Optimizing Codes For Many-Core at NERSC

Rebecca Hartman-Baker + Jack Deslippe

July, 2015
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 over 60 cores per node
- On-package high-bandwidth memory

Data Intensive Science Support
- NVRAM Burst Buffer to accelerate applications
- 28PB of disk and >700 GB/sec I/O bandwidth

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
- ~100 GB/s Memory Bandwidth

Cori (Knights-Landing):

- 60+ Physical Cores Per CPU
- 240+ 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

What is different about Cori?
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):**
- 60+ Physical Cores Per CPU
- 240+ 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
Basic Optimization Concepts
MPI Vs. OpenMP For Multi-Core Programming

**MPI**

- CPU Core
  - Memory
- Private Arrays
  - CPU Core
  - Memory
- Network Interconnect
  - CPU Core
  - Memory

**OpenMP**

- CPU Core
- CPU Core
- CPU Core
- CPU Core
- Memory, Shared Arrays etc.

Typically less memory overhead/duplication. Communication often implicit, through cache coherency and runtime.
INTEGER I, N
REAL A(100), B(100), TEMP, SUM

!$OMP PARALLEL DO PRIVATE(TEMP) REDUCTION(+:SUM)
DO I = 1, N
   TEMP = I * 5
   SUM = SUM + TEMP * (A(I) * B(I))
ENDDO

https://computing.llnl.gov/tutorials/openMP/exercise.html
Vectorization

There is another important form of on-node parallelism:

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

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

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

Vectorization:

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

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

- Intel Xeon Sandy-Bridge/Ivy-Bridge: 4 Double Precision Ops Concurrently
- Intel Xeon Phi: 8 Double Precision Ops Concurrently
- NVIDIA Kepler GPUs: 32+ SIMT
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:**

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

**Task forking:**

```plaintext
do i = 1, n
    if (a(i) < x) cycle
    if (a(i) > x) ...
enddo
```
Consider the following loop:

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

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

\[n \times m + n\]
Consider the following loop: Assume, $n$ & $m$ are very large such that $a$ & $b$ don’t fit into cache.

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

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 $n*m + n$.

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:

\[ n \times m + n \]

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

Loads From DRAM:

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

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)
Optimizing Code For Cori is like:

A. A Staircase?

B. A Labyrinth?

C. A Space Elevator?
OpenMP scales only to 4 Threads

Communication dominates beyond 100 nodes

Large cache miss rate

Code shows no improvements when turning on vectorization

50% Walltime is IO

IO bottlenecks

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.

Use Edison to Test/Add OpenMP

Improve Scalability.

Help from NERSC/Cray COE Available.

Utilize performant / portable libraries

Can you use a library?

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

Increase Memory Locality

Memory bandwidth bound kernel

Compute intensive doesn’t vectorize

Communication dominates beyond 100 nodes

An Ant Farm!

Code shows no improvements when turning on vectorization

50% Walltime is IO

IO bottlenecks
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.
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
Steps:

0. Use NX To Login Onto Babbage
https://www.nersc.gov/users/connecting-to-nersc/using-nx/

1. Get Code:
% git clone https://github.com/NERSC/training.git

2. Build Code:
% cd training/hackathon-201502/BGW
% ifort -g -O3 -xAVX -openmp bgw.f90 -o bgw.x

3. Run Code (interactively):
% qsub -I -X -q regular -l nodes=1 -l walltime=00:30:00 -l advres=csgf.90
% cd $PBS_O_WORKDIR
% ./bgw.x

4. Collect hotspots with vtune (in an interactive session):
% module load vtune
% amplxe-cl -collect hotspots -r test_1 -- ./bgw.x

5. To View Vtune Results do:
% amplxe-gui

Question 1 - Can you make the code faster by adding OpenMP to any hot loop?
!$OMP PARALLEL do private (...) ...

6. Collect collect bandwidth information d (in an interactive session):
% amplxe-cl -collect bandwidth -r test_2 -- ./bgw.x
… then view the output in the GUI

Question 2 - Is the code memory bandwidth bound?

Question 3 - Can you improve the code performance further through any optimization strategy described at the beginning of the session?