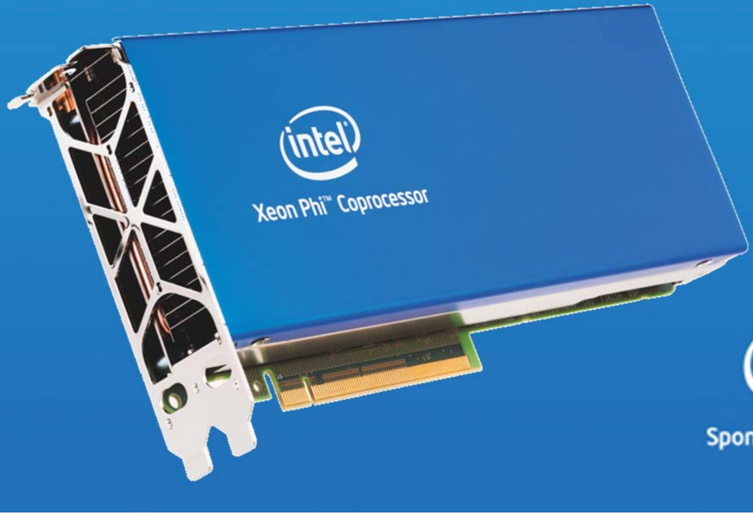
Xeon Phi™

First Intel® Xeon Phi™ Coprocessor
Technology Conference, iXPTC 2013
New York, NY | March 13-14th, 2013

MPSS – The Software Stack



Ravi Murty, Intel

ravi.murty@intel.com

Sponsors of Tomorrow.

Legal Disclaimer

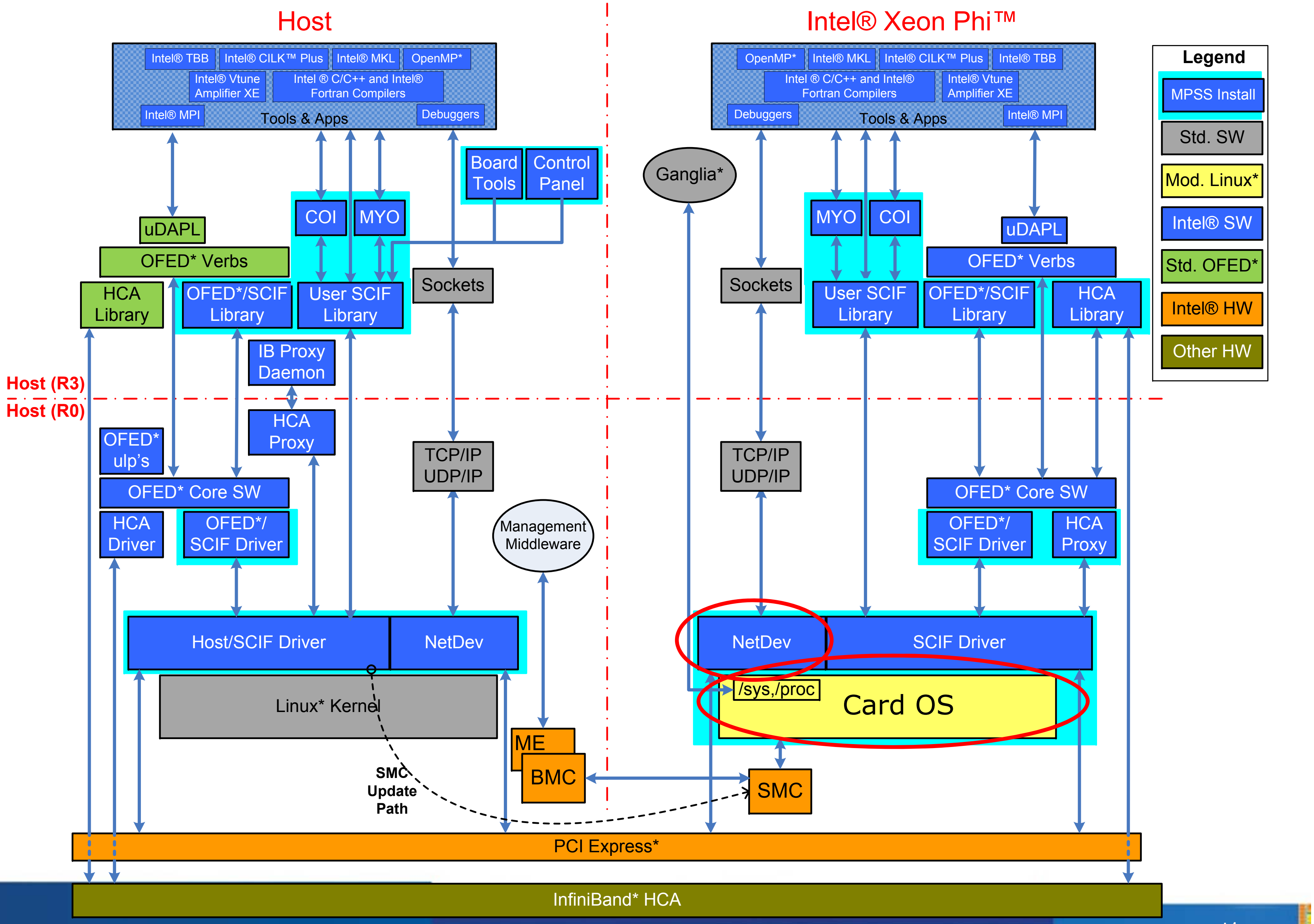
- INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPETY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL® PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.
- Intel may make changes to specifications and product descriptions at any time, without notice.
- All products, dates, and figures specified are preliminary based on current expectations, and are subject to change without notice.
- Intel, processors, chipsets, and desktop boards may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.
- Sandy Bridge and other code names featured are used internally within Intel to identify products that are in development and not yet publicly announced for release. Customers, licensees and other third parties are not authorized by Intel to use code names in advertising, promotion or marketing of any product or services and any such use of Intel's internal code names is at the sole risk of the user
- Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to http://www.intel.com/performance
- Intel, Core, Xeon, VTune, Cilk, Intel and Intel Sponsors of Tomorrow. and Intel Sponsors of Tomorrow. logo, and the Intel logo are trademarks of Intel Corporation in the United States and other countries.
- *Other names and brands may be claimed as the property of others.
-
- Hyper-Threading Technology: Requires an Intel® HT Technology enabled system, check with your PC manufacturer. Performance will vary depending on the specific hardware and software used. Not available on all Intel® Core™ processors. For more information including details on which processors support HT Technology, visit http://www.intel.com/info/hyperthreading
- Intel® 64 architecture: Requires a system with a 64-bit enabled processor, chipset, BIOS and software. Performance will vary depending on the specific hardware and software you use. Consult your PC manufacturer for more information. For more information, visit http://www.intel.com/info/em64t
- Intel® Turbo Boost Technology: Requires a system with Intel® Turbo Boost Technology capability. Consult your PC manufacturer. Performance varies depending on hardware, software and system configuration. For more information, visit http://www.intel.com/technology/turboboost

Agenda

- Section 1: The System SW Stack
  - Card OS
  - Symmetric Communications Interface (SCIF)
  - Code Examples
- Section 2: Compiler Runtimes
  - Coprocessor Offload Infrastructure (COI)
  - Code Examples
- Section 3: Coprocessor Communication Link (CCL)
  - IB-SCIF
  - CCL Direct and Proxy
  - MPI Dual-DAPL
- Section 4: Heterogeneous Programming with Offload
  - Code Examples

Intel® Xeon Phi™ Coprocessor Arch – System SW Perspective

- Large SMP UMA machine – a set of x86 cores to manage
  - 4 threads and 32KB L1I/D, 512KB L2 per core
  - Supports loadable kernel modules – we'll talk about one today
- Standard Linux kernel from kernel.org
  - 2.6.38 in the most recent release
  - Completely Fair Scheduler (CFS), VM subsystem, File I/O
- Virtual Ethernet driver– supports NFS mounts from Intel® Xeon Phi™ Coprocessor
- New vector register state per thread for Intel® IMCI
  - Supports "Device Not Available" for Lazy save/restore
- Different ABI – uses vector registers for passing floats
  - Still uses the x86_64 ABI for non-float parameter passing (rdi, rsi, rdx ..)



Host

Intel® Xeon Phi™

Intel® TBB
Intel® CILK™ Plus
Intel® MKL
OpenMP*
Intel® Vtune Amplifier XE
Intel ® C/C++ and Intel® Fortran Compilers
Intel® MPI
Tools & Apps
Debuggers

Board Tools
Control Panel
Ganglia*

uDAPL
OFED* Verbs
HCA Library
OFED*/SCIF Library
User SCIF Library
COI
MYO
Sockets
IB Proxy Daemon

Host (R3)
Host (R0)

OFED* ulp's
OFED* Core SW
HCA Driver
OFED*/ SCIF Driver
HCA Proxy
TCP/IP UDP/IP
Management Middleware

Host/SCIF Driver
NetDev
Linux* Kernel
ME
BMC
SMC Update Path

NetDev
SCIF Driver
/sys,/proc
Card OS
SMC

PCI Express*
InfiniBand* HCA

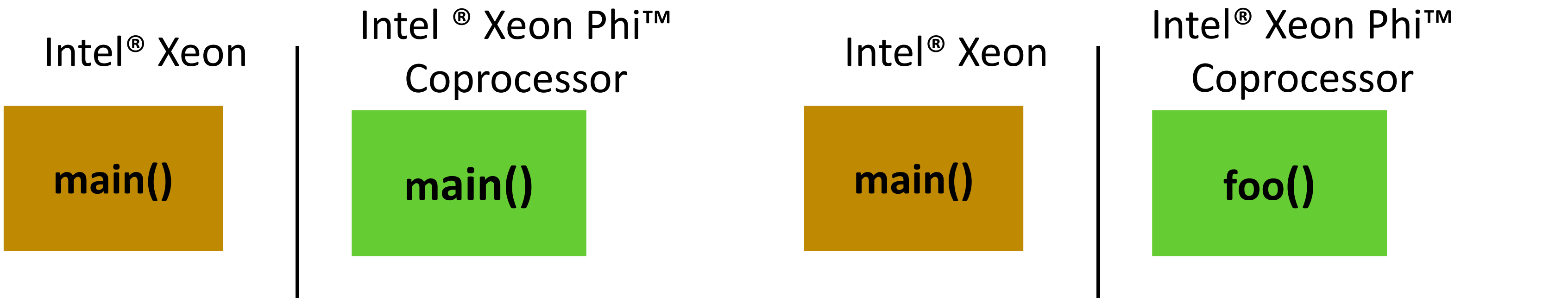
Legend
MPSS Install
Std. SW
Mod. Linux*
Intel® SW
Std. OFED*
Intel® HW
Other HW

Card OS

- Bootstrap
  - Memory training
  - Build MP Table, e820 map, load image in memory (bzImage)
  - Jump to 32-bit protected mode entry point in the kernel

- It’s *just* Linux with some minor modifications:
  - IPI format, local APIC calibration, no support for compatibility mode (CSTAR)
  - No *global bit* in PTE/PDE entries.
  - IOAPIC programmed via MMIO
  - Instructions not supported: cmov, in/out, monitor/mwait, fence
  - No support for MMX/SSE registers
  - Save/restore of vector state via DNA (CR0.TS=1)

Execution Modes



Intel® Xeon

main()

Intel ® Xeon Phi™ Coprocessor

main()

Intel® Xeon

main()

Intel® Xeon Phi™ Coprocessor

foo()

Native

- Card is an SMP machine running Linux
- Separate executables run on both MIC and Xeon
  - e.g. Standalone MPI applications
- No source code modifications most of the time
  - Recompile code for Xeon Phi™ Coprocessor
- Autonomous Compute Node (ACN)

Offload

- “main” runs on Xeon
- Parts of code are offloaded to MIC
- Code that can be
  - Multi-threaded, highly parallel
  - Vectorizable
  - Benefit from large memory BW
- Compiler Assisted vs. Automatic
  - #pragma offload (…)

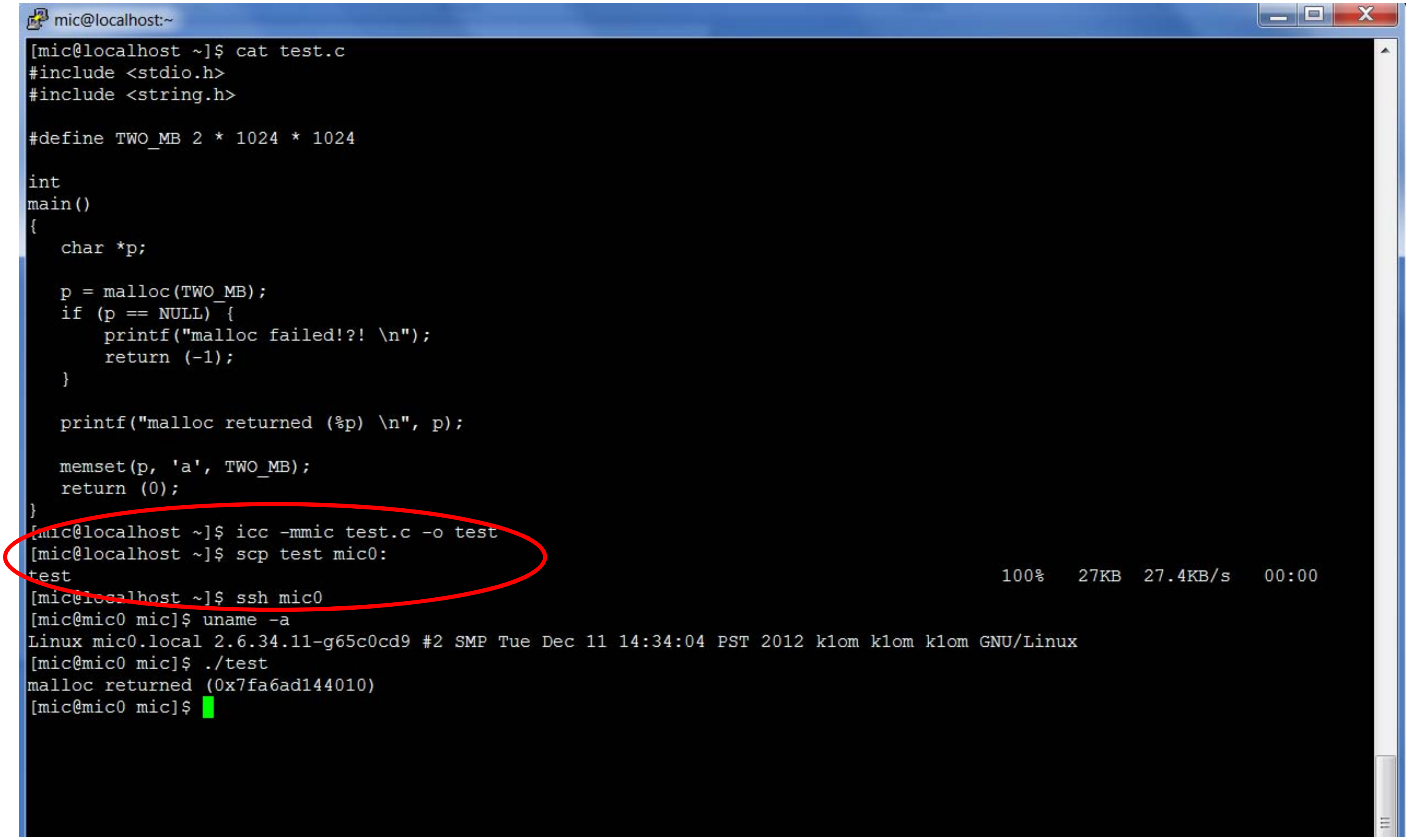
Native is Easy

- Cross compile your application for *k1om* arch
  - Intel C/C++ and Fortran compiler, k1om aware GCC port.
  - Binutils for k1om e.g. objdump
  - LSB – glibc, libm, librt, libcurses etc.
  - Busybox – minimal shell environment

- Virtual Ethernet driver allows:
  - ssh, scp
  - NFS mounts

You still have to spend time parallelizing and vectorizing your application for performance on Intel® Xeon Phi™ Coprocessor

“Hello World”



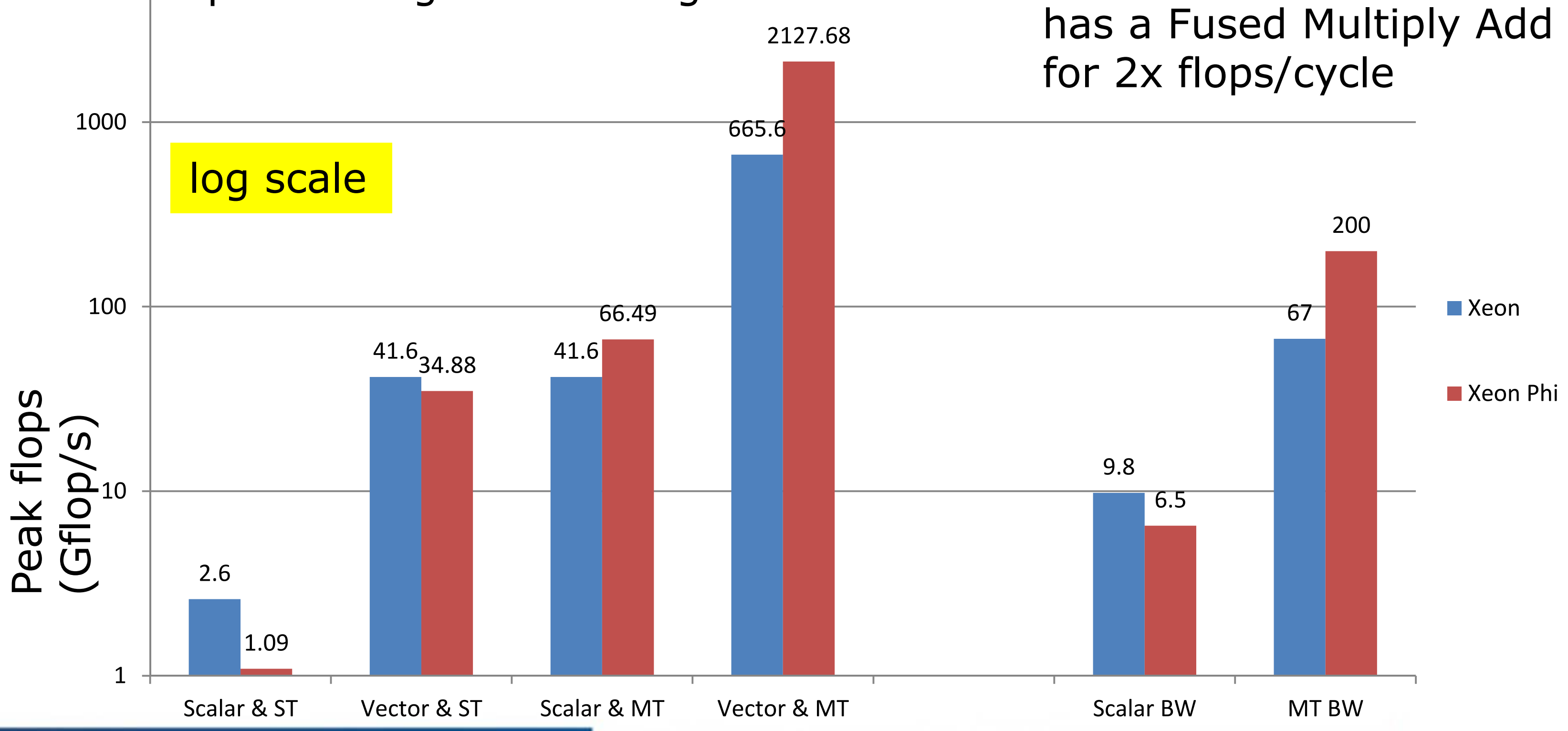
```
[mic@localhost ~]$ cat test.c
#include <stdio.h>
#include <string.h>

#define TWO_MB 2 * 1024 * 1024

int
main()
{
   char *p;

   p = malloc(TWO_MB);
   if (p == NULL) {
       printf("malloc failed!?! \n");
       return (-1);
   }

   printf("malloc returned (%p) \n", p);

   memset(p, 'a', TWO_MB);
   return (0);
}
[mic@localhost ~]$ icc -mmic test.c -o test
[mic@localhost ~]$ scp test mic0:
test                                                                         100%   27KB  27.4KB/s   00:00
[mic@localhost ~]$ ssh mic0
[mic@mic0 mic]$ uname -a
Linux mic0.local 2.6.34.11-g65c0cd9 #2 SMP Tue Dec 11 14:34:04 PST 2012 k1om k1om k1om GNU/Linux
[mic@mic0 mic]$ ./test
malloc returned (0x7fa6ad144010)
[mic@mic0 mic]$
```

Performance Considerations for Native

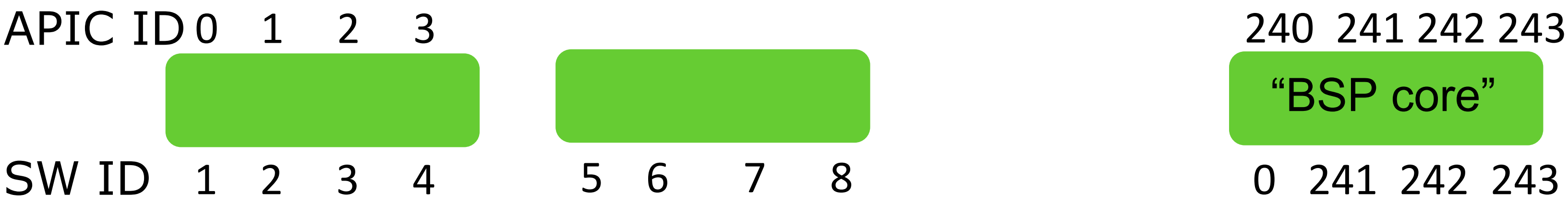
- Single vs. Multi-threaded application
  - Scalability is important
- Scalar vs. Vector code
- Explicit cache management
  - SW prefetching and evicting

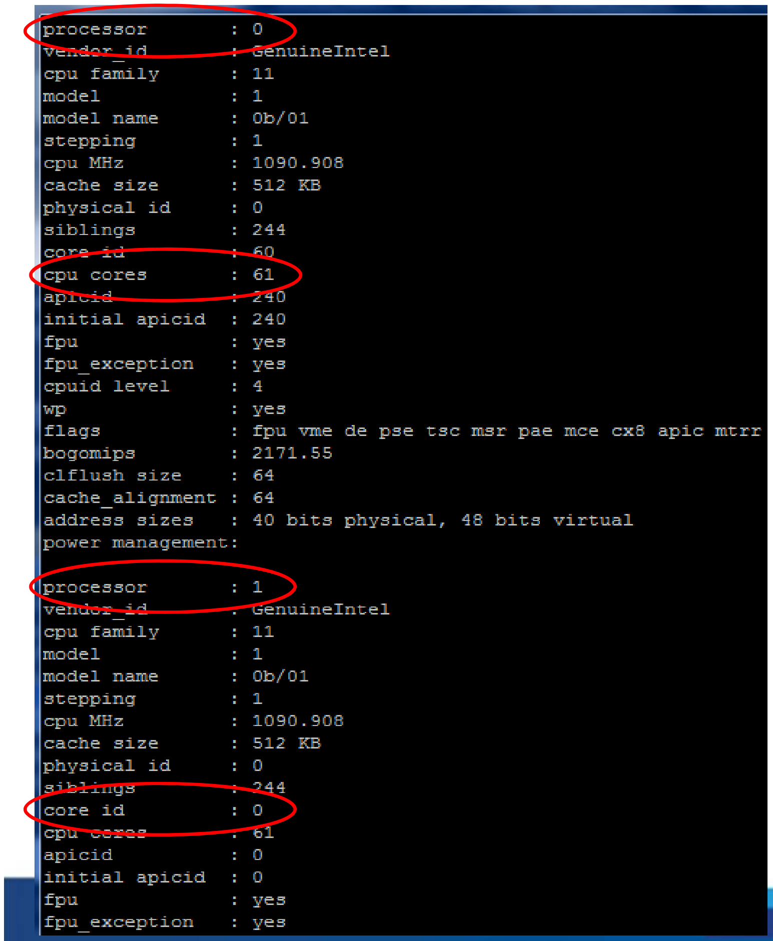
	Frequency	cores	vector width
Xeon	2.6	16	8
Xeon Phi™ Coprocessor	1.09	61	16

Intel® Xeon Phi™ Coprocessor has a Fused Multiply Add (FMA) for 2x flops/cycle



System topology and Thread Affinity





```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 11
model           : 1
model name      : 0b/01
stepping        : 1
cpu MHz         : 1090.908
cache size      : 512 KB
physical id     : 0
siblings        : 244
core id         : 60
cpu cores       : 61
apicid          : 240
initial apicid  : 240
fpu             : yes
fpu_exception   : yes
cpuid level     : 4
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic mtrr
bogomips        : 2171.55
clflush size    : 64
cache_alignment : 64
address sizes   : 40 bits physical, 48 bits virtual
power management:

processor       : 1
vendor_id       : GenuineIntel
cpu family      : 11
model           : 1
model name      : 0b/01
stepping        : 1
cpu MHz         : 1090.908
cache size      : 512 KB
physical id     : 0
siblings        : 244
core id         : 0
cpu cores       : 61
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
```

- Why – threads sharing L1 and L2 cache
- sched_affinity/pthread_setaffinity_np or KMP_AFFINITY=proclist=[…]

- But is your affinity correct or expected?
  - KMP_AFFINITY=explicit, proclist=[0-243]

```
OMP internal thread 0 -> CPU # 0
OMP internal thread 1 -> CPU # 1
OMP internal thread 2 -> CPU # 2
 …
OMP internal thread 243 -> CPU # 243
```

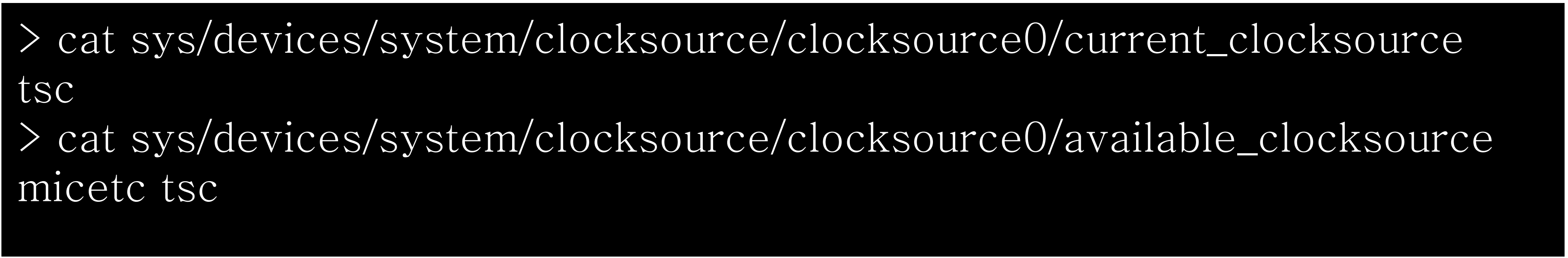
  - KMP_AFFINITY=explicit,proclist[1-243, 0]?

Memory – Huge Pages and Pre-faulting

- IA processors support multiple page sizes; commonly 4K and 2MB
- *Some* applications will benefit from using huge pages
  - Applications with sequential access patterns will improve due to larger TLB "reach"
- TLB miss vs. Cache miss
  - TLB miss means walking the 4 level page table hierarchy
    - Each page walk could result in additional cache misses
  - TLB is a scarce resource and you need to "manage" them well
- On Intel® Xeon Phi™ Coprocessor
  - 64 entries for 4K, 8 entries for 2MB
  - Additionally, 64 entries for second level DTLB.
    - Page cache for 4K, L2 TLB for 2MB pages
- Linux supports huge pages – CONFIG_HUGETLBFS
  - 2.6.38 also has support for Transparent Huge Pages (THP)
- Pre-faulting via MAP_POPULATE flag to mmap()

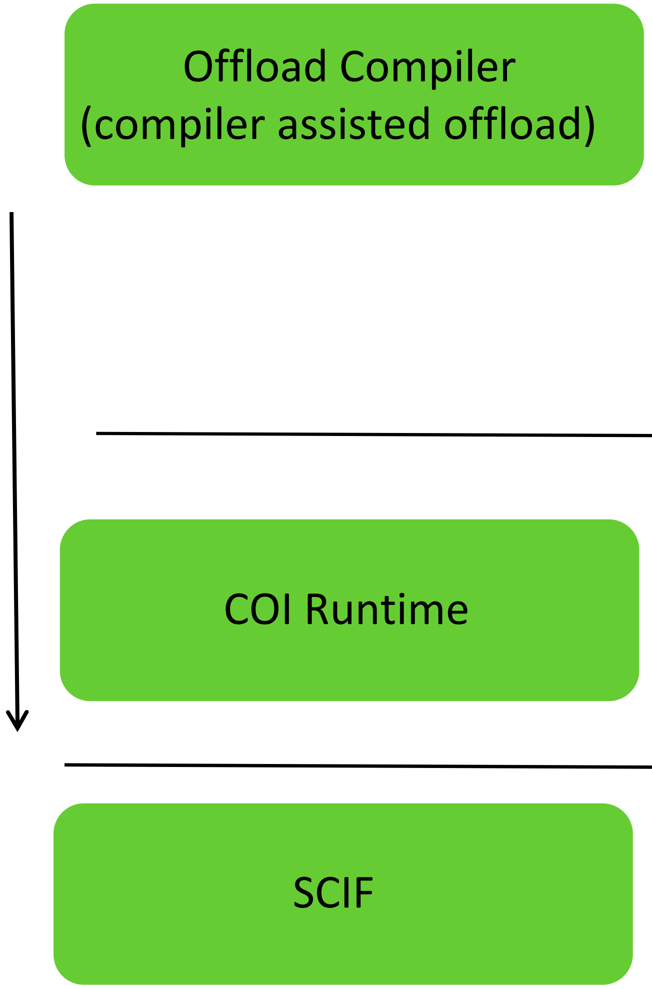
Clocksource and gettimeofday()

- A “clocksource” is a monotonically increasing counter
- Intel® Xeon Phi™ Coprocessor has three clocksources
  - jiffies, tsc, micetc
- The Local APIC in each HW thread has a timer (HZ)
  - jiffies is a good clocksource, but very low resolution
- TSC is a good clocksource
  - But TSC is not *frequency invariant* and *non-stop*
  - Future release will use another clocksource to fix this.
- Elapsed Time Counter (ETC) that is frequency invariant
  - Expensive when multiple gettimeofday() calls – involves an MMIO read
- Recommend using “clocksource = tsc” on kernel command line



```
> cat sys/devices/system/clocksource/clocksource0/current_clocksource
tsc
> cat sys/devices/system/clocksource/clocksource0/available_clocksource
micetc tsc
```

Coprocessor Offload Programming – The Big Picture

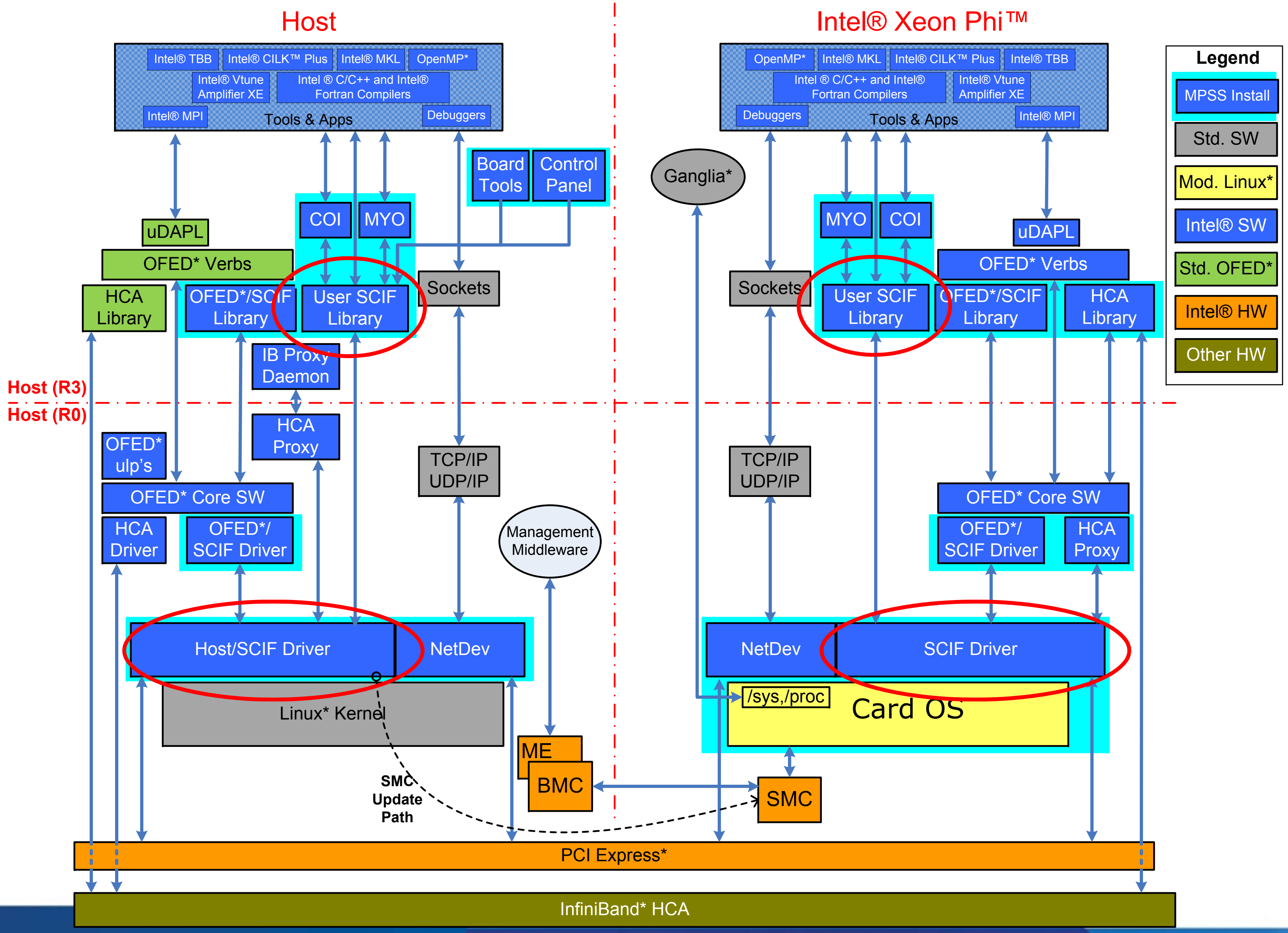


```
__declspec(target(mic)) int numFloats = 100;

__declspec(target(mic)) float input1[100], input2[100];

__declspec(target(mic)) float output[100];

pragma offload target(mic) in(input1, input2, numFloats) out (output) {
        for(int j=0; j<numFloats; j++)  {
                output[j] = input1[j] + input2[j];

        }

}
```

```
COIProcessCreateFromFile( … );

COIBufferCreate( … );

…
COIPipelineRunFunction ( … );
```

```
scif_vwriteto( … );
scif_send( … );
scif_recv( … );
scif_vreadfrom( … );
```

Symmetric Communications Interface (SCIF)

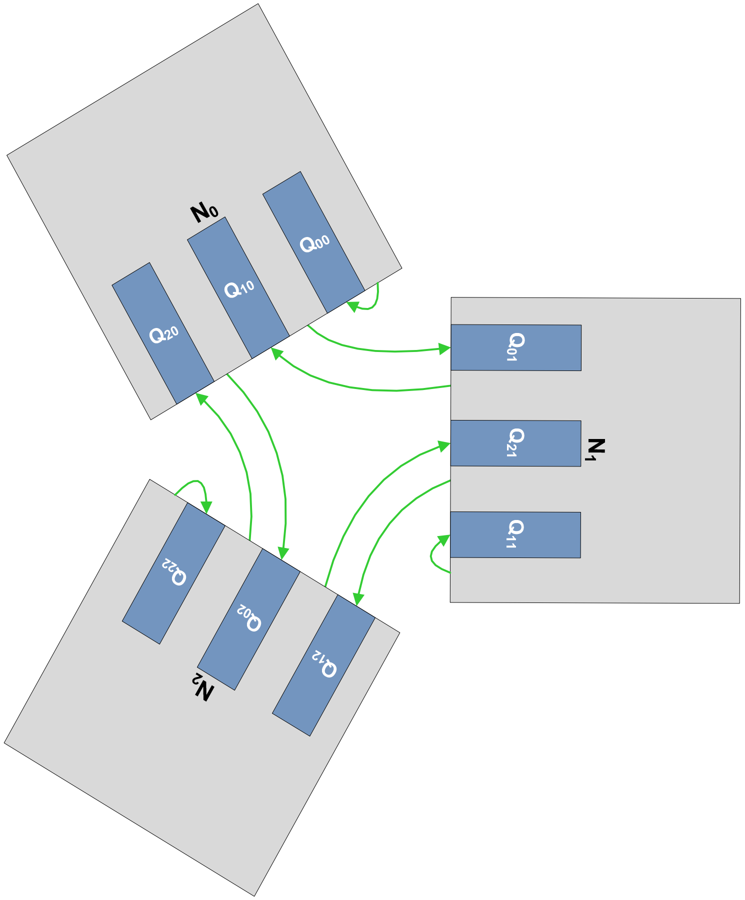

Host
Intel® Xeon Phi™
Intel® TBB
Intel® CILK™ Plus
Intel® MKL
OpenMP*
Intel® Vtune Amplifier XE
Intel ® C/C++ and Intel® Fortran Compilers
Intel® MPI
Tools & Apps
Debuggers
Board Tools
Control Panel
Ganglia*
COI
MYO
uDAPL
OFED* Verbs
HCA Library
OFED*/SCIF Library
User SCIF Library
Sockets
IB Proxy Daemon
Host (R3)
Host (R0)
HCA Proxy
OFED* ulp's
OFED* Core SW
TCP/IP UDP/IP
HCA Driver
OFED*/ SCIF Driver
Management Middleware
Host/SCIF Driver
NetDev
Linux* Kernel
ME
BMC
SMC Update Path
SCIF Driver
/sys,/proc
Card OS
SMC
PCI Express*
InfiniBand* HCA
Legend
MPSS Install
Std. SW
Mod. Linux*
Intel® SW
Std. OFED*
Intel® HW
Other HW

SCIF Introduction

- Primary goal: Simple, efficient communications interface between "nodes"
  - Symmetric across Xeon host and Xeon Phi™ Coprocessor cards
  - User mode (ring 3) and kernel mode (ring 0) APIs
    - Each has several mode specific functions
    - Otherwise virtually identical
  - Expose/leverage architectural capabilities to map host/card mapped memory and DMA engines
- Support a range of programming models
- Identical APIs on Linux and Windows

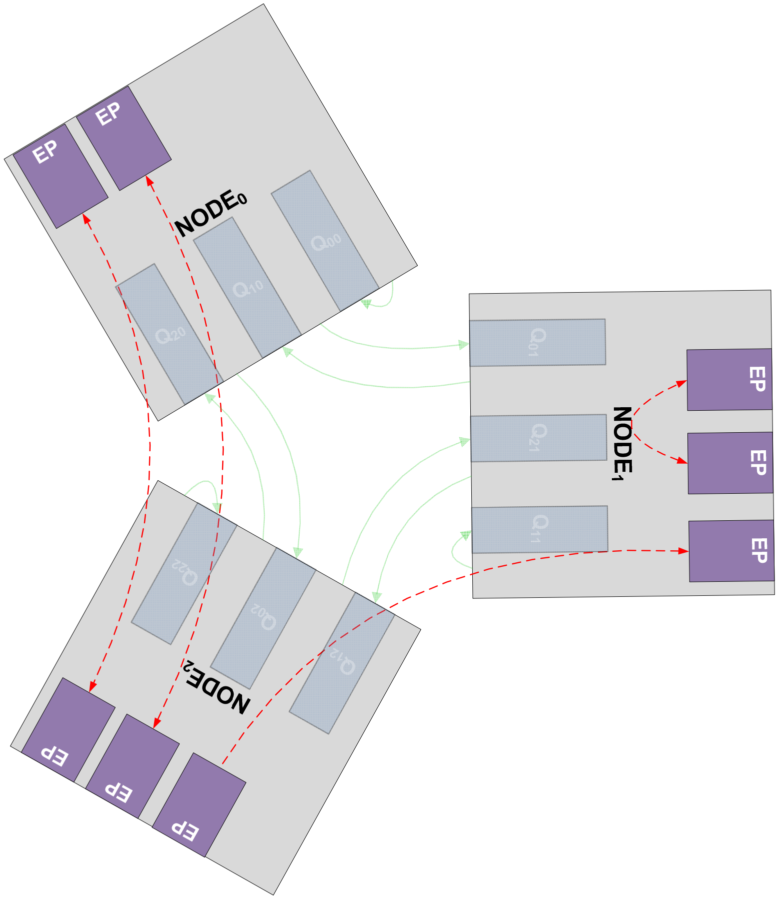
SCIF Introduction (2)

- Fully connected network of SCIF nodes
  - Each SCIF node communicates directly with each other node through the PCIe root complex
- Nodes are physical endpoints in the network
  - Xeon host and Xeon Phi™ Coprocessor cards are SCIF nodes
- SCIF communication is *intra-platform*
- Key concepts:
  - SCIF drivers communicate through dedicated queue pairs
  - one “ring0 QP” for each pair of nodes
  - A receive queue (Qij) in each node is directly written to from the other node.
  - Interrupt driven, relatively low latency



SCIF - Terminology

- A SCIF port is a logical destination on a node
  - (Node, port) pair analogous to a TCP/IP address… 192.168.1.240:22
- An endpoint is a logical entity, bound to a port (endpoint ≈ socket), and through which a process:
  - Accepts connection requests (listening endpoint), OR
  - Communicates with another process (connected endpoint)
- A connection is a pair of connected endpoints
- An endpoint can be connected to only one other endpoint
- A process may create an arbitrary number of connections
- Connected endpoints are typically on different nodes but may be on the same node (loopback)

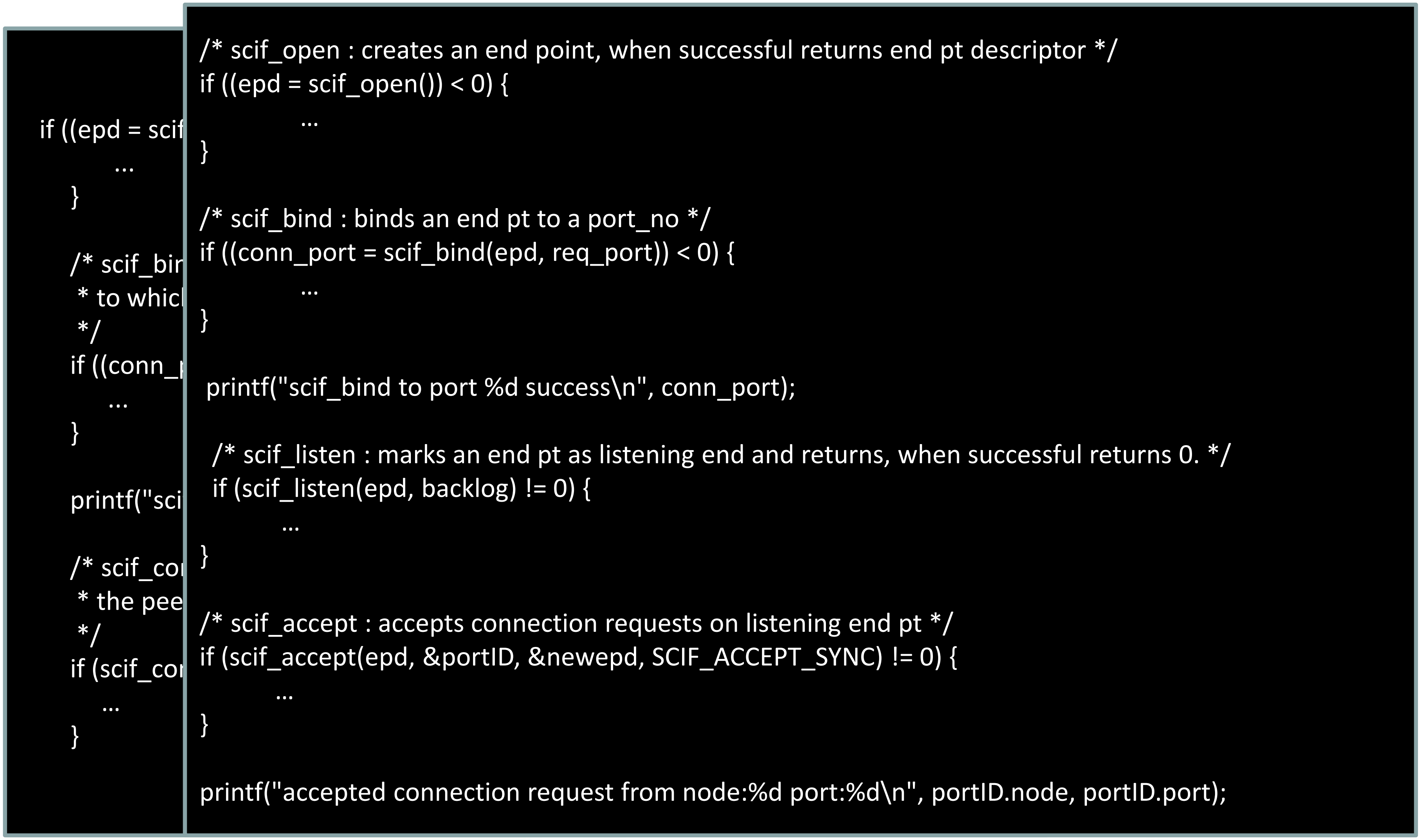


SCIF API: Connection

- Connection establishment between processes

- scif_epd_t **scif_open**(void)
- int **scif_bind**(scif_epd_t epd, uint16_t pn)
- int **scif_listen**(scif_epd_t epd, int backlog)
- int **scif_connect**(scif_epd_t epd, struct scif_portID *dst)
- int **scif_accept**(scif_epd_t epd, struct scif_portID *peer, scif_epd_t *newepd, int flags)
- int **scif_close**(scif_epd_t epd)

SCIF – Connection API example



```
if ((epd = scif
        ...
    }

    /* scif_bir
     * to whicl
     */
    if ((conn_
        ...
    }

    printf("sci

    /* scif_co
     * the pee
     */
    if (scif_co
        ...
    }
```

```
/* scif_open : creates an end point, when successful returns end pt descriptor */
if ((epd = scif_open()) < 0) {
                ...
}

/* scif_bind : binds an end pt to a port_no */
if ((conn_port = scif_bind(epd, req_port)) < 0) {
                ...
}

 printf("scif_bind to port %d success\n", conn_port);

  /* scif_listen : marks an end pt as listening end and returns, when successful returns 0. */
  if (scif_listen(epd, backlog) != 0) {
              ...
}

/* scif_accept : accepts connection requests on listening end pt */
if (scif_accept(epd, &portID, &newepd, SCIF_ACCEPT_SYNC) != 0) {
            ...
}

printf("accepted connection request from node:%d port:%d\n", portID.node, portID.port);
```

SCIF API: Messaging

- Send/Recv messages between connected endpoints. Good for non-latency/BW sensitive messages between end-points.
- Two sided communication

- int **scif_send**(scif_epd_t epd, void *msg, size_t len, int flags);
- int **scif_recv**(scif_epd_t epd, void *msg, size_t len, int flags);

SCIF – Messaging API example

```
/* send & recv small data to verify the
 * scif_send : send messages between
 * returns after sending entire msg ur
 * only those bytes that can be sent w
 */
send_buf = (char *)malloc(msg_size);
memset(send_buf, 0xbc, msg_size);
curr_addr = send_buf;
curr_size = msg_size;
while ((no_bytes = scif_send(epd, cur
            curr_addr = curr_addr + no_b
            curr_size = curr_size - no_byt
            if(curr_size == 0)
                   break;
}
if (no_bytes < 0) {
          …
}
```

```
recv_buf = (char *)malloc(msg_size);
      memset(recv_buf, 0x0, msg_size);
      curr_addr = recv_buf;
      curr_size = msg_size;
      while ((no_bytes = scif_recv(epd, curr_addr, curr_size, block)) >= 0) {
             curr_addr = curr_addr + no_bytes;
             curr_size = curr_size - no_bytes;
             if(curr_size == 0)
                    break;
      }
      if (no_bytes < 0) {
             …
```

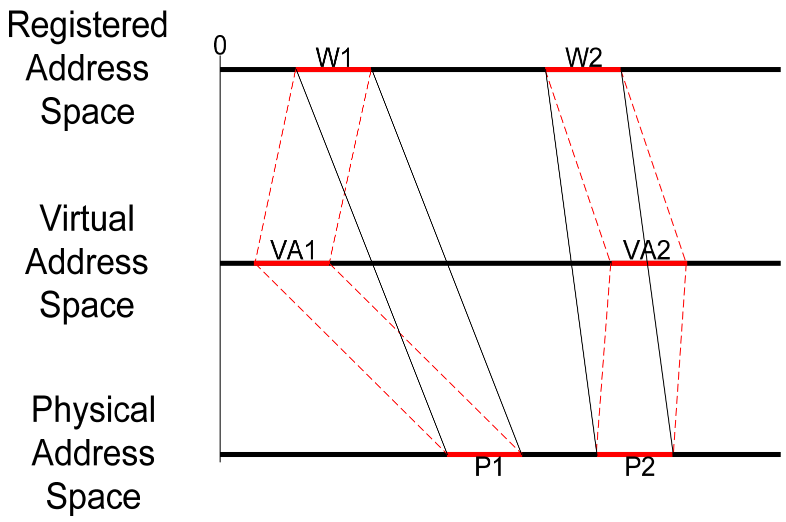
SCIF API: Registration

- Exposes local physical memory for remote access via a local Registered Address Space

- off_t **scif_register**(scif_epd_t epd, void *addr, size_t len, off_t offset, int prot_flags, int map_flags);
- int **scif_unregister**(scif_epd_t epd, off_t offset, size_t len);

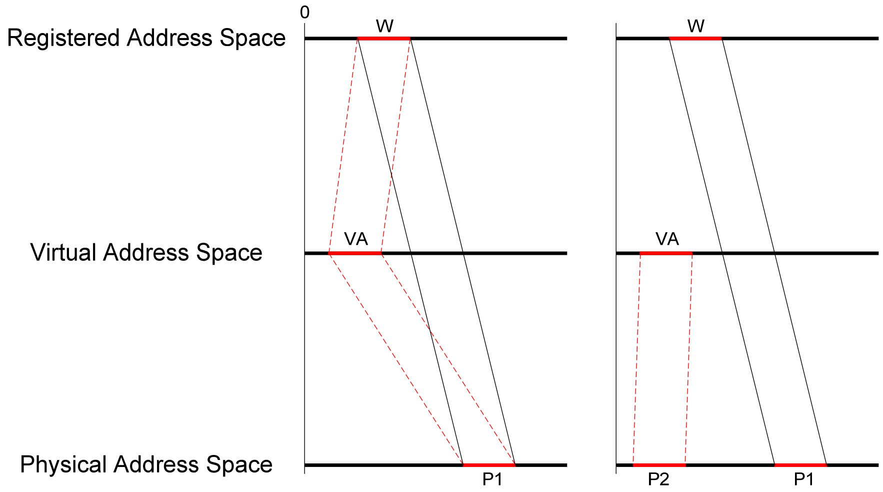
What is Memory Registration?

- *Registration* exposes local physical memory for remote access
- RMAs and other operations on remote memory are performed with registered addresses
- Each *connected* endpoint has a *local* registered address space (RAS).
  - Registration creates a registered window (window) that is a mapping from a range, W, of the RAS of some endpoint to the set of physical pages, P, backing some range, VA, of virtual address space
  - Physical pages are pinned as long as the window exists
- The registered address space of the peer endpoint is the *remote* RAS
  - Internally each endpoint has a copy of its peer's registered address space (the window mapping information)
  - This allows very efficient RMAs since both the local and remote physical pages addresses are available locally



Memory Registration (2)

- A window continues to represent the same physical pages even if the VA range is remapped or unmapped



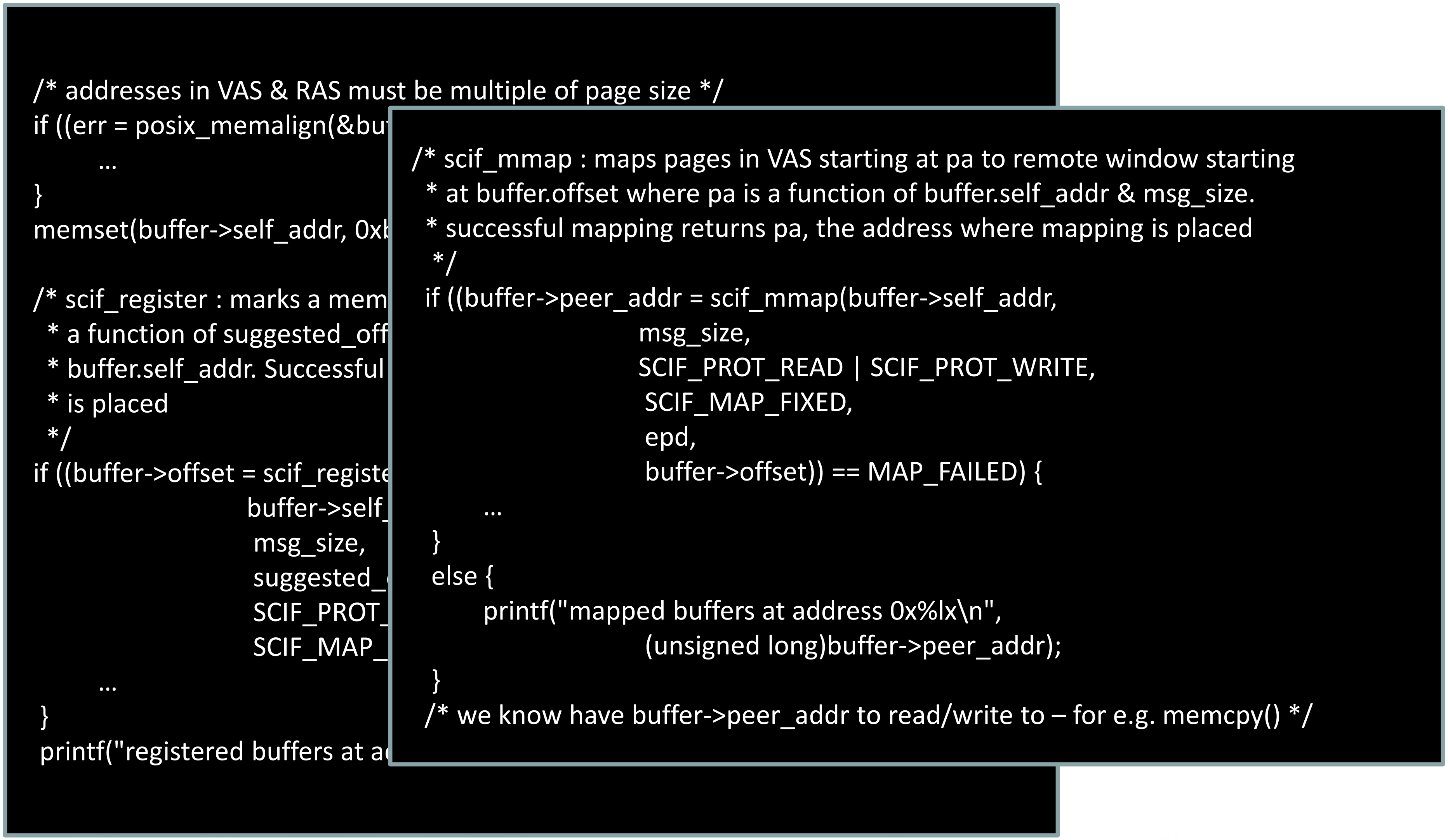
- scif_unregister() makes a window unavailable for subsequent RMA's, mappings
  - A window exists, and (therefore) the pages it represents remain pinned, as long as there are references against it:
    - In-process RMAs
    - scif_mmap()'d by the peer node.
  - Only after all references are removed is the unregistered window deleted

SCIF API: Mapping Remote Memory

- Maps remote physical memory pages into local virtual address space of process

- void *__scif_mmap__(void *addr, size_t len, int prot, int flags, scif_epd_t epd, off_t offset);
- int __scif_munmap__(void *addr, size_t len);

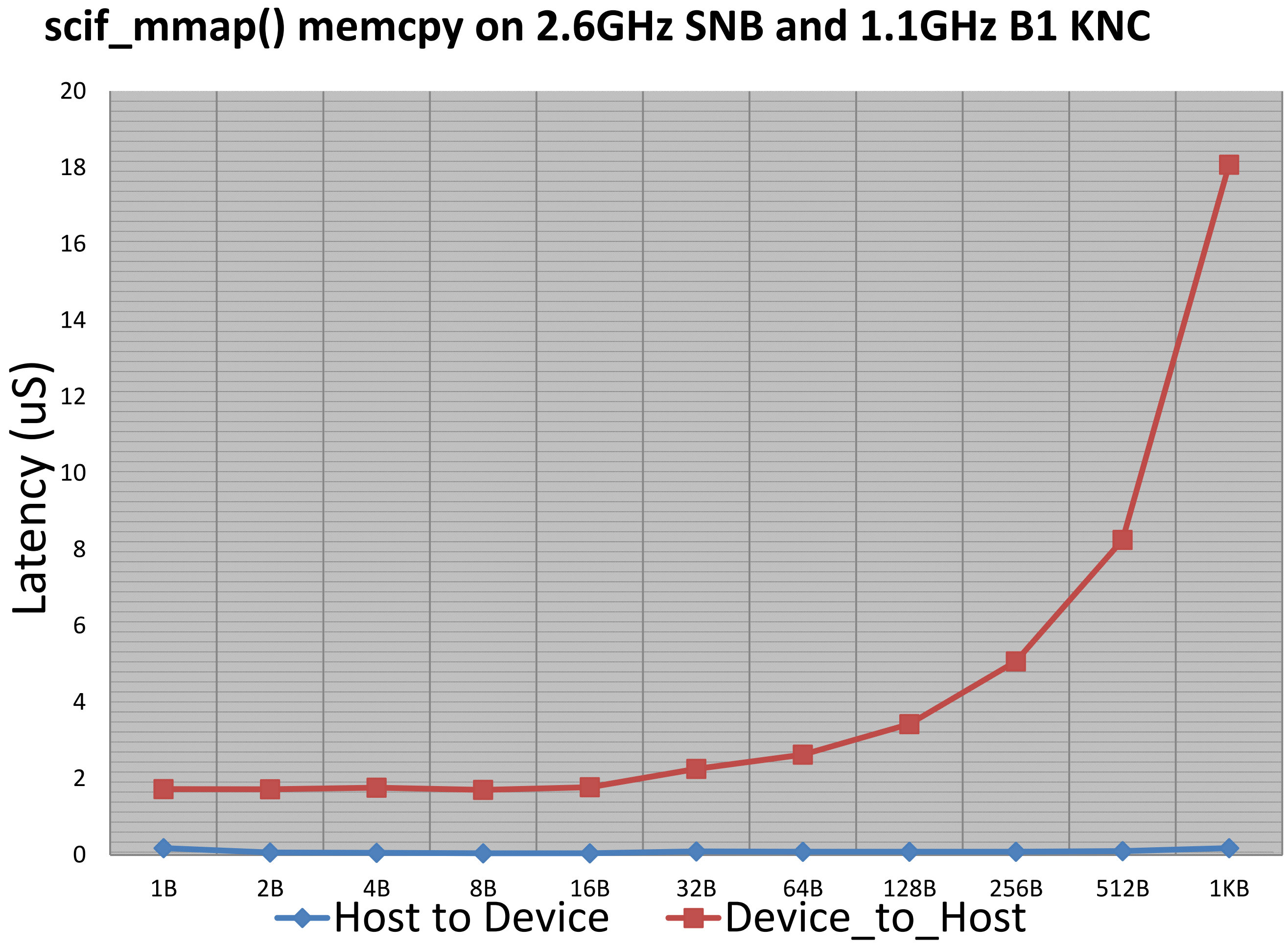
SCIF Example: Registration/mmap



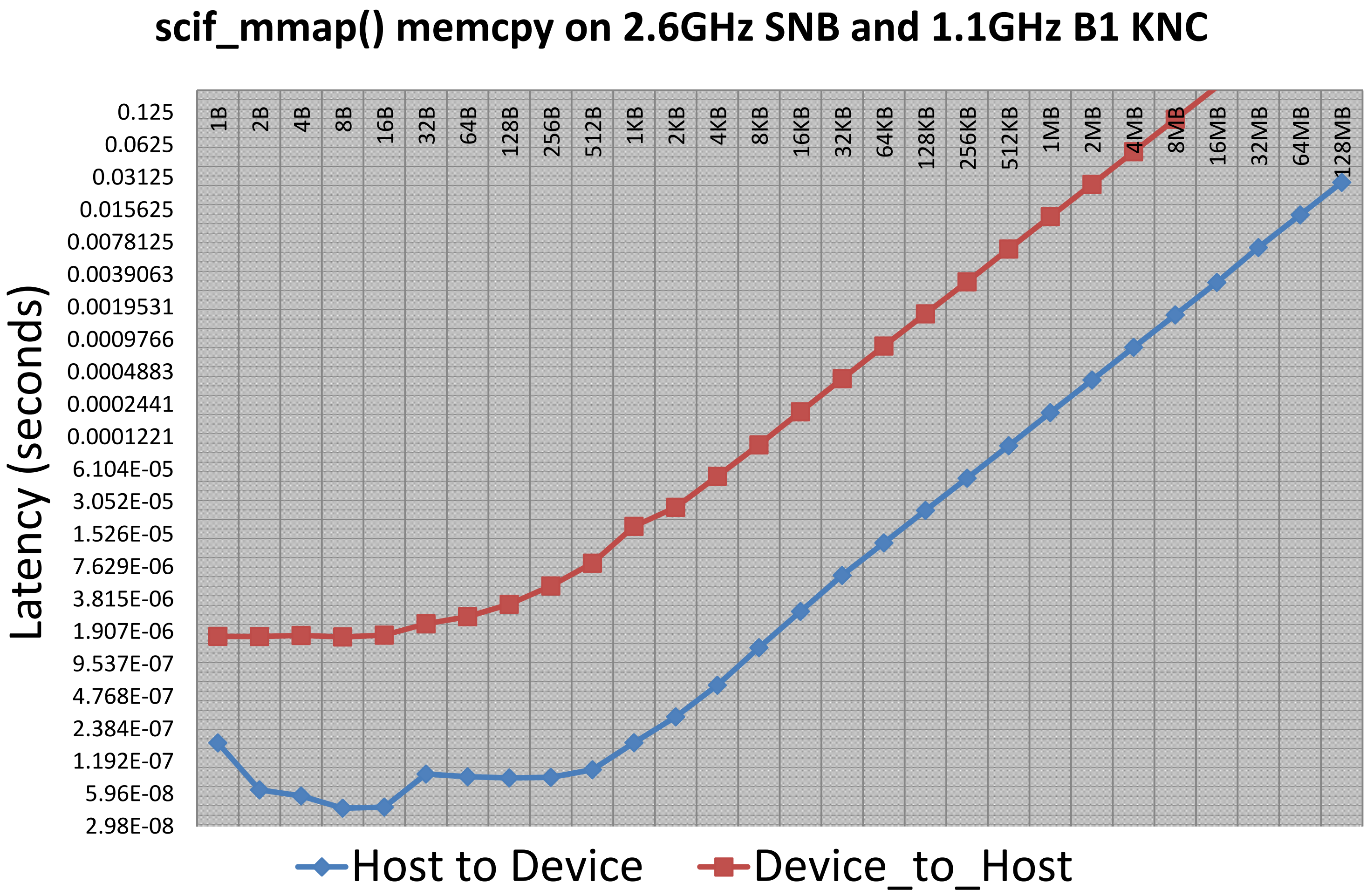
```
/* addresses in VAS & RAS must be multiple of page size */
if ((err = posix_memalign(&bu
        …
}
memset(buffer->self_addr, 0xb

/* scif_register : marks a mem
  * a function of suggested_off
  * buffer.self_addr. Successful
  * is placed
  */
if ((buffer->offset = scif_registe
                         buffer->self_
                          msg_size,
                          suggested_
                          SCIF_PROT_
                          SCIF_MAP_
        …
 }
 printf("registered buffers at ad
```

```
/* scif_mmap : maps pages in VAS starting at pa to remote window starting
  * at buffer.offset where pa is a function of buffer.self_addr & msg_size.
  * successful mapping returns pa, the address where mapping is placed
   */
 if ((buffer->peer_addr = scif_mmap(buffer->self_addr,
                          msg_size,
                          SCIF_PROT_READ | SCIF_PROT_WRITE,
                           SCIF_MAP_FIXED,
                           epd,
                           buffer->offset)) == MAP_FAILED) {
        …
  }
  else {
        printf("mapped buffers at address 0x%lx\n",
                           (unsigned long)buffer->peer_addr);
  }
 /* we know have buffer->peer_addr to read/write to – for e.g. memcpy() */
```

SCIF results: memcpy using scif_mmap() pointers – Xeon vs. Xeon Phi™ Coprocessor



SCIF results: memcpy using scif_mmap() pointers – Xeon vs. Xeon Phi™ Coprocessor



Remote Memory Access

- One-sided communication model
  - Initiator must know the source and destination “address”
  - RMAs are performed in the context of a connection specified by a local endpoint
  - Remote address range is always within a registered window of the peer endpoint
  - Local address can be in local endpoint’s registered window or a virtual address range

- Supports DMA or CPU based (memcpy) transfers (is there a cross over chart?)
  - DMA transfers definitely faster for large transfers
  - CPU based transfers may be faster for small transfers

- New flags
  - SCIF_RMA_USECACHE
  - SCIF_RMA_SYNC
  - SCIF_RMA_ORDERED

SCIF API: Remote Memory Access and Synchronization

- int **scif_readfrom**(scif_epd_t epd, off_t loffset, size_t len, off_t roffset, int rma_flags);
- int **scif_writeto**(scif_epd_t epd, off_t loffset, size_t len, off_t roffset, int rma_flags);
- int **scif_vreadfrom**(scif_epd_t epd, off_t *addr, size_t len, off_t roffset, int rma_flags);
- int **scif_vwriteto**(scif_epd_t epd, off_t *addr, size_t len, off_t roffset, int rma_flags);

- int **scif_fence_mark**(scif_epd_t epd, int flags, int *mark);
- int **scif_fence_wait**(scif_epd_t epd, int mark);
- int **scif_fence_signal**(scif_epd_t epd, off_t loff, uint64_t lval, off_t roff, uint64_t rval, int flags)

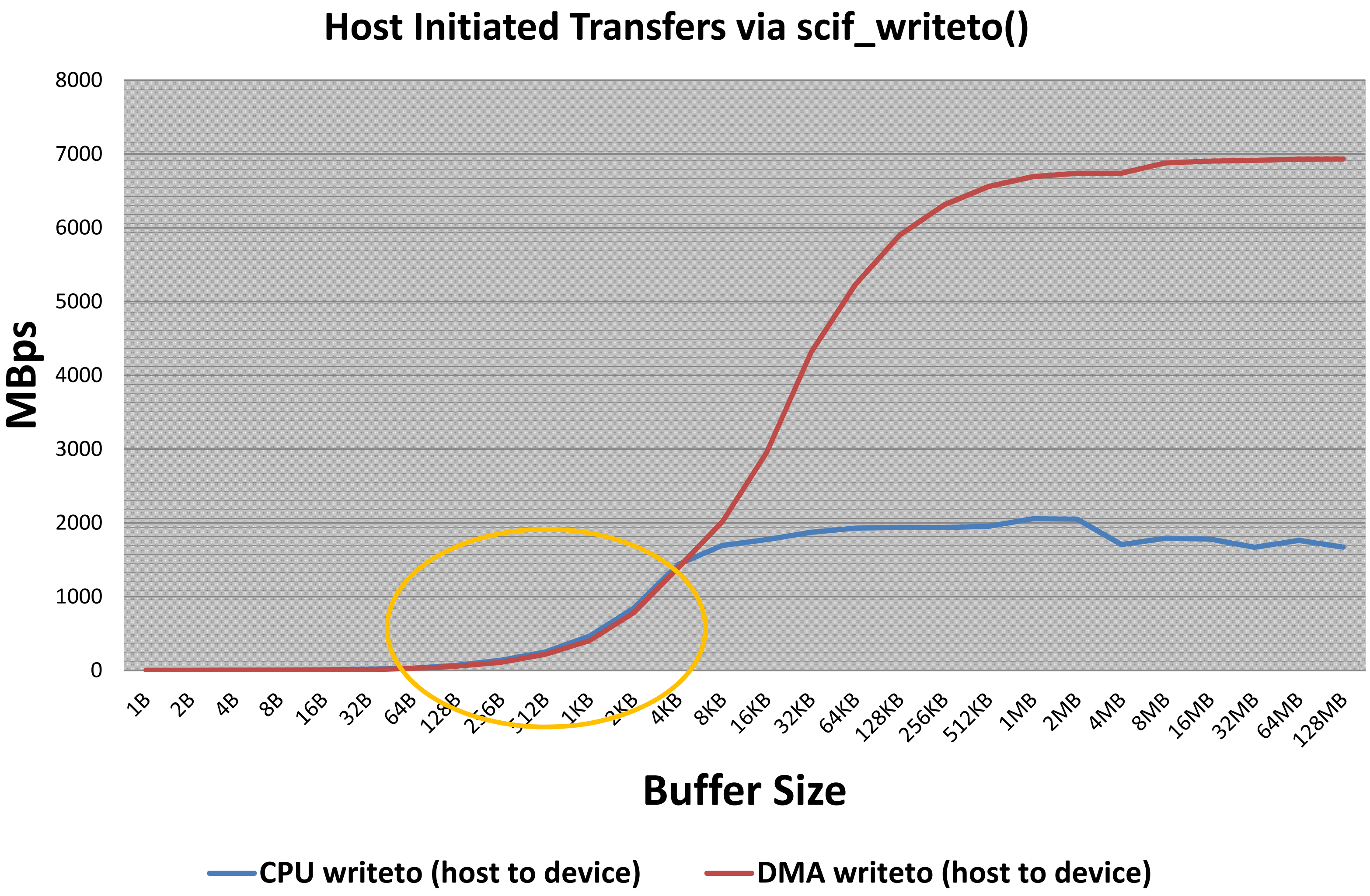
SCIF Example: Remote Memory Access

```
/* scif_vwriteto : copies msg_size bytes from local Virtual Addr Space to remote Registered Addr Space. */
if ((err = scif_vwriteto(epd,
                buffer.self_addr, /* local VAS addr */
                msg_size,
                remote_offset, /* remote RAS offfset */
                (use_cpu ? RMA_USECPU : 0) | SCIF_RMA_SYNC))) {
          …
}

/* scif_vreadfrom : copies msg_size bytes from remote Registered Addr Space to local Virtual Addr Space. */
if ((err = scif_vreadfrom(epd,
                buffer.self_addr, /* local VAS addr */
                msg_size,
                remote_offset, /* remote RAS offfset */
                (use_cpu ? RMA_USECPU : 0) | SCIF_RMA_SYNC))) {
     …
}
```

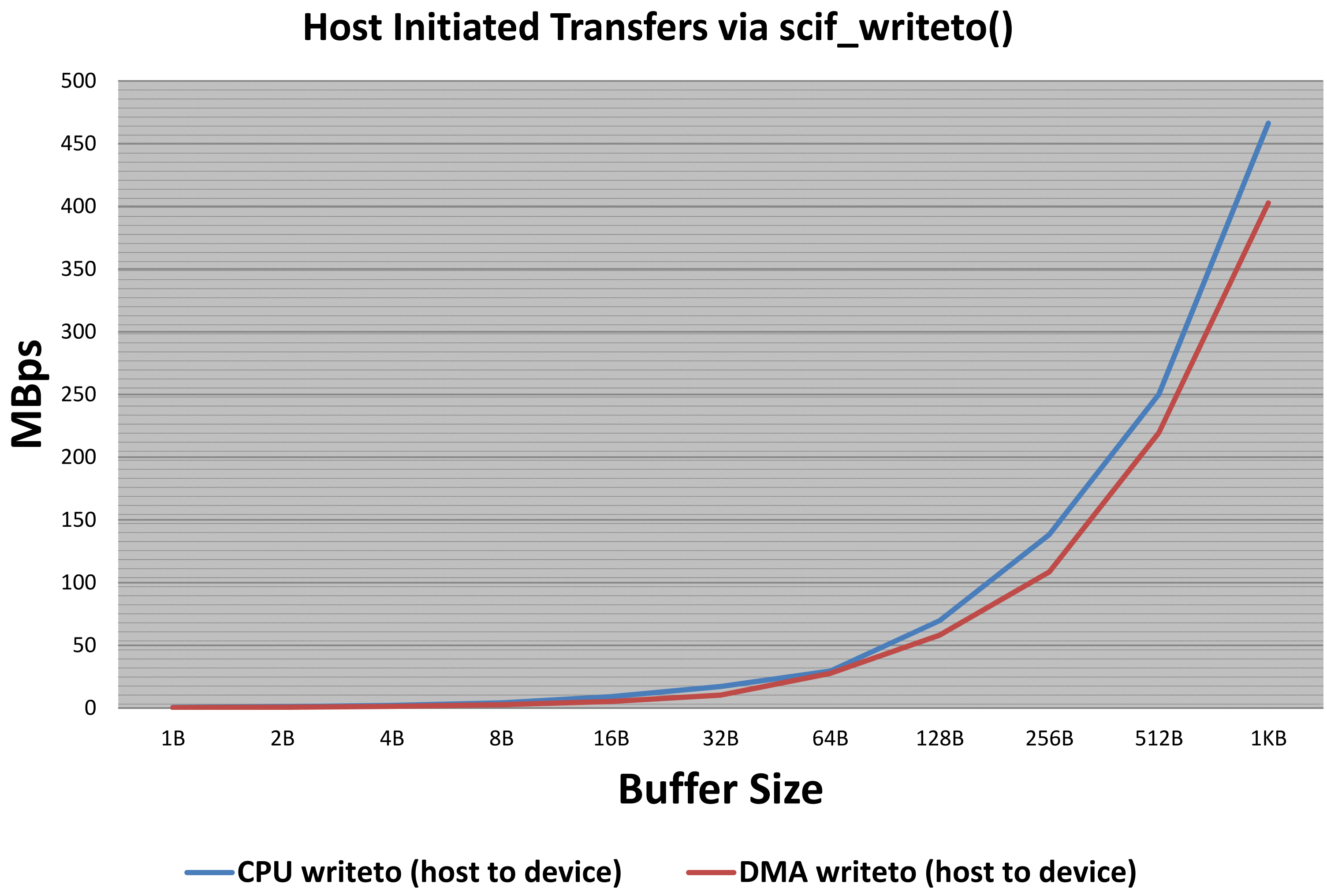
SCIF – Performance Considerations

- Choosing the right API
  - Simple messaging: scif_send()/recv(),
  - Bulk transfers: scif_(v)readfrom()/(v)writeto(),
  - Low latency paths from ring3: scif_mmap()
- How do you want to move your bytes? DMA vs. CPU
  - DMA is good for large buffers
  - Cost of programming DMA transfers + ring transition might be too high for small buffers – use CPU or scif_mmap()
- Where do you want to initiate the transfer from? Host vs. Card
  - Programming DMA engine is efficient from the host because single threaded perf of Xeon is higher
- Lastly, buffer *Alignment* matters for buffer transfers to/from Intel® Xeon Phi™ Coprocessor
- Internal SCIF optimization: Registration Caching
  - scif_vreadfrom()/scif_vwriteto() implicitly register regions, but registration is expensive
  - We avoid re-registration over and over again by “caching” the (va, len)

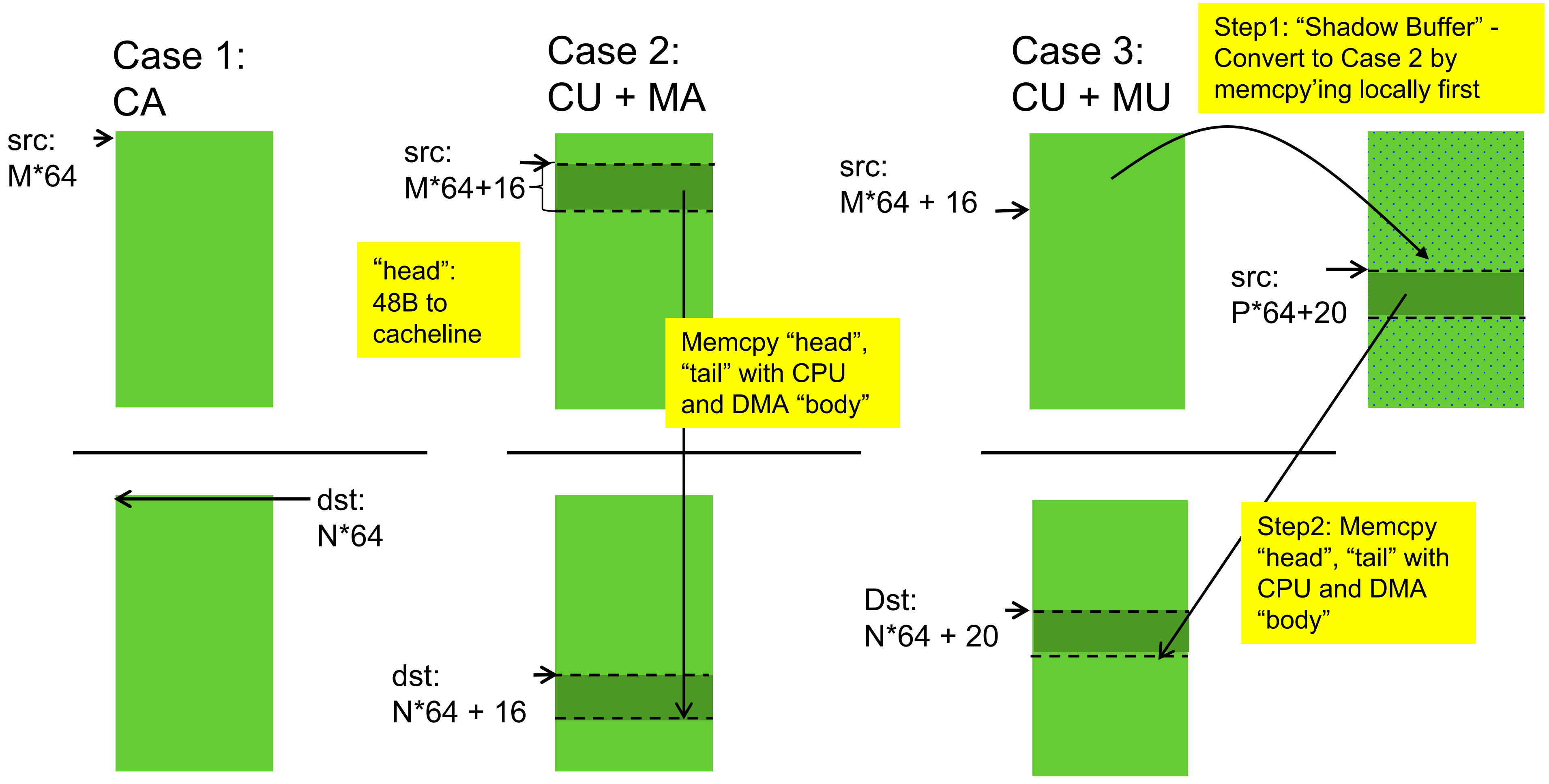
SCIF Results – Use DMA or CPU?



SCIF Results – Use DMA or CPU? (2)



Moving Bytes with DMA



Case 1: CA

Case 2: CU + MA

Case 3: CU + MU

Step1: “Shadow Buffer” - Convert to Case 2 by memcpy’ing locally first

“head”: 48B to cacheline

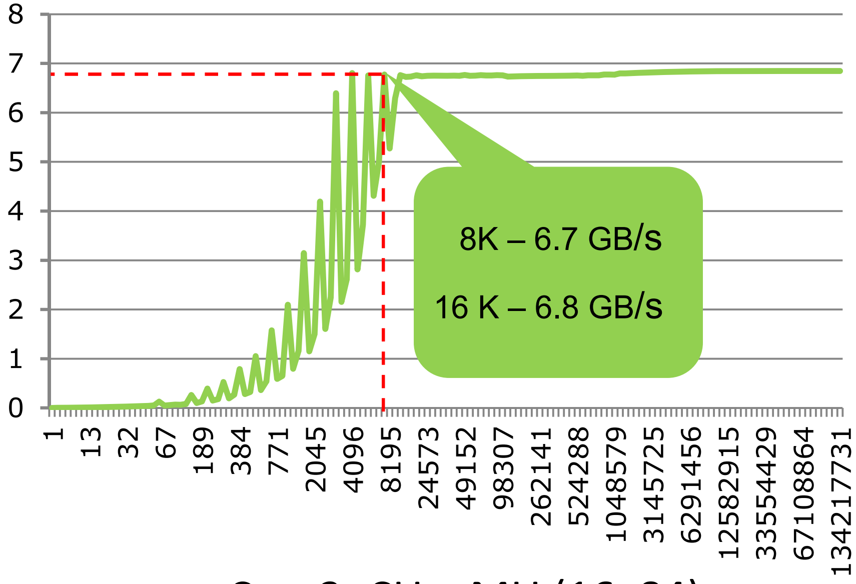
Memcpy “head”, “tail” with CPU and DMA “body”

Step2: Memcpy “head”, “tail” with CPU and DMA “body”

DMA engine can only move 64B cachelines

DMA alignment results

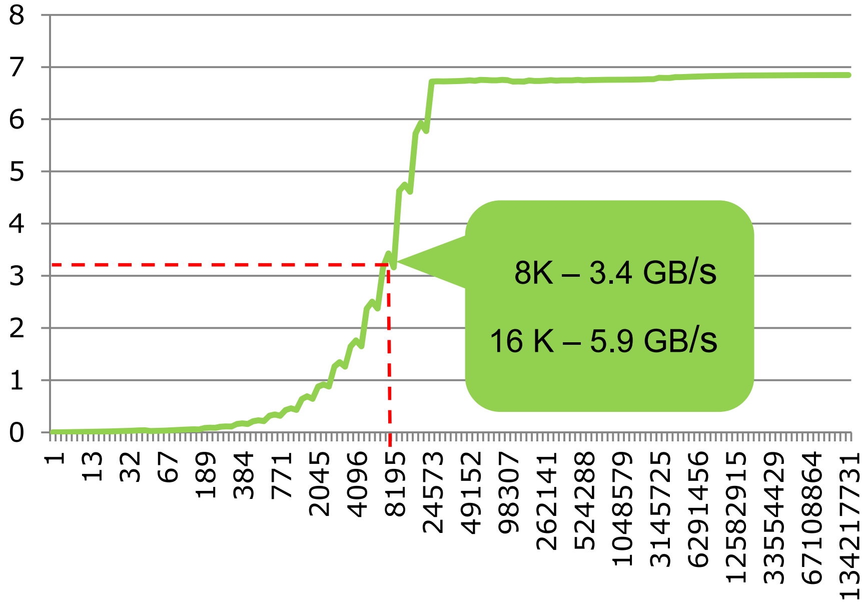
Case1: CA



8K – 6.7 GB/s

16 K – 6.8 GB/s

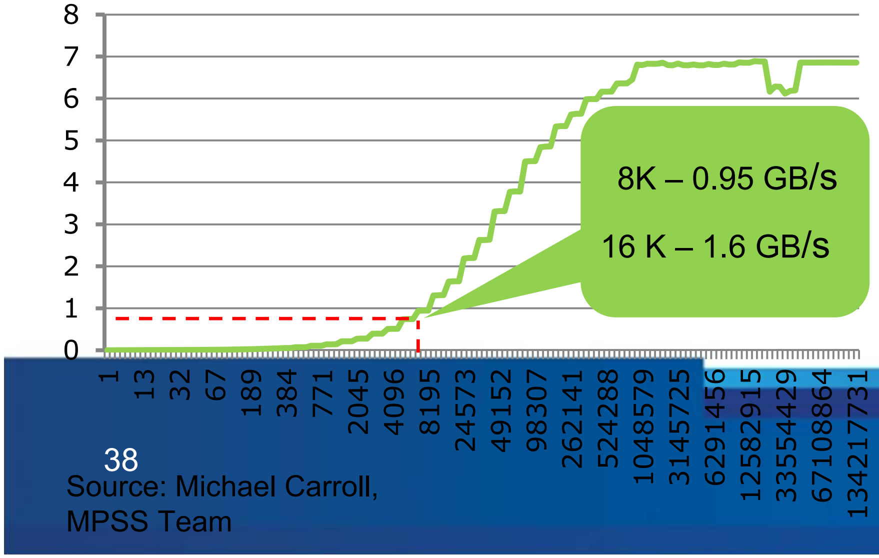
Case2: CU + MA (16, 272)



8K – 3.4 GB/s

16 K – 5.9 GB/s

Case3: CU + MU (16, 84)



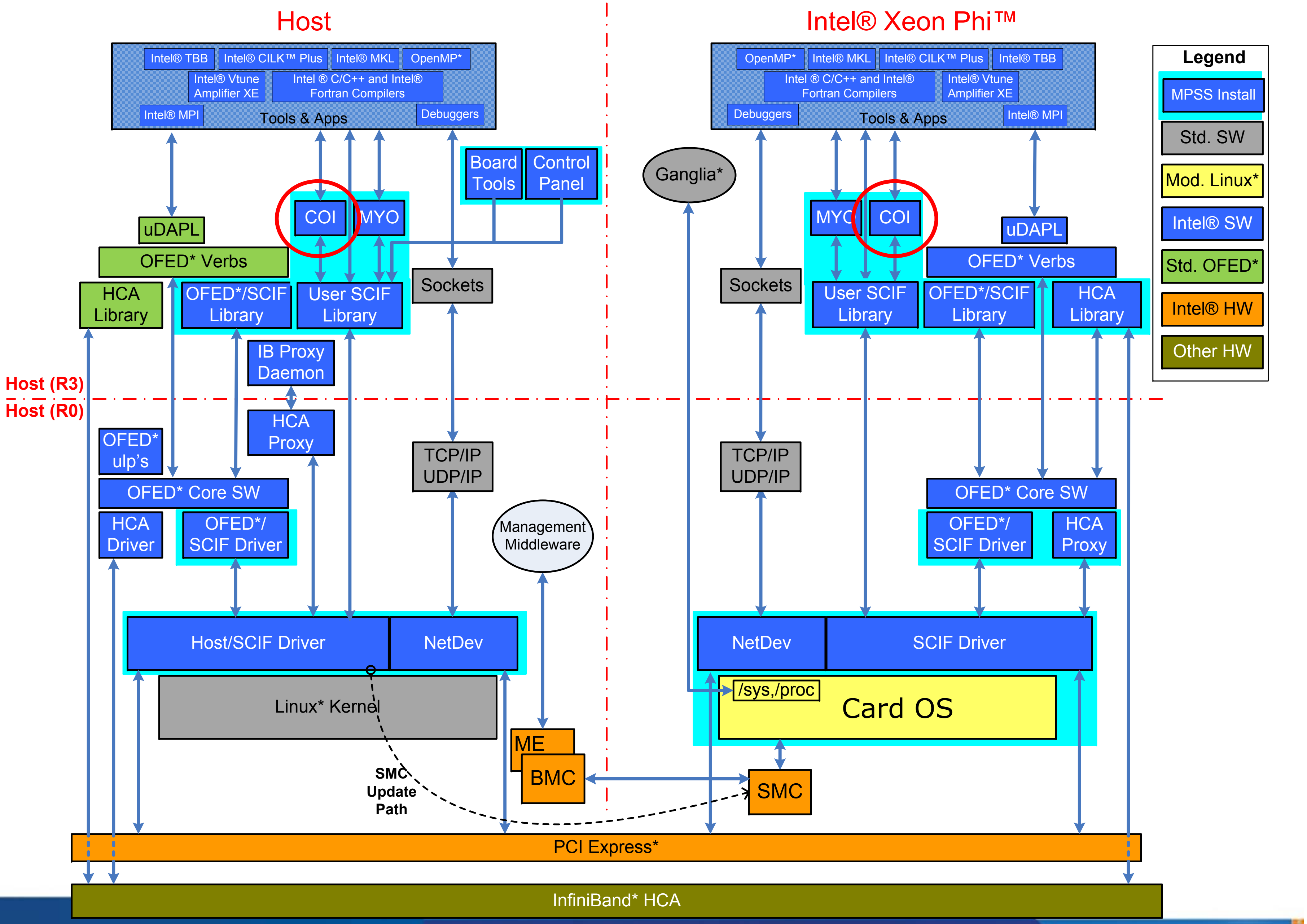
8K – 0.95 GB/s

16 K – 1.6 GB/s

(x, y) – x = source offset, y = destination offset
CA – Cacheline Aligned
CU – Not Cacheline Aligned
MA – Mutually Aligned between source and destination buffers
MU – Mutually Mis-aligned between source and destination buffers

Source: Michael Carroll, MPSS Team

Coprocessor Offload Infrastructure (COI)


Host
Intel® Xeon Phi™
Intel® TBB
Intel® CILK™ Plus
Intel® MKL
OpenMP*
Intel® Vtune Amplifier XE
Intel ® C/C++ and Intel® Fortran Compilers
Intel® MPI
Tools & Apps
Debuggers
OpenMP*
Intel® MKL
Intel® CILK™ Plus
Intel® TBB
Intel ® C/C++ and Intel® Fortran Compilers
Intel® Vtune Amplifier XE
Debuggers
Tools & Apps
Intel® MPI
Board Tools
Control Panel
Ganglia*
uDAPL
COI
MYO
OFED* Verbs
HCA Library
OFED*/SCIF Library
User SCIF Library
Sockets
IB Proxy Daemon
Host (R3)
Host (R0)
HCA Proxy
OFED* ulp's
OFED* Core SW
HCA Driver
OFED*/ SCIF Driver
TCP/IP UDP/IP
Management Middleware
Host/SCIF Driver
NetDev
Linux* Kernel
ME
BMC
SMC Update Path
MYO
COI
uDAPL
OFED* Verbs
Sockets
User SCIF Library
OFED*/SCIF Library
HCA Library
TCP/IP UDP/IP
OFED* Core SW
OFED*/ SCIF Driver
HCA Proxy
NetDev
SCIF Driver
/sys,/proc
Card OS
SMC
PCI Express*
InfiniBand* HCA
Legend
MPSS Install
Std. SW
Mod. Linux*
Intel® SW
Std. OFED*
Intel® HW
Other HW

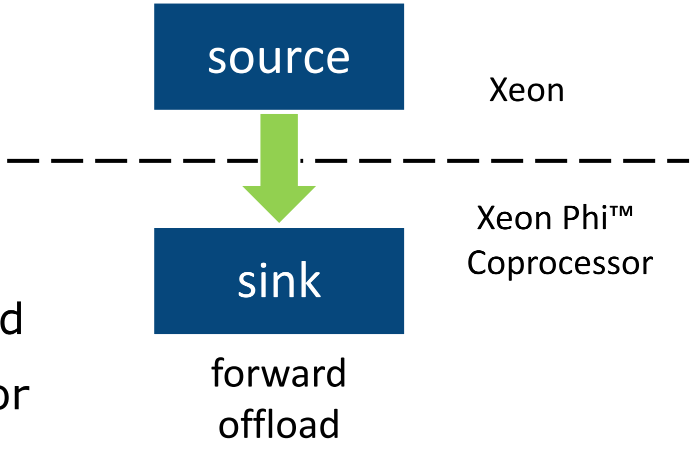
COI Introduction

- COI provides a set of APIs to simplify development for tools/apps using offload accelerator models
  - Simplifies running application on the Intel® Xeon Phi™ Coprocessor
    - Loading and launching device code without needing SSH passwords, NFS mounts, etc.
  - Simplifies asynchronous execution and data transfer
    - Can set up dependencies between asynchronous code execution and data movement
    - Device parallelism keeps the host, DMA engines and Phi device busy at the same time
  - Simplifies Resource Management
    - Automatically manages buffer space by reserving memory and evicting data as needed
  - Simplest way to get the best performance
    - COI includes features such as pinned buffers and same address buffers which make it easy to offload existing applications

COI Terminology

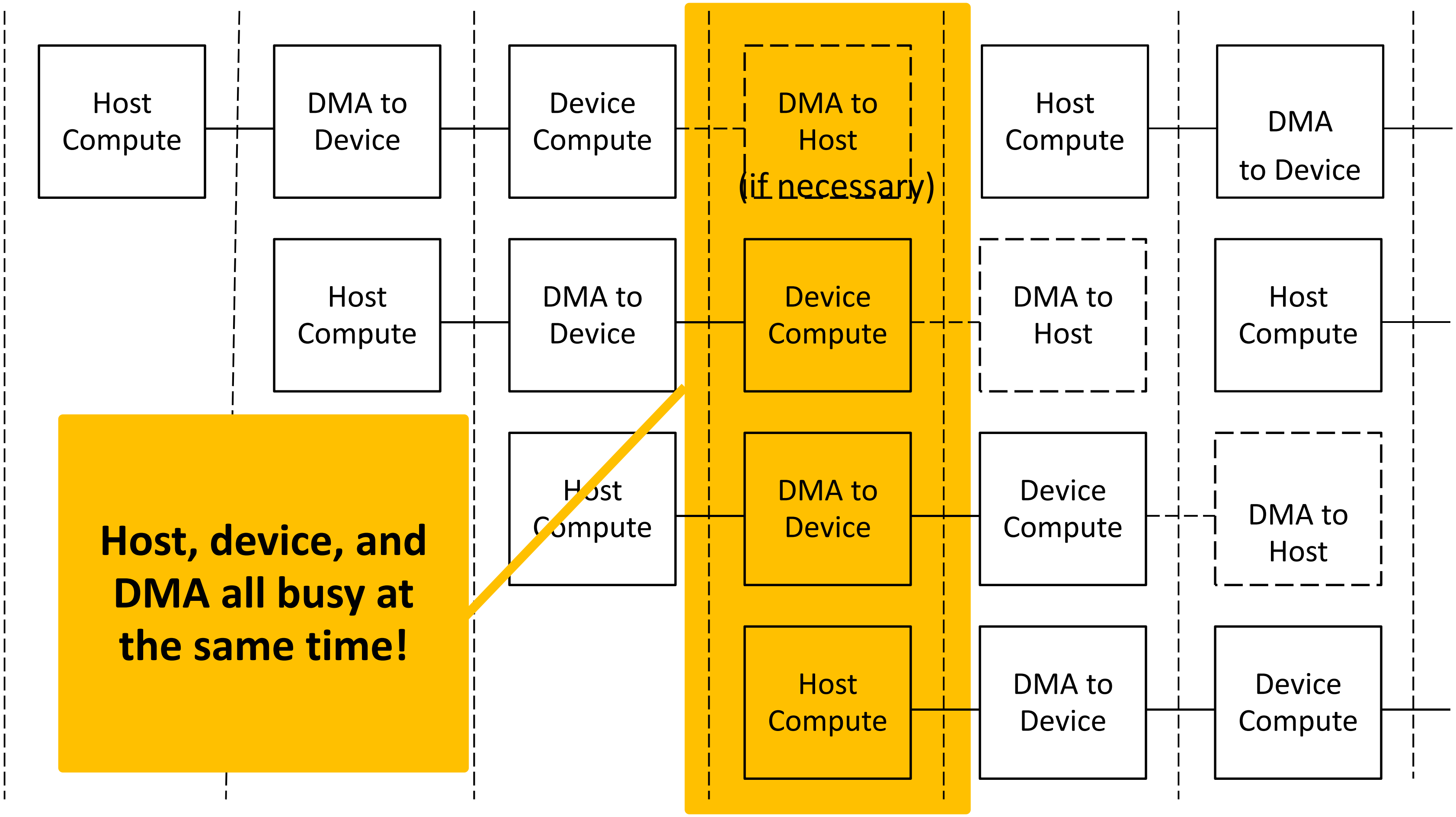
- COI allows commands to be sent from a “source” to a “sink”
  - Commands are asynchronous function invocations (“run functions”)
  - “Source” is where “run functions” are *initiated*
  - “Sink” is where “run functions” are *executed*

- A typical COI application is comprised of a source application and a sink offload *binary*
- The sink binary is a complete executable
  - Not just a shared library
  - Starts executing from main when it is loaded
- COI automatically loads dependent libraries prior to starting the offload binary on the sink
- COI has a *coi_daemon* that spawns sink processes and waits for them to exit



Host and Device Parallelism with COI

This of the instruction execution pipeline - Hennessy and Patterson



Host Compute — DMA to Device — Device Compute — DMA to Host (if necessary) — Host Compute — DMA to Device

Host Compute — DMA to Device — Device Compute — DMA to Host — Host Compute

Host Compute — DMA to Device — Device Compute — DMA to Host

Host Compute — DMA to Device — Device Compute

Host, device, and DMA all busy at the same time!

COI APIs – First glance

- COI exposes four major abstractions:
  - Use the simplest layer or add additional capabilities with more layers as needed
  - Each layer intended to interoperate with other available lower layers (e.g. SCIF)
- Enumeration: **COIEngine**, **COISysInfo**
  - Enumerate HW info; cards, APIC, cores, threads, caches, dynamic utilization
- Process Management: **COIProcess** (requires COIEngine)
  - Create remote processes; loads code and libraries, start/stop
- Execution Flow: **COIPipeline** (requires COIProcess)
  - COIPipelines are the RPC-like mechanism for flow control and remote execution
  - Can pass up to 32K of data with local pointers
- Data and Dependency Management: **COIBuffer**, **COIEvent** (requires COIPipeline)
  - COIBuffers are the basic unit of data movement and dependence managment
  - COIEvent optionally used to help manage dependences
  - COIBuffers and COIEvents are typically used with Run Functions executing on COIPipelines

API: COIEngine

- Abstracts the devices in the system
  - Host x86_64 device as well as Phi cards
- Provides device enumeration capabilities
  - Numbe
- Also provid
  - Current
- Does not re

```
/* Get the number of engines */
uint32_t    num_engines = 0;
COIEngineGetCount(COI_ISA_MIC,          /* The type of engine. */
                          &num_engines);    /* Outparam for number of engines */
                                            /* of this type. */

/* Get a handle to the first one */
COIENGINE engine;
COIEngineGetHandle(COI_ISA_MIC,       /* The type of engine.
                              0,              /* The index of the engine within
                                               * the array of engines of this type
                                               */
                         &engine);        /* Outparam for the engine handle */
```

API: COIProcess

```
/* Create a process */
COIPROCESS process;
COIProcessCreateFromFile(engine,
                         "sink_exe",
                         0, NULL,
                         false, NULL,
                         false, NULL,
                         1024 * 102
                          NULL,
                          &process);
// Get a handle to a function
COIFUNCTION function;
COIProcessGetFunctionHandles(proc
                                1,
              (const char*[]){"sink_fn"},    /* Name of function for which we
                                               * want a handle.
                                               */
                             &function);    /* Outparam for the function handle */
int8_t sink_return;
COI_SHUTDOWN_REASON exit_reason;
COIProcessDestroy(process,      /* Already created process */
                  -1,           /* Wait forever for the process to exit */
                   false,       /* Don't forcibly kill the process */
              &sink_return,  /* Get the return value from main */
              &exit_reason); /* Find out why the process exited */
```

```
int main(int argc, char** argv)
{
   COIPipelineStartExecutingRunFunctions();    /* Start executing run functions */
   COIProcessWaitForShutdown();                /* Wait for the "source" to call
                                                 * to call COIProcessDestroy.
                                                 */
   return 0;
}
void sink_fn(uint32_t, void**, uint64_t*, void*,
          uint16_t, void*, uint16_t)
{
}
```

es

ibraryFromFile,

API – COIPipeline

- With COIPipeline you can:

```
/* Call a run function in the simplest way possible */
COIFUNCTION function;
COIEVENT    event;
char* data = “Hello world!”;
COIPipelineRunFunction(pipeline,          /* The pipeline on v
                        function,         /* The handle of th
                        0, NULL, NULL,    /* No buffers this ti
                        0, NULL,          /* No dependencie
                        strlen(data) + 1, /* Small input data
                         NULL, 0,         /* Small output data
                         &event);         /* Completion event. */

/* Wait for the function to finish */
COIEventWait(1,           /* Number of events to wait for. */
             &event,  /* The completion event from the run function */
              -1,         /* Wait forever. */
             true,      /* Wait for all events to signal */
            NULL,       /* Only waiting for one event */
             NULL);
```

```
void sink_fn(
        uint32_t      in_BufferCount,
        void**        in_ppBufferPointers,
        uint64_t*     in_pBufferLengths,
        void*         in_pMiscData,
        uint16_t      in_MiscDataLength,
        void*         in_pReturnValue,
        uint16_t      in_ReturnValueLength)
{
  printf(“%s\n”, (char*)in_pMiscData);
}
```

API - COIBuffer

```
/* Create a normal buffer */
COIBUFFER buffer;
COIBufferCreate(1024 * 1024,                /* Size of the buffer */
                    COI_BUFFER_NORMAL,  /* Buffer type. */
                     0,
                     NULL,
                    1, &process,
                    &buffer);
/* Map the buffer */
char* data; COIMAPINSTANCE m
COIBufferMap(buffer,
                  0, 0,
                  COI_MAP_READ_
                  0, NULL,
                  NULL,
                  &mi,
                  (void**)&data);
sprintf(data, “Hello world!\n”);  /* fill some data */
COIBufferUnmap(mi);  /* done with it, unmap the buffer */

COI_ACCESS_FLAGS flags = COI_SINK_READ;
COIPipelineRunFunction(pipeline, function,
                            1,       /* One buffer. */
                            &buffer,  /* Here’s the buffer. */
                           &flags,   /* Buffer flags. */
                            0, NULL, NULL, 0, NULL, 0, NULL);
```

```
void sink_fn(
        uint32_t      in_BufferCount,
        void**        in_ppBufferPointers,
        uint64_t*     in_pBufferLengths,
        void*         in_pMiscData,
        uint16_t      in_MiscDataLength,
        void*         in_pReturnValue,
        uint16_t      in_ReturnValueLength)
{
   printf((char*)(in_ppBufferPointers[0]));  /* Print the data in the */
                                            /* first buffer. */
}
```

ead directly using

n

ateFromMemory,

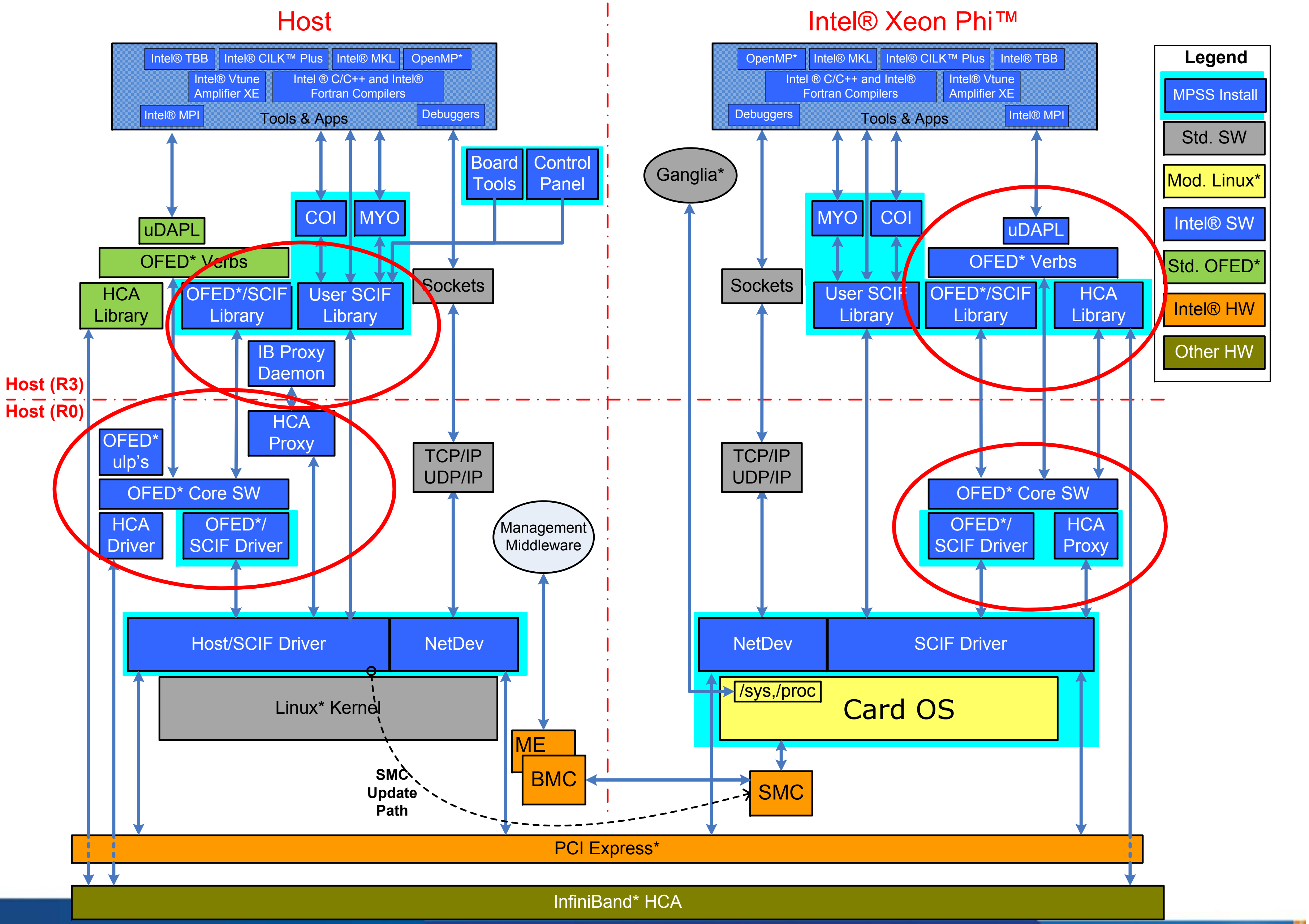
rUnmap,

API - COIEvent

- Used to explicitly manage dependencies in the system
  - Events can be passed as input dependencies to *many* COI APIs
  -
- U
- C
  a
- W
  b
- Wi
  -
  -
  -

```
COIPipelineRunFunction (pipeline1,          /* example with 2 pipelines to the same card */
                         function1,
                         0, NULL, NULL,   /* buffer-related params */
                         0, NULL,           /*dependency params */
                         NULL, 0,           /* misc input data params */
                         0, NULL,           /* return value params */
                         &event1);        /* returns a completion event */

COIPipelineRunFunction (pipeline2,
                         function2,
                        0, NULL, NULL,  /* buffer-related params */
                        1, {event1},      /* 1 dependency in the array */
                                          /* of dependencies */
                        NULL, 0,          /* misc input data params */
                       0, NULL,           /* return value params */
                       &event2);

COIEventWait(1,                           /* wait on a single event */
              &event2,                    /* array of events to wait on */
             -1, true,                    /* wait forever, on all events */
              NULL, NULL);                /* optional parameters */
```

Coprocessor Communication Link (CCL)



Host

Intel® TBB
Intel® CILK™ Plus
Intel® MKL
OpenMP*
Intel® Vtune Amplifier XE
Intel ® C/C++ and Intel® Fortran Compilers
Intel® MPI
Tools & Apps
Debuggers

Board Tools
Control Panel
COI
MYO
uDAPL
OFED* Verbs
HCA Library
OFED*/SCIF Library
User SCIF Library
Sockets
IB Proxy Daemon

Host (R3)
Host (R0)

HCA Proxy
OFED* ulp's
OFED* Core SW
HCA Driver
OFED*/ SCIF Driver
TCP/IP UDP/IP
Management Middleware
Host/SCIF Driver
NetDev
Linux* Kernel
ME
BMC
SMC Update Path

Intel® Xeon Phi™

OpenMP*
Intel® MKL
Intel® CILK™ Plus
Intel® TBB
Intel ® C/C++ and Intel® Fortran Compilers
Intel® Vtune Amplifier XE
Debuggers
Tools & Apps
Intel® MPI

Ganglia*
MYO
COI
uDAPL
OFED* Verbs
Sockets
User SCIF Library
OFED*/SCIF Library
HCA Library
TCP/IP UDP/IP
OFED* Core SW
OFED*/ SCIF Driver
HCA Proxy
NetDev
SCIF Driver
/sys,/proc
Card OS
SMC

PCI Express*

InfiniBand* HCA

Legend

MPSS Install

Std. SW

Mod. Linux*

Intel® SW

Std. OFED*

Intel® HW

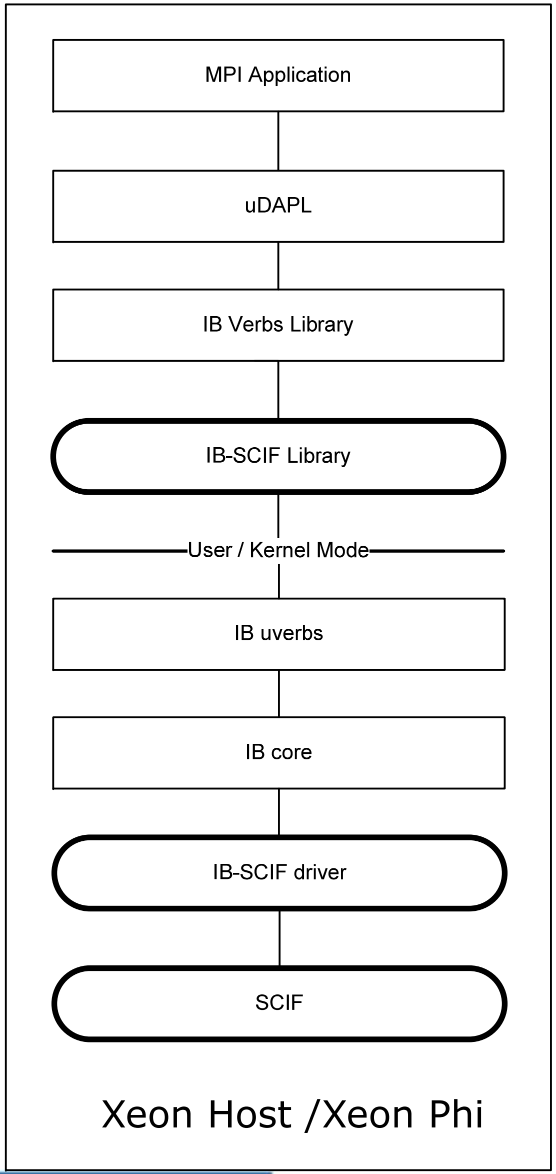
Other HW

Introduction

- OFED is the industry standard code used for messaging on high-end HPC clusters
  - Supports Intel MPI and all open source MPIs
  - Is in Linux and all the various Linux distributions
- RDMA over SCIF (IB-SCIF) – RDMA within the platform between the host and KNC or multiple KNCs
- Intel ® Xeon Phi ™ Coprocessor Communication Link (CCL) Direct
  - Direct access to InfiniBand HCA from Intel® Xeon Phi ™
  - Lowest latency data path
- Intel ® Xeon Phi ™ Coprocessor Communication Link (CCL) Proxy
  - Pipeline data through host memory to InfiniBand network
  - Higher bandwidth data path for some platform configurations
- Intel MPI dual-DAPL support
  - Uses best data path, direct path for small messages, and proxy path for large messages for best overall MPI performance

RDMA over IB-SCIF

- OFED for Intel® Xeon Phi™ Coprocessor uses the core OFA software modules from the Open Fabrics Alliance
- IB-SCIF is a new hardware specific driver and library that plugs into the OFED core mid-layer
  - SCIF is the lowest level in the SW stack as we saw earlier
  - Provides standard RDMA verbs interfaces within the platform, i.e., between the Intel® Xeon™ and Intel® Xeon Phi ™ Coprocessor cards within the same system.
  - IBSCIF 1 byte latency is in the 13us range, (host-KNC), peak BW is in the 6GB/s per sec. range



Intel® Xeon Phi ™ Coprocessor CCL Direct Software

- CCL-Direct
  - Allows access to an HCA directly from the Xeon Phi™ Coprocessor using standard OFED interfaces using PCI-E peer-to-peer transactions
  - Provides the lowest latency data path
  - For each hardware HCA, a unique vendor driver has to be developed.
    - e.g., mlx4, mthca, Intel® True Scale ™ hca etc
    - Currently support for Mellanox HCAs (mlx4) exists and is shipping in MPSS
    - Support for Intel® TrueScale™ InfiniBand NICs via PSM is under development, expected release in early 2013
- Implementation Limitations
  - Intel® Xeon Phi™ Coprocessor CCL Direct only supports user space clients, e.g. MPI
  - Peak bandwidth is limited on some platforms and configurations
  - CCL-Direct 1 byte latency is in the 2.5us range for Host-KNC, and 3.5-4us range for KNC-KNC across an InfiniBand HCA, peak BW varies depending on the Xeon platform (see later)

Intel® Xeon Phi™ Coprocessor CCL Proxy uDAPL provider

- Intel ® Xeon Phi™ Coprocessor CCL Proxy
  - A new OFED uDAPL provider client runs on the Intel® Xeon Phi™ Coprocessor and proxy daemon runs on the Intel® Xeon™ host.
  - The uDAPL client pipelines data from the Intel® Xeon Phi™ Coprocessor to the host memory using SCIF to the proxy daemon
  - The proxy daemon then pipelines the data out the InfiniBand HCA to remote Intel® Xeon™ or Intel® Xeon Phi™ Coprocessor cards.
  - Will be shipped as an experimental feature in MPSS in early 2013 (the next release)
- This is the best interface for getting high bandwidth on some platforms and configurations
  - CCL-Proxy 1 byte latency is ~29us Host-KNC, and ~38us for KNC-KNC across an InfiniBand fabric, peak BW is ~2.5-3Gbyte/sec unidirectional

Intel® Xeon Phi™ Coprocessor and HCA PCIe considerations

- KNC and HCA PCI-E topology considerations
  - Best performance is achieved when the HCA and the KNC are located in PCIe slots that are on the same CPU socket.
  - Cross socket performance KNC-KNC or KNC-HCA is not recommended as the PCIe peer-to-peer performance is less optimized in this path.
    - Only applies to CCL-Direct and KNC-KNC direct communication
      - Bandwidth can be limited to a few hundred Mbytes/sec in this configuration.
      - Small message latency is not effected.
    - Offload mode of computing (using the Offload compiler and/or COI) is not limited since it does not use PCI-E peer-to-peer between the host and the KNC.

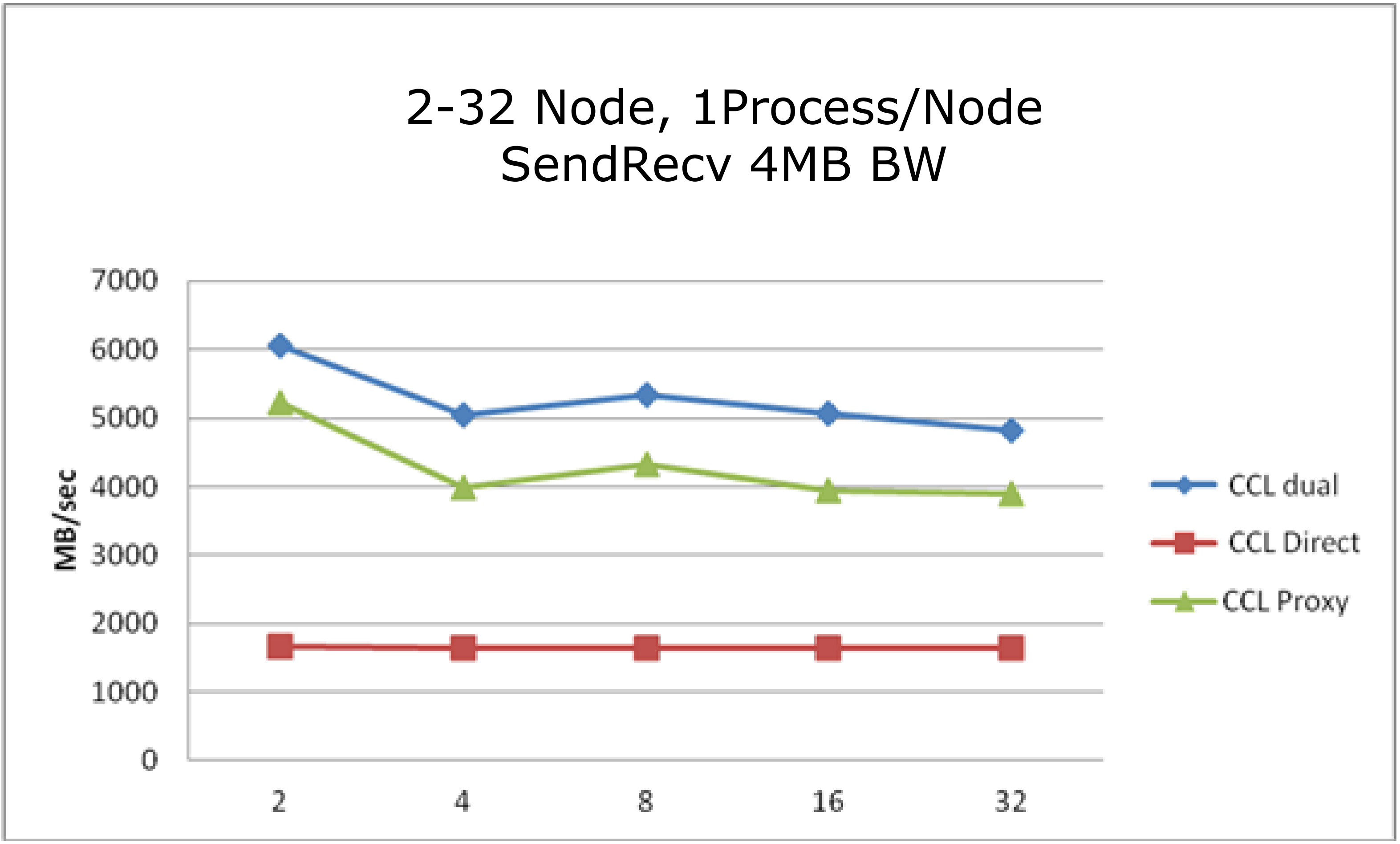
Intel MPI Dual DAPL support

- The Intel® Xeon Phi™ Coprocessor CCL Direct data path provides low latency and the Intel® Xeon Phi™ CCL Proxy data path provides high bandwidth.
- Intel MPI has a new feature called dual-DAPL.
  - MPI dual-DAPL sends small message down the Intel® Xeon Phi™ Coprocessor CCL Direct path and large messages down the Intel® Xeon Phi™ Coprocessor CCL Proxy path
  - This allows native Intel® Xeon Phi™ Coprocessor MPI applications to get both low latency and high bandwidth
  - This MPI feature is currently an experimental prototype but is expected to be available in a future Intel MPI release in 2013.
- Other MPIs could use this same technique to achieve both low latency and high bandwidth on InfiniBand from Intel® Xeon Phi™ Coprocessor platforms

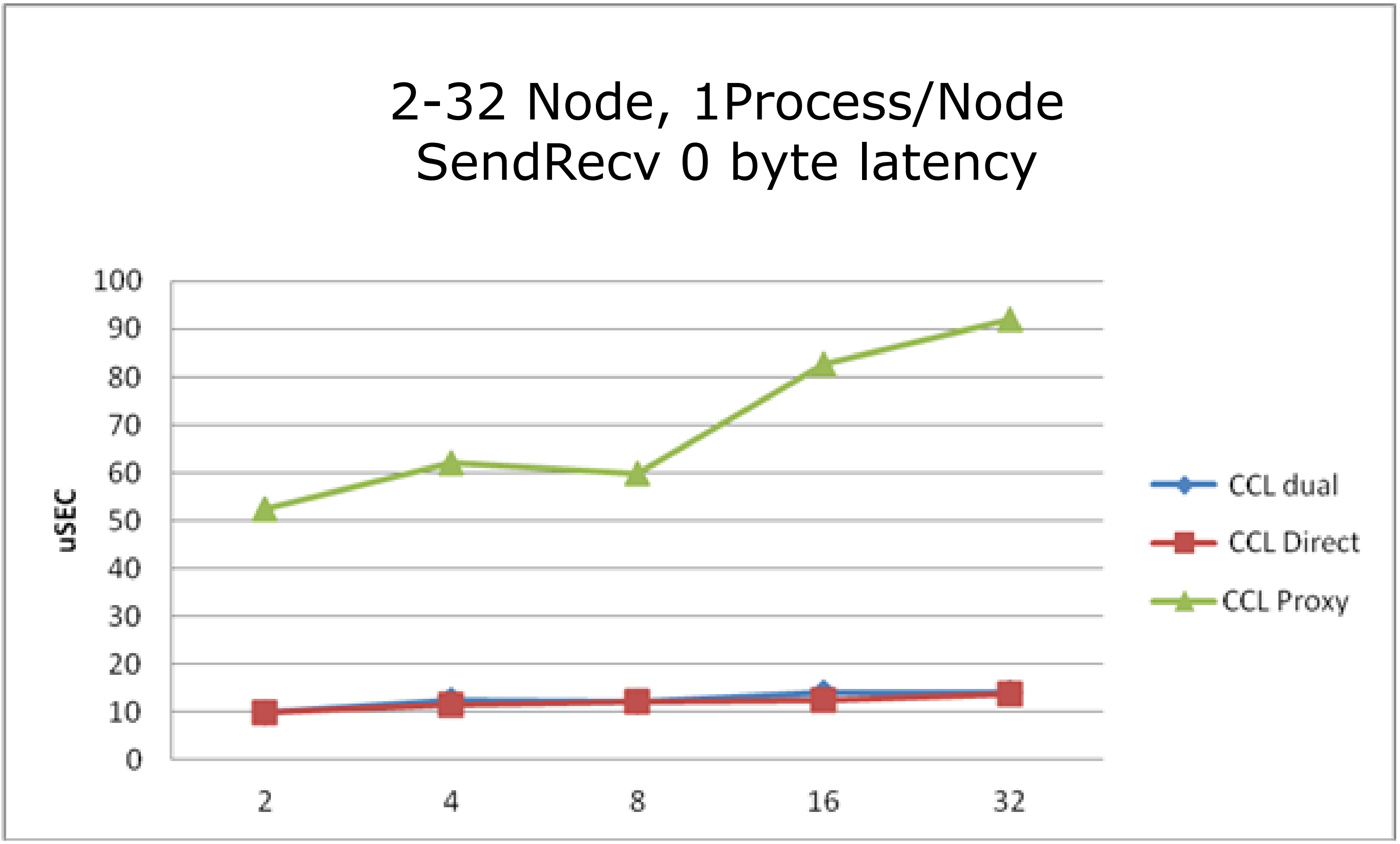
Intel® Xeon Phi™ Coprocessor CCL Direct and CCL Proxy Performance Results

- Platform Configuration
  - 32-node Intel internal cluster (Apollo)
  - Intel® Xeon™ CPU E5-2680 0 @ 2.70GHz
  - 64Gbytes memory
  - RHEL EL 6.2
  - Mellanox FDR 56Gbits (MT_1100120019), F/W - 2.10.700, 2K MTU
  - KNC B1 – 1Ghz
  - MPSS 2.1 Gold Release
  - Intel MPI 4.1 with dual-DAPL prototype
  - Intel ® Xeon Phi™ Coprocessor CCL Proxy prototype uDAPL

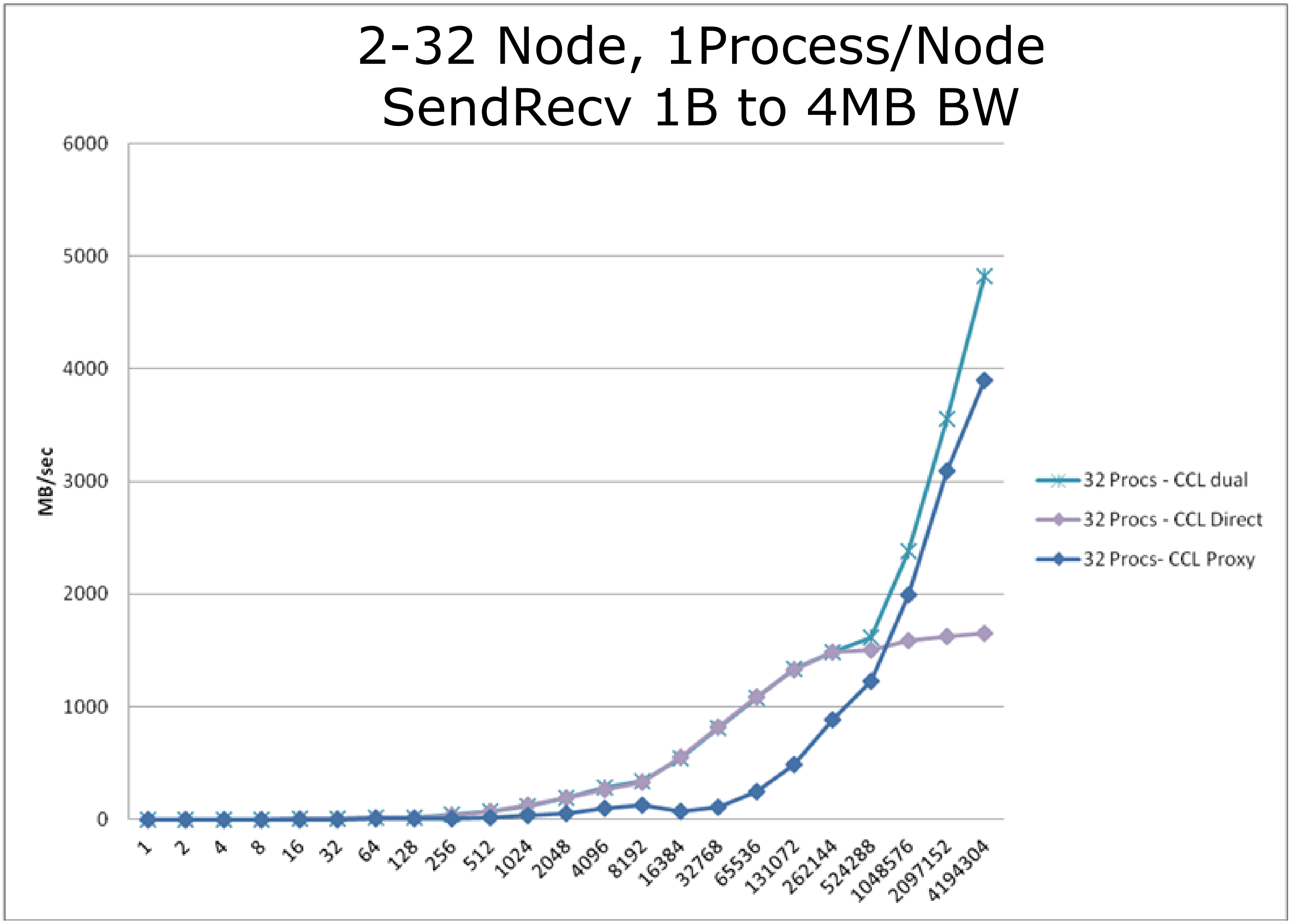
Large Message Bandwidth of Direct, Proxy and MPI Dual DAPL paths



Intel® Xeon Phi™ Coprocessor CCL Direct and CCL Proxy Performance Results



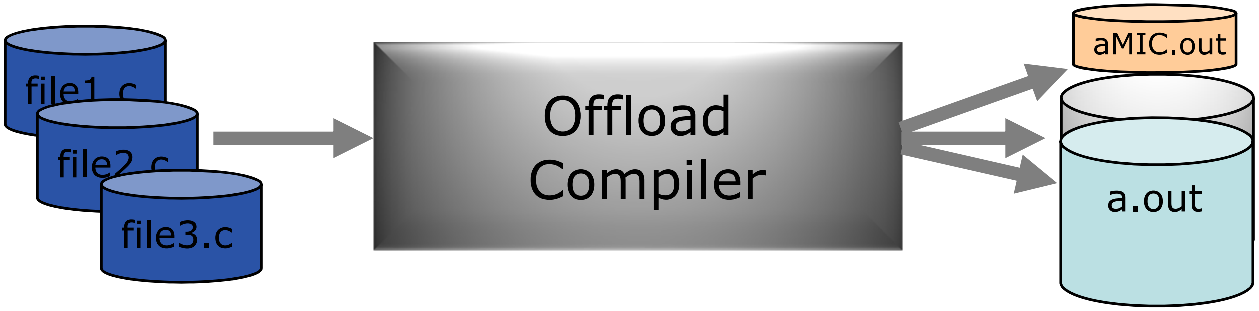
Bandwidth Curves of Direct, Proxy and MPI Dual-DAPL paths



Heterogeneous Programming with Offload

Heterogeneous Programming Model

- Programmer designates
  - Code sections to run on Intel® Xeon Phi™ Coprocessor
  - Explicit data sharing between Intel® Xeon host and Intel® Xeon Phi™ Coprocessor card
- Compiler with Runtime
  - Automatically manage setup/teardown, remote call, data transfer, synchronization
- Offload is *optional*
  - If MIC not present or busy, program runs entirely on CPU



Language Extensions for Offload (LEO)

- Offload pragma/directives
  - Provides offload capability with pragma/directive
  - #pragma offload   C/C++
  - !dir$ offload   Fortran

- Mark functions and variables for availability on MIC
  - __declspec (target (mic))
  - __attribute__ ((target (mic)))
  - #pragma offload_attribute(target (mic))

Heterogeneous Programming Model

Your Program

```
f()
{
    #pragma offload
        a = b + g();
}
```

```
__declspec (target (mic))
g()
{
}
```

```
h()
{
}
```

```
main()
{
    f();
    h();
}
```

Contents of Xeon Phi™ Program

```
f_part1()
{
        a = b + g();
}
```

```
g()
{
}
```

```
main
{
   // empty
}
```

So how does it work?

At first offload:

- if Xeon Phi™ Coprocessor is available
  - Xeon Phi™ Coprocessor program is loaded on card
  - Statement is executed
- else
  - Statement runs on CPU
- at program termination, MIC program is unloaded

Offload pragma/directives – C/C++

	C/C++ Syntax	Semantics
Offload pragma	`#pragma offload` *<clauses>* *<statement>*	Execute next statement on MIC (which could be an OpenMP parallel construct)

Function and variable	`__declspec ( target (mic))`
	<func/var>
	`__attribute__ (( target (mic))`
	<func/var>
	`#pragma offload_attribute (target (mic))`
	<func/var>
		Compile function and variable for CPU and MIC

Offload pragma/directives (2)

Variables restricted to scalars, structs, arrays and pointers to scalars/structs/arrays

Clauses	**Syntax**	**Semantics**
Target specification	`target ( mic [: <expr> ] )`	Where to run construct
If specifier	`if ( condition )`	Offload statement if condition is TRUE
Inputs	`in (var-list modifiers`$_{opt}$`)`	Copy CPU to target
Outputs	`out (var-list modifiers`$_{opt}$`)`	Copy target to CPU
Inputs & outputs	`inout (var-list modifiers`$_{opt}$`)`	Copy both ways
Non-copied data	`nocopy (var-list modifiers`$_{opt}$`)`	Data is local to target
Modifiers		
Specify pointer length	`length (element-count-expr)`	Copy that many pointer elements

Control pointer memory allocation	`alloc_if ( condition )`
	`free_if ( condition )`
		Allocate/free new block of memory for pointer if condition is TRUE

Alignment for pointer memory allocation	`align ( expression )`	Specify minimum data alignment

Offload Examples: OMP on Intel® Xeon Phi™ Coprocessor

C/C++ OpenMP

```
#pragma offload target (mic)
#pragma omp parallel for reduction(+:pi)
 for (i=0; i<count; i++)
 {
  float t = (float)((i+0.5)/count);
   pi += 4.0/(1.0+t*t);
 }
 pi /= count
```

Data Transfer Rules

- Automatically detected and transferred as INOUT
  - Named arrays in lexical scope
  - Scalars in lexical scope
- User can override automatic transfer with explicit IN/OUT/INOUT clauses
- Not automatically transferred
  - Memory pointed by pointers
  - Global variables used in functions called within the offloaded construct

Okay, so you’ve got this code …

```
int numFloats = 100;
float input1[100], input2[100];
float output[100];

main()
{
    read_input1(); read_input2();
    for(int j=0; j<numFloats; j++)
          {
            output[j] = input1[j] + input2[j];
          }
}
```

Offload it!

```
__declspec(target(mic)) int numFloats = 100;
__declspec(target(mic)) float input1[100], input2[100];
__declspec(target(mic)) float output[100];


main()
{
   read_input1(); read_input2();
   #pragma offload target(mic)
      for(int j=0; j<numFloats; j++) {
          output[j] = input1[j] + input2[j];
         }
}
```

It will work, but …

```
__declspec(target(mic)) int numFloats = 100;
__declspec(target(mic)) float input1[100], input2[100];
__declspec(target(mic)) float output[100];


main()
{
   read_input1(); read_input2();
   #pragma offload target(mic)
      for(int j=0; j<numFloats; j++) {
          output[j] = input1[j] + input2[j];
        }
}
```

What data is transferred ?

Optimal?

```
__declspec(target(mic)) int numFloats = 100;
__declspec(target(mic)) float input1[100], input2[100];
__declspec(target(mic)) float output[100];


main()
{
    read_input1(); read_input2();
    #pragma offload target(mic) \
            inout(input1, input2, output, numFloats)
       for(int j=0; j<numFloats; j++) {
            output[j] = input1[j] + input2[j];
          }
}
```

Is this optimal?

Optimize it a bit

```
__declspec(target(mic)) int numFloats = 100;
__declspec(target(mic)) float input1[100], input2[100];
__declspec(target(mic)) float output[100];


main()
{
   read_input1(); read_input2();
   #pragma offload target(mic) \
       in(input1, input2, numFloats) out (output)
     for(int j=0; j<numFloats; j++) {
         output[j] = input1[j] + input2[j];
       }
}
```

No!

Don’t need to send “output” to Intel® Xeon Phi™

Don’t need to get “input” from Intel® Xeon Phi™

Make it a function call

```
__declspec(target(mic)) int numFloats = 100;
__declspec(target(mic)) float input1[100], input2[100];
__declspec(target(mic)) float output[100];
__declspec(target(mic)) void real_work ()  {
     for(int j=0; j<numFloats; j++)  {
     output[j] = input1[j] + input2[j];
    }
}
main()
{
   read_input1(); read_input2();
   #pragma offload target(mic) in (input1, input2, numFloats) out (output)
          real_work ();
}
```

Globals referenced inside function on Intel® Xeon Phi™

Need for *alloc_if* and *free_if*

- Needed for pointers or allocatable arrays
  - Specify whether to allocate/free at each offload
  - Default is allocate/free at each offload
    - use free_if(1) to free memory
    - use alloc_if(1) to allocate memory
    - use free_if(0) to not free memory
    - use alloc_if(0) to not allocate memory
- Remember this was not needed for global allocated data
  - Data declared with __declspec (target(mic))
- Syntax:
  - #pragma offload in(myptr:length(n) alloc_if(expr))

Example usage scenarios

- For Readability you could define some macros
  - #define ALLOC alloc_if(1)
  - #define FREE free_if(1)
  - #define RETAIN free_if(0)
  - #define REUSE alloc_if(0)

  - Allocate and do not free

    ```
    #pragma offload target(mic) in (p:length(l) ALLOC RETAIN)
    ```
  - Reuse memory allocated above and do not free

    ```
    #pragma offload target(mic) in (p:length(l) REUSE RETAIN)
    ```
  - Reuse memory allocated above and free

    ```
    #pragma offload target(mic) in (p:length(l) REUSE FREE)
    ```

Example using malloc’d arrays

```
__declspec(target(mic)) float *input1, *input2, *output;
main() {
  input1 = malloc(1000); input2 = malloc(1000);
  output = malloc(1000);
  for (int i=0; i<10; i++) {
     read_input1(); read_input2();
     #pragma offload target(mic)
             in(input1:length(1000))
             in(input2:length(1000))
             out(output:length(1000))
     for(int j=0; j<1000; j++)
      output[j] = input1[j] + input2[j];
   }
}
```

What gets allocated/freed on Intel® Xeon Phi™

Why did my code just crash?

```
__declspec(target(mic)) float *input1, *input2, *output;
main() {
   input1 = malloc(1000); input2 = malloc(1000);
   output = malloc(1000);
   for (int i=0; i<10; i++) {
       read_input1(); read_input2();
       #pragma offload target(mic)
               in(input1:length(1000) alloc_if (i == 0))
               in(input2:length(1000) alloc_if (i == 0))
               out(output:length(1000) alloc_if (i == 0))
       for(int j=0; j<1000; j++)
        output[j] = input1[j] + input2[j];
    }
}
```

Sufficient?

It works! And it is efficient

```
__declspec(target(mic)) float *input1, *input2, *output;
main() {
   input1 = malloc(1000); input2 = malloc(1000);
   output = malloc(1000);
   for (int i=0; i<10; i++) {
       read_input1(); read_input2();
       #pragma offload target(mic)
              in(input1:length(1000) alloc_if (i == 0) free_if (i == 9))
              in(input2:length(1000) alloc_if (i == 0) free_if (i == 9))
              out(output:length(1000) alloc_if (i == 0) free_if (i == 9))
       for(int j=0; j<1000; j++)
        output[j] = input1[j] + input2[j];
    }
}
```

No, only free memory on the last loop!

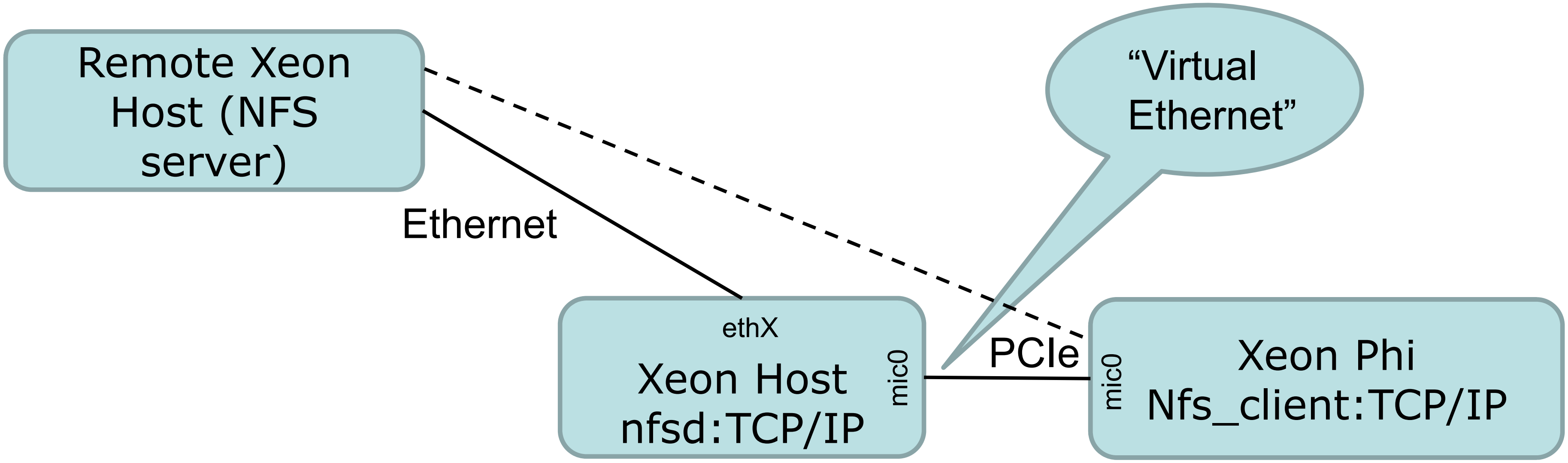
Demos (due to popular demand)

1. Code
   - SCIF sample using vreadfrom()
   - COI example to launch a program and have print something

2. simple_offload.c – move a buffer and execute some “stuff” on the Xeon Phi™ card
   - H_TRACE shows the output of the offload compiler – i.e. what it does
   - coitrace traces what COI APIs are used by the offload compiler.

3. omp.c – a simple example showing the use of OMP with offload
   - coitrace traces what COI APIs are used by the offload compiler

Demos (due to popular demand)



Your files on a remote server (or your local host) are accessible on Intel® Xeon Phi™

References:

[1] Why Huge Pages? - http://lwn.net/Articles/374424/

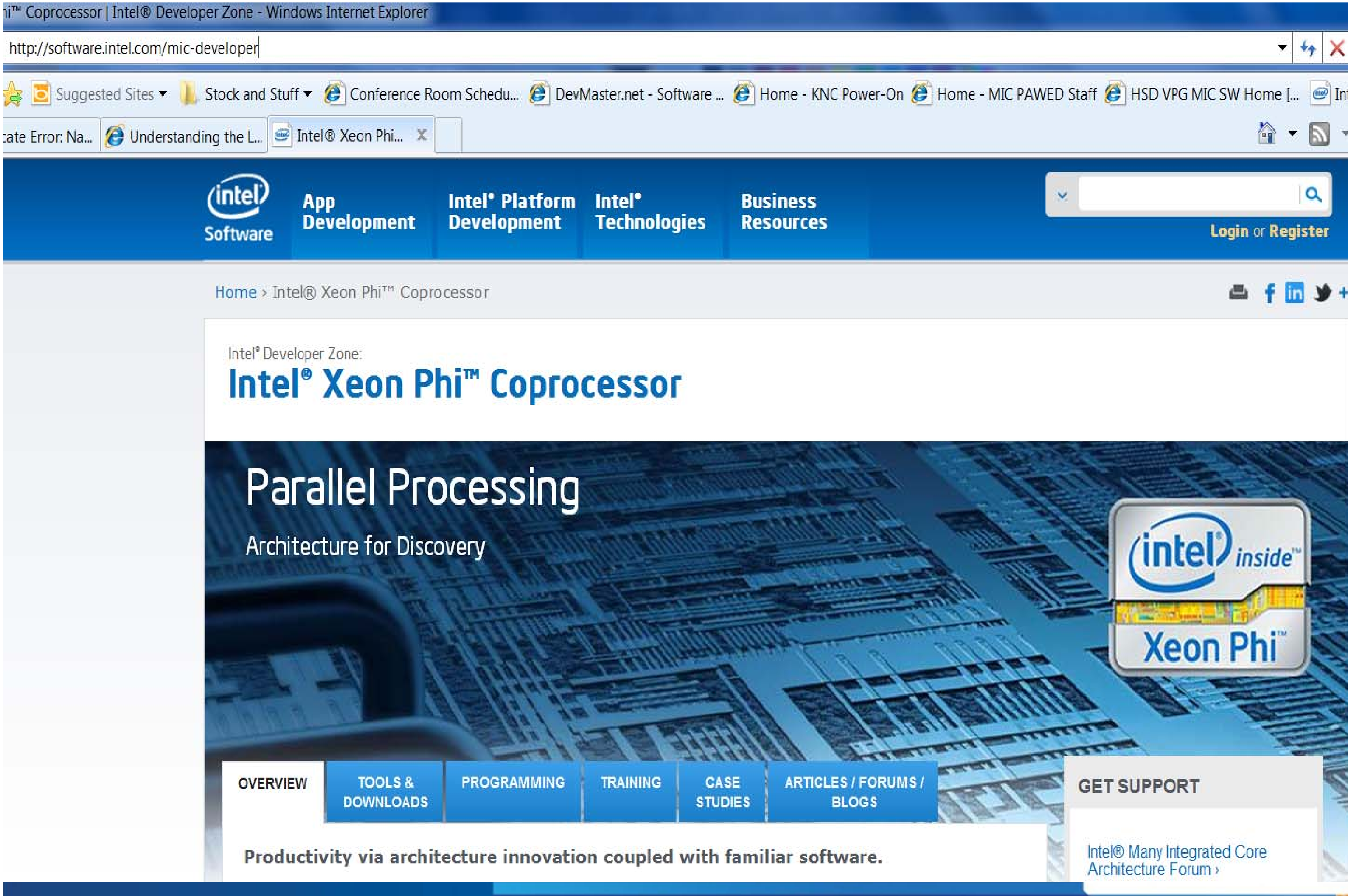
[2] What is libhugetlbfs? - http://lwn.net/Articles/171451/

[3] HOWTO use libhugetlbfs? - http://www.csn.ul.ie/~mel/projects/deb-libhugetlbfs/package/libhugetlbfs-1.2/HOWTO

[4] More on _morecore: http://software.intel.com/sites/default/files/article/299022/1.2.3-large-page-considerations.pdf

[5] Transparent Huge Pages - http://lwn.net/Articles/188056/

[6] Dumping the log buffer: http://lrbforce.ra.intel.com/wiki/index.php/UOS_Log_Buffer_Access_From_Host_Debugfs



Intel® Developer Zone:

Intel® Xeon Phi™ Coprocessor

Parallel Processing

Architecture for Discovery

Productivity via architecture innovation coupled with familiar software.

Intel® Many Integrated Core Architecture Forum ›

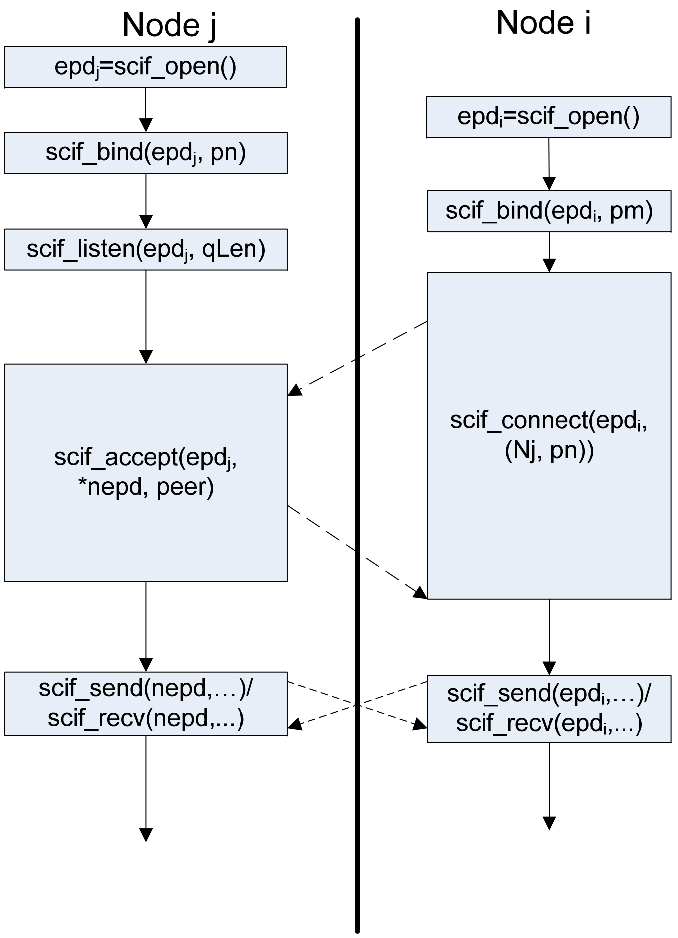
Backup Material

SCIF Backup Material

Connection Process

SCIF connection process is very socket-like

- Open an endpoint, ep, and bind it to some port, pn
- Mark the port as a listening port
- Wait for a connection request, then Accept the connection request and return a new endpoint, nepd
- The peer processes can communicate through the completed connection



Node j

epd$_j$=scif_open()

scif_bind(epd$_j$, pn)

scif_listen(epd$_j$, qLen)

scif_accept(epd$_j$, *nepd, peer)

scif_send(nepd,…)/ scif_recv(nepd,...)

Node i

epd$_i$=scif_open()

scif_bind(epd$_i$, pm)

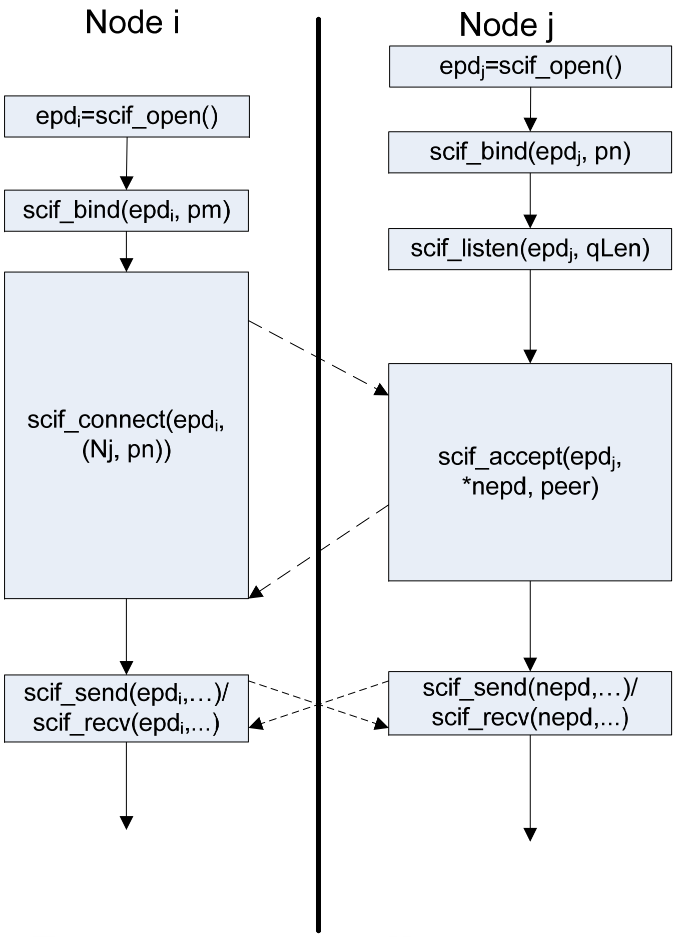
scif_connect(epd$_i$, (Nj, pn))

scif_send(epd$_i$,…)/ scif_recv(epd$_i$,...)

- Open an endpoint, ep, and bind it to some port, pn
- Request a connection to port pn on node j Connection is complete on return
- The peer processes can communicate through the completed connection

Launching Remote Processes

- Connection requires one process to listen and one to request a connection
- How do you get a process on a node that will accept your connection request?
- Some ports are “well known” and reserved for specific clients
  - COI, MYO, OFED, RAS, PM, NetDev
  - Each client typically has a daemon (launched with uOS at R3 or R0)
- Other apps can use services such as COI, micnativeloadex, or ssh (over virtual Ethernet) to launch a process on a node



Node i

epd$_i$=scif_open()

scif_bind(epd$_i$, pm)

scif_connect(epd$_i$, (Nj, pn))

scif_send(epd$_i$,…)/ scif_recv(epd$_i$,...)

Node j

epd$_j$=scif_open()

scif_bind(epd$_j$, pn)

scif_listen(epd$_j$, qLen)

scif_accept(epd$_j$, *nepd, peer)

scif_send(nepd,…)/ scif_recv(nepd,...)

Kernel mode APIs

- int **scif_pin_pages**(void *addr, size_t len, int prot_flags, int map_flags, scif_pinned_pages_t *pinned_pages);
- int **scif_unpin_pages**(scif_pinned_pages_t pinned_pages);
- off_t **scif_register_pinned_pages**(scif_epd_t epd, scif_pinned_pages_t pinned_pages, off_t offset, int map_flags);
- int **scif_get_pages**(scif_epd_t epd, off_t offset, size_t len, struct scif_range **pages);
- int **scif_put_pages**(struct scif_range *pages);
- int **scif_event_register**(scif_callback_t handler);
- int **scif_event_unregister**(scif_callback_t handler);
- int **scif_pci_info**(uint16_t node, struct scif_pci_info *dev);

SCIF APIs at a glance

- Endpoint connection:
  - connection establishment between processes
  - scif_open(), scif_bind(), scif_listen(), scif_accept(), scif_connect(), scif_close()
- Messaging:
  - send/receive messages between connected endpoints
  - scif_send(), scif_recv()
  - A message is an arbitrary sequence of bytes
  - These APIs are intended to be used for short data transfers e.g. commands
- Registration:
  - Exposes local physical memory for remote access via a local Registered Address Space
  - scif_register(), scif_unregister()
- Mapping:
  - scif_mmap(), scif_munmap()
  - Maps remote physical pages into local virtual address space of process

SCIF APIs at a glance (2)

- Remote Memory Access (RMA):
  - Perform DMA or programmed I/O transfers
  - scif_readfrom(), scif_writeto(), scif_vreadfrom(), scif_vwriteto()
  - Supports the notion of "one sided communication"
    - Push or pull data
  - Supports DMA (for large) or CPU (for small transfers) based transfers

- Synchronization:
  - Enables synchronization with RMA completion, now vs. later
  - Must comprehend RMAs completing out of order – multiple DMA channels
  - Scif_fence(), scif_fence_mark(), scif_fence_wait(), scif_fence_signal()

- Utility:
  - scif_get_nodeIDs(), scif_poll(), scif_get_fd()

- Requires HW support to make all of this work (and work well)
  - PCIe accesses
  - SMPT
  - Aperture mapped from the host
  - Interrupts in both directions (ICR and System interrupt registers)