Guiding Optimization on with the Roofline Model

February, 2020
NERSC’s Challenge

How to Enable NERSC’s diverse community of 7,000 users, 750 projects, and 700 codes to run on advanced architectures like Cori, Perlmutter and beyond?
What was different about Cori?

<table>
<thead>
<tr>
<th>Edison (&quot;Ivy Bridge&quot;)</th>
<th>Cori (&quot;Knights Landing&quot;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5576 nodes</td>
<td>9304 nodes</td>
</tr>
<tr>
<td>24 physical cores per node</td>
<td>68 physical cores per node</td>
</tr>
<tr>
<td>48 virtual cores per node</td>
<td>272 virtual cores per node</td>
</tr>
<tr>
<td>2.4 - 3.2 GHz</td>
<td>1.4 - 1.6 GHz</td>
</tr>
<tr>
<td>8 double precision ops/cycle</td>
<td>32 double precision ops/cycle</td>
</tr>
<tr>
<td>64 GB of DDR3 memory (2.5 GB per physical core)</td>
<td>16 GB of fast memory, 96 GB of DDR4 memory</td>
</tr>
<tr>
<td>~100 GB/s Memory Bandwidth</td>
<td>Fast memory has 400 - 500 GB/s</td>
</tr>
<tr>
<td></td>
<td>No L3 Cache</td>
</tr>
</tbody>
</table>
Optimization Challenges For Scientists

Science teams need a simple way to wrap their heads around performance when main focus is scientific productivity:

1. Need a sense of absolute performance when optimizing applications.
   - How Do I know if My Performance is Good?
   - Why am I not getting peak performance advertised
   - How Do I know when to stop?

2. Many potential optimization directions:
   - How do I know which to apply?
   - What is the limiting factor in my app’s performance?
   - Again, how do I know when to stop?
Optimizing Code For Cori is like:

A. A Staircase?

A. A Labyrinth?

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

large cache miss rate

Code shows no improvements when turning on vectorization

Communication dominates beyond 100 nodes

50% Walltime is IO

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

Utilize performant / portable libraries

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

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

The Ant Farm!
Communication dominates beyond 100 nodes

Compute intensive doesn’t vectorize

MPI/OpenMP Scaling Issue

IO bottlenecks

50% Walltime is IO

Compute intensive doesn’t vectorize

IO bottlenecks

50% Walltime is IO

Compute intensive doesn’t vectorize

IO bottlenecks

50% Walltime is IO

Communication dominates beyond 100 nodes

IO bottlenecks
Roofline helps visualize this information! Guides optimizations

WARP Optimizations:
1. Add tiling over grid targeting L2 cache on both Xeon-Phi Systems
1. Add particle sorting to further improve locality and memory access pattern
1. Apply vectorization over particles
NESAP Example
BerkeleyGW

- A massively parallel package for GW calculations
- Sits on top of DFT codes

- Computational motifs
  - FFTs
  - Dense linear algebra
  - Large reductions
Sigma-GPP

Pseudo Code

```
  do n1 = 1, nbands  n'  e.g. 2763
    do igp = 1, ngpown  G'  e.g. 6633
      do ig = 1, ncouls  G  e.g. 26529
        do iw = 1, nw  E  e.g. 3
          compute:  1. mixed data types
                    e.g. complex double, double, integer
                    2. various memory access patterns
                       e.g. (ig,igp)(ig,n1)(igp,n1)(iw,n1)(n1)
                    3. complex number divisions
                    4. nw is very small, will be unrolled

          reduction:  1. complex numbers
                       2. all top 3 loops, billions of iterations
```
BerkeleyGW NESAP Project Optimization Path

Optimization process for GPP Kernel

1. Add OpenMP
2. Initial Vectorization (loop reordering, conditional removal)
3. Cache-Blocking
4. Improved Vectorization (Divides)
5. Hyper-threading
Vectorization

ngpown typically in 100’s to 1000s. Good for many threads.

Original inner loop. Too small to vectorize!

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

Attempt to save work breaks vectorization and makes code slower.

```c
!$OMP DO reduction(+:achtetmp)
do my_igp = 1, ngpown
...
do iw=1,nfreq ! nfreq is 3
    scht=0D0
    wxt = wx_array(iw)
do ig = 1, ncouls
    !if (abs(wtilde_array(ig,my_igp) * eps(ig,my_igp)) .lt. TOL) cycle
    wdiff = wxt - wtilde_array(ig,my_igp)
    delw = wtilde_array(ig,my_igp) / wdiff
    ...
    scha(ig) = mygpvar1 * aqsntemp(ig) * delw * eps(ig,my_igp)
    scht = scht + scha(ig)
endo ! loop over g
    sch_array(iw) = sch_array(iw) + 0.5D0*scht
endo
achtetmp(:) = achtetmp(:) + sch_array(:) * vcoul(my_igp)
enddo
```
Change in Roofline

The loss of L3 on MIC makes locality more important.
Why KNC worse than Haswell for GPP Kernel?

!$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  
    do work (including divide)  

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 KNL and Haswell. But, Haswell has L3 to catch us.
Why KNC worse than Haswell for GPP Kernel?

!$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
            end do
        end do
    end do
end do

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 KNL and Haswell. But, Haswell has L3 to catch us.
Cache Blocking Optimization

Haswell Roofline Optimization Path

- Peak
- ILP
- AVX
- BGW

KNL Roofline Optimization Path

- Peak (HBM)
- Peak (DDR)
- ILP (HBM)
- ILP (DDR)
- AVX (HBM)
- AVX (DDR)
- BGW (DDR)
- BGW (HBM)
Cache Blocking Optimization (Hierarchical Roofline)

Original Code

Cache-Blocking Code
Additional Speedups from Hyperthreading

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
### GPP on GPUs in 8 Steps

1. Collapse $n'$, $G'$, and $G$ loops
2. Bring $n'$ loop in; collapse only $G'$ and $G$
3. Adjust threadblock size
4. Reduce branching; pull iw loop outside
5. Swap indices to suite parallelisation
6. Simplify code
7. Replace div. with rcp. and mul.
8. Replace abs with power of 2
9. Cache blocking

<table>
<thead>
<tr>
<th></th>
<th>TFLOPs</th>
<th>Time</th>
<th>TFLOP/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1.collapse3</td>
<td>3.71</td>
<td>1.63</td>
<td>2.27</td>
</tr>
<tr>
<td>v9.block</td>
<td>2.00</td>
<td>0.57</td>
<td>3.50</td>
</tr>
</tbody>
</table>

![Graph showing performance improvements](image-url)
V1. Naïve Implementation

- **Collapse the first 3 loops to gain parallelism**

```c
!$ACC PARALLEL LOOP COLLAPSE(3) REDUCTION(+: )
do n1 = 1, nbands
  do igp = 1, ngpown
    do ig = 1, ncouls
      do iw = 1, nw  #unrolled
        compute and reduction
```

<table>
<thead>
<tr>
<th>TFLOPs</th>
<th>Time (sec)</th>
<th>TFLOP/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1.collapse3</td>
<td>3.71</td>
<td>1.63</td>
</tr>
</tbody>
</table>
V2. More Compute Per Thread

- **Move n’ loop in, and collapse the first 2 loops**

  ```
  !$ACC PARALLEL LOOP COLLAPSE(2) REDUCTION(+: )
  do igp = 1, ngpown
    do ig = 1, ncouls
      do n1 = 1, nbands  #unrolled too!
        do iw = 1, nw    #unrolled
          compute and reduction
  ```

<table>
<thead>
<tr>
<th></th>
<th>TFLOPs</th>
<th>Time</th>
<th>TFLOP/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1.collapse3</td>
<td>3.71</td>
<td>1.63</td>
<td>2.27</td>
</tr>
<tr>
<td>v2.collapse2</td>
<td>3.71</td>
<td>1.73</td>
<td>2.15</td>
</tr>
</tbody>
</table>
V2. More Compute Per Thread

- L2/HBM AI increases!
- Register count at 186
  - Very low occupancy
  - 8 warps per SM
- Need more warps to hide lat
V3. Increase Threadblock Size

- Force threadblock size to be 512, instead of the default 128.
  !$ACC PARALLEL LOOP COLLAPSE(2) VECTOR_LENGTH(512) REDUCTION(+: )

- Register spills but performance may not be bad!

  0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
  ptxas info : Used 186 registers, 624 bytes cmem[0], 32 bytes cmem[2]

  104 bytes stack frame, 188 bytes spill stores, 168 bytes spill loads
  ptxas info : Used 128 registers, 624 bytes cmem[0], 32 bytes cmem[2]
V3. Increase Threadblock Size

- More bandwidth bound now but latency hiding is successful!

<table>
<thead>
<tr>
<th></th>
<th>TFLOPs</th>
<th>Time</th>
<th>TFLOP/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>v2.collapse2</td>
<td>3.71</td>
<td>1.73</td>
<td>2.15</td>
</tr>
<tr>
<td>v3.vector512</td>
<td>3.71</td>
<td>1.40</td>
<td>2.65</td>
</tr>
</tbody>
</table>
V4. Reduce Branching

- Bring iw loop outside of the kernel

```c
do iw = 1, nw       #reduce branching
!$ACC PARALLEL LOOP COLLAPSE(2) VECTOR_LENGTH(512) REDUCTION(+: )
do igp = 1, ngpown
do ig = 1, ncouls
  do n1 = 1, nbands   #unrolled
    compute and reduction
```

- Fewer variables to be reduced -> lower register pressure

  0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads

  ptxas info : Used 122 registers, 600 bytes cmem[0], 32 bytes cmem[2]
V4. Reduce Branching

- Aggregated data for all kernels
- BRA instruction count
  14,278,897,053
  5,975,051,812  x  2
  16%

<table>
<thead>
<tr>
<th></th>
<th>TFLOPs</th>
<th>Time</th>
<th>TFLOP/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>v3.vector512</td>
<td>3.71</td>
<td>1.40</td>
<td>2.65</td>
</tr>
<tr>
<td>v4.iwoutside</td>
<td>3.52</td>
<td>1.17</td>
<td>3.00</td>
</tr>
</tbody>
</table>
### V5. Swap Indices

<table>
<thead>
<tr>
<th>TFLOPs</th>
<th>Time</th>
<th>TFLOP/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>v4.iwoutside</td>
<td>3.52</td>
<td>1.17</td>
</tr>
<tr>
<td>v5.swapindices</td>
<td>3.52</td>
<td>1.16</td>
</tr>
</tbody>
</table>

```fortran
!$ACC PARALLEL LOOP
do iw = 1, nw
    !$ACC PARALLEL LOOP
    do igp = 1, ngpown
        do ig = 1, ncouls
            do n1 = 1, nbands
                wx_array(iw,n1) to (n1,iw)
        end do
    end do
end do
end do
```
V6. Simplify Code

- Fewer instructions -> less work
  - Pull repeated instructions outside the loop
  - Use temporary variables to hold intermediate values for reuse
- Less branches -> better programming
  - 3 branches is more than 1 branch worse than 2 branches!

<table>
<thead>
<tr>
<th></th>
<th>TFLOPs</th>
<th>Time</th>
<th>TFLOP/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>v5.swapindices</td>
<td>3.52</td>
<td>1.16</td>
<td>3.03</td>
</tr>
<tr>
<td>v6.simplify</td>
<td>3.30</td>
<td>1.10</td>
<td>3.00</td>
</tr>
</tbody>
</table>
V7. Replace Divides

- Replace (complex) div. with (double) rcp. and (complex) mul.
- Lower instruction count: 40%
- More bandwidth bound now!

<table>
<thead>
<tr>
<th></th>
<th>TFLOPs</th>
<th>Time</th>
<th>TFLOP/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>v6.simplify</td>
<td>3.30</td>
<td>1.10</td>
<td>3.00</td>
</tr>
<tr>
<td>v7.divs</td>
<td>2.09</td>
<td>0.66</td>
<td>3.18</td>
</tr>
</tbody>
</table>
V7. Replace Divides

- Can be confirmed by Nsight Compute profiles
V8. Replace \( \text{abs}(x) \) with \( x^{**2} \)

\[
\sqrt{x^2 + y^2} < z \quad \Rightarrow \quad x^2 + y^2 < z^2
\]

- `sqrt(complex)` vs power of 2
- Causing pipeline to wait

<table>
<thead>
<tr>
<th></th>
<th>TFLOPs</th>
<th>Time</th>
<th>TFLOP/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>v7.divs</td>
<td>2.09</td>
<td>0.66</td>
<td>3.18</td>
</tr>
<tr>
<td>v8.abs</td>
<td>1.99</td>
<td>0.62</td>
<td>3.23</td>
</tr>
</tbody>
</table>
V8. Replace \( \text{abs}(x) \) with \( x^{**2} \)

Before:

- **Wait:** warp stalled waiting on a fixed latency execution dependency
V8. Replace abs(x) with x**2

After:
- Wait: 46.6% -> 23.7%
V9. Cache Blocking

- Non-coalesced memory access for aqsntemp
- Causing Long Scoreboard Warp State
  - Warp stalled waiting for L1TEX (local, global, surface, tex) memory operation
V9. Cache Blocking

- Break loops into chunks and reuse data across threadblocks
- Increase L2 hit rate

```fortran
!$ACC LOOP GANG VECTOR
do ig_blk = 1, ig_blksize
    !$ACC LOOP SEQ
        do ig = ig_blk, ncouls, ig_blksize
```

<table>
<thead>
<tr>
<th></th>
<th>TFLOPs</th>
<th>Time</th>
<th>TFLOP/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>v8.abs</td>
<td>1.99</td>
<td>0.62</td>
<td>3.23</td>
</tr>
<tr>
<td>v9.block</td>
<td>2.00</td>
<td>0.57</td>
<td>3.50</td>
</tr>
</tbody>
</table>

TFLOPs, Time, TFLOP/s
V9. Cache Blocking

- Less Long Scoreboard samples and higher L2/L1 hit rate
8 Steps to Optimize Sigma-GPP

1. Collapse n’, G’, and G loops
2. Bring n’ loop in; collapse only G’ and G
3. Adjust threadblock size
4. Reduce branching; pull iw loop outside
5. Swap indices to suite parallelisation
6. Simplify code
7. Replace div. with rcp. and mul.
8. Replace abs with power of 2
9. Cache blocking

```fortran
!$ACC PARALLEL LOOP REDUCTION(+:
  do n1 = 1, nbands
    do igp = 1, ngpown
      do ig = 1, ncouls
        do iw = 1, nw
          compute and reduction
        end do
      end do
    end do
  end do
end do
```

<table>
<thead>
<tr>
<th></th>
<th>TFLOPs</th>
<th>Time</th>
<th>TFLOP/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1.collapse3</td>
<td>3.71</td>
<td>1.63</td>
<td>2.27</td>
</tr>
<tr>
<td>v9.block</td>
<td>2.00</td>
<td>0.57</td>
<td>3.50</td>
</tr>
</tbody>
</table>

3x !!
Conclusion

- Code is still bandwidth and latency bound
  - shared memory
  - lower register count
  - improve FMA ratio

- Together with profilers, Roofline provides the complete solution for your performance analysis and optimization needs!
Conclusions

Roofline is a great way to Frame Conversation with Application Teams

Helps Motivate, Direct and Visualize the optimization process.

Coming soon to a tool near you!!
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.**
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.
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

Run Example at “Half Clock” Speed

Is Performance affected by Half-Clock Speed?

Yes

You are at least Partially CPU Bound

No

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

Make Algorithm Changes

Likely Partially Memory Latency Bound (assuming not IO or Communication Bound)
So, you are Memory Bandwidth Bound?

What to do?

1. Try to improve memory locality, cache reuse

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

Profit by getting ~ 4-5x 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.

![Graph showing elapsed time vs. simultaneously utilized logical CPUs]

1. 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
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 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 Knights-Landing, each core will support up to 4 threads. Use them all.
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:

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

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

If your performance is no better, you are memory bound.

Quantum ESPRESSO Packed Vs. Unpacked Performance

- Packed
- Unpacked

Walltime (s)

0 100 200 300 400 500 600 700 800
Cori KNL System

Cray XC40 system with 9,600+ Intel Knights Landing (KNL) nodes:

- 68 cores, 272 Hardware Threads
- Up to 32 FLOPs per Cycle, 1.2-1.4 GHz Clock Rate
- Wide (512 Bit) vector Units
- Multiple Memory Tiers: 96 GB DRAM / 16 GB HBM
- NVRAM Burst Buffer 1.5 PB, 1.5 TB/sec