Optimizing Applications: an Ant Farm Approach

Jack Deslippe
August 2016
Agenda

1. Many Core vs Multi Core
2. Performance Optimization Concepts for Many Core
3. Performance Optimization Strategy for Many Core
4. Example Case Studies
Many Core HPC Systems
NERSC’s 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 68 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.
What is different about Many-Core

Edison (Multi-Core):
- 5000+ Ivy Bridge Nodes
- 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 (Many-Core):
- 9000+ Knights Landing Nodes
- 68 Physical Cores Per CPU
- Up to 272 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 ~ 4-5x DDR4 Bandwidth
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.

Figure Courtesy of Andrew Canning
There is another important form of on-node parallelism:

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

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

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.
Vectorization

There is another important form of on-node parallelism

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

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

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

Intel Xeon Phi: 8 Double Precision Ops Concurrently
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:**

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

**Task forking:**

```fortran
do i = 1, n
    if (a(i) < x) cycle
    if (a(i) > x) ...
enddo
```
Things that prevent vectorization in your code

Example From NERSC User Group Hackathon - (Astrophysics Transport Code)

```c
for (many iterations) {
    ... many flops ...
    et = exp(outcome1)
    tt = pow(outcome2,3)
    IN = IN * et + tt
}
```
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_Nzm1
  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

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_Nzm1
  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 +
    Ms(index_jp,5,index_ip,index_2D)* tmp_dfdz_v,
Things that prevent vectorization in your code

Original

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

do index_ip = 1, mesh_Nzml
  do index_jp = 1, mesh_Nrpm
    index_2dp = index_jp + mesh_Nrpm * (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_r(1:3) = tmp_r(1:3) + Ms(1:3, index_2dp, index_2D) * tmp_f_half_v
    tmp_r(4) = tmp_r(4) + Ms(4, index_2dp, index_2D) * tmp_dfdr_v +
```

Optimized

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

do index_ip = 1, mesh_Nzml
  do index_jp = 1, mesh_Nrpm
    index_2dp = index_jp + mesh_Nrpm * (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_r(index_jp, 1) = tmp_r(index_jp, 1) +
      Ms(index_jp, index_ip, 2D) * tmp_f_half_v
    tmp_r(index_jp, 2) = tmp_r(index_jp, 2) +
      Ms(index_jp, index_ip, 2D) * tmp_f_half_v
    tmp_r(index_jp, 3) = tmp_r(index_jp, 3) +
      Ms(index_jp, index_ip, 2D) * tmp_f_half_v
    tmp_r(index_jp, 4) = tmp_r(index_jp, 4) +
      Ms(index_jp, index_ip, 2D) * tmp_dfdr_v +
```

Example From Cray COE Work on XGC1

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

```plaintext
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 \(n \times m + n\).
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

\[ n \times m + n \]

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

Operational Intensity (Flops/Byte)
Processor Memory Hierarchy on MultiCore

CPU
L1 Cache
L2 Cache
L3 Cache

CPU
L1 Cache
L2 Cache

CPU
L1 Cache
L2 Cache

...
Improving Memory Locality

Improving Memory Locality. Reducing bandwidth required.

Loads From DRAM:

\[ n \times m + n \]

To improve locality, we can modify the loop:

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

Loads From DRAM:

\[ m / \text{block} \times (n + \text{block}) = n \times m / \text{block} + m \]
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

50% Walltime is IO

IO bottlenecks

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.

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

Utilize performant / portable libraries

The Ant Farm!

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 Increase Flops Per Byte Loaded From Memory in Your Algorithm?

- Make Algorithm Changes
- Explore Using HBM on Cori For Key Arrays

- Is Performance affected by Half-Clock Speed?
  - Yes
    - Run Example at “Half Clock” Speed
    - Is Performance affected by Half-Packing?
      - Yes
        - Your Code is at least Partially Memory Bandwidth Bound
      - No
        - Run Example in “Half Packed” Mode
        - Is Performance affected by Half-Packing?
          - No
            - Your Code is at least Partially Memory Bandwidth Bound
          - Yes
            - You are at least Partially CPU Bound

- Likely Partially Memory Latency Bound (assuming not IO or Communication Bound)
Can You Increase Flops Per Byte Loaded From Memory in Your Algorithm?

Yes

Explore Using HBM on Cori For Key Arrays

No

Your Code is at least Partially Memory Bandwidth Bound

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

Yes

Make Sure Your Code is Vectorized! Measure Cycles Per Instruction with VTune

No

You are at least Partially CPU Bound

Try Running With as Many Virtual Threads as Possible (> 240 Per Node on Cori)

No

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

Can You Reduce Memory Requests Per Flop In Algorithm?

Yes

No
1. Determine your roofline position:

http://www.nersc.gov/users/application-performance/measuring-arithmetic-intensity/
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

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:

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

VS

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

If you run on only half of the cores on a node, each core you do run has access to more bandwidth.
Are you memory or compute bound? Or both?

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

Run Example at “Half Clock” Speed

\[ \text{aprun --p-state=2400000 \ldots} \quad \text{VS} \quad \text{aprun --p-state=1900000 \ldots} \]

If your performance changes, you are at least partially compute 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`
High latency instructions: Complex-Division (without -fp model fast=2)
Are you latency 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
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.**
Computational Bottlenecks

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.

\[
\langle n'k | \Sigma_{CH}(E) | n'k \rangle = \frac{1}{2} \sum_{n''} \sum_{qGG'} M^*_{n'n}(k, -q, -G') M_{n''n'}(k, -q, -G')
\]

\[
\times \frac{\Omega^2_{GG'}(q) (1 - i \tan \phi_{GG'}(q))}{\tilde{\omega}_{GG'}(q) (E - E_{n''k-q} - \tilde{\omega}_{GG'}(q))} v(q+G')
\]
Optimization Path

Optimization process for Kernel-C (Sigma code):

1. Refactor (3 Loops for MPI, OpenMP, Vectors)
2. Add OpenMP
3. Initial Vectorization (loop reordering, conditional removal)
4. Cache-Blocking
5. Improved Vectorization
6. Hyper-threading
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.
The loss of L3 on MIC makes locality more important.
Why KNC worse than Haswell for GPP Kernel?

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

```fortran
!$OMP DO
do my_igp = 1, ngpown
   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
      enddo
   enddo
enddo
```

Required Cache size to reuse 3 times:
- 1536 KB
- L2 on KNL is 512 KB per core
- 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)

```
!$OMP DO
do my_igp = 1, ngpown
    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
            enddo
        enddo
    enddo
enddo
```

Required Cache size to reuse 3 times:

- L2 on KNL is 512 KB per core
- 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.
Cache Blocking Optimization

Haswell Roofline Optimization Path

- Peak
- ILP
- AVX
- BGW

GFLOP/s vs. Arithmetic Intensity

KNL Roofline Optimization Path

- Peak (HBM)
- Peak (DDR)
- ILP (HBM)
- ILP (DDR)
- AVX (HBM)
- AVX (DDR)
- BGW (DDR)
- BGW (HBM)

GFLOP/s vs. Arithmetic Intensity
Why Complex Divides so Slow?

Found significant x87 instructions from 1/complex_number instead of AVX/AVX-512

Can significantly speed up by

a) Doing complex divide manually

Or

b) Using -fp-model fast=2
Additional Speedups from Hyperthreading

Haswell Roofline Optimization Path

KNL Roofline Optimization Path

Arithmetic Intensity

GFLOP/s
Conclusions
1. Optimizing code is not always straightforward. It is a continual discovery process that involves many sequential and coupled changes.

2. Use profiling tools 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)
Thread Scaling

KNL DDR performance saturates at around 50 threads, becomes memory bandwidth limited.

KNL MCDRAM performance beats dual socket Haswell by 63%.
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)
\[
\begin{align*}
    r &= c \times \text{conjg}(c) \\
    r &= 1 / r \\
    c &= \text{conjg}(c) \times r
\end{align*}
\]

\( \text{c/d) Compiling with/without -fp-model fast=2} \)
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.
Breakdown of Application Hours on Hopper and Edison 2013
Early Lessons Learned

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).