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
      - GPUs w/ OpenCL, OpenACC — S. Motoki (2015), H. Matsufuru et al (2015)
      - Manycore processors and wide SIMD instructions, e.g. KNL, Intel Skylake — I. Kanamori and H. Matsufuru (2017)
      - **Vector processors**

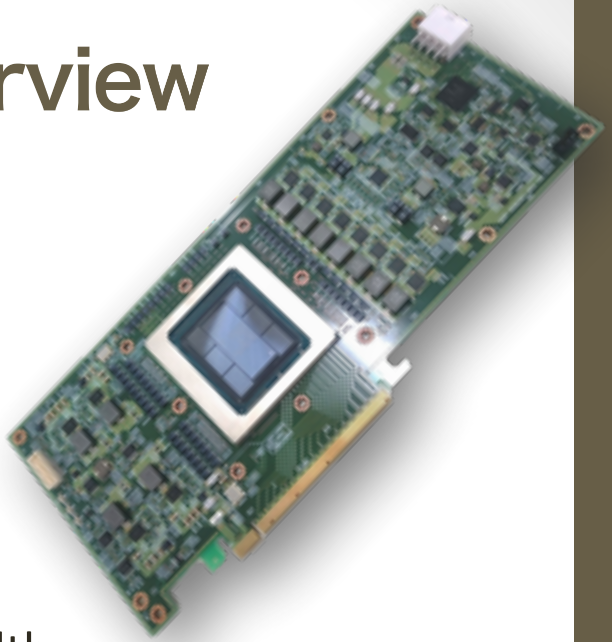
Bridge++: A Lattice QCD Code Set

- **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.

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

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

SX-Aurora TSUBASA

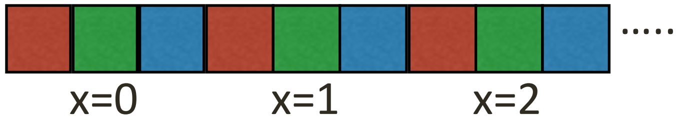
- **Programming Model**
  - “VE execution model”
    - Program runs on VE, as if on an ordinary node.
    - System calls e.g. I/O are offloaded to VH underneath.
    - cf. GPUs: offload tasks from host program to devices.
  - Ordinary C/C++/Fortran programs run just by recompilation.
    - Directives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

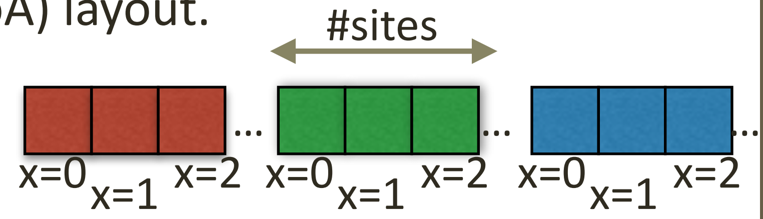
- **Related work**:
  - “Lattice QCD on a novel vector architecture”, B. Huth, N. Meyer, T. Wittig, LATTICE2019, arXIv:2001.07557[cs.DC].
    - Apply Grid Lattice QCD framework designed for processors with SIMD instructions to large vector registers of length 128x2^k bits.

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.



	core library (AoS)	Vector library (SoA)
Wilson mult on 1 core	1.07 GFlops	41.7 GFlops

Porting and Optimizing Bridge++

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

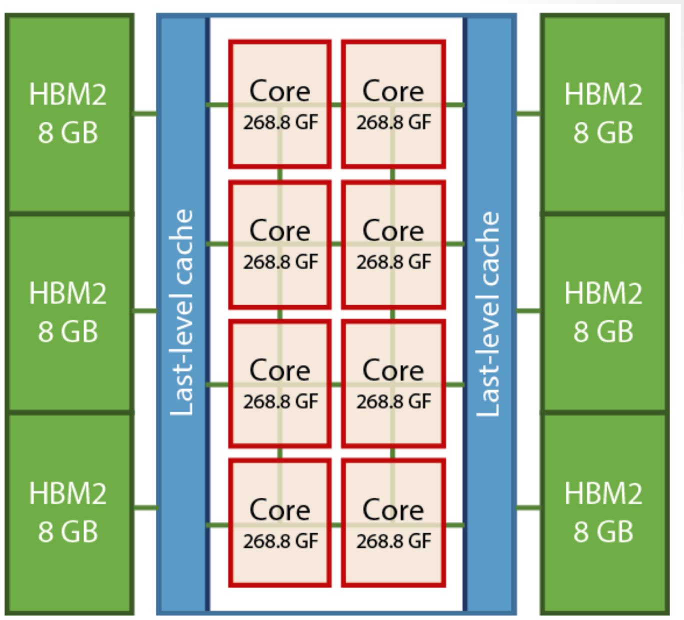
	Single core 16x16x8x8	1 VE (8 cores) 16x16x16x32 /[1,1,2,4]
Wilson mult	41.7 GFlops	81.8 GFlops
peak performance	308 GFlops	2.42 TFlops
theoretical bandwidth	409 GB/s	1.2 TB/s
expected from B/F ~ 2.2	180 GFlops	540 GFlops

  - 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

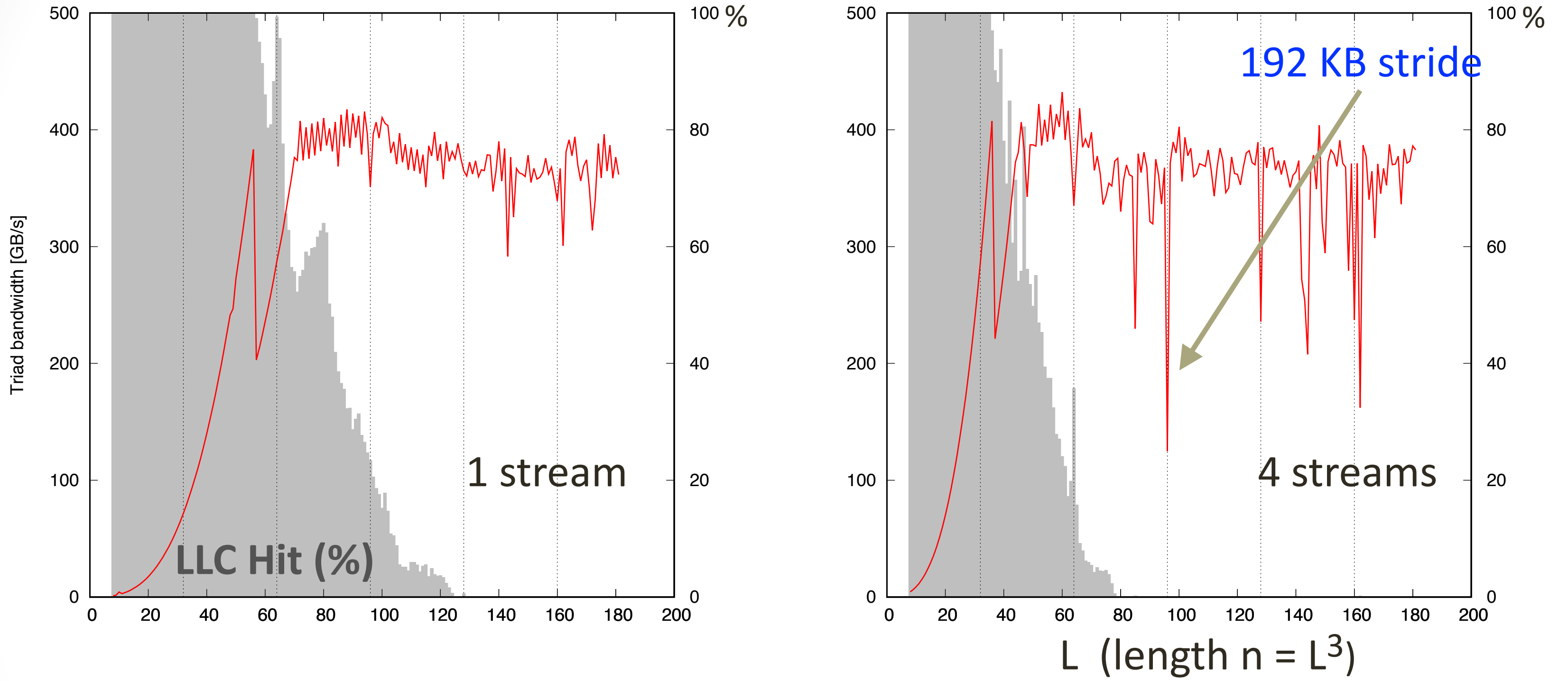


Block diagram of a vector processor

Further optimization

- STREAM benchmark with multiple streams.

```
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];
    ....
```



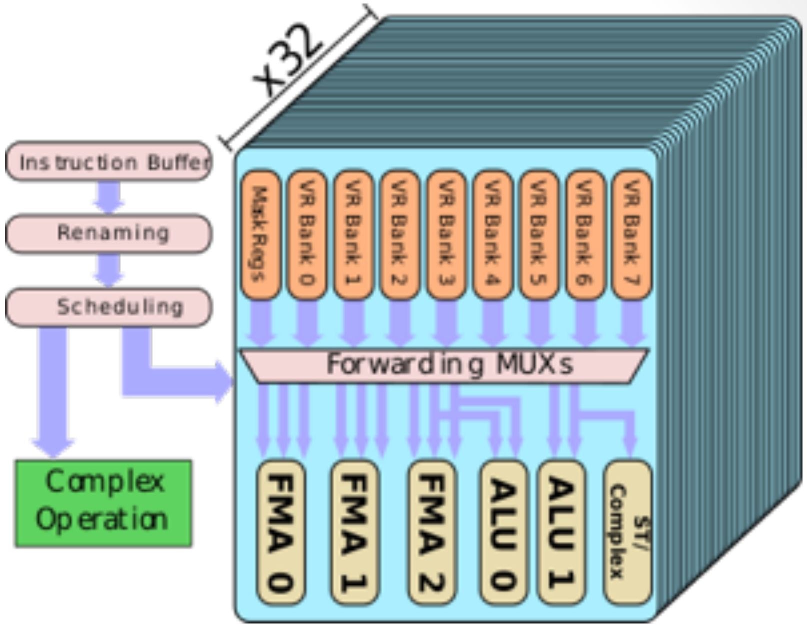
- Insert padding of appropriate size to avoid bank conflicts.
- Padding size to be chosen by examining memory access pattern.

x 1.5 for Wilson mult.

cf. Memory First: A performance tuning strategy focusing on memory access patterns, N. Ebata, R. Egawa, Y. Isobe, R. Takaki, H. Takizawa, Research Poster at ISC2019.

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.



from WikiChip

- Invoke vector instructions: two-fold loops as an idiom.
  - Inner vectorized loop (long enough i.e. ≧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) → (x,y) and (z,t) for Wilson mult

Further optimization

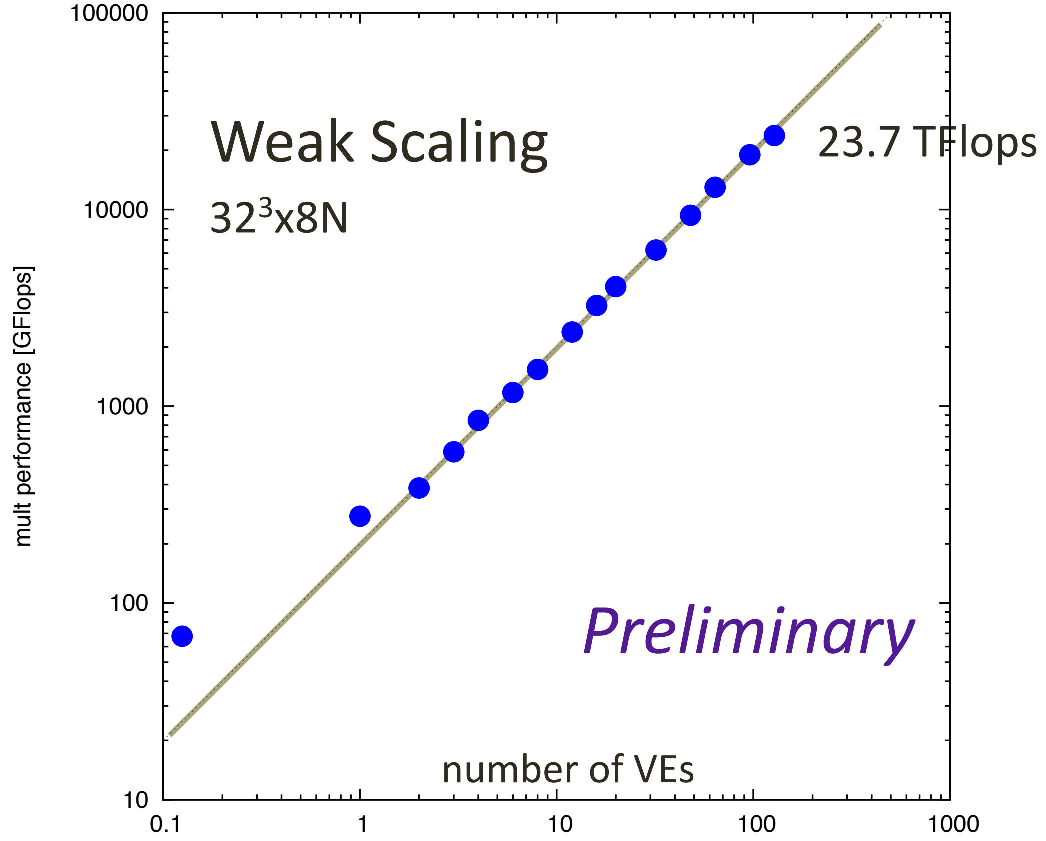
- **Wilson mult performance evolution:**

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

- 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 32^3x8N with N up to 128 VEs.



- 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.