting the peak value seems excessive. We will stick with a nominal 150 GB/s in the remainder of the chapter.

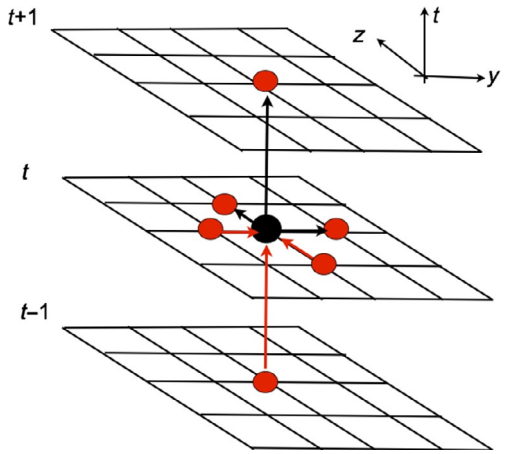

FIGURE 9.3

A graphical illustration of the Wilson-Dslash operator. Sites are checkerboarded. A given site needs its nearest neighbors (of opposite color). We have suppressed the X dimension in this figure. A site of a given color needs links of the same color emanating from it, and links of the opposite color from the backward neighbors ending on it.

essentially free FLOP-s provided by the node to perform decompression before use. This idea was explored in depth for GPU architectures in the QUDA library, and we sketch only the bare bones of it here.

Due to the $SU(3)$ nature of the gauge fields they have only eight real degrees of freedom: the coefficients of the eight $SU(3)$ generators. In principle, this means that instead of nine complex numbers, we can store the gauge fields as eight real numbers. However, re-constructing all nine complex numbers this way involves the use of some trigonometric functions.

A simpler approach is to consider two-row storage of the $SU(3)$ matrices. This idea has long been used to save space when writing gauge fields out to files, but was adapted as an on-the-fly bandwidth saving (de)compression technique (see the "For more information" section using "mixed precision solvers on GPUs"). The basic idea is to consider the rows of the matrix as row vectors:

$$\begin{pmatrix} a_0 & a_1 & a_2 \\ b_0 & b_1 & b_2 \\ c_0 & c_1 & c_2 \end{pmatrix} \quad \mathbf{a} = (a_0, a_1, a_2), \quad \mathbf{b} = (b_0, b_1, b_2), \quad \mathbf{c} = (c_0, c_1, c_2). \tag{9.6}$$

Then, if one has the first two rows: $\mathbf{a}$ and $\mathbf{b}$, both having been normalized to be of unit length, one can compute $\mathbf{c} = (\mathbf{a} \times \mathbf{b})^*$, that is, by taking the vector (cross) product of $\mathbf{a}$ and $\mathbf{b}$ and complex conjugating the elements of the result. This trick is quite simple, and reduces the size of the gauge links to 6 complex numbers, or 12 real numbers.

Now considering the formula in Eq. (9.5), we can compute the expected effects of neighbor spinor reuse, two-row compression and streaming stores. We make the simplifying assumption that $B_w = B_r$, and then can divide out the bandwidth, to get the arithmetic intensity. We show some results in the table shown in Figure 9.4. First, we note that even the naive arithmetic intensity of 0.92 FLOP/byte we computed initially, relies on not having read-for-write traffic when writing the output spinors, that is, it

R	Compress	Streaming Store	A/I (FLOP/Byte)
0	No	No	0.86
0	No	Yes	0.92
7	No	No	1.53
7	No	Yes	1.72
7	Yes	No	1.96
7	Yes	Yes	2.29

FIGURE 9.4

The expected effects of neighbor spinor reuse, compression, and streaming stores on the arithmetic intensity of Wilson-Dslash in single precision, with the simplifying assumption that $B_w = B_r$

needs streaming stores, without which the intensity drops to 0.86 FLOP/byte. Second, we see that by being able to reuse seven of our eight neighbor spinors, we can significantly improve in performance over the initial bound, to get an intensity between 1.53 and 1.72 FLOP/byte, depending on whether or not we use streaming stores. Finally, we see that we can benefit even further from gauge compression, to reach our highest predicted intensity of 2.29 FLOP/byte when cache reuse, streaming stores and compression are all present. We note when considering compression, we ignored the extra FLOP-s needed to perform the decompression, and counted only the *useful* FLOP-s.

FIRST IMPLEMENTATION AND PERFORMANCE

Before looking at the optimized implementation, let us take a look at a real-world implementation of Dslash which has been in production use for some years in the Chroma code. The package is called `cpp_wilson_dslash` and can be obtained from GitHub (see the "For more information" section). It is an offshoot of prior work by Chris McClendon. It is also in the included code bundle in the `Code/Original/cpp_wilson_dslash` directory. The package contains versions of the code for both single, and multinode versions of the Dslash operator, and we will focus our attention here on the single node, multithreaded version.

The code uses an "array of structures" (AoS) approach for the fields, keeping the internal indices of the spinors and gauge fields as local arrays as defined in `include/cpp_dslash_types.h`. We show the single precision versions of these types in Figure 9.5.

The structure of the `GaugeMatrix` (3×3 complex matrix) and of the `FourSpinor` (4×3 complex matrix) should be recognizable. The `HalfSpinor` type is the result of the projection, and because the code was targeted originally for Streaming SIMD Extensions (SSE), we have swapped the color and spin indices compared to the `FourSpinor` so that coupled with the two complex components they make up a group of four floats for SSE.

```
namespace Dslash32BitTypes {
 typedef float FourSpinor[4][3][2];
 typedef float HalfSpinor[3][2][2];
 typedef float GaugeMatrix[3][3][2];
}
```

FIGURE 9.5

Single precision data types for the naive-Dslash operator.

```
// The operator is coded as a function-object
// res and psi are really FourSpinor* objects cast to float*
// u is a pointer to the gauge fields of which there are 4 per site
void Dslash<float>::operator() (float* res,
                                float* psi,
                                float *u, /* Gauge field suitably packed */
                                int isign, /* apply D or D^\dagger */
                                int cb) /* target checkerboard */
{
  if (isign == 1) {

    // Dispatch our DPsiPlus kernel to the threads
    CPlusPlusWilsonDslash::dispatchToThreads((void (*)(size_t, size_t, int,
         const void *))&DPsiPlus,
                                             (void*)psi,
                                             (void*)res,
                                             (void *)u,
                                             (void *)s,
                                             1-cb,
                                             s->totalVolCB());


  }
  // below would follow the D^\dagger case, very similar to the D
  // case, so we omit it for space
```

FIGURE 9.6

Dispatching the site loop for Dslash over available threads.

The code already features threading. Looking at Figure 9.6 from the file `lib/cpp_dslash_scalar_32bit.cc` we see that the first major step of the user called `operator()` method is a dispatch of a function to the available threads.

The OpenMP version of the `dispatchToThreads` function is found in the file `lib/dispatch_scalar_openmp.cc` and is also shown in Figure 9.7. An important point to note here is that the

```
void dispatchToThreads(void (*func)(size_t, size_t, int, const void *),
                       void* source,
                       void* result,
                       void *u,
                       void *s,
                       int cb,
                       int n_sites)
{
  ThreadWorkerArgs a;
   int threads_num,myId,low,high;

  a.psi = source; a.res = result; a.u = u; a.cb = cb; a.s = s;

#pragma omp parallel shared(func, n_sites, a)                          \
  private(threads_num, myId, low, high) default(none)
    {

      threads_num = omp_get_num_threads();
      myId = omp_get_thread_num();
      low = n_sites * myId/threads_num;
      high = n_sites * (myId+1)/threads_num;
      (*func)(low, high, myId, &a);
    }
}
```

FIGURE 9.7

The OpenMP thread dispatch in the naive code.

dispatcher tries to split the total work (`n_sites`) into chunks. Each chunk is specified by a beginning (`low`) site and a last (`high`) site. As we will see when looking at the function `DPsiPlus`, each thread then traverses its own chunk in lexicographic order. We illustrate this schematically in Figure 9.8, with boxes around groups of sites representing the chunk belonging to a single thread, and arrows between the sites representing the order of traversal.

The main compute kernels are the routines `DPsiPlus` for Wilson-Dslash and `DPsiMinus` for its Hermitian conjugate. We show a snippet of `DPsiPlus` in Figure 9.9. We deleted some of the details, to fit it into the figure, but as usual the full source is available in the Code package in the `cpp_dslash_scalar_32bit.cc` file in the `lib/` subdirectory. The structure should be straightforward though. After unwrapping the arguments from the input `ptr` pointer, we enter a site loop from the `low` to the `high` index. We work out our "site" (`thissite`) from the current index in the lattice, and first find our forward and backward neighbors from the `ShiftTable` object s using the `forwardNeighbor` and `backwardNeighbor` methods, in direction 0. We set the input spinor pointer `sp1` to the forward neighbor, the gauge pointer `up1` to the gauge field, and we invoke `dslash_plus_dir0_forward` which performs the projection, multiplies, and reconstructs and accumulates into half spinors `r12_1` and `r34_1`. We then go through all the directions until we get to the last one, where we call `dslash_plus_dir3_backward_add_store` and store the result at pointer `*sn1`. The various utility functions, such as `dslash_plus_dir3_backward_add_store` can be found in the `include` directory in the `cpp_dslash_scalar_32bit_c.h` source file as functions to be inlined. There is also a version which has the functionality coded with SSE intrinsics in `cpp_dslash_scalar_32bit_sse.h`.

Finally, there are testing and timing routines in the `tests/` subdirectory. A test routine in the `testDslashFull.cc` function will test the Dslash operator against a slower variant coded in the QDP++ framework. This test is run from the code `t_dslash.cc`, whereas a straightforward timing harness, which executes Dslash in a loop and times the execution is in the `timeDslash.cc` file, that is called by the `time_dslash.cc` main-program. For both of these tests, one can choose the lattice volume and the number of timing iterations in the file `testvol.h`. For a many-core device like Intel Xeon Phi coprocessor, a reasonable lattice size is $32 \times 32 \times 32 \times 64$ sites, which we will use throughout.

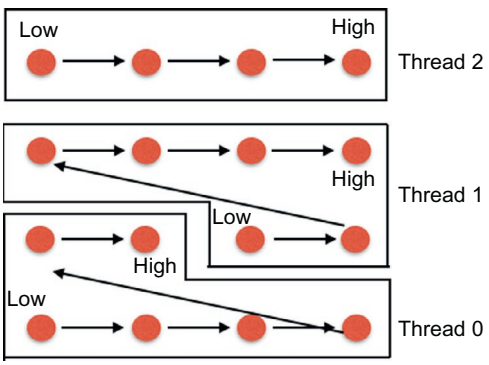

FIGURE 9.8

A schematic illustration of splitting lattice between three threads in the naive code. Within the chunks belonging to the individual threads the traversal is lexicographic.

```
void DPsiPlus(size_t lo, size_t hi, int id, const void *ptr)
{
  const ThreadWorkerArgs *a
    = (const ThreadWorkerArgs*)ptr;  /* Cast the (void *) to an (
        ThreadWorkerArgs*) */

  // Unwrapping of params deleted for space
  // please see source code...
  ...

  for (int ix1 = low; ix1 < high; ix1++)  {

    int thissite = s->siteTable( ix1 );
    int fsite = s->forwardNeighbor(ix1,0);
    int bsite = s->backwardNeighbor(ix1,0);

    /****************************** direction +0 ******************/
    /* ...(1-isign*gamma(0)) psi(x + \hat{0}) */
    sp1 = &psi[ fsite  ];
    up1 = &(gauge_field[ix1][0]);
    dslash_plus_dir0_forward(*sp1, *up1, r12_1, r34_1);

   sm1 = &psi[ bsite ];
    um1 = &(gauge_field[bsite][0]);
    dslash_plus_dir0_backward_add(*sm1, *um1, r12_1, r34_1);

    ... // Deleted branches in other directions for space

    // Last direction...
    sm1 = &psi[bsite];
    um1 = &(gauge_field[bsite][3]);
    sn1 = &res[ix1];  /*we always walk across the result
        lexicographically */

    dslash_plus_dir3_backward_add_store(*sm1, *um1, r12_1, r34_1, *sn1);
  } // for loop
} // function
```

FIGURE 9.9

The site loop in the `DPsiPlus` function, as executed by each thread.

In terms of timing benchmarks, we typically set the iterations so that the test will run for a couple of seconds which can typically be achieved with about 400-500 iterations. However, for purposes of performance measurement in VTune, we often use a smaller number of iterations, for example, 10 or 50.

RUNNING THE NAIVE CODE ON INTEL XEON PHI COPROCESSOR

We ran the correctness and timing tests with the above lattice size on 59 cores (to stay off the last core with O/S functions) of an Intel Xeon Phi 5110P coprocessor on which we have enabled the *icache_snoop_off* option in the kernel to gain additional memory bandwidth. We went "all in," asking for 4 threads per core, or 236 threads. To ensure that only the first 59 cores were used we set compact affinity using the `KMP_AFFINITY` environment variable. After verifying correctness, we achieved the output from the timing test shown in Figure 9.10 showing a performance of 34.8 GFLOPS in single precision, and around 25.2 GFLOPS in double precision.

This figure seems somewhat low. How does it compare to running on a Intel Xeon processor? As a comparison, we ran on a dual socket Intel Xeon E5-2560 (codenamed Sandy Bridge) processor, 2.0 GHz,

```
QDP uses OpenMP threading. We have 236 threads
Lattice initialized:
  problem size = 32 32 32 64
  layout size = 32 32 32 64
  logical machine size = 1 1 1 1
  subgrid size = 32 32 32 64
  total number of nodes = 1
  total volume = 2097152
  subgrid volume = 2097152
Finished init of RNG
Finished lattice layout
Running Test: timeDslash

         Timing with 50 counts
         50 iterations in 1.988364 seconds
         39767.28 u sec/iteration
         Performance is: 34805.5064364473 Mflops (sp) in Total
         Performance is: 34805.5064364473 per MPI Process

         Timing with 50 counts
         50 iterations in 2.748181 seconds
         54963.62 u sec/iteration
         Performance is: 25182.4810665673 Mflops (dp) in Total
         Performance is: 25182.4810665673 per MPI Process
  OK
Summary: 1 Tests Tried
         1 Tests Succeeded
         0 Tests Failed on some nodes
of which 0 Tests Failed in Unexpected Ways on some nodes
```

FIGURE 9.10

Timings of the naive Dslash on an Intel Xeon Phi coprocessor 5110P using 59 cores, or 236 threads. We see a single precision performance of 34.8 GFLOPS in single precision and 25.2 GFLOPS in double precision.

with 2×8 cores, and hyperthreading enabled, giving us 32 threads. Further, since we know that the memory allocation in this code is not NUMA aware we used the `numactl` utility to interleave memory allocation between the sockets using the `numactl --interleave=0,1` command to launch the code. Without enabling the SSE optimization code-branches, the result we obtained on the processor was 34.7 GFLOPS in single precision and 19.3 GFLOPS in double. So a single coprocessor performed roughly as fast (slightly faster) than a dual socket Xeon system using the same code.

While this is not an unreasonable first performance *ratio* between Intel Xeon processor and Intel Xeon Phi coprocessor, the result is nonetheless somewhat disappointing in terms of absolute performance. Our models tell us that the code should be memory bandwidth bound, so one immediate question is how much memory bandwidth are we utilizing? To answer this we profiled the code using VTune, using its bandwidth collection experiments which reported the code sustaining a memory bandwidth of 42.3 GB/s in the single precision Wilson-Dslash on the Intel Xeon Phi coprocessor.

EVALUATION OF THE NAIVE CODE

At this point we should pause and reflect. VTune reported that our single precision memory bandwidth on Intel Xeon Phi coprocessor was 42.3 GB/s. Taking the most naive arithmetic intensity of 0.86 FLOP/byte (no caching, no compression, no streaming stores), the model predicts a performance of 36.4 GFLOPS as our maximum. We are sustaining 34.8 GFLOPS of this, which is around 96% of the model prediction. However, the 42.3 GB/s is quite a low bandwidth for the Intel Xeon Phi coprocessor.

Second, right now we are not really using the vector units on the Intel Xeon Phi coprocessor, at least not deliberately. If the compiler were to generate code to carry out our scalar arithmetic using just one lane of the vector units, then the peak performance we can expect is two flops per cycle (a multiply and add), which at 1.053 GHz from 60 cores is 126.36 GFLOPS. If we could exhaust our bandwidth of 150 GB/s our most naive arithmetic intensity of 0.86 FLOP/byte predicts a performance of 129 GFLOPS. This is a little higher than, but still in the same ballpark as, the peak performance of unvectorized arithmetic. However, our most optimistic intensity of 2.29 FLOP/byte (cache reuse, compression, streaming stores) would predict 343.5 GFLOPS, which is substantially more than can be achieved with unvectorized arithmetic. Therefore, we need to ensure we make more effective use of the vector units than is done in the current version of the code.

Finally, our measured performance (34.8 GFLOPS) is commensurate with not getting any cache reuse at all ($R = 0$), stopping us from attaining our more optimistic model estimates. Therefore, it is necessary to improve our cache reuse.

OPTIMIZED CODE: QPhiX AND QPhiX-CODEGEN

In this section, we will illustrate how to solve the problems of the original code regarding cache reuse, vectorization, and better memory handling (prefetching). We will use the *QPhiX* code which is available on GitHub, and is also included in the code-package for this chapter. QPhiX is essentially an evolution of `cpp_wilson_dslash` and has two main parts: First, it deals with looping on the lattice to implement the technique of 3.5D blocking and it also implements a heuristic-based load balancing scheme to schedule the blocks to the cores. Second, it contains *kernels* to evaluate Dslash, on a *tile of sites* in parallel, in order to efficiently utilize the vector units of the Intel Xeon Phi coprocessor. These kernels are generated by a *code-generator* in a package called `qphix-codegen` and can then be copied into the `qphix` source package. The current `qphix` package on GitHub contains the best known performing kernels already pregenerated. In the Code package, we provide several variants of `qphix` with different selections of kernels to illustrate our points regarding optimization. Before launching into the optimizations, let us consider the important features of QPhiX and the code-generator.

DATA LAYOUT FOR VECTORIZATION

In our first attempt to optimize Dslash, we wanted to vectorize wholly over the X dimension; however, this proved rather restrictive. Since we split our lattice into even and odd subsets by dividing the global X dimension, for a lattice of dimensions $L_x \times L_y \times L_z \times L_t$ we have checkerboarded subsets of size $L_x/2 \times Ly \times L_z \times L_t$. For full vector utilization in single precision (16 floats), we would have had to have L_x be multiples of 32. While this is fine for our working example of $L_x = 32$, it places a strong constraint on our choice of problem sizes. To ameliorate this problem we have decided to vectorize over X-Y tiles. We could fill the length 16 vector registers using either 16 sites along X, a tile of size 8×2 in the X-Y plane, or a tile of size 4×4 in the X-Y plane. By using the gather engine on Intel Xeon Phi coprocessor, we could have been even more flexible, but we found that using *load-unpack* and *pack-store* instructions was more efficient, and it allowed us these combinations. As a result, instead of the *AoS* layout we had in our original code, we have opted to use a *Array of Structures of Arrays* (AoSoA) layout which we define in the file `geometry.h` in the `include/qphix` directory of the `qphix` package.

We have two key parameters which we supply as templates to almost every component of *QPhiX*. The length of the vector unit which we call `VECLEN` or `V` and the length of the inner arrays of the AoSoA structures, which we call the SOA length `SOALEN` or `S` in the code. We show the basic data types for Spinors and Gauge fields in QPhiX in Figure 9.11. We see that the `FourSpinorBlock` has an innermost array dimension of `S`, and we will have to *gather* `ngy=V/S` such blocks into a vector register. These blocks will come from `ngy` consecutive values of the *y* coordinate as shown in Figure 9.12.

The `TwoSpinorBlock` type is used only internally in our multinode implementation to store the results of projections on the faces of the lattice. By this time, the gathers have taken place prior to the projection so it is fine to leave the inner length as `V` for this type.

We have several comments about the type `SU3MatrixBlock` which represents the gauge fields. First, we have made the choice to double copy the gauge field, which means we store not just the four forward links from each site, but also the four backward links. This was done so that the fields could be accessed in unit stride access, hence the leftmost array dimension is eight, for the eight directions. Second, the gauge fields are read only, never written, and so, we could absorb the gather operation of the spinor blocks into a layout re-definition, essentially *pregathering* the gauge field. Hence, it has `V` as its innermost length. Finally, we implemented the two-row compression technique. It has turned out that, for Intel Xeon processors, the hardware prefetcher would still read the third row of the gauge fields, even when we did not reference its elements directly in the code, and thus we did not realize the benefits of compression on processors initially (although we did on Intel Xeon Phi coprocessors). One way to

```
template<typename T, int V, int S, const bool compressP>
 class Geometry {
 public:
   // Later change this to depend on compressP
   typedef T  FourSpinorBlock[3][4][2][S];
   typedef T  TwoSpinorBlock[3][2][2][V];
   typedef T  SU3MatrixBlock[8][ ( compressP ? 2 : 3 ) ][3][2][V];
   ...
```

FIGURE 9.11

The basic data types in QPhiX.

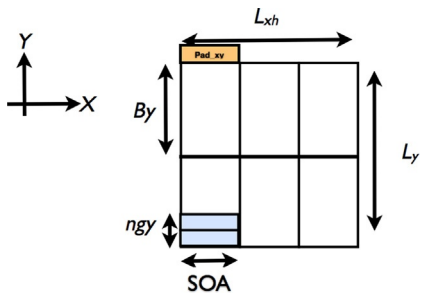

FIGURE 9.12

Schematic view of the data layout. We show blocking in the *Y*-direction with block length B_y, and vectorization in the *X*-*Y* plane, where blocks of length `SOA` in *X* are gathered from `ngy` consecutive values of the *Y*-coordinate. Additionally the padding can be introduced between *X*-*Y* slices of the lattice (Pad_xy in the figure) and also between *XY Z* volumes (Pad_xyz—not shown).

circumvent this was to reduce the number of rows in the data structure explicitly from 3 to 2, when compression is enabled. However, this is a compile time decision, hence we have an extra template parameter: `compressP` and the number of rows is decided based on this template at compile time. Finally, we added the option of padding our fields both following each *X-Y* plane of the lattice and after every *X-Y-Z* subvolume.

3.5D BLOCKING

The basic strategy in QPhiX was to implement a technique known as 3.5D blocking. In the form previously discussed for Intel Xeon (codenamed Westmere) processors in "High-performance lattice QCD for multicore-based parallel systems using a cache-friendly hybrid threaded-MPI approach" (see the "For more information" section), considered vectorizing over the *X* dimension of the lattice, and blocking over the *Y* and *Z* dimensions, while streaming up through the *T* dimension. With the tiled layout for vectorization, the idea still holds, except the *Y-Z* blocks now must contain an integer number of tiles. The idea here can be followed looking at Figure 9.3. As one scans along a *Y-Z* block, at *T* coordinate *t*, one needs in principle to bring in only the neighbor with coordinate $t + 1$. The co-planar *X-Y-Z* neighbors should have been brought into cache when working on the previous *T*-slice. The *Y-Z* plane of the lattice can then be divided into blocks, resulting in *Y ZT* bricks, which can be assigned to the individual cores as shown in Figure 9.13. The width of these blocks can be denoted B_y and B_z, and need to be chosen so that at least three *T*-slices of spinors fit into cache. Data that is on the *Y-Z* boundaries of a

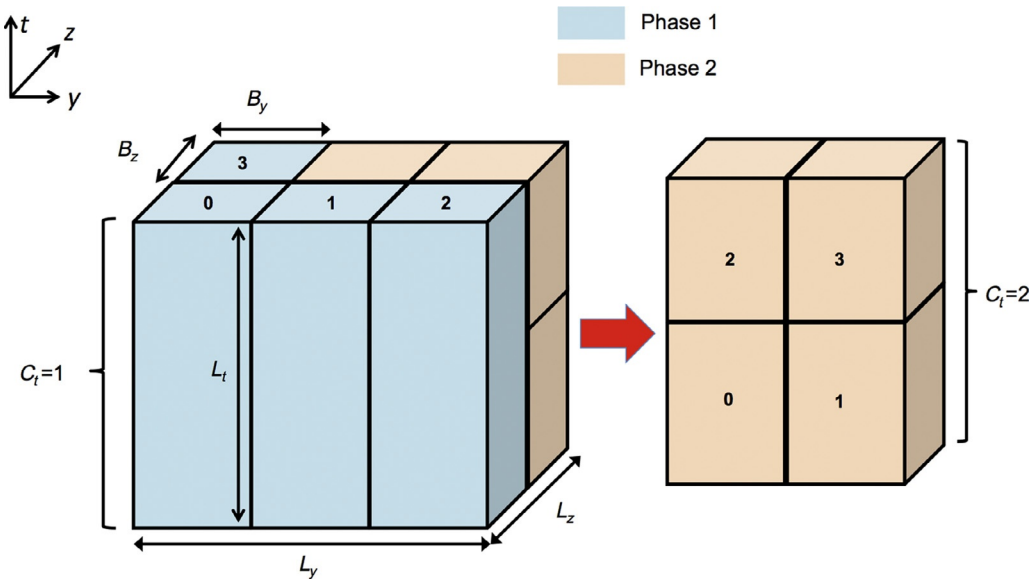

FIGURE 9.13

We show our load balancing scheme, with six blocks scheduled over four cores. In the first round, each core is assigned a full block. In the second round, the two remaining blocks are split in the time direction, to yield four blocks, each with shorter *T*-extent. These are then scheduled to the four cores, keeping all four cores occupied.

$B_y \times B_z$ block is needed both by the block, and its neighbor, however, if the computations are appropriately orchestrated and the cores working on the neighboring blocks are synchronized, this may result in both cores being able to use the data in each other's L2 cache rather than going to main memory.

LOAD BALANCING

It may be the case, depending on the lattice size and B_y and B_z, that there are comparatively few blocks given the number of cores. For example, in our test case of $32^3 \times 64$ sites, with $B_y = B_z = 4$ on Intel Xeon Phi coprocessor, one has only 64 blocks. This could result in two rounds of block processing, one in which up to 60 cores are fully engaged, and a second round where only 4 blocks are working and the remaining 56 are idle. To ameliorate this situation, we developed a heuristic load balancing scheme, which we describe below and show in Figure 9.13.

We perform the traversal of blocks in *phases*. In any given phase, if there are more blocks than cores, we simply assign one block per core, and all the cores are fully occupied. We keep repeating this until we have either finished processing all the blocks or we arrive at a phase that we have fewer blocks than cores. Once we have fewer blocks then cores, we consider splitting them in the time direction, which will increase the number of remaining blocks. If the number of blocks is now more than or equal to the number of cores, we carry on as before, and so on. Ultimately, the time-lengths of the blocks may be so short, that it will not be worth splitting them further. At this point, we take the load imbalance and finish the work in the final phase.

In terms of implementation, we keep a list of blocks and the number of splits in the time dimension; C_t; for each phase. We can compute the block assignments for each phase up front on constructing the Dslash. In the QPhiX code, the basic phase info is set up in the `geometry.h` file in the `include/qphix` directory, as shown in Figure 9.14. In the code, `rem` stores the number of remaining blocks. This is initialized to `ly*lz` where `ly` and `lz` are the number of blocks in Y and Z, respectively. In each iteration, a value of `p.Ct` is computed. Depending on `p.Ct` the number of cores over which the Y-Z blocks can be allocated is found: `p.Cyz`. The first block for this phase is noted in `p.startBlock` after which `p.Cyz`

```
int ly = Ny_ / By;
int lz = Nz_ / Bz;
int rem = ly * lz;
int stblk = 0;
n_phases = 0;
int n_cores_per_minct = num_cores / MinCt;
while(rem > 0) {
  int ctd = n_cores_per_minct / rem;
  int ctu = (n_cores_per_minct + rem - 1) / rem;
  CorePhase& p = getCorePhase(n_phases);
  p.Ct = (ctu <= 4 ? ctu : ctd)*MinCt;
  p.Cyz = num_cores / p.Ct;
  if(p.Cyz > rem) p.Cyz = rem;
  p.startBlock = stblk;
  stblk += p.Cyz;
  rem -= p.Cyz;
  n_phases++;
}
```

FIGURE 9.14

Load balancing phase set-up in `geometry.h`.

blocks are removed from the block-list and the next phase is processed. In addition, one can set a minimum value of C_t (known as `minCt`), which can be useful for example, in a multisocket situation where `minCt` can be set to the number of sockets. When allocating memory, the phases are traversed and the right core can touch the appropriate area of memory in a cc-NUMA aware implementation. Our heuristic termination criterion for the splitting is to not split the T dimension into more than four `MinCt` blocks.

This concludes the setup of the assignment of cores to phases; however, it is also useful to be able to look up information about how the blocks are assigned to phases. This is done in the *constructor* for Dslash in the file `dslash_body.h` in `include/qphix` that we show in Figure 9.15. Here, for each thread, we set up which block it should process in each phase. For each phase, the thread ID is split into a core-ID and SMT-thread ID. The core-ID (`cid`) is split into a T-coordinate and YZ-coordinate (`binfo.cid_t` and `binfo.cid_yz`, respectively) and the coordinates of the block origin, `binfo.by`, `binfo.bz`, and `binfo.bt` are computed along with the temporal extent for the blocks `binfo.nt`.

SMT THREADING

With four SMT threads available on each core of Intel Xeon Phi coprocessor, we have a variety of ways we can imagine traversing a Y-Z block. We show three possible ways in Figure 9.16. In the end, we have opted to let the user specify the "shape" of the SMT threads, by specifying a S_y and S_z as the dimensions of a Y-Z grid. The lattice sites in a Y-Z block are then interleaved between the SMT threads. It is simplest to describe this in the case of the Y interleaved case ($S_y = 4, S_z = 1$). In this instance, thread 0 will work with a site that has Y-coordinate y, thread 1 will work with the site with Y-coordinate $y + 1$, thread 2 will

```
#pragma omp parallel shared(num_phases)
    {
      int tid = omp_get_thread_num();
      int cid = tid / n_threads_per_core;
      int smtid = tid - n_threads_per_core * cid;
      int ly = Ny/By;

      for(int ph =0; ph < num_phases; ph++){
        const CorePhase& phase = s->getCorePhase(ph);
        BlockPhase& binfo = block_info[num_phases*tid+ph];

        int nActiveCores = phase.Cyz * phase.Ct;
        if( cid > nActiveCores )  continue;
        binfo.cid_t = cid / phase.Cyz;
        binfo.cid_yz = cid - binfo.cid_t * phase.Cyz;
        int syz = phase.startBlock + binfo.cid_yz;
        binfo.bz = syz / ly;
        binfo.by = syz - binfo.bz * ly;
        binfo.bt = (Nt*binfo.cid_t) / phase.Ct;
        binfo.nt = (Nt*(binfo.cid_t+1)) / phase.Ct - binfo.bt;
        binfo.by *= By;
        binfo.bz *= Bz;
        int ngroup = phase.Cyz*Sy*Sz;
        binfo.group_tid = tid % ngroup;
      }
    } // OMP parallel
```

FIGURE 9.15

Load balancing block phase set-up in `dslash_body.h`.

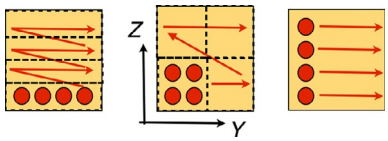

FIGURE 9.16

We show three possible ways for four SMT threads to traverse the sites in a block: lexicographic in *Y* interleaved (left), a 2 × 2 block in *Y-Z* interleaved (middle) or scanning along *Y* for four separate *Z*-coordinates interleaved (right).

work with *Y*-coordinate $y + 2$, and thread 3 will work with $y + 3$. Then thread 0 will work with $y + 4$, thread 1 will work with $y + 5$, and so forth traversing the block essentially lexicographically. Another alternative would be to assign *sub-blocks* to each SMT thread. So, for example, in a block with $(B_y,B_z) = (4,4)$, we could have each thread work with a 2 × 2 sub-block. However, we restricted ourselves to the interleaved traversals in the hope that interleaving the SMT threads among the sites would improve sharing in the L1 cache.

LATTICE TRAVERSAL

Armed now with knowledge about the phases from load balancing, the SMT thread layout, and the specifics of the vectorization, we can look at a simplified form of the traversal of the lattice. We show some code from the file `dslash_body.h` from the function `DyzPlus` in Figure 9.18, where we have removed a lot of the minor details and computation of prefetch pointers for clarity. The full code is available in the source package. The crucial points to make are the following: each thread loops over the phases, first identifying its block in that phase or whether it is idle. Once the block is found, looping is over *T*, *Z*, and *Y* as we expect. The loop over *Z* is done in increments of `Sz` and the loop over *Y* is done in increments of `ngy*Sy` (actually misspelled `nyg` in the code) where *Sy* and *Sz* are the SMT thread grid dimensions. The *X* loop is over SOAs (`nvecs` is the number of SOA lengths in the checkerboarded *X* dimensions).

Accessing the neighbors of the current *X-Y* tile in the *Z* and *T* dimensions is straightforward, and their tiles can be identified as soon as the *Z* and *T* coordinates are known. They are pointed to by pointers `tfBase` (*T* forward), `tbBase` (*T* backward), `zfBase`, and `zbBase`.

However, because of vectorization in *X* and *Y*, accessing the neighbors in these dimensions is tricky as it may need data from neighboring tiles. We show a simple case in Figure 9.17. Depending on the *Y*-coordinate, the first site of a given color (e.g., red) may be truly the first site for that *Y*-coordinate in the lattice (xodd=0 in the figure). In that case, its forward neighbors come from the first tile of the other checkerboard in a straightforward way. However, it can be that the first site of the given checkerboard is actually the second site in the lattice (xodd=1). In this case, the forward neighbor sites actually come from two different tiles of the other color and must be blended together. Further, due to periodic boundary conditions, one has to separate the cases for when the output tile under consideration is at the beginning or end of the *X* dimension (with wraparound to the tiles at the opposite end) or not. The necessary offsets are precomputed in the constructor into arrays `xb0ffs_x0_xodd` (back neighbors at start of *X* dimension) and `xf0ffs_xn_xodd` (forward neighbors at end of *X* dimension), and

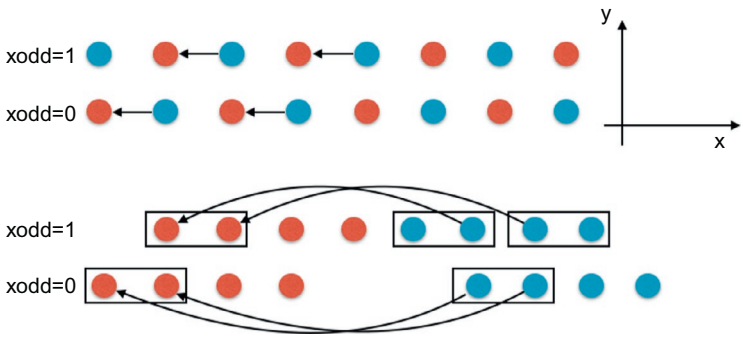

FIGURE 9.17

Locating neighbors in the forward *X*-direction, in a simple case with an *X*-*Y* tile size of 2×1. In the top diagram, we show the sites as they appear in the lattice. We show the forward neighbors needed by the first two red-sites in each row by connecting them to their partner sites with arrows. In the lower diagram, we have separated the differently colored sites into separate arrays, which are then tiled over. For the first red tile on line xodd=0, one needs only the same corresponding blue tile. However, for the first red tile on line xodd=1, the neighboring blue sites come from two blue tiles and need to be blended.

`xb0ffs_xodd` (back *X* neighbor offsets) and `xf0ffs_xodd` (forward *X* neighbor offsets) for the other cases. Likewise, similar offset arrays exist to identify neighbors in *Y*.

The last steps in the *X*-loop in Figure 9.18 are to identify whether one has the case of xodd=0, or xodd=1, and to select from the precomputed offsets appropriately depending on the *x*-coordinate into `xf0ffs`, `xb0ffs` (backward and forward *X* offsets, respectively) and similarly into `yb0ffs` and `yf0ffs` for the *Y*-direction. The pointer `oBase` is the pointer to the base of the *Y*-*Z* tile of the output spinor, `gBase` is the pointer to the appropriate portion of the gauge field. Other quantities computed (in the source but not printed here) are mostly prefetch pointers for the next iteration of work for this thread.

Once everything is worked out, the code finally calls the `dslash_plus_vec` function which carries out the work of the Dslash for the pointers identified in this loop. This is code that is generated by the *code-generator*, which we now turn to consider.

CODE GENERATION WITH QphiX-CODEGEN

We now consider how we write the kernels for the key functions `dslash_plus_vec` and its variants. To do this we decided to write a code generator, which can be found in the code distribution under the subdirectory `qphix-codegen` but can also be obtained from GitHub (see the "For more information" section).

Our initial motivations for writing the code-generator were the following: first, we wanted to eventually insert prefetches into the code to improve memory traffic. This was awkward to do when we had the code split into lots of little inline functions. It made sense to have a little code generator which could produce the fully inlined body of the Dslash kernel for an entire vector (and later *X*-*Y* tile). Using the code generator also allowed us to use compiler intrinsics and access streaming stores, gathers, load-pack, and other instructions directly. A second feature of the code-generator was once we had written

```
int num_phases = s->getNumPhases();
 for(int ph=0; ph < num_phases; ph++) {
   const CorePhase& phase = s->getCorePhase(ph);
   const BlockPhase& binfo = block_info[tid*num_phases + ph];
   ...
   // Loop over timeslices
   int Nct = binfo.nt;
   for(int ct = 0 ; ct < Nct; ct++) {
     int t = ct + binfo.bt;  // add origin to local loop variable
     ...
     // Loop over z. Start at smtid_z and work up to Ncz
     // in steps of Sz
     // (Ncz truncated for the last block so should be OK)
     for( int cz = smtid_z; cz < Bz; cz += Sz ) {
         int z = cz + binfo.bz; // Add on origin of block
         ...
         // Base address for XY tile for neighbors.
         const FourSpinorBlock *xyBase = &psi[t*Pxyz+z*Pxy];

         // Base address for neighbor tiles in Z & T
         const FourSpinorBlock *zbBase = &psi[t*Pxyz] + ... ;
         const FourSpinorBlock *zfBase = &psi[t*Pxyz] + ... ;
         const FourSpinorBlock *tbBase = &psi[z*Pxy] + ...;
         const FourSpinorBlock *tfBase = &psi[z*Pxy] + ...;

         // Base address for output
         FourSpinorBlock *oBase = &res[t*Pxyz+z*Pxy];
         ...
         // Loop over 'y' in nyg * Sy steps (should be ngy :( )
         for( int cy = nyg*smtid_y; cy < By; cy += nyg*Sy ) {
           int yi = cy + binfo.by;  // Offset local y-coordinate with
                block origin
           // work out if first x-coordinate is on boundary...
           const int xodd = (yi + z + t + cb) & 1;

           // cx loops over the soalen partial vectors
           for(int cx = 0; cx < nvecs; cx++) {
             // Base address for gauge fields
             const SU3MatrixBlock *gBase = &u[...];
             int X=nvecs*yi+cx;
             ...
             // Offsets for gathering neighbors from back and forward x
             // may be in different tiles
             xbOffs = (cx == 0 ? xbOffs_x0_xodd[xodd] : xbOffs_xodd[xodd])
                  ;
             xfOffs = (cx == nvecs-1 ? xfOffs_xn_xodd[xodd] : xfOffs_xodd[
                  xodd]);
             ...
             // Offsets for gathering neighbors from back and forward Y
             // may be in different tiles
             ybOffs = (yi == 0 ? ybOffs_y0 : ybOffs_yn0);
             yfOffs = (yi == Ny - nyg ? yfOffs_yn : yfOffs_ynn);
             ...
             // Call kernel with appropriate pointers
             dslash_plus_vec<FT,veclen,soalen,compress12>(...)
           }
         } // End for over scanlines y
       } // End for over scalines z
       if( ct % BARRIER_TSLICES == 0 ) barriers[ph][binfo.cid_t]->wait(
           binfo.group_tid);
     } // end for over t
  } // phases
```

FIGURE 9.18

The basic loop structure in QPhiX, with a lot of auxiliary computations removed for clarity such as the indices of the next site, prefetch computations, etc. The full code is available in `include/dslash_body.h`, for example, in the function `DyzPlus`.

the code for Xeon-Phi intrinsics, it was straightforward also to replicate the features for AVX intrinsics to target Intel Xeon processors. Currently we feature a "scalar" target (no vectorization) and there are even targets for BlueGene/Q vectorization with QPX intrinsics, and Intel AVX2, although these latter targets are still experimental.

We should note that the use of small domain-specific code generators is not a new idea in itself. In QCD, there have been several frameworks including BAGEL and QA0. The QUDA library for GPUs has also at one point used an internal Python code generator. The BAGEL generator, for example, also performs some simulation of architectural pipelines and generates assembly code rather than intrinsics. In our case, the simple generator we wrote was more for convenience and saving typing when trying to re-space prefetches.

QphiX-CODEGEN CODE STRUCTURE

The code generator is called `qphix-codegen` and can be found as the subdirectory of the same name within the code-package. In `qphix-codegen`, we consider three primary objects: *instructions*, *addresses*, and *vector registers*. These are defined in the `instructions.h` and `address_types.h` files. In particular, the vector registers are referred to as `FVec`, and instructions and addresses are derivations of the base `Instruction` and `Address` classes. We also distinguish between regular `Instructions` and those that access memory (`MemRefInstruction`-s).

The `FVec` objects contain a "name" which will be the name of the identifier associated with the FVec in the generated code. All instructions and addresses have a method called `serialize()` which return the code for that instruction as a `std::string`. Since we are generating code, we need a couple of auxiliary higher level "instructions" to add conditional blocks, scope delimiters, or to generate declarations.

Ultimately, the code-generator generates lists of `Instruction`-s that are held in a standard vector from the C++ standard library. We alias the type of such a vector of instructions to type `InstVector` (for Instruction Vector). In turn, the instructions reference `FVec` and `Address` objects.

The remaining attributes for addresses and instructions were mostly added so we can perform analysis on the generated code. For example, one could look for `MemRefInstructions`, and extract their referenced `Address`-es for automatic prefetch generation, or to count the balance of arithmetic versus memory referencing instructions.

Finally, at the end of the file `instructions.h` we define some utility functions such as `mulFVec` that take an instruction vector, two `FVec`-s from which they generate a `MulFVec` object and insert is into the instruction vector. The majority of the code for the Dslash is written with these utility functions.

IMPLEMENTING THE INSTRUCTIONS

The actual implementations of the `Instructions` are in a variety of files with names like `inst_sp_vec16.cc`? that, in particular, refers to generating single precision code working with length 16 vectors, that is for Intel Xeon Phi coprocessor. First and foremost we declare the actual types of `FVec`-s as `__m512`. After this, we basically declare the `serialize()` methods of the various instructions which is where we generate the actual intrinsics. We show the example for `LoadFVec` and an FMA (fused multiply-add) in Figure 9.19 but, of course, the entire source is available in the Code package. We note that we have implemented the use of 16-bit precision in this work and the `LoadFVec` shows how

```
string LoadFVec::serialize() const
{
    std::ostringstream buf;
    string upConv = "_MM_UPCONV_PS_NONE";
    string lmask = mask;

    if(mask.empty()) {
        lmask = fullMask;
    }

    if(a->isHalfType()) {
        upConv = "_MM_UPCONV_PS_FLOAT16";
    }

    buf << v.getName() << " = _mm512_mask_extload_ps("
        << v.getName() << ", " << lmask << ", "  << a->serialize()
        << ", " << upConv << ", _MM_BROADCAST32_NONE, _MM_HINT_NONE);" <<endl
            ;

    return buf.str();
}

string FMAdd::serialize() const
{
    if(mask.empty()) {
        return  ret.getName()+" = _mm512_fmadd_ps("+a.getName()+", "+b.
            getName()+", "+c.getName()+");" ;
    }
    else {
        return  ret.getName()+" = _mm512_mask_mov_ps(" + ret.getName() + ", "
             + mask + ", _mm512_fmadd_ps(" +a.getName()+", "+b.getName()+", "
            "+c.getName()+"));" ;
    }
}
```

FIGURE 9.19

The serialize methods for `LoadFVec` and `FMadd` instructions from `qphix-codegen`. They can be found in the file `inst_sp_vec16.cc`.

a vector of `unsigned short`-s can be loaded using *up-conversion* in the `_mm512_mask_extload_ps` intrinsic.

GENERATING DSLASH

Generating Dslash is coded in the `dslash.cc` and `dslash_common.cc` files. The basic Dslash body is reproduced in code Figure 9.20, although we have removed some of the branches relating to having to emulate masking for clarity. Modulo some code for dealing with additional scale factors to multiply into the results on accumulation (`beta_vec`), and adding conditionals and masks for accumulating (to do with boundary processing in a multinode implementation), the structure should be very similar to the pseudocode in Figure 9.2.

To round out this story, we show a brief snippet of the code to multiply $SU(3)$ matrices with three-vectors in Figure 9.21, from the file `dslash_common.cc`. First, we declare a global set of `FVec`-s, for each spin, color and complex component that we will use. In our case, `b` refers to the result of the projection and `ub` refers to the result of the multiplication. It is convenient to collect these `FVec`-s into arrays with indices mirroring our desired index structure, as shown in the code for the half-spinor

```
void dslash_body(InstVector& ivector, bool compress12, proj_ops *ops,
    recons_ops *rec_ops_bw, recons_ops *rec_ops_fw, FVec outspinor[4][3][2])
{
    for(int dim = 0; dim < 4; dim++) {
        for(int dir = 0; dir < 2; dir++) {
            int d = dim * 2 + dir;
            stringstream d_str;
            d_str << d;
            string mask;
            bool adjMul;
            recons_ops rec_op;

            if(dir == 0) {
                adjMul = true;
                rec_op = rec_ops_bw[dim];
            }
            else {
                adjMul = false;
                rec_op = rec_ops_fw[dim];
            }
            ifStatement(ivector,"accumulate[" + d_str.str() + "]");
            {
                declareFVecFromFVec(ivector, beta_vec);
                loadBroadcastScalar(ivector, beta_vec, beta_names[d],
                    SpinorType);
                if(requireAllOneCheck[dim]) {
                    mask = "accMask";
                    declareMask(ivector, mask);
                    intToMask(ivector, mask, "accumulate[" + d_str.str() + "]
                        ");
                }

                for(int s = 0; s < 2; s++) {
                    project(ivector, basenames[d], offsnames[d], ops[d],
                        false, mask, d, s);

                    if(s==0) {
                        loadGaugeDir(ivector, d, compress12);
                    }

                    matMultVec(ivector, adjMul, s);
                    recons_add(ivector, rec_op, outspinor, mask, s);
                }
            }
            endScope(ivector);
        }
    }
}
```

FIGURE 9.20

The basic code to generate the body of the Dslash (excluding additional code to deal with software emulated masking on non-Intel Xeon Phi coprocessor platforms.

`b_spinor`. Finally, we generate the matrix multiply in the function `matMultVec` using convenience functions defined earlier for complex arithmetic. Note that there is a loop in the function. When the function is executed that loop will essentially create all the operations in the matrix multiply into a single contiguous instruction stream.

```
FVec b_S0_C0_RE("b_S0_C0_RE");
FVec b_S0_C0_IM("b_S0_C0_IM");
...

FVec b_spinor[2][3][2] = {
    { {b_S0_C0_RE, b_S0_C0_IM}, {b_S0_C1_RE, b_S0_C1_IM}, {b_S0_C2_RE,
        b_S0_C2_IM} },
    { {b_S1_C0_RE, b_S1_C0_IM}, {b_S1_C1_RE, b_S1_C1_IM}, {b_S1_C2_RE,
        b_S1_C2_IM} }
};
...

// r is an array of length 2, as are s1 and s2 -- for complex numbers
void mulCVec(InstVector& ivector, FVec *r, FVec *s1, FVec *s2, string &mask)
{
    mulFVec(ivector, r[RE], s1[RE], s2[RE], mask);
    fnmaddFVec(ivector, r[RE], s1[IM], s2[IM], r[RE], mask);
    mulFVec(ivector, r[IM], s1[RE], s2[IM], mask);
    fmaddFVec(ivector, r[IM], s1[IM], s2[RE], r[IM], mask);
}
...
// deal with spin component 's'
void matMultVec(InstVector& ivector, bool adjMul, int s)
{
    string mask;

    for(int c1 = 0; c1 < 3; c1++) {
        if(!adjMul) {
            mulCVec(ivector, ub_spinor[s][c1], u_gauge[0][c1], b_spinor[s
                ][0], mask);
            fmaddCVec(ivector, ub_spinor[s][c1], u_gauge[1][c1], b_spinor[s
                ][1], ub_spinor[s][c1], mask);
            fmaddCVec(ivector, ub_spinor[s][c1], u_gauge[2][c1], b_spinor[s
                ][2], ub_spinor[s][c1], mask);
        }
        else {
            mulConjCVec(ivector, ub_spinor[s][c1], u_gauge[c1][0], b_spinor[s
                ][0], mask);
            fmaddConjCVec(ivector, ub_spinor[s][c1], u_gauge[c1][1], b_spinor
                [s][1], ub_spinor[s][c1], mask);
            fmaddConjCVec(ivector, ub_spinor[s][c1], u_gauge[c1][2], b_spinor
                [s][2], ub_spinor[s][c1], mask);
        }
    }
}
```

FIGURE 9.21

Code to generate *SU*(3) matrix-vector multiply, from the `dslash_common.cc` file.

PREFETCHING

We implement prefetching in the code generator in two different ways, for L1 and L2 prefetching, respectively. We define custom L1 prefetch functions in the file `data_types.h` which we call explicitly just before starting to work with the data. We show the idea, for example, in the projection operation in Figure 9.22.

Our L2 prefetching is slightly different. We need to fetch further ahead than for L1, and our approach has been for every thread to prefetch the forward T neighbor of the next output block that it will work. While blocks that are co-planar in Y and Z should already be in L2, they may reside in

```
// File: dslash_common.cc
void project(InstVector& ivector, string base, string offset, proj_ops& ops,
     bool isFace, string mask, int dir)
{
    string tmask("");
    PrefetchL1FullSpinorDirIn(ivector, base, offset, dir);

    for(int s = 0; s < 2; s++) {
        for(int c = 0; c < 3; c++) {
            LoadSpinorElement(ivector, psi[0][RE], base, offset, ops.s[s][0],
                 c, RE, isFace,  mask, dir);
            LoadSpinorElement(ivector, psi[0][IM], base, offset, ops.s[s][0],
                 c, IM, isFace,  mask, dir);
            LoadSpinorElement(ivector, psi[1][RE], base, offset, ops.s[s][1],
                 c, RE, isFace,  mask, dir);
            LoadSpinorElement(ivector, psi[1][IM], base, offset, ops.s[s][1],
                 c, IM, isFace,  mask, dir);

            ops.CVecFunc[s](ivector, b_spinor[s][c], psi[0], psi[1], /*mask*/
                 tmask); // Not using mask here
        }
    }
}
```

FIGURE 9.22

Generating L1 prefetches for the projection operation. We explicitly prefetch the whole spinor into L1, before element wise loading.

the caches of other cores than the one which needs them. Therefore, we allow the possibility of prefetching up to four neighboring spinors. For each of these we supply a pointer and an offset. The base pointers are `xyBase` (current tile), `pfBase2`, `pfBase3`, and `pfBase4`. The prefetch offsets to these pointers are `siprefdist1`, `siprefdist2`, and so on up to `siprefdist4`. We also prefetch gauge fields with offset `gprefdist` from the current `gBase` and can prefetch the next output spinor with base `outBase`. Some of these are commented out or not used in the implementation, but that is the basic idea.

The base pointers and `siprefdist` offsets are calculated during the lattice traversal loop in functions like `DyzPlus` in the QPhiX library. In Figure 9.18, we have hidden these computations for clarity, but they can certainly be found in the attached code bundle.

When the code generation is run, the L2 prefetches are generated into a *separate instruction vector* from the mainline code, and are merged with the main Dslash code just before serializing to try and space them evenly through the generated code. This merging is done by the function `mergeIvectorWithL2Prefetches` in the source file `dslash.cc`

GENERATING THE CODE

Now that we have covered the mechanics, let us generate the code. To do this we need to look at two makefiles. One is the main `Makefile`. Here our main work is to choose the architecture by editing the variable `mode` in the first line. Right now, it is set to `mic` which is fine for Intel Xeon Phi coprocessors. We can also set it to, for example, `avx` for AVX code, or `scalar` for nonvectorized code.

Second, for each supported `mode` there is a file called `customMake.xxx`, where `xxx` is the mode. We can look, for example, at `customMake.mic`. In this file, we can set a variety of options for code generation, for example, whether to use streaming stores, or whether we should use L1, L2 prefetching, etc. We show a snippet of `customMake.mic` in Figure 9.23 where we enabled all the prefetching options and

```
#Prefetching options
# FOR MIC SET THESE ALL TO 1
#
PREF_L1_SPINOR_IN = 1
PREF_L2_SPINOR_IN = 1
PREF_L1_SPINOR_OUT = 1
PREF_L2_SPINOR_OUT = 1
PREF_L1_GAUGE = 1
PREF_L2_GAUGE = 1
PREF_L1_CLOVER = 1
PREF_L2_CLOVER = 1

# Gather / Scatter options
USE_LDUNPK = 1            # Use loadunpack instead of gather
USE_PKST = 1              # Use packstore instead of scatter
USE_SHUFFLES = 0          # Use loads & Shuffles to transpose spinor when
    SOALEN>4
NO_GPREF_L1 = 1           # Generate bunch of normal prefetches instead of
    one gather prefetch for L1
NO_GPREF_L2 = 1           # Generate bunch of normal prefetches instead of
    one gather prefetch for L2

# Enable nontemporal streaming stores
ENABLE_STREAMING_STORES ?= 1
USE_PACKED_GAUGES ?= 1     # Use 2D xy packing for Gauges
```

FIGURE 9.23

Code generation options for Intel Xeon Phi coprocessor, from `customMake.mic`. We set every prefetching option, as well as asking for the use of *load-unpack*, *pack-store* and the use of streaming stores.

asked to use *load unpack* and *pack-store* operators for gather-scatter like operations. As a result, we do not want to generate *gather prefetches* and we have disabled those.

Once all these options are set, one can execute the makefile by running `make xxx` where, again, `xxx` is the value used for mode. The code generator should then build and run, with the resulting code being placed in the `xxx` subdirectory as a bunch of files. We note that these files are just the bodies of functions (not even containing function prototypes, etc.). In the QPhiX packages, these files get placed in the directory `include/qphix/xxx/generated` where as usual `xxx` is the value of the `mode`. There are files in `include/qphix/xxx` that define the function headers and include these files appropriately using a mixture of templates, and C-preprocessor macros. In the case of the `mic` target, these files are in `include/qphix/mic` and are called `dslash_mic_complete_specializations_form.h` which defines either the base templates for functions like `dslash_plus_vec`, or if certain preprocessor macros are defined, it defines a specialization, which includes the appropriate generated file.

This inclusion, is driven by the second file: `dslash_mic_complete_specialization.h` which cycles through the various possibilities of the macros, defines them, includes the `_form.h` file and then undefines the macros. This approach is rather messy and we wish a better one was available.

PERFORMANCE RESULTS FOR QPhiX

After chewing our way through the structure of QPhiX and the code-generator, it is finally time to see how well all this hard work has paid off. We will look at the following cases of generated code:

- *scalar*: this uses code generated for the `scalar` mode of the code-generator. It uses all of the 3.5D looping, but no prefetch or vectorization. A version of QPhiX with this set of generated files is available in the Code package in directory `Scalar`.
- *vector*: this uses code generated for the `mic` target but apart from generating vector code, all prefetching options have been turned off. The code for this in the package is in `Vector`. We use explicitly a vector length of 16, and an SOA length of 16.
- *vector + L2*: this is the vector code with L2 prefetches. Code is in `VectorPrefL2`.
- *vector + L1*: this is the vector code with L1 prefetches but not L2. Code is in `VectorPrefL1`.
- *vector + L1 + L2*: this is the vector code with both L1 and L2 prefetching, code is in `VectorPrefL1L2`.
- *vector + L1 + L2 + Barrier*: this is the previous code, but every now and again we resynchronize the cores using a lightweight barrier—similar in style to the Plesiochronous Phasing Barriers discussed in Chapter 5 of the first volume of High Performance Parallelism Pearls. The code for this is in `VectorPrefL1L2Barrier`.
- *vector + L1 + L2 + Barrier, SOALEN=8*: this is the previous code, but using an SOALEN=8, instead of 16. In other words, this test will work with an 8×2 tile. The code for this is in `VectorPrefL1L2BarreirS8`.

We ran all these tests on an Intel Xeon Phi coprocessor model 5110P, booted with the *icache_snoop_off* feature enabled to gain some extra memory bandwidth. We used 59 out of the 60 cores, and an affinity setting of `KMP_AFFINITY=compact,granularity=thread`. We added one unit of padding to our data structures in the *XY* plane, to be free of associativity misses. We present the results in the table shown in Figure 9.24. A couple of things to notice: just moving to the QPhiX lattice traversals, but keeping the code unvectorized, did have some benefit, but not very much. The performance only jumped from 35 GFLOPS (Original) to 51 GFLOPS. However, vectorization even without prefetching had a large impact. The performance reached 184 GFLOPS when using compression.

Looking at the difference between the vector + L1 case and the vector + L2 case, it is clear that L1 prefetching without L2 prefetching is not very beneficial, since probably most L1 prefetches will not hit

Mode	Performance No Compression (GFLOPS)	Performance 2-Row Compression (GFLOPS)
Original	35	N/A
Scalar	51	43
Vector	172	184
Vector+L1	144	169
Vector+L2	176	212
Vector+L1+L2	191	239
Vector +L1+L2+barrier	199	250
Vector +L1+L2+barrier, SOA=8	234	286

FIGURE 9.24

Performance tests run on a lattice with $32^3 \times 64$ sites, using 59 cores (236 threads) on an Intel Xeon Phi coprocessor model 5110P with *icache_snoop_off* feature enabled.

in L2. In contrast, L2 prefetching was very effective. Combining L1 and L2 prefetching we reached 239 GFLOPS with compression.

Finally we note that occasional synchronization with lightweight barriers is beneficial. Let us consider this for a moment, since it is often said that we should avoid synchronizations if we can, since typically they are quite costly. In this situation, we use lightweight barriers contributed by Intel corporation, and the barriers are not global. In any given *phase* of the lattice traversal barriers are called only among a group of cores that work on blocks of the same time-length chunk. This information is set up when we set up the block-phase information (see Figure 9.15)—a `group_tid` is computed for each thread in each block. At construction each `group_tid` is assigned its own barrier. As the threads step through the time-coordinates of a given block, when it comes time to synchronize, threads with the same `group_tid` call a wait on their barrier as can be seen at the end of Figure 9.18. Even within this chunk, we can tune the frequency of the barriers. In our case, we found that we got the highest performance when we execute them once every 16 *t*-slices. The barriers help, because slight drifts in synchronization may mean that cores which are supposed to be working on the same *T*-slice may end up working on different slices. In this situation, it could be that a core which could otherwise access a piece of data from the cache of another core, instead has to go to memory. The cost of this can outweigh the cost of the occasional lightweight barrier. This is similar in spirit to the use of plesiochronous phasing barriers described by Jim Dempsey in Chapter 5 of the first volume of High Performance Parallelism Pearls.

We finish off this discussion by noting that we got the best performance by switching to using a layout with *SOALEN=8*, rather than 16. This seems to hold up generally. We find it surprising, since loading two chunks of 8 instead of a single chunk of 16 seem like it ought to need higher bookkeeping overhead, and possibly more instructions.

We have come a long way from the performances seen in the original code. When we examine the memory bandwidth utilization of the best performing configuration with VTune, the memory bandwidth utilized is now reported as 130 GB/s. For this bandwidth, our model predicts a performance of 298 GFLOPS. Our 286 GFLOPS is 96% of this prediction.

OTHER BENEFITS

We may recall that our original code performed at the level of around 34.7 GFLOPS on a particular dual socket Intel Xeon processor. One of the benefits of all our hard work with blocking and the code generator is the improved performance of the new code on Intel Xeon processors. In the Code package, we have a setup to build QPhiX with AVX vectorization in the `FinalAVX` directory. This build does not perform barriers, and because of the excellent hardware prefetching on Intel Xeon processor, we have disabled software prefetching too. Finally, since this is a dual socket cc-NUMA system, we run the code setting the minimum number of time-splits C_t to be 2, which is the number of sockets. With the `KMP_AFFINITY` set to `compact` and because array initialization proceeds in lattice traversal order, the code has become NUMA-aware. Thanks to the large-shared L3 cache of the Intel Xeon processor we can choose slightly larger blocks. A choice of $(B_y,B_z) = (8,8)$ allows for 16 blocks in the *Y*-*Z* plane, which after the further split by the minimum value of C_t becomes 32, with 16 blocks assigned to each 8-core socket. This workload can then be processed in two phases without imbalance. We keep the

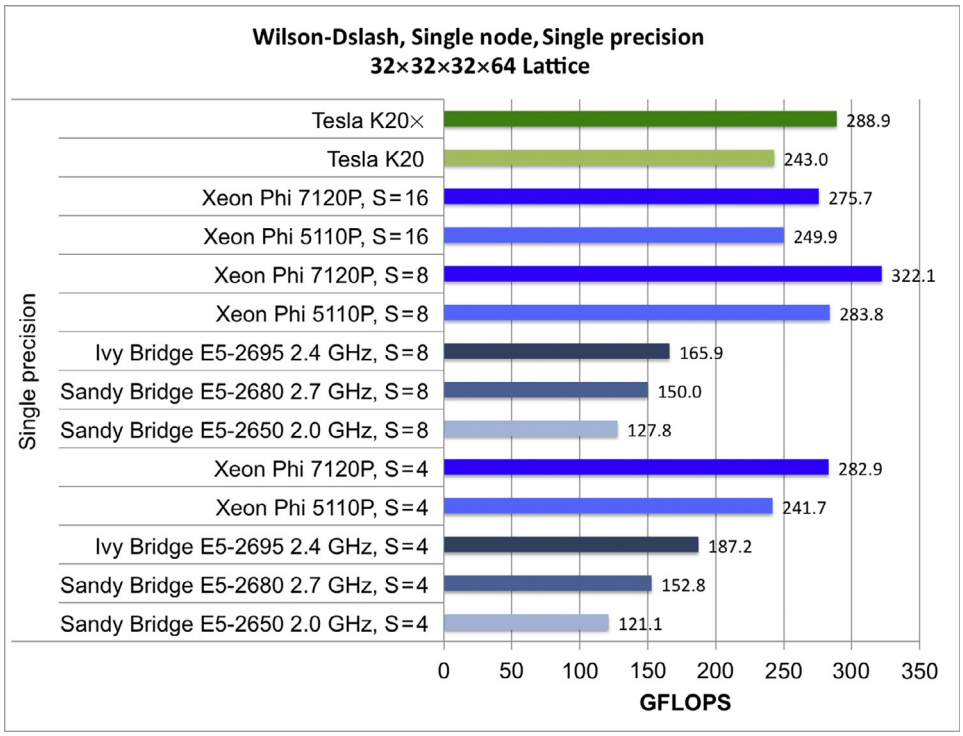

FIGURE 9.25

Performance of single node, singe precision Wilson-Dslash with two-row compression on a variety of systems. The GPU performance numbers were generated using the QUDA library.

same padding parameters, but set the SMT thread geometry to 1×2 in the *Y-Z* plane, since we have only two hyperthreads. On our system, the Wilson-Dslash timing test ran at **127 GFLOPS**, *a speed-up of about* ***3.6***$\times$ *over the original code without SSE optimizations*

To be completely fair to the original code, we have also built it with SSE optimizations enabled, and, as before ran it with the `numactl-interleave=0,1` command to ameliorate its non-NUMA nature. We find that the best performance on our test case in single precision is 48-49 GFLOPS. So QPhiX also gains over this configuration by a factor of 2.6×, which is a very significant gain. Further, we should note that it is possible that the processor may have entered its *turbo-boost* mode during our tests, we have not investigated this aspect in significant detail.

We shown in Figure 9.25 the performance of on a variety of systems including a comparison to a Wilson-Dslash code running on NVIDIA K20, and K20X GPUs. The GPU performances were measured using the QUDA library running with comparable options (single precision, two-row compression) and the same volume. We show also performance on the Intel Xeon Phi coprocessor model 7120P, and consider varying the SOA-length. In this plot, the final result for Intel Xeon Phi coprocessor

model 5110P with SOA-length 8 is 283.8 GFLOPs, which is two to three GFLOPS less than the numbers we have measured earlier. This we take to be simply a fluctuation which at this rate is at about the 1% level. Finally we note that in the same way that Intel Xeon Phi coprocessor performed best with an SOA-length of 8 (half its vector length), in a similar way, the Xeon results are best with an SOA-length of 4 (half of its vector length).

We do make the disclaimer that to get this level of performance on the Intel Xeon Phi coprocessor requires the *icache_snoop_off* feature to be enabled at boot time. Without this feature performances on the Xeon Phi can be lowered compared to our numbers. As an example on a 61 core Xeon Phi in the Intel Endeavor cluster (*icache_snoop_off* enabled) we sustained 320 GFLOPS for a particular benchmark. The same problem size running on Stampede (similar 61 core Xeon Phi part) without the *icache_snoop_off* feature ran at 271 GFLOPS.

THE END OF THE ROAD?

Of course not! However, we are approaching the end of the chapter. We have gone through a lot of work to optimize the single node performance of Wilson-Dslash and the effort has been very fruitful. However, there are many avenues left, which we have not had the space to explore in this chapter.

One very important aspect is scaling Wilson-Dslash onto multiple nodes, either of Intel Xeon Phi coprocessors or Intel Xeon processors. The code to do this is present in *QPhiX* as well as the original code (although that uses a slightly different algorithm). In its current setting as a PCIe card, several challenges have been encountered with multinode scaling which could merit their own chapter. The interested reader should refer to the "For more information" section at the end of this chapter.

Another important consideration is that the Wilson-Dslash operator is typically the computationally most expensive piece of larger composite operators. Typically one needs to solve linear equations with the larger composite operator. Recently there have been advances in how to solve these systems, not just by the usual Krylov subspace methods such Conjugate Gradients or Stabilized Bi-Conjugate Gradients but also by newer algorithms such as Generalized Conjugate Residuals or Flexible GMRES, which allow the use of Domain Decomposed preconditioners and deflation, and also by Algebraic Multi-Grid approaches. In particular, the domain decomposed preconditioners have very nice properties: small domains can be blocked into L2 cache on the Intel Xeon Phi coprocessor, essentially moving the bandwidth bottleneck from main memory bandwidth to cache bandwidth. A thorough exposition of optimizing a solver with a domain decomposed preconditioned for Intel Xeon Phi coprocessor can be found in our Supercomputing 2014 paper (see the "For more information" section).

When one needs to solve for several systems, one can also consider vectorizing over the systems, which can lead to simpler vectorization than considered in QPhiX. This was done for the different way of formulating QCD than presented here in the "HISQ inverter" paper (see the "For more information" section), and work on Wilson-Dslash is in progress with promising preliminary results.

What would we like to emphasize as the take home messages from this chapter? Perhaps the following:

- To exploit the system for maximum efficiency, it helps to have some model of the performance. Is it memory bound? Is it compute bound? Can expected performance be related to system parameters?

In our case, we made several refinements to our model, for example, to take account of streaming stores, cache reuse, or compression.
- In our case, several optimization steps had to be employed. Just threading for parallelism was not sufficient to get the best performance. Vectorization, prefetching, blocking, and some synchronization were also needed.
- Be prepared for surprises. In our case, these came from the improvement from barriers, and from the finding that an SOA-length that is half of the vector length somehow was more performant than working with an SOA-length equal to the vector length.
- Domain-specific code generators can be helpful. They need not be super sophisticated, and can lower the pain of exploring certain optimizations (e.g., strategies to load, prefetch, etc.). In some cases, like ours, they can also help with performance on other architectures.

FOR MORE INFORMATION

- Babich, R., Brannick, J., Brower, R.C., Clark, M.A., Manteuffel, T.A., et al., 2010. Adaptive multigrid algorithm for the lattice Wilson-Dirac operator. Phys. Rev. Lett. 105, 201602.
- Babich, R., Clark, M.A., Joó, B., Shi, G., Brower, R.C., Gottlieb, S., 2011. Scaling lattice QCD beyond 100 GPUS. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, New York, NY, USA. ACM, pp. 70:1-70:11.
- Boyle, P.A., 2009. The {BAGEL} assembler generation library. Comput. Phys. Commun. 180 (12), 2739-2748. 40 {YEARS} {OF} CPC: a celebratory issue focused on quality software for high performance, grid and novel computing architectures.
- Clark, M.A., Babich, R., Barros, K., Brower, R.C., Rebbi, C., 2010. Solving lattice QCD systems of equations using mixed precision solvers on GPUs. Comput. Phys. Commun. 181, 1517-1528.
- Edwards, R.G., Joó, B., 2005. The Chroma software system for lattice QCD. Nucl. Phys. Proc. Suppl. 140, 832.
- Frommer, A., Kahl, K., Krieg, S., Leder, B., Rottmann, M., 2014. Adaptive aggregation based domain decomposition multigrid for the lattice Wilson-Dirac operator. SIAM J. Sci. Comput. 36, A1581-A1608.
- Frommer, A., Nobile, A., Zingler, P., 2012. Deflation and flexible SAP-preconditioning of GMRES in lattice QCD simulation. ArXiv e-prints, April.
- Heybrock, S., Joó, B., Kalamkar, D.D., Smelyanskiy, M., Vaidyanathan, K., Wettig, T., Dubey, P., 2014. Lattice QCD with domain decomposition on Intel® Xeon Phi™ co-processors. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '14, Piscataway, NJ, USA. IEEE Press, pp. 69-80.
- Hestenes, M.R., Stiefel, E., 1952. Methods of conjugate gradients for solving linear systems. J. Res. Natl Bur. Stand. 49 (6), 409-436.
- Jeffers, J., Reinders, J., 2013. Intel® Xeon Phi™ Coprocessor High-Performance Programming, first ed. Morgan-Kauffman, Elsevier, Waltham, MA, USA.
- Jeffers, J., Reinders, J., 2014. High Performance Parallelism Pearls, first ed. Morgan-Kauffman, Elsevier, Waltham, MA, USA.

- Joó, B., Kalamkar, D.D., Vaidyanathan, K., Smelyanskiy, M., Pamnany, K., Lee, V.W., Dubey, P., Watson, W., 2013. Lattice QCD on Intel Xeon Phi coprocessors. In: Kunkel, J.M., Ludwig, T., Meuer, H.W. (Eds.), Supercomputing, volume 7905 of Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, pp. 40-54.
- O. Kaczmarek, Schmidt, C., Steinbrecher, P., Mukherjee, S., Wagner, M., 2014. HISQ inverter on Intel Xeon Phi and NVIDIA GPUs, CoRR abs/1409.1510, http://arxiv.org/abs/1409.1510.
- Luscher, M., 2007. Local coherence and deflation of the low quark modes in lattice QCD. JHEP 0707, 081.
- McClendon, C., 2001. Optimized Lattice QCD Kernels for a Pentium 4 Cluster. Technical Report JLAB-THY-01-29, Jefferson Lab, September.
- Nguyen, A., Satish, N., Chhugani, J., Kim, C., Dubey, P., 2010. 3.5D blocking optimization for stencil computations on modern CPUS and GPUS. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10, Washington, DC, USA. IEEE Computer Society, pp. 1-13.
- Pochinksy, A.V. QA0 Code Generator. http://www.mit.edu/∼avp/qa0.
- Jefferson Lab GitHub Projects. CPP Wilson-Dslash. https://github.com/jeffersonlab/cpp_wilson_dslash.git.
- Jefferson Lab GitHub Projects. QPhiX-Codegen Code Generator. https://github.com/jeffersonlab/qphix-codegen.git.
- Jefferson Lab GitHub Projects. QPhiX Library. https://github.com/jeffersonlab/qphix.git.
- Smelyanskiy, M., Vaidyanathan, K., Choi, J., Joo, B., Chhugani, J., Clark, M.A., Dubey, P., 2011. High-performance lattice QCD for multicore based parallel systems using a cache-friendly hybrid threaded-MPI approach. In: 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC), November, pp. 1-10.
- van der Vorst, H.A., 1992. Bi-CGSTAB: a fast and smoothly converging variant of bi-CG for the solution of nonsymmetric linear systems. SIAM J. Sci. Stat. Comput. 13 (2), 631-644.
- Vaidyanathan, K., Pamnany, K., Kalamkar, D.D., Heinecke, A., Smelyanskiy, M., Park, J., Kim, D., Shet, G.A., Kaul, B., Joo, B., Dubey, P., 2014. Improving communication performance and scalability of native applications on Intel Xeon Phi coprocessor clusters. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, May, pp. 1083-1092.
- Download the code from this, and other chapters, http://lotsofcores.com.