Cori Application Readiness Strategy and Early Experiences

March, 2016
What is different about Cori?

**Edison (Ivy-Bridge):**
- 12 Cores Per CPU
- 24 Virtual Cores Per CPU
- 2.4-3.2 GHz
- Can do 4 Double Precision Operations per Cycle (+ multiply/add)
- 2.5 GB of Memory Per Core
- ~100 GB/s Memory Bandwidth

**Cori (Knights-Landing):**
- Up to 72 Physical Cores Per CPU
- Up to 288 Virtual Cores Per CPU
- Much slower GHz
- Can do 8 Double Precision Operations per Cycle (+ multiply/add)
- < 0.3 GB of Fast Memory Per Core
- < 2 GB of Slow Memory Per Core
- Fast memory has ~ 5x DDR4 bandwidth
NESAP
The NERSC Exascale Science Application Program
Code Coverage

Breakdown of Application Hours on Hopper and Edison 2013

NESAP Tier-1, 2 Code
NESAP Proxy Code or Tier-3 Code
Resources for Code Teams

• Early access to hardware
  – Access to Babbage (KNC cluster) and early “white box” test systems expected in 2015
  – Early access and significant time on the full Cori system

• Technical deep dives
  – Access to Cray and Intel staff on-site staff for application optimization and performance analysis
  – Multi-day deep dive (‘dungeon’ session) with Intel staff at Oregon Campus to examine specific optimization issues

• User Training Sessions
  – From NERSC, Cray and Intel staff on OpenMP, vectorization, application profiling
  – Knights Landing architectural briefings from Intel

• NERSC Staff as Code Team Liaisons (Hands on assistance)

• 8 Postdocs
NESAP Postdocs

Taylor Barnes
Quantum ESPRESSO

Brian Friesen
Boxlib

Andrey Ovsyannikov
Chombo-Crunch

Mathieu Lobet
WARP

Tuomas Koskela
XGC1

Tareq Malas
EMGeo

Target Application Team Concept
(1 FTE Postdoc +)
0.2 FTE AR Staff

0.25 FTE COE
1.0 FTE User Dev.

1 Dungeon Ses. +
2 Week on site w/ Chip vendor staff
NERSC Staff associated with NESAP

Katie Antypas  Nick Wright  Richard Gerber  Brian Austin  Zhengji Zhao  Helen He  Ankit Bhagatwala  Stephen Leak

Woo-Sun Yang  Rebecca Hartman-Baker  Doug Doerfler  Jack Deslippe  Brandon Cook  Thorsten Kurth

Target Application Team
Concept

(1 FTE Postdoc +)
0.2 FTE AR Staff
0.25 FTE COE
1.0 FTE User Dev.

1 Dungeon Ses. +
2 Week on site w/ Chip vendor staff
Timeline

- Jan 2014: Requirements Evaluation
- May 2014: Gather Early Experiences and Optimization Strategy
- Jan 2015: Prototype Code Teams (BerkeleyGW / Staff)
  - Prototype good practices for dungeon sessions and use of on site staff.
- Jan 2016: Code Team Activity
  - Post-Doc Program
  - Center of Excellence
- Jan 2017: Chip Vendor On-Site Personnel / Dungeon Sessions
  - White Box Access
  - Delivery
- NERSC Led OpenMP and Vectorization Training (One Per Quarter)
- Vendor General Training

NERSC User and 3rd Party Developer Conferences
Timeline

Jan 2014
Requirements Evaluation
Gather Early Experiences and Optimization Strategy
Prototype Code Teams (BerkeleyGW / Staff)
-Prototype good practices for dungeon sessions and use of on site staff.

May 2014

Jan 2015
NERSC Led OpenMP and Vectorization Training (One Per Quarter)

Jan 2016
Vendor General Training
Code Team Activity
Post-Doc Program
Center of Excellence
Chip Vendor On-Site Personnel / Dungeon Sessions
White Box Access
Delivery

Jan 2017
Vendor General Training
NERSC User and 3rd Party Developer Conferences
Working With Vendors

NERSC is uniquely positioned between HPC Vendors and HPC Users and Applications developers.

NESAP provides a power venue for these two groups to interact.
Optimization Strategy
Important Optimization Concepts

• MPI+X (Where X is MPI, OpenMP, PGAS etc)

• Vectorization

• Understanding Memory Bandwidth
OpenMP scales only to 4 Threads

large cache miss rate

Communication dominates beyond 100 nodes

Code shows no improvements when turning on vectorization

50% Walltime is IO

IO bottlenecks

Compute intensive doesn’t vectorize

MPI/OpenMP Scaling Issue

Use Edison to Test/Add OpenMP Improve Scalability. Help from NERSC/Cray COE Available.

Can you use a library?

Increase Memory Locality

Create micro-kernels or examples to examine thread level performance, vectorization, cache use, locality.

Utilize High-Level IO-Libraries. Consult with NERSC about use of Burst Buffer.

The Dungeon: Simulate kernels on KNL. Plan use of on package memory, vector instructions.
Can You Increase Flops Per Byte Loaded From Memory in Your Algorithm?

Make Algorithm Changes

Run Example Changes

Using HBM on Cori For Key Arrays

Is Performance affected by Half-Clock Speed?

Run Example at “Half Clock” Speed

Is Performance affected by Half-Packing?

No

Yes

Your Code is at least Partially Memory Bandwidth Bound

Make Sure Your Code is Vectorized!

Measure Cycles Per Instruction with VTune

Likely Partially Memory Latency Bound (assuming not IO or Communication Bound)

Use IPM and Darshan to Measure and Remove Communication and IO Bottlenecks from Code

Yes

No

No

Yes

You are at least Partially CPU Bound

The Ant Farm Flow Chart
Can You Increase Flops Per Byte Loaded From Memory in Your Algorithm?

- Yes
  - Explore Using HBM on Cori For Key Arrays

- No
  - Can You Reduce Memory Requests Per Flop In Algorithm?
    - Yes
      - Try Running With as Many Virtual Threads as Possible (> 240 Per Node on Cori)
    - No
      - Make Sure Your Code is Vectorized! Measure Cycles Per Instruction with VTune
Are you memory or compute bound? Or both?

Run Example in “Half Packed” Mode

If you run on only half of the cores on a node, each core you do run has access to more bandwidth

```
srun -N 2 -n 24 -c 2 - S 6 ...
```

VS

```
srun -N 1 -n 24 -c 1 ...
```

If your performance changes, you are at least partially memory bandwidth bound
If your performance changes, you are at least partially memory bandwidth bound.

Are you memory or compute bound? Or both?

Run Example in “Half Packed” Mode.

If you run on only half of the cores on a node, each core you do run has access to more bandwidth.

srun -n 24 -N 12 - S 6 ...

vs

aprun -n 24 -N 24 -S 12 ...

If your performance improves, you are at least partially compute bound.

Quantum ESPRESSO Packed Vs. Unpacked Performance

Walltime (s)

Packed

Unpacked
Measure memory bandwidth usage in VTune. (Next Talk)

Compare to Stream GB/s.

If 90% of stream, you are memory bandwidth bound.

If less, more tests need to be done.
Are you memory or compute bound? Or both?

Run Example at “Half Clock” Speed

Reducing the CPU speed slows down computation, but doesn’t reduce memory bandwidth available.

```
srun --cpu-freq=2400000 ...
```

VS

```
srun --cpu-freq=1900000 ...
```

If your performance changes, you are at least partially compute bound.
So, you are Memory Bandwidth Bound?

What to do?

1. Try to improve memory locality, cache reuse

2. Identify the key arrays leading to high memory bandwidth usage and make sure they are/will-be allocated in HBM on Cori.

   Profit by getting ~ 5x more bandwidth GB/s.
So, you are Compute Bound?

What to do?

1. Make sure you have good OpenMP scalability. Look at VTune to see thread activity for major OpenMP regions.

2. Make sure your code is vectorizing. Look at Cycles per Instruction (CPI) and VPU utilization in vtune.

See whether intel compiler vectorized loop using compiler flag: -qopt-report=5
Things that prevent vectorization in your code

Example From Cray COE Work on XGC1

Original

real(8), dimension
(5, (col_f_nvr-1)*(col_f_nvz-1),
(col_f_nvr-1)*(col_f_nvz-1)) :: Ms

do index_ip = 1, mesh_Nzml
    do index_jp = 1, mesh_Nrml
        index_2dp = index_jp+mesh_Nrml*(index_ip-1)
        tmp_vol = cs2%local_center_volume(index_jp)
        tmp_f_half_v = f_half(index_jp, index_ip) * tmp_vol
        tmp_dfdr_v = dfdr(index_jp, index_ip) * tmp_vol
        tmp_dfdz_v = dfdz(index_jp, index_ip) * tmp_vol
        tmp(1:3) = tmp(1:3) +
        Ms(1:3, index_2dp, index_2D) * tmp_f_half_v
        tmp(5) = tmp(5) +
        Ms(4, index_2dp, index_2D) * tmp_dfdr_v +

Optimized

real(8), dimension
((col_f_nvr-1), 5, (col_f_nvz-1),
(col_f_nvr-1)*(col_f_nvz-1)) :: Ms

do index_ip = 1, mesh_Nzml
    do index_jp = 1, mesh_Nrml
        index_2dp = index_jp+mesh_Nrml*(index_ip-1)
        tmp_vol = cs2%local_center_volume(index_jp)
        tmp_f_half_v = f_half(index_jp, index_ip) * tmp_vol
        tmp_dfdr_v = dfdr(index_jp, index_ip) * tmp_vol
        tmp_dfdz_v = dfdz(index_jp, index_ip) * tmp_vol
        tmp(1:3) = tmp(1:3) +
        Ms(index_ip, 1, index_jp, 1, index_2D) *
        tmp_f_half_v
        tmp(5) = tmp(5) +
        Ms(index_ip, 4, index_jp, 1, index_2D) *
        tmp_dfdr_v
Things that prevent vectorization in your code

Example From Cray COE Work on XGC1

Original

```fortran
real(8), dimension
(5,(col_f_nvr-1)*(col_f_nvz-1),
(col_f_nvr-1)*(col_f_nvz-1)) :: Ms

do index_ip = 1, mesh_Nzml
  do index jp = 1, mesh_Nrml
    index_2dp = index jp+mesh_Nrml*(index_ip-1)

    tmp_vol = cs2%local_center_volume(index jp)
    tmp f half_v = f_half(index jp, index ip) * tmp_vol
    tmp dfdr v = dfdr(index jp, index ip) * tmp_vol
    tmp dfdz v = dfdz(index jp, index ip) * tmp_vol

    tmpr(1:3) = tmpr(1:3) +
    Ms(1:3,index_2dp,index_2D) * tmp f half v
    tmpr(5) = tmpr(5) +
    Ms(4,index_2dp,index_2D)*tmp dfdr v +

Optimized

```fortran
real (8), dimension
{(col f nvr-1),5,(col f nvz-1),
(col f nvr-1)*(col f nvz-1)) :: Ms

do index_ip = 1, mesh_Nzml
  do index jp = 1, mesh_Nrml
    index_2dp = index jp+mesh_Nrml*(index ip-1)

    tmp_vol = cs2%local_center_volume(index jp)
    tmp f half_v = f_half(index jp, index ip) * tmp_vol
    tmp dfdr v = dfdr(index jp, index ip) * tmp_vol
    tmp dfdz v = dfdz(index jp, index ip) * tmp_vol

    tmpr(index ip,1) = tmpr(index ip,1) +
    Ms(index jp,1,index ip,index 2D)* tmp f half v
    tmpr(index ip,2) = tmpr(index ip,2) +
    Ms(index jp,2,index ip,index 2D)* tmp f half v
    tmpr(index ip,3) = tmpr(index ip,3) +
    Ms(index jp,3,index ip,index 2D)* tmp f half v
    tmpr(index ip,5) = tmpr(index ip,5) +
    Ms(index jp,4,index ip,index 2D)* tmp dfdr v +
```

~40% speed up for kernel
NESAP Case Studies  (More on Thursday)
Current deposition (particle-to-grid) and Field gather (grid-to-particle) are the most time consuming subroutines.

Large time spent in memory accesses.

Low vectorization.

NESAP Lead Ankit Bhagatwala, Mathieu Lobet
Optimization 1: Tiling (Sep 2015)

- Improve memory locality by tiling particle and grid quantities

**Former data layout in PICSAR**

- Particles randomly distributed on the global process grid
- Poor cache reuse

**Tiled layout**

- Particles grouped in tiles small enough that local particle/grid arrays fit in cache
- Particles deposit charge/current on local grid array in cache
- Reduction of local charge/current arrays in global array
- Slight extra overhead of reduction
Performance improvement from tiling

- Problem size: 80x80x80 cells
- ~10 particles per cell
Optimization 2: Vectorized current deposition

- Vectorize over particles
  - Non-contiguous memory accesses over neighboring grid points
- Vectorize over 8 neighboring grid points
  - 8x Memory overhead but substantial speedup

\[ \text{mesh} = \text{mesh} + \text{particle\_value} \]
NESAP Lead Zhengji Zhao
VASP profiling- memory bandwidth bound?
Estimating the performance impact of HBW memory to VASP code using AutoHBW tool on Edison

Edison, a Cray XC30, with dual-socket Ivy Bridge nodes interconnected with Cray’s Aries network, the bandwidths of the near socket memory (simulating MCDRAM) and the far socket memory via QPI (simulating DDR) differ by 33%
VASP+FASTMEM performance on Edison

VASP performance comparison between runs when everything was allocated in the DDR memory (blue/slow), when only a few selected arrays were allocated to HBM (red/mixed), and when everything was allocated to HBM (green/fast). The test case PdO@Pd-slab was used, and the tests were run on a single Edison node.
• Spent a lot of time threading and vectorizing app. Performance still slightly worse on KNC than Haswell

• 2S Haswell 27.9s   KNC 39.9s   (Bandwidth bound on KNC, but not on Haswell)
  
do my_igp = 1, ngpown (OpenMP)
  
do iw = 1 , 3
    
do ig = 1, igmax
      
        load wtilda_array(ig,my_igp) 819 MB, 512KB per row
        load aqsntemp(ig,n1) 256 MB, 512KB per row
        load l_eps_array(ig,my_igp) 819 MB, 512KB per row
        do work (including complex divide) depends on ig, iw ...
BerkeleyGW: Why KNC worse than Haswell for GPP Kernel?

- 2S Haswell 27.9s  KNC 39.9s  (Bandwidth bound on KNC but not on Haswell)

```java
  do my_igp = 1, ngpown (OpenMP)
    do iw = 1, 3
      do ig = 1, igmax
        load wtilede_array(iw,my_igp) 819 MB, 512KB per row
        load aqsntemp(iw,n1) 256 MB, 512KB per row
        load l_eps_array(iw,my_igp) 819 MB, 512KB per row
        do work (including divide)
  
  Required Cache size to reuse 3 times:

  1536 KB

L2 on KNC is 256 KB per Hardware Thread
L2 on Has. is 256 KB per core

L3 on Has. is 3800 KB per core
```
BerkeleyGW: Why KNC worse than Haswell for GPP Kernel?

• 2S Haswell 27.9s    KNC 39.9s    (Bandwidth bound on KNC but not on Haswell)

```plaintext
do my_igp = 1, ngpown (OpenMP)
do iw = 1 , 3
do ig = 1, igmax
  load wtilde_array(ig,my_igp) 819 MB, 512KB per row
  load aqsntemp(ig,n1) 256 MB, 512KB per row
  load l_eps_array(ig,my_igp) 819 MB, 512KB per row
  do work (including divide)
```

Required Cache size to reuse 3 times:

- 1536 KB

L2 on KNC is 256 KB per Hardware Thread
L2 on Has. is 256 KB per core

L3 on Has. is 3800 KB per core

Without blocking we spill out of L2 on KNC and Haswell. But, Haswell has L3 to catch us.
• 2S Haswell 27.9s  KNC 39.9s  (Bandwidth bound on KNC but not on Haswell)

    igblk = 2048
    do my_igp = 1, ngpown (OpenMP)
       do igbeg = 1, igmax, igblk
          do iw = 1, 3
             do ig = igbeg, min(igbeg + igblk, igmax)
                load wtilde_array(ig,my_igp) 819 MB, 512KB per row
                load aqsntemp(ig,n1) 256 MB, 512KB per row
                load l_eps_array(ig,my_igp) 819 MB, 512KB per row
                do work (including divide)

    Required Cache size to reuse 3 times:

    1536 KB

    L2 on KNC is 256 KB per Hardware Thread
    L2 on Has. is 256 KB per core
    L3 on Has. is 3800 KB per core

Without blocking we spill out of L2 on KNC and Haswell. But, Haswell has L3 to catch us.
gppkernel speedups

Igblk=2048 - to enable reuse of L2 cache on KNC

• Morning: 2S Haswell 27.9s  KNC 39.9s
• Afternoon: 2S Haswell 27.5s  KNC 29.7s

The loss of L3 on MIC makes locality more important.
Conclusions
1. Optimizing code for Cori is not always straightforward. It is a continual discovery process that involves many sequential and coupled changes.
High Level Lessons

1. Optimizing code for Cori is not always straightforward. It is a continual discovery process that involves many sequential and coupled changes.

2. Use profiling tools like VTune and CrayPat on Edison to find and characterize hotspots.

3. Understanding bandwidth and compute limitations of hotspots are key to deciding how to improve code.

4. NERSC is in a unique position to facilitate the transition of DOE science codes, with application teams and vendors.