CHAPTER 10

Application Performance Tuning on Xeon Phi

This chapter explains how to tune the performance of applications developed for Xeon Phi. The work of achieving optimal performance starts by designing your application with proper consideration to application design and implementations, as discussed in Chapter 9. Once an application has been developed, you can tune it further by optimizing the code you have developed for the Xeon Phi coprocessor architecture. The tuning process involves the use of tools such as VTune, compiler, code structuring, and libraries in conjunction with your understanding of architecture to fix the issues that cause performance bottlenecks. The "artistic" aspect of the tuning process will emerge incrementally during the course of your hands-on work with the hardware and the application as you figure out how to apply various tools efficiently to optimize the code fragment that cause the bottleneck. This chapter will provide the *best-known methods* (BKMs) to start optimizing code for the Xeon Phi coprocessor. I will assume in this chapter that you have already parallelized the code as part of your algorithm design, as discussed in Chapter 9.

The optimization process can be broken into two main categories:

- *Node-level optimization*, in which you optimize the code for a single node.
- *Cluster-level optimization*, in which you optimize the performance at the cluster level.

The node-level optimization cycle consists of the following steps:

1. Set up a benchmark and baseline for the application you will be tuning for a single node.
2. Create a profile of the application and locate potential performance bottlenecks using a single node profiling tool such as VTune or a cluster-profiling tool such as Intel Trace Collector or Intel Trace Analyzer.
3. Set the target performance. From the application profile, you can try to estimate the application performance on the hardware. This process involves getting some estimate of the hardware performance using a common benchmark such as STREAM[1] or Scalable Heterogeneous Computing (SHOC) Benchmark[2] to measure the various performance metrics of Xeon Phi hardware. For example, if your application is bandwidth-bound,

[1]John D. McCalpin, "STREAM: Sustainable Memory Bandwidth in High Performance Computers," a continually updated technical report (1991-2007), available at http://www.cs.virginia.edu/stream/.

[2]Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter, "The Scalable Heterogeneous Computing (SHOC) Benchmark Suite," Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, March 14, 2010, Pittsburgh, Pennsylvania. Xeon Phi version of the benchmark available at https://github.com/vetter/shoc-mic.

you can use the stream number to predict how much headway you have. VTune can tell you how much bandwidth you are using, so that you can see what headroom you have for your application if it is memory-bound.

4. Experiment with applicable system-level configurations, such as transparent huge page settings, to address the performance issues.
5. Use compiler switches, pragmas, and code restructuring to go around the performance issues pointed to by VTune or other profilers.
6. If the code uses common math functions that are available in the MKL libraries, use instead the libraries that are optimized for Xeon Phi.
7. If the threading overhead is high, look for ways to reduce the threading imbalance or threading constructs to be more efficient. Try playing with such parameters as the thread affinity and the number of threads.
8. For symmetric applications, make sure you have load balance between the host and coprocessor components of your application.
9. Collect the new performance numbers and repeat from step 2 till you are satisfied with the code performance.

Once you are happy with the node-level optimization, proceed to cluster-level optimization as follows:

1. Use cluster-profiling tools such as Intel Trace Collector and Intel Trace Analyzer to see whether you have load imbalance.
2. Optimize the MPI parameters for lower message overhead. You may need code restructuring for optimal MPI performance.
3. Optimize OS, MPSS, and network configuration for best performance.

This chapter will explain each of these steps for extracting extract good performance out of the Xeon Phi coprocessor-based system.

Getting Baseline Data

It is important to set up a performance discipline that will let you know whether the optimization work you do has any benefit. Not all optimization efforts will provide performance gains. Indeed, some optimization strategies that seem good choices on theoretical grounds may even degrade performance. That is why it is important to take a disciplined approach to performance measurement. You need to set up a tracking process—perhaps an Excel spreadsheet where you log the system setup details such as the host processor and the Xeon Phi coprocessor specifications, the BIOS configuration of the host such as hyper threading on/off, the memory size, and setup such as the ECC status, the tools and library versions used, cluster configuration, and so forth—that will allow you to reproduce the benchmark results on a different machine if need be or provide run-to-run consistency.

Once you have set up a discipline, you will need to get a baseline performance number for your application. A timer is used to measure the performance of your application. It is important that you use the same timer on the coprocessor OS or host processor (if measuring offload code) when measuring application performance over time to be able to assess the effect of optimizations that you incorporate in the codebase. The coprocessor OS supports two hardware-based timers: the *elapsed-time counter* (ETC) and the *timestamp counter* (TSC) as a clock source. So if your application calls the `gettimeofday()` routine, the time values will be provided by the coprocessor OS based on the source you select as a timer source. It is important to make sure you keep the timer source constant in run-to-run measurements, as the resolutions are different between the timers, and that you select the proper timer for accuracy.

There is one ETC for the whole coprocessor chip and it is independent of chip power management. It has a high consistency as it is not affected by power management, but its access time is approximately 100x slower than that of TSC when using the `gettimeofday` function call. It is always a good idea to have your code timed at high granularity—say, by increasing the number of iterations timed to make the timer overhead negligible to time spent in the code execution that you are trying to time.

The TSC timer is faster to access, but it is affected by the power management. So if you are measuring performance with TSC, you need to make sure the chip power management in turned off. You can check which clock source is being used by looking at `/sys/devices/system/clocksource/clocksource0/current_clocksource` on the coprocessor.

Timing Applications

Timers are functions or tools that allow you to time the execution of a code fragment you are trying to optimize. For example, the coprocessor BusyBox utility supports the `time` command, which allows you to time the total runtime of the application at a coarse level. It will output the real time, user time, and system time after the application run completes. The *real time* is the elapsed wall clock time that you can measure with a stop watch. The *user time* is the sum of all processor time spent in user mode code. Note that for parallel applications, the sum of the time spent in each user process can be larger than the elapsed time. The *system time* accounts for time spent in the kernel for the process being timed.

Sometimes you will need a finer resolution timer than provided by the Linux `time` command. For example, in Xeon Phi–based code optimization, the focus is often on the part of the code that is offloaded to the coprocessor or of interest in native or symmetric runs. In such cases, you will need to have some function calls in your code that allow you to time specific sections of the code.

In Fortran, you can use the `RTC`, `DTIME`, and `ETIME` functions to time sections of the code. These functions are supported by the Intel Fortran compiler. In the C/C++ compiler, you can use `clock`, `times`, `getrusage`, `gettimeofday()`, or other functions to time your code. Code Listing 10-1 shows how you can use `gettimeofday` in your code to measure time. Here you can call elapsed time before and after a code segment you want to measure to collect time spent in your code.

Code Listing 10-1. Timing Routine Returning Time Value in Seconds

```
#include <sys/time.h>

extern double elapsedTime (void)
{
    struct timeval t;
    gettimeofday(&t, 0);
    return ((double)t.tv_sec + ((double)t.tv_usec / 1000000.0));
}
```

If you are interested in cycle-level timing, there is an intrinsic `__rdtsc()` (read time stamp counter) which reads from a 64-bit register and counts the number of cycles since the last reset. You need to use the core frequency to convert these cycles to `second` units. This counter is affected, however, by the power management on the core and it has to be used carefully.

Remember to find the timer resolution you will be using to measure a section of code segment. The resolution should be high enough to make the measurement relevant.

Detecting Application Execution Bottlenecks

Once you have your baseline data, you are ready to investigate whether there are opportunities for code optimization. At this step, you will use some sort of profiling tool such as Intel VTune Amplifier XE to locate the hotspots in your code. In general, *hotspot* refers to a code section where the code is spending most of its time. For technical computing applications, you should locate the top ten functions where more than 95 percent of the time is being spent. The fewer the number of functions showing up in your top 95 percent of execution, the better your chance of being able to optimize the code to get better performance with minimal effort. The greater the number of hotspot functions in your applications, as is often the case for flat-profile applications, the greater the effort and time required to get better performance out of those applications.

First, you need to resolve system-level issues, such as whether your offloaded code is being limited by the PCIe data transfer bandwidth. Using your timing routine, you can see whether or not the PCIe data transfer bandwidth is acceptable. To get a baseline, you can run some open source benchmark such as SHOC to get your estimated PCIe bandwidth. Once you are happy with the system-level performance, you can use VTune to profile the code running on Xeon Phi using performance monitoring hardware counters. The hardware is a black box to software developers with respect to how the instructions flow through the pipeline as the code executes. One way the hardware communicates with the software users is though these performance monitoring counters implemented in the coprocessors. These counters expose how the instructions are flowing through the cores, what type of cache hits and misses are happening, and how memory bandwidth is consumed for reading and writing data using the knowledge of cores and uncores in the coprocessor. Hardware architects usually build these counters to help with this bottleneck detection process.

The performance monitoring works by the hardware supporting various *performance events* related to core execution and methods to configure and capture those events. VTune allows you to select which performance event you are interested in and lets you choose how often the hardware event is sampled. When an application is running, you can select to monitor its effect on the hardware instruction flow pipeline to locate what type of issues you are facing. I will not be covering this topic in this chapter, but you can find details in the Intel Xeon Phi Coprocessor Performance Monitoring Units documentation.[3]

Performance events are associated with various architecture features that were discussed in Chapters 3 through 5. Most of the performance monitoring counters that were available on Pentium processor core are available on Xeon Phi. In addition, new counters appropriate to a multicore processor with a new MIC instruction set and features are added to Xeon Phi to expose the new architectural behavior for code execution.

Using these events, you can gather some interesting architectural behavior while executing your application. From the data for these events, you can create useful metrics for memory access performance, such as various levels of cache miss rate (miss/references); for core execution issues, such as vector unit usage efficiency; for parallel overhead, which can be discovered by the percentage of time spent in noncomputational code such as OpenMP or MPI library code; and for many other areas of architectural behavior.

From these metrics, you will be able to figure out whether the hotspot code is getting limited by memory access issues such as bandwidth, latencies due to cache miss, floating-point execution issues, integer execution issues, and so on. Equipped with such knowledge, you will be able to apply the various tuning techniques discussed in this chapter. VTune can also show you whether your code is spending too much time in MPI/OpenMP or parallel constructs that are provided to make use of the cores in parallel but that do not do any of the useful computational work needed by your application. Your goal is to increase the ratio of useful task to parallel overhead in Xeon Phi architecture. This can be accomplished by increasing the *effective computation*—that is, the actual computation that your threads in thread-level parallelism or MPI tasks are doing compared to communication/synchronization overheads.

[3]Intel Xeon Phi Coprocessor (codename: Knights Corner) Performance Monitoring Units, Document Number:327357-001 http://software.intel.com/sites/default/files/forum/278102/intelr-xeon-phitm-pmu-rev1.01.pdf.

Some Basic Performance Events

Although there are many performance events that are supported in the Xeon Phi coprocessor, not all of them are needed in the initial phase of performance root cause. I have listed some of the events that may be useful in your exploration process. These events, together with VTune profile analysis feature, can be useful in detecting your application issues related to execution bottleneck, data fetching overhead such as latencies or bandwidth limitations, and parallelization overhead.

Locating Hotspots

The most important events that you need to collect in order to locate hotspots are CPU_CLK_UNHALTED, which gives you the number of cycles executed in the core, and INSTRUCTIONS_EXECUTED, which gives you the number of instructions executed by the core. Using these two events, you will be able to pinpoint the code fragments and functions that are taking most of your code's execution time. Although software developers have a tendency to associate *cycles per instruction* (CPI) to performance, it is important to note that you are interested in reducing the total number of cycles taken by your application, not necessarily the CPI value. If you are executing mainly vector instructions, you might have a higher CPI than if you were executing scalar instructions, because vector instruction latencies might be 4 cycles compared to some scalar operations. Using vector instructions will, however, reduce the number of instructions by up to 16x for floating-point operations, so overall runtime will be lower.

Code Execution Issues

Once you have located the hotspots, you may want to proceed to investigate the vectorization efficiency of the hotspot sections of your code. As discussed in Chapter 3, utilizing vector units efficiently is key to achieving high performance on Xeon Phi. You can use the ratio of `VPU_ELEMENTS_ACTIVE`, the number of elements in a VPU register that were not masked out and actively participated in the vector execution to `VPU_INSTRCUTIONS_EXECUTED`, so signifying the number of vector instructions executed by the thread, to get the approximate vector efficiency. The theoretical maximum for single-precision arithmetic, for example, is 16, because there are 16 elements that can be acted on per vector instruction. This is only an approximate indicator because the `VPU_INSTRUCTIONS_EXECUTED` includes vector loads, masks manipulation instructions, and so on—so the ratio may be lower than the actual vector execution efficiency. To make an assessment of how good the metric is as applied to your code, look at the corresponding assembly code block. You will learn in this chapter how to improve the vectorization of your code by applying the compiler and tools.

In addition to vectorization inefficiencies, too many branches might lower your code's performance. You can use the performance events, `VPU_INSTRUCTIONS_EXECUTED`, and `INSTRUCTIONS_EXECUTED`, to figure out what percentage of executed instructions is vectorized. However, not all vectorized instructions are efficient because the usage of masks may prevent the vector unit from working on all vector elements at the same time. Sometimes the compiler will generate vector code even for scalar operations. To see how many average vector elements are active with each VPU instruction executed, you can use the `VPU_ELEMENTS_ACTIVE/VPU_INSTRUCTIONS_EXECUTED` ratio.

Data Access Issues/Stalls

The next step is to look at the cache hit rate. You might have good vectorization efficiency, yet your data are arriving late. Most of the issues in Xeon Phi optimization are related to accessing memory efficiently. You can approximate the average clock cycle spent in executing the vector instructions in a code segment by dividing `EXEC_STAGE_CYCLES by VPU_INSTRUCTIONS_EXECUTED` to estimate the instruction latencies. If you find that this metric is higher than expected

for an efficiently vectorized loop—say, more than the latencies for each instruction—you will need to root-cause the stalls happening in the processing the loop. Possible reasons why the memory subsystem may be having an issue delivering application data to the execution units on time include:

- Missing L1/L2 caches for data access
- Missing TLB entries
- Saturating the number of data streams that the hardware is able to fetch simultaneously

Recognizing Memory Access Latencies

To get the L1 miss ratio, divide the per thread events `DATA_READ_MISS_OR_WRITE_MISS` (the L1 references that missed the cache) by `DATA_READ_OR_WRITE` (the total number of L1 cache accesses). It should be high (> 95%) to achieve good results. If you are missing the L1 and L2 cache by too much, you may be stalled trying to access the memory.

Recognizing TLB Issues

TLB misses are another possible source of data access issues. As discussed in Chapter 4, TLB misses can lead to delay in the memory load, which may in turn cause reduced performance. This phenomenon is often seen in technical computing applications where the memory access lacks locality and jumps across page boundaries many more times than the number of TLB entries available for a given page size within a tight loop. This effect can be measured using the L1 TLB miss ratio, which is obtained by dividing the `DATA_PAGE_WALK` events (the number of L1 TLB misses) by the `DATA_READ_WRITE` events (the number of read-write operations). You can also calculate the L2 TLB miss ratio by dividing the `LONG_DATA_PAGE_WALK` events (the number of L2 TLB misses) by the number of `DATA_READ_WRITE` events (the total read-write operations performed by the application). For a 4K page, if sequentially accessed, this number should be 1/64 since there are 64 cache lines in a 4K page. So anything near 1 indicates that there is a heavy miss in the TLB due to a capacity or associativity conflict, which needs to be looked at carefully in the code. In this case, you can restructure the code and data structure to ameliorate this issue.

Recognizing Bandwidth Saturation

You also need to find out whether the code is saturating the memory by measuring the bandwidth used by your application. The VTune analyzer contains a custom profile for Xeon Phi to help with the process using uncore events. If you are saturating the memory bus, you will need to restructure the code or modify the algorithm to put less pressure on the memory subsystem and increase data reuse using cached data. If you are not saturating the memory bus, you may be able to insert proper prefetches to reduce the L1 and L2 misses. If the bus is saturated, you may be able to restructure code and data to reduce bus pressure.

Parallel Execution Overhead

Finally, you want to reduce the parallel overhead by balancing your workload and increasing the work for each core. Parallel execution overhead will show up in the profile as `CPU_CLK_UNHALTED` in OpenMP/MPI or other parallelization constructs. If you look at the parallel execution profile that shows up as a timeline in VTune, you will recognize the imbalance in the thread execution that may be causing the high OpenMP overhead. For MPI processes, you can use the cluster checker/profiler tool to recognize the parallel execution imbalance and overhead in the application.

Setting Target Performance

After recognizing that the code is memory bandwidth-, memory latency-, or compute-bound, you need to set a target performance for your application. Your application performance may also be bound by the PCIe bus bandwidth. The first step to setting the performance target is to use some standard Xeon Phi-optimized benchmarks or microbenchmarks to set the expected optimal performance of the Xeon Phi hardware you will be using. The SHOC benchmark that has been optimized for MIC can be used as a starting point. For example, the components of the SHOC benchmark, *BusSpeedDownload* and *BusSpeedUpload*, allow you to see the bandwidth and latencies of data transfer over the PCIe bus for various data sizes. You can use these benchmarks to estimate the performance of your application if it is PCIe bandwidth-bound. Once this is set, you can use various methods to reduce the bottleneck to change your code to go around the bottleneck. Similarly, you can run the Xeon Phi-ported STREAM benchmark to see and set the possible GDDR memory bandwidth achievable on your applications. Sometimes, because there is a limitation on the number of outstanding read buffers that the hardware provides, if your application has more streams of data being accessed than the hardware is capable of supporting, you may see a drop in achievable bandwidth. In this case, you may want to modify the STREAM or create your own microbenchmark to view the performance when the number of data streams you access is more than that used in the STREAM benchmark. Similarly, you may be able to use the peak achievable floating-point operation for single and double precision measured with the SHOC MaxFlops benchmark.

As for memory latency-bound applications, experience shows that the optimization of these workloads converts to bandwidth-bound applications. This is due to the fact that the optimization of such workloads includes prefetching of data in addition to other algorithmic or code restructuring work. Predicting the performance of such code is possible by using the bandwidth achievable by such applications when properly optimized.

Once you know the theoretical limit, a simple method for estimating your performance is to use the following equation:

$$P_T = P_C \times B_A/B_C$$

where

P_T = *target code performance*

P_C = *current code performance*

B_A = *achievable performance of bottleneck metric. This will be the peak achievable bandwidth, flops, and so forth for the benchmark condition.*

B_C = *current performance of the bottleneck metric for the code segment under optimization.*

It is extremely important to set a target performance for better understanding of the hardware performance as well as your code. This also guides you as to how much headroom there is for you to optimize a section of the application code and thus plan accordingly. It might also be possible for you to recognize that the code may not be suitable for optimization on Xeon Phi at this stage. For example, if the code works on byte-sized data (8-bit units) as the basic part of computation and cannot be cast into the basic vector units of Xeon Phi, which are 32-bit units, this may not be suitable for utilizing vector units properly and may require a fundamental change in the algorithm/data structure to make the code fit the architecture.

Optimizing Code

Once you have set a target performance, it is time to use various tools to optimize out the hotspots. The first step is to determine the tool to use. If you are looking at some math functions, it is best to see whether these are supported by any math library that has optimized code for Xeon Phi. This will help you get the maximum benefit out of the optimization work done by the library developers to achieve performance in a short time. The next step might be to use a compiler to optimize the code. This is a fairly involved process and requires understanding how the code compiler does some code transformations and their effect on bottlenecks.

Compiler-Driven Optimizations

To begin with, it may be possible to play with various compiler switches that help you generate a code stream based on the knowledge you have for the application. Optimization engineers have figured out ways to restructure your code so that it can get around various bottlenecks. Table 10-1 lists such optimizations and their effect on the various types of bottlenecks already discussed. Many of the code changes listed in the table can be done by modern compilers such as Intel Compiler automatically or through pragmas without requiring manual changes to code.

Table 10-1. *Code Restructuring Techniques to Reduce Specific Performance Bottlenecks*

Optimizations	Bottlenecks					
	Vector Instruction Generation and Execution	Branch Issues	TLB Issues	Data Access Latencies	Memory Bandwidth Issue	Parallel Overhead
Prefetching				+	–	
Data alignment	+			+		
Removing pointer aliasing	+					
Streaming Store					+	
Using large pages			+	–		
Loop interchange			+	+		
Loop fusion	+			+	+/–	
Loop fission	+					
Loop peeling		+				
Cache blocking			+	+		
Unroll	+					
Unroll and jam					+	
Using Intel Cilk Plus array notation	+					+
OpenMP/Cilk Plus/ TBB Optimization						+
Using compiler supported classes/ elemental function/ libraries (memset/ svmlclasses)	+				+	

Note: + indicates reduces the bottleneck; – indicates may add to bottleneck.

Prefetching

Prefetching is a technique that can be used for reducing memory load latencies, but it may increase the GDDR memory bus bandwidth. Prefetching the data needed by a vector computation can reduce the data latencies and thus improve the vector performance. The Xeon Phi coprocessor supports a hardware prefetcher to prefetch L2 data when it recognizes certain data access patterns. But you will need to use software prefetches to get data to the L1 cache and for cases where the hardware is not able to recognize the access patterns. Intel Compiler provides compiler options and pragmas to help with software prefetching. Since the coprocessor fetches a cache line at a time, a single prefetch is needed to get 16 single-precision or 8 double-precision consecutive numbers in a cache line. Prefetching data helps when the data are not in the cache and is used in the subsequent computation before being evicted by a subsequent data load without being used. It is important that you use the proper prefetch distance so that the data fetch is timed properly to be used by the intended instruction. Prefetching uses bandwidth, so untimed or unnecessary data fetch may cause a loss of useful bandwidth, cache lines, and consequently the application performance.

Prefetching is turned on by default in the Intel Compiler at an optimization level at and over '–O2' and issued for regular memory access inside a loop. You can use the compiler report option "`-opt-report-phase hlo –opt-report 3`" to see the prefetching the compiler generates for each loop. You can use '`-no-opt-prefetch`' to turn off prefetching. Here 'hlo' stands for *high-level optimization*. Often this helps in testing whether compiler-generated prefetching is helping or hurting your application performance.

Intel Compiler can also generate prefetches for pointer access where addresses can be predicted in advance. Intel Compiler generates two prefetches: `VPREFETCH1` from memory to L2, and `VPREFETCH0` from L2 to L1 cache for each memory access. The prefetch distance is determined by the compiler heuristics but can be controlled by the compiler option `–opt-prefetch-distance=d1[,d2]`, where d1 is the prefetch distance for `vprefetch1` and optional d2 is for `vprefetch0.` The prefetch distance is expressed as the number of loop iterations after the loop is vectorized. If you want to prefetch only from L2 to L1, you can set d1 to 0 in the above compiler option.

You can also use compiler supported intrinsics to add your own prefetch instructions to the code. This is especially useful for indirect accesses like `a[index[i]]`. The Xeon Phi hardware prefetcher does not kick in if the software prefetch is successful.

Intel Compiler supports two C++ compiler pragmas and corresponding Fortran directives—`pragma prefetch var:hint:distance` and `pragma noprefetch`—to turn on or off the prefetch for a specific loop or function. If there are a lot of L2 misses, software prefetching is critical for the Xeon Phi coprocessor as this indicates that hardware prefetching is ineffective. In this case, it is critical to play with the software prefetch intrinsics, pragmas, and compiler switches for improving application performance. You can also provide clues to the compiler by using "loop count" directives to help the compiler with the software prefetch code generation.

Data Alignment

Data alignment is a technique that can be used to improve vector code generation and reduce memory load latencies. Aligning your computational data and letting the compiler know that the data being accessed are aligned to the 64-byte boundary can help the compiler generate efficient vector code for the Xeon Phi coprocessor. If the compiler knows that the array element accessed in a vectorized loop is aligned to the 64-byte boundary, it can avoid generating some prologue code needed to deal with non-cacheline-size-aligned arrays.

You can define an array to be aligned to a certain bytes' boundary by using `__attribute__((aligned(Byte_aligned))` in C++ and `!dir$ attributes align:Byte_aligned` in Fortran.

For example, to allow the compiler to align array X to the 64-byte boundary, you can say `float X[1000] __attribute__(aligned(64))`. You can also use the compiler switch '–align' to allocate all arrays to certain byte boundaries.

For dynamic array allocation to aligned boundary, you can use `__mm_malloc()` and `__mm_free()` functions in the C++ compiler.

To communicate to the compiler that an array or pointer is aligned so that it can generate efficient vector code, you can use the pragma or directive "vector aligned" in Intel Compiler or attribute "aligned" with OpenMP4.0 directive `omp simd` or `omp declare simd`.

To declare that a specific data element is aligned to a certain byte boundary, you can use the `__assume_aligned (data, byte_aligned)` macro in C++ or the `ASSUME_ALIGNED data:byte aligned` directive in Fortran.

Once a loop is vectorized with aligned memory, there may still be some remainder loop to help with data access at the end of the loop that is less than the cache line size. In this case, you can pad your array with extra bytes to make them a multiple of cache lines. You can use a compile time switch `-opt-assume-safe-padding` to tell the compiler to assume that the arrays are padded properly to make their length a multiple of the cache line size. This will allow the compiler to remove the reminder/epilogue loop and reduce your code path length and hence potentially improve performance.

Data alignments also help data transfer over the PCIe bus. If you align data to the cache line boundary, the compiler may be able to DMA application data directly over the PCIe bus from the host to the Xeon Phi coprocessor or vice versa.

Removing Pointer Aliasing

Removing pointer aliasing is a technique that can be used for vectorizing code. Intel C/C++ Compiler assumes that more than one pointer in a code segment may point to the same location and thus uses caution when optimizing these codes, often resulting in nonvectorized code generation. Even though Fortran language supports pointers, the language definition assumes that the pointer is not aliased by default, unlike the C language definition.

To let the compiler know that your code adheres to the ISO C aliasing rule, use the `-ansi-alias` compiler switch. This allows aggressive optimization of the code, and you will often find the code to be vectorized by the compiler compared to the case where the switch is not used. You can also use the `-restrict` option together with the C `restrict` keyword to express pointer nonaliasing of the function parameters.

Streaming Store

Steaming store is a technique that can be used for reducing GDDR memory bus bandwidth pressure. In general, the Xeon Phi memory write includes a memory read to get the data from the memory into the cache line. However, this causes unnecessary waste of bandwidth if the data are write-only. Xeon Phi implements special instructions `vmovnrngoaps` and `vmovnrngoapd` for the case in which the write is a streaming store and the data need not be read into the cache. These instructions are useful when you have unmasked cacheline-aligned vector writes. By default the compiler should generate these instructions using its own heuristics, but you need to make sure this happens if the code generated by the compiler does not contain these special instructions. To help the compiler generate streaming store for a loop, you first need to make sure that the array is aligned by using the pragma or directive "vector aligned" for the data array and specifying "`#pragma vector nontemporal`" or "`!DEC$ vector nontemporal`" for the data element. You can also force the compiler to use streaming store by using the switch "`-opt-streaming-store always`".

You can use the compiler reporting mechanism "`-vec-report6`" to see whether or not the compiler generated streaming stores for a loop.

Using Large Pages

The *page size* is a block of virtual address space that the coprocessor OS uses for memory management. For example, if the page size is 4K, 4MB of the page requested by an application will use 1000 pages. The Xeon Phi coprocessor's memory management unit supports page sizes 4kB and 2MB. A TLB is used to cache the virtual to physical page address mapping so that when applications try to access a physical page, depending on a virtual address, the TLB entry can be used to locate the physical memory location quickly. When a mapping is not cached in the TLB entries, the translation is done manually by walking a four-level table structure, which is kept in memory. The walking of the table structure involves pointer-chasing code and is time-consuming, especially on low-frequency cores such as that of the Xeon Phi coprocessor. Sometimes when an application is accessing memory sequentially and the page misses are too high, it may be beneficial to use larger page sizes so that more address space mapping is available through the TLB structure of the coprocessor, thus reducing the number of TLB misses. With 8 TLB entries in 2MB pages, you can

cover 16 MB of memory address spaces, whereas with 64 entries for 4kB pages you can only cover 248 kB only. On the other hand, if you are randomly accessing pages (say, larger than 2MB) all over the virtual address space and you do not use much data out of that page, it may be more beneficial to use 4kB pages as they provide more TLB entries than 2MB pages and thus could reduce the number of TLB misses on Xeon Phi. Another problem with using 2MB pages is the possibility of allocating more memory than needed—if, say, only 1 byte per 2MB pages is touched, you may want to use 4kB pages. The goal is always to reduce the TLB misses.

The Xeon Phi coprocessor OS supports *transparent huge pages* (THP). The THP support automatically promotes or demotes page sizes on the Xeon Phi coprocessor OS. You can control the huge page support in Xeon Phi by using the "`/sys/kernel/mm/transparent_hugepage/enabled`" file in the coprocessor OS virtual file system.

To disable the THP, do:

```
echo never > /sys/kernel/mm/transparent_hugepage/enabled
```

To enable the THP, do:

```
echo always > /sys/kernel/mm/transparent_hugepage/enabled
```

To be able to control the THP programmatically, do:

```
echo madvise > /sys/kernel/mm/transparent_hugepage/enabled
```

The `madvise` is used when you do not want to waste page memory by enabling the THP system wide but want to do that for specific memory region. In this case, you can use the system call `int madvise(void *addr, size_t length, int MADV_HUGEPAGE);` to set the address range where the THP support is to be enabled for the applications.

Always play with this option in the Xeon Phi coprocessor, as it could have significant impact on code performance one way or the other.

Loop Optimizations

Intel Compiler provides various high-level loop optimizations that relieve the programmer from having to do them manually. Some of these are discussed below. In general, you should let the compiler perform the optimizations and do them manually only when the compiler is unable to perform them as required. Manually performing these transformations causes the code to be hard to read and maintain and may make them specific to a hardware architecture.

Loop Interchange

Intel Compiler can interchange loop indices to provide efficient memory access. This optimization is used to increase single stride references and thus make better use of cache lines and fewer TLB misses for large memory access.

Consider, for example, the following loop in C language.

Code Listing 10-2. Example of Loop Interchange before the Modification

```
for(j=0; j< LENGTH; j++){
        for(i=0; i<LENGTH; i++){
           data[i][j] = 0.0;
       }
}
```

Since the 'C' arrays are 'row' major, to access consecutive bytes, you need to increment the 'j' index inside the inner loop to make use of cachelines more efficiently. This can be done by interchanging these for loops and bringing the 'j' loop inside, as shown in Code Listing 10-3.

Code Listing 10-3. Example of Loop Interchange for Optimal Performance

```
for(i=0; i< LENGTH; j++){
        for(j=0; j<LENGTH; j++){
           data[i][j] = 0.0;
       }
}
```

This is done by Intel Compiler and can be detected if you turn on `-O3 -opt-report-phase HLO` in compile time options. It may look something like the following code.

Code Listing 10-4. Compiler Output Reporting Loop Interchange

```
LOOP INTERCHANGE in loops at line: x y z
Loopnest permutation ( 1 2 3 ) --> ( 2 3 1 )
```

Loop Fusion/Fission

Loop fusion is the process of fusing or merging two separate loops into a single loop to increase the data reuse or remove unnecessary data movement. Often in technical computing applications, one loop runs through a data stream, does some computation, and creates a secondary array, which is scanned again in a second loop immediately following the given loop to do some more computations. In such cases, it may be beneficial to merge the loops to make better use of bus bandwidth and increase the flops/byte ratio. Intel Compiler can perform loop fusion at optimization level `O3`. You can determine whether or not the compiler is fusing loops by turning on the *high-level optimization* (HLO) report. Sometimes you may want to tell the compiler not to fuse a set of loops, as they may hurt performance due to lack of enough hardware resources. For example, source codes with multiple memory read streams distributed across different loops may perform worse with loop fusion due to limits on the number of read buffers. In this case, you can use `#pragma nofusion` to prevent the compiler from fusing the loops that may cause performance degradation.

Loop fission is the opposite of loop fusion, distributing large loops into two smaller loops. It is useful when vectorization or software pipelining cannot take place due to high register pressure—that is, the code generator runs out of register to be able to vectorize the code. Intel Compiler supports pragma/directive "`distribute point`," which allows programmers to do loop fission without manual restructuring. If placed before a loop, the compiler will try to use heuristics to fission the loop. You can also place the directive inside the loop to explicitly tell the compiler where to perform the loop fission.

Loop Peeling

Often conditional statements are put inside a loop to handle boundary conditions, memory alignment, and so on. This may prevent vectorization, as the code inside the loop may become complex. In this case, it is possible to simplify the loop by peeling out the conditional case—say, the unaligned data access case outside the loop—and performing operations on the aligned data inside the loop, thus allowing vectorization of the loop.

This is another optimization that Intel Compiler performs and can be detected in the HLO report before implementing it manually.

Cache Blocking

Cache blocking is a very useful optimization for the Xeon Phi coprocessor to make optimal use of cache data. Since memory access is often the main bottleneck in many technical computing applications, it is a common optimization applied to such code. The optimization involves restructuring the data access in a loop so that they fit in the L1 or L2 cache. This is done by breaking the large array into smaller blocks of memory area, pulling the memory fragments into the cache, and working on them before moving to the next block of data. By controlling the data cache locality, the application can benefit from a high cache hit rate and thus improve performance.

The effectiveness of such optimization depends on the data block size, the cache size, and the reuse of the cache block. Intel Compiler applies cache blocking at the –03 optimization level and is reported by the HLO compiler optimization report.

If the compiler is not able to perform this optimization, it is often possible to perform this manually to get performance improvement on Xeon Phi applications.

Loop Unrolling

Loop unrolling is another common loop optimization performed by the Intel Compiler. By unrolling a loop, you provide more work to the coprocessor for each loop iteration, reducing the branches and the number of cache misses. For example, you can convert the loop in Code Listing 10-5 to something like that in Code Listing 10-6.

Code Listing 10-5. Loop Performing Data Copy

```
for(i=0; i<SIZE;i++){
   x[i] = y[i];
}
```

Listing 10-6. Loop Unrolled by 4

```
int lastBlock = 4*SIZE/4;
for(i=0; i<lastBlock;i+=4){
   x[i] = y[i];
   x[i+1] = y[i+1];
   x[i+2] = y[i+2];
   x[i+3] = y[i+3];
}

for(i=lastBlock;i<SIZE;i++){
   X[i] = y[i];
}
```

Fortunately, Intel Compiler performs these transformations for you, so you do not need to do them manually. It is often a good idea to let the compiler do the unroll optimization, because performing it manually may hurt other optimizations that the compiler may be able to perform if the original loop was maintained.

Intel Compiler supports the pragmas/directives “pragma unroll,” “pragma unroll(n),” and “pragma nounroll” to control loop unroll optimization.

Unroll and Jam

Unroll and jam refers to unrolling the outer loops and jamming them together into a new loop. This technique helps increase the flops/bytes ratio of the computations performed in the loop and is an important optimization for Xeon Phi architecture. Keep in mind, however, that this optimization increases the register pressure as the amount

of unrolling is increased and may cause performance drop if the loop runs out of registers causing register spill and fill. Intel Compiler supports the pragmas/directives "`pragma unroll_and_jam`" and "`pragma nounroll_and_jam`" to control this optimization.

Using Intel Cilk Plus Array Notation

Intel Cilk Plus array notation is a C/C++ language extension implemented by Intel Compiler to express a data-parallel operation on array objects. Expressing array operations in this notation helps Intel Compiler map these operations to Xeon Phi-implemented vector data and instructions.[4]

For example, you can express an array multiply and add of three arrays a, b and c of the same dimensions as follows:

```
c[:] = a[:]*b[:] + c[:];
```

This notation not only removes the explicit loop needed to operate on these arrays as done in conventional C program but also helps the compiler to map these array operations directly to the Xeon Phi vector FMA operations described in Chapter 3. This notation is a very useful and an easy way of expressing vector operations in code that can also provide a good performance gain.

Parallelizing Code with OpenMP/Cilk Plus/TBB

Intel Compiler supports OpenMP constructs and may help you parallelize your code easily to maximize the core utilization and performance on the Xeon Phi architecture. You can use the `-openmp-report` compiler switch to see the sections of the code/loop with OpenMP pragmas that the compiler is able to parallelize. Parallelization with low parallel runtime overhead and vector unit utilization is a must for Xeon Phi performance. In addition, runtime for OpenMP contains various functions or environment variables to control and improve OpenMP performance on Xeon Phi. In order to get optimal application performance, you need to make sure the load is balanced and can use various OpenMP scheduling schemes to enforce them. You can also play with the number of threads that provides optimal performance on your workload.

In order to get good performance, you need to make sure that the OpenMP threads are affinitized to hardware cores on the Xeon Phi coprocessors. You can use the "`KMP_AFFINITY`" environment variable to do so. You can determine, depending on how the data are shared among the threads, whether to place them in compact, scattered, balanced, or explicit modes. You can also use the environment variable `KMP_PLACE_THREADS` to place threads on the subset of the coprocessor cores, and it is easier to use than using explicit values in `KMP_AFFINITY`. Another useful environment variable is `KMP_SETTINGS`. When this variable is set, the OpenMP runtime will print out various OpenMP variable settings being used by the runtime. This will allow you to debug and tune the OpenMP execution.

A useful optimization with OpenMP is the *loop collapse* directive. In order to efficiently use all the cores, it is important that each core gets enough task to amortize for the OpenMP thread start/stop and synchronization overhead. In multiple nested loops, it may be possible to collapse loops so that each loop gets an increased amount of work between synchronization points. This is done through the OpenMP directive `omp parallel for collapse`.

OpenMP barriers implicitly used at the end of loops for thread synchronization can be a large overhead in Xeon Phi due to the sheer number of threads involved. You need to look for opportunities to remove such barriers by using the `nowait` clause where possible.

One of the drawbacks of OpenMP is the lack of *composability*. This means that if you are calling into a library—say, MKL—which may itself be using OpenMP parallelism, you may end up oversubscribing the cores of Xeon Phi processor, thereby causing degraded performance. This is the result of the OpenMP runtime for your code spawning a certain number of threads and the MKL library itself creating another set of threads. Users of OpenMP should

[4]For detailed syntax of Cilk Plus array notation, see *Intel® C++ Compiler XE 13.1 User and Reference Guide.* Document number: 323273-131US. (http://software.intel.com/sites/products/documentation/doclib/iss/2013/compiler/cpp-lin/).

carefully play with OpenMP threads for their applications and underlying libraries that use OpenMP to control oversubscription. For example, MKL provides its own environment variable to tell it how many threads to spawn to perform its tasks. Cilk Plus[5] or TBB[6] parallel constructs do not have these problems, because the parallelization in these libraries is based on the "work stealing" concept, whereby the number of threads is kept constant for the applications and, depending on the availability of idle cores, tasks are executed as needed. Note that OpenMP is available in both C++ and Fortran languages, whereas Cilk Plus is mainly C/C+, and TBB is purely a C++ language-based implementation.

Using Xeon Phi Optimized Class, Elemental Function, and Libraries in Intel Compiler

Intel Compiler provides various utility libraries and classes to allow users to make use of optimized code. The Short Vector Math Library (SVML) is supported by Intel Compiler and provides various vectorizable math functions. Intel Compiler may detect use of these functions in your code and use the SVML library to vectorize the code. Thus calling these routines in your code can help in producing vector code and thus providing optimal performance. The SVML includes transcendental functions such as cos, sin, tan, exp, log, erf, and so forth. The Intel Compiler reference guide for SVML lists these functions and their corresponding intrinsics.

Elemental function allows you to write data-parallel functions. This in turn allows the compiler to vectorize code at the function call site. So if you call such functions from inside a loop, the loop can be vectorized by the compiler.[7]

Intel Compiler has various default options that allow it to recognize certain constructs and apply optimized library calls instead of generating compiled code for those constructs. For example, it can recognize a matrix multiplication loop nest and use an optimized matrix multiplication library instead. You can use the compiler switch `-no-opt-matmul` if you want to use your own code. The compiler also has its own implementation of dynamic memory usage routines such as `_intel_fast_memcpy`, `_intel_fast_memset`, and `_intel_fast_memcmp`. These can be used by the compiler instead of generic memcpy, memset, or memcmp routines at the –O3 optimization level.

Vectorization with Intel Compiler

Intel Compiler provides strong support for vectorization using auto-vectorization and pragma-driven vectorization. It is a must to have efficient vector code running in order for Xeon Phi to provide performance, and Intel Compiler plays a big role in it. The steps for vectorization with Intel Compiler follow.

Step 1: Using Compiler Report

The vector report of Intel Compiler can tell you which loops are vectorized and which are not. The report will give you some clue as to why it did not vectorize specific loops. The –vec-report levels (0 to 6) control emission of the following vectorization report messages:

```
LOOP WAS VECTORIZED
loop was not vectorized: << reason >>
Reason, dependence information: dependence from xx to yy
```

[5] http://software.intel.com/en-us/intel-cilk-plus
[6] http://software.intel.com/en-us/intel-tbb/
[7] For detailed syntax of Cilk Plus array notation, see *Intel® C++ Compiler XE 13.1 User and Reference Guide.* Document number: 323273-131US. (http://software.intel.com/sites/products/documentation/doclib/iss/2013/compiler/cpp-lin/).

Step 2: Vectorizing Code

There are various techniques to vectorize a code if it does not autovectorize. The first step is to use the *guided autoparallelization* (GAP) feature in Intel Compiler. This is activated by using the –guide switch. It will provide advice on code changes, applying certain pragmas in your code and adding command line options to help vectorize the code.

The second method is to use the Cilk Plus array notation discussed in this section. Try to maintain unit stride array access for optimal code performance. You can apply other techniques shown in Table 10-1 that help with vectorization. For example, using the SVML can help vectorize the code using the random number generator inside a loop. It would also make sense to define your functions as vector elemental functions if they are called from inside a loop, which would help vectorize the loop that otherwise might not be vectorized.

Sometimes you will see that you vectorize a code yet it does not show any performance gain. Such failure might be due to inefficient vector code generation—as in the case of using scatter/gather instruction, which is not very efficient in Xeon Phi implementation. In such cases, you may want to restructure the code to let the compiler generate unit stride code.

Using the Math Kernel Library

Many technical computing applications may benefit from the MKL ported to Xeon Phi. If your application uses any library function available in the MKL, try using them for Xeon Phi. You can use following coding techniques to extract maximum performance out of the MKL on Xeon Phi processor:

1. MKL routines perform well when 2MB page sizes are used to hold the input/output data to MKL function calls. So make sure to play with THP support or other methods to utilize 2MB pages in MKL-based applications.
2. Align data to a 64-byte boundary.
3. Specifying `MKL_MIC_MAX_MEMORY` to set aside memory that can be used by MKL automatic offload can enhance MKL routine performance, as this allows the math routines to reserve and keep memory allocated on the coprocessor for optimized performance.
4. It may be beneficial to use suggested memory affinity for OpenMP when using certain MKL routines.

   For BLAS, LAPACK, and Sparse BLAS routines, set OpenMP affinity to:

   ```
   KMP_AFFINITY= compact, granularity=fine
   ```

   For FFT:

   ```
   KMP_AFFINITY=scatter, granularity=fine
   ```

   For FFT:

- Set OMP_NUM_THREADS to the power of 2.
- Set the number of threads to 128 if the total size of input and output data is less than 30MB, the approximate size of last level cache. Otherwise set to 4* (number of Xeon Phi cores).
- For 2D or higher FFTs and for single-precision use, leading dimensions should be divisible by 8 (half the vector length), not 16; and for double precision—the leading dimensions should be divisible by 4, not 8.

The section “Parallelizing Code with OpenMP/Cilk Plus/TBB” discussed how to detect and reduce threading overhead.

Cluster-Level Tuning

MPI can be used to develop and run applications on Xeon Phi in three different models. In the *coprocessor-only model*, all MPI ranks run on Xeon Phi natively and communicate with other Xeon Phi on the host or other cards in the cluster. In the *symmetric model*, the MPI ranks run on both hosts and Xeon Phi in a cluster. In the *offload model*, each MPI rank can run on the host and perform offload.

In all of these programming models, the MPI overhead on Xeon Phi may become a serious bottleneck and need some sort of debugging/profiling tools to resolve the issues. The Intel cluster tools Trace Collector and Analyzer[8] may help you with the cluster-level optimization of code running on a cluster of the Xeon Phi processor. This is a two-level process in which you first detect and resolve any MPI issue at the node level, where the Xeon Phi card looks like a separate node to the host. You use the Trace Collector and Analyzer to detect node-level issues. Here you can fix the load imbalance between the host code and the coprocessor code to achieve optimal node performance by optimizing the computation-to-communication ratio.

Remember that since the compute power is different between the host and Xeon Phi, you need to carefully think through how the work must be divided between the host and coprocessor for balanced execution. It is important to minimize the MPI communication between the host and the coprocessor in a node and across the node. You also need to think carefully about the MPI rank topology and use a mix of MPI+OpenMP on Xeon Phi to improve the computation-to-communication overhead within a node and across the nodes. Usually it is better to use the least number of MPI processes on Xeon Phi for communicating with other MPI processes on the host or other nodes in a cluster. This can be achieved with the hierarchical MPI or MPI+X (OpenMP or another threading mechanism) programming model.

Another option that may be used for cluster-level programming with the Xeon Phi coprocessor is to apply the offload model to use Xeon Phi from the MPI ranks running on each host node in the cluster. This MPI+offload model reduces the number of MPI communications between the nodes and thus reduces the cluster-level complexity during program execution.

Summary

This chapter reviewed the optimization techniques and processes for code development for the Xeon Phi processor. It looked at how Intel Compiler pragmas provide alternatives to manual code changes to achieve performance improvements. It also explained how to deploy various libraries such as the MKL and SVML to improve code performances.

The next chapter will discuss various case studies and techniques for optimizing a category of applications on Xeon Phi.

[8] http://software.intel.com/en-us/intel-trace-analyzer