This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction plans, our ability to expand and penetrate our addressable markets and other statements that are not historical facts. These statements are only predictions and actual results may materially vary from those projected. Please refer to Cray’s documents filed with the SEC from time to time concerning factors that could affect the Company and these forward-looking statements.
Cray Compiler Environment (CCE) for KNL

- **Load target architecture**
  - module load craype-mic-knl

- **Load compiler environment**
  - module load PrgEnv-cray
  - The PrgEnv-xxx modules will give you compiler wrappers:
    - cc, CC and ftn,
    - that by default target the loaded architecture.

- **KNL support from compiler version 8.5**
  - CCE > 8.5 continuously enhancements and updates
  - cce/8.5.5 default on TDS
Other compilers

- PrgEnv-intel and PrgEnv-gnu work in the same way
  - module swap PrgEnv-cray PrgEnv-intel
  - By default: cc -xMIC-AVX512 ...
  - module swap PrgEnv-cray PrgEnv-gnu
  - By default: cc --march=knl ...
  - Hint: {cc|CC|ftn} -craype-verbose

- Intel 17 has additional KNL enhancements
  - Under the covers
#!/bin/bash
#PBS -N affinity-check
#PBS -l walltime=00:30:00
#PBS -l nodes=16
#PBS -l os=CLE_quad_cache

cd $PBS_WORK_DIR
aprun -n 1024 -N 64 exe
Glossary of terms

Processing Element (PE)
- A discrete software process with an individual address space. One PE is equivalent to: 1 MPI Rank, 1 Coarray Image, 1 UPC Thread.

CPU
- The minimum piece of hardware capable of running a PE. It may share some or all of its hardware resources with other CPU’s. Equivalent to a single “Intel hyper-thread”

Thread
- A logically separate stream of execution inside a parent PE that shares the same address space.

Compute Unit
- The individual unit of hardware for processing, usually described as a “core”. May provide one or more CPU’s.

Compute Node
- Single KNL instance running its own Linux instance
core has 4 CPU’s (hyper-threads)
tile has 2 cores with a shared L2-cache
node has 34 tiles (on XC)
Running an application using ALPS + aprun

ALPS : Application Level Placement Scheduler

- **aprun** is the ALPS application launcher
  - It **must** be used to run application on the XC compute nodes.
  - **aprun** launches groups of Processing Elements on the compute nodes.
  - Cannot get more resources for **aprun** than requested via PBS.
  - The 5 most important parameters (man-page for more examples)

<table>
<thead>
<tr>
<th>Description</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Number of PEs used by the application</td>
<td>-n</td>
</tr>
<tr>
<td>Number of PEs per compute node</td>
<td>-N</td>
</tr>
<tr>
<td>Number of PE’s per numa domain</td>
<td>-S</td>
</tr>
<tr>
<td>Number of threads per PE</td>
<td>-d</td>
</tr>
<tr>
<td>(More precise, the “stride” between 2 PEs on a node)</td>
<td></td>
</tr>
<tr>
<td>Number of to CPUs to use per Compute Unit</td>
<td>-j</td>
</tr>
</tbody>
</table>
Cray XC Basic MPI-Jobs Examples

Single node, Single task
Run a job on one task on one node.

Single node, Multiple Ranks
Run a pure MPI job with 16 Ranks on one node.

Multiple nodes, Multiple Ranks
Run a pure MPI job on 4 nodes with 16 MPI ranks on each node.

```
... #PBS -l nodes=1
... aprun -n 1 ./<exe>
```

```
... #PBS -l nodes=1
... aprun -n 16 ./<exe>
```

```
... #PBS -l nodes=4
... aprun -n 64 -N 16 ./<exe>
```
XC Hybrid MPI/OpenMP Jobs (Example)

**Pure OpenMP Job**
Using 68 threads on a single node

```bash
... #PBS -l nodes=1
... export OMP_NUM_THREADS=68
aprun -n 1 -d $OMP_NUM_THREADS ./omp_exe
```

**Hybrid MPI/OpenMP job**
on 2 nodes with 34 MPI ranks per node, 8 threads for each rank, using 4 Hyper-threads per core.

```bash
... #PBS -l nodes=2
... export OMP_NUM_THREADS=8
aprun -n 68 -d $OMP_NUM_THREADS -N 34 -j 4 ./hybrid_exe
```
Implementing the Parallel Programming Model on hardware

Parallel Application

Threads

Processing Element

Programming Model

Node 0

Node 1

1 Software Thread is bound to 1 Hardware CPU

Linux Process

Hardware implementation
Default Binding - CPU

- By default aprun will bind each PE to a single CPU for the duration of the run.
- This prevents PEs moving between CPUs.
- All child processes of the PE are bound to the same CPU.
- PEs are assigned to CPUs on the node in increasing order from 0. e.g.

```
aprun -n 32 -N 16 -j1 a.out
```
Default Thread Binding (pt 1)

- You can inform aprun how many threads will be created by each PE by passing arguments to the –d (depth) flag.
- aprun does not create threads, just the master PE.
- PEs are bound to CPU spaced by the depth argument, e.g.

```
aprun -n 16 -N 8 -d2 -j1 a.out
```

1 Software PE is bound to 1 Hardware CPU
Default Thread Binding (pt 2)

- Each subsequently created child processes/thread is bound by the OS to the next CPU \((\text{modulo by the depth argument})\). e.g.

```
OMP_NUM_THREADS=2
aprun -n 16 -N 8 -d2 -j1 a.out
```

- Each PE becomes the master thread and spawns a new child thread. The OS binds this child thread to the next CPU.
Default Thread Binding (pt 3)

- `aprun` cannot prevent PEs from spawning more threads than requested.
- In such cases, threads will start to “wrap around” and be assigned to earlier CPUs.
- e.g.

```plaintext
OMP_NUM_THREADS=3
aprun -n 16 -N 8 -d2 -j1 a.out
```

```
Node 0
0.2 0.1 1.2 1.1 7.2 7.1 14 15
Node 1
8.2 8.1 9.0 0.1 15.2 15.1 14 15
```

- In this case, the third thread is assigned to the same CPU as the master PE causing contention for resources.
Removing binding entirely

● aprun can be prevented from binding PEs and their children to CPUs, by specifying “–cc none”. E.g.

OMP_NUM_THREADS=3
aprun -n 16 -N 8 –cc none –j1 a.out

● All PEs and their child processes and threads are allowed to migrate across cores as determined by the standard Linux process scheduler.
● This is useful where PEs spawn many short lived children (e.g. compilation scripts) or over-subscribe the node.
● -d removed as it no longer serves a purpose
Binding to NUMA nodes

- As well as completely removing binding, it is also possible to make aprun bind PEs to all the CPUs on a NUMA node.

`aprun -n 16 -N 8 -S 4 -j1 -cc numa_node a.out`

- PEs will be assigned to the NUMA node that their original PE would have been assigned to with CPU binding and the same options.
- OS allowed to migrate processes within the NUMA node, should be better performance than no binding. “–cc none”
Binding to a CPU set: -depth

- An extension to “numa_node” is the option -cc depth.
- depth defines that a ‘cpu set’ should be used where all threads belonging to a rank are “unbound”.
  cpu-set is given by the –d option
  ```
  aprun -n 4 -d4 -cc depth -j1 a.out
  ```
- Solves the ‘Intel Helper Thread’ issue and also the ‘oversubscribing’ of threads.
- Note that a ‘cpu set’ could spawn cores from different ‘numa nodes’ -> Performance issue
Custom Binding

● aprun also allows users to customise the binding of PEs to CPUs.
  ● Users may pass a colon separated list of CPU binding options to the -cc option.
  ● The n\textsuperscript{th} PE on the node is bound by the n\textsuperscript{th} binding option.

● Each PE binding option may be either a single CPU or a comma separated list of CPUs.
  ● Specifying a single CPU forces the PE and all children and threads to the same PE
  ● Specifying a comma separated list binds the PE to the first CPU in the list and children and threads on to the subsequent CPUs (round-robin)
  ● Additional PEs will be left unbound.
Custom Binding (example)

- Custom binding can be hard to get right. The xthi application is useful for testing binding.

```bash
> export OMP_NUM_THREADS=2
> aprun -n 4 -N 16 -cc 3,2:7,8:9,10,4:1 xthi | sort
Hello from rank 0, thread 0, on nid00009. (core affinity = 3)
Hello from rank 0, thread 1, on nid00009. (core affinity = 2)
Hello from rank 1, thread 0, on nid00009. (core affinity = 7)
Hello from rank 1, thread 1, on nid00009. (core affinity = 8)
Hello from rank 2, thread 0, on nid00009. (core affinity = 9)
Hello from rank 2, thread 1, on nid00009. (core affinity = 10)
Hello from rank 3, thread 0, on nid00009. (core affinity = 1)
Hello from rank 3, thread 1, on nid00009. (core affinity = 1)
```
**XTHI**

- Simple program to check the binding of the threads

Hello from rank 53, thread 0, on nid00032. (core affinity = 53)
Hello from rank 53, thread 1, on nid00032. (core affinity = 121)
Hello from rank 22, thread 1, on nid00032. (core affinity = 90)
Hello from rank 22, thread 0, on nid00032. (core affinity = 22)
Hello from rank 64, thread 1, on nid00032. (core affinity = 132)
Hello from rank 64, thread 0, on nid00032. (core affinity = 64)

- Source code can be found on
  - /tmp/workshop/xthi.c
  - [http://docs.cray.com/books/S-2496-4101/html-S-2496-4101/cnlexamples.html](http://docs.cray.com/books/S-2496-4101/html-S-2496-4101/cnlexamples.html) (9.8)

- Advise: use xthi as part of your job-script
  - to compile xthi.c use same compiler as your application
  - same aprun parameters as application run
  - if using CCE you can also use; export CRAY_OMP_CHECK_AFFINITY=TRUE
Hyper-threads on the XC with ALPS

● Hyper-Threading is a method of improving throughput by sharing execution resources of one CPU
  ● When one thread stalls, the processor can execute instructions from another thread instead of sitting idle
  ● on KNL the number of hyper-threads per core is limited to 4
  ● With aprun, hyper-threading is controlled with -j
    ● -j 1 = no hyper-threading (default)
      (a KNL node is treated to contain 68 CPU’s)
    ● -j 2 = hyper-threading enabled
      (a KNL node is treated to contain 136 CPU’s)
    ● -j 3 = hyper-threading enabled
      (a KNL node is treated to contain 204 CPU’s)
    ● -j 4 = hyper-threading enabled
      (a KNL node is treated to contain 272 CPU’s)

```bash
... #PBS -l nodes=1
... aprun -n 68 -j1 ./<exe>
```
Ignore Hyperthreads; “-j1” quad-mode

In this mode, aprun binds PEs and ranks to the 68 Compute Units (e.g. only use CPUs 0-67)
Include Hyperthreads; “-j2” quad-mode

In this mode, aprun binds PEs and ranks to the 136 Compute Units (e.g. only use CPUs 0-135)

Threads share cores with non –j1 binding

NUMA Node 0

Hyperthread / Compute Unit

CPUs 136-271 Ignored
Include Hyperthreads; “-j4” quad-mode

In this mode, aprun binds PEs and ranks to all 272 Compute Units
OpenMP binding

● For OpenMP thread affinity
  ● OMP_PLACES=cores for affinity between OpenMP threads and cores or
  ● OMP_PLACES=threads for affinity between OpenMP threads and hardware threads.
  ● OMP_PROC_BIND={true,close,spread}

● Can be useful if aprun is not available
  ● slurm with srun
  ● KNL on non-XC systems
With CCE OpenMP, try wait policy passive

- Many programs run faster with OpenMP and 2-way HT
  - aprun –n xxx -N 64 -d 2 -j 2

- With active wait policy, worker threads spin-wait when idle
  - Spin-waiting consumes instruction issue bandwidth
  - KNL is 2-way instruction issue, Xeon has 4-way issue

- With passive wait policy, worker threads halt when idle (after a short spin-wait)
  - CCE default wait policy is active
  - Intel default wait policy is passive

- To reduce issue bottleneck, set passive wait policy
  - export OMP_WAIT_POLICY=passive
  - May also increase OpenMP parallel overhead due to wakeup time
Be Aware—Intel Helper Threads

● The Intel OpenMP runtime:
  ● creates a thread to help... with things... (n+1 threads spawned)
  ● has it's own method of binding to CPUs (KMP_AFFINITY)

● Unfortunately both of these options can make things more complicated.

● Cray advice...
  ● Don’t use KMP_AFFINITY to bind threads:
    ● export KMP_AFFINITY=disabled (threads still float around)
    ● unset KMP_AFFINITY
  ● And use one of the following options:
    ● aprun -cc [numa_node|none|depth] <exe>
    ● aprun -cc 0,x,1,... (the x means don’t bind this thread)

● Getting affinity wrong can be 2X or 3X or more slow down
Example with binding on TDS

● **Examples scripts:**
  ● /tmp/workshop/jobcce.scr
  ● /tmp/workshop/jobintel.scr

● **Using CCE**
  ● export OMP_NUM_THREADS
  ● aprun -d $OMP_NUM_THREADS -cc cpu

● **Using Intel**
  ● unset KMP_AFFINITY
  ● export OMP_NUM_THREADS
  ● aprun -d $OMP_NUM_NUM_THREADS -cc depth
NUMA and cluster modes
## Node configuration

- Multiple NUMA and Memory configurations
  - Choose both at submit time (.. and wait for 22 minutes)

<table>
<thead>
<tr>
<th>Cluster Mode</th>
<th>Moab OS Provisioning</th>
<th>Moab Pre-definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>All-to-all</td>
<td>a2a</td>
<td>os=CLE_a2a...</td>
</tr>
<tr>
<td>Quadrant</td>
<td>quad</td>
<td>os=CLE_quad...</td>
</tr>
<tr>
<td>Sub-NUMA cluster 2</td>
<td>snc2</td>
<td>os=CLE_snc2...</td>
</tr>
<tr>
<td>Sub-NUMA cluster 4</td>
<td>snc4</td>
<td>os=CLE_snc4...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MCDRAM Usage</th>
<th>Moab OS Provisioning</th>
<th>Moab Pre-definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache</td>
<td>cache</td>
<td>os=..._cache</td>
</tr>
<tr>
<td>Flat</td>
<td>flat</td>
<td>os=..._flat</td>
</tr>
<tr>
<td>Hybrid 50-50</td>
<td>equal</td>
<td>os=..._equal</td>
</tr>
<tr>
<td>Hybrid 25-75</td>
<td>split</td>
<td>os=..._split</td>
</tr>
</tbody>
</table>
Numa nodes

- No longer one NUMA node per die

- quad / flat
  - NUMA 0: has 96 GB DRAM-DDR4
  - NUMA 1: has 16 GB MCDRAM

- SNC2 / flat
  - NUMA 0,1 have each 48 GB of DRAM-DDR4
  - NUMA 2,3 have each 8 GB of MCDRAM

- SNC4 / flat
  - NUMA 0,1,2,3 have each 24 GB of DRAM-DDR4
  - NUMA 4,5,6,7 have each 4 GB of MCDRAM

- In quad / cache mode there is only NUMA 0.
NUMA nodes

- On KNL, cores are associated with **DDR4 NUMA nodes**
  - The aprun "-N" option is sufficient to run with fewer than 68 processes per node when a single DDR4 NUMA node exists,
  - but "-S" (number of processes per NUMA node) is required for other configurations to balance over the available memory controllers.

- **To run a job using 64 cores per node, 1 MPI process per core, in quadrant, alltoall, or hemisphere mode:**
  - aprun -n XXX -N 64 a.out

- **To run a job using 64 cores per node, 1 MPI process per core, in **snc2** mode:**
  - aprun -n XXX -S 32 a.out

- **To run a job using 64 cores per node, 1 MPI process per core, in **snc4** mode:**
  - aprun -n XXX -S 16 a.out
NUMA clusters and core numbering

● SNC2 / flat
  ● NUMA nodes 0 and 1 are 48 GB of DRAM-DDR4
  ● NUMA node 0 has cores 0-33 (34 cores)
  ● NUMA node 1 has cores 34-67 (34 cores)
  ● NUMA nodes 2 and 3 are 8 GB of MCDRAM

● SNC4 / flat
  ● NUMA nodes 0, 1, 2 and 3 are 24 GB of DRAM-DDR4
  ● NUMA node 0 has cores 0-17 (18 cores)
  ● NUMA node 1 has cores 18-33 (18 cores)
  ● NUMA node 2 has cores 36-50 (16 cores)
  ● NUMA node 3 has cores 52-67 (16 cores)
  ● NUMA nodes 4, 5, 6 and 7 are 4 GB of MCDRAM
Include Hyperthreads “-j2” on SNC2
Examples

● Want to run on 68-core KNL nodes configured as snc2/cache with 64 ranks per node and 2 OpenMP threads per rank
  ● snc2/cache has 2 NUMA nodes 0 and 1
  ● -S = --pes-per-numa-node
  ● aprun -n $NTASKS -N 64 -S 32 -d 2 -j 2 xthi_cce
  ● aprun -n $NTASKS -N 64 -S 32 -d 2 -j 2 -cc depth xthi_intel

● How about snc2/flat?
  ● Now there are 4 NUMA nodes. 0,1 are DRAM-DDR4; 2,3 are HBM
  ● However, -S really == PEs per NUMA node that have any CPUs associated, so same command works
Using numactl to use MCDRAM

- To get all memory allocated in MCDRAM, use numactl
  - Configure MCDRAM as flat
  - MCDRAM will be NUMA node 1
  - Per-node memory limit is 16 GB (MCDRAM size)
  - Run using numactl

        aprun -n 320 -N 64 numactl --membind=1 a.out

- To use MCDRAM first, but overflow into DDR4
  - Configure flat as above
  - Use --preferred=1 instead of --membind=1
  - Per node memory limit includes all memory (MCDRAM+DDR4)
  - Often does not help much
    - Only first allocations will be to MCDRAM
    - Later allocations will overflow into DDR4
More examples

● How much is MCDRAM helping me in quad/flat?

```
/usr/bin/time -p aprun ... numactl --membind=1 a.out >& out.mcdram

/usr/bin/time -p aprun ... numactl --membind=0 a.out >& out.ddr
```
KNL HBM and NUMA configuration

- **Recommendations:**

- About 50% of applications prefer 100% cache
  - Caution: MCDRAM as cache is direct mapped and potential for cache thrashing

- About 50% of applications prefer 0% cache especially if in quad/flat using < 16 GB per node
  - `aprun ... numactl --membind=1 a.out`
  - `aprun ... numactl --preferred=1 a.out` #allocate out of MCDRAM until exhausted

- snc2 and snc4 are interesting but haven’t found any apps that benefit a lot
  - snc2/cache has 2 numa nodes; snc2/flat has 4 numa nodes; snc4/flat has 8 numa nodes
  - Still early to judge snc2 and snc4

- Never use all2all or hemisphere. No way they could beat quad.
Cray MPI on KNL
Use Core Specialization

- **MPI sync time can rise due to OS noise**
  - Usually when a large number of nodes is used

- **May show up as large times for collectives**
  - MPI_Allreduce

- **To reduce OS noise by 50%, use core specialization**
  - Impact on collective calls even larger than 50%

- **Core specialization reserves hardware to handle OS work**
  - Highest numbered available cpu is selected first

- **ALPS:** aprun -r 1
  - Don’t oversubscribe:
    - Illegal: aprun -r1 -n 272 -N 272 -j 4 a.out
    - Legal: aprun -r1 -n 271 -N 271 -j 4 a.out
    - Legal: aprun -r8 -n 264 -N 264 -j 4 a.out
Hybrid MPI/OpenMP Applications: Design Alternatives

Option A: *(Top Down)*

! Move OpenMP near the top of the call stack

```fortran
!#OMP PARALLEL
DO WHILE (t .LT. tend)
  !#OMP DO SCHEDULE(GUIDED)
  DO work = 1, work_end
    CALL update_work()
    ! All threads drive MPI
  END DO
END DO
```

```
“Top-Down” SPMD model is more appealing for KNL
```

Option B: *(Bottom Up)*

! Keep OpenMP within a “compute” loop

```fortran
DO WHILE (t .LT. tend)
  DO work = 1, work_end
    CALL update_work()
    ! MPI driven by single thread
  END DO
END DO
```

```fortran
SUBROUTINE update_work()
  !$OMP PARALLEL DO
  SCHEDULE(STATIC)
  DO i = 1, nx
    …do work…
  END DO
END SUBROUTINE
```
Tips and tricks
Tips and tricks

● Static executables are faster than dynamic
  ● export CRAYPE_LINK_TYPE=static #often default
  ● More noticeable on KNL

● When using OpenMP with Hyper-Threads try
  OMP_WAIT_POLICY=passive
  ● Default Cray policy is active; default Intel policy is passive
  ● Passive is usually best at higher level threading; active is usually best for loop level threading

● Try rank reordering for large jobs to maximize on node
  MPI communication
  ● CrayPat, perftools “grid_order”
**KNL: Known issues**

- 68 core compute nodes
  - Not a power of two

- SNC4 configuration has 9 tiles on two NUMA nodes, 8 tiles on other two NUMA nodes

- Application placement may work better using 64 cores per compute node

- Using all 272 cpus on a compute node as MPI task stresses Aries resources for Cray PMI application and performance can suffer (depending on MPI usage).
Intel-MPI linked code running on XC

1. **CCM**
   - module swap PrgEnv-cracy PrgEnv-intel
   - module load ccm
   - export PATH=/.../impi/5.0.1.035/intel64/bin/mpirun:$PATH
   - ccmrun mpirun -np 2 -ppn 1 pp2.test

2. **Intel/Cray compatibility (better)**
   - module swap PrgEnv-cracy PrgEnv-intel
   - module swap cray-mpich cray-mpich-abi
   - export
     - LD_LIBRARY_PATH=$CRAY_LD_LIBRARY_PATH:$LD_LIBRARY_PATH
   - aprun -n 2 -N 1 pp2.test

Method 2 works for any MPICH-ABI compliant library: MPICH, IBM MPI (https://www.mpich.org/abi/)
Intel tools on Cray XC

● Vtune support on XC is limited
  ● required patches and hooks that are not present in CLE
  ● there is no ssh option to compute nodes
  ● Cray and Intel are working together at Argonne for better Vtune support

● Extra helper thread with OpenMP

● KMP_AFFINITY settings
Application suitability

- **Good:** vectorised code
  - matrix-matic-vector: BLAS, LaPACK, FFT,
  - finite-difference
  - kernel fit in MCDRAM (16 GB)

- **Try:** vector-scalar code
  - finite element
  - OpenMP scaling
  - memory bound
  - short loops

- **not so good:** non-vectorised code
  - scalar code
  - many branches
  - no OpenMP
**KNL strategy**

1. **Compile and run code on all 68 cores in quad_cache**
   - include xthi program to check affinity
   - try OpenMP and hyper-threading
   - check correctness of results
   - use core-specialization –r1

2. **Profile code to find bottlenecks**
   - check if most time consuming kernels are vectorized
   - add more OpenMP parallelism where possible

3. **Find out memory footprint on one node**
   - smaller than 16 GB:
     - flat-cache will be the best mode
     - quad/cache can be good as well
   - larger than 16 GB:
     - quad-cache might be the best mode
     - try quad-flat modes with selected parts allocated in MCDRAM