CHAPTER 11

Algorithm and Data Structures for Xeon Phi

Algorithms and data structures appropriate for Xeon Phi are active fields of research and deserve a book on their own. This chapter will touch only on some common algorithm and data structure optimization techniques that I have found useful for common technical computing applications. These algorithms will definitely evolve as we gain more experience with the hardware. This chapter does not derive the algorithms but rather focuses on optimization techniques to achieve good performance on Xeon Phi. For example, I assume familiarity with Monte Carlo simulation techniques and the algorithms used in financial applications and instead focus on those components of the algorithms that be optimized to make the most effective use of Xeon Phi architecture capabilities.

This chapter looks at how the Xeon Phi architecture affects the design of the algorithm and data structures that are targeted for this coprocessor. Although existing algorithms optimized for Intel Xeon processors will work on Xeon Phi, not all algorithms will scale by reason of the following differences in the cores:

- Xeon Phi has a much higher number of threads (240) compared with the 24 threads supported on the typical Intel Xeon processor core. Moreover, the single-threaded performance of Xeon Phi is much slower than that of the Xeon processor. As a result of these two characteristics of the Xeon Phi coprocessor, the part of an algorithm that depends on single-threaded performance will be adversely affected and need modification. The low serial performance causes thread synchronization primitives to take on a higher percentage of the overall runtime and may require a fundamental change in the algorithm to reduce the parallel overhead.
- Xeon Phi has a larger vector length than Xeon: 512 bits vs. 256 bits. Loops that benefit from a 256-bit vector length may not perform well on 512 bits due to insufficient loop counts.
- The total flops achievable on the 60-core Xeon Phi coprocessor is 2 petaflops (PF) single precision vs. ~1 teraflops (TF) on a two-socket Xeon E5-2670 processor running at 2.6 GHz. This is achieved by using all the cores and all the vector operations efficiently. On the other hand, the practical memory bandwidth per core on Xeon Phi is less than that on Xeon. The lower flops/byte supported by the Xeon Phi coprocessor compared to the Xeon processor may require different algorithms to solve the same problem.
- The amount of memory per core, ~266 MB per Xeon Phi core vs. 4 GB per core on a 24-core Xeon two-socket system with 96 GB memory, will in many cases require changes in the algorithm to accommodate core scheduling changes to handle a large number of tasks with a small work size per task.

These data points show that the only way to achieve better performance on Xeon Phi is to have the algorithms exploit the massive flops available on the Xeon Phi and mostly running from cache. The Xeon Phi coprocessor will also need to be paired with the Xeon processor capabilities of large memory and high single-threaded performance

to achieve this goal. Since the vector sizes are wide, the data to be vectorized will need to be wide enough to make use of the vector capabilities. Key to achieving high performance on Xeon Phi architecture is implementing an efficient algorithm paired with an appropriate workload size.

Algorithm and Data Structure Design Rules for Xeon Phi

To exploit the power performance efficiency of Xeon Phi processor, an algorithm should conform to the following rules:

- Rule 1. *Scalable parallelization*: The code should be able to scale to all the cores and preferably all the threads. Not only will you need to vectorize at the thread level, but you must also scale the algorithm to a large number of threads.
- Rule 2. *Efficient vectorization*: The code should be able to efficiently use 512-bit vectors on Xeon Phi. This requires that the input dataset be large enough to provide enough work for the coprocessor such that:
  - The input array size is long enough.
  - The working dataset residing on memory is properly aligned for optimal vector load and store.
  - The working dataset accessed is coalesced so that, when filling the cache line, the consecutive bytes come from consecutive regions rather than needing to be gathered from multiple data locations.
- Rule 3. *Optimal cache reuse*: Maximum reuse of the cache should overcome the memory bandwidth limitation and increase sustainable flops/byte.

Rules 1 and 2 require that the problem size match the machine size, which is defined as the (vector length × number of cores). For a 60-core Xeon Phi, the machine size is (64 bytes × 60 cores) = 3.8 kB. To fully utilize the hardware, the algorithm will need at least 3.8 kB of data to work on.

Each algorithm should go through the following high-level steps to perform optimally on the Xeon Phi coprocessor:

1. Divide the data equally among the cores and threads within a core.
2. Divide the data so that each portion of working dataset can fit in the L2 cache of each core.
3. Work on the data using vector primitives.
4. Merge the results back using vector and optimal reduction algorithms.

Each data structure should possess the following characteristics for optimal cache reuse and minimal pressure on memory interconnects:

1. If needed and possible, convert the array of structures to an aligned structure of arrays for single-stride access for all threads running on the coprocessor.
2. Block the data structure. The data should be blockable so that they can fit in the cache and be worked on.

Let's look at the logical thinking behind some common algorithms to see how they can be implemented to run efficiently on Xeon Phi architecture.

General Matrix-Matrix Multiply Algorithm (GEMM)

General matrix-matrix multiplication (GEMM) is one of the most commonly used algorithms in technical computing applications and is available in the basic linear algebra library (BLAS) in the form of the routines SGEMM or DGEMM for single-precision and double-precision dense matrix-matrix multiply operations. xGEMM, where x can be 's' for single precision and 'd' for double precision, computes the new value of matrix C based on the matrix product of matrices A and B and the old value of matrix C, formulated as:

```
C = alpha*op(A) * op(B) + beta*C
```

where `alpha` and `beta` are scalars and `op` indicates a possible transpose of the matrices.

The general operation is a triple-nested loop:

```
for(i=0; i<M;i++)
  for(j=0;j<N;j++)
    for(l=0;l<k;l++)
         C(i,j) = C(i,j) + A(i,l)*B(i,j)
```

The problem with this triple-nested loop algorithm is that it:

- Will not vectorize by the compiler because the compiler is unable to resolve aliasing among arrays A, B, and C.
- Has in general a very large memory footprint to fit in the L2 cache of each core.
- Demands a lot of memory bandwidth, requiring three memory accesses to A, B, and C to perform an FMA operation.

Rules 1 and 3: Scalable Parallelization and Optimal Cache Reuse

In order to optimize this algorithm for Xeon Phi, make use of the blocked dense matrix multiply method. It can be shown that if A, B, and C matrices are divided into blocks with conformable block sizes, the blocked matrix-matrix multiplication can be done like a regular matrix-matrix multiplication.[1] This property enables you to utilize the same triple nested loop algorithm described previously to work on blocks of matrices instead of individual elements and arrive at the same result as multiplying the whole matrices. This method allows you to break up the work among the cores of Xeon Phi and optimize cache reuse.

You will need to break up the dataset so that you can perform multiple GEMM operations in parallel on each available thread on the coprocessor and work out of the cache. For illustration's sake, assume that we are using two cores of the Xeon Phi. Figure 11-1 shows how input matrices for `op()` set to no-transpose, A, B, and C are broken into blocks of a certain size to be distributed between these two cores. The blocks of matrices B and C that are colored black will be worked on by core 0 till finished with all the blocks, and the blocks of matrices B and C that are colored light gray will be worked on by core 1. The dark gray blocks of matrix A will be needed by each core and will be replicated in the respective L2 caches of each processor. The results will be the desired output once all the tasks distributed to all the cores are finished and the destination memory is updated. The block size depends on the L2 cache size. It should be chosen such that the multiple blocks fit in the L2 cache (512 kB) of each core and the most commonly used A and B blocks also fit the L1 cache (32 kB) to provide fastest access.

[1] Howard Eves, *Elementary Matrix Theory*. Dover Books, 1980.

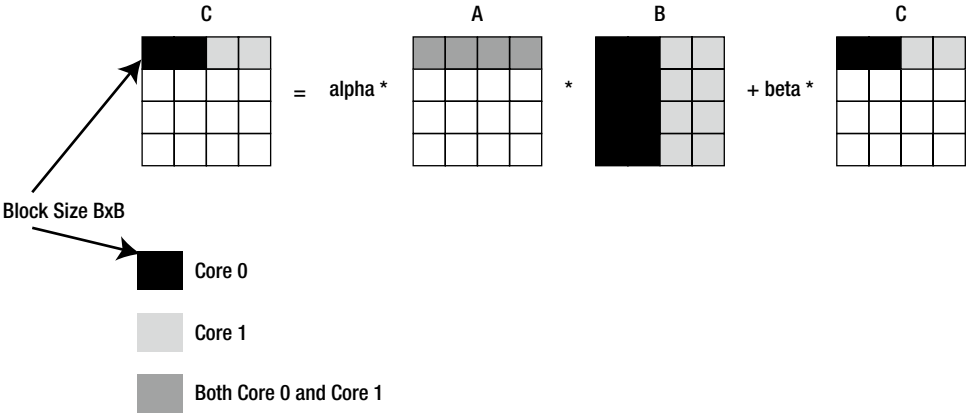

Figure 11-1. *xGEMM parallelization using blocked matrix*

Rule 2: Efficient Vectorization

The blocks submitted to each core are worked on by multiple SMT threads in parallel on each core (Figure 11-2). For sGEMM (single precision), each block is broken into 16x16 subblocks aligned to the cache line boundary. The 16-element width of the subblocks is chosen to match the vector width of the Xeon Phi coprocessor. The subblocks of C can be aligned loaded into 16 vector registers (16x16 register blocking). Each row of B subblock (16x1) is loaded into another vector register by memory-aligned load. (In Xeon Phi, there are 32 vector registers per thread.)

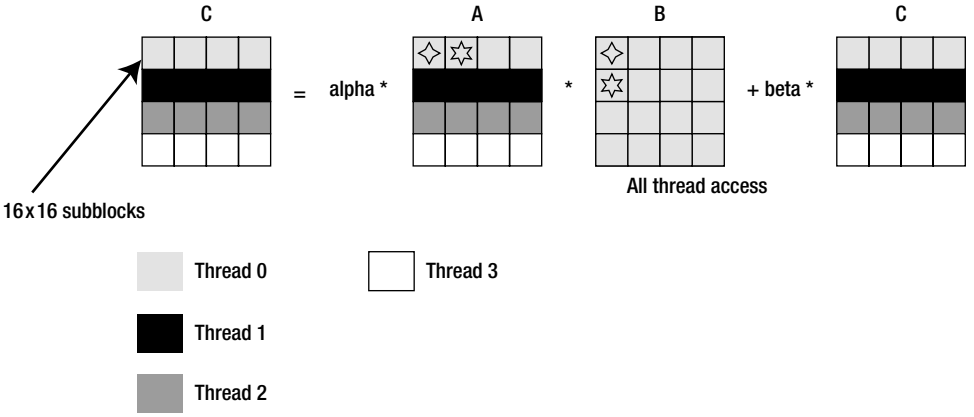

Figure 11-2. *Per core operation on a matrix block. Here A, B, and C represent blocks of the original matrix in Figure 11-1 assigned to a core*

In Figure 11-2, to compute a subblock of C, a column of A coming from the L1 cache can be multiplied by a row of B and accumulated into 16 vector registers holding C. To illustrate the process, think about a 2x2 square matrix A, B, and C. By sketching on a sheet of paper, you will see that the first components of C11, C12, C21, and C22 can be obtained by multiplying A11 × B11, A11 × B12, A21 × B11, and A21 × B12, which is obtained by multiplying a column of A (A11, A21) by a row of B (B11 and B12). Assuming a 2-wide vector register, you load and broadcast A11 in one register and multiply with a register loaded with a row of B to update C registers holding C11 and C12 values (first components) of the matrix multiplication. This can be extended to the 16x16 matrix subblocks discussed here.

Using the cache blocking and software prefetch, you can get the values of the matrix blocks into cache levels L2 and L1 to achieve good efficiency with this algorithm on Xeon Phi.

It can be verified that this algorithm conforms to all three design rules of an optimal algorithm for the Xeon Phi coprocessor. It satisfies Rule 1 by dividing the input matrices into smaller chunks to be able to distribute them equally among the cores and threads within a core. The algorithm satisfies Rule 3 by fitting the data into the L2 and L1 caches and including prefetches to reduce data access latencies. And it satisfies Rule 2 by working on the data unit by vector instructions. At the output generation step, the output is done in parallel by each core and individual threads within a core without contention by working on the disjoint area of array C.

The simple square matrix multiplication example considered in this section is an artificial example. Implementing matrix multiplication in practice requires handling various shapes and boundary conditions for optimal performance. I highly recommend that you make use of off-the-shelf libraries such as Intel MKL for such operations. These libraries implement various BLAS and Sparse BLAS routines and make it easier for users to optimize for Xeon Phi coprocessors.

Molecular Dynamics

Molecular dynamics (MD) are a subset of the broader class of applications known as the *particle dynamics* or *n-body applications.* The goal of these applications is to simulate the interaction between particles using the Newtonian equations of motion caused by force fields defined by the underlying physics. In the MD problem, the goal is to determine the molecular movement over a period of time till the molecules and atoms involved in the simulation get to a steady state. The forces and potential energy involved are defined by molecular mechanics and used to compute the velocities and movement of the atoms and molecules. The MD algorithm is used in many fields besides classical mechanics, including material science, biochemistry, and biophysics.

The basic algorithm of MD simulation is very simple and straightforward. It is a time-stepped simulation where, at each time step, the application calculates the forces between the atoms. The forces may be bonded or non-bonded forces. Using the calculated forces, the atoms' positions are updated. The time step is repeated till either steady state is reached or a certain number of time steps have been performed.

The molecules and atoms are represented by an array of data structures containing the forces, velocities, and positions of each particle. The most time-consuming computation of MD code is computing the nonbonded force field, which requires computing the force exerted on a particle by every other particle in the system. This computation consumes almost 80 percent of execution time for practical workload sizes. A common optimization is to set a cutoff distance for the neighbor list of interacting molecules and to calculate forces only among them. MD code can also benefit from spatial sorting of the neighbor list to help optimize the force calculation. This section looks at the time-consuming force-computation component of MD applications on Xeon Phi. Other computations such as the neighbor-list build and sort may be done on the host processor.

In general, MD code can compute various types of forces between the molecules. Code Listing 11-1 shows the pseudocode for a *Lennard-Jones force calculation* in an MD kernel.[2] The Lennard-Jones potential is widely used in MD code to describe interactions between the molecules or atoms being simulated.

Code Listing 11-1. Lennard-Jones Force Calculation

```
// loop over all the atoms
for (int i = 0; i < numAtoms; i++)
{
    //Get position info for a given atom
    POSITION ipos = position[i];

    neighList = getNeigborList(i);
    numNeighbors = neighList.getCount();
```

[2]A similar implementation of this code can be found in the SHOC benchmark for Xeon Phi. https://github.com/vetter/shoc-mic

```
    // loop over all the neighbors of the given atom
    for(int j = 0;j < numNeighbors;j++)
    {
        int jidx = neighList[j];
        // get position of the nrighbor jidx
        POSITION jpos = position[jidx];

        double delx = ipos.x - jpos.x;
        double dely = ipos.y - jpos.y;
        double delz = ipos.z - jpos.z;

        // compute distance between the neighbors
        double r2inv = delx*delx + dely*dely + delz*delz;

        // if the distance is less than cutoff distance
        if (r2inv < cutsq) {

            // calculate the force
            r2inv = 1.0f/r2inv;
            double r6inv = r2inv * r2inv * r2inv;
            double force = r2inv*r6inv*(lj1*r6inv - lj2);
            // accumulate the force value for this atom
            Force[i].x += delx * force;
            Force[i].y += dely * force;
            Force[i].z += delz * force;
            Force[jidx].x -= delx * force;
            Force[jidx].y -= dely * force;
            Force[jidx].z -= delz * force;

        }
    }
}
```

The goal here is to show how this code could be optimized for the Xeon Phi coprocessor. The code behaves as follows:

1. The code goes over each atom one at a time.
2. The code loops over all the neighbors of the atom.
3. The code gets the positions of the neighboring atoms and calculates the force using the distance between the atoms.
4. If the distance between the atoms is less than the cutoff distance, the code updates the force for this atom and the interacting atoms.

Rule 1: Scalable Parallelization

In order to run this code for Xeon Phi in accordance with the rules for optimal algorithms, I make the modifications shown in Code Listing 11-2. The first rule dictates that the code should be scalable to all the threads. In this case, I use OpenMP parallel and dynamic schedule to achieve this scalability. Since the amount of work for each atom depends on the neighbor size and may not be equal, dynamic scheduling will prevent load imbalance in the overall force computation task on Xeon Phi. This modification is provided in line 1 of Code Listing 11-2. Since in general MD code will have millions of atoms to be simulated per node, this will provide enough work for each thread.

Code Listing 11-2. Xeon Phi Lennard-Jones Force Calculation

```
// loop over all the atoms in parallel
#pragma omp parallel for schedule(dynamic)   ----- line 1
for (int i = 0; i < numAtoms; i++)
{
    POSITION ipos = position[i];
    neighList = get NeigborList(i);
    numNeighbors = neighList.getCount();
    // loop over all the neighbors of the given atom
    for(int j = 0;j < numNeighbors;j++)
    {
        int jidx = neighList[j];
        // prefetch position array to cache level 1 and level 2
        // Note you need to prefetch 16 elements to help with the
        //gather – just shown 2 for clarity
        __mm_prefetch(&position[j+offset], _MM_HINT_T0);  -- line 15 // prefetch to L0 cache
        __mm_prefetch(&position[j+offset],  __MM_HINT_T1;  -- line 16 // prefetch to L1 cache
        // get position of the nrighbor jidx
        POSITION jpos = position[jidx];
        double delx = ipos.x - jpos.x;
        double dely = ipos.y - jpos.y;
        double delz = ipos.z - jpos.z;

        // compute distance between the neighbors
        double r2inv = delx*delx + dely*dely + delz*delz;

        // if the distance is less than cutoff distance
        if (r2inv < cutsq) {
            // calculate the force
            r2inv = 1.0f/r2inv;
            double r6inv = r2inv * r2inv * r2inv;
            double force = r2inv*r6inv*(lj1*r6inv - lj2);
            // accumulate the force value for this atom
            Force[i].x += delx * force;
            Force[i].y += dely * force;
            Force[i].z += delz * force;
            Force[jidx].x -= delx * force;
            Force[jidx].y -= dely * force;
            Force[jidx].z -= delz * force;
        }
    }
}
```

Rules 2 and 3: Efficient Vectorization and Optimal Cache Reuse

Rules 2 and 3 for optimization of algorithms for Xeon Phi are hard to meet for MD problems, which by their very nature entail that the neighbor atoms be distributed randomly in the memory address space. This property constrains that the bytes accessed by the vector code not be unit-stride, which results in inefficient vector gather/scatter operation and the possibility that the various arrays being operated on, such as force arrays, may not be properly aligned. Moreover, the neighbor list needs to be updated from time to time as the atoms may move out or enter the

neighbor list. This is why MD code may not be able to exploit the full computational potential of Xeon Phi architecture unless it is reorganized by data structure to allow efficient vector unit usage of the hardware. You can improve the vectorization efficiency by the following techniques:

1. Since the different data structures such as position and force in the code are accessed using the vector gather/scatter instruction, which results in path length increase and memory access latencies, a big performance gain may be obtained by inserting prefetch as shown in lines 15 and 16. If successfully prefetched into the cache level, the inefficiency due to scatter gather code is vastly reduced, thus providing good performance.
2. Data structures such as positions and the neighbor list can be aligned to cacheline boundaries when allocating the data structures and padded to be multiples of cache line size.
3. Since the gather operation works in a loop where each iteration brings in a cache line, it might be possible to have more neighbor data in a cache line if the neighbor position matches the memory layout of the neighbor list. This can be effected by spatial sorting of the neighbor list so that particle orders in the sorted neighbor list are in the nearby cache-friendly memory region.[3]
4. Divide the nonbonded force compute between the host and Xeon Phi using asynchronous computations.
5. In some cases it may be possible to utilize mixed-precision arithmetic—such as computing force and position in single precision and accumulating in double precision—to achieve higher performance.

Stencil Operation

Stencil operations are commonly used in many technical computing applications for simulating diffusion in fields such as computational fluid dynamics, electromagnetics, and heat propagation. These problems are associated with structured grids and use mathematical finite difference representation of differential operators—such as Laplacian ($u_{t+1} \leftarrow \nabla^2 u_t$) operators, divergence, and gradient—to find answers to the problems formulated. *Stencil* refers to the predetermined set or pattern of nearest neighbors including the element itself. The stencil can be used to compute the value of various elements in an array at a given time-step based on its neighbors' values computed from previous time-steps. The algorithm in general steps in time with some given initial conditions over all the elements to simulate the diffusion or other physical process in time.

Figure 11-3 graphically represents a stencil operation performed on an element in a 3D array for a Laplacian operator. This particular stencil is denoted the *13-point stencil* because its computation involves 13 elements. Equivalent code can be written as shown in Code Listing 11-3. The `fout[z,y,x]` element is the center element shown in Figure 11-3 and has value based on neighboring elements `f[],` as shown in Figure 11-3.

[3]S. Meloni, M. Rosati, and L. Colombo, "Efficient Particle Labeling in Atomistic Simulations," *Journal of Chemical Physics*, Vol. 126, No. 12, 2007.

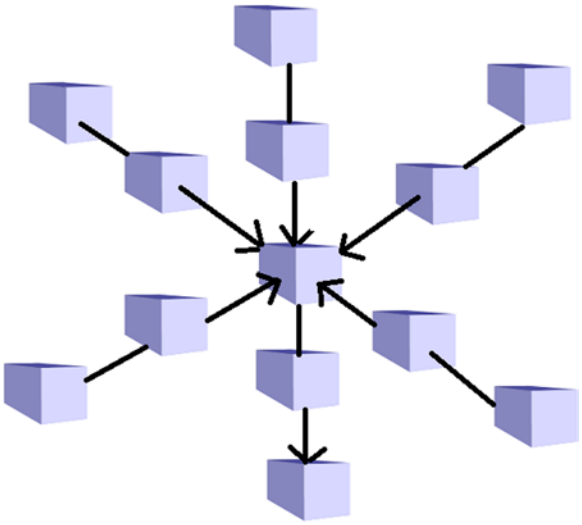

Figure 11-3. *Graphical representation of a 3D stencil operation*

Code Listing 11-3. Stencil Operation Pseudo-code

```
for(t=0; t<numTimeSteps;t++){
   for(z=2; z<NZ-2; z++){
     for(y=2; y<N2; y++){
      for(x=2; x<NX-2; x++){
       fout[z,y,x] = c1*f[z,y,x] +
        c2*f[z,y,x-2] + c3*f[z,y,x-1] + c4*f[z,y,x+1] + c5*f[z,y,x+2] +
        c6*f[z,y-2,x] + c7*f[z,y-1,x] + c8*f[z,y+1,x] + c9*f[z,y+2,x] +
        c10*f[z-2,y,x] + c11*f[z-1,y,x] + c12*f[z+1,y,x] + c13*f[z+2,y,x]
   }}}
  double *tmp = fout; fout = f; f=tmp; //switch buffers
}
```

The constants c1-c11 represent the weighted contribution of the various neighboring elements of the given cell value being computed and defined by the physics of the problem. In this code you are working with one input and one output array and switching between them in every time step. For other operators, such as gradient and divergence, the number of input and output arrays may be multiple and you may need simultaneous access to multiple arrays. In addition, the physics of the problem may impose formulations with more data arrays to be read per grid point computation, increasing the data traffic even more.

The code sweeps through the input 3D array `f[]`, which is usually larger than the L2 cache of the Xeon Phi hardware. This causes low flops/byte for this operation, in turn causing performance to be memory bandwidth-bound. Having more arrays to work on puts more pressure on the memory subsystem of Xeon Phi, causing a further performance drop.

Rule 1: Scalable Parallelization

In the stencil-based application, parallelization happens by naturally breaking up the larger 3D grid into smaller ones per card. The boundary (*halo*) exchange may happen between neighboring cells at every time step to exchange the boundary data. Within a card, because this is a structured grid, a parallelization technique is often used to divide the contiguous blocks allocated to Xeon Phi card equally among the cores so that the threads in a core share one block.

To parallelize the loop, you can do OpenMP-based parallelization, as shown in Code Listing 11-4, on the outer two loops and vectorizing the inner loop. To increase the amount of work for each OpenMP thread, you can use the OpenMP loop collapse construct on the outer loops as well, as shown in Code Listing 11-4. Also make sure to affinitize the code to the Xeon Phi core to make optimal cache reuse using `KMP_AFFINITY`.

Code Listing 11-4. Stencil Calculation Pseudo-code Parallelized with OpenMP

```
for(t=0; t<numTimeSteps;t++){
#pragma omp for collapse(2)
   for(z=2; z<NZ-2; z++){
     for(y=2; y<NY-2; y++){
      for(x=2; x<NX-2; x++){
       fout[z,y,x] = c1*f[z,y,x] +
        c2*f[z,y,x-2] + c3*f[z,y,x-1] + c4*f[z,y,x+1] + c5*f[z,y,x+2] +
        c6*f[z,y-2,x] + c7*f[z,y-1,x] + c8*f[z,y+1,x] + c9*f[z,y+2,x] +
        c10*f[z-2,y,x] + c11*f[z-1,y,x] + c12*f[z+1,y,x] + c13*f[z+2,y,x]
   }}}
  double *tmp = fout; fout = f; f=tmp; //switch buffers
}
```

You can now execute the code in parallel on all the available Xeon Phi cores to attain better performance compared with that of the serial version of the code.

Rule 2: Efficient Vectorization

The stencil methods have a high degree of parallelism but usually are not friendly to vector architectures such as Xeon Phi. To start with, the compiler is unable to vectorize the inner loop in Code Listing 11-4 due to the possible alias between the two c pointers: `fout[]` and `f[]`. In order to vectorize this code, you need to tell the compiler to assume the array pointers point to the disjoint location by using `#pragma vector always` or `#pragma ivdep`. This pragma will vectorize the inner loop, but the vectorization is inefficient due to memory access issue, as you will see shortly. To help vectorize the code more efficiently, you can remove the prologue and epilogue of the vectorized code by aligning the array data to the cacheline boundary and padding the arrays so that the innermost dimension is a multiple of the cacheline size. Aligning the data will also help with cache-way oversubscription in some scenarios. Once the data are aligned, you can use `pragma vector aligned` to tell the compiler to assume aligned vectors. Finally, you don't want to waste BW when writing to the `fout[]` array, which causes output data to be read, to complete the read for ownership operation. This can be done by indicating to the compiler that `fout` is a nontemporal write and thus not to waste valuable BW needed to read the data in.

The compiler-generated code created by the above changes and directives will be inefficient. Inefficient vectorization happens because you are adding shifted versions of data elements such as `f[z,y,x-2] .. f[z,y,x+2]` in the same computing statement. This inefficiency is known as *stream alignment conflict* and refers to the fact that the vectors needed to perform the computation from the same data stream but are not aligned to one another, thus requiring extra data manipulations. In this case, the compiler generates code that requires redundant load and inter-register shifts following a load operation. To get around this problem, you will need to use a data structure transformation such as dimension-lifted transposition.[4]

[4]T. Henretty, K. Stock, L. N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan, "Data layout transformation for stencil computations on short-vector SIMD architectures," *Compiler Construction*: Lecture Notes in Computer Science, Vol. 6601, 2011, pp. 225–245. http://link.springer.com/chapter/10.1007/978-3-642-19861-8_13#page-1

Rule 3: Optimal Cache Reuse

Cache use can be improved by using cache-blocking in the X and Y directions.[5] By blocking the code, you can increase the temporal locality. The goal of cache-blocking is to render those elements that are needed by the stencil code available in cache as the code works along the column of the Z-axis. This is shown in the hypothetical example in Code Listing 11-5 and graphically represented in Figure 11-4. If threads in the same core work on the same block or neighbor blocks, data reuse would be higher and balanced affinitization would enable better performance.

Code Listing 11-5. Stencil Pseudo-code Optimized with Blocking and Vectorization

```
for(t=0; t<numTimeSteps;t++){
#pragma omp for collapse(3)
   for(yy=2;yy<NY-2;yy+=By)
   for(xx=2;xx<NX-2;xx+=Bx)
    for(z=2; z<NZ-2; z++){
     for(y=yy; y<min(NY-2,yy+By-2); y++){
#pragma vector aligned //indicate no alising
#pragma vector nontemporal (fout) // indicate fout can be streaming store
      for(x=xx; x<min(NX-2,xx+Bx-2); x++){
       fout[z,y,x] = c1*f[z,y,x] +
        c2*f[z,y,x-2] + c3*f[z,y,x-1] + c4*f[z,y,x+1] + c5*f[z,y,x+2] +
        c6*f[z,y-2,x] + c7*f[z,y-1,x] + c8*f[z,y+1,x] + c9*f[z,y+2,x] +
        c10*f[z-2,y,x] + c11*f[z-1,y,x] + c12*f[z+1,y,x] + c13*f[z+2,y,x]
    }}}
   }}}
  double *tmp = fout; fout = f; f=tmp; //switch buffers
}
```

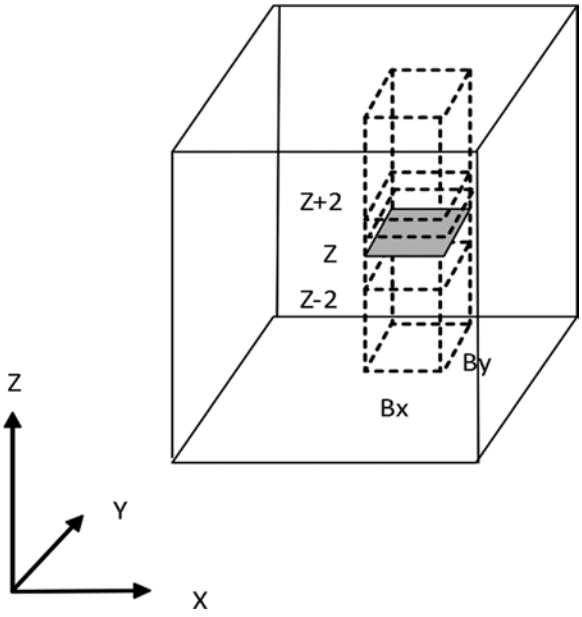

Figure 11-4. *Graphical representation of blocking optimization on cache reuse*

[5]G. Rivera and C. Tseng, "Tiling optimizations for 3D scientific computations." In *Proceedings of SC'00*, Dallas, TX, November 2000.

You need to pick the block sizes in Bx and By such that the XY planes formed by these blocks fit in the cache for the values calculated along the Z-axis. Since for each z iteration the fout[x,y,z] requires 5 planes of size Bx*By block of data to be residing in cache, the condition for selecting block sizes is 5*(Bx*By) < L2 cache size. The number **5** comes from the fact that there are five Bx*By planes corresponding to Z-1, Z-2, Z, Z+1, and Z+2 in the cache for the calculated stencil for all the points in that subblock shown in gray in Figure 11-4. You may improve the cache hit by pulling in the cache lines using software prefetch for z iterations.

European Option Pricing Using Monte Carlo Simulation in Financial Applications

Monte Carlo simulation is a computational method widely used in the financial sector to model option prices[6] of an underlying stock. This section looks at optimizing the Monte Carlo algorithm used for European option pricing on the Xeon Phi coprocessor.

Figure 11-5 is a simplified illustration of the basic step of Monte Carlo simulation of European option pricing. It suggests how such simulation can be optimized for Xeon Phi coprocessors. The general Monte Carlo process of option pricing starts by generating a set of random numbers (Step 1 in Figure 11-5). The simulation process takes a batch of queries to set option prices of the underlying security by specifying their input parameters, such as current stock price, option strike price, expiration date, and so forth. At Step 2, for each of these options to be calculated, the computation generates a block of possible stock price at the expiry date using a solution to the stochastic process that simulates underlying stock prices over time. The solution to the stochastic process for each random variate x generated in Step 1 results in a function that is of the form:

```
S(T) = S(0) Exp2(f(x,T))
        where:
        S(T) = Stock price at maturity.
        S(0) = Stock price at the option issue date.
        Exp2(f(x,T)) = exponential of a function of variate ‘x’ and time T
```

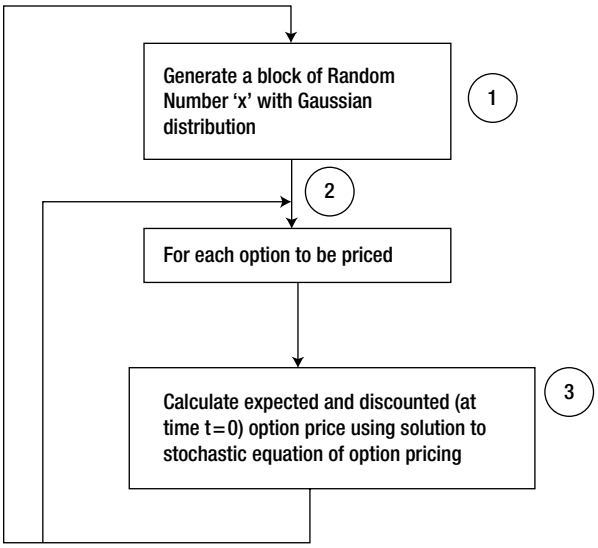

Figure 11-5. *Monte Carlo simulation of option pricing*

[6]Phelim P. Boyle, “Options: A Monte Carlo Approach,” *Journal of Financial Economics*, Vol. 4, No. 3, pp. 323-338, May 1977.

Profiling of the Monte Carlo simulation shows that the most compute-intensive part of the computation is evaluating the S(T) price in Step 3.

Rule 1: Scalable Parallelization

The Monte Carlo simulation is a highly parallel process involving the evaluation of thousands of options at a time. So it is easily distributed across Xeon Phi threads such that each thread can work on a subset of options to be evaluated in parallel with dynamic load balancing. This can be easily implemented by inserting the OpenMP '`parallel for`' construct in the code at Step 2 of Figure 11-5. The high-degree parallelization of this compute-bound algorithm can exploit a large number of cores in Xeon Phi, giving good speed up.

Rule 2: Efficient Vectorization

The code can be vectorized efficiently using vectorized versions of random number generators and transcendental functions. The MKL's vector statistical functions contain vector random number generators with various statistical distributions, including the Gaussian distribution needed by Monte Carlo simulation. Functions such as vsRngGaussian or vdRngGaussian from the Intel MKL can be used to generate vectorized single- and double-precision random numbers.[7] The loop at Step 2 in Figure 11-5 is vectorized by the compiler by using vector transcendental functions. For single-precision arithmetic, high-throughput transcendental instructions such as `vexp223ps` are implemented in Xeon Phi hardware and can provide an excellent boost in performance by vectorizing the key calculation at Step 3 in the loop. In my experiments, the Intel Compiler was able to vectorize such loops. For double-precision computation, you can use the Intel short vector math library to vectorize the computation as well. The biggest benefit and performance gain can be obtained by using the fast hardware-implemented transcendental functions provided the algorithm can handle the lower precision of these functions.

Rule 3: Optimal Cache Reuse

Monte Carlo simulations are compute-bound applications and the data usage easily fits in the L2 cache. So cache reuse is fairly optimal and not a concern for this algorithm.

Conforming to all three rules for optimal algorithms—scalable parallelization, efficient vectorization, and optimal cache reuse—Monte Carlo simulation is ideally suited to reap the potential benefits of the Intel Xeon Phi hardware.

Summary

This chapter looked at various common algorithms and their data structures and how they help and hinder performance while executing on Xeon Phi. You have seen that there are ways to restructure code and data structure so that they follow the three rules of scalable parallelization, efficient vectorization, and optimal cache reuse to achieve high performance on Xeon Phi. This chapter is in no way a comprehensive treatment of technical computing applications, but it does impart a sense of the vista of exciting possibilities now before you, as you prepare to leverage what you have learned in this book to develop new algorithms and data structures for the Intel Xeon Phi coprocessor.

[7] http://software.intel.com/sites/products/documentation/hpc/mkl/mklman/GUID-63196F25-5013-4038-8BCD-2613C4EF3DE4.htm