Extending Bridge++ Lattice Simulation Code to Vector Processors

T. Aoyama (KEK), I. Kanamori (RIKEN), H. Matsufuru (KEK), Y. Namekawa (YITP) for Bridge++ project

August 4-7, 2020
Asia-Pacific Symposium for Lattice Field Theory (APLAT 2020)

Outline

- Bridge++ Project for General-purpose Code Set of Lattice Gauge Theory Simulations
- Overview of NEC SX-Aurora TSUBASA system
- Porting and Optimizing Bridge++ to SX-Aurora
- Summary
Bridge++: A Lattice QCD Code Set

• **Overview**
  • General-purpose code set for simulations of Lattice Gauge Theory.
  • Object-oriented design using C++.
  • Development policy:
    • Readable, Extendable, Portable, and High-performance.

• **History**
  • Launched in 2009, first public release in 2012.
  • Latest version: 1.5.4 (March 2020).
  • Adopted in research works, acknowledged in 48 papers.

• **Members**
  • Y. Akahoshi, S. Aoki, Y. Namekawa (YITP), T. Aoyama, H. Matsufuru (KEK), I. Kanamori (RIKEN), K. Kanaya, Y. Taniguchi (Tsukuba), H. Nemura (RCNP), and contributors.
Bridge++: A Lattice QCD Code Set

- Development of Bridge++
  - Provides a tool set as a library for User applications, including:
    - Various Fermion and Gauge actions
    - Linear Solver algorithms, Linear algebraic operations
    - Simulation algorithms, Random numbers, I/O manipulations

- Target platforms
  - “core” library:
    - scalar processors, multicore cluster systems
    - OpenMP + MPI for parallelization

- Extensions:
  - system-specific implementations
    - Manycore processors and wide SIMD instructions, e.g. KNL, Intel Skylake I. Kanamori and H. Matsufuru (2017)
  - Vector processors
Bridge++: A Lattice QCD Code Set

- **Platform-specific extensions**: “alternative” to core library
  - “core” library
    - Fixed data type (double precision) and layout (AoS-type).
  - “alternative”
    - Arbitrary data type and layout.
    - Allows code structure with e.g. directives and separate kernels.

- Drop-in replacement of “core” library modules
  - Keeps class structure same between core and alternatives.
  - Sets interface layer.
    - e.g. “propagator calculation class” that calls Dirac op and Solver
    - Data layout conversion built-in.

- Need to implement performance-aware part for each platform.
  Algorithms common to platforms are implemented as C++ templates for code reuse.
SX-Aurora TSUBASA: Overview

- **Newest product of NEC SX series**
  - Processors on PCIe card form factor, equipped in Xeon servers via PCIe Gen3.

- **Vector Engine (VE)**
  - Vector processor with 8 cores.
  - 64 vector registers of 16 kbit each.
  - HBM2 memory 6ch provides 1.2 TB/s bandwidth.

<table>
<thead>
<tr>
<th></th>
<th>SX-Aurora</th>
<th>Xeon CascadeLake</th>
<th>NVIDIA V100</th>
</tr>
</thead>
<tbody>
<tr>
<td># cores</td>
<td>8</td>
<td>28</td>
<td>5120</td>
</tr>
<tr>
<td>DP Performance</td>
<td>2.45 TFlops</td>
<td>2.42 TFlops</td>
<td>7.8 TFlops</td>
</tr>
<tr>
<td>Memory Capacity</td>
<td>48/24 GB</td>
<td>up to 1 TB</td>
<td>32/16 GB</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>1.2 TB/s</td>
<td>140 GB/s</td>
<td>900 GB/s</td>
</tr>
<tr>
<td>Memory Type</td>
<td>HBM2</td>
<td>DDR4</td>
<td>HBM2</td>
</tr>
<tr>
<td>Cache</td>
<td>16 MB shared</td>
<td>38.5 MB</td>
<td>6 MB</td>
</tr>
<tr>
<td>B/F</td>
<td>0.5</td>
<td>0.06</td>
<td>0.12</td>
</tr>
</tbody>
</table>

- **Vector Host (VH)**
  - Xeon server accommodates 8 VEs.
  - VHs interconnected by Infiniband EDR: up to 64 VEs in a Rack.
SX-Aurora TSUBASA

- **Programming Model**
  - “VE execution model”
    - Program runs on VE, as if on an ordinary node.
    - System calls e.g. I/O are offloaded to VH underneath.
    - cf. GPUs: offload tasks from host program to devices.
  - Ordinary C/C++/Fortran programs run just by recompilation.
    - Directives to control in detail.

- **Software Environment**
  - NEC C/C++/Fortran Compilers with auto vectorization. OpenMP supported. MPI library provided.
  - BLAS, LAPACK, and other optimized mathematical libraries.
  - Profiler and Debugger.
  - LLVM-based compiler also being developed.
    - Intrinsics available. Recent release supports auto-vectorization.
    - cf. LLVM-VE-RV project on github.
SX-Aurora TSUBASA

• Related work:
  
  
  • Apply Grid Lattice QCD framework designed for processors with SIMD instructions to large vector registers of length $128 \times 2^k$ bits.
Porting and Optimizing Bridge++

- **Strategy**
  - Vectorization along site-loop.
    - Rely on compiler’s auto-vectorization.
    - Promote loop unrolling for color/spin d.o.f. using constants.
  - Switching Data Layout
    - “core” library: Array of Structure (AoS) layout.
      - site d.o.f. packed innermost.
    - Vector library: Structure of Array (SoA) layout.
      - contiguous w.r.t. site loop index.

---

<table>
<thead>
<tr>
<th></th>
<th>core library (AoS)</th>
<th>Vector library (SoA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wilson mult on 1 core</td>
<td>1.07 GFlops</td>
<td>41.7 GFlops</td>
</tr>
</tbody>
</table>
Porting and Optimizing Bridge++

- **Parallelization within VE**
  - Using 8 cores in a VE via flat-MPI.
  - Performance:

<table>
<thead>
<tr>
<th></th>
<th>Single core 16x16x8x8</th>
<th>1 VE (8 cores) 16x16x16x32 /[1,1,2,4]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wilson mult</td>
<td>41.7 GFlops</td>
<td>81.8 GFlops</td>
</tr>
<tr>
<td>peak performance</td>
<td>308 GFlops</td>
<td>2.42 TFlops</td>
</tr>
<tr>
<td>theoretical bandwidth</td>
<td>409 GB/s</td>
<td>1.2 TB/s</td>
</tr>
<tr>
<td>expected from B/F ~ 2.2</td>
<td>180 GFlops</td>
<td>540 GFlops</td>
</tr>
</tbody>
</table>

- **Profiling:**
  - Vector instruction ratio: ~99.90 %
  - Average Vector Length: 256.0 → seems well vectorized.
Further optimization

- **Memory subsystem overview**
  - 6 HBM2 modules connected.
  - 8 channels on each module.
    - 128bits interface x 8ch x 1.6GHz x 6HBM2 = 1.2 TB/s
  - 128 byte-cells within module.
    - classified in 32 banks
    - access to contiguous 128 bytes will be most effective
  - VE equips 16 MB LLC shared by 8 cores.
    - connected by NOC (network on chip).
    - bandwidth between LLC and core = 409.6 GB/s.

- Bank conflict occurs by simultaneous access with 192KB strides
  - 128bytes x 6HBM2 x 8ch x 32banks (round-robin) = 192KB
Further optimization

- STREAM benchmark with multiple streams.

```c
for (i=0; i<n; ++i) {
    a0[i] = b0[i] + v * c0[i];
    a1[i] = b1[i] + v * c1[i];
    a2[i] = b2[i] + v * c2[i];
    ....
}
```

- Insert **padding** of appropriate size to avoid bank conflicts.
- Padding size to be chosen by examining memory access pattern.
  - x 1.5 for Wilson mult.

Further optimization

- **Vector processing unit overview**
  - 64 vector registers, vector length 256 elem. of 8B.
  - 32 vector pipelines (VPP).
  - 1 vector instruction execute 256 arith ops with 8 clock cycles.
  - 6 execution pipes:
    - 3 FMAs, 2 ALUs, 1 DIV/SQRT.

- Invoke vector instructions: two-fold loops as an idiom.
  - Inner vectorized loop (long enough i.e. \( \geq 256 \) to fill vector registers)
    - apply blocking to inner loop in unit of vector length (VL=256), and specify compiler directives for optimization.
  - Outer loop (maybe further parallelized over threads)

\[(x,y,z,t) \rightarrow (x,y) \text{ and } (z,t) \text{ for Wilson mult}\]
Further optimization

- **Wilson mult performance evolution:**

<table>
<thead>
<tr>
<th></th>
<th>Single core</th>
<th>1 VE (8 cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td>core library (AoS)</td>
<td>1.07 GFlops</td>
<td></td>
</tr>
<tr>
<td>Vector library (SoA)</td>
<td>41.7 GFlops</td>
<td>81.8 GFlops</td>
</tr>
<tr>
<td>insert padding</td>
<td></td>
<td>133 GFlops</td>
</tr>
<tr>
<td>two-fold loop, blocking, vector register directives</td>
<td></td>
<td>~200 GFlops</td>
</tr>
<tr>
<td>pack/unpack revised</td>
<td>70.1 GFlops</td>
<td>271.5 GFlops</td>
</tr>
<tr>
<td>peak performance</td>
<td>308 GFlops</td>
<td>2.42 TFlops</td>
</tr>
</tbody>
</table>

- About 20% (single core), or 10% (1 VE) of peak performance obtained.
- Compared to the expected performance from memory bandwidth, further improvement may be possible.
Further optimization

- **Wilson mult performance evolution:**
  - Multiple VEs for $32^3 \times 8N$ with $N$ up to 128 VEs.

  \[
  \text{Weak Scaling} \\
  32^3 \times 8N
  \]

  \[
  \begin{array}{c|c}
  \text{number of VEs} & \text{mult performance [GFlops]} \\
  \hline
  10 & 100 \\
  100 & 1000 \\
  1000 & 10000 \\
  10000 & 100000 \\
  100000 & 23.7 \text{Tflops}
  \end{array}
  \]

  *Preliminary*

- Shows better weak scaling. Performance relies on volume and communication overhead. Further investigation is needed.
Summary

- Extension of Bridge++, a general-purpose code set for lattice simulations, to vector processors is being carried out. Strategy for platform-specific extension is overviewed.

- Data layout significantly affects the performance. Data should be aligned contiguously along site loop index (SoA format), efficient for load into vector registers.

- Further elaborations on memory access pattern, e.g. including padding to bank conflict improve performance significantly.

- An idiom to write vector loops will be two-fold loops, vectorized inner loop and outer loop. Further optimization to be applied e.g. loop blocking and compiler directives.

- A good weak scaling is observed up to 128 VEs, though further investigation and improvement will be needed.