NERSC8 CoE
Key Actions when optimizing for KNL

Nathan Wichmann
wichmann@cray.com
Outline

- **Characterization and Multi-node Considerations**
  - Target Science
  - Profiles and Hotspots
  - Scaling and Communication

- **Single node optimizations**
  - Memory and cache footprint analysis
  - Memory bandwidth requirements
  - Vectorization
  - Creating a kernel to aid in further analysis and testing

- **Example:** BerkeleyGW - FF kernel
What Science do you want to run on Cori

● Identify 1 or a few science problems that you anticipate running on Cori
  ● Identifying the science problems will help focus efforts on what routines and issues are important

● Estimate how many nodes you will use during the run
  ● Does the code already scale this high?
  ● What can we say about communication

● The combination of science problem and number of nodes will allow one to estimate memory footprints, array sizes, and trip count sizes
  ● This information is critical
Scaling and communication

● How high does the code scale

● Does your code use both OpenMP and MPI?
  ● How many OpenMP threads can you utilize

● What is limiting your scaling?
  ● Communication overhead?
  ● Lack of parallelism on a given science problem

● Understand and optimizing scaling is critical
  ● KNL requires scaling to higher numbers of cores to achieve the same level of performance
  ● Scaling impacts loop trip counts, memory footprints, and more
Where is the time being spent

- Are you sure? Verify
- Use statistical profilers to determine where the time is being spent
  - Are there obvious key routines that time up a significant percentage of time?
  - Are there key loops or code sections?
  - How many routines before you hit 80% of the run time
- Is the profile different for different science problems?
Understanding your memory footprint is critical

- Do you expect to your problem to consume a significant amount of main memory?
  - Main memory is about 96 Gbytes

- Is it possible that your problem will fit into fast memory
  - Fast memory is 16 Gbytes per node
    - Can be configured as a “memory cache”
    - Can be configured 50% cache and 50% explicitly managed
    - Can be configured 25% cache and 75% explicitly managed
    - Can be configured at 100% explicitly managed

- What is the memory access pattern for the routines and loops identified as important
  - What are the trip counts in that loop nest?
  - How much data is accessed?
  - How much is reused more than once?
Vectorization

● Do the loops vectorize?

● Vectorization is very important to achieving high performance rates
  ● Edison vectors are 4 DP words, Cori is longer
  ● Cannot take full advantage of functional units without vectorization
  ● Unlikely to take full advantage of memory bandwidth
  ● Scalar performance on Cori

● Common inhibitors
  ● Dependencies
  ● Indirect addressing may prevent vectorization or make is less efficient
    ● e.g. \( A(\text{indx}(i)) = \)
  ● Function / subroutine calls
  ● If tests inside of inner loops may slow execution and prevent vectorization
  ● More…
Are your kernels memory bandwidth bound

● Do you expect to your problem to consume a significant amount of main memory?
  ● Main memory is about 96 Gbytes

● Is it possible that your problem will fit into fast memory
  ● Fast memory is 16 Gbytes per node
    ● Can be configured as a “memory cache”
    ● Can be configured 50% cache and 50% explicitly managed
    ● Can be configured at 100% explicitly managed

● What is the memory access pattern for the routines and loops identified as important
  ● What are the trip counts in that loop nest?
  ● How much data is accessed?
  ● How much is reused more than once?
How can you tell if you are memory bandwidth bound?

- **Sometimes it is easy**
  - One or more loop nests are streaming through a huge amount of data
  - Little to no reuse
  - Easy to determine the

- **Sometimes it is difficult**
  - Some trip counts are large
  - But some data are reused
  - Not obvious what the compiler did
  - Not obvious if the data remains in cache

- **Counters can be difficult to interpret**
  - Difficult to keep track of different levels of cache

- **Try to run kernel using 1 or 2 fewer cores**
  - Adjust the number of OMP threads
  - Use aprun –S option to spread mpi ranks across more sockets
  - If performance per socket does not change, kernel may be bandwidth bound

- **Try and examine trip counts and reference patterns**
Create kernel that are representative of critical loops

- Use all of the information previously discussed to create kernels to be used for further investigation

- Trip counts and array sizes per node should be as accurate as possible
  - Goal is to reflect what are real science problem running on a significant portion of the machine would look like on a single socket

- Kernel should use all of the cores of a single socket on Edison
  - Kernels that only run on a single core will not capture the full memory footprint and bandwidth characteristics of the real code
Why do we need a kernel?

- **Extreme flexibility and portability**
  - Cannot assume we will always run on a multi-node supercomputer
  - Might not even run it “directly” on a computer

- **Run on many different platforms**
  - Single socket of edison
  - KNC whitebox
  - KNL simulator or emulator
  - Early KNL hardware
  - KNL whitebox

- **Focused analysis**
  - Some tools may not be able to run a full program
  - Want to focus on a particularly important area

- **Flexible experimentation**
  - Try different compilers and options without porting entire code
  - May want to try different “decompositions” and optimizations that would (temporarily) break the larger code
Example Analysis and Optimizations:

BerkeleyGW
BerkeleyGW

- Identified 4-6 kernels
  - GPP
  - FF
  - BSE
  - Chi Summation
  - FFT (library not analyzed by Cray)
  - Scalapack (library not analyzed by Cray)

- Cray analyzed and provided potential optimizations GPP, FF, BSE, and Chi Summation for:
  - Vectorization
  - Memory footprint requirements
  - Memory bandwidth requirements
  - OpenMP effectiveness
  - Cray and Intel compiler

- Next few slides review some of the work done for FF
BerkeleyGW kernels: FF

- **Excellent vectorization and OpenMP**
- **Used craypat to examine where time was being spent**
  
  module unload darshan  # darshan does not seem to play well with craypat
  module load perftools
  ftn -rm -o ffkernel.x ffkernel.f90
  pat_build ffkernel.x
  run
  pat_report ffkernel.x+pat+36422-5701s.xf > ffkernel.manyfreq.patreport

- **Generates both a routine level...**
  
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0%</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>81.9%</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>64.4%</td>
</tr>
<tr>
<td>12.6%</td>
</tr>
<tr>
<td>3.8%</td>
</tr>
</tbody>
</table>

- **... and a line level statistical profile report**
  
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>64.4%</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>25.6%</td>
</tr>
<tr>
<td>12.5%</td>
</tr>
<tr>
<td>25.1%</td>
</tr>
</tbody>
</table>
BerkeleyGW kernels: FF

- Line level statistical profile report
  
  |   | 64.4% | 377.0 | -- | -- | ffkernel_LOOP@li.388  
|---|---|---|---|---|---------------------------|
  4  | 25.6% | 150.0 | 29.6 | 18.8% | line.406  
  4  | 12.5% | 73.0  | 11.2 | 12.7% | line.408  
  4  | 25.1% | 147.0 | 14.0 | 10.1% | line.414

```c
!$OMP PARALLEL do private (my_igp, igp, indigp, igmax, ig, schDtt, I_epsRggp_int, &
!$OMP I_epsAggp_int, schD, schDt, ifreq) reduction(+:schdt_array) !This was line 388 in the source
do ifreq=1,nFreq
  do my_igp = 1, ngpown
    do ig = 1, igmax
      I_epsRggp_int = I_epsR(ig,my_igp,ifreq) !This was line 406 in the source
      I_epsAggp_int = I_epsA(ig,my_igp,ifreq) !This was line 408 in the source
      schD = I_epsRggp_int - I_epsAggp_int
      schDtt = schDtt + matngmat(ig,my_igp)*schD !This was line 414 in the source
    enddo
  enddo
enddo
```

- Don’t focus too much on the time spent in one line vs another…
- The point is that it is clear that a very significant amount of time is being spent in this loop
  nest / region
BerkeleyGW kernels: FF

- Examined trip counts and declarations to calculate memory footprint and reuse
  - nFreq = 20000
  - Ngpown = 20
  - Igmax = 1000

!$OMP PARALLEL do private (...) &
!$OMP (...) reduction(+:schdt_array)
do ifreq=1,nFreq
  do my_igp = 1, ngpown
    do ig = 1, igmax
      I_epsRggp_int = I_epsR(ig,my_igp,ifreq)
      I_epsAggp_int = I_epsA(ig,my_igp,ifreq)
      schD = I_epsRggp_int - I_epsAggp_int
      schDtt = schDtt + matngmat(ig,my_igp)*schD
    enddo
  enddo
enddo

- Lots and lots of parallelism
- I_epsR and I_epsA were each about 1.6 Gbytes with no immediate reuse
- matngmat about 80 kbytes, and shared across threads
BerkeleyGW kernels: FF

- Let’s examine OpenMP scaling for a moment

- Virtually no improvement in performance after 8 threads
- Yet we know there is lots and lots of parallelism
BerkeleyGW kernels: FF

- Streaming data arrays that are more than 3 Gbytes in size
- Lots of parallelism, but performance stops improving

- Conclusion: Loop was memory bandwidth bound

- On Cori I_epsR might fit into fast memory
  - But then we would still just be limited by the bandwidth of fast memory
- Only way to go faster is to find more data reuse

```plaintext
do ifreq=1,nFreq
  do my_igp = 1, ngpown
    do ig = 1, igmax
      I_epsRggp_int = I_epsR(ig,my_igp,ifreq)
      I_epsAggp_int = I_epsA(ig,my_igp,ifreq)
      schD=I_epsRggp_int-I_epsAggp_int
      schDtt = schDtt + matngmat(ig,my_igp)*schD
    enddo
  enddo
enddo
```
BerkeleyGW kernels: FF

- Realized there was a “nbands” loop at a relatively high level that reused the I_eps* variables
  - Worked to effectively cache block main loops

```fortran
do ifreq=1,nFreq
  do igbeg = 1,igmax,igblk
    igend = min(igbeg+igblk-1,igmax)
    do my_igp_beg = 1, ngpown,cblk
      my_igp_end = min(my_igp_beg+cblk-1,ngpown)
      do n1_beg=1,number_bands,cblk
        n1_end = min(n1_beg+cblk-1,number_bands)
        do my_igp = my_igp_beg,my_igp_end
          do n1=n1_beg,n1_end
            ... do ig = igbeg, igend
              I_epsRggp_int = I_epsR(ig,my_igp,ifreq)
              I_epsAggp_int = I_epsA(ig,my_igp,ifreq)
              schD=I_epsRggp_int-I_epsAggp_int
              schDtt=schDtt+aqsnntemp(ig,n1) *CONJG(aqsmntemp(igp,n1)) *schD
              schdt_matrix(ifreq,n1) = schdt_matrix(ifreq,n1) + schDtt
            enddo
          enddo
        enddo
      enddo
    enddo
  enddo
enddo
```

- Resulted in a 4X improvement in wall-clock time on XEON

I_eps arrays do not change with n1
BerkeleyGW kernels: FF

- Realized there was a “nbands” loop at a relatively high level that reused the I_eps* variables
  - Worked to effectively cache block main loops
  ```
  do ifreq=1,nFreq
    do igbeg = 1,igmax,igblk
      igend = min(igbeg+igblk-1,igmax)
      do my_igp_beg = 1, ngpown,cblk
        my_igp_end = min(my_igp_beg+cblk-1,ngpown)
        do n1_beg=1,number_bands,cblk
          n1_end = min(n1_beg+cblk-1,number_bands)
          do my_igp = my_igp_beg,my_igp_end
            do n1=n1_beg,n1_end
              I_epsRggp_int = I_epsR(ig,my_igp,ifreq)
              I_epsAggp_int = I_epsA(ig,my_igp,ifreq)
              schD=I_epsRggp_int-I_epsAggp_int
              schDtt=schDtt+aqsntemp(ig,n1) *CONJG(aqmtemp(igp,n1)) *schD
              schdt_matrix(ifreq,n1) = schdt_matrix(ifreq,n1) + schDtt
            enddo
          enddo
        enddo
      enddo
    enddo
  enddo
  ```
- Resulted in a 4X improvement in wall-clock time on XEON

aqsntemp array does not change with my_igp
Data Reuse will be important

- Data reuse will be critical to performance
- Reuse out of HBM will reduce requirements on main memory
- Reuse out of lower levels of cache will lower requirements on HBM
- In order to know how to cache block properly we need to know the trip counts of loops and the sizes of various arrays as accurately as possible
Summary

- Code Characterization will be an important first step in preparing for Cori
  - Target Science
  - Target Scaling
  - Hotspot identification

- Cori node is different from Edison node
  - Single node optimizations will be an early focus
  - A properly designed kernel will help with optimization efforts
  - Vectorization will be more important in the future

- Data reuse will be important, but how important will depend on memory footprints and access patterns