Case Study:
Fusion PIC Code XGC1 on Cori KNL

Charlene Yang
Application Performance Specialist
NERSC, LBNL
cjyang@lbl.gov
XGC1: Particle-In-Cell Simulation

PI: CS Chang (PPPL) | ECP: High-Fidelity Whole Device Modeling of Magnetically Confined Fusion Plasma
XGC1: Code Flowchart

Gather Fields from Mesh to Ions

Solve Fields on Mesh

Ion Push

Deposit Charge From Particles to Mesh

Collision Operator

*Computation
*Mapping

Electron Push Sub-Cycling
push electrons without updating fields or collisions – only field gather and push

~50x
XGC1: Code Timings

Unoptimized XGC1 Timings on 1024 Cori KNL nodes in Quad-Flat mode
XGC1: Code Profile on Roofline

Performance (GFLOPS)

Force
Interpolate
Search

Self Elapsed Time: 5.322 s  Total Time: 89.318 s

Arithmetic Intensity (FLOP/Byte)
1. **Search** for nearest 3 mesh nodes to the particle position (for multi-mesh refinement)

2. **Interpolate** fields from 3 mesh points to particle position

3. **Calculate force** on particle from fields

4. **Push** particle for time step $dt$
ToyPush: Baseline Profile - Timings

Unoptimized ToyPush Timings on Cori KNL in Quad-Cache mode
Roofline Model shows information timings alone cannot show:

- Kernel Force Calculation: close to vector peak performance
- Kernel Interpolate and Search: less than scalar peak performance

Data collected with Intel Advisor and analyzed with pyAdvisor
- Single thread rooflines on Cori KNL
- Should focus optimization efforts on Interpolate and Search kernels
## Interpolation: Vectorization – L1 Blocking

### veclength optimizations

- **Baseline:** $2^9$
  - Low L1 Hit Rate, L2 Hit Bound

<table>
<thead>
<tr>
<th>Grouping: Function / Call Stack</th>
<th>Clockticks</th>
<th>Instructions Retired</th>
<th>L1 Hit Rate</th>
<th>L2 Hit Rate</th>
<th>L2 Hit Bound</th>
<th>L2 Miss Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>e_interp_term</td>
<td>105,271,600,000</td>
<td>64,954,400,000</td>
<td>80.8%</td>
<td>94.4%</td>
<td>36.7%</td>
<td>29.5%</td>
</tr>
<tr>
<td>eom_eval</td>
<td>73,858,400,000</td>
<td>65,283,400,000</td>
<td>67.3%</td>
<td>99.9%</td>
<td>100.0%</td>
<td>0.8%</td>
</tr>
<tr>
<td>b_interp_term analytic</td>
<td>60,141,200,000</td>
<td>23,109,800,000</td>
<td>90.3%</td>
<td>100.0%</td>
<td>4.2%</td>
<td>0.0%</td>
</tr>
<tr>
<td>__intel__mic__avx512f__memset</td>
<td>35,288,400,000</td>
<td>3,441,200,000</td>
<td>42.1%</td>
<td>100.0%</td>
<td>0.8%</td>
<td>0.0%</td>
</tr>
<tr>
<td>rk4_push</td>
<td>20,528,200,000</td>
<td>14,898,800,000</td>
<td>31.9%</td>
<td>100.0%</td>
<td>100.0%</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

- **Optimized:** $2^6$
  - High L1 Hit Rate

<table>
<thead>
<tr>
<th>Grouping: Function / Call Stack</th>
<th>Clockticks</th>
<th>Instructions Retired</th>
<th>L1 Hit Rate</th>
<th>L2 Hit Rate</th>
<th>L2 Hit Bound</th>
<th>L2 Miss Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>e_interp_term</td>
<td>97,042,400,000</td>
<td>76,687,800,000</td>
<td>99.4%</td>
<td>100.0%</td>
<td>0.9%</td>
<td>0.0%</td>
</tr>
<tr>
<td>eom_eval</td>
<td>66,556,000,000</td>
<td>67,110,400,000</td>
<td>99.0%</td>
<td>100.0%</td>
<td>3.3%</td>
<td>0.0%</td>
</tr>
<tr>
<td>b_interp_term analytic</td>
<td>16,360,400,000</td>
<td>23,641,800,000</td>
<td>99.3%</td>
<td>100.0%</td>
<td>0.3%</td>
<td>0.0%</td>
</tr>
<tr>
<td>proc_reg_read</td>
<td>14,984,200,000</td>
<td>75,600,000</td>
<td>100.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>rk4_push</td>
<td>14,954,800,000</td>
<td>19,702,200,000</td>
<td>98.5%</td>
<td>100.0%</td>
<td>24.8%</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

~1.5x improvement (MCDRAM Flat); ~2x improvement (DDR Flat)
Problems:

- Field data is stored on grid nodes, particles access nearest 3 grid nodes indirectly via triangle index.
  \[ \text{efield}(j, \text{tri}(i, \text{itri}(iv))) \]

- Interpolation loop is vectorized but inefficiently because of gather loads

Intel Compiler Vectorization Report

LOOP BEGIN at interpolate_aos.F90(67,48)

reference \( \text{itri}(iv) \) has unaligned access
reference \( y(iv,1) \) has unaligned access
reference \( y(iv,3) \) has unaligned access
reference \( \text{evec}(iv,icomp) \) has unaligned access
reference \( \text{evec}(iv,icomp) \) has unaligned access

.....

irregularly indexed load was generated for the variable \(<\text{grid_mapping}_{(1,3,\text{itri}(iv))}>\), 64-bit indexed, part of index is read from memory

.....

LOOP WAS VECTORIZED
unmasked unaligned unit stride loads: 6
unmasked unaligned unit stride stores: 3
unmasked indexed (or gather) loads: 18

.....

18 Gathers per loop iteration
(3 nodes x 3 components x 2)
Interpolation: Vectorization - Memory Access

Optimizations:

• Group particles that access the same triangle together, access grid nodes **directly** with a scalar index

• Single mesh: Trivial

• Multiple mesh: Feasible for number of particles $>>$ number of grid nodes

• Align arrays during compile time.

Intel Compiler Vectorization Report

LOOP BEGIN at interpolate_aos.F90(72,51)
reference $y(iv,1)$ has aligned access
reference $y(iv,3)$ has aligned access
reference $evec(iv, icomp)$ has aligned access

.....

SIMD LOOP WAS VECTORIZED

.....

unmasked aligned unit stride loads: 5
unmasked aligned unit stride stores: 3

~1.6x improvement
Interpolation: Vectorization – memset

Problem:

- Initialization of large arrays with avx512_memset at every time step before entering vector loop becomes memory bandwidth bound.

Optimizations:

- Initialize array inside the vector loop (if you can)
- Use threads for initialization

Intel Compiler Vectorization Report

LOOP BEGIN at interpolate_aos.F90(57,5)

memset generated
loop was not vectorized:
loop was transformed to memset or memcpy

LOOP END

~5% improvement
Higher if no. of particle increases
Interpolation: Optimization Path on Roofline

- Kernel moved to compute bound regime
- AI increased due to memory access pattern change
- Peak compute performance is nearly reached

**Baseline Case (w/ Indirect access)**
- Replace Indirect Access with Scalar Access
- Optimize Vector Length
- Access Grid Data in Scalar Chunks
Search: Vectorization – ‘cycle’ + SIMD

Problems:
• Multiple exits due to ‘cycle’ statement prevents vectorization
• Assumed read after write (RAW) dependency prevent vectorization

Optimization:
• Replace exit condition with a logical mask
• Vectorize with omp simd directive, declare private arrays simd private

Intel Compiler Vectorization Report
LOOP BEGIN at search.F90(62,8)
loop was not vectorized: loop with multiple exits cannot be vectorized unless it meets search loop idiom criteria

LOOP BEGIN at search.F90(66,8)
  reference y(iv,1) has aligned access
  reference y(iv,3) has aligned access
  reference id(iv) has aligned access
  reference continue_search(iv) has aligned access
data layout of a private variable bc_coords was optimized, converted to SoA
  OpenMP SIMD LOOP WAS VECTORIZED
unmasked aligned unit stride loads: 4
unmasked aligned unit stride stores: 1

1.5x improvement
Forcing SIMD vectorization doesn’t work initially due to multiple exits.

Once exits are eliminated, code vectorizes.
Toypush: Optimized Performance

- **Force Kernel**: still good performance, close to vector add peak
- **Interpolate Kernel**: 10x speedup, closer to vector FMA peak
- **Search Kernel**: 3x speedup, closer to L2 bandwidth roof

![Diagram showing marker size as CPU time vs. arithmetic intensity](attachment:image.png)

- Code is available at [https://github.com/tkoskela/toypush](https://github.com/tkoskela/toypush)
XGC1 Timings on 1024 Cori KNL nodes in Quad-Flat mode
XGC1: Optimized Performance on Roofline

Performance (GFLOPS)

Interpolate

Force

Search

Self Elapsed Time: **0.120** s  Total Time: **0.660** s
XGC1: Code Profile on Roofline

Performance (GFLOPS) vs. Arithmetic Intensity (FLOP/Byte)

- Force
- Interpolate
- Search

Self Elapsed Time: 5.322 s  Total Time: 89.318 s
Summary

• XGC1 -> ToyPush -> XGC1

• Roofline Model can help
  – Identify performance bottlenecks (compute, bandwidth, latency, etc)
  – Prioritize optimization efforts (routines, vectorization, memory access, etc)
  – Tell when to stop (realistic achievable performance, distance to roofs)

• Intel Advisor can take care of the rest!
  – Integrated compiler reports, static binary analysis (instruction set, data types, etc) and dynamic analysis (CPU sampling)
  – FLOPS/trip counts, vectorization efficiency, dependency and memory access pattern
  – Roofline charts of various flavors 😊
    • Original DRAM-based Roofline (DRAM <-> Core)
    • Cache-aware Roofline (L1 <-> Core) ✔
    • Cache-simulator based Roofline (L1, L2, LLC, MCDRAM and DRAM <-> Core) ✔