Performance Portability of QCD with Kokkos

Balint Joo - Jefferson Lab
Jack Deslippe, Thorsten Kurth - NERSC
Kate Clark - NVIDIA
Dan Ibanez, Dan Sunderland - Sandia National Lab

IXPUG 2017 US Fall Meeting, Oct 26, TACC Austin TX

Introduction

- With researchers being faced by a diverse array of hardware, and looking to future systems, performance portability of code becomes increasingly important
  - researchers do not have time to rewrite code for each new architecture
  - but they may have to run at several centers (TACC, NERSC, OLCF, BlueWaters…)
- Kokkos is a C++ template library that provides abstractions for parallel patterns which enable programming in a performance portable way:
  - parallel_for, parallel_reduction, parallel_scan, in several Execution Spaces (OpenMP,CUDA…)
  - multi-dimensional arrays via Kokkos::Views, in several Memory Spaces (Host, MCDRAM,…)
  - back ends for OpenMP, CUDA, OpenMP Target Device, pthreads, qthreads, etc.
  - optimizations for a variety of processors for e.g. atomics (BWD, KNL, Power, Kepler, Pascal)
- Here, we report on study of performance portability of an Lattice QCD (LQCD) Kernel

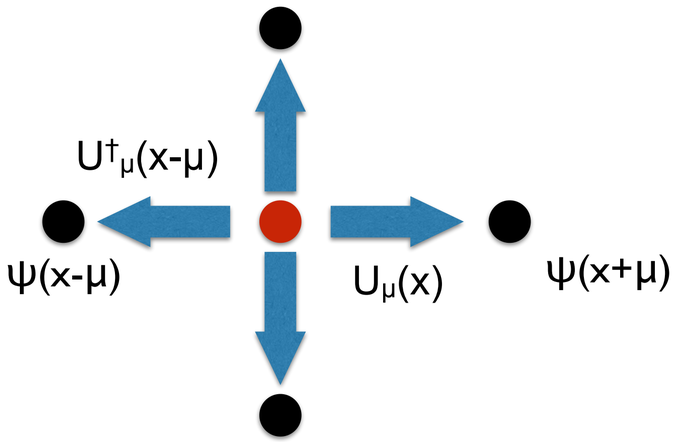
Wilson Dslash Operator

$$D^{ab\alpha\beta}_{y,x}\psi^{b\beta}_{x} = \sum_{\mu}(1-\gamma_\mu)^{\alpha\beta}U^{ab}_{\mu}(x)\psi^{b\beta}_{x+\hat{\mu}} + (1+\gamma_\mu)^{\alpha\beta}U^{ab\ \dagger}_{\mu}(x-\hat{\mu})\psi^{b\beta}_{x,x-\hat{\mu}}$$

spinor:
4 spins (β)
3 colors (b)
= 12 complex numbers

gauge link:
3x3 matrix (a,b)
= 18 complex numbers

spin projector:
projects out 2 spin degrees of freedom
from 4
(sparse 4x4 matrix)



forward neighbor spinor
in μ direction

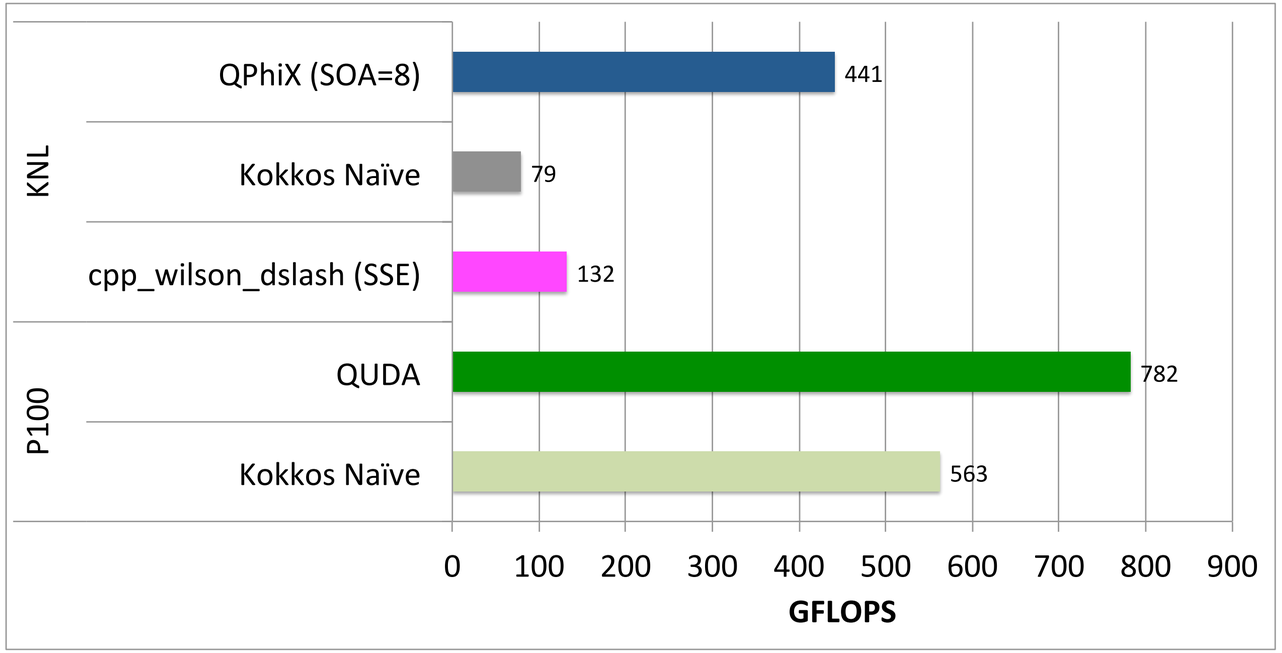
back neighbor spinor
in μ direction

Or in pseudo-pseudo code

```
Kokkos::parallel_for(Kokkos::TeamThreadRange(team,start_idx,end_idx),KOKKOS_LAMBDA(const int site) {

      SpinorSiteView<TST> res_sum;              // 4-spins: struct {  TST _data[3][4]; };  with TST=complex<>

      HalfSpinorSiteView<TST> proj_res;         // 2-spins: struct {  TST _data[3][2]; }:  with TST=complex<>
      HalfSpinorSiteView<TST> mult_proj_res;

      for(int color=0; color < 3; ++color)
         for(int spin=0; spin < 4; ++spin)
             ComplexZero(res_sum(color,spin));

      // T - minus
      KokkosProjectDir3<ST,TST,isign>(s_in, proj_res,neigh_table(site,target_cb,T_MINUS));
      mult_adj_u_halfspinor<GT,TST>(g_in_src_cb,proj_res,mult_proj_res,neigh_table(site,target_cb,T_MINUS),3);
      KokkosRecons23Dir3<TST,isign>(mult_proj_res,res_sum);

      // Z - minus
      KokkosProjectDir2<ST,TST,isign>(s_in, proj_res,neigh_table(site,target_cb,Z_MINUS));
      mult_adj_u_halfspinor<GT,TST>(g_in_src_cb,proj_res,mult_proj_res,neigh_table(site,target_cb,Z_MINUS),2);
      KokkosRecons23Dir2<TST,isign>(mult_proj_res,res_sum);

      // … other directions: Y-, X-, X+, Y+, Z+, T+ not shown for lack of space
});
```

Initial Kokkos Results

- KNL
  - Performance was very low
  - Even lower than previous legacy codes
  - ***Reason: no vectorization***
    - 3x3 matrices, 3 vectors
    - loop over sites outermost
- GPU
  - Initially large amount of register spill to local memory
  - Needed to adjust CUDA launch bounds for kernels (Kokkos::LaunchBounds policy)
  - After this performance was good



Single RHS Dslash Performances: Vol=32x32x32x32 sites

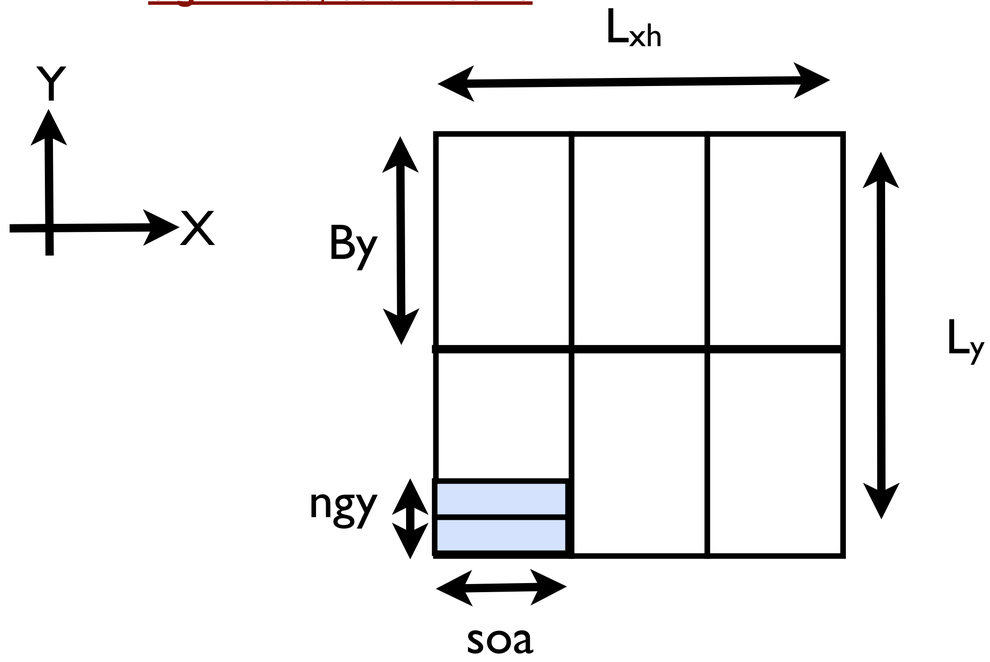
Vectorization Potential

- **Lots** of parallelism over lattice sites (L^4 sites with L=4..32 sites)
- Naive operator: Arrays of Structures (AOS) - no real vectorization potential
  - In the past vectorized 3x3 matrix - vector operations using SSE, AVX, possibly over directions/spin components.
  - gets cumbersome for longer vectors (e.g. length 16).
- Today most implementations vectorize over lattice sites in some way (next slide)
- Alternative: **Multiple-Right Hand Side (MRHS)** Operator: $\chi_i = D(U)\ \psi_i \quad (i=1..N)$
  - Valid science case for solving many systems at once (e.g. quark propagators)
  - Keep AOS layout, trivially vectorize over i
  - Also can reuse Gauge field by a factor of N.
  - Same as Diagonal operator of 4D formulation of DWF fermions

Vector Single Right Hand Side cases

X-Y Tiling, e.g. QPhiX (idea by D. Kalamkar)

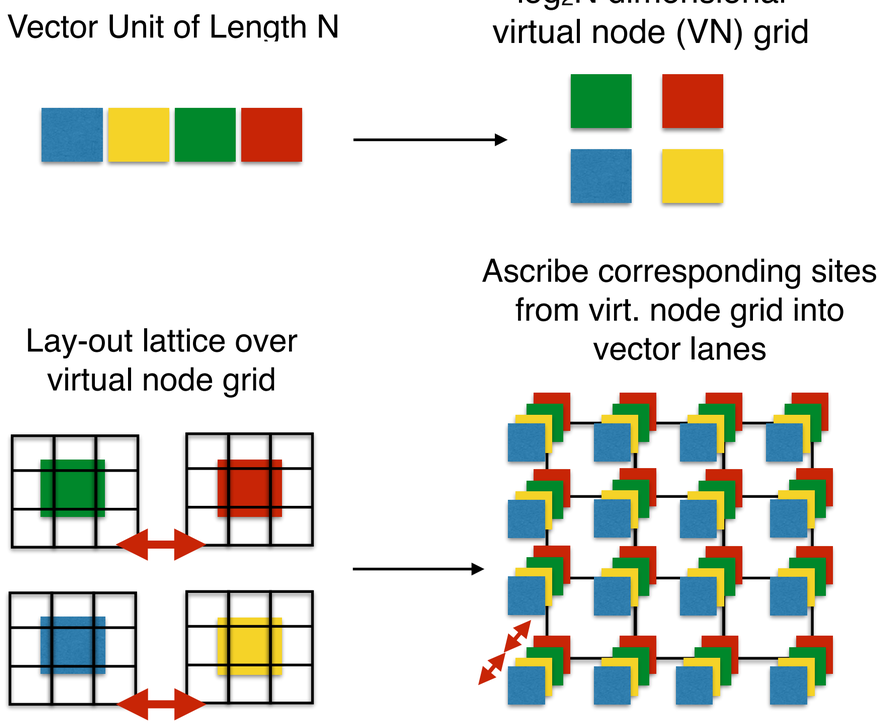
e.g. B. Joo, et. al. ISC'13



Assemble full Vector from ngy x soa pieces:
- ***e.g. ngy=4, soa=4, ngy=2 soa=8, or general gather***
- ***unaligned loads for some neighbors in x-y plane***
- ***user now has to choose soa to suit problem***

Virtual Node Vectorization (P. Boyle, e.g. in Grid, BFM)

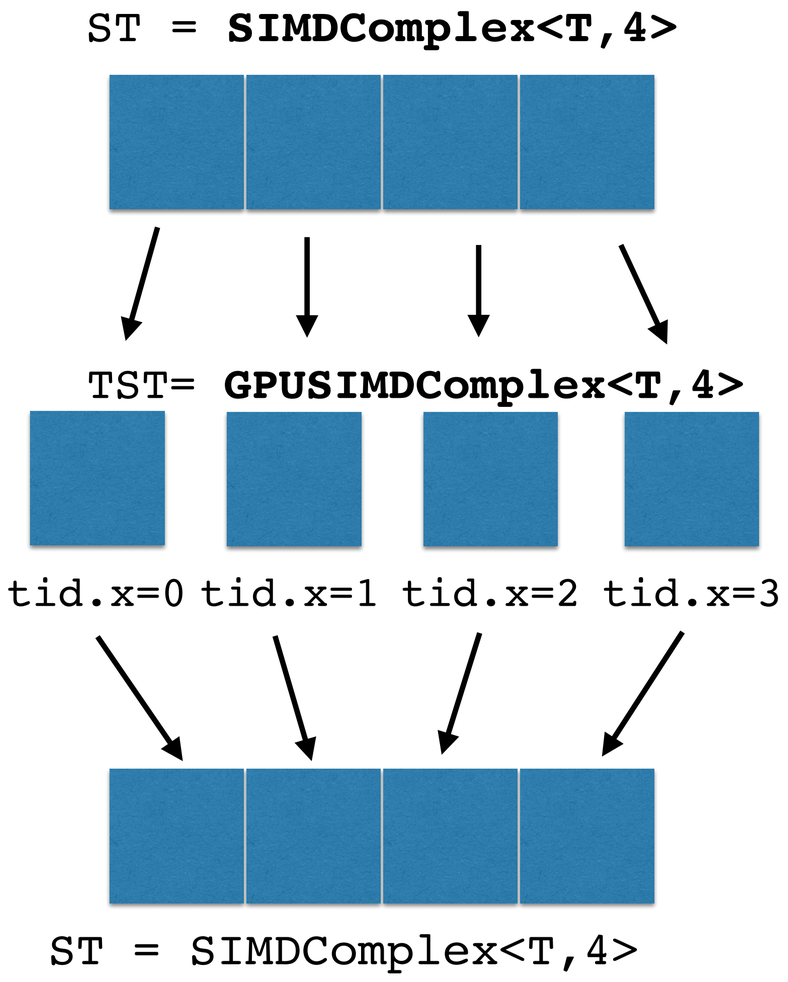
e.g. arXiv:1512.03487[hep-lat]



Nearest neighbor from other VN : data from other lane => lane permutations

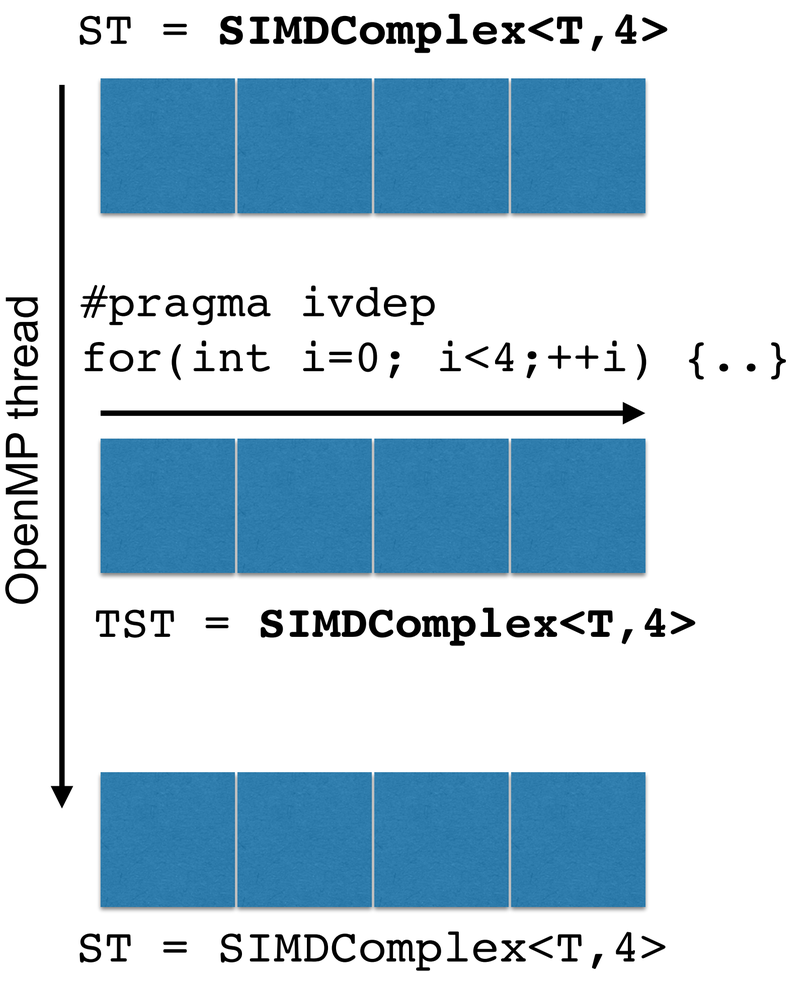
SIMD Types & Portable Vectorization

CUDA backend




```
// view_in and view_out are Kokkos
// views holding type ST
// e.g. View<ST[num_sites][3]>

// Thread private Temporary
TST tmp;

// indices j, k fixed by outer
// parallel_for
parallel_for(
ThreadVectorRange(4),
[=](int i) {

   tmp(i) = (view_in(j,k))(i);

   (view_out(j,k))(i) =
                      2.0*tmp(i);
});
```

OpenMP backend



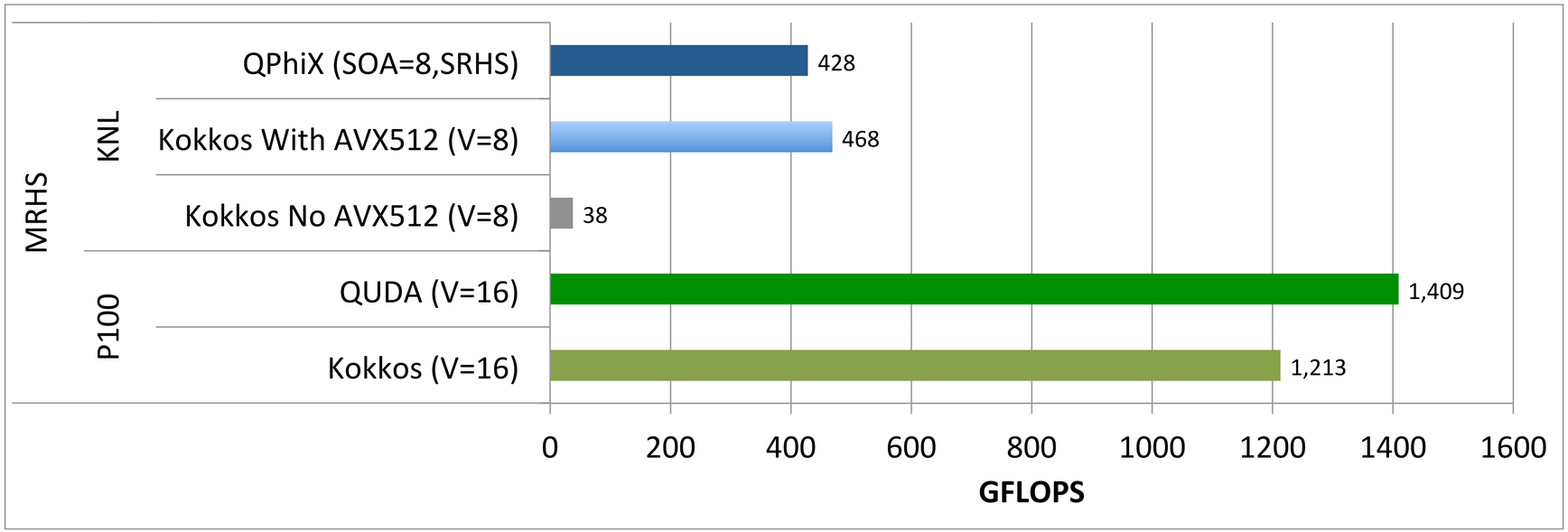
Implementing MRHS Operator

- Dslash Code Identical to SRHS Operator thanks to templates
  - Instead of *MGComplex<T>* use *SIMDComplex<T,N>* in template types for Global Arrays (ST, GT)
  - *SIMDComplex<T,N>* or *GPUSIMDComplex<T,N>* for Thread Local arrays (TST)
- Dispatch Kernels, with *Kokkos::TeamPolicy*, setting Vector length To *N*
  - generates X-dimension of length N for GPU Thread blocks
- KNL:
  - Wrote specializations for Complex Number operations on SIMDComplex<float,8> using AVX512 intrinsics
- GPU:
  - Created struct for complex numbers deriving from 'float2' type for coalesced reads/writes
  - ThreadVectorRange had high overhead for short (single vector) loop, hacked this and by hand inserted threadIdx.x

```
template<>     // KNL Specialization
KOKKOS_FORCEINLINE_FUNCTION
void
A_add_sign_B<float,8,SIMDComplex,SIMDComplex,SIMDComplex>(
                                  SIMDComplex<float,8>& res,
                                  const SIMDComplex<float,8>& a,
                                  const float& sign,
                                  const SIMDComplex<float,8>& b)
{
  __m512 sgnvec = _mm512_set1_ps(sign);
  res._vdata = _mm512_fmadd_ps(sgnvec,b._vdata,a._vdata);
}
```

```
template<typename T, int N> // GPU Specialization
KOKKOS_FORCEINLINE_FUNCTION
void A_add_sign_B( GPUThreadSIMD<T,N>& res,
                   const GPUThreadSIMD<T,N>& a,
                   const T& sign,
                   const GPUThreadSIMD<T,N>& b)
{
  auto _a = a(threadIdx.x); auto _b = b(threadIdx.x);
  T res_re = _a.real();     res_re +=  sign*_b.real();
  T res_im = _a.imag();     res_im +=  sign*_b.imag();
  res(threadIdx.x) = MGComplex<T>(res_re,res_im);
}
```

MRHS Results

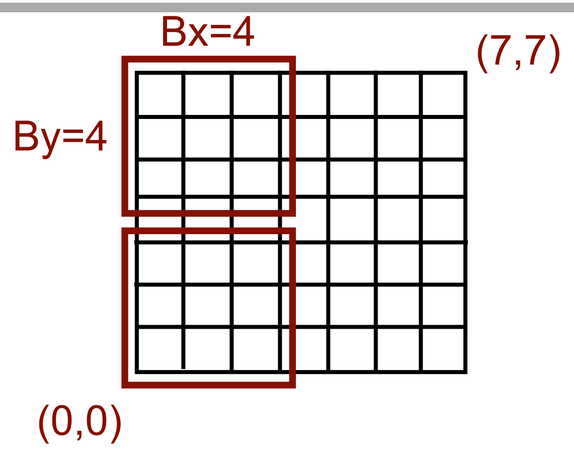


Multiple RHS Dslash Performances:
Vol = 16x16x13x32 sites
V=8 for KNL
V=16 for GPUs
(V=vector length=#RHS)

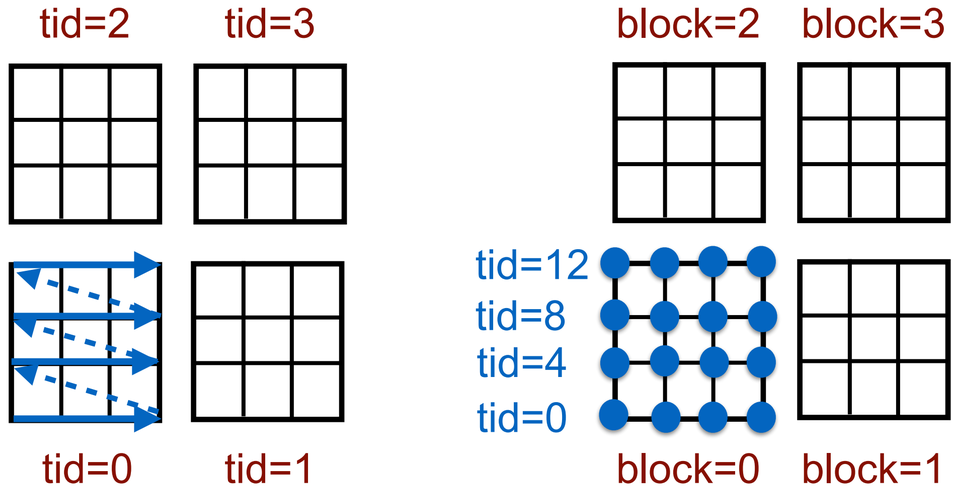
- KNL results are good with AVX512 optimization (slightly more than QPhiX SRHS)
- KNL results are less good without AVX512 optimization (38 GF is v. slow)
- Once ‘threadIdx.x’ specialization was added GPU MRHS code was also great:
  - About 86% of QUDA version.

Vectorized SRHS Operator

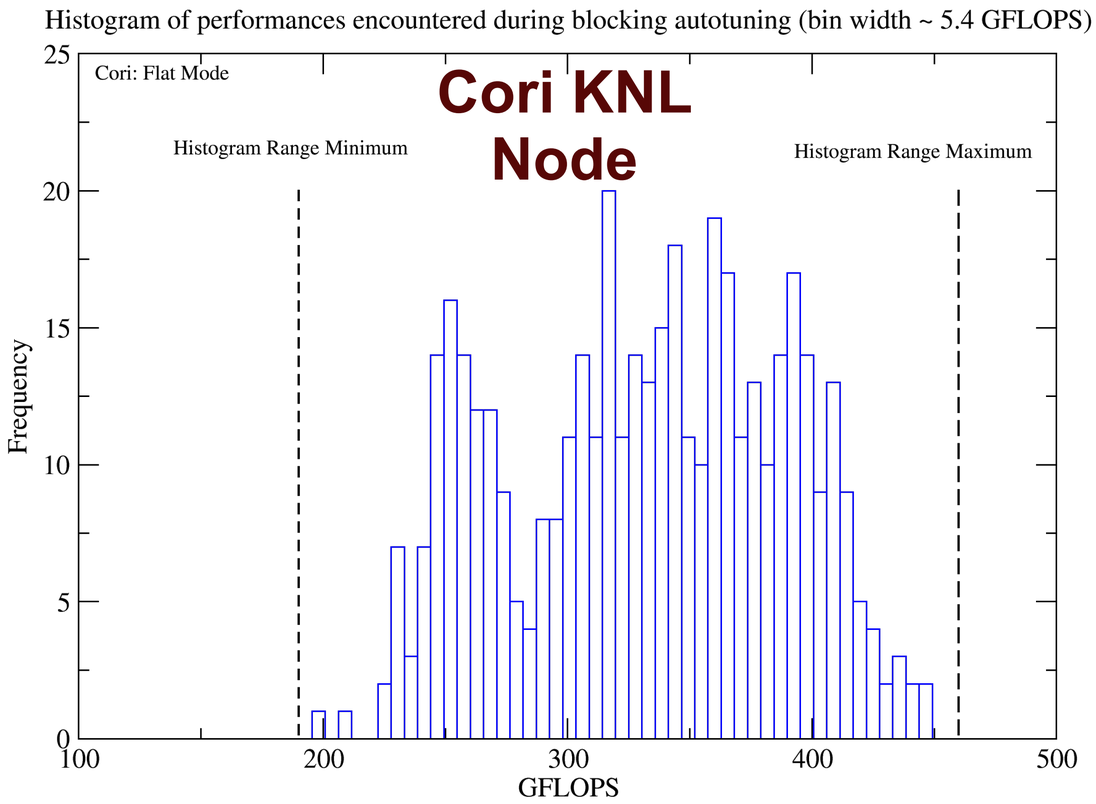
- We implemented the Virtual Node Mode approach
- Discussions about future Kokkos SIMD Type on GPUs:
  - likely SIMD length will be 1, to suit Kokkos::LayoutLeft on GPUs
  - In this case **Vectorized Operator is the same as the Naive operator on GPU,** which is already known performant!
  - GPU Permutes are trivial (identity/not needed)
- Extra KNL optimizations
  - optimize permutes using _m512_permutexvar_ps()
  - spin <-> color interchange (to color fastest): L1 locality
  - 4D Blocking using Kokkos::MDRange exec. policy
    - Autotune block size for performance
  - Gauge Field Access:
    - keep copies of back pointing links => unit stride access for gauge
    - pre-permute links from back neighbor: no gauge permute in Dslash

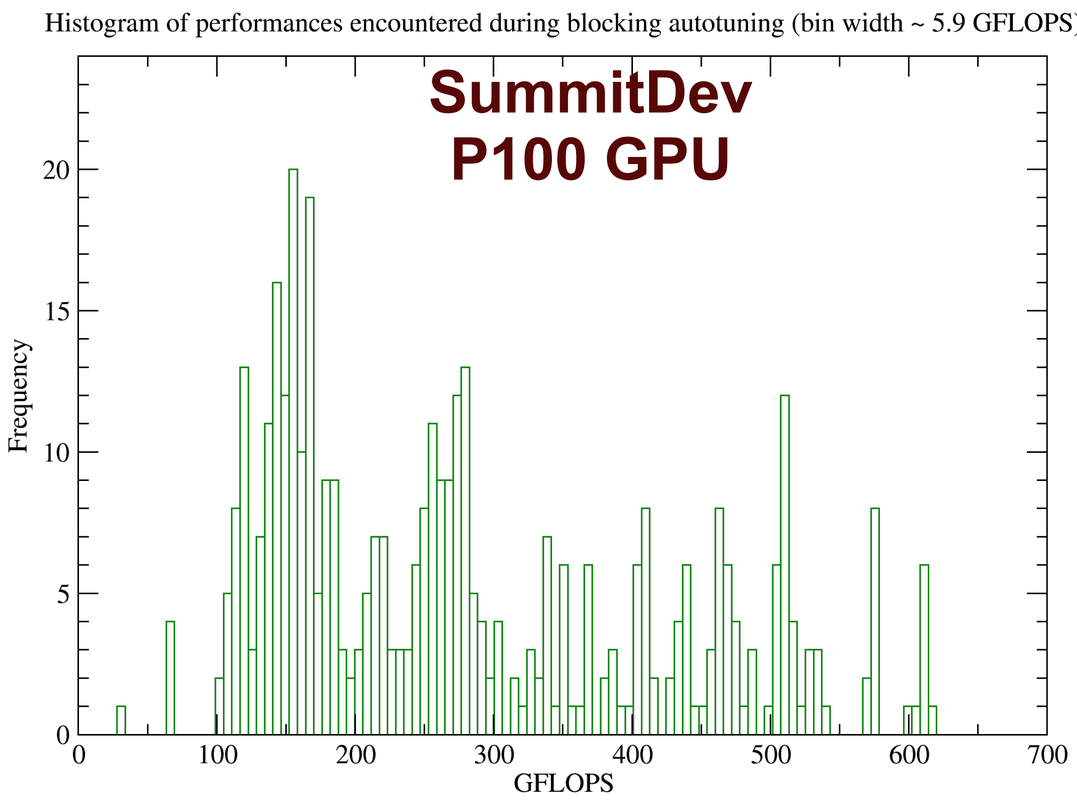



```
MDRangePolicy<2,IterateLeft,IterateLeft>
                policy({0,0},{8,8},{4,4});
```



Block Tuning for MDRange

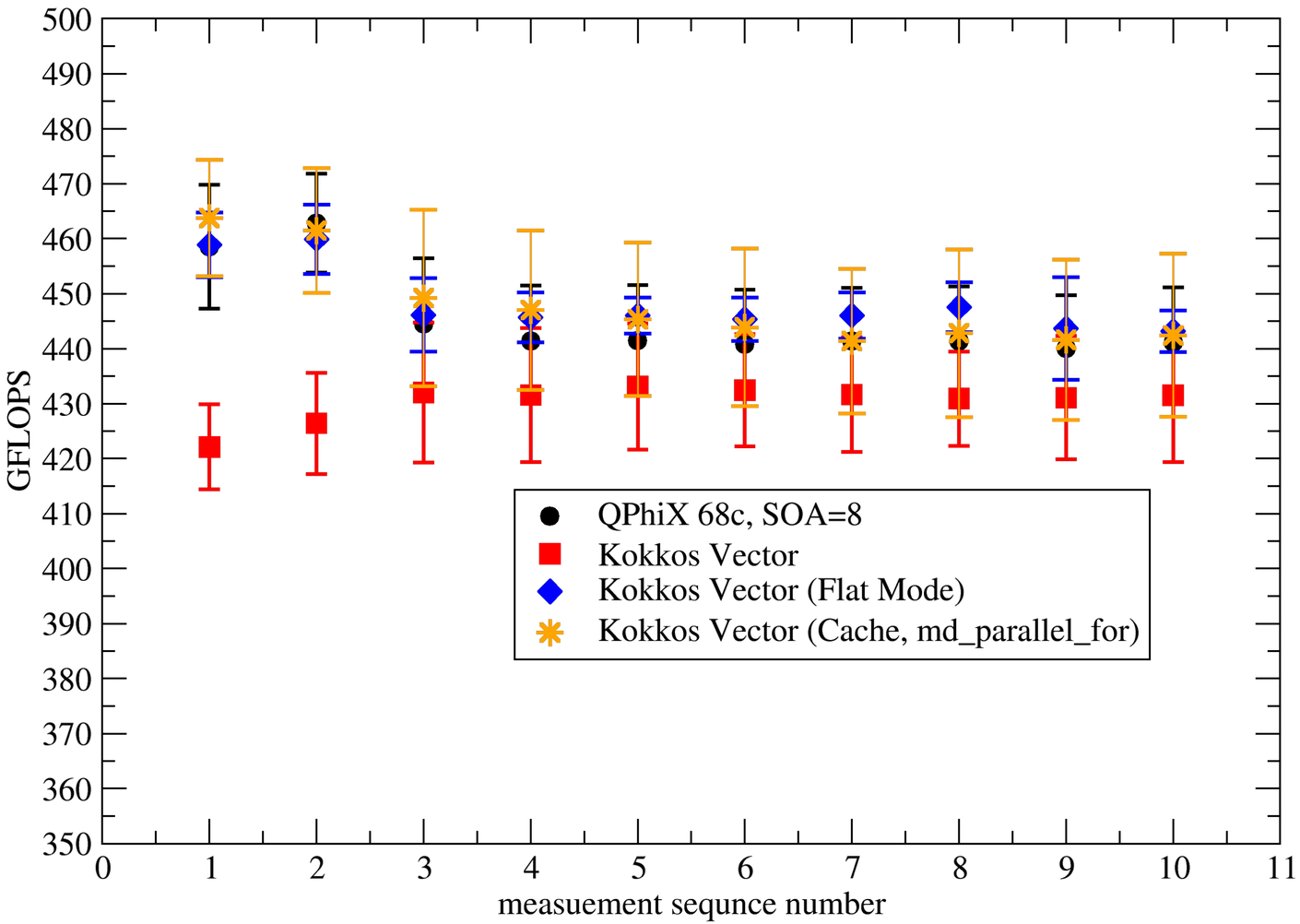




- Block Tuning gave broad performance distributions, with few (<10) tunings giving the highest performance. Autotuning is a must & unfortunately the space is big (4D).

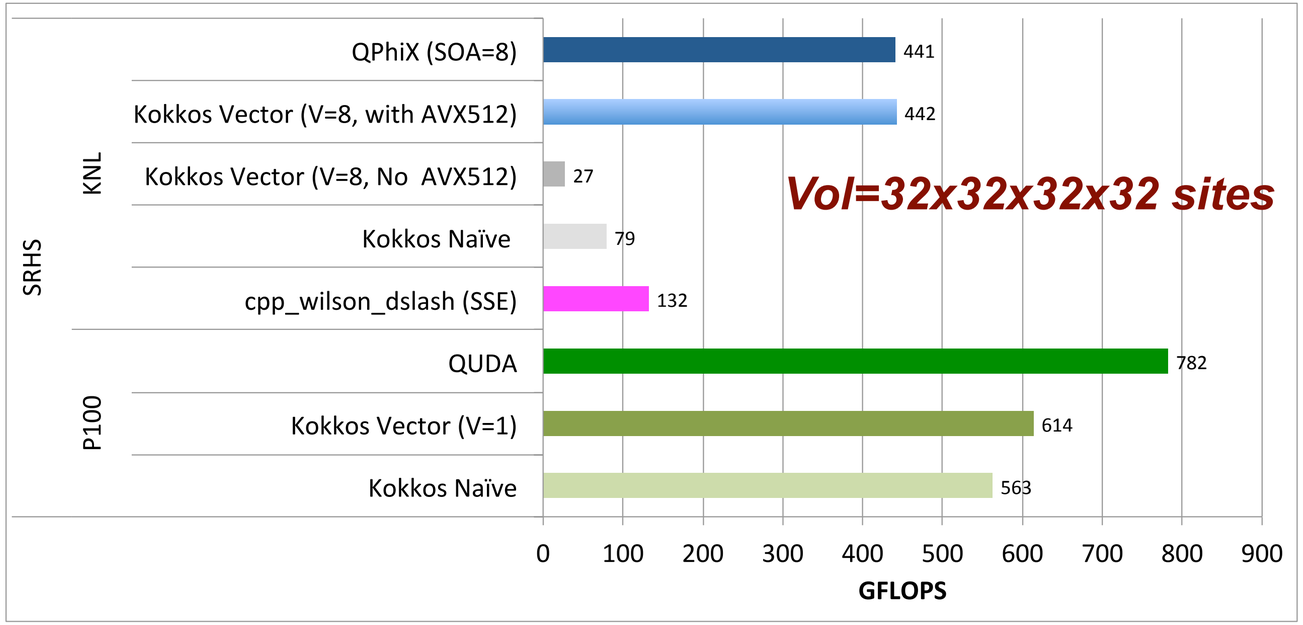
A note about perf. measurements

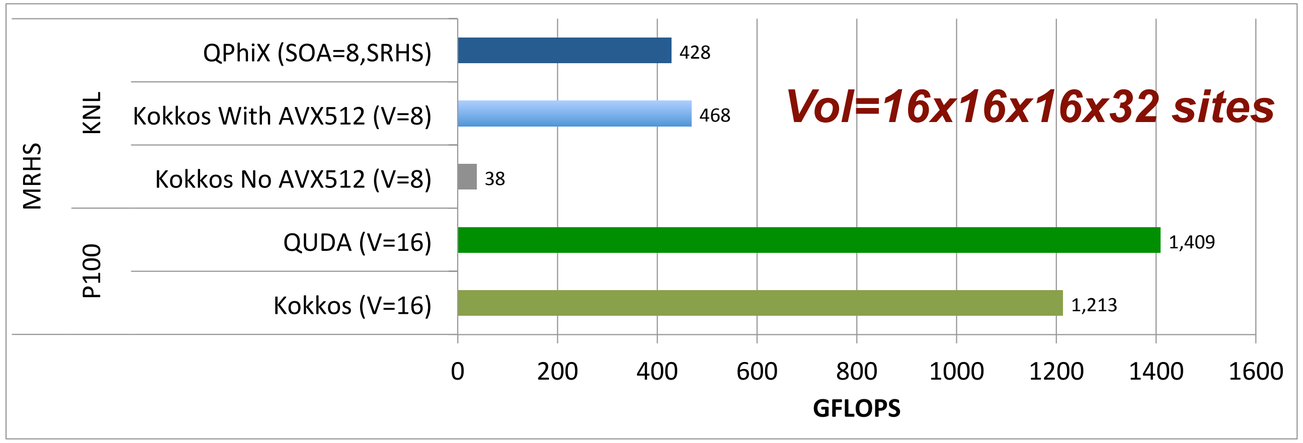
- KNL Performance measurement is not always easy:
  - 5 runs, with 10 timing measurements each
  - Run to run variability (error bars on the measurements)
  - Often first few measurements of the higher than rest?
    - Turbo followed by down-clocking?
    - Can potentially affect autotuning?
- I now quote numbers from the plateaux region



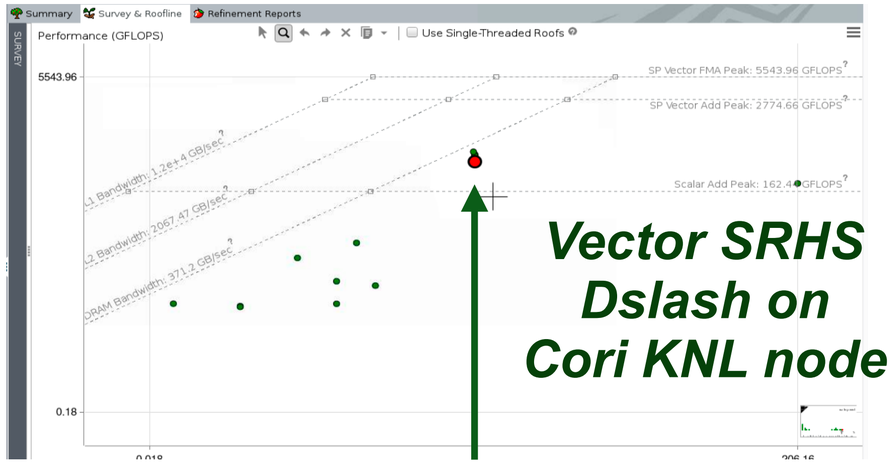
Current Performance Summary

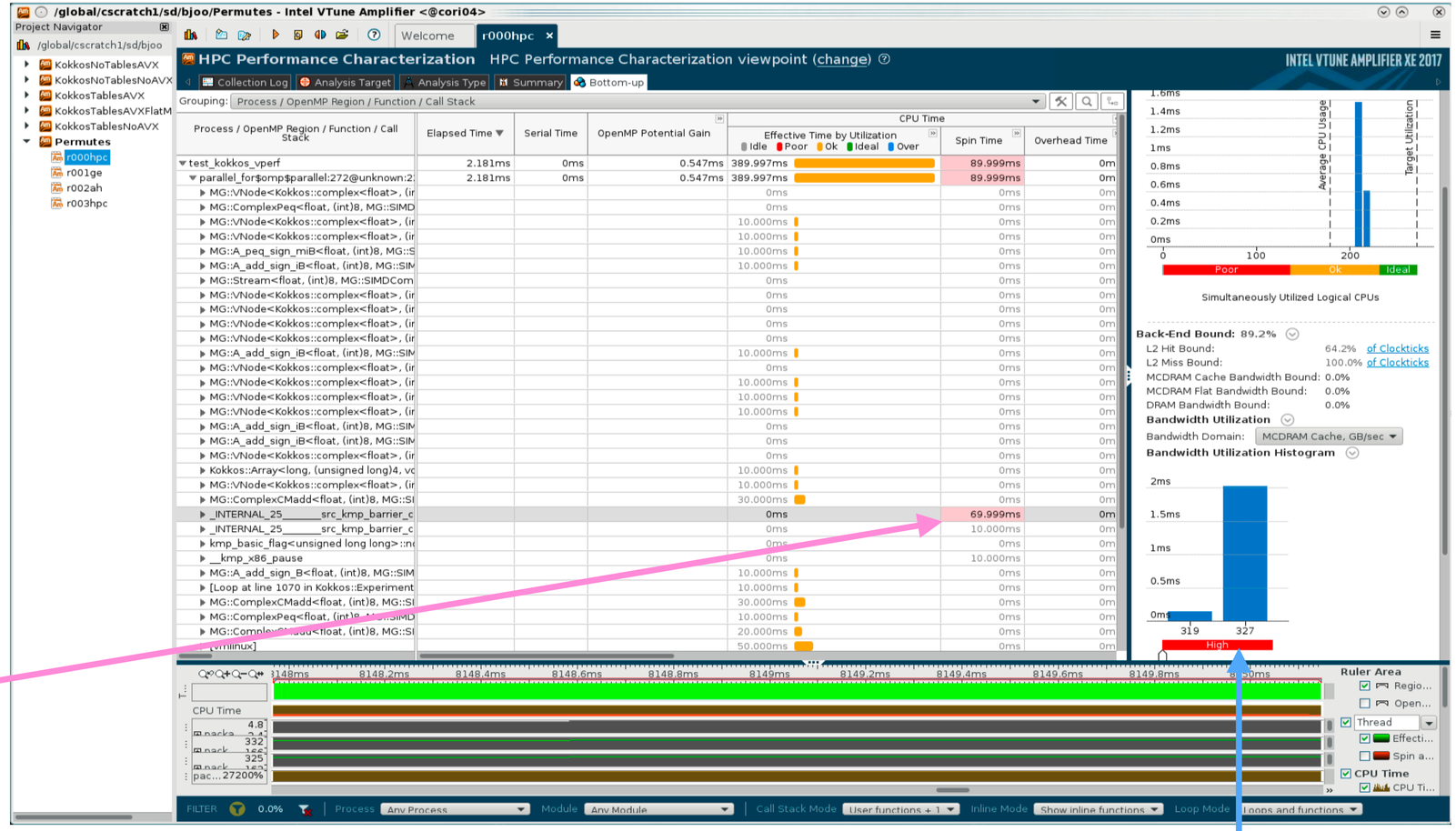
- SRHS Case:
  - ***Kokkos Vectorized Dslash with AVX512 and tuned blocking matches QPhiX on Cori KNL node (68 cores, 272 threads)***
  - Unvectorized & No AVX cases are slow
  - Kokkos Naive CUDA version is 72% of QUDA on P100 (SummitDev)
  - ***Vectorized (but V=1) QUDA version benefits from block tuning, memory & locality optimizations and md_parallel_for: 79% of QUDA on P100 (SummitDev)***
- MRHS Case:
  - ***Kokkos With AVX512 exceeds corresp. QPhiX SRHS performance on Cori KNL node for 8 RHS***
  - Kokkos Without AVX512 is very slow
  - ***Kokkos CUDA version is 86% of QUDA for 16 RHS on SummitDev (P100)***





Absolute Performance is good too





- Little Below Roofline. Reported L1 AI=1.97
- Vtune is concerned with spin wait in barrier
- Good CPU usage and mem BW Usage

Comments & Discussion

- Since Virtual Node Vector SRHS Dslash works well on both CPU and GPU, one may consider dead-ending the naive and MRHS implementations for the future.
- Can implement an MRHS implementation based on Vector Dslash adding inner loops for the code processing the neighbors in each direction.
  - Keep the gauge reuse, but get vectorization from sites. Allows arbitrary number of RHS, not just multiples of the vector length without loss of efficiency.
- For vectors longer than length 1, the specialized X-thread GPU SIMD technique (using threadIdx.x) is not currently compatible with MDRange exec policy. We cannot rely on threadIdx.x being the ‘vector lane’, as MDRange uses it for its own purposes.
  - In this case ThreadVectorRange construct reduces to a simple loop for vectors longer than length 1
  - General ThreadVectorRange code will work but specializations using threadIdx.x must be disabled (will fail)
- We could implement Vectorized Dslash ***not*** using MDRange, but regular ThreadTeam policy. Then X-thread GPU SIMD technique could be implemented for vectors longer than length 1.
  - Lane permutes could possibly be implemented with __shfl() instructions

Possible Future Work

- A bit more performance exploration (different volumes, weak scale on node etc.)
- Work towards a Performance Portable Multi-Grid Solver library for LQCD
  - Full Wilson-Clover Linear Operators & Basic Krylov Subspace solvers
  - Restriction and Prolongation Operators, Coarse Operator, Coarse Solvers
- Interface with Trilinos - Solvers Component Of USQCD ECP Project
  - Can we leverage Trilinos' solvers rather than rewriting my own?
- Work with Kokkos developers on areas of common interest
  - SIMD Types (my interest being specifically SIMD Complex)
  - Other index-traversal policies & layout combinations (e.g. cache oblivious)
  - Multi-node aspects: efficient halo exchanges for current & future hardware

Conclusions

- Excellent performance reached, rivaling or exceeding existing optimized libraries.
- Kokkos Parallel Pattern Constructs, Policies and Views
  - freed us from CUDA & OpenMP nuts and bolts, and index-order worries for the most part
  - allowed the main logic of the code to be portable
  - provided an easy to use efficient blocking construct (MDRange, md_parallel_for)
  - **did not get in the way of performance**
- For performance we still had to:
  - **implement performance-portable** (vectorization oriented) **algorithms** (MRHS and VSRHS)
  - **perform regular perf. optimization work** (use perf tools, rearrange memory access, etc.)
  - be aware of hardware/programming model issues (e.g. caches, CUDA launch-bounds, spills)
  - manually vectorize complex arithmetic (AVX512 on KNL, directly use threadIdx.x on GPU)
    - these generic features can be added to Kokkos (SIMD type and ThreadSIMDRange?)

Acknowledgments

- B. Joo acknowledges funding from the U.S. Department of Energy, Office of Science, Offices of Nuclear Physics, High Energy Physics and Advanced Scientific Computing Research under the SciDAC-3 program.
- B. Joo acknowledges funding from the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under the USQCD Exascale Computing Project.
- B. Joo acknowledges travel funding from NERSC for a summer Affiliate Appointment for work on Kokkos.
- The 2017 ORNL Hackathon at NASA was a collaboration between and used resources of both the National Aeronautics and Space Administration and the Oak Ridge Leadership Computing Facility at Oak Ridge National Laboratory. Oak Ridge Nation Laboratory is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.
- We gratefully acknowledge use of computer time at JeffersonLab (SciPhi XVI cluster), K80 Development node, NERSC Cori and SummitDev.

Compiler Setup

- Code at: https://github.com/JeffersonLab/mg_proto.git on the mdrange branch
  - The kokkos code is in the tests/kokkos directory. Right now all of MG proto needs to be built and QDP++ is a prerequisite for testing. Working on splitting this code out as a separate entity
- Cori KNL setup: Intel/2018.beta compiler,
  - CXXFLAGS="-g -O3 -std=c++11"
  - OMP_NUM_THREADS=272, OMP_PROC_BIND=spread, OMP_PLACES=threads
  - srun -n 1 -c 272 —cpu_bind=threads …
- SummitDev setup:
  - gcc-5.4.0, CUDA 8.0.54, nvcc_wrapper from Kokkos
  - CXXFLAGS="-g -O3 -std=c++11"
  - OMP_NUM_THREADS=10, OMP_PROC_BIND=spread, OMP_PLACES=threads
    - less relevant since these are single GPU jobs and performance is on the GPU