NERSC Early KNL Experiences

Yun (Helen) He
NERSC/LBNL

NCAR Multi-core 6 Workshop
Sept 13-14, 2016
Introduction
NERSC Exascale Science Application Program (NESAP)

- The NESAP program was launched in Fall 2014 to prepare NERSC user community for Cori KNL architecture.
- 20 applications were selected as Tier 1 (with postdocs) and Tier 2 applications to work closely with Cray, Intel and NERSC staff. Additional 26 Tier 3 teams.
- 80% of NERSC hours are represented by Tiers 1,2,3 and proxy codes.
NERSC KNL System: Cori Phase 2

• Cori KNL: 9,304 nodes. Main features:
  – Many cores: 68 cores per node, 4 hardware threads per core.
    • 3 times of cores (6 times of logical cores) per node than NERSC IvyBridge Edison.
  – Larger vector units (supports AVX-512 instruction set)
    • Dual 512-bit SIMD units with FMA: 32 double precision flops/cycle
    • Edison (IvyBridge) has 256-bit AVX: 8 double precision flops/cycle
  – On package high bandwidth memory: MCDRAM
    • 450 GB/sec STREAM measurement as compared to 85 GB/sec from DDR4.
    • No direct L3 cache
  – Burst Buffer

• Cori Phase 1 & Phase 2 under merge starting from Sept 19, 2016, for about 6 weeks

• NESAP teams will have access first in early Nov

• Gating procedure for general users
  – Need to show performance and scaling effort/results
KNL Test Systems

• **Carl (white boxes from Intel, single nodes only)**
  - B0: 64 cores @1.3 GHz
  - B1: 68 cores @1.4 GHz

• **Gerty (test system from Cray, with Aries network)**
  - Similar to real Cori Phase1 & Phase2 system
  - P1: Haswell. Dual sockets, 16 cores/socket @ 2.3 GHz
  - P2: KNL. B1. 68 cores. @1.4 GHz

• **All KNL nodes have**
  - 4 hardware threads per core
  - 9600 GB DDR4 and 16 GB MCDRAM
Intel Tools are Useful
• Use -qopt-report=5 for detailed compiler reports on which optimizations have been performed, why certain loops are vectorized or not, etc.
Intel VTune

• **Memory Access**
  – Detect memory hierarchy access issues (such as false sharing) and NUMA problems
  – measure DRAM and MCDRAM bandwidth
  – suggest data structures to allocate to MCDRAM

• **General Exploration**
  – Code efficiency

• **Advanced Hot Spots**
  – MPI/OpenMP load balance, potential gain

*Diagram from Intel*
Intel Advisor

- **Vectorization Advisor**
  - Sorts loops by potential performance gain
  - Vectorization analysis
  - Memory access pattern data
  - Roofline Analysis

- **Threading advisor**
  - Threading design tool
  - Suitability analysis with expected speedup
Intel Inspector

- Detect memory errors
  - memory leak
  - data race, deadlock etc.
- Memory growth measurement
Guide and Understand Optimization with Roofline Model
The Roofline Model

Arithmetic Intensity (flops/bytes)

Attainable Performance (GFlop/s)

Peak GFlop/s (w/FMA)

(no FMA)

Peak Memory Bandwidth (GB/s)

Memory Bandwidth (GB/s)

(no FMA, no vectorization)

Cache Bandwidth (GB/s)

Ideal Arithmetic Intensity

(no cache blocking)

Arithmetic Intensity = \frac{Total \, FLOPs}{Total \, Bytes}

Attainable FLOPs / sec = \min \left\{ \frac{Peak \, FLOPs}{Peak \, Memory \, Bandwidth \times Arithmetic \, Intensity} \right\}

Arithmetic Intensity:

- O(1)
- O(\log(N))
- O(N)

Applications:

- SpMV, BLAS1,2
- FFTs
- Lattice Methods
- Stencils (PDEs)
- Dense Linear Algebra, (BLAS3)
- Particle Methods

*Slide from Doug Doerfler et. al., IXPUG at ISC2016*
KNL Roofline Results

- Using 2 threads/core
- Max L1, L2 and MCDRAM
  - 1 FLOP/iteration
  - 4 MPI + 32 threads
- Max GFLOP/s
  - 64 FLOPs/iteration
  - 2 MPI + 64 threads

All Bandwidths are in GB/s
(a) Values assume an AVX frequency of 1.1 GHz
(b) L2 assumed ~ (L1 / 4)?
(c) MCDRAM BW is for 1R/1W per iteration

Slide from Doug Doerfler et. al., IXPUG at ISC2016
How to Measure Arithmetic Intensity


• Intel SDE measures “Total Flops”

• Intel Vtune measures “Total Bytes”

• Haswell consistently attains a higher arithmetic intensity than KNL
  
  – KNL generally moves more data to/from memory than Haswell due to lack of L3 cache
  
  – The higher theoretical performance benefits of MCDRAM bandwidth may not be fully realized due to this extra data movement
PICSAR Example

- **Optimizations**
  - Original code spatially decomposes the problem with MPI
  - MPI subdomains are subdivided into large number of tiles handled with OpenMP, improving memory locality, hence cache reuse of tiles, and load balance.
  - Deposition and Interpolation steps were rewritten to enable more efficient vectorization, plus particle cell sorting was added to again improve memory locality and hence cache reuse.

- **Observations**
  - Tiling and vectorization increase the arithmetic intensity to take advantage of additional effective memory bandwidth.
  - Not memory bound so more optimization potential.

---

**Work by Mathieu Lobet et. al.**
Slide adapted from Doug Doerfler et. al., IXPUG at ISC2016
MFDn Example

- **Optimizations**
  - Use case requires all memory on node (HBM + DDR). Explicitly place important arrays into MCDRAM with FASTMEM directives.
  - Use blocked (nRHS) to improve bandwidth and locality (the larger sparse matrix resides in DDR4)

- **Observations**
  - Code is highly memory bandwidth bound
  - More RHS helps to increase arithmetic intensity. However, the number of RHS is limited by MCDRAM capacity.

![Graphs showing arithmetic intensity vs. GFLOP/s for Haswell and KNL MCDRAM](image)

*Work by Brandon Cook et al.*

*Slide adapted from Doug Doerfler et. al., IXPUG at ISC2016*
BerkeleyGW Example

• **Optimization Steps**
  1. Refactor (3 Loops for MPI, OpenMP, Vectors)
  2. Add OpenMP
  3. Initial Vectorization (loop reordering, conditional removal)
  4. Cache-Blocking to better reuse last level cache
  5. Improved Vectorization
  6. Add hyper-threading

• **Observations**
  – Arithmetic Intensity reduced from step 2 to 3. Problem size larger than L2 but Haswell has L3 to catch
  – From step 3 to 4: No Haswell speedup since it fits in L3. Good improvement for KNL
  – Has potential for further optimization

*From Jack Deslippe*
Boxlib Example

- **Optimization**
  - Loop tiling: Divide boxes into smaller tiles. Divide tiles among OpenMP threads

- **Observations**
  - Memory bandwidth bound
  - Effective L2 and L3 cache reuse on Haswell.

From Brian Friesen
Overall NESAP Optimization Results

Results from the NESAP teams
Significant speed usually involves code restructuring to improve vectorization and data locality.

Speedup is mostly larger on KNL since fewer and faster Haswell (with L3 cache) is more forgiving to imperfect thread scaling and vectorization. (WARP, BerkeleyGW)

- Boxlib is an exception. Tiling has more benefit for memory bandwidth bound on Haswell (no HBM)
EMGeo, MILC, and Chroma see large KNL vs. Haswell speedup: memory bandwidth bound. Speedup came from effectively use MCDRAM.
• **EMGeo, MILC, Chroma, MFDn** are memory bandwidth bound. Speedup mostly matches MCDRAM vs. DDR bandwidth ratio
• KNL generally moves more data to/from memory than Haswell due to lack of L3 cache
• Codes effectively use L3 cache may perform better on Haswell than on KNL: Boxlib, XGC1
BerkeleyGW sees large AVX512 (vectorization effect) vs. scalar instructions.
Not many codes see large effect with FMA. However, FMA used in libraries are not measured here.
MPI/OpenMP Process and Thread Affinity
Affinity Goal

• Correct process and thread affinity for hybrid MPI/OpenMP programs is the base for getting optimal performance on KNL. It is also essential for guiding further performance optimizations.

• Our goal is to promote OpenMP4 standard settings for portability. For example, OMP_PROC_BIND and OMP_PLACES are preferred to Intel specific KMP_AFFINITY settings.

• Discovered in an Intel Dungeon session with CESM HOMME that OMP settings ran slower than KMP settings.
  – What can be the cause?
  – Investigation started on this ...
HOMME: 7 Test Runs

Expect to see same performance from all 7 cases on a 64-core KNL quad flat node

- case 1: mpirun -n 32 -env OMP_NUM_THREADS 2 -env KMP_AFFINITY compact,verbose -env KMP_PLACE_THREADS 1T numactl -m 1 ./app
- case 2: mpirun -n 32 -env KMP_AFFINITY compact,verbose -env KMP_PLACE_THREADS 2C,1T numactl -m 1 ./app
- case 3: mpirun -n 32 -env OMP_NUM_THREADS 2 -env KMP_AFFINITY scatter,verbose numactl -m 1 ./app
- case 4: mpirun -n 32 -env OMP_NUM_THREADS 2 -env OMP_PROC_BIND spread -env OMP_PLACES threads numactl -m 1 ./app
- case 5: mpirun -n 32 -env OMP_NUM_THREADS 2 -env OMP_PROC_BIND close -env OMP_PLACES cores numactl -m 1 ./app
- case 6: mpirun -n 32 -env KMP_AFFINITY scatter,verbose -env KMP_PLACE_THREADS=2C,1T numactl -m 1 ../app
- case 7: mpirun -n 32 -env KMP_AFFINITY scatter,verbose -env KMP_PLACE_THREADS=2C,1T -env OMP_NUM_THREADS 2 numactl -m 1 ./app
Affinity Analysis

• Confirmed with another application this is the case (same performance from all tests)
• Confirmed with my simple affinity test case that core bindings are all equivalent
• However, initial runs see different results from the 7 test cases for HOMME. Some cases are >2X slower. Also quad cache performance is about 5% slower.
• Further investigations showed even though asking for 2 threads only, the code is running with 4. It was discovered later nested OpenMP is set in the code by default!
• Using the modified code with explicit num_threads clauses specified for nested OpenMP regions, all 7 cases then perform the same on quad flat and quad cache nodes.
Good nested OpenMP scaling achieved after code bug is fixed
What About a 68-core KNL Node?

% mpirun -n 8 -env OMP_PROC_BIND spread -env OMP_PLACES threads -env OMP_NUM_THREADS 4 ./xthi |sort -k4n,6n
Hello from rank 0, thread 0, on ekm118. (core affinity = 0)
Hello from rank 0, thread 1, on ekm118. (core affinity = 70)
Hello from rank 0, thread 2, on ekm118. (core affinity = 72)
Hello from rank 0, thread 3, on ekm118. (core affinity = 142)
Hello from rank 1, thread 0, on ekm118. (core affinity = 144)
Hello from rank 1, thread 1, on ekm118. (core affinity = 214)
Hello from rank 1, thread 2, on ekm118. (core affinity = 216)
Hello from rank 1, thread 3, on ekm118. (core affinity = 15)

Use I_MPI_PIN_DOMAIN to set to number of logical cores per MPI task. Otherwise, OMP tasks are crossing tile boundaries. Good to waste extra 4 cores on purpose if #MPI tasks is not divisible by 68.

% mpirun -n 8 -env OMP_PROC_BIND spread -env OMP_PLACES threads -env OMP_NUM_THREADS 4 -env I_MPI_PIN_DOMAIN 32 ./xthi |sort -k4n,6n
Hello from rank 0, thread 0, on ekm118. (core affinity = 0)
Hello from rank 0, thread 1, on ekm118. (core affinity = 2)
Hello from rank 0, thread 2, on ekm118. (core affinity = 4)
Hello from rank 0, thread 3, on ekm118. (core affinity = 6)
Hello from rank 1, thread 0, on ekm118. (core affinity = 8)
... 
Hello from rank 7, thread 0, on ekm118. (core affinity = 56)
Hello from rank 7, thread 1, on ekm118. (core affinity = 58)
Hello from rank 7, thread 2, on ekm118. (core affinity = 60)
Hello from rank 7, thread 3, on ekm118. (core affinity = 62)

• xthi.c and xthi-nested.c test codes available upon request
• Requested to OpenMP Standard to provide Intel KMP_AFFINITY=verbose or CRAY_OMP_CHECK_AFFINITY=TRUE equivalent
Nested OpenMP Thread Affinity

• Again, I_MPI_PIN_DOMAIN is important
• Sample settings for 2 MPI tasks, 4 outer OpenMP threads, and 4 inner OpenMP threads:
  
  % export OMP_NUM_THREADS=4,4
  % export OMP_PROC_BIND=spread,close
  % export OMP_PLACES=threads
  % export OMP_NESTED=true
  % export I_MPI_PIN_DOMAIN=128 # 32 physical cores
  % export KMP_HOT_TEAMS=1
  % export KMP_HOT_TEAMS_MAX_LEVELS=2

• Use num_threads clause in source codes to set threads for nested regions. For most other non-nested regions, use OMP_NUM_THREADS environment variable for simplicity and flexibility.
Choice of Default Cluster and Memory Modes
Available Modes for KNL Nodes

- KNL has configurable on-chip interconnect for NUMA and memory mode.
- Sub-NUMA Cluster (SNC) modes
  - No SNC (Quad, all-2-all, Hemisphere), SNC-2, SNC-4
- Memory modes
  - Cache, Flat, Hybrid
- No SNC and Cache modes are relatively easier to use
- Takes about 11 to 26 min of reboot time in order to switch to another mode
- Ongoing analysis for setting default mode(s) for NERSC (> 5,000 users, 800 projects)?
General Strategies and Observations

• If application memory \( \leq 16\) GB, use Flat mode to allocate all in MCDRAM is best. If not, manual placement is needed.
• Cache mode gives pretty good start for most apps.
• Cache mode can be beaten by Flat mode + manual data placement in MCDRAM.
• Performance with Cache mode can vary a lot more than Flat mode.
  – Different memory placement of allocations and different fragmentation caused by previous jobs.
• SNC4/SNC2 provide small advantages over quadrant mode for some (not all) apps, but relatively harder to use. (different number of cores per NUMA domain for SNC4)
• We have yet to try Hybrid mode with MCDRAM
• Default cluster and memory mode(s) on Cori have not been finalized.
  – Most likely, quad flat (and some quad cache and/or SNC4 flat).
  – Also not finalized whether to allow users to switch modes (most likely yes to a certain extent, but node reboot time will be billed as run time)
Summary

- Optimizing for KNL requires good thread level scaling (OpenMP), vectorization and effective use of HBM.
- Use available Intel tools to help identify areas to work on for optimization.
- Use Roofline model to guide optimization potential.
- Correct process and thread affinity is the base for getting optimal performance.
- Keep portability in mind, use portable programming models.
Thank you.