Intel® Xeon Phi™Core Micro-architecture

Abstract

A processor core is the heart that determines the characteristics of a computer architecture. It is where the arithmetic and logic functions are mostly concentrated. The Instruction Set Architecture (ISA) is implemented in this portion of the circuitry. Although it should be noted that in a modern day architecture like Intel® Xeon Phi™ coprocessor, less than 20% of the chip area is dedicated to the core. Let us look at what led to the development of the Intel Xeon Phi architecture that will provide us some hints why the coprocessor core is designed the way it is done.

Article

The Intel® Pentium Pro–based processors designed around long execution pipeline and high degree of out of order instruction execution, reached a power barrier while trying to increase processor performance by increasing frequency. The demand for computing was still in its infancy and the computing industry started the move towards parallel and muticore programming to feed the demand. The technical computing industry needed more compute power than provided by existing multicore architecture to continue modeling real world problems in wide range of fields from oil and gas exploration to biomedical engineering. As an example, in oil and gas exploration field, high performance computing is a competitive advantage for those who can make use of these computing resources to drill efficiently for oil recovery. By reducing the number of holes drilled on earth, in search for oil reservoirs, helps the environment in addition to saving millions of dollars in expenses. However, good simulation needed for this requires processor performance and power efficiency in the near future may not be satiable by current serial or muticore based architecture. Current architectural design and roadmap will need disruptive technology to make each core substantially faster within the power envelope to meet the technical computing demand of the near future.

As a result, the computing industry started looking for alternate attached coprocessor solutions from Clearspeed*, various FPGA solutions, and graphics chips such as from AMD*, NVIDIA*, to improve performance of the scientific applications. However developing and maintaining such software seemed prohibitively costly for practical industrial development environment. What was needed was a universal coprocessor programming language and/or processor architecture to leverage the development and tuning expertise of today's software engineers to achieve the power and performance goals of future computational needs. The architecture team within Intel found out that the cores based on Intel® Pentium designs could be extremely power efficient on current semiconductor process architecture due to short pipelines and low frequency operations. These cores could also retain many of the existing programming models that most of the developers in the world were already using. For the technical computing industry, the years of investment that has been put in the software development has to be preserved. The underlying hardware may change, however compatibility and ease of portability of existing software plays a critical role in technical computing applications in selecting different hardware platforms.

This decision to use Intel® Pentium cores started the effort to develop Intel's first publicly available many integrated core architecture dubbed as Intel® MIC architecture.

In this architecture, a single in-order core is replicated up to 61 times in Intel® Xeon Phi™ design and placed in a high performance bidirectional ring network with fully coherent L2 caches. Each of the cores supports four hyper-threads to keep the core's computing process busy by pulling in data to hide latency. The cores are also designed to run at turbo modes, that is if the power envelop allows, the core frequency could be increased to increase performance.

These cores have dual issue pipelines with Intel 64 instruction support and 16 floating-point (32-bit) wide SIMD units with FMA support that can work on 16 single precision or 8 double precision data with a single instruction. The instructions can be pipelined at a throughput rate of one vector instructions per cycle. The core contains a 32-KB 8-way set associative L1 data and instruction cache. There are 512 KB per core L2 cache shared among four threads and there is a hardware prefetcher to prefetch cache data. The L2 caches between the cores are fully coherent.

The core is a 2-wide processor meaning it can execute two instructions per cycle, one on U-pipe and the other on V-pipe. It also contains an x87 unit to perform floating point instructions when needed.

The Intel Xeon Phi core has implemented a 512-bit Vector ISA that can execute 16 single-precision floating-point or 32-bit integer and 8 double-precision floating-point or 64-bit integer vector instructions. Vector units consist of 32x 512-bit vector registers and 8 mask registers to allow predicated execution on the vector elements. Support for Scatter/Gather

vector memory instructions makes assembly code generation easier for assembly coders or compiler engineers. Floating point operations are IEEE 745 2008 compliant. Intel Xeon Phi architecture supports single-precision transcendental instructions for exp, log, recip, sqrt functions in hardware.
The vector unit communicates with the core and executes vector instructions allocated in the U or V pipeline. The core can execute two instructions per clock, one on U-pipe and another on the V-pipe. The V-pipe executes a subset of the instructions and is governed by instruction pairing rules, which is important to account for in getting optimum processor performance.

Calculating Theoretical Performance of Intel Xeon Phi cores

For an instantiation of Intel® Xeon Phi™ coprocessor with 60 usable cores, running at 1.1 GHz, you can compute theoretical performance as follows (for single precision operations):

- GFLOP/sec =16 (SP SIMD Lane) x 2 (FMA) x 1.1 (GHZ) x 60 (# cores) = 2112 for single precision arithmetic
- GFLOP/sec = 8 (DP SIMD Lane) x 2 (FMA) x 1.1 (GHZ) x 60 (# cores) = 1056 for double precision arithmetic

Note: Since Intel® Xeon Phi™ processor runs an OS inside, which make take up a core to service hardware/software requests like interrupts. As such, often a 61 core processor may end up with 60 cores available for pure computation tasks.

Core Pipeline Stages

The core pipeline is divided into seven stages for integer instructions plus six extra stages for vector pipeline as shown in the Figure 1 below.
Each stage including E and prior stages is speculative since things such as a branch mispredict, data cache miss, or TLB miss can invalidate all the work done up to this stage. Once it enters the WB stage, it is done and updates the machines states.
Each core is 4-way multithreaded; that is, each core can concurrently execute instructions from four threads/processes. This helps reduce the effect of vector pipeline latency and memory access latencies, thus keeping the execution units busy.
Traditional instruction fetch stage (IF) is broken down into two stages. The first two stages PPF (pre thread picker) and PF (thread picker) are called thread picker stages, where it selects the thread to execute. The PPF stage prefetches instructions for a thread context into the prefetch buffers. There are four buffers per thread in the prefetch buffer space and these can contain 32 bytes each per buffer. There are two streams per thread. Once one of the stream is stalled, say due to branch mispredict, a second stream is switched in while the branched target stream is being prefetched.
The PF stage selects the thread to execute by sending the instruction pairs to decode stages. The hardware cannot issue instructions back to back from the same thread in the core. To reach full execution unit utilization at least two threads must be running at all times. This is important to note as this may affect execution performance in some scenarios [see code example below].Each of the four threads has a ready-to-run buffer (prefetch buffer) of two instructions deep as each core is able to issue two instructions per clock (U-pipe + V-pipe). The picker function (PF) examines the prefetch buffer to determine the next thread to schedule. Priority to refill (PPF) a prefetch buffer corresponding to a thread is given to the thread executing at current cycle. If the executing thread has a control transfer to a target not in the buffer, it will flush the buffer and will try to load the instruction from the instruction cache. If it misses the instruction cache, a core install will happen and this may cause a performance penalty. The prefetch function behaves in a round robin fashion when instructions are in the prefetch buffer. It is not possible to issue instructions from the same context in back-to-back cycles. The refill of the instruction prefetch buffer takes 4–5 cycles, which means it may take 3–4 threads running for optimal performance. When PPF and PF are properly synchronized, the core can execute in full speed even with two hardware contexts. When they are not synchronized (as in the case of a cache miss), a one clock bubble may be inserted. One possible solution to avoid this performance loss is to run three or more threads in such cases(useful for optimization).

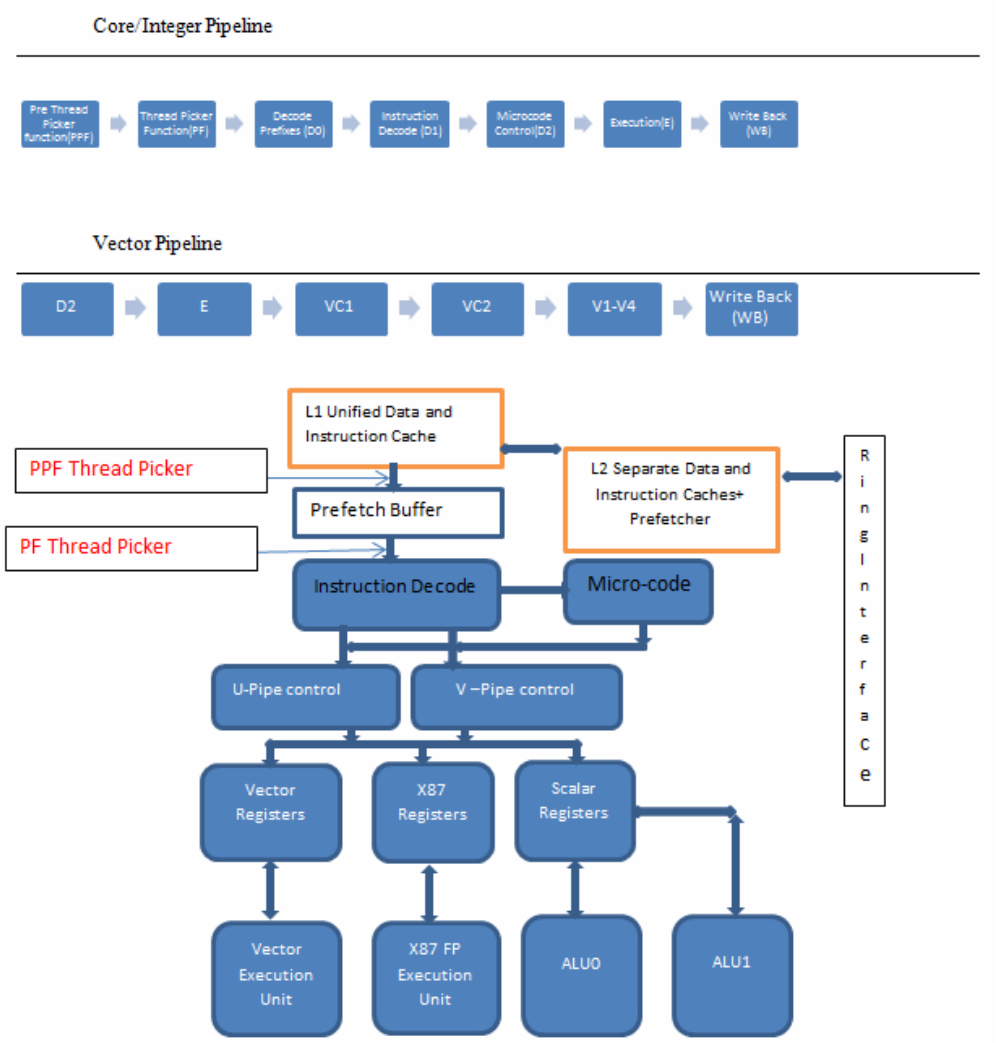

Figure 1 Coprocessor Core Integer and Vector Pipeline and Simplified Instruction/Data Flow Model

Once the thread picker has chosen an instruction to send down the pipe to instruction decode stages. Stage D0 and D1 decode them at the rate of two per clock. At the D0 stage we do the fast prefix decoding where a given set of prefixes can be decoded without penalty. Other sets of prefixes may imply a two clock penalty (legacy prefixes). The D1 stage ucode ROM is also a source of microcode, which is muxed in with the ucodes generated by the previous decoding stage. The processor reads the general purpose register file at D2 stage, does the address computation, and looks up the speculative data cache. The decoded instructions are sent down to execution unit using two paths called the U and V pipelines, named as such for historic reasons. U is the first path taken by the first instruction in the pair and the second instruction if pairable (there are some pairing rules that dictate which instruction can pair up with the instructions sent down the U-pipe) is sent down the V-pipe. At this stage for integer instructions, the instructions are executed in the ALUs. For scalar integer instructions, once they reach the writeback(WB) stage they are done. There is a separate pipeline for x87 floating point and vector instructions that starts after the core pipeline. For vector instructions, when they reach the WB stage, the core thinks they are done, but they are not done, because the vector unit keeps working on them until they are done at the end of the vector pipeline five cycles later. At this stage they don't raise any exceptions and will get done.

Intel Xeon Phi processors have global stall pipeline architecture; that is, part of the pipeline will stall if one of the stages is stalled for some reason. Modern Intel Xeon architecture has queues to buffer the stalls between the front and back end.

Many changes were made to the original 32-bit P54c architecture to make it into an Intel Xeon Phi 64-bit processor. The data cache was modified to non-blocking by implementing thread-specific flush. When a thread has a cache miss, it is

now possible to flush only the pipeline corresponding to that thread without blocking other threads. When the data is available for the thread that had a cache misses, it will be wake up.

Cache and TLB Structure

The details of the L1 instruction and data cache structure are shown in Table 1. The data cache allows simultaneous read and write allowing cache line replacement to happen in a single cycle. The L1 cache consists of 8 ways set associative 32 KB L1 instruction and 32 KB L1 data cache. The L1 cache access time is approximately 3 cycles as we shall measure in the exploration section. L2 cache is 8 way set associative and 512 KB in size. The cache is unified, that is it caches both data and instructions. The L2 cache latency could be as small as 14-15 cycles as measured in the core exploration section below.

Table 1 Intel® Xeon Phi™ L1 I/D Cache Configuration

Size	32KB
Associativity	8-way
Line Size	64 bytes
Bank Size	8 bytes
Outstanding Misses	8
Data Return	Out of order

L1 data TLB supports three page sizes, 4 KB, 64 KB, and 2MB, as shown in Table 2. It also has a L2 TLB that acts as a true second level TLM for 2-MB pages or acts as a cache for page directory entries (PDE) for 4-KB and 64-KB pages. If one misses L1 and also misses L2 TLB, one has to walk four levels of page table, which is pretty expensive. For 4-KB/64-KB pages, if one misses the L1 TLB but hits the L2 TLB, it will provide the page directory entry (PDE – see Figure 2) entry directly and be done with the page translation.

The page translation mechanism allows the applications to use much larger address space than physically available in the processor. The size of physical address is implementation specific of the hardware. Intel Xeon Phi supports 40 bit physical address in 64-bit mode that is the coprocessor will generate 40 bit physical address signal on the memory address bus of the coprocessor. Although Intel Xeon Phi supports various virtual address mode like 32 bit, physical address extension (36 bit) mode, we shall focus mainly on 64 bit mode as that is what is implemented through the micro os running on the coprocessor. In 64 bit mode, there is architectural support for applications to use 64 bits linear address. The ‘paging’ mechanism implemented in operating system allows these linear address used by an application to map to physical address which can be less than 64 bit. 64 bit linear address is used to address code, data and stack. The micro os running on the coprocessor uses 4 level hierarchical paging. Linear address generated by an application is grouped in fixed length interval known as pages. The micro-OS running on Intel Xeon Phi supports 4KB and 2MBpage sizes (although 4MB is implemented in the hardware is not supported by current micro-os). Application or OS may chose and move between various page sizes to reduce the TLB misses. The current micro-os as of this writing implements transparent huge page (THP) to automatically promote or demote pages sizes during an application run.

The operating system createsdata structures known as page table data structures which the hardware uses to translate linear addresses into physical address. These page tabledata structures reside in the memory and created and managed by the operating system or micro-os in case of Intel Xeon Phi. There are four levels of page table data structures:

Page Global Directory
Page Upper Directory
Page Middel Directory
Page Table

There is a special processor register, known as CR3 which points to the base of the data structure Page Global Directory. Each data structure contains a set of entries that points to next lower level table structure. The lowest level entries in the hierarchy points to the translated page which when combined with the offset from the linear address provide the physical address that can be used to access the memory location. The translation process is shown in Figure 2.

The linear address can be logically divided into

“Page Global Directory Offset” that combines with the base address of ‘Page Global Directory Table’ found in CR3 register to locate Page Upper Directory table.

“Page Directory Pointer entry” that selects an entry from Page Upper Directory Offset table to locate ‘Page Middle Directory Table’.

Page Directory Entrythat selects and entry from Page Middle Directory Offset table to determine the page table location in memory,
Page Table Entry that selects the physical page address by indexing into the Page Table discovered in step 3 above.
Page offset then provides the actual physical address from the page address discovered in step 4. This address is used by the hardware to fetch the data.
The sole work of TLB (translation look aside buffer) is to reduce the page walk necessary to locate the page corresponding to steps 1 through 4 and save the Page address discovered in step 4 in the TLB cache.
Each applications running in the operating system has a separate set of this table structure and switched in and out by changing the CR3 register value corresponding to the application.

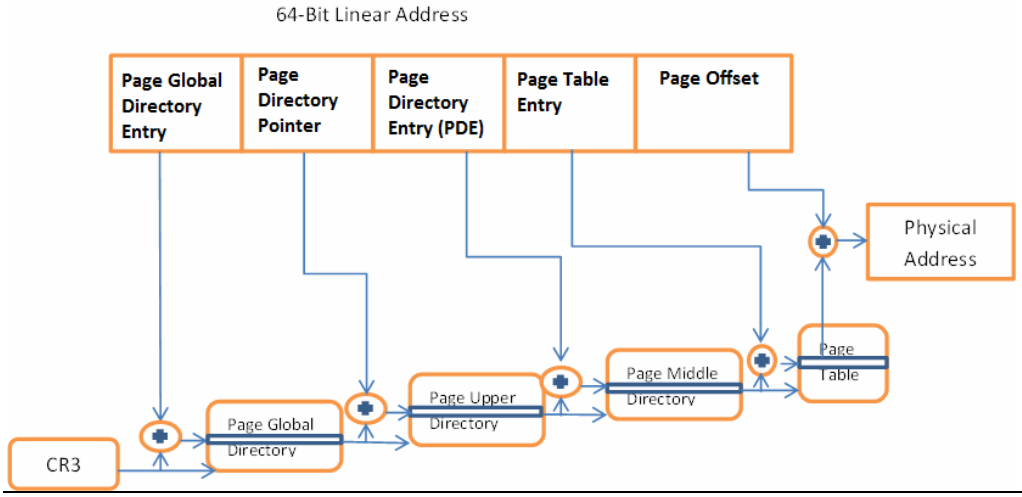

Figure 2: Linear to Physical address translation in Intel Xeon Phi Coprocessor

Table 2 Intel® Xeon Phi™ TLB Configuration

	Page Size	Entries	Associativity
L1 Data TLB	4KB	64	4-way
	64KB	32	4-way
	2MB	8	4-way
L1 Instruction TLB	4KB	64	4-way
L2 TLB	4KB, 64KB, 2MB	64	4-way

L2 Cache Structure

The L2 cache is the secondary cache for the core. L2 cache is inclusive of L1 cache. It is okay to have a cache line in L2 only, but L1 cache lines must have a copy in L2. L2 cache associated with each core is of size 512 KB. The cache is divided into 1024 sets and 8 ways per set with 64 bytes/1 cache line per way. The cache is divided into two logical banks.
L2 cache can deliver 64 bytes of read data to corresponding cores every two cycles and 64 bytes of write data every cycle.

Multithreading

The Intel Xeon Phi coprocessor uses time-multiplexed multithreading. The PPF thread picker determines the request to be sent to instruction cache and put into the prefetch buffer and the second one selects what entries to read from the prefetch buffer and send them to the decoder. It uses a smart round-robin mechanism to pick only threads that have work to do and avoids threads that are inactive due to various conditions like cache miss. One can control thread scheduling by putting delay loops in the thread that the thread picker will not schedule in the actual hardware(which is useful for optimization)..
In order to support multithreading, all architectural states are replicated four times. Micro architectural states of prefetch buffers, instruction pointers, segment descriptors, and exception logic are replicated four times as well. The design uses ID bits to distinguish between threads for shared structures like ITLB, DTLB, and BTB. All memory stalls are converted into thread-specific flushes.

Performance Considerations

Run two or more threads to have the core completely busy. Integer operations and mask instructions are single cycle. Most vector instructions have four-cycle latency and single-cycle throughput, so having four threads one will not see the vector unit latency if all threads are executing vector instructions. Use four threads to hide vector unit latencies.

Address Generation Interlock

AGI latencies are three clock cycles. When one writes to a GPR that is used as a base or index register, these instructions have to be spaced properly as the address generation is done in a separate stage in the pipeline. For example consider the following instruction sequence:

```
Add rbx,4
Mov rax,[rbx]
```

Here the calculation of the actual linear address from [rbx] is done in a separate stage before the memory fetch happens at line 2 above. In this case hardware will insert two clock delays between these two instructions. If running more than one thread, one may not see it as instructions from other threads can run during the dead clock cycles for another thread.

There is also dependent load/store latency. The cores do not forward the store buffer contents, even if the data is available there requested by the load instructions. So a load followed by a store to the same location will need the store buffer to write to the cache and then readback from the cache. In this case the latency is four clock cycles.

If there is a bank conflict, that is the U- and V-pipe try to access a memory location that resides on the same L2 cache bank, the core will introduce two clock-cycle delays for the V-pipe. When the cache line has to be replaced, it involves two clock-cycle delays. So if the code has a lot of vector instructions that miss cache, the core will have to put in a two cycle delay for each miss to do the replacement.

Prefix Decode

There are two classes of prefixes. The fast prefixes are decoded in 0 cycles with dedicated hardware blocks and include following prefixes: 62 for vector instructions, c4/c5 for mask instructions, REX prefix for integer instructions for 64-bit code and 0f prefix. All other prefixes are decoded in two-cycle latencies and include 66 prefixes for selecting 16-bit operand size, address size 67, lock prefix, segment prefix, and REP prefix for string instructions.

Pairing Rules

There are specific instructions that are pairable and thus can execute in both the U and V pipeline. These instructions include independent single-cycle instructions. Some dependency exceptions where two dependent instructions can execute in pair are cmp/jcc, push/push, pop/pop by incorporating instructions semantic knowledge into the hardware. For pairing to occur, instructions cannot have both displacement and immediate, and the instructions cannot be longer than 8 bytes. 62, c4 and c5 prefixed instructions with 32-bit displacement do not pair. Microcode instructions do not pair.

Integer multiply operations are done in x87 floating point units; that is why it is long latency instructions (10 cycles) in KNC. One needs to shift the integer operands to FP unitsto do the multiply and shift the results back to the integer pipeline.

Probing The Core

Measuring Peak Gflops

In this section we will examine the core microarchitecture and pipelines described above through experiments. Let's look at a code example of how can we measure the core computing power of a 61 core Intel® Xeon Phi™. In these processors there are 60 cores available for computation while one of the cores services the operating system. The code is listed in the code listing 4.1 below.

To start with the code needs to be designed such that the computation is core bound and not limited by the memory or cache bandwidth. To do so, we need the data to fit in the processor vector registers. This is limited to 32 for each thread in Intel Xeon Phi cores. The code uses OpenMP parallelism to run on 240 threads on the 60 core processors (with 4 threads/core). Please see appendix A for a brief back ground on OpenMP. In order to build OpenMP code we include <omp.h> header file. This file allows us to query various OpenMP parameters.

Writing the code in OpenMP will allow us to run the code from one core to multiple cores using an environment switch. Thus we can experiment with the effect of multiple threads in a core, to running the code on all cores to understand the compute power of the processor.

Following the code, In line 42 and 43 we declare two constants. The first constant 16 indicates the number of double precisions we should be using for our computation. In this architecture the vector units can work on 8 DP number per cycle. As wehave seen in architecture description section, each of the four threads per core has a ready-to-run buffer (prefetch buffer) of two instructions deep as each core is able to issue two instructions per clock. Thus having 16 DP elements allow us to put a pair of independent instructions in the fill buffer.I have also made the number of iterations ITER multiple of 240, the number of maximum threads I shall be running to balance the workload.

The function elapsed time used Linux* gettimeofday function to get the time elapsed for computation. Line 52 declared three arrays that I shall be working on each in each thread. I have aligned the arrays to 64 byte cacheline boundaries so that they can be loaded efficiently by the compiler. However, we will need to let the compiler know that the data is aligned by “pragma vector aligned” at line number 64 and 72. If you look carefully, all threads will be writing back to array a[] in line 65 and 73 in the code listing below. This will cause a race condition and will not be useful in real application code. However, for illustrative purposes, it can be ignored at this time.

In line 62 through 66 we warm up the cache and initialize OpenMP threads so that these overheads are not counted during the code timing loops that happen between lines 70 through 74. For easy reference, if you look the code fragment below, line 70 is a repetition of the omp parallel for first encountered at line 62. As such, all OpenMP overhead related to thread creation is captured at line 62 and the threads are reused at line #70. The OpenMP “pragma omp parallel for” will divide up the loop iterations statically in this default case among the available threads set by the environment variable. The pragma vector aligned at line #72 tells the compiler that the arrays a, b and c are all aligned to 64 byte boundary for Intel Xeon Phi and does not need to do any special load manipulations needed for unaligned data.

At line 78 and 79 the Gflops operations are computed. As the operations in line 73 are a fused multiply add operation, it is in effect computing 2 DP FLOP operations. Since there are SIZE number of double precision elements each operated over ITER number of times by one or multiple threads, depending on the OpenMP environment variable OMP_NUM_THREADS setting, the amount of computation is equal to 2*SIZE*ITER/1e+9 in Giga floating point operations or GFLOP. Dividing this number by duration measured by the timer routines at line 68 and 76 will allow us to compute the GFLOP/seconds which are an indication of the core performance. In addition, the Intel compiler allows us to control the code generation such that we can turn vectorization on or off.

Note that we have used special array notation supported by Intel Compiler known as Cilk+ notation described in Appendix B. In Cilk+ array notation, we are asking compiler to work on SIZE elements of double precision data. This help indicate to the array that the elements of the arrays a,b,c in this case are independent and thus allowing the compiler to vectorize the code which otherwise could be ambiguous to the compiler as for its vectorizability.

Code Listing 1

```
38 #include <stdio.h>
 39 #include <stdlib.h>
 40 #include <omp.h>
 41
 42 unsigned int const SIZE=16;
 43 unsigned int const ITER=48000000;
 44
 45 extern double elapsedTime (void);
 46
 47 int main()
 48 {
 49       double startTime,  duration;
 50       int i;
 51
 52       __declspec(aligned(64)) double a[SIZE],b[SIZE],c[SIZE];
 53
 54
 55       //intialize
 56       for (i=0; i<SIZE;i++)
 57       {
 58            c[i]=b[i]=a[i]=(double)rand();
 59       }
 60
 61 //warm up cache
 62       #pragma omp parallel for
 63       for(i=0; i<ITER;i++) {
 64       #pragma vector aligned (a,b,c)
 65        a[0:SIZE]=b[0:SIZE]*c[0:SIZE]+a[0:SIZE];
 66       }
 67
 68       startTime = elapsedTime();
 69
 70       #pragma omp parallel for
 71       for(i=0; i<ITER;i++) {
 72       #pragma vector aligned (a,b,c)
 73        a[0:SIZE]=b[0:SIZE]*c[0:SIZE]+a[0:SIZE];
 74       }
 75
 76       duration = elapsedTime() - startTime;
 77
 78       double Gflop = 2*SIZE*ITER/1e+9;
 79       double Gflops = Gflop/duration;
 80
81       printf("Running %d openmp threads\n",
omp_get_max_threads());
 82       printf("DP GFlops = %f\n", Gflops); 82
 83
 84   return 0;
 85
 86 }
```

Now that we are familiar with what the code is doing, let's measure the core performance of this architecture. To begin with we would need to build the code with vectorization turned off. The can be done by using the '-no-vec' command sent to Intel compiler. To build with the Intel Compiler with vectorization turned off we will use the following command line:


```
Command_prompt >icpc -O3 -mmic -opt-threads-per-core=2-no-vec  -openmp -vec-report3  dpflops.cpp
gettime.cpp -o dpflops.out
```

Where:

1. icpc is the Intel c++ compiler invocation command.
2. The source file, dpflops.cpp contains the code described in the code listing 4.1, and gettime.cpp contains calls to gettimeofday function to get elapsed time in seconds as shown in the following code segment:

```
#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));
}
```

3. The –mmicswitch dictates the compiler to generate cross compiled code for Intel Xeon Phi coprocessor.
4. -opt-threads-per-core=2 switch allows the compiler code generator to schedule code generation assuming 2 threads are running in each core.
5. –no-vec switch asks compiler not to vectorize the code even if it can be.
6. –openmp switch allows the compiler to understand the OpenMp pragmas during the compile time and link in appropriate OpenMp libraries.
7. –vec-report3 tells the compiler to print out details information about the vectorization being performed on the code as it is being compiled.

In order to see the effect of vectorization, let's first compile the code with –no-vec switch and run it on a single core by setting OMP_NUM_THREADS=1.
Once the code is compiled, copy the file produces, dpflops.out to mic card by using scp command as follows:

```
command_prompt-host >scp ./dpflops.out mic0:/tmp
dpflops.out                   100%   19KB  18.6KB/s   00:00
command_prompt-host >
```

This will upload the ./dpflops.out to the Intel Xeon Phi card and place it on the /tmp directory on the ramdisk of the card. The operating system on the Intel Xeon Phi usually allocates some portion of the GDDR memory to be used as a ramdisk and hosts the file system on the card.
You will also need to upload the dependent openmp library, libiomp5.so to the card by following command:

```
command_prompt-host >scp /opt/intel/composerxe/lib/mic/libiomp5.so mic0:/tmp
```

The libiomp5.so is the dynamic library necessary for running OpenMP programs compiled with Intel® compilers. The Intel Xeon Phi version of the library is available at the location "/opt/intel/composerxe/lib/mic/" provided you installed the compiler at the default install path on the system you are building the application.
Once the card is uploaded with the file, you can log onto the card using the command:

```
command_prompt-host>ssh mic0
```

To run the code you now need to setup the environment as shown in the screen capture below in Figure 3. The first thing you need to do is set the LD_LIBRARY_PATH to /tmp to be able to find the runtime openmp library loaded to /tmp directory on the Xeon Phi card. We would now set the number of threads to 1 by setting the environment variable OMP_NUM_THREADS=1. Also set KMP_AFFINITY=compact, so that OpenMP threads for thread id 0-3 are tied to core 1 and so on.
The figure shows that the single threaded non vectorized run only provided ~0.66 Gflops. However, expected DP flops for a single core run is 2(FMA)x1(DP elements-non vectorized)x1.1 GHz = 2.2 Gflops. However, as described in architecture section, the hardware cannot issue instructions back to back from the same thread in the core. To reach full execution unit utilization at least two threads must be running at all times. So running on 2 threads (OMP_NUM_THREADS=2), we are able to reach 1.45 Gflops per core. As the core still uses vector unit to perform scalar arithmetic, the code for scalar arithmetic is very inefficient. For each fma on a DP element, it has to broadcast the

element to all the lanes. Operate on the vector register with a mask and then store the single element back to memory. We also notice that increasing threads per core does not improve performance as the instruction can be issued every cycle for this case. Now if we extend to 240 threads utilizing all 60 cores we can achieve 86 Gflops as shown in the figure. Now let's see whether we can get close to designed 1 TeraFlops for Intel Xeon Phi by turning on vectorization.

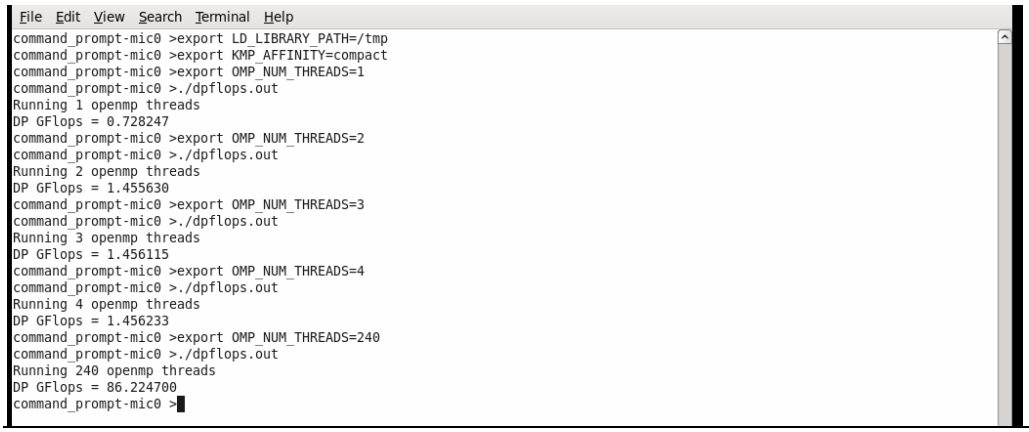

Figure 3: Executing non-vectorized code on Intel® Xeo Phi™

Now, let's turn on the vectorization by removing the –no-vec switch. If we recompile the code with following command line:

```
icpc -O3 -opt-threads-per-core=2 -mmic -openmp -vec-report3 dpflops.cpp gettime.cpp -o
dpflops.out
```

This will print out following info:

```
command_prompt-host >./buildmic.sh
dpflops.cpp(58): (col. 29) remark: loop was not vectorized: statement cannot be vectorized.
dpflops.cpp(65): (col. 16) remark: LOOP WAS VECTORIZED.
dpflops.cpp(63): (col. 9) remark: loop was not vectorized: not inner loop.
dpflops.cpp(73): (col. 16) remark: LOOP WAS VECTORIZED.
dpflops.cpp(71): (col. 4) remark: loop was not vectorized: not inner loop.
```

We will get vectorized version of the code. In above generated compiler report you can see that line 65 and line 73 of the code listing 4.1 has been vectorized by the compiler. Let's upload the binaries to Intel Xeon Phi by scp command as before and run it under variousconditions. With an FMA and double precision arithmetic which can work on 8 DP elements at a time, we should see around 2(for Fused Multiply and Add (FMA))x8 (DP elements)x1.1 GHz = 17.6 Gflops per core.

Figure 4 shows the results of experimentation with the vectorized double precision code. With 1 thread, we are able to 8.7 Gflops, not 17.6 as expected. As described above this is due to the issue BW is not fully utilized thus execution unit is not utilized each cycle. In order to achieve so we need to use at least two threads. Thus setting OMP_NUM_THREADS=2 with KMP_AFFINITY=compact, we made sure the code was running on the same core, but two threads were executing and scheduling instructions every other cycle. This kept execution unit fully utilized and was able to achieve near peak performance of 17.5 Gflops per core on the double precision fused multiply add arithmetic. We also observe in the figure that increasing number of threads to 3 and 4 did not improve performance since we already saturated the double precision execution unit on the core. Utilizing all 60 cores, with 2 threads each we can see that we could achieve 1022 Gflops which is near the theoretical peak of 1055 Gflops on this coprocessor. You can also see that I have set KMP_AFFINITY=balanced to make sure the threads are affinitized to different cores to distribute computations evenly when the number of threads is less than the maximum number of threads needed to saturate the cores, in this case 240 threads. Since we are running 120 threads with 2 threads per core on 60 cores, having affinity set to balanced, there will be two threads per core distributed across cores.

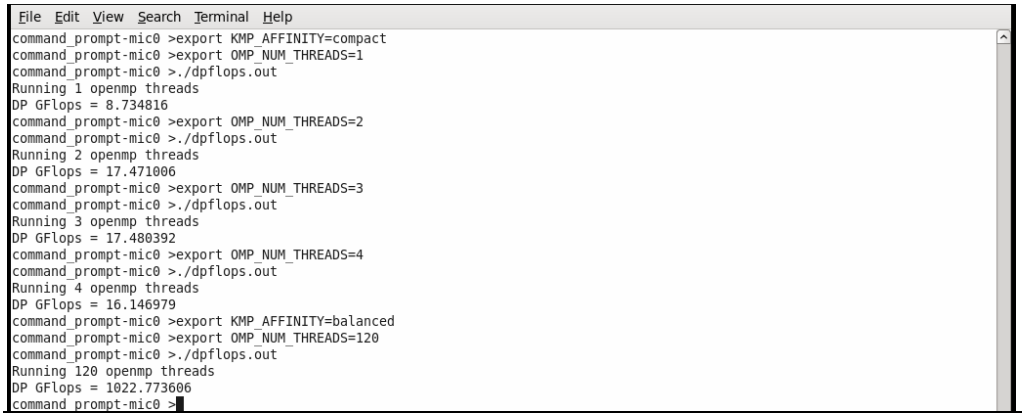

Figure 4: Execution of Vectorized Double Precision Code on Intel Xeon Phi

Next turn our attention to single precision arithmetic. We would expect the performance to be 2x the single precision. The changes to the code segment in code listing 1 will be to change double to float. However, we need to keep the float calculation same as the number of operations are the same. However, since the vector unit can work on 16 elements at a time with the float, in order to be able to fill up the dispatch queue for each thread, we would need twice as many elements as double, so I have adjusted the size to 32 elements instead of 16 elements as shown below.

For single precision arithmetic, we did the same experimentation as before and the output is captured below in Figure 5. One can observe that the single precision behavior is the same for single core.

Code Listing 2

```
38 #include <stdio.h>
 39 #include <stdlib.h>
 40 #include <omp.h>
 41
 42 unsigned int const SIZE=32;
 43 unsigned int const ITER=48000000;
 44
 45 extern double elapsedTime (void);
 46
 47 int main()
 48 {
 49       double startTime,  duration;
 50       int i;
 51
 52       __declspec(aligned(64)) float a[SIZE],b[SIZE],c[SIZE];
 53
 54
 55       //intialize
 56       for (i=0; i<SIZE;i++)
 57       {
 58            c[i]=b[i]=a[i]=(double)rand();
 59       }
 60
 61 //warm up cache
 62       #pragma omp parallel for
 63       for(i=0; i<ITER;i++) {
 64       #pragma vector aligned (a,b,c)
 65        a[0:SIZE]=b[0:SIZE]*c[0:SIZE]+a[0:SIZE];
 66       }
 67
 68       startTime = elapsedTime();
 69
 70       #pragma omp parallel for
 71       for(i=0; i<ITER;i++) {
 72       #pragma vector aligned (a,b,c)
 73        a[0:SIZE]=b[0:SIZE]*c[0:SIZE]+a[0:SIZE];
 74       }
 75
 76       duration = elapsedTime() - startTime;
 77
 78       double Gflop = 2*SIZE*ITER/1e+9;
 79       double Gflops = Gflop/duration;
 80
81       printf("Running %d openmp threads\n",
omp_get_max_threads());
 82       printf("SP GFlops = %f\n", Gflops);
 83
 84   return 0;
 85
 86 }
```

execution where it can execute optimally with 2 threads per core for utilizing execution bandwidth properly which it cannot do with single thread. However, adding more threads to core on this core bound code does not help. Binding the threads to a core was enforced by the KMP_AFFINITY switch. After changing the affinity to balanced, we allowed the

threads to spread out to various cores first to make use of independent execution units. Making the number of threads 120 (2xNumber of cores for this case), we were able to get the performance to 1979 GFlops.

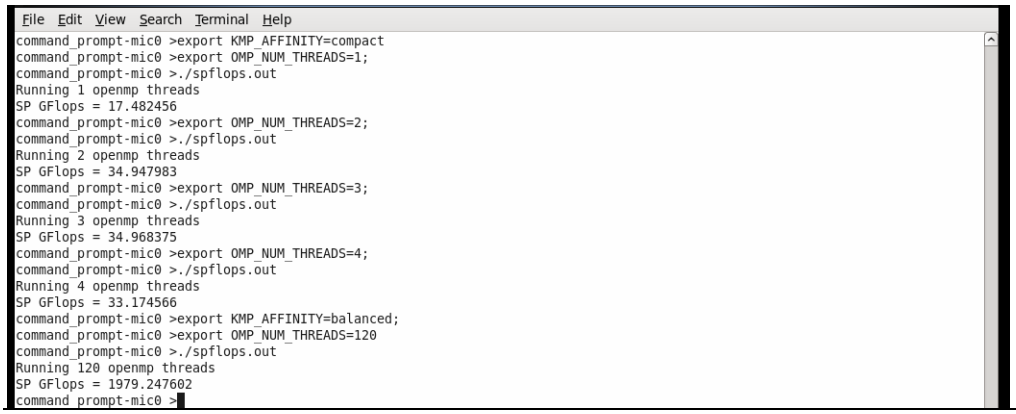

Figure 5: Execution of Vectorized Single Precision Vector Code on Intel Xeon Phi

Understanding Intel Xeon Phi Cache Performance

In order to understand the cache latencies, I shall be using the publicly available memory latency benchmark component of lmbench benchmark available at http://www.bitmover.com/lmbench/.
The lmbench was developed by the authors to make them portable and wholly written in C. That was the reason I picked the benchmark to explore the memory hierarchy of Intel Xeon Phi architecture. One of the benchmark components that I was interested in was lat_mem_rd that finds out memory read latencies at various memory hierarchy.
In order to build this software, I downloaded the bits from benchmark download page. I created a copy of the Makefile in src directory to Makefile.mic then modified src/Makefile.mic compiler options to add –mmic switch as shown below so that it could be cross compiled for Intel® Xeon Phi™.
@env CFLAGS="-Wall -ansi -mmic" MAKE="$(MAKE)" MAKEFLAGS="$(MAKEFLAGS)" CC="${CC}" OS="${OS}"
../scripts/build all opt

Next I set CC to point to icc or icpc, the Intel® compiler to build the suite. I modified scripts/compiler file as follows to point to icc.

```
/bin/sh

if [ "X$CC" != "X" ] && echo "$CC" | grep -q '`'
then
   CC=
fi

if [ X$CC = X ]
then   CC=icc
     for p in `echo $PATH | sed 's/:/ /g'`
     do     if [ -f $p/gcc ]
            then   CC=icc
            fi
     done
fi
echo $CC
```

Once the Make file and compiler is set properly I did a “command_prompt> make –f Makefile.mic” to build the benchmark binaries which in my case ended up in the folder “bin/x86_64-linux-gnu”. Next step was to copy benchmark binary I am interested in “lat_meme_rd” to the /tmp directory on the coprocessor micro os using scp command.
Once the benchmark binary was transferred to the card I ran the latency benchmark as:

```
command_prompt-mic0 >./lat_mem_rd -P 1  -N 10 32 64
```


The benchmark ran with 32 MB of memory that fill the L1 cache, in single threaded mode with 64 byte stride so that I am reading data from different cache line in subsequent calls. The output is captured in the screenshot in Figure 6 below. It can be seen that for sizes up to 32 KB which is the L1-Data cache size for each core, the cache latency is about 2.8 ns which is approximately 3 cycles on this coprocessor. Moving beyond L1-D cache sizes, we see that the latency goes up to 13.27 ns (~14.5 cycles).

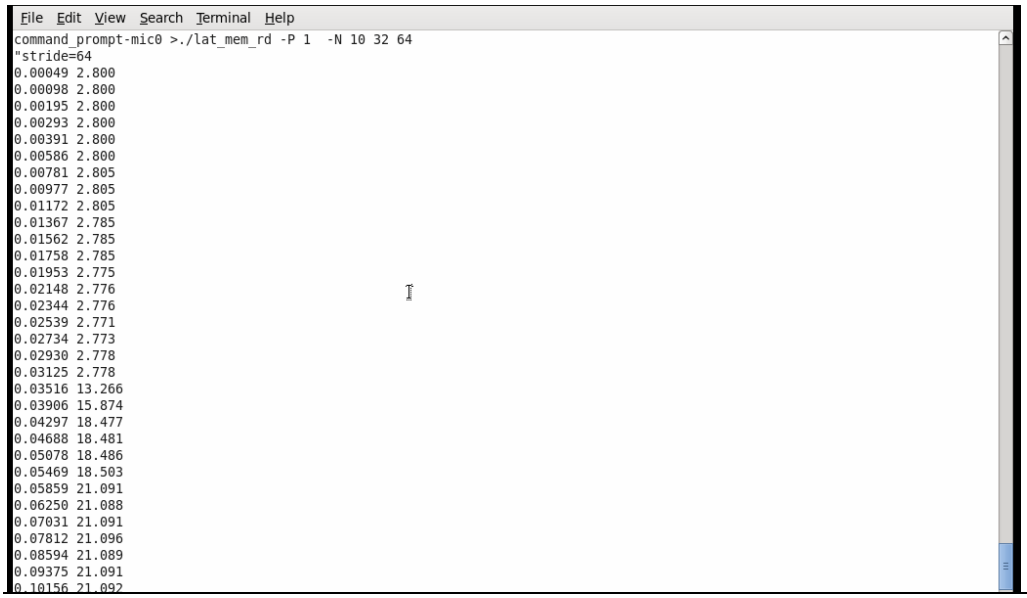

Figure 6: lmbenchmark lat_mem_rd running on Intel Xeon Phi

==========

This article is based on material found in the book Intel® Xeon Phi™ Coprocesser Micro Architecture and Tools. Visit the Intel Press web site to learn more about this book: http://noggin.intel.com/intelpress/categories/books/intel%C2%AE-xeon-phi%E2%84%A2-coprocesser-micro-architecture-and-tools

Also see our Recommended Reading List for related topics: www.intel.com/technology/rr

About the Author

Reza Rahman is a Sr. Staff Engineer at Intel Software and Services Group. Reza lead the worldwide technical enabling team for Intel Xeon Phi™ product through Intel software engineering team. He played a key role during the inception of Intel MIC product line by presenting value of such architecture for technical computing and leading a team of software engineers to work with 100s of customers outside Intel Corporation to optimize code on Intel® Xeon Phi™. He worked internally with hardware architects and Intel compiler and tools team to optimize and add features to improve performance of Intel MIC software and hardware components to meet the need of technical computing customers. He has been with Intel for 19 years. During his first seven years he was at Intel Labs working on developing audio/video technologies and device drivers. Rest of the time at Intel he has been involved in optimizing technical computing applications as part of software enabling team. He is also believes in standardization process allowing software and hardware solutions to interoperate and has been involved in various industry standardization group like World Wide Web consortium (W3C).

Reza holds a Masters in computer Science from Texas A&M university and Bachelors in Electrical Engineering from Bangladesh University of engineering and Technology.

==========

