

CHAPTER 6

Xeon Phi PCIe Bus Data Transfer and Power Management

This chapter looks at how the coprocessor is configured in a Xeon-based server platform and communicates with the host. It will also look at the power management capabilities built into the coprocessor to help reduce power consumption while idle. Figure 6-1 shows a system with multiple Intel Xeon Phi and two socket Intel Xeon processors. The coprocessor connects to the host using PCI Express 2.0 interface x16 lanes. Data transfer between the host memory and the GDDR memory can be through programmed I/O or through *direct memory access* (DMA) transfer. In order to optimize the data transfer bandwidth for large buffers, one needs to use the DMA transfer mechanism. This section will explain how to use high-level language features to allow DMA transfer. The hardware also allows peer-to-peer data transfers between two Intel Xeon Phi cards. Various data transfer scenarios are shown in Figure 6-1. The two Xeon Phi coprocessors A and B in the figure connect to the PCIe channels attached to the same socket and can do a local peer-to-peer data transfer. The data transfer between Xeon Phi coprocessors B and C will be a remote data transfer. These configurations play a key role in determining how the cards need to be set up for optimal performance.

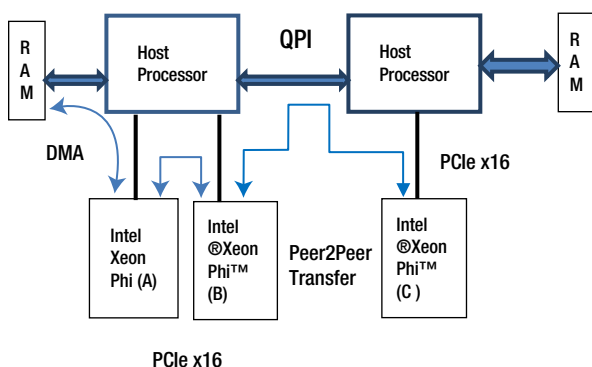


Figure 6-1. Intel Xeon Phi-based system configuration

Figure 6-2 shows the various important components of a card that allow it to operate as a coprocessor in conjunction with host processors and other PCIe devices and coprocessor in the system.

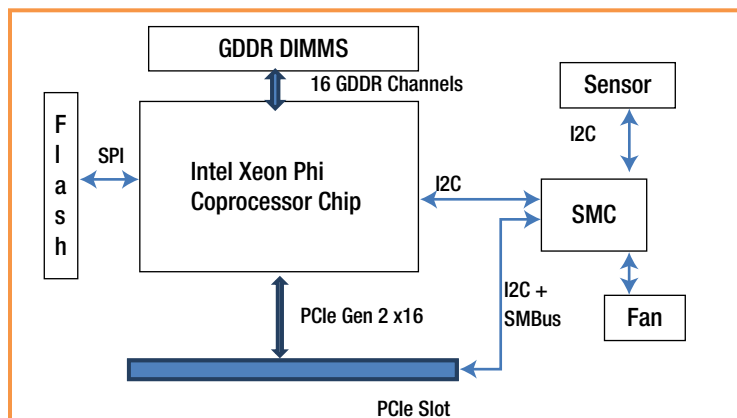


Figure 6-2. Components of an Intel Xeon Phi card

As seen in Figure 6-2, the card contains the Intel Xeon Phi coprocessor and is connected to the onboard DDR5 DIMMS with up to 16 channels distributed on both sides of the *printed circuit board* (PCB). Each memory channel supports two 16-bit-wide GDDR devices. The board also contains a *system management controller* (SMC in the diagram), thermal sensors, and a fan in actively cooled SKUs of the product. There are certain SKUs of Intel Xeon Phi which are passively cooled and do not require the cooling fan. The clock system uses a PCI Express 100 MHz reference clock and includes onboard 100 MHz \pm 50 ppm reference. The card contains an onboard flash allowing it to load a coprocessor OS on boot. You can find full physical and electrical specifications in the Intel Xeon Phi coprocessor data sheet.¹

The SMC shown in Figure 6-2 has three I2C interfaces. One of them connects directly to the coprocessor chip to collect coprocessor thermal and status data, the second one connects to on-card sensors, and the third one is the *system management bus* (SMBus) for system fan control for cards with passive heat sink. The SMBus is also used for integration with the node management controller using the *Intelligent Platform Management Bus* (IPMB) protocol.

The Intel Xeon Phi coprocessor uses a system interface functional unit to communicate with the host over a PCIe gen 2 interface and uses x16 lines for maximum transfer bandwidth. It is shown to achieve greater than 6 GB/s for both host-to-device and device-to-host data transfers. The unit also supports remote DMA to host to optimize data transfers. The PCIe clock mode supports both PCIe 1 and 2 and uses an external buffer to support external PCIe 100 MHz reference clocks. The PCIe conforms to the PCIe gen 2 standard and supports 64- to 256-byte packets, peer-to-peer read/writes. The peer-to-peer interface allows two Intel Xeon Phi cards to communicate with each other without host processor intervention. This capability is particularly useful when running MPI programs across multiple Intel Xeon Phi cards (see the section in this chapter, “Placement of PCIe Cards for Optimal Data Transfer BW”).

The system interface consists of the coprocessor *system interface unit* (SIU) and the *transaction control unit* (TCU). The PCIe protocol engine is the part of the SIU of Intel Xeon Phi architecture that implements the PCIe logic. The system interface component also includes the *serial peripheral interface* (SPI), which allows one to load the flash and coprocessor OS. In addition, the SIU includes the I2C logic components necessary to control fan speed on the card and *advanced programmable interrupt controller* (APIC) logic to allow communication between the host and the card.

The TCU connects the system bus to the internal ring bus of the Intel Xeon Phi coprocessor core. The TCU implements the DMA engine and buffers to control the flow of data to and from the system bus to the ring interconnect encryption/decryption engine, memory mapped I/O (MMIO) registers, and flow-control logic and instructions.

¹Intel Xeon Phi Coprocessor Data Sheet, Reference Number: 328209-001EN. www.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-datasheet.html.

DMA Engine

The following data transfer scenarios are supported in a system with one or more Intel Xeon Phi cards:

1. Peer-to-peer communication between coprocessor GDDR5 apertures
2. Intel Xeon Phi coprocessor to host system memory (device-to-host)
3. Host system memory to Intel Xeon Phi coprocessor GDDR5 (aperture or DMA)
4. Intra-GDDR5 block transfers within Intel Xeon Phi coprocessors

DMA allows data transfer to happen between the host memory and the coprocessor memory without host CPU intervention. The DMA transfer is done by programming the DMA controller with the source and destination addresses in the *channel descriptor ring*. The descriptor contains the length of data in cache line sizes in addition to the source and destination addresses. One can use the DMA controller on the card or host to do the transfer either way.

A descriptor ring is a circular buffer with up to 128K entries and aligns to the cache line boundary. The software manages the ring by moving a head pointer to the circular buffer after it fills out a descriptor entry. The head pointer is copied to the DMA controller head pointer register for the appropriate DMA channel. The DMA channel contains a tail pointer which points to the same location as the head pointer upon initialization and is updated as descriptors are fetched by the DMA controller into a local descriptor queue inside the controller for each channel. The local descriptor queue has 64-bit entries and maps to a sliding window in the system descriptor ring. The tail pointer is periodically written to system memory to update the DMA transfer status. The head and tail pointers are 40-bit wide and the *most significant bit* (MSB) of the pointer indicates the location of the descriptor ring. If the high order bit is 1, the descriptor ring resides in the system memory; otherwise it resides in the Intel Xeon Phi coprocessor memory.²

The DMA descriptor rings can be programmed by the host driver or the coprocessor OS. Up to eight DMA channels, each corresponding to one circular DMA ring of descriptors, can be written by the driver or coprocessor OS and operate in parallel. The descriptor rings written by the host driver must be present in the host system memory, and those written by coprocessor OS must reside in the on-card GDDR5 memory.

The DMA can be host- or device-initiated and supports data transfer in both directions, from host to device and vice versa. The DMA transfer happens using the physical addresses and generates an interrupt after the transfer is done.

The DMA transfer happens at core clock rate, and the eight independent channels can move data in parallel from the host system memory to the Intel Xeon Phi GDDR5 and from the Intel Xeon Phi GDDR5 memory to the host system memory.

Intel Xeon Phi supports from 64 up to 256 bytes per DMA-based PCIe transaction. The transaction size is programmable by PCI commands.

The ring descriptors in a channel are operated on sequentially. The multiple DMA channels can, however, be opened to provide arbitration capabilities between the channels if needed.

Measuring the Data Transfer Bandwidth over the PCIe Bus

This section will walk you through actual data transfer using the offload programming language. In Code Listing 6-1, I show how you can use the offload programming language to set up a high-level data transfer between hosts to Intel Xeon Phi. Although there are lower level APIs exposed by Intel Xeon Phi programming environments available through Intel Xeon Phi system programming software, such as the *symmetric communication interface* (SCIF), I will focus on high-level offload language extensions for C to do the data transfer. The Intel compiler uses appropriate lower-level system interfaces for such tasks as setting up the DMA channel for proper transfer and so forth. You will see, however, there are some requirements for optimizing data transfer performance.

²For details of the descriptor rings format, see *Intel Xeon Phi Coprocessor System Software Developers Guide*. IBL Doc. ID: 488596.

Code Listing 6-1. Example of Host to Device Data Transfer Over the PCIe Bus

```

38 //Define number of floats for 64 MB data transfer
39 #define SIZE (64*1000*1000/sizeof(float))
40 #define ITER 10
41 // set cache line size alignment
42 #define ALIGN (64)
43 __declspec(target(MIC)) static float *a;
44 extern double elapsedTime (void);
45 int main()
46 {
47     double startTime, duration;
48     int i, j;
49
50     //allocate a
51     a = (float*)_mm_malloc(SIZE*sizeof(float),ALIGN);
52
53     //initialize arrays
54     #pragma omp parallel for
55     for (i=0; i<SIZE;i++)
56     {
57         a[i]=(float)1.0f;
58     }
59     // Allocate memory on the card
60     #pragma offload_transfer target(mic) \
61     in(a:length(SIZE) free_if(0) alloc_if(1) align(ALIGN) )
62
63
64
65     startTime = elapsedTime();
66     for(i=0; i<ITER;i++) {
67         //transfer data over the PCI express bus
68         #pragma offload_transfer target(mic) \
69         in(a:length(SIZE) free_if(0) alloc_if(0) align(ALIGN) )
70
71     }
72     duration = elapsedTime() - startTime; ;
73     // free memory on the card
74     #pragma offload_transfer target(mic) \
75     in(a:length(SIZE) alloc_if(0) free_if(1) )
76
77
78
79
80 //free the host system memory
81     _mm_free(a);
82     double GB = SIZE*sizeof(float)/(1000.0*1000.0*1000.0);
83     double GBps = ITER*GB/duration;
84     printf("SP ArraySize = %0.4lf MB, ALIGN=%dB, PCIe Data transfer bandwidth Host->Device
85 GB/s = %0.2lf\n", GB*1000.0, ALIGN, GBps);
86
87     return 0;
88 }

```

Recall that the maximum data transfer performance can be achieved by using the DMA engine. Since DMA engines transfer data in 64 bytes (cache line size), you need to make sure the data length is a 64-byte multiple. At Line 39, I choose the data array size to transfer as 64MB. I chose a large size to amortize the DMA transfer overhead needed to set up the DMA engine with proper parameters. The second important datum of information needed to optimize the transfer is to make sure that the data to be transferred are cacheline-aligned to enable transfer of the data with the DMA engine. I used the 'C' macro '#define ALIGN' in Line 42 to make sure the allocated memory is 64-byte aligned. In Line 43, I define a pointer 'a' that will be visible to offloaded code executing on the card by declaring it with "`__declspec(target(MIC))`."

This pointer will be allocated on the host and transferred over to the coprocessor using offload pragmas. In order to do that, you need to allocate space for the buffer using `_mm_malloc` intrinsics supported by the Intel compiler in Line 51. The `_mm_malloc` intrinsic, in addition to behaving like a `malloc` function, also allows data alignment to be specified as part of the request. This line allocates 64MB of data area aligned to the cache line boundary (64 bytes).

The first data transfer happens in Line 60, as shown below:

```
60     #pragma offload target(mic) \
61     in(a:length(SIZE) alloc_if(1) free_if(0) align(ALIGN) )
62     {
63     }
```

The data transfer begins by a call "`#pragma offload target(mic)`" that tells the runtime library to start a process started on coprocessor. The 'in' parameter tells it to send the array pointed to by '*a' to the coprocessor, the `alloc_if(1)` clause in the offload statement allocates memory on the card for pointer 'a' and copies the data over to coprocessor. Since the size of the buffer and alignment is large enough, the underlying libraries will use the DMA engine to set up and transfer these data to the coprocessor. The `free_if(0)` clause tells the runtime not to free the buffer that was allocated on the card so it could be reused. The '`align(ALIGN)`' clause in the same statement tells the buffer that is allocated on the card to be aligned to ALIGN bytes, in this case 64 bytes.

I have separated out the first offload pragma call as it involves start-up overhead that I do not want to count as part of the transfer time. The overhead involves sending the coprocessor part of the binary to the card and sending necessary runtime compiler library like `openmp` library to the card as well. You will also need to allocate space for array 'a' on the card to copy the data in, which happens during this first offload statement.

Lines 65 through 72 are the timing loop where the actual data transfer rate is measured. Here the data transfer is repeated ITER (10) times to average out the any run to run variation in transfer time. The '`pragma offload`' statement is similar to what was discussed earlier to show the initialization of coprocessor code and data:

```
68     #pragma offload target(mic) \
69     in(a:length(SIZE) free_if(0) alloc_if(0) align(ALIGN) )
70     {}
```

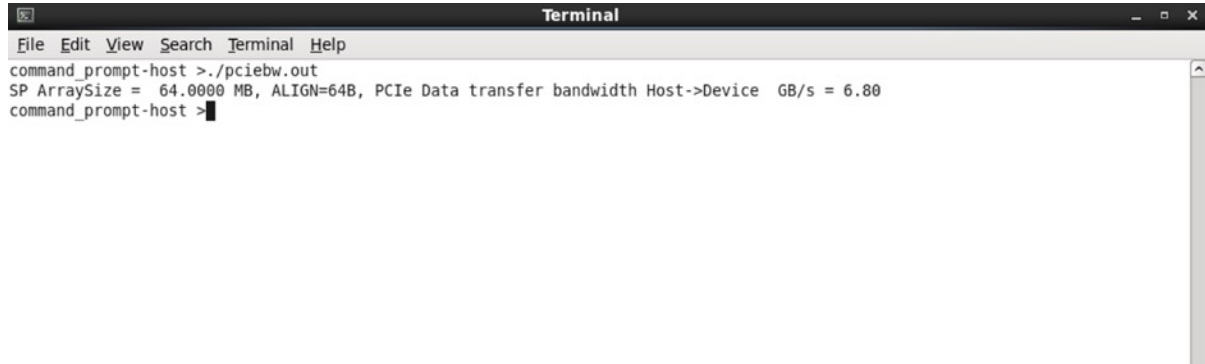
The main thing to keep in mind is that in this statement we have both the '`free_if(0)`' and '`alloc_if(0)`' clause set to zero, thus telling the runtime not to allocate any more space for 'a' and also not to free the space created previously with pragma offload code. Note that if you made the previous statement to free the buffer allocated in the first call by calling with `free_if(1)`, the buffer would have been freed after the '`pragma offload`' statement.

In that case you would need to reallocate the buffer with the `alloc_if(1)` call instead of `alloc_if(0)` call. It is a common source of error in offload programming if the allocation and free clauses do not properly match up. If you have more '`alloc_if(1)`' calls without matching `free_if(1)` calls, you will see memory leaks on the coprocessor card. Similarly, if you try to copy the input buffer with the `alloc_if(0)` call assuming the buffer is present, but where it was accidentally freed in the previous call, you will get a general protection fault.

In Line 75, I am done with the data transfer measurements and have freed the memory allocated on the card by using the `free_if(1)` call with a matching `alloc_if(0)` call so that I do not need to allocate the buffer on entering the offload region.

I also need to free the memory allocated on the host using `_mm_malloc` by matching `_mm_free` intrinsics. You need to make sure you use `_mm_free` instead of `free` calls since they use a different heap manager and a different space for buffer management. Figure 6-3a shows the output of “`pciebw.out`” application built by the following command with Intel compiler:

```
Command_prompt-host > icpc -O3 -openmp -vec-report3 pciebw.cpp gettimeofday.cpp -o pciebw.out
```



```
Terminal
File Edit View Search Terminal Help
command_prompt-host > ./pciebw.out
SP ArraySize = 64.0000 MB, ALIGN=64B, PCIe Data transfer bandwidth Host->Device GB/s = 6.80
command_prompt-host >
```


Figure 6-3a. Output of `pciebw` test run

The Intel compiler recognizes offload pragmas and generates the code necessary for data transfer to the coprocessor.

The output captured on Figure 6-3a shows that the data transfer of 64 MB buffer size with 64-byte aligned achieved an approximately 6.8 GB/s transfer rate over the PCIe bus in moving from the host to the coprocessor (device). You can also see the effect of various buffer sizes and alignment if you play around with the alignment and data buffer size. To see the effect of various buffer sizes, let’s modify the code to run with 64 kB instead of 64 MB. This can be done by changing the buffer size in Line 39 to:

```
39 #define SIZE (64*1000/sizeof(float))
```

I recompiled and ran the same code with the modified buffer size and the output is shown in Figure 6-3b. As you can see, the BW drops to 3.88 GB/s compared to 6.8 GB/s for the 64 MB buffer size. This is due to DMA transfer overhead, because smaller data transfer sizes cut down on overall performance.



```
Terminal
File Edit View Search Terminal Help
command_prompt-host > ./pciebw.out
SP ArraySize = 0.0640 MB, ALIGN=64B, PCIe Data transfer bandwidth Host->Device GB/s = 3.88
command_prompt-host >
```

Figure 6-3b. Output of the `pciebw` test with 64K data buffer

Now, to see what the effect of alignment is, let’s change the buffer alignment from 64 bytes to 6 bytes of unaligned data. As you can see from Figure 6-3c, the BW drops even further to 2.29 GB/s.



```

Terminal
File Edit View Search Terminal Help
command_prompt-host > ./pciebw.out
SP ArraySize = 0.0640 MB, ALIGN=6B, PCIe Data transfer bandwidth Host->Device GB/s = 2.29
command_prompt-host >

```

Figure 6-3c. Output of the *pciebw* test with 64K data buffer with 6B alignment

It has been found that for smaller data sizes (< 4 KB), the data transfer may be faster when it is done with MMIO through CPU write to card memory than through DMA transfer. The compiler may choose accordingly to use different data transfer methods depending on the data buffer size for optimal data transfer bandwidth. You could run the same example with `OFFLOAD_REPORT=1`. This will print out the CPU time taken to transfer the data and should correlate to the performance number for various alignment cases discussed in this section.

Reading Data from the Coprocessor

The previous section examined the bandwidth to transfer data from the host to the coprocessor. The data read back from the coprocessor to the host can be tested with very similar code. The main difference is that reading the data from the coprocessor involves using the ‘out’ clause instead of the ‘in’ clause in Line 69 of Code Listing 6-1. The code adjusted for device-to-host data transfer is shown in Listing 6-2.

Code Listing 6-2. Example of Device-to-Host Data Transfer Over the PCIe Bus

```

38 //Define number of floats for 64 MB data transfer
39 #define SIZE (64*1000*1000/sizeof(float))
40 #define ITER 10
41 // set cache line size alignment
42 #define ALIGN (64)
43 __declspec(target(MIC)) static float *a;
44 extern double elapsedTime (void);
45 int main()
46 {
47     double startTime, duration;
48     int i, j;
49
50     //allocate a
51     a = (float*)_mm_malloc(SIZE*sizeof(float),ALIGN);
52
53     //initialize arrays
54     #pragma omp parallel for
55     for (i=0; i<SIZE;i++)
56     {
57         a[i]=(float)1.0f;
58     }
59     // Allocate memory on the card
60     #pragma offload transfer target(mic) \
61     in(a:length(SIZE) free_if(0) alloc_if(1) align(ALIGN) )


```

```

62
63
64     startTime = elapsedTime();
65     for(i=0; i<ITER;i++) {
66         //transfer data over the PCI express bus
67         #pragma offload_transfer target(mic) \
68             out(a:length(SIZE) free_if(0) alloc_if(0) align(ALIGN) )
69     }
70
71     }
72     duration = elapsedTime() - startTime;
73     // free memory on the card
74     #pragma offload_transfer target(mic) \
75         in(a:length(SIZE) alloc_if(0) free_if(1) )
76
77
78
79 //free the host system memory
80     _mm_free(a);
81
82 }

```

The device-to-host code can be built using the same command line, and the output is shown in Figure 6-4. It can be seen from the output that the transfer bandwidth for read back is approximately 6.89 GB/s, which is slightly faster than the host-to-device transfer speed.



```

Terminal
File Edit View Search Terminal Help
command_prompt-host >./pciebw-d2h.out
SP ArraySize = 64.0000 MB, ALIGN=64B, PCIe Data transfer bandwidth Coprocessor->Host GB/s = 6.89
command_prompt-host >

```

Figure 6-4. Output of the PCIe BW-d2h test with 64MB data read with 64B alignment

Low-Level Data Transfer APIs for Intel Xeon Phi

The Intel Xeon Phi MPSS software stack contains two lower-level APIs for more control over data allocation and transfer in case you need it. In general, developers are encouraged to use high-level programming constructs in Fortran and C/C++ to do the transfer. If the need arises, however, to have more control over data transfer, you can use either of two lower-level APIs: *Common Offload Infrastructure* (COI) and *Symmetric Communication Interface* (SCIF) (Figure 6-5). Although COI is an interesting interface, it is mainly used and exposed by the offload programming model and will not be discussed here. SCIF provides a low-latency communication channel between different SCIF clients, which can be any device including the host. SCIF provides the communication backbone between the host processor and the Xeon Phi coprocessors and between multiple Xeon Phi coprocessors. Other communication APIs such as COI, Virtual IP, and MPI can be built on top of the SCIF API.

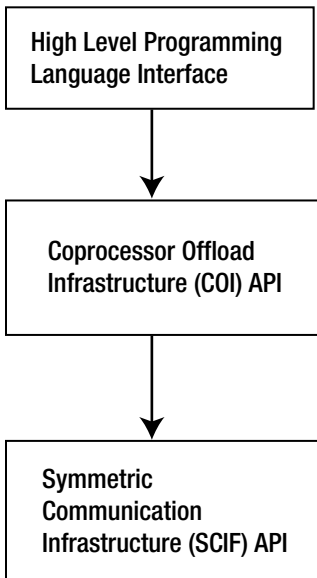


Figure 6-5. SCIF/COI API

The SCIF API exposes DMA capabilities for high-bandwidth data transfer. It also provides MMIO by mapping the host or coprocessor system memory address into the address space of the processes running on the host or coprocessor. SCIF uses the direct peer-to-peer memory access model for communicating with each of its nodes. So if two nodes are on two separate Xeon Phi coprocessors, the nodes can communicate with each other directly without going through the host memory.³ SCIF does not provide any reliability features for the transmitted data but depends on the underlying PCIe reliability feature, thus making the software layer low-overhead.

As its name implies, SCIF is designed to provide you a symmetrical view, whether you are running on the host or on the coprocessor. With multiple Xeon Phi coprocessors in a system, SCIF treats each card and host as communication end points and supports an arbitrary number of nodes. The SCIF code is optimized for up to eight Xeon Phi devices. All of the coprocessor memory is visible to the host and other coprocessors; conversely, the host memory is visible to the coprocessors.

The SCIF implementations contain two components: one running in user mode (ring 3) and the other in kernel mode (ring 0). There are five categories of APIs provided by SCIF for lightweight communication purposes:

- *Connection API*: Establish connections among various SCIF nodes, following the socket programming paradigm.
- *Messaging API*: Support two-sided communications between SCIF nodes, intended for short-latency sensitive messages.
- *Registration API*: Allow mapping addresses for one node into the process space of the other node between which connections are established by the connection API.
- *RMA API*: Support communication using the address space mapped through registration APIs. This allows DMA and programmed I/O transfers to the mapped memory space and relevant synchronization APIs.
- *Utility API*: Provide the utility services necessary to perform above functionalities.

³Intel Xeon Phi Coprocessor System Software Developers Guide. SKU 328207-001EN. www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-phi-software-developers-guide.pdf.

Placement of PCIe Cards for Optimal Data Transfer BW

In a multisocket platform, there are multiple PCIe slots near the sockets, as shown in Figure 6-1. Since the Xeon Phi card is connected to the PCIe slots and can coexist with other Xeon Phi or networking cards such as Infiniband cards on the same node, you need to carefully consider placement of the coprocessor cards in the available PCIe slots for optimal performance.

For optimal communication bandwidth between the cards, you can create the following configurations:

1. For card-to-card data transfers, such as MPI ranks communicating between tasks running on two cards, it is better to have them connected to the PCIe lanes that are connected to the same socket. This configuration—known as *local peer-to-peer configuration*—will provide optimal bandwidth for data transfer between the cards. If the cards are placed on the PCIe bus corresponding to different sockets—known as *remote peer-to-peer configuration*—performance will suffer.
2. The card-to-card data transfer case is also true for Infiniband or other networking cards used in PCIe configurations. Always try to gravitate toward local communication between the cards or processor and the card for optimal data transfer bandwidth. For example, if you are setting up a cluster with Infiniband cards, it may be useful to pin your process to a socket which is local to the Infiniband card used for communication.
3. For configurations of one or more Intel Xeon Phi cards and Infiniband cards, putting them in *local P2P configuration* (i.e., connected to the same socket) will provide optimal performance.

Power Management and Reliability

According to top500.org, as of June 2013, the fastest supercomputer system was the Tianhe-2 with an Rpeak (theoretical peak performance of the system) of 54.9024 petaflops. According to computational power growth over time as recorded by top500.org,⁴ if the trend continues, the top performing supercomputer will attain 1 exaflops around 2018, allowing us to perform technical computing to solve problems not yet attempted. Projecting the current rate of power consumption by the top 500 supercomputers, 100 MW—approximately the output of a small nuclear power station—will be required to power a 1 exaflops supercomputer, which is impractical. The Department of Energy projects that the power consumption needs to be cut to 20 MW or less to make an exaflops supercomputer practical.⁵

It was with this target in mind that Intel MIC architects designed the power management and reliability features in the Intel Xeon Phi implementation. The design philosophy has been that maximum power efficiency can be achieved when the software running on the machine is able to dictate to the hardware how to manage the power states the hardware supports. Accordingly, the Intel Xeon Phi is built so that the coprocessor OS/driver layer (the MPSS layer) manages the runtime power requirements of the hardware.

There are several power management states implemented by the Intel Xeon Phi coprocessor and controlled by system and application software: the Turbo mode, package P-states, core C-states, and memory M-states. In Turbo mode, the core can make use of power/thermal headroom to increase frequency and voltage depending on the number of active cores. There is a utility shipped with Intel Xeon Phi driver package that enables turbo on/off states of the coprocessor.

⁴Exponential growth of supercomputing power as recoded by top500.org. <http://top500.org/statistics/perfdevel>

⁵“The Opportunity and Challenges of Exascale Computing.” Summary report of the Exascale Subcommittee of the Advanced Scientific Computing Advisory Committee (ASCAC), Fall 2010. http://science.energy.gov/~media/ascr/ascac/pdf/reports/Exascale_subcommittee_report.pdf; and Thomas Sterling (with Peter Kogge). “An Overview of Exascale Architecture Challenges.” SC08 Workshop on the Path to Exascale, 2008. www.lbl.gov/CS/html/SC08ExascalePowerWorkshop/Exarch.pdf.

P-states (short for *performance states*) are different frequency settings that the OS or applications can request when the cores are in executing state (also known as *C0 state* for cores). All cores run at the same P-state frequency, as there is only one clock source for the coprocessor. Since the frequency changes require voltage changes to the core, each P-state change changes the frequency and voltages in pairs. P1 is the highest P-state setting and it can have multiple sequentially lower frequency settings. As the device characteristics may vary from device to device, each device is carefully calibrated and programmed at the factory with the proper pair of voltage and frequency settings so that different devices may have different voltage settings for the same set of frequencies. All devices within an SKU will have the same P-state settings but may vary from SKU to SKU.

In C0 state, the core and memory (*M0 state*), the card runs at its maximum *thermal design power* (TDP) rating. In core C1 state, clocks are shut off to the core (including VPU). In order to enter this state, the threads have to be halted. The last thread's halt instruction performs the necessary housekeeping before shutting off the clock. A coprocessor can have some cores in C0 state and some in C1 state with memory in M0 state. In this case, clocks are gated on a core-by-core basis, allowing the cores in C1 state to lose clock source. For the actively cooled SKUs (where the coprocessor is cooled by a cooling fan on the card), it is possible to turn the fan speed down.

If all the cores enter C1 state, the coprocessor automatically enters the Auto-Package C3 (PC3) state. In this stage, the clock source to the memory can be gated off, thus reducing memory power. This is known as M1 state for the memory.

When all cores are in C1 halt state, the coprocessor package can reduce the core voltage and enters the Deep-PC3 state. The Package C3 state shuts off clocks once all the boxes are idle, enabling deeper package states. The fan runs at ~20 percent of peak for active SKUs in this phase. The VR (voltage regulator) on the cards also enters the low power stage in Deep-PC3 mode. The memory enters the M2 state, where it is put in self-refresh mode, further reducing memory power.

Core C6 state shuts the clock and power off for the core. If power down is desired, the coprocessor OS will write to a certain status register before issuing HALT to all the threads active on that core. Power is shut off after the specified delay and the core is electrically isolated. An interrupted active thread exiting core C6 state will use the content of an internal register to determine its branch address after completing CC6 reset. At this stage, the memory clock can be fully stopped to reduce memory power and memory subsystem enters M3 state.

Note that the time stamp counter lives in the uncore region and is always running except in the package C3/C6 state and it is restored to the core on C1 exit.

Power management in the MPSS is performed in the background. In periods of idle state, the coprocessor OS puts the cores in a lower power idle state to reduce the overall power consumption.

There are two major components to power management software running on Intel Xeon Phi. One is part of the coprocessor OS and the other is part of the host driver.

Power management capabilities in the coprocessor OS are performed in ring 0 except in PC6 states, where the power is shut down to the core and needs bootstrapping of the core. Here the OS boot loader helps in bringing the core up after the core shuts down. The OS kernel performs the following functions related to power management and exposes services to perform these functions:

- Selects and sets the coprocessor P-states, including Turbo mode. There is a utility (*micsmc*) shipped with the MPSS driver software package that lets users turn the Turbo mode on/off with the help of the coprocessor OS.
- Collects the device usage and power/thermal data for the coprocessor and exposes it through the MPSS utility to the coprocessor user.
- Sets the core power states as C-states.
- Saves and restores the CPU context on core C6 entry and exit. After C6 state exit, the boot loader performs reset initialization and passes control to the GDDR resident coprocessor OS kernel.

Special instructions, such as HLT used by OS or MWAIT used by applications for idling routines, enable processors to take advantage of hardware C-states. The OS boot loader (see Chapter 7) helps to get the core back to the running state after a return to C6 power-saving state. It is put into service on PC6 state exit. The PC6 state lowers

the core voltage V_{ccP} to zero. As a result, when the core voltage and clocks are restored, they start executing from the default instruction pointer address $FFFF_FFF0h$. However, rather than going through the normal boot sequence, as in the case of a cold restart, the PC6 state maintains the GDDR content by self-refresh and saves the hardware state before entering the C6 state. This minimizes the time required to bring the cores to full operational mode.

The host driver component of MPSS helps in power management by performing the following functions:

- Monitors and manages coprocessor idle states
- Replies to server management queries
- Provides a power management command status interface between the host and coprocessor software as may be required by cluster management tools

Power reduction is managed by the *power management* (PM) software using the states and services described above. Carefully selecting the P-states and idle states, the PM software can reduce power consumption without impacting application performance. It uses *demand-based switching* (DBS) to select the P-state. A decrease in CPU demand causes a reduction in the P-state (i.e., an increase in core frequency), and an increase in CPU demand causes an increase in the P-state (i.e., a reduction in core frequency). The P-state selection algorithm is thus tuned to detect changes in workload demands executing on the coprocessor with the help of some tunable system parameters. These system parameters can be set with the help of system policies selected by users. The P-state selection modules register with the coprocessor OS through a timer task and get invoked from time to time to evaluate the system P-states. The OS maintains a per-coprocessor running counter to compute the coprocessor core utilization. The evaluation tasks are invoked in the regular timer interval and executed in the background. These tasks use the per-core utilization data to compute maximum utilization values across all the cores to determine the target P-state and Turbo modes. (See the Intel Xeon Phi data sheet for the details of the power state management algorithm employed by MPSS.)

The P-state control module implements the P-states on the coprocessor. The P-state module of the system software exposes this functionality to the power management software. This module is part of the coprocessor's OS and allows the client software to set/get the P-state, register notifications, set core frequency, and voltage fuse values.

Idle State Management

Proper idle state selection allows the coprocessor to save even more power in addition to the P-state management described above. The policy uses the expected time that the cores are assumed to be idle together with the latency to get into and out of an idle state to select the specific idle state. The deeper sleep state requires longer latency to enter and exit the state.

The coprocessor supports package idle states such as Auto-C3, where all the cores and agents on the ring are clock-gated: the deeper-PC3 state, which reduces the voltage further, and the package C6 state, which shuts off the power to the package while keeping the GDDR memory content intact by autorefresh. The main differences between the package idle and core idle states are:

1. The package idle state requires all the cores to be idle.
2. The package idle state is controlled by the host driver except for Auto-C3 state.
3. Wake up from the package idle state requires external hardware events or coprocessor driver interrupts.
4. Package idle state causes the package *timestamp counter* (TSC) and local APIC timer to freeze, thus requiring an external timer, such as the *elapsed time counter* (ETC), outside the core to synchronize TSC when the package wakes up.

The coprocessor OS and host driver together play a central role in managing package idle states. The host processor may overwrite the coprocessor OS determination of PC3 selection, as the coprocessor may not have visibility in the uncore events. Also some package states, such as the Deep-PC3 and PC6 states, need host driver intervention to wake up the core.

Reliability Availability and Serviceability Features in the Intel Xeon Phi Coprocessor

Intel Xeon Phi is expected to be used in a cluster environment requiring reliability as a feature to support technical computing applications. It has built-in hardware support and tools to reduce the error rate and improve node availability and serviceability for fault tolerance. Intel Xeon Phi coprocessors implement extended machine check (MCA) features to help the software detect imminent failure and perform graceful service degradation when a component such as the core fails. An Intel Xeon Phi coprocessor reads bits from flash memory at boot time to disable the coprocessor component that MCA has reported as failing. (Please refer to the Intel Xeon Phi data sheet for details of MCA implementation in Xeon Phi.)

The memory controller of the coprocessor uses *cyclic redundancy control* (CRC) and *error correction code* (ECC) protection of the memory content. These enable machine MCA events. The controller can detect single- and double-bit errors and fix the single-bit errors. The coprocessor supports parity on the command and address interface as part of the error detection mechanism. ECC is also used to reduce error rates in cache lines and other memory arrays in the coprocessor.

Summary

This chapter discussed the characteristics of the communication and system management hardware of the Xeon Phi coprocessor. You looked at how data are transferred between the host and the coprocessor using DMA, and you saw the conditions that cause poor data transfer and how to avoid them. The chapter also covered reliability and power management features built into the Xeon Phi coprocessor.

The next chapter will introduce you to the system software layers that will help you manage and use the Xeon Phi coprocessor.