Helen He (and XGC1 team)

Cray Quarterly Meeting
July 22, 2015
XGC1 NESAP Team Members

• **Name with *:** Attended the Dungeon Session
• **Code Team:**
  – PPPL: C.S. Chang (PI), Seung-Hoe Ku, Jianying Lang*, Stephan Ethier, Robert Hager
  – ORNL: Ed D’Azevedo*, Pat Worley
  – LBNL: Mark Adams
  – RPI: Eisung Yoon*
  – A good mix of physics, performance and library people
• **Cray:** Nathan Wichmann*
• **Intel:** Thanh Phung, Dmitry Nemirov*, Antonio Valles*
• **NERSC Liaison:** Helen He*
XGC1: a PIC Fusion Code

- Particle-in-cell code used to study turbulent transport in magnetic confinement fusion plasmas.
- Uses fixed unstructured grid. Hybrid MPI/OpenMP for both spatial grid and particle data. (plus PGI CUDA Fortran, OpenACC)
- Excellent overall MPI scalability
- Internal profiling timer borrowed from CESM
- Uses PETSc Poisson Solver (separate NESAP effort)
- 60k+ lines of Fortran90 codes.
- For each time step:
  - Deposit charges on grid
  - Solve elliptic equation to obtain electro-magnetic potential
  - Push particles to follow trajectories using forces computed from background potential (~50-70% of time)
  - Account for collision and boundary effects on velocity grid
- Most time spent in Particle Push and Charge Deposition

Unstructured triangular mesh grid due to complicated edge geometry

Sample Matrix of communication volume
Programming Portability

- Currently XGC1 runs on many platforms
- Part of NESAP and ORNL CAAR programs
- Applied for ANL Theta program
- Have `#ifdef _OpenACC` and `#ifdef _OpenMP` in code.
- OpenMP 4.0 target directives
- PGI CUDA Fortran
- As fewer compiler dependent directives as possible.
- Nested OpenMP is used
- Need thread safe PSPLIB and PETSc libraries.
Multi Species Collision Kernel
• Had many iterations of single-species collision kernel.
  – Good OpenMP scaling
  – Hotspot is 33% of cpu time. Multiple loops with ~80% vectorization efficiency
  – Many accesses unaligned. Non sequential access too.

• Multi-species version ready on July 6.

• Nathan optimized with initialization and vectorization: 20150708 version

• Two main goals
  – Vectorization
  – Evaluate for HBM analysis
"-heap-arrays 64" Compiler Flag

• Slows down both the collision and pushe kernels by >6x.
• Puts automatic arrays and arrays created for temporary computations of size (64 kbytes or larger) on the heap instead of the stack.
• Allocation and access of private copies on the heap are very expensive.
• Does no affect explicit-shape arrays.
• Removed this flag for the collision kernel, and set OMP_STACKSIZE to a large value. Now 20150708 version: Intel compiler 43 sec. (improved from 348 sec)
• Alternative: use !$OMP THREADPRIVATE. Downside: data has to be static, not allocatable.
XGC1 Collision Dungeon: Tools Examined

• **OMP imbalance**
  – Using 18 threads on 1 socket on Haswell EX, imbalance is 0.7%
  – Using 60 threads on the node: imbalance is 22%
  – Added KMP_BLOCKTIME=infinite variable to prevent “sleeping” of some threads

• **Vector Advisor**
  – 4 hotspots
  – Used “!DIR$ nounroll” and “!DIR$ loopcount(31)” to help vectorization

• **Vtune Memory Bandwidth analysis**
  – Reach peak bandwidth at times

• **Vtune Memory Access analysis**
  – Original collection shows the array name as unknown due to not dynamically allocated.
  – Modified array declaration, put into FASTMEM, 24% faster with 14 threads on 1 socket on Haswell EP

• **SDE for collecting instruction mixes**
  – 11% less instructions on KNL vs Haswell AVX2
# Vector Advisor after Vectorization of Scalar Loop

<table>
<thead>
<tr>
<th>Loops</th>
<th>Vector Issues</th>
<th>Self Time</th>
<th>Total Time</th>
<th>Loop Type</th>
<th>Why No Vectorization?</th>
<th>Vectorized Loops</th>
<th>Vectorized Efficiency</th>
<th>Estim.</th>
<th>Vectorized Loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>[loop in col_f_angle_avg.m at col_f_core.m,F90:546]</td>
<td>3 Ineffecti...</td>
<td>177.826s</td>
<td>177.826s</td>
<td>Vectorized...</td>
<td>AVX2 ~8%</td>
<td>3.49</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[loop in col_f_angle_avg.m at col_f_core.m,F90:590]</td>
<td>3 Ineffecti...</td>
<td>164.763s</td>
<td>164.763s</td>
<td>Vectorized...</td>
<td>AVX2 ~8%</td>
<td>1.33</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[loop in _kmp_launch_thread at kmp_runtime.c,F90:5600]</td>
<td>3 Ineffecti...</td>
<td>15.395s</td>
<td>597.070s</td>
<td>Vectorized...</td>
<td>AVX2 ~8%</td>
<td>3.25</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[loop in col_f_e_and_d_s at col_f_core.s,F90:833]</td>
<td>2 Ineffecti...</td>
<td>10.889s</td>
<td>10.889s</td>
<td>Vectorized...</td>
<td>AVX2 ~8%</td>
<td>3.31</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[loop in col_f_e_and_d_s at col_f_core.s,F90:866]</td>
<td>3 Ineffecti...</td>
<td>8.810s</td>
<td>8.810s</td>
<td>Vectorized...</td>
<td>AVX2 ~8%</td>
<td>3.1</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[loop in col_f_e_and_d_s at col_f_core.s,F90:811]</td>
<td>2 Ineffecti...</td>
<td>6.424s</td>
<td>26.148s</td>
<td>Scalar</td>
<td>inner loop w...</td>
<td>AVX2 ~8%</td>
<td>2.77</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>[loop in col_f_angle_avg.m at col_f_core.s,F90:586]</td>
<td>3 Ineffecti...</td>
<td>6.416s</td>
<td>13.221s</td>
<td>Scalar</td>
<td>inner loop w...</td>
<td>AVX2 ~8%</td>
<td>2.27</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>[loop in _intel_avx_rep_memset]</td>
<td>2 Ineffecti...</td>
<td>3.549s</td>
<td>3.549s</td>
<td>Scalar</td>
<td>inner loop w...</td>
<td>AVX2 ~56%</td>
<td>4.45</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>[loop in col_f_e_and_d_s at col_f_core.s,F90:669]</td>
<td>2 Ineffecti...</td>
<td>2.190s</td>
<td>56.232s</td>
<td>Scalar</td>
<td>inner loop w...</td>
<td>AVX2 ~8%</td>
<td>2.27</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>[loop in col_f_angle_avg.m at col_f_core.m,F90:522]</td>
<td>2 Ineffecti...</td>
<td>0.7925</td>
<td>508.520s</td>
<td>Scalar</td>
<td>inner loop th...</td>
<td>AVX2 ~8%</td>
<td>2.27</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>[loop in col_f_e_and_d_s at col_f_core.m,F90:503]</td>
<td>2 Ineffecti...</td>
<td>0.700s</td>
<td>18.59s</td>
<td>Scalar</td>
<td>inner loop w...</td>
<td>AVX2 ~8%</td>
<td>2.27</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>[loop in col_f_e_and_d_s at col_f_core.m,F90:659]</td>
<td>2 Ineffecti...</td>
<td>0.220s</td>
<td>56.452s</td>
<td>Scalar</td>
<td>inner loop w...</td>
<td>AVX2 ~8%</td>
<td>2.27</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>[loop in col_f_e_and_d_s at col_f_core.m,F90:910]</td>
<td>2 Ineffecti...</td>
<td>0.190s</td>
<td>0.190s</td>
<td>Scalar</td>
<td>inner loop w...</td>
<td>AVX2 ~8%</td>
<td>2.27</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>[loop in col_f_picard_step at col_f_core.s,F90:1098]</td>
<td>3 Gather/S...</td>
<td>0.140s</td>
<td>0.140s</td>
<td>Vectorized...</td>
<td>AVX2 ~13%</td>
<td>1.03</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[loop in col_f_core.m at col_f_core.m,F90:248]</td>
<td>1 System f...</td>
<td>0.090s</td>
<td>0.200s</td>
<td>Vectorized...</td>
<td>AVX2 ~8%</td>
<td>3.11</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[loop in col_f_core.m at col_f_core.m,F90:245]</td>
<td>1 System f...</td>
<td>0.080s</td>
<td>0.170s</td>
<td>Vectorized...</td>
<td>AVX2 ~8%</td>
<td>3.11</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[loop in bsolver at bsolver.F90:93]</td>
<td>1 Assumed...</td>
<td>0.080s</td>
<td>0.350s</td>
<td>Threaded (O...</td>
<td>vector depe...</td>
<td>AVX2 ~8%</td>
<td>3.11</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>[loop in bsolver at bsolver.F90:96]</td>
<td>1 Assumed...</td>
<td>0.070s</td>
<td>0.070s</td>
<td>Vectorized...</td>
<td>AVX2 ~8%</td>
<td>3.11</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[loop in col_f_core.m at col_f_core.m,F90:259]</td>
<td>2 Ineffecti...</td>
<td>0.070s</td>
<td>0.070s</td>
<td>Vectorized...</td>
<td>AVX2 ~8%</td>
<td>3.11</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[loop in col_f_df at col_f_core.s,F90:716]</td>
<td>2 Ineffecti...</td>
<td>0.060s</td>
<td>0.060s</td>
<td>Vectorized...</td>
<td>AVX2 ~8%</td>
<td>3.11</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[loop in col_f_core.m at col_f_core.m,F90:261]</td>
<td>2 Ineffecti...</td>
<td>0.060s</td>
<td>0.060s</td>
<td>Vectorized...</td>
<td>AVX2 ~8%</td>
<td>3.11</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[loop in col_f_e_and_d_s at col_f_core.s,F90:802]</td>
<td>2 Ineffecti...</td>
<td>0.060s</td>
<td>26.208s</td>
<td>Vectorized...</td>
<td>AVX2 ~8%</td>
<td>3.11</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[loop in col_f_core.m at col_f_core.m,F90:130]</td>
<td>2 Ineffecti...</td>
<td>0.050s</td>
<td>0.050s</td>
<td>Vectorized...</td>
<td>AVX2 ~8%</td>
<td>3.11</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
To simulate using HBM directive following options should be added:

- `!DIR$ ATTRIBUTES FASTMEM :: <data name> should be add immediately after ‘allocatable’ in source code`
- `-ljemalloc -lmemkind -lpthread -lnuma -L/ <PATH_to_jemalloc> -L/<PATH_to_memkind>` to compile options
- set `MEMKIND_HBW_NODES=0`
- Run the application using `numactl --membind=1 --cpunodebind=0 <binary>`
PushE Kernel
XGC1 PushE Kernel

- Lots of iterations to reach barebone:
  - No PETSc or ADIOS, no ChargeE
  - Only initialization and PSPLINE are kept
- Does not vectorize, need major code work
  - Many loop counts smaller than vector length
- Initially only pushes 1 particle at a time, appears good cache hit rate
- Not memory bandwidth bound
- Two main goals:
  - Vectorization
  - Evaluate for HBM analysis
Particle Push Kernel

- Code modified to push groups of particles (group size is sml_veclen in input) to encourage vectorization

- Removed “-heap-arrays” option and adjusted OMP_STACKSIZE

- Code modified to avoid vector notation with modulo(). Bug filed for Intel compiler.
Top Hotspots

• Top hot spots related to particle search and evaluation of bicubic spline interpolation
• Performance limited by data movement
Search and Spline Evaluation

• Search operation: Find which triangle contains the particle to perform interpolation. Current grid has $O(10^5)$ triangles. Future grid has $O(10^6)$ triangles
  – Geometric hashing of particle coordinates to 2D uniform rectangular grid
  – Search short list of triangles that overlap with that grid cell (fewer than 10 triangles)

• Spline evaluation:
  – Geometric hashing to 2D rectangular grid
  – Evaluate bicubic polynomial using coefficients from table in that grid cell
Original Search Routine

- Initial version has short vectors of length 2
- Indirect addressing in “itrig”
- Early exit when the triangle is found

```jlo = lbound( grid%guess_table, 2 )
jhi = ubound( grid%guess_table, 2 )
ij = (xy - grid%guess_min)*grid%inv_guess_d + 1
i = max(i0, min(ihi, ij(1)))
j = max(jlo, min(jhi, ij(2)))
istart = grid%guess_xtable(i,j)
iend = istart + grid%guess_count(i,j) - 1
itr = -1
do k=istart,iend
   itrig = grid%guess_list(k)
dx(1:2) = xy(1:2) - grid%mapping(1:2,3,itrig)
p(1:2) = grid%mapping(1:2,1,itrig)*dx(1) +
   grid%mapping(1:2,2,itrig)*dx(2)
p(3) = 1.0d0 - p(1) - p(2)
   if (minval(p) .ge. -eps) then
      itr = itrig; exit
   endif
endo"
Strip-mine (Loop blocking / Loop tiling)  
Search Routine

<table>
<thead>
<tr>
<th>Source</th>
<th>CPU Time</th>
<th>Ins Re</th>
</tr>
</thead>
<tbody>
<tr>
<td>start = grid%guess_xtable(i,j)</td>
<td>0.233s</td>
<td>0s 1.2s</td>
</tr>
<tr>
<td>end = istart + grid%guess_count(i,j) - 1</td>
<td>0.646s</td>
<td>0s 1.6s</td>
</tr>
<tr>
<td>tr = i - 1</td>
<td>1.331s</td>
<td>0s 3.6s</td>
</tr>
<tr>
<td>se_vector = (iend-istart+1 ge veclen)</td>
<td>0.222s</td>
<td>0s 199s</td>
</tr>
<tr>
<td>f (use_vector) then</td>
<td>0.055s</td>
<td>0s 45s</td>
</tr>
<tr>
<td>do kstart=istart,iend,veclen</td>
<td>0.055s</td>
<td>0s 201s</td>
</tr>
<tr>
<td>kend = min(iend, kstart+veclen-1)</td>
<td>0.038s</td>
<td>0s 111s</td>
</tr>
<tr>
<td>klen = kend - kstart + 1</td>
<td>0.191s</td>
<td>0s 388s</td>
</tr>
<tr>
<td>grid_mapping(1:2,1:3,1:klen) = grid_mapping(1:2,1:3,grid%guess_list(kstart:kend))</td>
<td>2.734s</td>
<td>0s 7.3s</td>
</tr>
<tr>
<td>ir unroll 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>do kk=1,min(klen,veclen)</td>
<td>0.260s</td>
<td>0s 1.2s</td>
</tr>
<tr>
<td>k = (kstart-1) + kk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>! itrig = grid%guess_list(k)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dx_vec1(kk) = xy(1) - grid_mapping(1,3,kk)</td>
<td>0.108s</td>
<td>0s 487s</td>
</tr>
<tr>
<td>dx_vec2(kk) = xy(2) - grid_mapping(2,3,kk)</td>
<td>0.153s</td>
<td>0s 838s</td>
</tr>
<tr>
<td>p_vec1(kk)= grid_mapping(1,1,kk)*dx_vec1(kk) + grid_mapping(1,2,kk)*dx_vec2(kk) &amp;</td>
<td>0.279s</td>
<td>0s 18s</td>
</tr>
<tr>
<td>p_vec2(kk)= grid_mapping(2,1,kk)*dx_vec1(kk) + grid_mapping(2,2,kk)*dx_vec2(kk) &amp;</td>
<td>0.035s</td>
<td>0s 203s</td>
</tr>
<tr>
<td></td>
<td>0.122s</td>
<td>0s 916s</td>
</tr>
<tr>
<td></td>
<td>0.020s</td>
<td>0s 61s</td>
</tr>
</tbody>
</table>
Future Work

• Collision kernel
  – Explore nested OpenMP

• Pusher kernel
  – Explore particle sorting for each grid cell and global data rearrangement before push. Needs major code modification.
  – Design prototype test to examine effectiveness of sorting.
  – Explore replicating triangle data structure to avoid indirect addressing, may use 5X more memory (~200 MB per MPI task)
  – Reorganize OpenMP outer loop over grid cells and pusher particles in cell
Thank you.