Optimizing a Particle-in-Cell Code on Intel Knights Landing

Minhua Wen*, Min Chen†, James Lin*‡,

*Center for HPC, Shanghai Jiao Tong University(SJTU), Shanghai
†School of Physics and Astronomy, SJTU, Shanghai
‡ Tokyo Institute of Technology, Japan
Outline

• Introduction

• Optimizations

• Results

• Conclusion
Introduction to VLPL-S PIC code

- **Particle-in-cell** (PIC) method is a widely used first-principles model for laser plasmas simulations, with many well-known implementations for different scenarios, such as OSIRIS, EPOCH, VSim, VLPL.

- **Virtual laser plasma laboratory** (VLPL) PIC code is originally developed by A. Pukhov in Germany, VLPL-S is a modified In-house code SJTU Laboratory For Laser Plasmas.

- **VLPL-S** focus on the interaction between high intensity ultrashort laser and plasma, used for both theoretical and experimental purpose
  - laser-driven electron acceleration
  - acceleration of ion beams
  - generation of X-ray and gamma ray radiation
The key algorithm of VLPL-S is as general PIC method

### Flow Chart

1. **Begin**
2. **Initialize**
3. **Field solver**
4. **Force Computation**
5. **Particle Push**
6. **Current Deposition**
7. **Output**
8. **End**

### Equations

Solving the Maxwell’s equations with the FDTD method

- **Field Solver**
  \[
  \nabla \times B = \frac{4\pi}{c} J + \frac{1}{c} \frac{\partial E}{\partial t} \\
  \nabla \times E = -\frac{1}{c} \frac{\partial B}{\partial t}
  \]

- **Current Deposition**
  \[
  j(r) = \sum_j q_j v_j \delta(r - r_j)
  \]

- **Field Interpolation & Force Computation**
  \[
  F_i = q_i \left( E(r_i) + \frac{1}{c} v_i \times B(r_i) \right) \\
  \frac{\partial p_i}{\partial t} = F_i \\
  \frac{\partial r_i}{\partial t} = v_i = \frac{p_i}{m_i} \left( 1 + \left( \frac{p_i}{m_i c} \right)^2 \right)^{-1/2}
  \]

### Workloads

<table>
<thead>
<tr>
<th>Features</th>
<th>Number of Cells</th>
<th>Particles per Cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>testA</td>
<td>Load balanced case used for benchmark.</td>
<td>1200x1200</td>
</tr>
<tr>
<td>testB</td>
<td>A typical case used in researches, load imbalance.</td>
<td>2500x240</td>
</tr>
</tbody>
</table>
Intel Knights Landing overview

- 36 Tiles, 2 cores per tile
- 4 Threads per core
- 2 x 512bit Vector Processing Units per core
- 6 channels of DDR4 2400 up to 384GB @ ~81 GB/s Streams Triad
- 8GB/16GB of on-package MCDRAM memory @ ~419 GB/s Streams Triad
- 2.7 TF DGEMM.
- OPA (Intel® Omni-Path Architecture) is integrated.
Related work

• Many implementations proved the parallelism of the origin PIC algorithm.
  • Bastrakov et al. reported their implementation of PIC code achieved up to 7x speedup on an 8-core Xeon E5-2690 processor. (Sergey Bastrakov, et al. Particle-in-cell plasma simulation on cpus, gpus and xeon phi coprocessors. In ISC, pages 513–514. Springer, 2014.)

• Ionization is included in the VLPL-S code, thus more memory consumption and more communication cost inevitably happen.
## Test machine configurations

<table>
<thead>
<tr>
<th>Name</th>
<th>Xeon Phi 7210</th>
<th>E5-2699v3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sockets</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Frequency</td>
<td>1.3GHz</td>
<td>2.3GHz</td>
</tr>
<tr>
<td>Cores</td>
<td>64</td>
<td>36</td>
</tr>
<tr>
<td>Threads</td>
<td>256</td>
<td>72</td>
</tr>
<tr>
<td>DDR4 Memory</td>
<td>6×16GB</td>
<td>8×16GB</td>
</tr>
<tr>
<td>MCDRAM</td>
<td>16GB</td>
<td>N/A</td>
</tr>
<tr>
<td>Short Name</td>
<td>KNL</td>
<td>HSW</td>
</tr>
</tbody>
</table>
Performance of the initial implementation of VLPL-S code

- Quadrant/Flat/DDR 256P
- Quadrant/Flat/DDR 128P
- Quadrant/Flat/DDR 64P
- Quadrant/Flat/MCDRAM 256P
- Quadrant/Flat/MCDRAM 128P
- Quadrant/Flat/MCDRAM 64P
- Quadrant/Cache 256P
- Quadrant/Cache 128P
- Quadrant/Cache 64P
- SNC4/Cache 256P
- SNC4/Cache 128P
- SNC4/Cache 64P

The lower the better

Elapsed Time(s)

Code version: original code
Performance analysis of the initial implementation of VLPL-S

- Due to the first-principle nature, PIC simulations generally require intensive computation
- AOS data structure causes inefficient memory access
- Vectorization is not used
- IO takes almost 10% of overall time in production cases
- Load imbalance exists in many production cases
  - Distribution of particles is not uniform in initial condition
  - New particles are generated during iterations
Outline

• Introduction

• Optimizations

• Results

• Conclusion
Optimizations

• Compute-oriented Optimizations
  • Memory access optimization
  • Thread level parallelization
  • Vectorization
• Parallel IO
• Dynamic load balancing optimization
Memory access optimization

- The hotspot is traversing over linked list.
  - Irregular memory access hurts the performance.
- Prefetching with intrinsics
  - Prefetch next particle into L1 cache.
  - Prefetch next cell into LLC.
Thread level optimization

• Applying OpenMP results in data hazard.
• Atomic operation ensures the result, but hurts the performance.
• We propose a passive contribution (buffering then reduction) method to avoid the data hazard.
Interim method of vectorization

- Vectorization requires the AOS (array of structure) data structure.
- Packing 8 particles from AOS to SOA (structure of array) temporarily.
Parallel IO

• Rebuild the HDF5 library with
  “CC=mpiicpc ./configure --enable-parallel”

• Modifications on Save function
  • All MPI Processes write to the same datasheet
  • Only one write operation for one process
Dynamic load balancing optimization (Early stage)

• The distribution of particles in the whole space is not uniform and changing during the iterations in most real test cases
• Evaluate the process load according to the computational time of a whole row/column as a reference
• Tune the computational load by exchanging the cells of whole rows/columns if the load imbalance threshold value is achieved
Outline

• Introduction

• Optimizations

• Results

• Conclusion
Performance is improved up to \textbf{1.53x} on KNL by Compute-oriented Optimizations

\textbf{The higher the better}

Speedup

<table>
<thead>
<tr>
<th></th>
<th>testA</th>
<th>testB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xeon</td>
<td>1.19</td>
<td>1.12</td>
</tr>
<tr>
<td>Xeon Phi</td>
<td>1.25</td>
<td>1.40</td>
</tr>
<tr>
<td>Xeon</td>
<td>1.33</td>
<td>1.17</td>
</tr>
<tr>
<td>Xeon Phi</td>
<td>1.53</td>
<td>1.31</td>
</tr>
</tbody>
</table>


KNL : 7210P Flat Quadrant
VLPL-S is **1.77X** faster on KNL to two Haswell CPU

![Speedup Chart](chart.png)

**The higher the better**

<table>
<thead>
<tr>
<th></th>
<th>HSW 18Px2T</th>
<th>KNL 64Px4T</th>
</tr>
</thead>
<tbody>
<tr>
<td>testA</td>
<td>1</td>
<td>1.77</td>
</tr>
<tr>
<td>testB</td>
<td>1</td>
<td>0.95</td>
</tr>
</tbody>
</table>
I/O is boosted up to **10X**

The lower the better

![Graph showing I/O times for testA and testB with Serial IO and Parallel IO comparisons.](image-url)
Dynamic load balancing optimization improves 11% performance on KNL

![Graph showing speedup comparison between baseline and dynamic load balancing optimization.](image)

**Speedup**  The higher the better

- Test A: Baseline 0.99, Dynamic load balancing optimization 1
- Test B: Baseline 1, Dynamic load balancing optimization 1.11

**KnL**: 7210P Flat Quadrant
Outline

• Introduction
• Optimizations
• Results
• Conclusion
Conclusion

- The performance of VLPL-S on KNL 7210 is up to **1.77X faster** than it on a two-socket E5-2699v3 node

- Optimizations of VLPL-S mentioned above work both on Xeon and Xeon Phi

- MCDRAM greatly improves the performance for memory-bound applications

- Thread level parallelism helps reducing the communication overhead and improves the load balancing among threads, thus improves the performance, especially on KNL Clusters
Future work

- Completely rewrite the linked list to SOA and evaluate the performance of vectorization

- Improve workload optimization to further avoid load imbalance