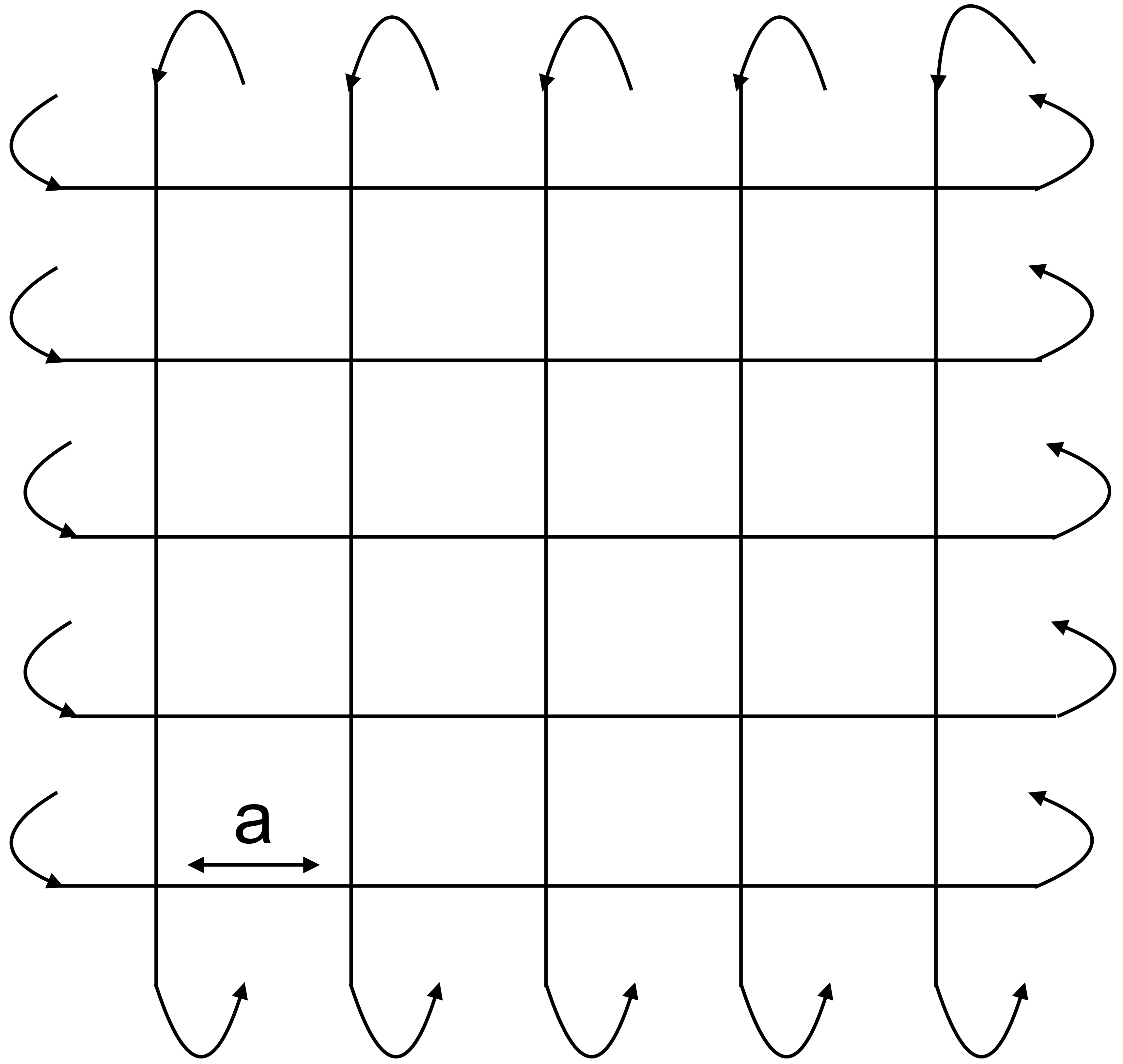
# Lattice QCD, Programming Models and Porting LQCD codes to Exascale

Bálint Joó - Jefferson Lab
Feb 19, 2020

HPC Roundtable

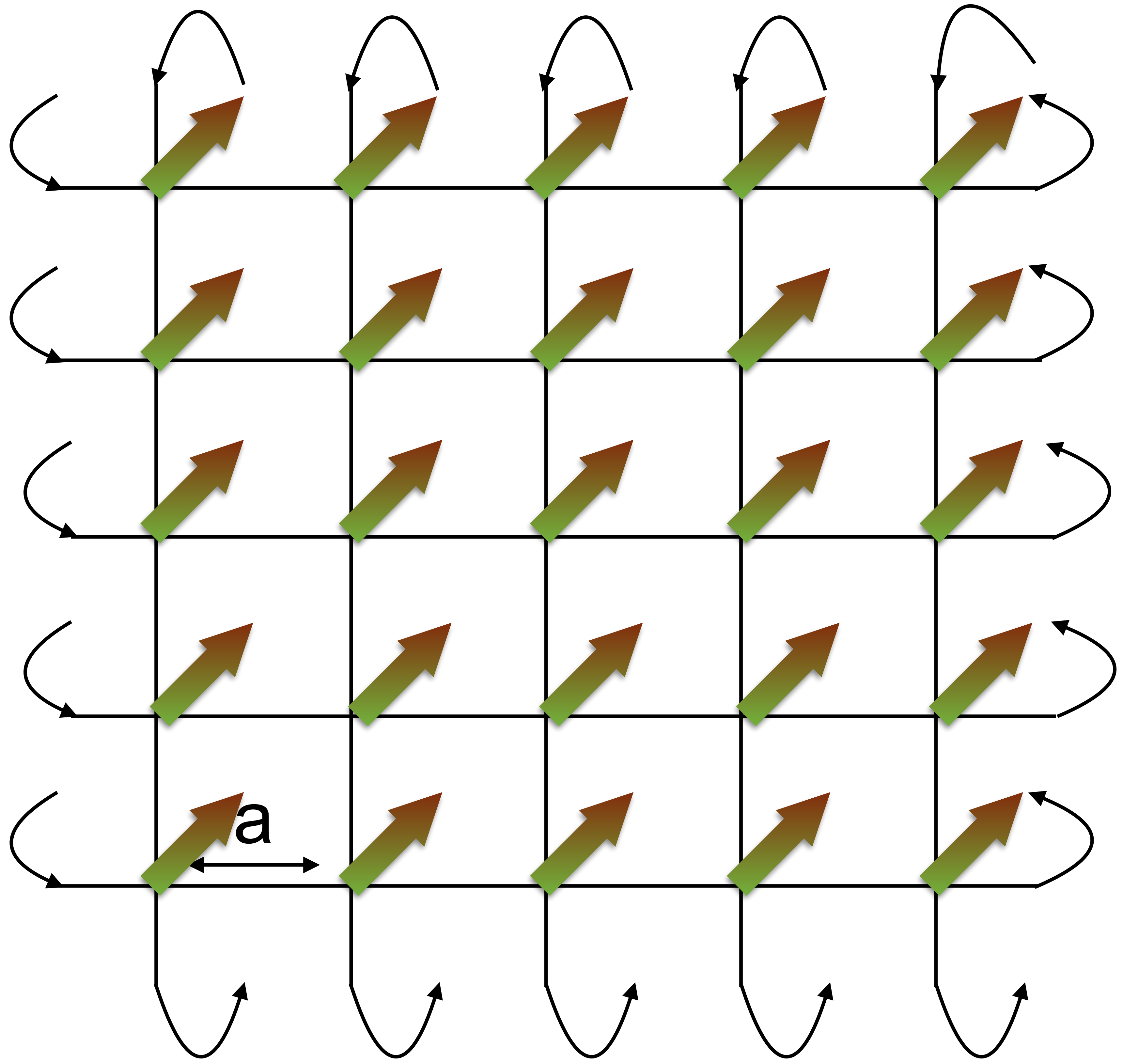
# LQCD as an application

- Replace Spacetime with a 4-Dimentional Lattice



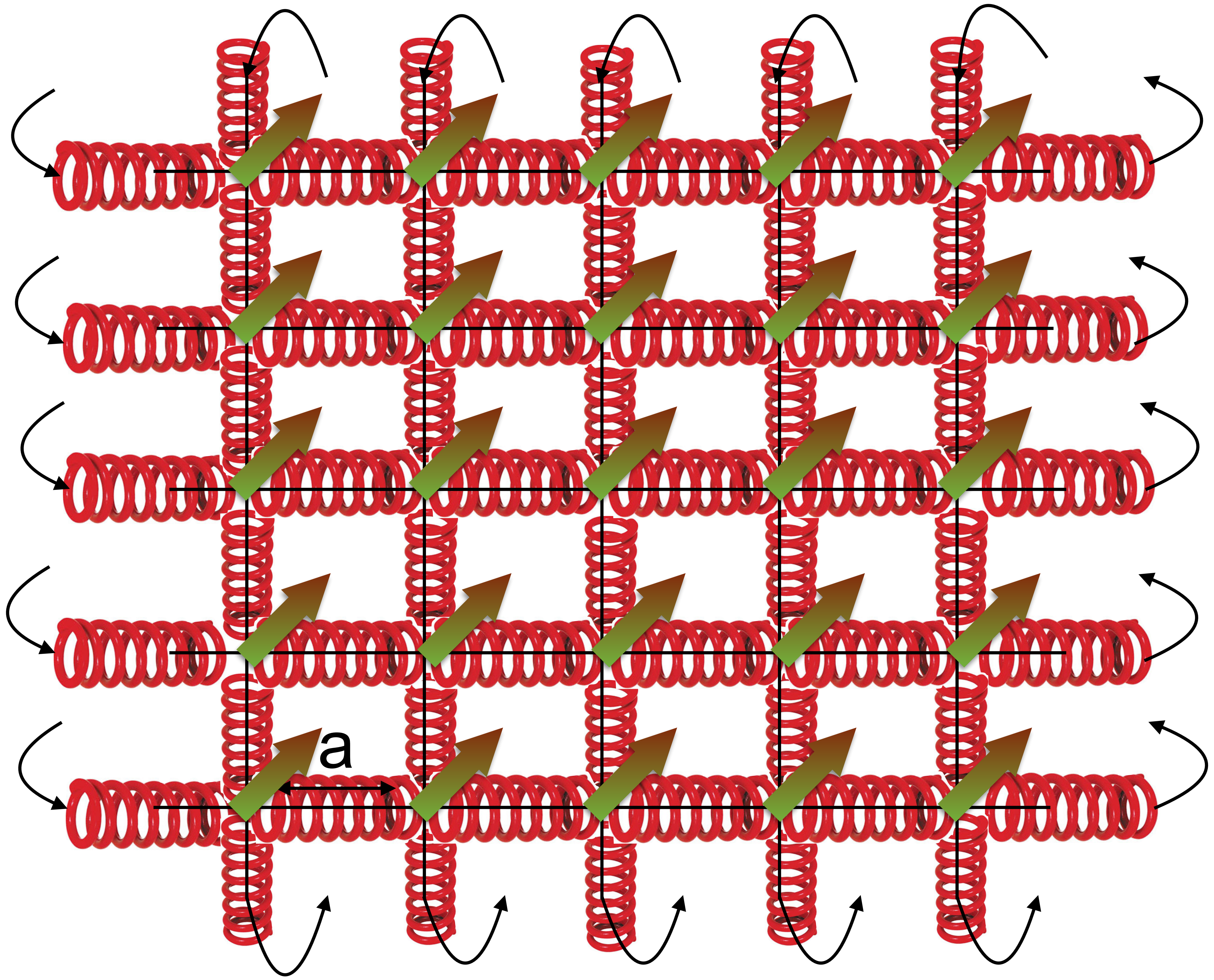
# LQCD as an application

- Replace Spacetime with a 4-Dimentional Lattice
- Quark fields on the lattice sites: spinors (either complex 3-vectors, or 4x3 “vectors”)



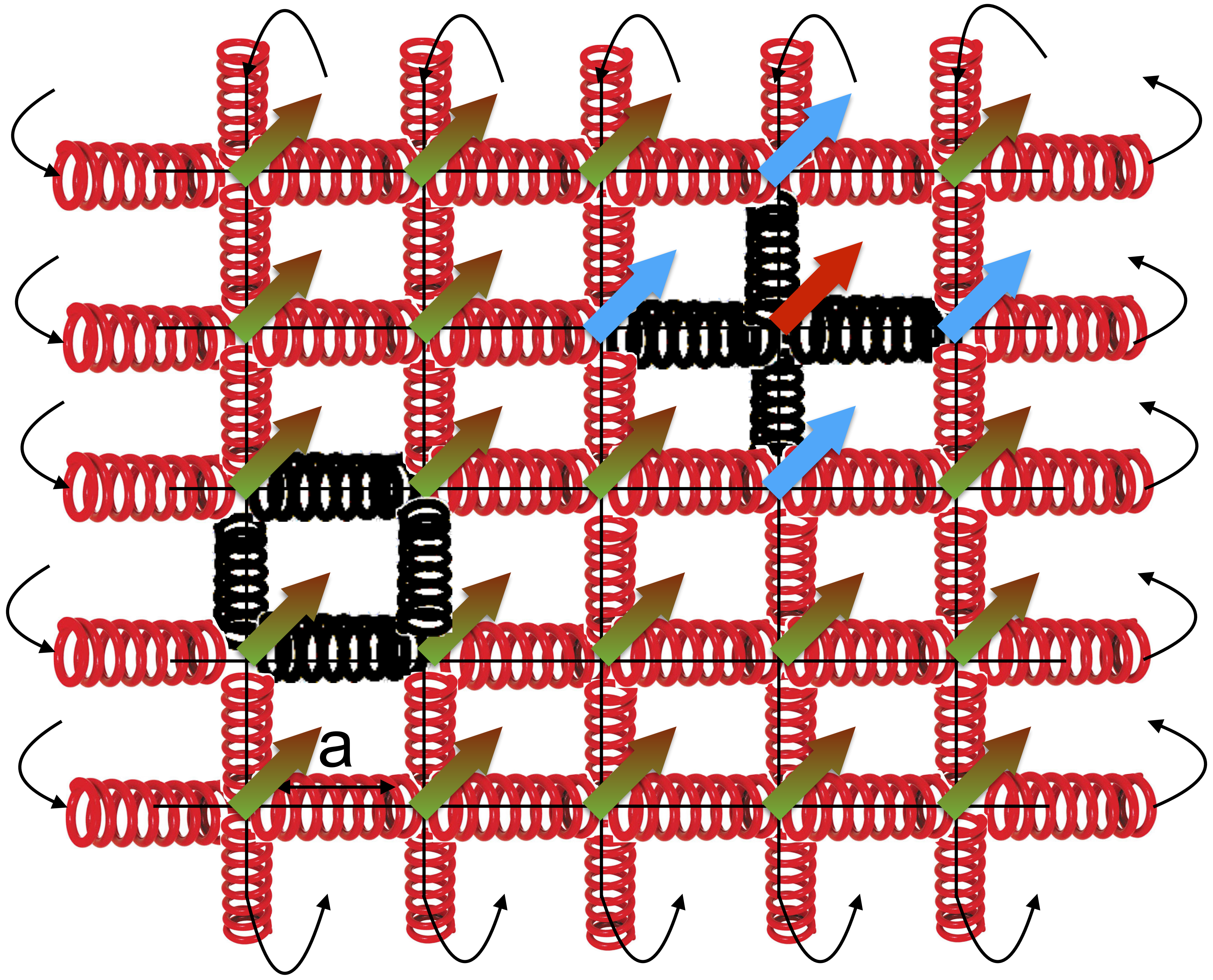
# LQCD as an application

- Replace Spacetime with a 4-Dimentional Lattice
- Quark fields on the lattice sites: spinors (either complex 3-vectors, or 4x3 “vectors”)
- Strong Force Gauge fields on links: 3x3 complex matrices

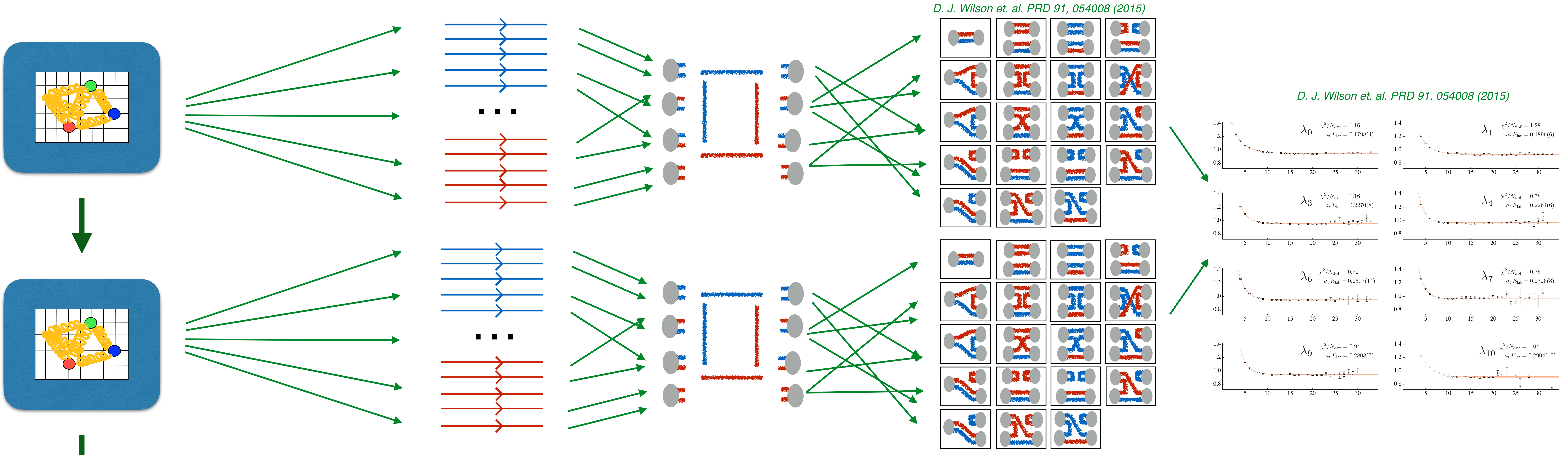


# LQCD as an application

- Replace Spacetime with a 4-Dimentional Lattice
- Quark fields on the lattice sites: spinors (either complex 3-vectors, or 4x3 “vectors”)
- Strong Force Gauge fields on links: 3x3 complex matrices
- Interactions are typically local
  - closed loops (3-matrix x 3-matrix)
  - covariant stencils (3-matrix x 3-vector )
- Also lattice wide summations:
  - global sums, inner products etc.
- Extremely well suited to **data-parallel** approaches
  - complex numbers and factors of 3 are often unfriendly to automatic vectorization - we need to usually build that in.



# Typical LQCD Workflow




***Configuration Generation***

- Hybrid Molecular Dynamics Monte Carlo
- Linear Solves for Fermion Forces
- Data parallel code for non-solver parts
- Strong Scaling Limited
- ‘Large’ long running jobs

***Propagators, graph nodes & edges eigenvectors etc.***

- Linear Solves for quark propagators on sources
- e.g. **O(1M) solves/config** for spectroscopy
- Solver: same matrix, many right hand sides
- Throughput limited
- Ensemble: Many small jobs

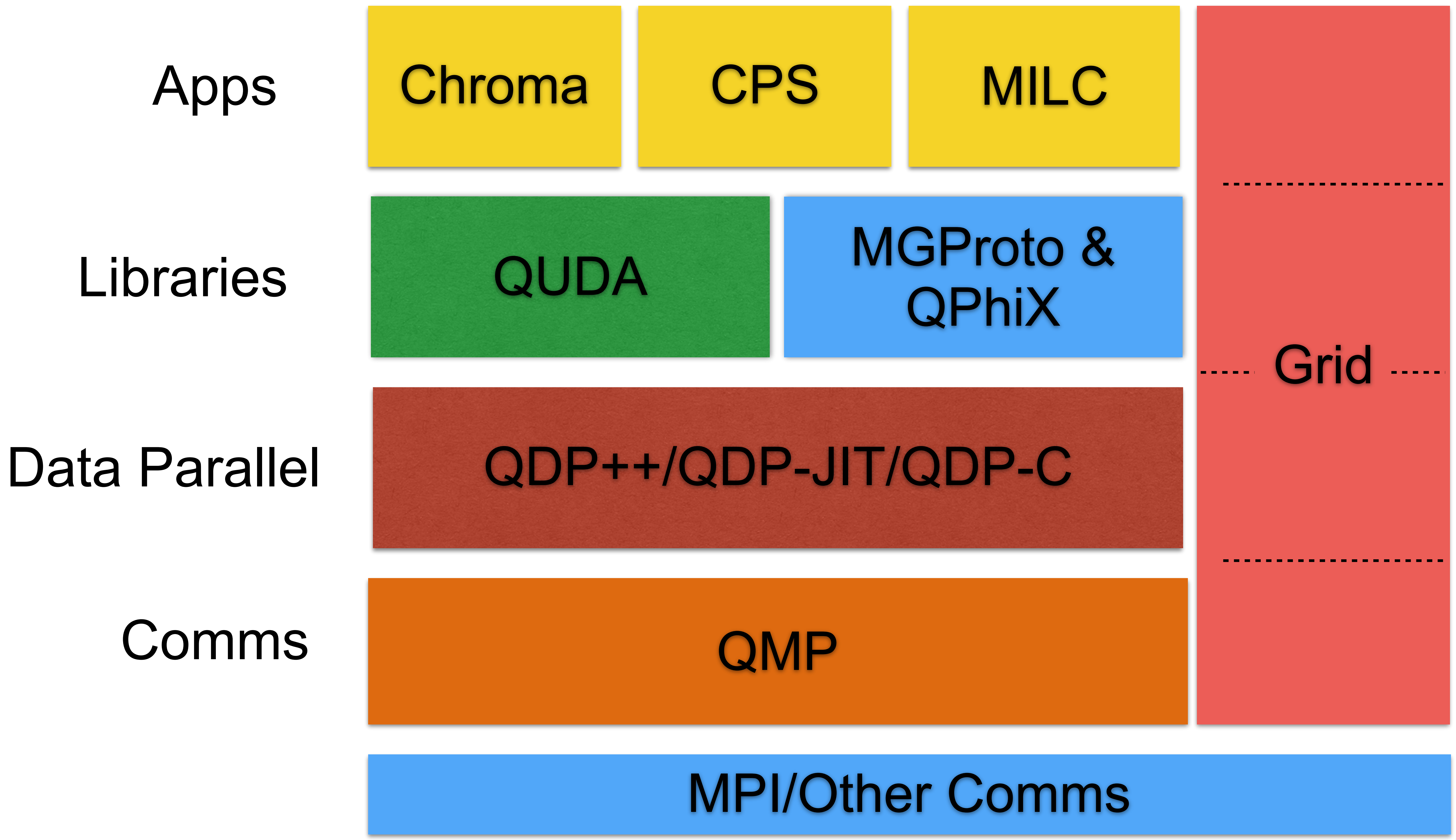
***Graph Contractions***

- O(10K)-O(100K) diagrams
- sub-diagram reuse challenge
- main operation is batched ZGEMM
- Potential large scale I/O challenge
- Ensemble: Many single node jobs

***Correlation Function Fitting and Analysis***

- workstations

# General Software Organization




- Level structure worked out over last 4 iterations of the SciDAC program
- Data Parallel Layer (QDP) over a communications abstraction layer, presents programmer with a 'virtual grid machine'
- Applications can be written on top of the Data Parallel Layer, calling out to Highly Optimized Libraries as needed.
- Grid is a new code, also providing a data parallel layer, and similar layering internally (but not broken out into separate packages)

# General Software Organization

Apps

Libraries

Data Parallel

Comms

Key Goals:
Port Data Parallel Layer,
Port Libraries,
Aim for Performance
Portability

ver last 4

ver a
layer,
'virtual

f the
out to
s

viding a
ar
roken out
into separate packages)

# Exascale & Pre-Exascale Systems

- Perlmutter (formerly NERSC-9)
  - AMD CPUs, NVIDIA Next Gen GPUs.
  - Slingshot fabric from Cray
- Aurora
  - Xeon CPUs + Intel Xe Accelerators
  - Slingshot fabric from Cray
- Frontier
  - AMD CPUs + AMD Radeon GPUs
  - Slingshot fabric from Cray
- MPI + X programming model
- Horsepower for all the systems will come from accelerators
- But the accelerators are different between the 3 systems



# Node Programming Model Options

| Support | OpenMP Offload | Kokkos/Raja | DPC++/SYCL | HIP | C++ pSTL | CUDA |
|---|---|---|---|---|---|---|
| **NVIDIA GPU** | Supported | Supported | Can be made to work | Supported | In development | Supported |
| **AMD GPU** | Supported | In development | Can be made to work | Supported | In development | Not supported |
| **Intel Xe** | Supported | In development | Supported | Not supported | In development | Not supported |
| **CPUs** | Supported | Supported | Supported | Not supported | In development | Not supported |
| **Fortran** | Supported | Not supported | Not supported | Can be made to work | Not supported | Can be made to work |
| **FPGAs** | In development | In development | Supported | Not supported | In development | Not supported |
| **Comments** | Compilers Maturing, some C++ issues | DPC++ and HIP back ends in development | NVIDIA via POCL or Codeplay Backend, AMD via hipSYCL for now, well supported for Intel | Fortran via cross calling, well supported for AMD GPUs | The way of the future? parallelism in the base language. Tech previews just now | Fortran via PGI CUDA Fortran, well supported for NVIDIA GPUs |

Supported

In development or aspirational

Can be made to work via 3rd party extension or product or hack

Not supported

Disclaimer: this is my current view, products and support levels can change. This picture may become out of date very soon

# OpenMP Offload

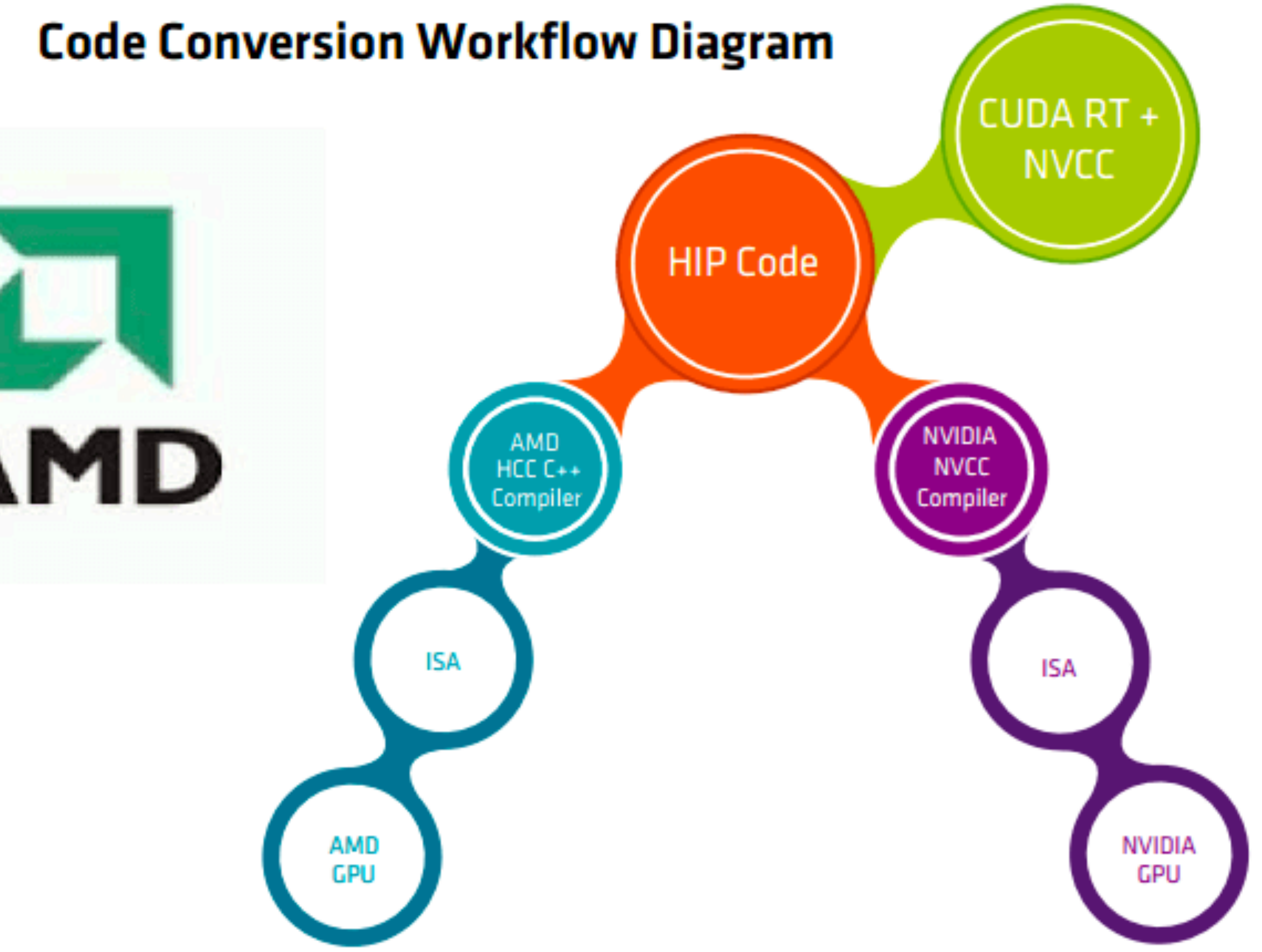
- Offloaded axpy in OpenMP

```
#pragma omp target teams distribute parallel for simd
   map(to:z[:N]) map(a,x[:N],y[:N])
for(int i=0; i < N; i++) // N is large
{
   z[i] = a*x[i] + b[i];
}
```

- Collapses:
  - omp target - target the accelerator,
  - omp teams - create a league of teams
  - omp distribute - distribute the works amongst the teams
  - omp parallel for simd - perform a SIMD-ized parallel for
  - map a, x and y to the accelerator and map resulting z back out (data movement).

# HIP

- HIP is AMD's "C++ Heterogeneous-Compute Interface for Portability"
- Take your CUDA API and replace 'cuda' with 'hip':
  - `cudaMemcpy()` -> `hipMemcpy()`
  - `kernel<<<>>>( )` -> `hipLaunchKernelGGL(kernel,…)`
  - and other slight changes.
  - You can use ***hipify*** tool to do first pass of conversion automatically
- Open Source
- Portability between NVIDIA and AMD GPUs only.



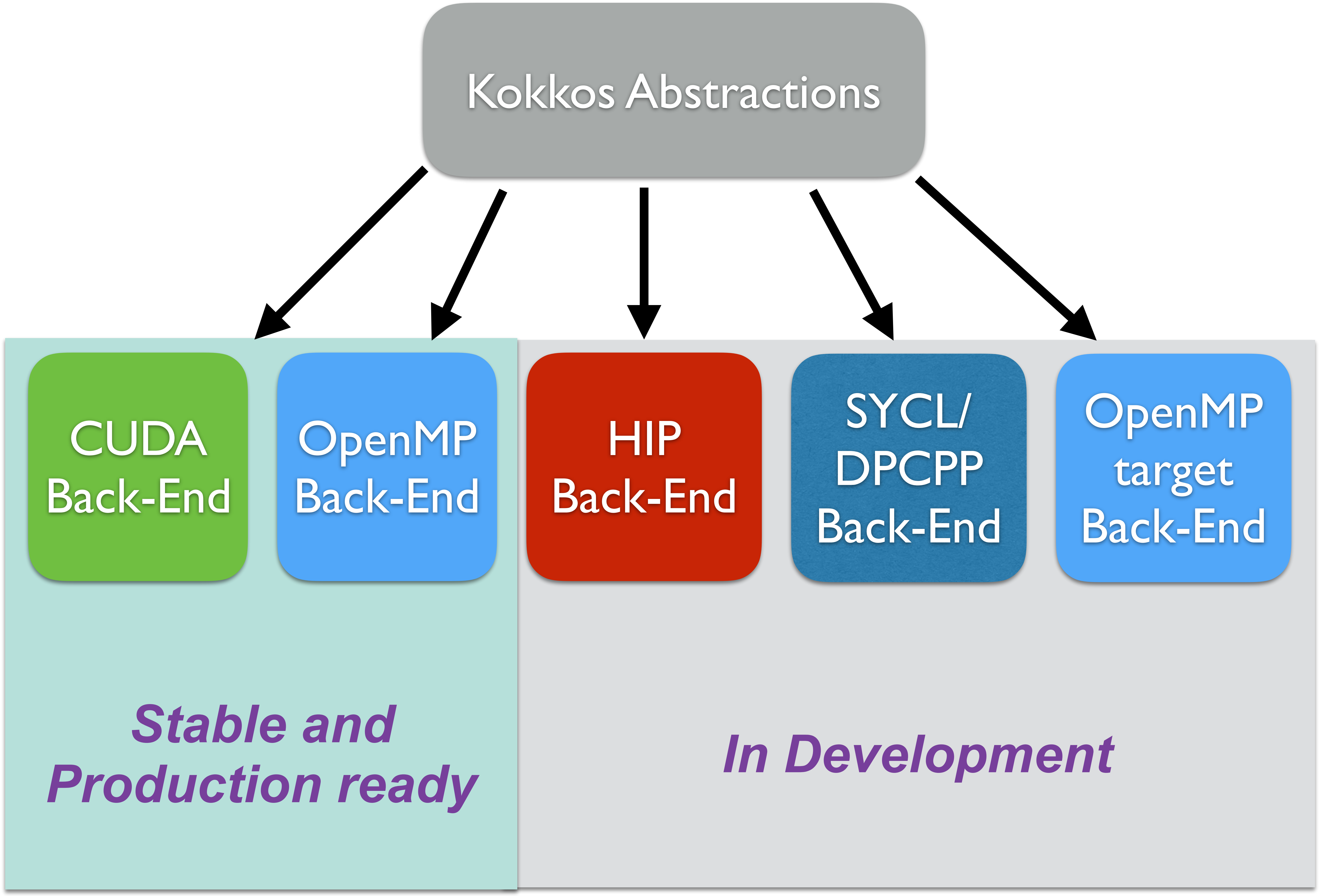
# Kokkos

```
Kokkos::View<float[N],LayoutLeft,CudaSpace> x(“x”); // N is large
Kokkos::View<float[N],LayoutLeft,CudaSpace> y(“y”);
Kokkos::View<float[N],LayoutLeft,CudaSpace> z(“z”);

float a=0.5;

Kokkos::parallel_for(“zaxpy", N, KOKKOS_LAMBDA (const int& i) {
  z(i) = a*x(i) + y(i);   // view provides indexing operator()
});
```

- View - multi-dimensional array, index order specified by Layout, location by MemorySpace policy. Layout allows appropriate memory access for CPU/GPU
- Parallel for dispatches a C++ lambda
- Kokkos developers on C++ standards committee - work to fold features into C++

# Portability via Kokkos

- Kokkos provides portability via back-ends: e.g. OpenMP, CUDA, …
- Most abstractions are provided in a C++ Header library
  - parallel_for, reduction, scans
- Kokkos provides the Kokkos View data-type
  - user can customize index order
  - explicit memory movement only
  - select memory space via policy
- Bind Execution to Execution Space
  - select back end via policy



# SYCL

- SYCL manages buffers
- Only access buffers via accessors
- can track accessor use and build data dependency graph to automate data movement
- What does this mean for non SyCL Libraries with pointers? (e.g. MPI)

```
sycl::queue myQueue;
sycl::buffer<float,1> x_buf(LARGE_N);
sycl::buffer<float,1> y_buf(LARGE_N);
sycl::buffer<float,1> z_buf(LARGE_N);

// … fill buffers somehow …
float a = 0.5;
{
  myQueue.submit([&](handler& cgh) {
    auto x=x_buf.getAccess<access::mode::read>(cgh);
    auto y=y_buf.getAccess<access::mode::read>(cgh);
    auto z=z_buf.getAccess<access::mode::write>(cgh);

    cgh.parallel_for<class zaxpy>(LARGE_N,[=](id<1> id){
      auto i = id[0];
      z[i]=a*x[i] + y[i];
    });
  });
}
```

SYCL runtime manages data in buffers

access buffer data via accessors in command group (cgh) scope or host accessor

kernels must have a unique name in C++

# Intel OneAPI DPC++ extensions

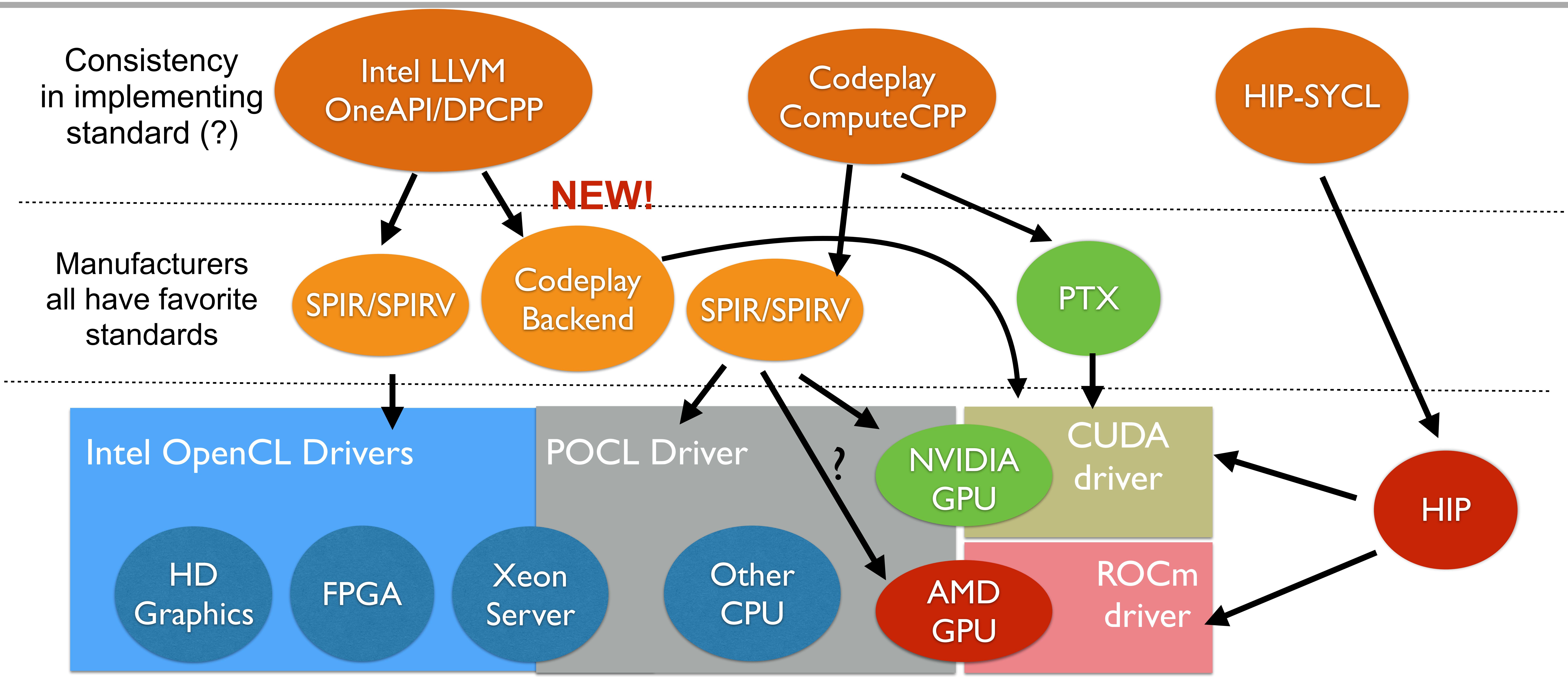
- USM extension allows management of arrays via pointers (more CUDA-like)
- Memcpy ops to move data between host and device (not shown here)
- Reductions !!
- Unnamed Lambda extension obviates need for a class name for parallel for
- Libraries (e.g. MPI) can do intelligent things with USM pointers (e.g. direct device access)
- Subgroup Extension allows more explicit SIMD-ization

```
sycl::queue myQueue;
sycl::device dev=myQueue.get_device();
sycl::context con=myQueue.get_context();

float* x=sycl::malloc_device(LARGE_N*sizeof(float),dev,con);
float* y=sycl::malloc_device(LARGE_N*sizeof(float),dev,con);
float* z=sycl::malloc_device(LARGE_N*sizeof(float),dev,con);

// … fill aarrays somehow  somehow …
float a = 0.5;
{
  myQueue.submit([&](handler& cgh) {
   cgh.parallel_for(LARGE_N,[=](id<1> id){
      auto i = id[0];
      z[i]=a*x[i] + y[i];
   });
  });
}
// free pointers etc..
```

USM gives host/ device pointers and

Unnamed lambda extension

# Portability via SYCL



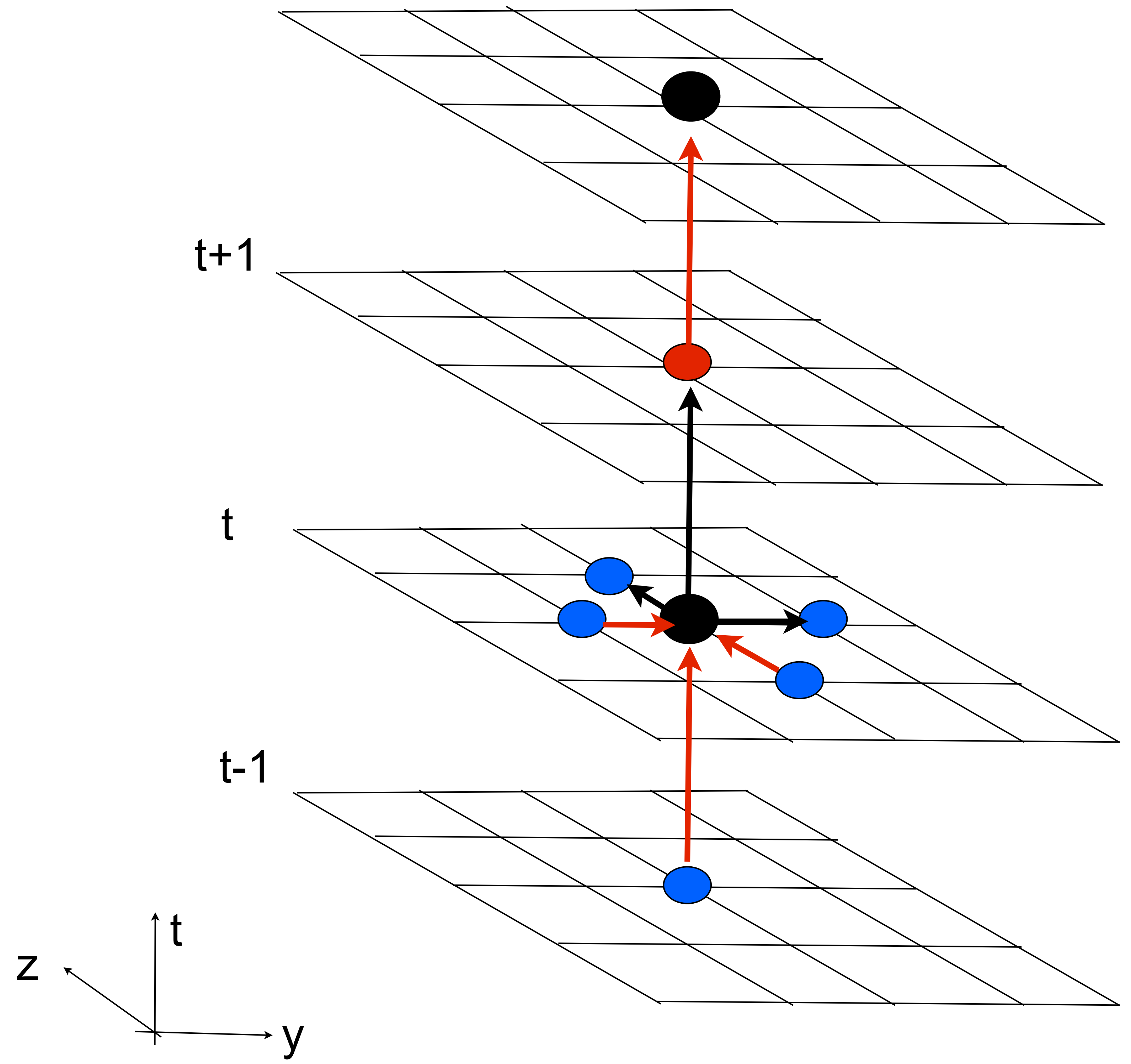
# US LQCD Codes are C++/C

- For C/C++ codes, OpenMP offload, Kokkos/Raja, or DPC++ and SYCL are the most obvious candidates currently. pSTL may become interesting in the near future
- Performance Portability Experiments:
  - OpenMP Offload: P. A.Boyle, K. Clark, C. DeTar, M. Lin, V. Rana, A. V. Aviles-Castro, “Performance Portability Strategies for Grid C++ expression templates” arxiv:1710.09409
  - OpenMP Offload:  P. Steinbrecher and HotQCD - OpenMP implementation for Intel Gen9
  - Kokkos and SYCL: B. Joo, P3HPC @ SC19
  - Early pSTL experiments by K. Clark
- The lattice developer community is paying attention to DPC++/SYCL, HIP, and OpenMP offload as the porting work to the new machines becomes more urgent.
- I will focus on our local work with the Chroma code and Kokkos and SYCL

# Wilson Dslash in Kokkos and SYCL

- When looking at a new programming model, it helps to have a “simple” mini-app to evaluate whether the model is viable
- We chose the Wilson-Dslash operator as it is
  - sufficiently nontrivial.
  - well understood in terms of performance
  - has many hand optimized implementations, e.g. QPhiX on KNL, QUDA on NVIDIA GPUs
- Initial work in Kokkos looked at vectorization
- More recently we looked at porting to SYCL, and seeing how portable SYCL is

$$D_{x,y} = \sum_{\mu} \left[ (1-\gamma_\mu) U_{x,\mu} \delta_{x+\hat{\mu},y} + (1+\gamma_\mu) U^\dagger_{x-\hat{\mu},\mu} \delta_{x-\hat{\mu},y} \right]$$



# Basic Performance Bound for Dslash

- R = no of reused input spinors
- Br = read bandwidth
- Bw = write bandwidth
- G = size of Gauge Link matrix (bytes)
- S = size of Spinor (bytes)
- r = 1 (read-for-write), =0 (no read-for-write)
- Simplify: Assume Br = Bw = B

$$F = \frac{1320}{8G/B_r + (8 - R + r)S/B_r + S/B_w}$$

$$\mathrm{AI} = \frac{1320}{8G + (9 - R + r)S}$$

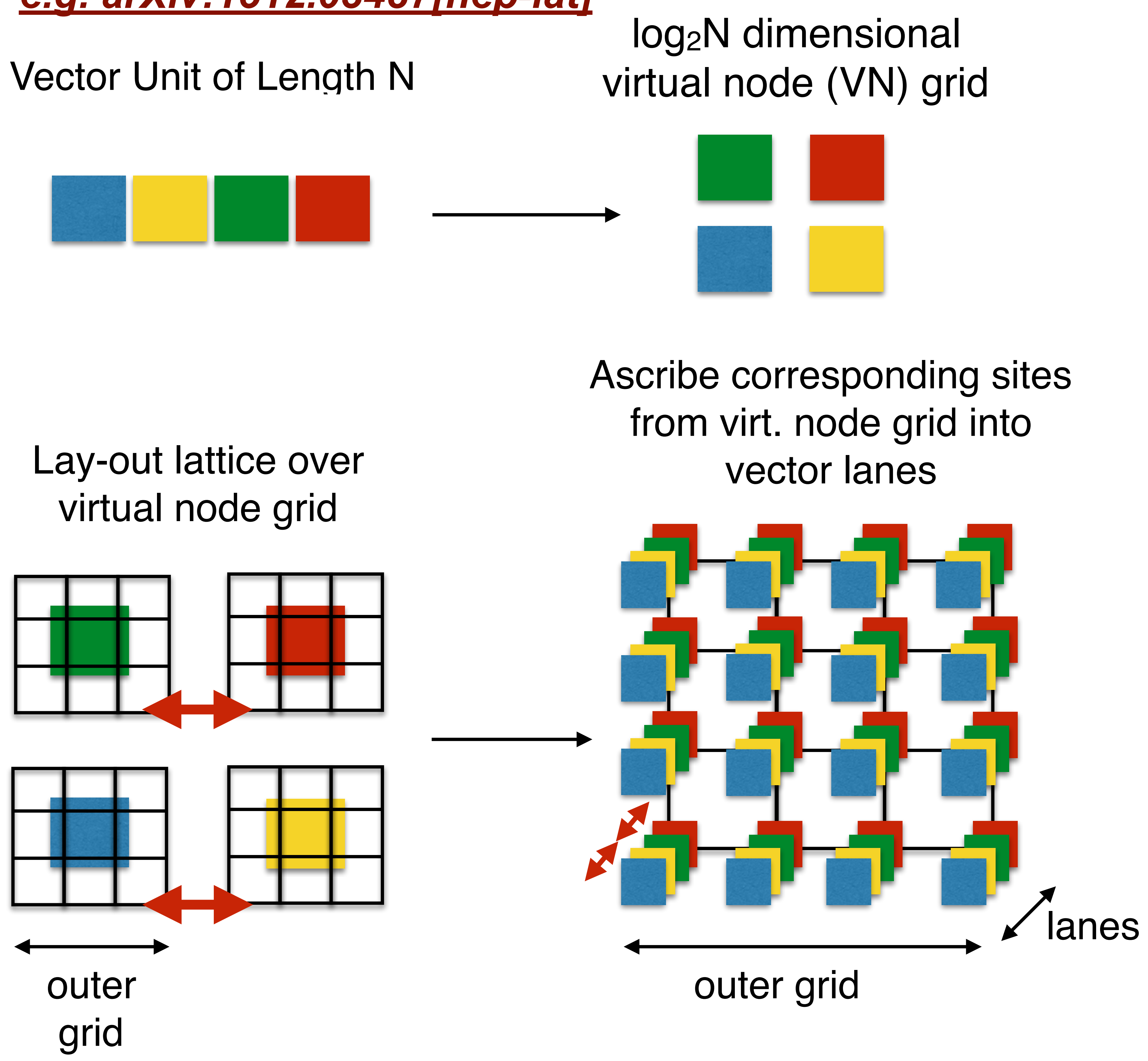
***Wilson Dslash Arithmetic Intensities (F/B) for 32-bit floating point numbers (G=72B, S=96B)***

| | R=0 | R=1 | R=2 | R=3 | R=4 | R=5 | R=6 | R=7 |
|---|---|---|---|---|---|---|---|---|
| **r=0** | 0.92 | 0.98 | 1.06 | 1.15 | 1.25 | 1.38 | 1.53 | 1.72 |
| **r=1** | 0.86 | 0.92 | 0.98 | 1.06 | 1.15 | .1.25 | 1.38 | 1.53 |

# Vectorizing Dslash for Single RHS

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

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




- Treat SIMD lanes like a grid of virtual computing elements (virtual nodes, VNs)
- Lay-out lattice onto VN grid
  - original site -> ( 'outer' site, lane )
- All arithmetic changes to straightforward SIMD arithmetic
- Accessing nearest neighbors
  - on edge of `outer lattice` communicate between 'virtual nodes' (lanes).
  - this is a shuffle operations (e.g. _mm512_shuffle_ps in AVX512 )
- On GPUs
  - use N=1 (no vectorization) => trivial shuffles. ✅
  - Or use warp/subgroup level SIMD (less portable) ×

# Kokkos Implementation: Kernel

```
template<typename VN, typename GT, typename ST, typename TGT, typename TST, const int isign, const int target_cb>
struct VDslashFunctor {

  VSpinorView<ST,VN> s_in;
  VGaugeView<GT,VN> g_in;
  VSpinorView<ST,VN> s_out;
  SiteTable<VN> neigh_table;

  KOKKOS_FORCEINLINE_FUNCTION
 void operator()(const int& xcb, const int& y, const int& z, const int& t) const
 {
   int site = neigh_table.coords_to_idx(xcb,y,z,t);
   int n_idx;

   typename VN::MaskType mask;
   SpinorSiteView<TST> res_sum ;
   HalfSpinorSiteView<TST> proj_res , mult_proj_res;

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

   neigh_table.NeighborTMinus(xcb,y,z,t,n_idx,mask);                  // Get neighbor and permutation mask
   KokkosProjectDir3Perm<ST,VN,TST,isign>(s_in, proj_res,n_idx,mask);  // spin project
   mult_adj_u_halfspinor<GT,VN,TST,0>(g_in, proj_res,mult_proj_res,site);  // matrix multiply (neighbor matrix permuted already)
   KokkosRecons23Dir3<TST,VN,isign>(mult_proj_res,res_sum);           // reconstruct

   // Other dirs. (Z-, Y-, X-, X+, Y+, Z+, T+
   #pragma unroll
   for(int spin=0; spin < 4; ++spin)
     for(int color=0; color < 3; ++color) {
        Stream(s_out(site,spin,color),res_sum(color,spin));
 }};
```

operator() gets 4 indices from the multi dimensional range policy

Neighbouring site

Vectorisation Permutation mask: for edges

# Kokkos Implementation: Dispatch

```
template<typename VN, typename GT, typename ST,  typename TGT, typename TST>
class KokkosVDslash {
public:
  const LatticeInfo& _info;
  SiteTable<VN> _neigh_table;

  KokkosVDslash(const LatticeInfo& info) : _info(info),
    _neigh_table(info.GetCBLatticeDimensions()[0],info.GetCBLatticeDimensions()[1],info.GetCBLatticeDimensions()[2],info.GetCBLatticeDimensions()[3]) {}

  void operator()(const KokkosCBFineVSpinor<ST,VN,4>& fine_in, const KokkosCBFineVGaugeFieldDoubleCopy<GT,VN>& gauge_in,
                  KokkosCBFineVSpinor<ST,VN,4>& fine_out, int plus_minus, const IndexArray& blocks) const
  {
   int source_cb = fine_in.GetCB();
   int target_cb = (source_cb == EVEN) ? ODD : EVEN;

   const VSpinorView<ST,VN>& s_in = fine_in.GetData();
   const VGaugeView<GT,VN>& g_in = gauge_in.GetData();
    VSpinorView<ST,VN>& s_out = fine_out.GetData();

   IndexArray cb_latdims = _info.GetCBLatticeDimensions();


    MDPolicy policy({0,0,0,0}, {cb_latdims[0],cb_latdims[1],cb_latdims[2],cb_latdims[3]}, {blocks[0],blocks[1],blocks[2],blocks[3]});

    if( plus_minus == 1 ) {
     if (target_cb == 0 ) {
       VDslashFunctor<VN,GT,ST,TGT,TST,1,0> f = {s_in, g_in, s_out, _neigh_table}; // Instantiate functor: set fields
       Kokkos::parallel_for(policy, f);                                            // Dispatch
     }
     else {
      …
     }    }}};
```

4D Blocked Lattice Traversal Dispatch

# SYCL Kernel Dispatch

```
template<typename VN, typename GT, typename ST, int dir, int cb>. class dslash_loop;   // Just to give SyCL Kernel a name; Yuck!

template<typename VN, typename GT, typename ST>
class SyCLVDslash {
  const LatticeInfo& _info;
  SiteTable _neigh_table;
public:
  SyCLVDslash(const LatticeInfo& info) : _info(info),
   _neigh_table(info.GetCBLatticeDimensions()[0],info.GetCBLatticeDimensions()[1],info.GetCBLatticeDimensions()[2],info.GetCBLatticeDimensions()
[3]) {}

  void operator()(const SyCLCBFineVSpinor<ST,VN,4>& fine_in, const SyCLCBFineVGaugeFieldDoubleCopy<GT,VN>& gauge_in,
                        SyCLCBFineVSpinor<ST,VN,4>& fine_out, int plus_minus)

  {
      int source_cb = fine_in.GetCB();  int target_cb = (source_cb == EVEN) ? ODD : EVEN;
      SyCLVSpinorView<ST,VN> s_in = fine_in.GetData();
      SyCLVGaugeView<GT,VN> g_in = gauge_in.GetData();
      SyCLVSpinorView<ST,VN> s_out = fine_out.GetData();
      IndexArray cb_latdims = _info.GetCBLatticeDimensions();
      int num_sites = fine_in.GetInfo().GetNumCBSites();

      cl::sycl::queue q;
      if( plus_minus == 1 ) {
        if (target_cb == 0 ) {
          q.submit( [&](cl::sycl::handler& cgh) {
         VDslashFunctor<VN,GT,ST,1,0> f{
                      s_in.template get_access<cl::sycl::access::mode::read>(cgh),
                      g_in.template get_access<cl::sycl::access::mode::read>(cgh),
                      s_out.template get_access<cl::sycl::access::mode::write>(cgh),
                      _neigh_table.template get_access<cl::sycl::access::mode::read>(cgh)
               };                                                                                 // Setup Functor

               cgh.parallel_for<dslash_loop<VN,GT,ST,1,0>>(cl::sycl::range<1>(num_sites), f);
           });
        }
        else {
```

Ugly: Need a ‘typename’ for dispatches, unless you have Intel -funnamed-lambda extension

Get Views our of user data types

Pass ViewAccessors to functor

1D Dispatch for now

# SYCL Kernel Dispatch

```
template<typename VN, typename GT, typename ST, int dir, int cb>. class dslash_loop;   // Just to give SyCL Kernel a name; Yuck!

template<typename VN, typename GT, typename ST>
class SyCLVDslash {
  const LatticeInfo& _info;
  SiteTable _neigh_table;
public:
  SyCLVDslash(const LatticeInfo& info) : _info(info),
   _neigh_table(info.GetCBLatticeDimensions()[0],info.GetCBLatticeDimensions()[1],info.GetCBLatticeDimensions()[2],info.GetCBLatticeDimensions()
[3]) {}

  void operator()(const SyCLCBFineVSpinor<ST,VN,4>
                        SyCLCBFineVSpinor<ST,VN,4>

  {
      int source_cb = fine_in.GetCB();  int target
      SyCLVSpinorView<ST,VN> s_in = fine_in.GetDat
      SyCLVGaugeView<GT,VN> g_in = gauge_in.GetDat
      SyCLVSpinorView<ST,VN> s_out = fine_out.GetD
      IndexArray cb_latdims = _info.GetCBLatticeDi
      int num_sites = fine_in.GetInfo().GetNumCBSi

      cl::sycl::queue q;
      if( plus_minus == 1 ) {
        if (target_cb == 0 ) {
          q.submit( [&](cl::sycl::handler& cgh) {
         VDslashFunctor<VN,GT,ST,1,0> f{
                    s_in.template get_access<cl:
                    g_in.template get_access<cl::sycl::access::mode::read>(cgh),
                    s_out.template get_access<cl::sycl::access::mode::write>(cgh),
                    _neigh_table.template get_access<cl::sycl::access::mode::read>(cgh)
              };                                                                         // Setup Functor

              cgh.parallel_for<dslash_loop<VN,GT,ST,1,0>>(cl::sycl::range<1>(num_sites), f);      // Dispatch (1D for now)
          });
        }
        else {
```

Ugly: Need a 'typename' for dispatches, unless you have Intel -funnamed-lambda extension

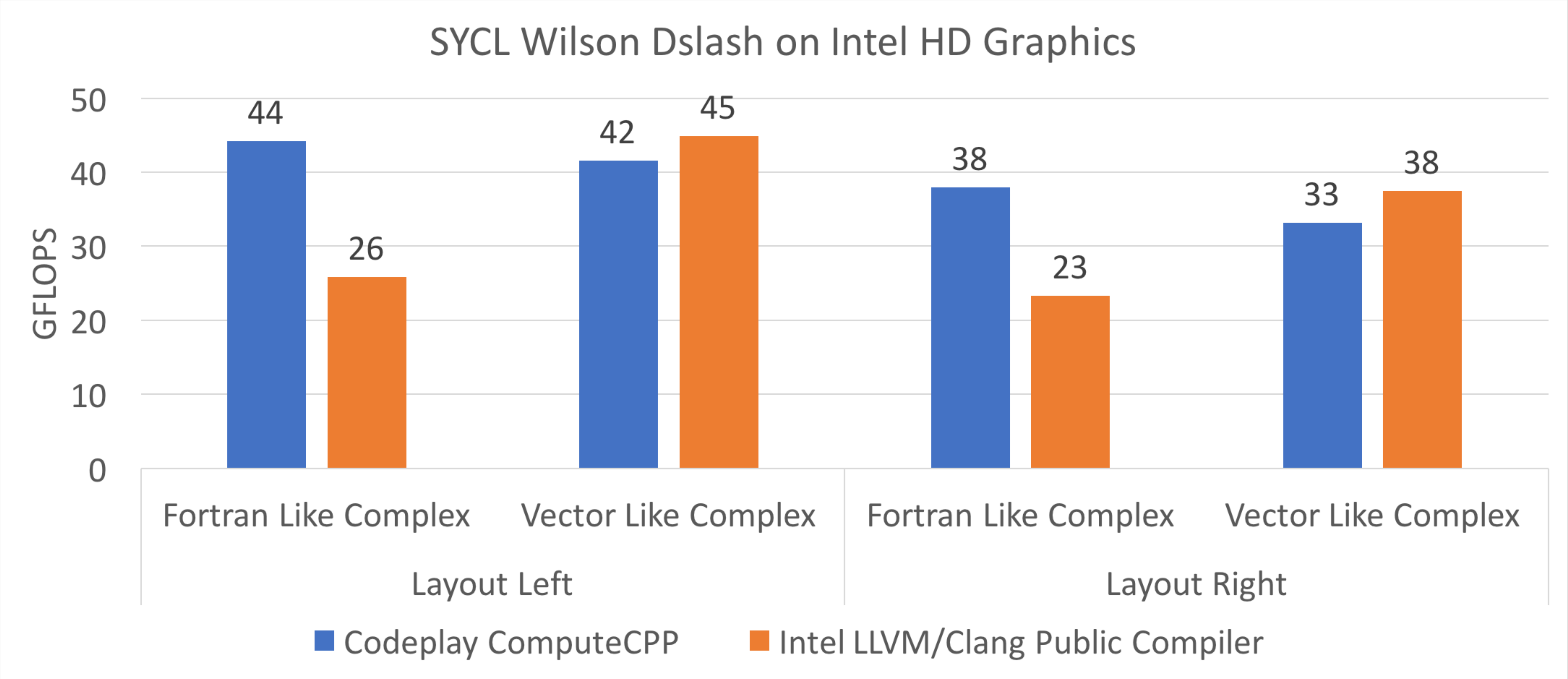
Future: instead of accessors use USM pointers, or Views implemented using USM pointers

s to functor

# Experiments & Standard Candles

- We measured the performance of Kokkos & SYCL Dslash kernels on
  - Volta V100 GPUs. using Cori GPU system at NERSC
  - Skylake CPUs (single socket) using the CPUs on Cori GPU system at NERSC
  - KNL Systems using Jefferson Lab 18p cluster nodes
  - Gen9 GPU using an Intel NUC System
- Performance 'Standard Candles'
  - On GPU:  Dslash from QUDA Library, with equivalent compression/precision options
    - Highly optimized QCD library for GPUs, M. A. Clark et. al. Comput Phys. Commun. 181, 1517 (2010) [arXiv:0911.3191 [hep-lat],  Download via: http://lattice.github.io/quda/
  - On CPU/KNL: Dslash from QPhiX Library with equivalend compression/precision options
    - Joo et. al. Kunkel J.M., Ludwig T., Meuer H.W. (eds) Supercomputing. ISC 2013. Lecture Notes in Computer Science, vol 7905. Springer, Berlin, Heidelberg, https://github.com/jeffersonlab/qphix
- To use SYCL on KNL and GPUs we used POCL v1.8: http://portablecl.org/

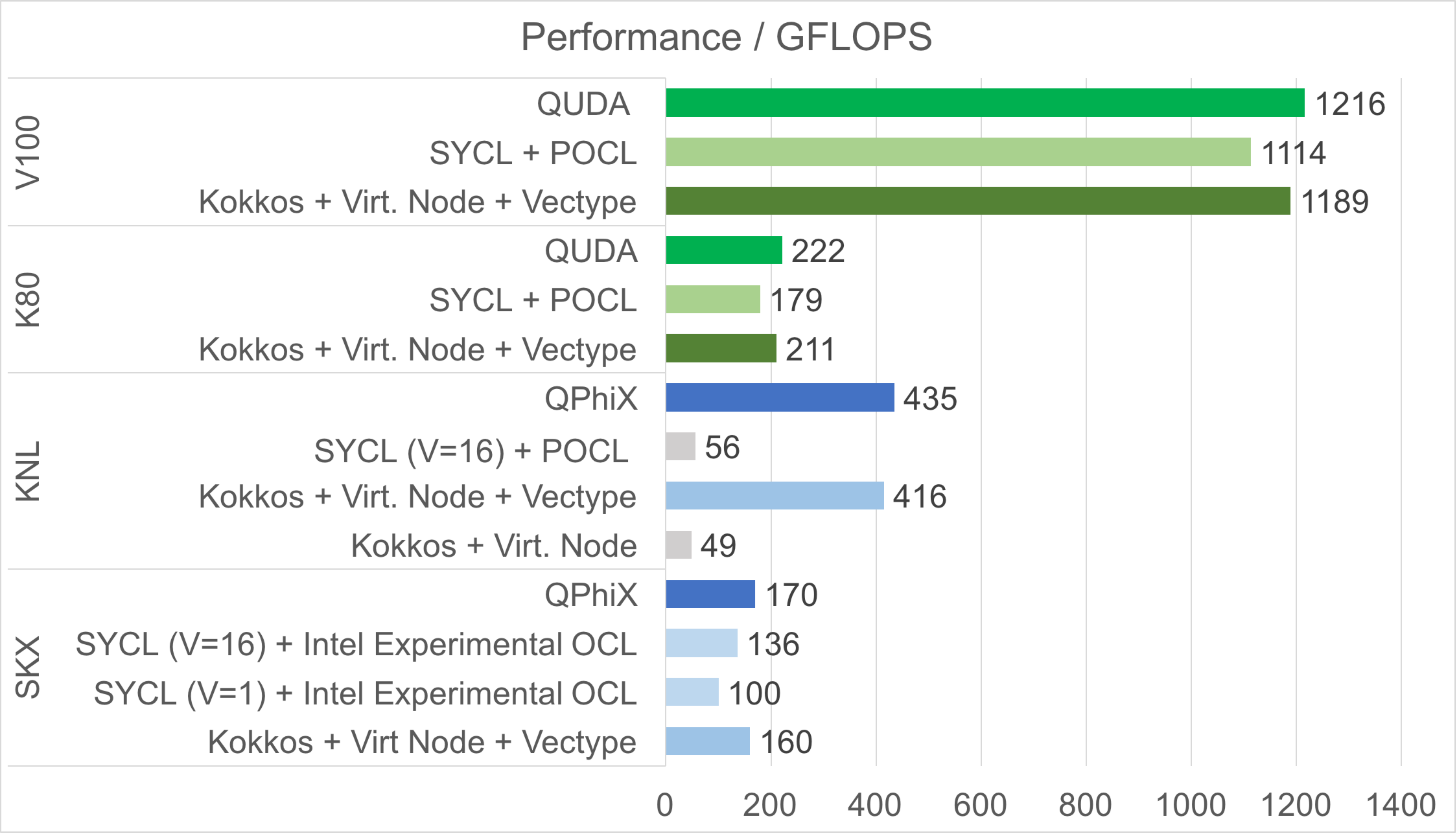
# SYCL on Intel HD Graphics




- Gen-9 GPU in a NUC (max DRAM bandwidth ~ 38 GB/sec, lattice had $32^4$ sites
- Used Codeplay Community Edition (1.0.4 Ubuntu) and Intel Public LLVM-based SYCL Compiler (version in the paper).
- Fortran like complex: (RIRIRI…), Vector Like complex: (RRRR…IIII…).
  - since V=1 these are the same layout but different operations
- Best performance: sustain 32-36 GB/sec, ~45 GFLOPS => AI ~ 1.25 => R=4-5.

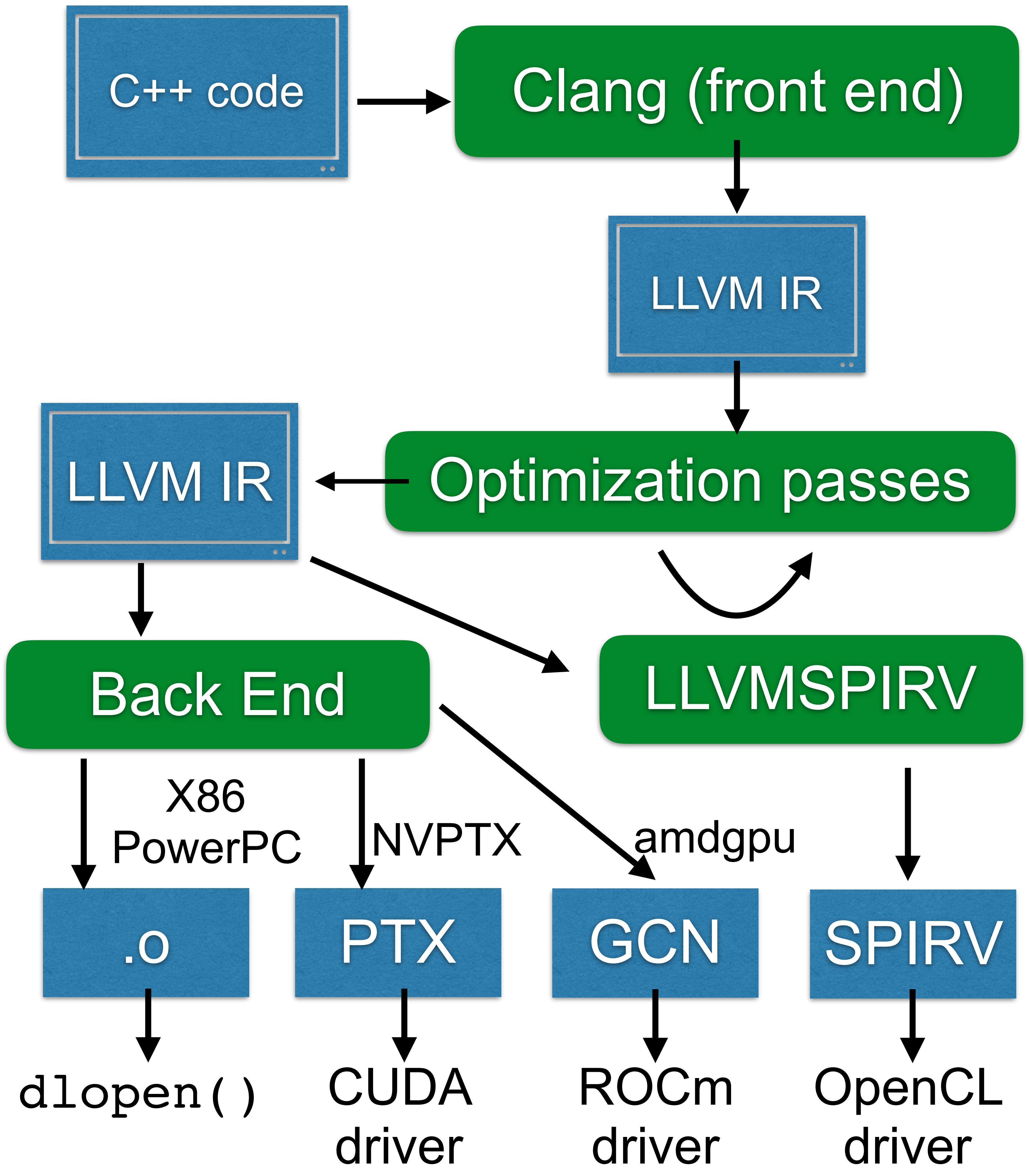
# Combined Single RHS Results

- Kokkos using the virtual node SIMD with a 'Vector Type' seems to work well
  - 'Vectype' is AVX512 or our complex type based on float2
  - Kokkos::complex with 'alignas' keyword works as well as float2
- SYCL + POCL did well on GPUs (had linear lattice traversal, if we implemented 4D it may be on par with Kokkos & QUDA - future work)
- Kokkos without Vectype did not do well on KNL - we anticipate the compiler doesn't do well with SIMD-izing complex operations(?)



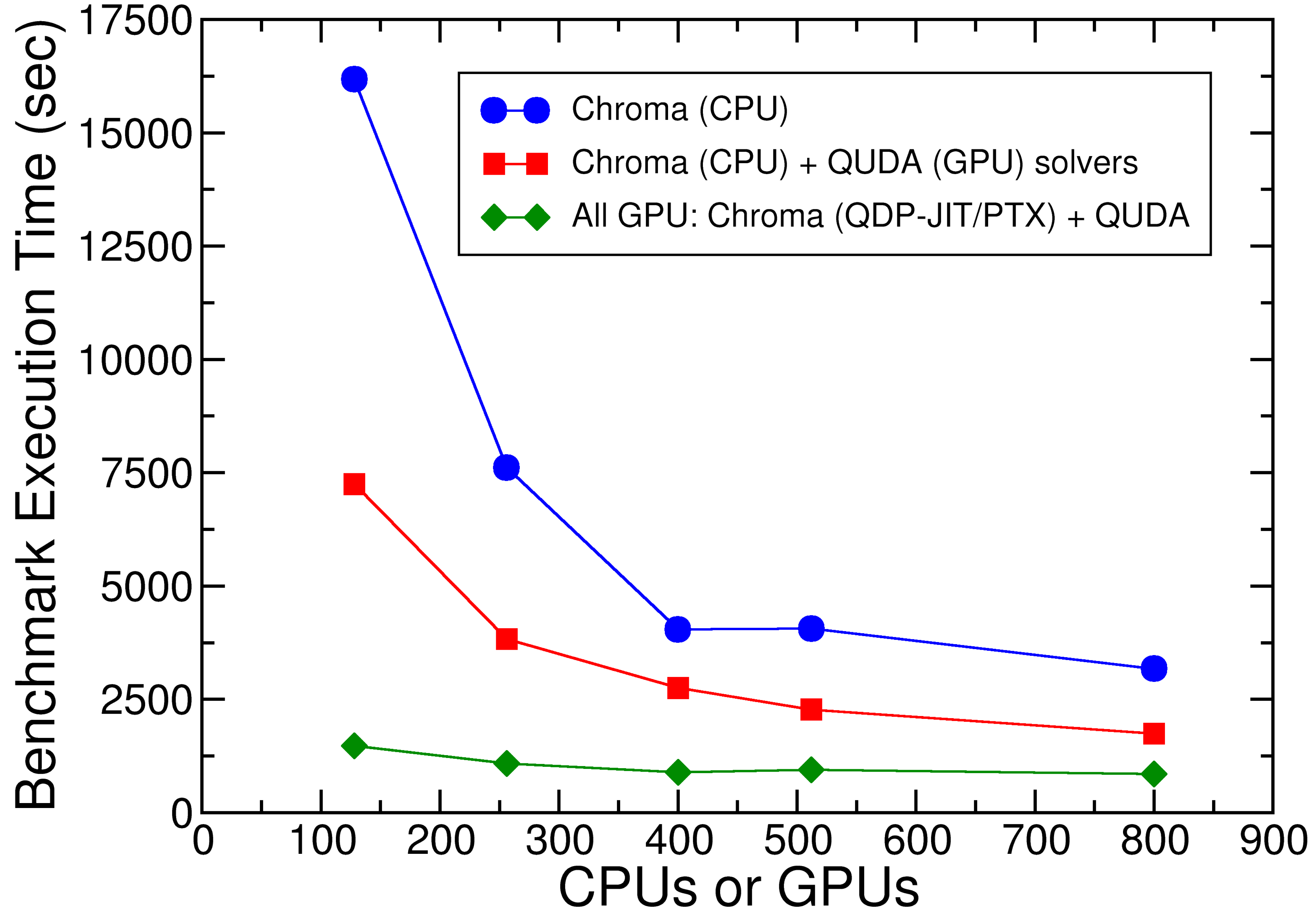
# LLVM: The Swiss Army Knife

- LLVM is compiler technology which underlies the implementations of current programming models:
  - Intel DPC++, HIPCC/HCC, NVCC, …
- Key concepts are
  - a front end: e.g. Clang for C++
  - an intermediate representation (IR)
  - back ends: NVPTX,AMDGPU,X86,Power,Arm etc.
- LLVM also includes Just-In-Time Compilers
  - compile functions/kernels at run-time
  - powering high level languages like Julia
- LLVM can be used to write portable and efficient Domain Specific Languages (DSLs).



# QDP-JIT, QDP++ as a DSL

- QDP-JIT  developed by F. Winter at JLab allowed us to move all of the QDP++ data parallel layer to GPUs.
  - Expression Templates (ET) generated CUDA PTX kernels
  - PTX Kernels were launched by CUDA driver
  - Automated Memory movement between host/device (via software cache)
  - Provided data layout flexibility
- Later, PTX generation moved to LLVM libraries
  - turns QDP-JIT into a DSL for QCD
- CPU version was developed to target x86/KNL
  - No ‘driver’, LLVM JIT-ed to objects (LLVM Modules)
  - Vector friendly layout was supported (including matching QPhiX)
- Reduced Amdahl’s law by accelerating the whole application, rather than just a library




F. T. Winter, M.A. Clark, R. G. Edwards, B. Joo, “A Framework for Lattice QCD Calculations on GPUs”, IPDPS’14, arXiv:1408.5925 [hep-lat]  (replotted)

# QDP-JIT via LLVM for AMD & Intel Xe?

# Conclusions & Future Work

- Both Kokkos and SYCL were sufficiently expressive for Dslash (parallel_for)
- Kokkos Dslash performed on par with QUDA on NVIDIA GPUs, and QPhiX on KNL (with SIMD type)
- SYCL performance depends a lot on the combination of compiler and driver
- LLVM is universal and allows constructing DSLs such as QDP-JIT
  - Ports of QDP-JIT will likely have different branches for each architecture (different dispatch, etc)
- Libraries are also being ported (not discussed here)
- Ongoing / Future work with Kokkos and SYCL
  - Warp/Subgroup level SIMD - in progress using Intel's SYCL Subgroup-ND range extension
  - Targeting AMD - in progress using new Kokkos HIP Back End, now looking at performance
  - Trying out the Kokkos SYCL/DPC++ back end and OpenMP offload back-ends as they develop
  - Evaluate using Kokkos to implement QDP++
  - Considering multi-node device aspects (communication)
- Lots of ongoing work by the LQCD Software Community on porting codes to ECP systems

# References

- KokkosDslash MiniApp:
  - Repo: https://github.com/bjoo/KokkosDslash.git
  - Workspace repo (with dependencies): https://github.com/bjoo/KokkosDslashWorkspace.git
- SyCLDslash MiniApp:
  - Repo: https://github.com/bjoo/SyCLDslash.git
  - Workspace repo (with dependencies): https://github.com/bjoo/SyCLDslashWorkspace.git
- Remember to clone with '—recursive' !!!
- Intel Publicly available SyCL Compiler: https://github.com/intel/llvm
  - sycl branch
- Kokkos: https://github.com/kokkos
- SyCL: https://www.khronos.org/sycl/
- CodePlay Compiler: https://www.codeplay.com/products/computesuite/computecpp
- USM Extension: https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/USM/USM.adoc
- Subgroup SIMD extension : https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/SubGroupNDRange/SubGroupNDRange.md
- QUDA: https://github.com/lattice/quda, https://lattice.github.io/quda, M. A. Clark et. al. Comput Phys. Commun. 181, 1517 (2010) [arXiv:0911.3191 [hep-lat]]
- QPhiX: https://github.com/jeffersonlab/qphix

# Acknowledgments


- B. Joo acknowledges funding from the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under the Exascale Computing Project (2.2.1.01 ADSE03 Lattice QCD )
- 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-4 program.
- 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 Cori-GPU, OLCF Summit