Optimizing Code for Intel Xeon Phi 7250 (Knight’s Landing)
## Multicore vs. manycore

<table>
<thead>
<tr>
<th>Multicore (Edison)</th>
<th>Manycore (Cori-KNL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000 nodes</td>
<td>9600 nodes</td>
</tr>
<tr>
<td>12 physical cores/CPU</td>
<td>68 physical cores/CPU</td>
</tr>
<tr>
<td>24 HW threads/CPU</td>
<td>272 HW threads/CPU</td>
</tr>
<tr>
<td>2.4-3.2 GHz</td>
<td>1.2-1.6 GHz</td>
</tr>
<tr>
<td>4 DP ops/cycle</td>
<td>2x8 DP ops/cycle</td>
</tr>
<tr>
<td>30 MB L3 cache</td>
<td>no L3 cache</td>
</tr>
<tr>
<td>64 GB/node</td>
<td>16 GB/node (fast)</td>
</tr>
<tr>
<td></td>
<td>96 GB/node (slow)</td>
</tr>
<tr>
<td></td>
<td>100 GB/s memory</td>
</tr>
<tr>
<td></td>
<td>bandwidth</td>
</tr>
<tr>
<td></td>
<td>450 GB/s memory</td>
</tr>
<tr>
<td></td>
<td>bandwidth (fast)</td>
</tr>
</tbody>
</table>
Recompile and go?

- **x86-64 compatible**: can use codes for older architectures or recompile
- **self-hosted**: no need for offloading

- median speedup vs. Edison: 1.15x
- median speedup vs. Haswell: 0.70x
Why should I optimize my code?

• pros
  • **get more for your bucks:** making efficient use of existing manycore HPC systems
  • **fast success possible:** many low hanging fruits in unoptimized codes
  • **investing in the future:** heterogeneous architectures are energy efficient and thus will stay around for a while
  • **benefits on multicore:** optimizations targeting manycore architectures mostly improve performance on multicore systems as well

• cons
  • **effort:** many most beneficial optimizations require significant code changes
  • **investing in the future:** what if I bet on the wrong horse?
Optimization targets

• single node performance
  • start here: for representative local problem size, single node performance is upper bound of what you get in multi-node
  • many optimization opportunities, fast turnaround times
  • many profiling tools available

• multi-node performance
  • fewer optimization opportunities, profiling/debugging tedious

• IO performance
  • not many opportunities for improvement
Where do I start?

- **get to know your application**: don’t assume you already do!
- **determine hotspots**
  - **manual timers**: be careful with thread safety/sync barriers
  - **profiling tools**: NERSC offers a lot
    - [CrayPat](#) (very lightweight)
    - [Advisor](#) (find time-consuming loops)
    - [VTune](#) (can do a lot of things but also very slow)
    - [MAP](#) (comparably lightweight)
- **found hotspots, now what?**
What architectural feature shall I target?

- KNL has many new features to explore
  - many threads
  - bigger vector units
  - complex intrinsics (ISA)
  - multiple memory tiers
- understand your hotspots
  - compute bound: more threads, vectorization, ISA
  - memory BW bound: memory tiers, more threads
  - memory latency bound: more threads, vectorization
Prerequisites - compile and run

• recompile your code for KNL: code for older CPUs is supported but those do not make full use of new architecture
  
  • Cray (wrappers):
    module swap craype-haswell craype-mic-knl
  
  • Intel: -xmic-avx512
  
  • GNU: -march=knl
  
• use proper OpenMP settings:
  export OMP_NUM_THREADS=64
  export OMP_PLACES=threads
  export OMP_PROC_BIND=spread

• use job-script-generator on my.nersc.gov or NERSC website

• node configuration: use -C knl,quad,cache as a start
# Prerequisites - #FLOPS

• **#FLOPS**: number of floating point operations
  - manual calculation:
    - float addition and multiplication: +1
    - complex multiplication: +6 (4 multiplications + 2 additions)
    - etc.
  - measure with SDE:
    - using SDE is more precise, because it accounts for masking

```c
__SSC_MARK(0x111); // start SDE tracing, note it uses 2 underscores

for (k=0; k<NTIMES; k++) {
    <my super expensive loop body>
}

__SSC_MARK(0x222); // stop SDE tracing

srun -n ... sde64 -knl -d -iform 1
       -omix my_mix.out -i global_region
       -start_ssc_mark 111:repeat -stop_ssc_mark 222:repeat
       -- my_super_slow_app.exe
```
Prerequisites - #BYTES

- **#BYTES**: number of bytes transferred from main memory
  - manual calculation (not recommended, but good check):
    - count the bytes of data to be read and written in the kernel
    - does not account for data reuse through caching
  - measure with VTune:
    - precisely obtain uncore counter events
What is limiting my performance?

- **Roofline performance model**
- arithmetic intensity
  \[ AI = \frac{\#\text{FLOPS}}{\#\text{BYTES}} \]
- performance
  \[ P = \frac{\#\text{FLOPS}}{\text{time}[s]} \]
- plot P vs. AI with architectural roofline R
  \[ R(AI) = \min(\text{memory\_bw} \cdot AI, \text{peak\_flops}) \]
Example roofline

![Graph showing roofline analysis with different performance measures for Peak HBM, Peak DDR, ILP, and Vectorization.]
Example roofline

Memory bandwidth bound

GFLOP/s

Arithmetic Intensity

Peak HBM
Peak DDR
- ILP
- Vectorization

memory bandwidth bound
Example roofline

Use MCDRAM
Example roofline

- Use MCDRAM when computing at the roofline bound.
Example roofline

- Use MCDRAM
- Utilize vectorization
Example roofline

- Use MCDRAM
- (possibly) memory latency bound
- Utilize vectorization

Graph showing GFLOP/s vs. Arithmetic Intensity with lines for Peak HBM, Peak DDR, ILP, and Vectorization.
Example roofline

- Use MCDRAM
- Utilize vectorization
- Threads, vectorization
- Use MCDRAM

Graph showing performance metrics with lines indicating Peak HBM, Peak DDR, ILP, and Vectorization.
Example roofline

- Utilize vectorization
- Threading, vectorization
- Use MCDRAM
- Improve AI

GFLOP/s vs. Arithmetic Intensity
How to improve AI?

• definition of arithmetic intensity

\[ AI = \frac{\#FLOPS}{\#BYTES} \]

• two possibilities
  • number of flops ↑ number of bytes ➞
    
    (not possible/easy, choice of algorithm determines flops)
  • number of flops ➞ number of bytes ↓

• reality: tradeoff between both
Create more work/thread

- **loop/kernel fusion**: improves cache re-use and reduce overhead

```
#pragma omp parallel for
for(unsigned int i=0; i<N; ++i){
    r[i] = 0.0;
    for(unsigned int j=0; j<N; ++j){
        r[i] += A[j+i*N]*x[j];
    }
}
```

- **collapse nested loops**:

```
#pragma omp parallel for
collapse(3)
for(unsigned int z=0; z<Nz; z++){
    for(unsigned int y=0; y<Ny; y++){
        for(unsigned int x=0; x<Nx; x++){
            ...
        }
    }
}
```

- **rearrange data structures**: move **OpenMP out** (coarse grain)
Loop transformations I

- **loop tiling**: improves cache re-use and can significantly improve performance

```c
1  !$omp parallel do collapse(2) firstprivate(n_jr, n_ir)
2  !$omp private(jr, ir)
3  DO jr = 1, n_jr
4     DO ir = 1, n_ir
5     c(ir,jr) = a(ir,jr) * b(ir)
6     ENDDO
7  ENDDO
8  !$omp end parallel do
```
Loop transformations I

- **loop tiling:** improves cache re-use and can significantly improve performance

```fortran
12 nblock=2048
13 n_irt = n_ir / nblock
14 if (mod(n_ir, nblock).ne.0) n_irt = n_irt + 1
15 !$omp parallel do collapse(2) firstprivate(n_jr, n_ir, n_irt, nblock)
16 !$omp private (ir_start, ir_end, jr, ir)
17 do ir = 1, n_irt
18    do jr = 1, n_jr
19       ir_start = (ir-1)*nblock + 1
20       ir_end = min(ir_start+nblock-1, n_ir)
21    do ir = ir_start, ir_end
22       c(ir, jr) = a(ir, jr) * b(ir)
23   enddo
24 enddo
25 !$omp end parallel do
```

- especially relevant on KNL because of missing L3
- blocking to **shared L2 (512KiB)** usually good
- was my transformation successful? check L1, L2 miss rates, e.g. in VTune
• **short loop unrolling**: helps the compiler vectorizing the **right** loops

```c
#pragma omp parallel for collapse(3)
for(unsigned int z=0; z<Nz; z++){
  for(unsigned int y=0; y<Ny; y++){  
    for(unsigned int x=0; x<Nx; x++){
      vnorm(x,y,z) = 0.;
      for(unsigned int d=0; d<3; d++){  
        vnorm(x,y,z) += a(x,y,z)[d] * a(x,y,z)[d];
      }
    }
  }
}
```
Loop transformations II

• short loop unrolling: helps the compiler vectorizing the right loops

```c
#pragma omp parallel for simd collapse(3)
for(unsigned int z=0; z<Nz; z++){
    for(unsigned int y=0; y<Ny; y++){
        for(unsigned int x=0; x<Nx; x++){
            vnorm(x,y,z) = a(x,y,z)[0] * a(x,y,z)[0] + a(x,y,z)[1] * a(x,y,z)[1] + a(x,y,z)[2] * a(x,y,z)[2];
        }
    }
}
```

• unrolling pragmas are helpful too
• check compiler optimization reports
• use Intel Advisor
Data alignment

- align (and pad) data to 64bit words to improve prefetching
- can be done easily in major programming languages
  - FORTRAN: -align array64byte
    (ifort, gfortran does it automagically)
  - C/C++: aligned_alloc(64, <size>),
    __attribute__((aligned(64))), __declspec(align(64))
  - C++ trick: overload new operator
- advanced: manually pad data if array extents are power of 2 to minimize cache associativity conflicts
Make use of ISA

- help the compiler to generate efficient intrinsics

```c
for (int n = 0; n < nc; n++) {
    for (int k = lo[2]; k <= hi[2]; k++) {
        for (int j = lo[1]; j <= hi[1]; j++) {

            if (((lo[0] + j + k + rb) % 2) != 0) continue;

            Real cf0 = ((i == blo[0]) && (m0(IntVect(blo[0] - 1, j, k)) == 0) ? f0(IntVect(i, blo[0], j, k)) : 0.0);
            Real cf1 = ((j == blo[1]) && (m1(IntVect(i, blo[1] - 1, k)) == 0) ? f1(IntVect(i, blo[1], j, k)) : 0.0);
            Real cf2 = ((k == blo[2]) && (m2(IntVect(i, j, blo[2] - 1)) == 0) ? f2(IntVect(i, j, blo[2])) : 0.0);
            Real cf3 = ((j == bhil[0]) && (m3(IntVect(bhil[0] + 1, j, k)) == 0) ? f3(IntVect(bhil[0], j, k)) : 0.0);
            Real cf4 = ((j == bhil[1]) && (m4(IntVect(bhil[1] + 1, k)) == 0) ? f4(IntVect(bhil[1], k)) : 0.0);
            Real cf5 = ((k == bhil[2]) && (m5(IntVect(i, jbhil[2] + 1))) == 0) ? f5(IntVect(i, jbhil[2])) : 0.0);

            // assign ORA constants
            double gamma = alpha * a(IntVect(i, j, k))

            + bx(2) * (bx(IntVect(i, j, k))) * bx(2 * INT_VECT(i + 1, j, k))
            + dx(4) * (dx(4) * b(IntVect(i, j, k)) * bx(4) + bx(4) * b(IntVect(i + 1, j, k)) * bx(4))

            + dx(2) * (bx(IntVect(i, j, k)) * cf0 + bx(IntVect(i + 1, j, k)) * cf3)
            + dx(4) * (bx(IntVect(i, j, k)) * cf1 + bx(IntVect(i + 1, j, k)) * cf4)
            + dx(2) * (bx(IntVect(i, j, k)) * cf2 + bx(IntVect(i + 1, j, k)) * cf5)

            double rho = -dx(2) * (bx(IntVect(i, j, k)) * phi(IntVect(i + 1, j, k), n) + bx(IntVect(i + 1, j, k)) * phi(IntVect(i + 1, j, k), n))
            + dx(4) * (bx(IntVect(i, j, k)) * phi(IntVect(i, j - 1, k), n) + bx(IntVect(i, j - 1, k), n) * phi(IntVect(i, j - 1, k), n))
            + dx(2) * (bx(IntVect(i, j, k)) * phi(IntVect(i, j, k - 1), n) + bx(IntVect(i, j, k - 1), n) * phi(IntVect(i, j, k - 1), n))

            double res = rhs(IntVect(i, j, k), n) - gamma + phi(IntVect(i, j, k), n) + rho;
            phi(IntVect(i, j, k), n) = omega * g_m_d * res;
        }
    }
}
```

runtime example for app with kernel: 1.2 sec
Make use of ISA

- help the compiler to generate efficient intrinsics

```c
for (int n = 0; n < nnc; n++) {
    for (int k = lo[2]; k <= hi[2]; ++k) {
        for (int j = lo[1]; j <= hi[1]; ++j) {
            #pragma omp simd
            for (int i = lo[0]; i <= hi[0]; ++i) {
                if (((lo[0] + j + k + rb)%2)==0) continue;

                // BC terms
                Real cf0 = (i == blo[0]) || (m0(IntVect(blo[0]-1,j,k)) < 0) ? f0(IntVect(blo[0],j,k)) : 0.0;
                Real cf1 = (i == blo[1]) || (m1(IntVect(i,blo[1]-1,j,k)) < 0) ? f1(IntVect(i,blo[1],j,k)) : 0.0;
                Real cf2 = (i == blo[2]) || (m2(IntVect(i,blo[2]-1,j,k)) < 0) ? f2(IntVect(i,blo[2],j,k)) : 0.0;
                Real cf3 = (i == bh[0]) || (m3(IntVect(bh[0]+1,j,k)) < 0) ? f3(IntVect(bh[0],j,k)) : 0.0;
                Real cf4 = (j == bh[1]) || (m4(IntVect(i,bh[1]+1,j,k)) < 0) ? f4(IntVect(i,bh[1],j,k)) : 0.0;
                Real cf5 = (k == bh[2]) || (m5(IntVect(i,bh[2]+1,j,k)) < 0) ? f5(IntVect(i,bh[2],j,k)) : 0.0;

                // assign ORA constants
                double gamma = alpha * a(IntVect(i,j,k))
                    + dX * (bX(IntVect(i,j,k)) + bX(IntVect(i+1,j,k)))
                    + dY * (bY(IntVect(i,j,k)) + bY(IntVect(i,j+1,k)))
                    + dZ * (bZ(IntVect(i,j,k)) + bZ(IntVect(i,j,k+1)))
                double g_m_d = gamma
                    - dX * (bX(IntVect(i,j,k)) * cf0 + bX(IntVect(i+1,j,k)) * cf3)
                    - dY * (bY(IntVect(i,j,k)) * cf1 + bY(IntVect(i,j+1,k)) * cf4)
                    - dZ * (bZ(IntVect(i,j,k)) * cf2 + bZ(IntVect(i,j,k+1)) * cf5)
                double rho = -dX * (bX(IntVect(i,j,k)) * phi(IntVect(i-1,j,k),n) + bX(IntVect(i+1,j,k)) * phi(IntVect(i+1,j,k),n))
                    + dY * (bY(IntVect(i,j,k)) * phi(IntVect(i,j+1,k),n) + bY(IntVect(i,j+1,k),n))
                    + dZ * (bZ(IntVect(i,j,k)) * phi(IntVect(i,j,k+1),n) + bZ(IntVect(i,j,k+1),n))
                double res = rhs(IntVect(i,j,k,n)) - gamma * phi(IntVect(i,j,k),n) + rho;
            }
        }
    }
}
```

runtime example for app with kernel: 1.2 sec
• help the compiler to generate efficient intrinsics

```c
for (int n = 0; n < nc; n++)
    for (int i = lo[0]; i <= hi[0]; ++i)
        for (int j = lo[1]; j <= hi[1]; ++j) {
            Int off = (lo[0] + j + k + rb)%2;
            #pragma omp simd
            for (int i = lo[0]; i <= hi[0]; i += 2) {
                int cfo = (i >= lo[0]) & (m0(IntVect(blo[0]-1, i, j, k)) < 0) ? f0(IntVect(blo[0], i, j, k)) : 0.0;
                int cfo = (i >= lo[1]) & (m1(IntVect(i, blo[1]-1, j, k)) < 0) ? f1(IntVect(i, blo[1], j, k)) : 0.0;
                int cfo = (i >= lo[2]) & (m2(IntVect(i, j, blo[2]-1)) < 0) ? f2(IntVect(i, j, blo[2])) : 0.0;
                int cfo = (i >= hi[0]) & (m3(IntVect(bhi[0]-1, i, j, k)) < 0) ? f3(IntVect(bhi[0], i, j, k)) : 0.0;
                int cfo = (i >= hi[1]) & (m4(IntVect(i, bhi[1]-1, j, k)) < 0) ? f4(IntVect(i, bhi[1], j, k)) : 0.0;
                int cfo = (i >= hi[2]) & (m5(IntVect(i, j, bhi[2]+1)) < 0) ? f5(IntVect(i, j, bhi[2])) : 0.0;

                double gamma = alpha * a(IntVect(i, j, k));
                + dhx * (bX(IntVect(i, j, k)) + bX(IntVect(i+1, j, k)));
                + dhy * (bY(IntVect(i, j, k)) + bY(IntVect(i, j+1, k)));
                + dhz * (bZ(IntVect(i, j, k)) + bZ(IntVect(i, j, k+1)));

                double g_m_d = gamma;
                - dhx * (bX(IntVect(i, j, k)) + bX(IntVect(i+1, j, k)) * cfo);
                - dhy * (bY(IntVect(i, j, k)) + bY(IntVect(i, j+1, k)) * cfo);
                - dhz * (bZ(IntVect(i, j, k)) + bZ(IntVect(i, j, k+1)) * cfo);

                double rho = dhx * (bX(IntVect(i, j, k)) * phi(IntVect(i-1, j, k, n)) + bX(IntVect(i+1, j, k)) * phi(IntVect(i+1, j, k, n)));
                + dhy * (bY(IntVect(i, j, k)) * phi(IntVect(i, j-1, k, n)) + bY(IntVect(i, j+1, k)) * phi(IntVect(i, j+1, k, n)));
                + dhz * (bZ(IntVect(i, j, k)) * phi(IntVect(i, j, k-1, n)) + bZ(IntVect(i, j, k+1)) * phi(IntVect(i, j, k+1, n)));

                double res = rhs(IntVect(i, j, k, n)) - gamma * phi(IntVect(i, j, k, n)) + rho;
                phi(IntVect(i, j, k, n)) += omega/g_m_d * res;
            }
        }
```
Reduced precision math

- transcendental functions, square roots, etc. are expensive
- use `-fp-model fast=2 -no-prec-div` during compilation
- replace divisions by constants with multiplications with inverse

```c
double norm=1.234;
#pragma omp parallel for firstprivate(norm)
for(unsigned int i=0; i<N; i++){
    a[i]/=norm;;
}
```

```c
double invnorm=1./1.234;
#pragma omp parallel for firstprivate(invnorm)
for(unsigned int i=0; i<N; i++){
    a[i]*=invnorm;
}
```

- do not expect too much: benefits usually only visible in heavily compute-bound code sections
- reduced precision might not always be acceptable
Benefits of AVX-512

- median speedup: 1.2x
- benefits can be larger than 2x (probably more efficient prefetching)
- automatically enabled when compiling for KNL architecture
Use MCDRAM

- always use 16GiB on-package memory (MCDRAM)

- cache works well: request with `-C knl,cache`
- code fits into 16GiB: request `-C knl,flat` and prepend executable with `numactl -m 1`
A note on heap allocation

• KNL memory allocation is comparably slow
• avoid allocating and de-allocating memory frequently
  • remove allocations/deallocations in loop bodies or functions which are called many times
• too involved? **pool allocator libraries** (e.g. Intel TBB scalable memory pools)
  • pros:
    • overloads new/malloc, no/minimal source code changes necessary
    • can give significant performance boost for certain codes
    • take care of thread-safety
  • cons:
    • memory footprint needs to be known/computed in advance
    • code might become less portable
Multi-node optimizations

• single KNL thread cannot saturate Aries injection rate

![Cori Multi-Rank Bandwidth graph]

• use thread-level communication or **multiple MPI ranks per node**

• recommended: **>4 ranks per node**

• **dedicate cores to OS:** `-S <ncores>` in sbatch (`ncores=2` good choice)
Hugepages, DMAPP and hardware AMO

- **hugepages** can reduce Aries TLB misses
- load corresponding module at compile and runtime
  ```
  [[[tkurth@cori04 ~]$ module avail craype-hugepages

  ________________________________________________________________ /opt/cray/pe/craype/2.5.7/modulefiles ________________
  craype-hugepages128M craype-hugepages256M craype-hugepages32M craype-hugepages512M craype-hugepages8M
  craype-hugepages16M craype-hugepages2M craype-hugepages4M craype-hugepages64M
  ```
- can use **different modules at compile and runtime**
- MPI-collective-heavy codes: **enable DMAPP** (add `-ldmapp`)
  ```
  export MPICH_RMA_OVER_DMAPP=1
  export MPICH_USE_DMAPP_COLL=1
  export MPICH_NETWORK_BUFFER_COLL_OPT=1
  ```
- enable hardware AMO for MPI-3 RMA atomics
  ```
  export MPICH_RMA_USE_NETWORK_AMO=1
  ```
Some notes on IO

Single Core I/O Performance on Cori

- **Write Bandwidth (MB/s):**
  - Buffered I/O: 900 MB/s
  - Sync I/O: 300 MB/s
  - Direct I/O: 0 MB/s

- **Relative Performance (KNL/HSW):**
  - HSW: 0.9
  - KNL: 0.3
  - KNL/HSW: 0.0

- **Legend:**
  - Blue: HSW
  - Orange: KNL
  - Grey: KNL/HSW
Use multiple processes

- use more processes (e.g. with MPIIO)
- unfortunately, no good threaded IO solutions available yet
- always: pool (write big chunks), reduce file operations (open, close)
- large files: burst buffer
• median speedup vs. Edison: 1.15x
• median speedup vs. Haswell: 0.70x
Does it help?

- median speedup vs. Edison: 1.8x
- median speedup vs. Haswell: 1.0x
Summary

• single node performance (go for that one first)
  • loop fusion and tiling
  • ensure good vectorization
  • use MCDRAM
• multi-node performance
  • hugepages
  • DMAPP
• IO performance
  • use multiple nodes, pool IO, reduce file operations to minimum
NERSC training material

- running jobs
- process/thread binding
- code profiling and tools
- measuring arithmetic intensity (AI)
- improving OpenMP scaling
- vectorization help
- how to use MCDRAM
- NESAP case studies
Thank you