Intel Xeon Phi Coprocessors

Reference: Parallel Programming and Optimization with Intel Xeon Phi Coprocessors, by A. Vladimirov and V. Karpusenko, 2013
Ring Bus on Intel Xeon Phi

Example with 8 cores
Xeon Phi Card

• Coprocessor connected via PCIe Gen 2 (6 GB/s)
• 6-8 GB memory on card
Intel Xeon Phi Coprocessors

• “Many Integrated Core” (MIC) architecture, with 57-61 cores on a chip
  – Processors connected via a ring bus
  – Peak approx 1 Tflop/s double precision

• Simple x86 cores at approx 1 GHz clock speed
  – Enhances code portability
  – In-order processor
  – Each core supports 4-way hyperthreading; each hyperthread issues instructions every other cycle

• Cores have 512-bit SIMD units
Memory characteristics

• L1D: 32 KB, L1I: 32 KB (per core)
• L2: 512 KB (per core) unified
• Memory BW: 350 GB/s peak; 200 GB/s attainable
Programming Modes

• Offload
  – Program executes on host and “offloads” work to coprocessors

• Native (coprocessor runs Linux uOS)
  – Could run several MPI processes

• Symmetric
  (MPI between hosts and coprocessors)
Where’s the disk?

- File system is stored on a virtual file system (stored in DRAM on the card)
- When the coprocessors are rebooted, home directories are reset and necessary libraries must be copied to the file system
  - Or, an actual disk file system can be mounted on the coprocessor, but access would be slow (across PCIe)
Offloading vs. Native/Symmetric Mode

• Offloading is good when
  – Application is not highly parallel throughout and not all cores of the coprocessor can always be used (few fast cores vs. many slow cores)
  – Application has high memory requirements where offloaded portions can use less memory (coprocessor limited to 8 GB in native mode)

• Offloading is bad when
  – Overhead of data transfer is high compared to offload computation
Offloading

• Common offload procedure:
  #pragma offload target(mic) inout(data: length(size))

This is performed automatically:
  – allocate memory on coprocessor
  – transfer data to coprocessor
  – perform offload calculation
  – transfer data to host
  – deallocate memory on coprocessor

• Fall back to host if no coprocessor available
• Code still works if directives are disabled
Persistent data in offloads

- Data **allocation** and **transfer** are expensive
- Can reuse memory space **allocated** on the coprocessor (alloc_if, free_if)
- Can control when data **transfer** occurs between host and coprocessor
  - in, out, inout
  - nocopy to avoid transferring statically allocated variables
  - length(0) to avoid transferring data referenced by pointers
What does this code do?

```c
SetupPersistentData(N, persistent);

#pragma offload_target target(mic:0) \ in(persistent : length(N) alloc_if(1) free_if(0) )

for (int iter = 0; iter < nIterations; iter++) {
    SetupDataset(iter, dataset);
    #pragma offload_target target(mic:0) \ in (dataset : length(N) alloc_if(iter==0) free_if(iter==nIterations-1) ) \ out (results : length(N) alloc_if(iter==0) free_if(iter==nIterations-1) ) \ nocopy (persistent : length(N) alloc_if(0) free_if(iter==nIterations-1) )
    {
        Compute(N, dataset, results, persistent);
    }
}
ProcessResults(N, results);
```
Offload latencies

Latency, ms

Array Size

Default offload: standard TLB pages, 2MB buffers
With memory retention: standard TLB pages, 2MB buffers
With data persistence: standard TLB pages, 2MB buffers

Default offload (memory allocation + data transfer)
With memory retention (data transfer only)
With data persistence (no memory allocation or data transfer)
Asynchronous Offloading

• Host program blocks until the offload completes (default)
• For non-blocking offload, use signal clause
• Works for asynchronous data transfer as well

```c
char* offload0;
#pragma offload target(mic:0) signal(offload0) in(data : length(N))
{ /* ... will not block code execution because of clause "signal" */ }  

DoSomethingElse();

/* Now block until offload signalled by pointer "offload0" completes */
#pragma offload_wait target(mic:0) wait(offload0)
```
Offloading on Intel Xeon Phi

• Two methods
  – pragma offload (fast, managed data allocation and transfer)
  – shared virtual memory model (convenient but slower due to software managing coherency)

• Compare to GPU offloading
  – CUDA approach: launch kernels from the host; explicit function calls to allocate data on GPU and to transfer data between host and GPU
  – OpenACC: pragmas

• Other options
  – OpenMP (pragmas for offloading)
  – OpenCL (explicit function calls), more suitable for GPUs
Virtual Shared Memory for Offloading

- Logically shared memory between host and coprocessor
- Programmer marks variables that are shared
- Runtime maintains coherence at the beginning and end of offload statements (only modified data is copied)
- `_Cilk_shared` keyword to mark shared variables/data
  - shared variables have the same addresses on host and coprocessor, to simply offloading of complex data structures
  - shared variables are allocated dynamically (not on the stack)
- `_Cilk_offload` keyword to mark offloaded functions
- Dynamically allocated memory can also be shared: `_Offload_shared_malloc`, `_Offload_shared_free`
Multiple coprocessors

Use OpenMP threads; one thread offloads to one coprocessor.
On coprocessor, use OpenMP to parallelize across cores.
Native/Symmetric MPI vs. MPI+Offload

Why MPI+Offload Helps: Fewer MPI End-Points

Native MPI: $4 \times P$ end-points for all-to-all MPI_Allgather

MPI+Offload: $P$ end-points for all-to-all MPI_Allgather