NERSC, Cori, Knights Landing, And Other Matters

Jack Deslippe
February, 2016
Agenda

1. Overview of NERSC
2. Things I wished I knew back when I took CS267
3. Optimizing Applications For Cori Phase 2
4. Example Case Study
NERSC is the production HPC center for the DOE Office of Science

- 5000 users, 600 projects
- From 48 states; 65% from universities
- Hundreds of users each day
- 1500 publications per year

Systems designed for science
The NERSC-8 System: Cori

- Cori will support the broad Office of Science research community and begin to transition the workload to more energy efficient architectures
- Cray XC system with over 9,300 Intel Knights Landing compute nodes – mid 2016
  - Self-hosted, (not an accelerator) manycore processor with up to 72 cores per node
  - On-package high-bandwidth memory
- Data Intensive Science Support
  - 10 Haswell processor cabinets (Phase 1) to support data intensive applications – Summer 2015
  - NVRAM Burst Buffer to accelerate data intensive applications
  - 28 PB of disk, >700 GB/sec I/O bandwidth
- Robust Application Readiness Plan
  - Outreach and training for user community
  - Application deep dives with Intel and Cray
NERSC’s Current Big System is Edison

- Edison is the HPCS* demo system (serial #1)
- First Cray Petascale system with Intel processors (Ivy Bridge), Aries interconnect topology
- Very high memory bandwidth (100 GB/s per node)
- 5,576 nodes, 133K cores, 64 GB/node
- Exceptional application performance

*DARPA High Productivity Computing System program
NERSC moved into Wang Hall late 2015

- Four story, 140,000 GSF, 300 offices, 20Ksf
  HPC floor, 12.5->40 MW
- Located for collaboration
  - LBNL, CRD, Esnet, UCB
- Exceptional energy efficiency
  - Natural air and water cooling
  - Heat recovery
  - PUE < 1.1
Things I Wish I Knew When I was In CS267
This code hangs because both Task 0 and Task N-1 are blocking on MPI_Recv

```c
if(task_no==0) {
    ret = MPI_Recv(&herBuffer, 50, MPI_DOUBLE, totTasks-1, 0, MPI_COMM_WORLD, &status);
    ret = MPI_Send(&myBuffer, 50, MPI_DOUBLE, totTasks-1, 0, MPI_COMM_WORLD);
} else if (task_no==(totTasks-1)) {
    ret = MPI_Recv(&herBuffer, 50, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
    ret = MPI_Send(&myBuffer, 50, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
}
```
NERSC NX – Accelerate Your X Connection

NERSC NX SERVICE - X-WINDOWS ACCELERATION AT NERSC

Introduction

NX is a computer program that handles remote X Window System connections and it provides three benefits for NERSC users:

- **SPEED**: NX can greatly improve the performance of X Windows, allowing users with slow, high latency connections (e.g., on cell phone network, traveling in Africa) to use complex X Windows programs (such as rotating a plot in Matlab).
- **SESSION**: NX provides sessions that allow a user to disconnect from the session and reconnect to it at a later time while keeping the state of all running applications inside the session.
- **DESKTOP**: NX gives users a virtual desktop that’s running at NERSC. You can customize the desktop according to your work requirement.
Compile & Start DDT

Compile for debugging

```
edison% make
c -c -g hello.c
c -o hello -g hello.o
```

Set up the parallel run environment

```
edison% qsub -I -V -lmppwidth=24
edison% cd $PBS_O_WORKDIR
```

Start the DDT debugger

```
edison% module load ddt
edison% ddt ./hello
```
At hang, tasks are in 3 different places.

Task 0 is at line 44

Press Go and then Pause when code appears hung.
Vendors are starting to listen (DDT)
Debuggers on NERSC machines

- **Parallel debuggers with a graphical user interface**
  - DDT (Distributed Debugging Tool)
  - TotalView

- **Specialized debuggers on Hopper and Edison**
  - STAT (Stack Trace Analysis Tool)
    - Collect stack backtraces from all (MPI) tasks
  - ATP (Abnormal Termination Processing)
    - Collect stack backtraces from all (MPI) tasks when an application fails
  - CCDB (Cray Comparative Debugger)
    - Comparative debugging
Profile Your Application (VTune / CrayPat)
Cori Phase 2
Cori will begin to transition the workload to more energy efficient architectures

Cray XC system with over 9300 Intel Knights Landing (Xeon-Phi) compute nodes
  - Self-hosted, (not an accelerator) manycore processor with 72 cores per node
  - On-package high-bandwidth memory

System named after Gerty Cori, Biochemist and first American woman to receive the Nobel prize in science.
### 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
- \( \sim 100 \) 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 \( \sim 4-5x \) DDR4 Bandwidth
Breakdown of Application Hours on Hopper and Edison 2013
Basic Optimization Concepts
Need to explicitly consider both inter and on-node parallelism in application.

Existing applications may suffer from:
- Memory overhead due to duplicated data in traditional MPI tasks
- Lack of SIMD/Vectorization expressiveness in app.
- Potential MPI latency in all-to-all communication patterns

Possible Solutions:
MPI+MPI, MPI+OpenMP, PGAS (MPI+PGAS), Task Based Programming
PARATEC computes parallel FFTs across all processors.

Involves MPI all-to-all communication (small messages, latency bound).

Reducing the number of MPI tasks in favor OpenMP threads makes large improvement in overall runtime.
Vectorization

There is another important form of on-node parallelism:

\[
\begin{align*}
\text{do } i = 1, n \\
\quad a(i) &= b(i) + c(i) \\
\text{enddo}
\end{align*}
\]

\[
\begin{pmatrix}
  a_1 \\
  \vdots \\
  a_n
\end{pmatrix} =
\begin{pmatrix}
  b_1 \\
  \vdots \\
  b_n
\end{pmatrix} +
\begin{pmatrix}
  c_1 \\
  \vdots \\
  c_n
\end{pmatrix}
\]

Vectorization: CPU does identical operations on different data; e.g., multiple iterations of the above loop can be done concurrently. Works best with long/aligned vectors.
There is another important form of on-node parallelism.

$$\begin{align*}
\text{do } i = 1, n \\
a(i) &= b(i) + c(i) \\
\text{enddo}
\end{align*}$$

$$\begin{pmatrix} a_1 \\ \vdots \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \end{pmatrix} + \begin{pmatrix} c_1 \\ \vdots \end{pmatrix}$$

**Intel Xeon Sandy-Bridge/Ivy-Bridge:** 4 Double Precision Ops Concurrently

**Intel Xeon Phi:** 8 Double Precision Ops Concurrently
Things that prevent vectorization in your code

Compilers want to “vectorize” your loops whenever possible. But sometimes they get stumped. Here are a few things that prevent your code from vectorizing:

Loop dependency:

```
do i = 1, n
   a(i) = a(i-1) + b(i)
endo
```

Task forking:

```
do i = 1, n
   if (a(i) < x) cycle
   if (a(i) > x) ...
endo
```
for (many iterations) {
  ... many flops ... 
  et = exp(outcome1)
  tt = pow(outcome2,3)
  IN = IN * et +tt
}

Example From NERSC User Group Hackathon - (Astrophysics Transport Code)
for (many iterations) {
    ... many flops ...
    et = exp(outcome1)
    tt = pow(outcome2,3)
    IN = IN * et + tt
}

for (many iterations) {
    ... many flops ...
    et(i) = exp(outcome1)
    tt(i) = pow(outcome2,3)
}
for (many iterations) {
    IN = IN * et(i) + tt(i)
}
for (many iterations) {
    … many flops …
    et = exp(outcome1)
    tt = pow(outcome2,3)
    IN = IN * et + tt
}

30% speed up for entire application!
Things that prevent vectorization in your code

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_Nrm1
    index_2dp = index_jp+mesh_Nrm1*(index_ip-1)
    
    tmp_vol = cs2%local_center_volume(index_jp)
    tmp_f_half_v = f_half(index_jp, index_ip) * tmp_vol
    
    tmp_dfr_v = dfr(index_jp, index_ip) * tmp_vol
    tmp_dfdz_v = dfdz(index_jp, index_ip) * tmp_vol
    
    tmp_r(1:3) = tmp_r(1:3) +
    Ms(1:3,index_2dp,index_2D)* tmp_f_half_v
    tmp_r(5) = tmp_r(5) +
    Ms(4,index_2dp,index_2D)*tmp_dfr_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_Nrm1
    index_2dp = index_jp+mesh_Nrm1*(index_ip-1)
    tmp_vol = cs2%local_center_volume(index_jp)
    tmp_f_half_v = f_half(index_jp, index_ip) * 
    tmp_vol
    
    tmp_dfr_v = dfr(index_jp, index_ip) * tmp_vol
    tmp_dfdz_v = dfdz(index_jp, index_ip) * tmp_vol
    
    tmp_r(1:3) = tmp_r(1:3) +
    Ms(index_jp,1,index_ip,1,INDEX_2D) * 
    Ms(index_jp,1,index_ip,INDEX_2D)* 
    tmp_f_half_v
    tmp_r(5) = tmp_r(5) +
    Ms(index_jp,4,index_ip,INDEX_2D)* 
    tmp_dfr_v +

Things that prevent vectorization in your code

Example From Cray COE Work on XGC1

~40% speed up for kernel
Consider the following loop:

```
  do i = 1, n
    do j = 1, m
      c = c + a(i) * b(j)
    enddo
  enddo
```

Assume, $n$ & $m$ are very large such that $a$ & $b$ don’t fit into cache.

Then,

During execution, the number of loads From DRAM is $n*m + n$
Consider the following loop: Assume, n & m are very large such that a & b don’t fit into cache.

```fortran
  do i = 1, n
    do j = 1, m
      c = c + a(i) * b(j)
    enddo
  enddo
```

Assume, n & m are very large such that a & b don’t fit into cache.

Then,

During execution, the number of loads From DRAM is

\[ n \times m + n \]

Requires 8 bytes loaded from DRAM per FMA (if supported). Assuming 100 GB/s bandwidth on Edison, we can at most achieve 25 GFlops/second (2 Flops per FMA).

Much lower than 460 GFlops/second peak on Edison node. Loop is memory bandwidth bound.
Roofline Model For Edison

Edison Node Roofline Based on Stream of 85GB/s and Peak Flops of 460 GFlop/Sec

- Roofline
- Unbalanced Ceiling
- Unbalanced No SIMD Ceiling

Attainable GFlops/Sec vs. Operational Intensity (Flops/Byte)
Improving Memory Locality

Improving Memory Locality. Reducing bandwidth required.

Loads From DRAM:

\[ n \times m + n \]

Improving Memory Locality.

Loads From DRAM:

\[ \frac{m}{\text{block}} \times (n + \text{block}) = \frac{n \times m}{\text{block}} + m \]
Improving Memory Locality Moves you to the Right on the Roofline

Edison Node Roofline Based on Stream of 86GB/s and Peak Flops of 460 GFlop/Sec

- Roofline
- Unbalanced Ceiling
- Unbalanced No SIMD Ceiling

Attainable GFlops/Sec

Operational Intensity (Flops/Byte)
Optimization Strategy
OpenMP scales only to 4 Threads

Communication dominates beyond 100 nodes

large cache miss rate

Code shows no improvements when turning on vectorization

MPI/OpenMP Scaling Issue

50% Walltime is IO

IO bottlenecks

Memory bandwidth bound kernel

Can you use a library?

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

The Dungeon: Simulate kernels on KNL. Plan use of on-package memory, vector instructions.

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

Utilize performant / portable libraries

Use Edison to Test/Add OpenMP

Improve Scalability. Help from NERSC/Cray COE Available.

The Ant Farm!

Compute intensive doesn't vectorize

Increase Memory Locality

Increase Memory Locality

The Dungeon:
Can You Increase Flops Per Byte Loaded From Memory in Your Algorithm?

Make Algorithm Changes

Run Example in “Half Packed” Mode

Is Performance affected by Half-Packing?

Yes

Your Code is at least Partially Memory Bandwidth Bound

No

Make Algorithm Changes

Run Example at “Half Clock” Speed

Is Performance affected by Half-Clock Speed?

Yes

You are at least Partially CPU Bound

No

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

Make Sure Your Code is Vectorized!

Measure Cycles Per Instruction with VTune

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

Your Code is at least Partially Memory Bandwidth Bound

- Yes: Explore Using HBM on Cori For Key Arrays
- No: Make Sure Your Code is Vectorized! Measure Cycles Per Instruction with VTune

You are at least Partially CPU Bound

- Yes: Can You Reduce Memory Requests Per Flop In Algorithm?
- No: Try Running With as Many Virtual Threads as Possible (> 240 Per Node on Cori)

Likely Partially Memory Latency Bound (assuming not IO or Communication Bound)
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 in “Half Packed” Mode
  - Is Performance affected by Half-Packing?
    - Yes: Your Code is at least Partially Memory Bandwidth Bound
    - No: Run Example at “Half Clock” Speed
  - No: Run Example at “Half Clock” Speed
    - Is Performance affected by Half-Clock Speed?
      - Yes: You are at least Partially CPU Bound
      - No: Likely Partially Memory Latency Bound (assuming not IO or Communication 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

```
aprun -n 24 -N 12 -S 6 ...
```

VS

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

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

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.

```
aprun -n 24 -N 12 -S 6 ...
aprun -n 24 -N 24 -S 12 ...
```

If your performance is not bound, try this:
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.

```
aprun --p-state=2400000 ...
```

VS

```
aprun --p-state=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.
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
Complex-Division (without -fp model fast=2)
So, you are neither compute nor memory bandwidth bound?

You may be memory latency bound (or you may be spending all your time in IO and Communication).

If running with hyper-threading on Edison improves performance, you *might* be latency bound:

```
	aprun -j 2 -n 48 ....  
```

VS

```
	aprun -n 24 ....  
```

If you can, try to reduce the number of memory requests per flop by accessing contiguous and predictable segments of memory and reusing variables in cache as much as possible.

On Cori, each core will support up to 4 threads. Use them all.
NESAP Case Study
BerkeleyGW Use Case

★ Big systems require more memory. Cost scales as $N_{\text{atoms}}^2$ to store the data.
★ In an MPI GW implementation, in practice, to avoid communication, data is duplicated and each MPI task has a memory overhead.
★ Users sometimes forced to use 1 of 24 available cores, in order to provide MPI tasks with enough memory. **90% of the computing capability is lost.**
In house code (I’m one of main developers). Use as “prototype” for App Readiness.
In house code (I’m one of main developers). Use as “prototype” for App Readiness.

Significant Bottleneck is large matrix reduction like operations. Turning arrays into numbers.

\[
\langle n \kvec | \Sigma_{\mathrm{CH}}^\text{C} (E') | n' \kvec \rangle = \frac{1}{2} \sum_{n''} \sum_{\mathbf{q} \mathbf{GG'}^\text{C}} M_{n''n}^* (\kvec, -\mathbf{q}, -\mathbf{G}) M_{n''n'} (\kvec, -\mathbf{q}, -\mathbf{G'}) \\
\times \frac{\Omega_{\mathbf{GG'}}^2 (\mathbf{q}) (1 - i \tan \phi_{\mathbf{GG'}} (\mathbf{q}))}{\tilde{\omega}_{\mathbf{GG'}} (\mathbf{q}) (E - E_{n'' \kvec - \mathbf{q}} - \tilde{\omega}_{\mathbf{GG'}} (\mathbf{q}))} v(\mathbf{q} + \mathbf{G'})
\]
1. Target more on-node parallelism. (MPI model already failing users)
2. Ensure key loops/kernels can be vectorized.

Example: Optimization steps for Xeon Phi Coprocessor

Refactor to Have 3 Loop Structure:
- Outer: MPI
- Middle: OpenMP
- Inner: Vectorization

Add OpenMP
Ensure Vectorization
ngpown typically in 100’s to 1000s. Good for many threads.

ncouls typically in 1000s - 10,000s. Good for vectorization.

Original inner loop. Too small to vectorize!

Attempt to save work breaks vectorization and makes code slower.
We’ve had two dungeon sessions with Intel. What kinds of questions can they help with?

1. Why is KNC slower than Haswell for this problem?

   - Known to be bandwidth bound on KNC, but not on Haswell.
   - How is it bound on Haswell?

2. How much gain can we get by allocating just a few arrays in HBM?
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 I_eps_array(ig,my_igp) 819 MB, 512KB per row
    do work (including complex divide) depends on ig, iw ...
```
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)
doiw = 1, 3
doi = 1, igmax
   load wtilde_array(ig,my_igp) 819 MB, 512KB per row
   load aqsntemp(ig,n1) 256 MB, 512KB per row
   load I_eps_array(ig,my_igp) 819 MB, 512KB per row
   do work (including divide)
```

**Required Cache size to reuse 3 times:**

- 1536 KB

**L2 Cache:**
- L2 on KNC is 256 KB per Hardware Thread
- L2 on Has. is 256 KB per core

**L3 Cache:**
- L3 on Has. is 3800 KB per core
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 I_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.
Why KNC worse than Haswell for GPP Kernel?

• 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 I_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

lgbblk=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.
How much performance can we get from 3 arrays in Fast Memory?

- **Identify the candidate (key arrays) for HBM**
  - VTune Memory Access tool can help to find key arrays
  - Using NUMA affinity to simulate HBM on a dual socket system
  - Use FASTMEM directives and link with jemalloc/memkind libraries

On Edison (NERSC Cray XC30):

```fortran
real, allocatable :: a(:,,:), b(:,,:), c(:)
!DIR$ ATTRIBUTE FASTMEM :: a, b, c
module load memkind jemalloc
% ftn -dynamic -g -O3 -openmp mycode.f90
% export MEMKIND_HBW_NODES=0
% aprun -n 1 -cc numa_node numactl --membind=1 --cpunodebind=0 ./myexecutable
```

On Haswell:

```
Link with '-ljemalloc -lmemkind -lpthread -lnuma’
% numactl --membind=1 --cpunodebind=0 ./myexecutable
```

<table>
<thead>
<tr>
<th>Application</th>
<th>All memory on far memory</th>
<th>All memory on near memory</th>
<th>Key arrays on near memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>BerkeleyGW</td>
<td>baseline</td>
<td>52% faster</td>
<td>52.4% faster</td>
</tr>
<tr>
<td>EmGeo</td>
<td>baseline</td>
<td>40% faster</td>
<td>32% faster</td>
</tr>
<tr>
<td>XGC1</td>
<td>baseline</td>
<td>24% faster</td>
<td></td>
</tr>
</tbody>
</table>
General Exploration of two OpenMP regions

The dynamic loop is now core bound, not memory bound. Removing the divide shows it to be the culprit!
Conclusions
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.
2013 - Poor locality, loop ordering issues
2014 - Refactored loops, improved locality

Edison Node Roofline Based on Stream of 89GB/s and Peak Flops of 460 GFlop/Sec

- Roofline
- Unbalanced Ceiling
- Unbalanced No SIMD Ceiling
2014 - Vectorized Code

Edison Node Roofline Based on Stream of 89GB/s and Peak Flops of 460 GFlop/Sec

- Roofline
- Unbalanced Ceiling
- Unbalanced No SIMD Ceiling

Disclaimer - this is a rough schematic
2015 - Cache Blocking

Edison Node Roofline Based on Stream of 89GB/s and Peak Flops of 460 GFlop/Sec

- Roofline
- Unbalanced Ceiling
- Unbalanced No SIMD Ceiling

Operational Intensity (Flops/Byte)

Attainable GFlops/Sec

Disclaimer - this is a rough schematic
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.
The End (Extra Slides)
Why Complex Divides so Slow?

Code performance now limited by complex divides

why??

For complex division in performance critical loop, I had already removed the explicit complex divide but what is faster?

a) \( c = 1 / c \) vs. b) 
   - \( r = c \times \text{conjg}(c) \)
   - \( r = 1 / r \)
   - \( c = \text{conjg}(c) \times r \)

c/d) Compiling with/without -fp-model fast=2
Real-Division (with or without -fp model fast=2)
Complex-Division (with -fp model fast=2)
Approximation:

a. Real Division

b. Complex Division

c. Complex Division + -fp-model fast=2

Wall Time:

6.37 seconds
4.99 seconds
5.30 seconds
Approximation:

a. Real Division

b. Complex Division

c. Complex Division + -fp-model=fast

Wall Time:

6.37 seconds

4.99 seconds

5.30 seconds
Approximation:

a. Real Division
b. Complex Division
c. Complex Division + -
   fp-model fast=2
d. Complex Division + -
   fp-model=fast=2 + !
   dir$ nounroll

Wall Time:

6.37 seconds
4.99 seconds
5.30 seconds
4.89 seconds
Early NESAP (Advances with Cray and Intel) Advances

### Overall Improvement

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>BGW GPP Kernel</td>
<td>0-10%</td>
</tr>
<tr>
<td>BGW FF Kernel</td>
<td>2x-4x</td>
</tr>
<tr>
<td>BGW Chi Kernel</td>
<td>10-30%</td>
</tr>
<tr>
<td>BGW BSE Kernel</td>
<td>10-50%</td>
</tr>
</tbody>
</table>

#### Notes

- **BGW GPP Kernel**: Pretty optimized to begin with. Thread scalability improved by fixing ifort allocation performance.
- **BGW FF Kernel**: Unoptimized to begin with. Cache reuse improvements.
- **BGW Chi Kernel**: Moved threaded region outward in code.
- **BGW BSE Kernel**: Created custom vector matmuls.
Cray and Intel very helpful in profiling/optimizing the code. See following slides for using Intel resources effectively.

Generating small tangible kernels is important for success.

Targeting Many-Core greatly helps performance back on Xeon.

Complex division is slow on (particularly on KNC).