PERFORMANCE PORTABILITY FOR LATTICE QCD TOWARDS EXASCALE

Meifeng Lin

Computational Science Initiative
Brookhaven National Laboratory

The Ninth International Workshop on Numerical Analysis and Lattice QFT (QCDNA),
University of Edinburgh, August 1-3, 2016

OUTLINE

1. Introduction
2. Source-to-Source Compiler
3. OpenMP and OpenACC
4. Summary and Outlook

INTRODUCTION

WHAT IS PERFORMANCE PORTABILITY?

- Performance
  - High performance code achieved through platform-specific optimizations (the hero codes).
  - BAGEL (Bluegenes), QUDA (NVIDIA GPUs), MDWF (Intel CPUs), QPhiX (Intel MICs), ...
- Portability
  - The ability to have a single version of the software/algorithm across platforms
- Performance portability?
  - Is it possible to achieve **reasonable/acceptable** performance with a portable code?
  - What is acceptable performance?

- What is considered portable?
- In a way, Lattice QCD as an application is already performance portable: high-level application codes built on top of platform specific optimized lower-level APIs on current architectures.
- Chroma, CPS, FUEL, MILC, QLUA, ...

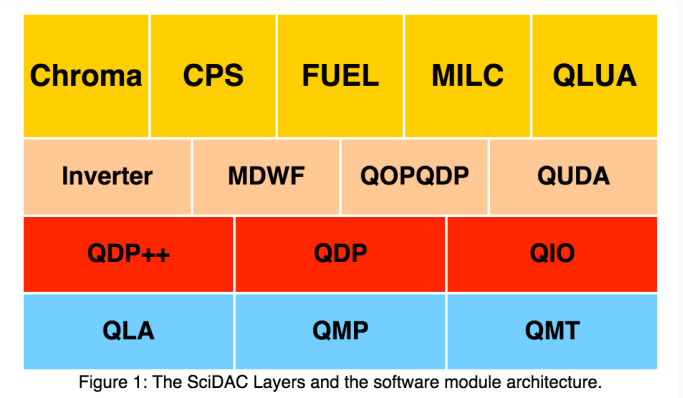


Figure 1: The SciDAC Layers and the software module architecture.

WHY DO WE CARE ABOUT PERFORMANCE PORTABILITY?

- A single version of portable code is easier to maintain.
- Less time spent on integrating the low-level APIs with the application layer, and more time on physics and algorithm development.
- Question: how much performance are we willing to lose in exchange for portability?
- The answer may be ”0”. But looking towards the future, with potentially more diverse architectures, are we able to continue our current approach?
- Performance portability is also one of the requirements for the US DOE exascale project.

Source: https://asc.llnl.gov/DOE-COE-Mtg-2016/

WAYS TO PERFORMANCE PORTABILITY?

Various tools are under development for performance[1] portability.

- High-level programming abstractions:
  - RAJA (LLNL)
  - Kokkos (Sandia)
  - SYCL (Kronos)
  - C++ AMP (Microsoft)
  - ...
- Source-to-source compilers/code generators:
  - JIT: QDP-JIT (JLab/Frank Winter)
  - Nim: QEX (ANL/James Osborn)
  - R-Stream compiler (Reservoir Labs)
- High-level programming directives
  - OpenMP
  - OpenACC

Question: should we design our new software with portability in mind first and then optimize for performance later, or the other way around? Can we design our software with performance portability in mind from the beginning?

[1] Your performance mileage may vary.

SOURCE-TO-SOURCE COMPILER

PARALLELIZATION AND OPTIMIZATION OF THE DOMAIN WALL DSLASH WITH THE R-STREAM SOURCE-TO-SOURCE COMPILER

DOE SBIR project. Work done with:

- Stony Brook University
  - **Eric Papenhausen** (CS PhD Student)
- Reservoir Labs Inc.
  - M. Harper Langston
  - Benoit Meister
  - Muthu Baskaran
- BNL
  - Chulwoo Jung
  - Taku Izubuchi

MOTIVATION

- In Lattice QCD (LQCD) simulations, the most computation intensive part is the inversion of the fermion Dirac matrix, M.
  - In quark propagator calculation, need to solve $M\phi = b$.
  - In gauge ensemble generation, need to solve $M^\dagger M\chi = \eta$.
- The recurring component of the matrix inversions is the application of the Dirac matrix on a fermion vector.
- For Wilson fermions, the Dirac matrix can be written as

$$M = 1 - \kappa D, \tag{1}$$

  up to a normalization factor, where κ is the hopping parameter, and D is the derivative part of the fermion matrix, the Dslash operator.
- The matrix-vector multiplication in LQCD essentially reduces to the application of the Dslash operator on a fermion vector.
- The motivations for this work are
  - to see if source-to-source code generators can produce reasonably performant code if only given a naive implementation of the Dslash operator as an input;
  - to investigate optimization strategies in terms of SIMD vectorization, OpenMP multithreading and multinode scaling with MPI.

- The Domain Wall (DW) fermion matrix can be written as

$$M^{DW}_{x,s;x',s'} = (4 - m_5)\delta_{x,x'}\delta_{s,s'} - \frac{1}{2}D^{W}_{x,x'}\delta_{s,s'} + D^{5}_{s,s'}\delta_{x,x'}, \tag{2}$$

  where m_5 is the domain wall height, $D^{W}_{x,x'}$ is the Wilson Dslash operator, and $D^{5}_{ss'}$ is the fermion mass term that couples the two boundaries in the 5th dimension,

$$\begin{aligned} D^{5}_{ss'} \quad = \quad & -\frac{1}{2}\left[(1-\gamma_5)\delta_{s+1,s'} + (1+\gamma_5)\delta_{s-1,s'} - 2\delta_{s,s'}\right] \\ & + \frac{m_f}{2}\left[(1-\gamma_5)\delta_{s,L_s-1}\delta_{0,s'} + (1+\gamma_5)\delta_{s,0}\delta_{L_s-1,s'}\right]. \end{aligned} \tag{3}$$

- Most FLOPs are in the 4D derivative term (DWF 4D Dslash) in Eq.(2): **1320 flops per site**.
- $\hookrightarrow$ focus of our optimizations.

R-STREAM SOURCE-TO-SOURCE COMPILER

- The R-Stream source-to-source compiler developed by Reservoir Labs Inc. takes serial C programs as inputs and can perform optimizations in terms of parallelization, memory management, data locality etc. to target a range of different architectures.

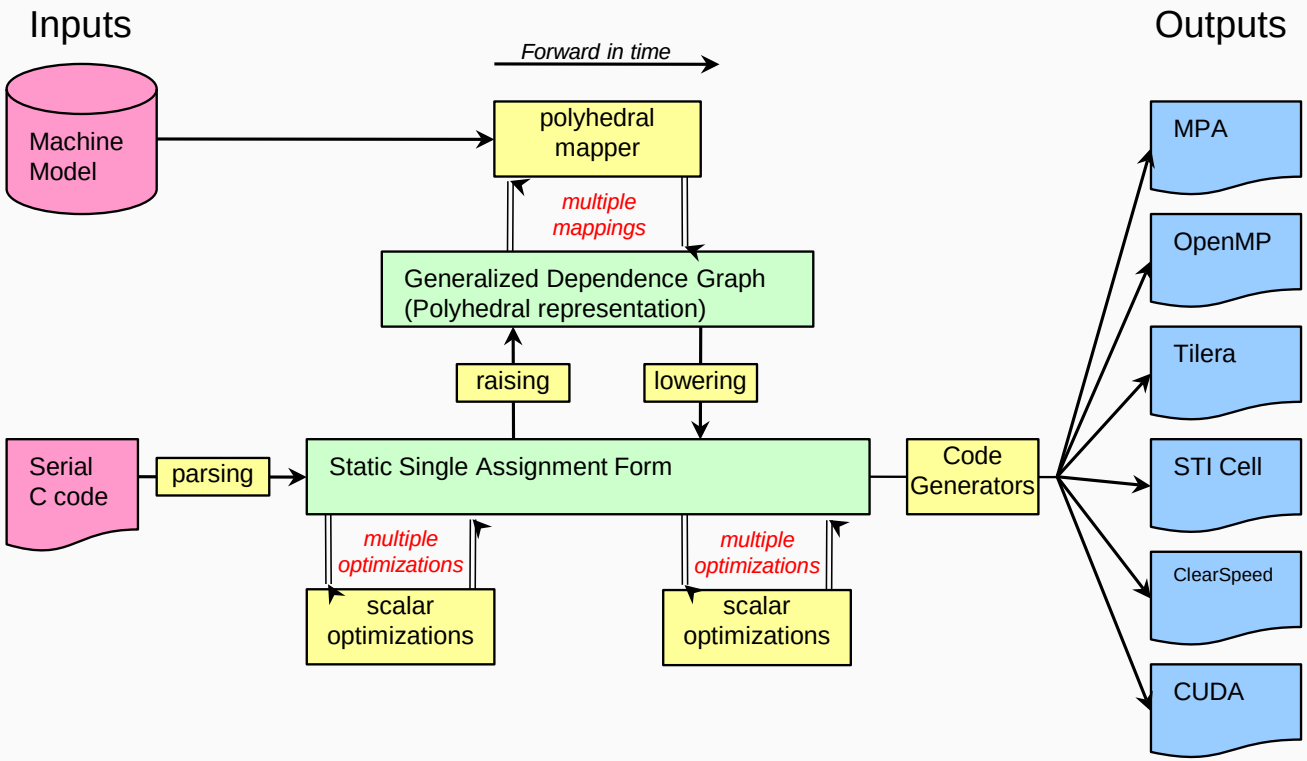


Figure 1: R-Stream workflow. Image from Papenhausen et al., VISSOFT15 Proceedings.

R-STREAM TRANSFORMATION EXAMPLE

Input Code Example

```
#pragma rstream map
void matmult_c(real_t Ai[NSIZE][NSIZE],
       real_t Bi[NSIZE][NSIZE],
       real_t Ci[NSIZE][NSIZE]) {
  int i, j, k;
  for (i = 0; i < NSIZE; i++) {
    for (j = 0; j < NSIZE; j++) {
      for (k = 0; k < NSIZE; k++) {
        Ci[i][j] += Ai[i][k] * Bi[k][j];
  } } }
}
```

- Matrix multiplication example written in C; not parallelized.
- Generated code has loop tiling and OpenMP multithreading.

R-Stream OpenMP Output

```
void matmult_c(real_t (* Ai)[1024],
       real_t (* Bi)[1024], real_t (* Ci)
       [1024]) {
  int i;
  #pragma omp parallel for private(i)
  for (i = 0; i <= 15; i++) {
    int _t1;
    int j;
    _t1 = - i + 15 >> 4;
    for (j = - i + 15 >> 4; j <= _t1; j
       ++) {
      int k;
      for (k = 0; k <= 1; k++) {
        int i1;
  for (i1 = 0; i1 <= 511; i1++) {
    int j1;
    for (j1 = 0; j1 <= 1023; j1++) {
      int k1;
      for (k1 = 0; k1 <= 63; k1++) {
      Ci[512 * k + i1][64 * i + (1024 *
       j + k1)] = Ci[512 * k +
        i1][64 * i + (1024 * j + k1)] +
       Ai[512 * k + i1][j1] *
        Bi[j1][64 * i + (1024 * j + k1)
       ];
  } }} }} }
}
```

R-STREAM TRANSFORMATION OF THE DW 4D DSLASH

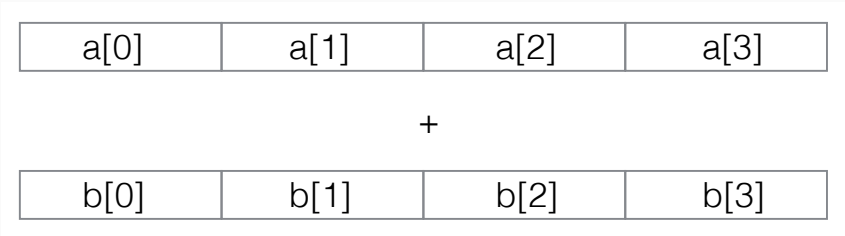
- The input code we used is the unoptimized `noarch` implementation of the Dslash in CPS.
- Most straightforward implementation, direct transcription of the Dslash definition.
- Some manual code transformation was needed to get R-Stream to parse the code:
  - **Delinearized array access**: 1D array $\rightarrow$ multidimensional array
  - **Removal of the modulo statements**: introduced boundary padding.

- With these changes, R-Stream was able to produce generated code. However, the resulting code did not give very good performance. Some hand tuning was required.

SINGLE INSTRUCTION MULTIPLE DATA (SIMD)

- Modern CPUs, both by Intel and AMD, support vector instructions.
  - **SSE**: 128-bit vector register, capable of 2 DP/4 SP flops per cycle.
  - **AVX**: 256-bit vector register, capable of 4 DP/8 SP flops per cycle.
  - **AVX2**: AVX with fused multiply-add (FMA).
  - AVX512: Intel KNL, Skylake, ...
- Data layout is the key: Data in one SIMD operation need to fit into the same vector register. With AVX, the following instructions should be able to execute in one clock cycle.

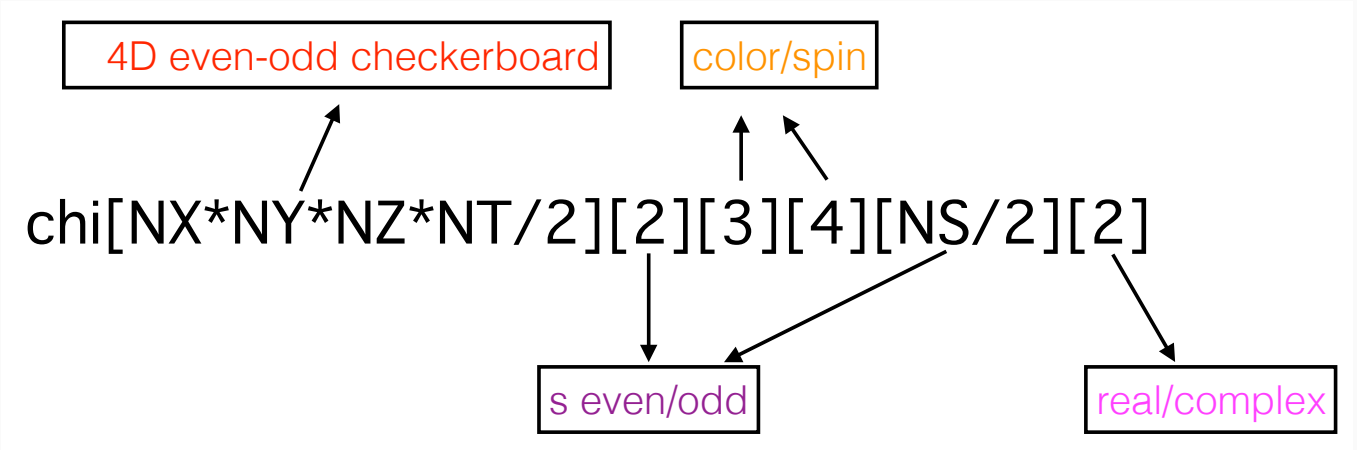
```
double a[4], b[4], c[4];
for (int n=0; n<4; n++) c[n] = a[n] + b[n];
```



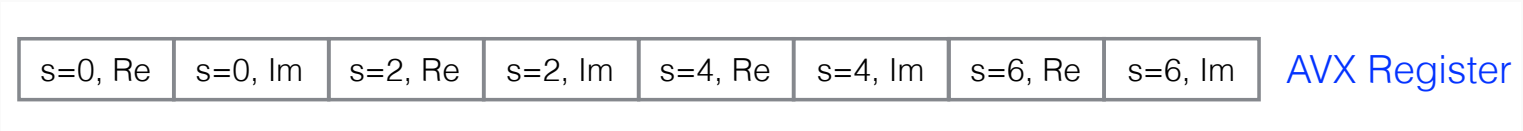
- There also cannot be any data dependencies among the SIMD data.
- In DWF 4D Dslash, the s coordinates are completely independent. $\hookrightarrow$ Good place to vectorize.

DWF DATA LAYOUT

- We chose the following data layout to enable us to vectorize in the fifth (s) dimension.



- In one AVX register, with single precision, the data mapping goes



- SIMD intrinsics were used to implement the vectorized DWF Dslash.

- Caveat: L_s is restricted to be multiples of 8 in single precision, and multiples of 4 in double precision.

OTHER OPTIMIZATIONS

- **FMA**: AVX2 provides intrinsics to perform fused multiply-add. However, we found that simply turning on -mfma compiler option for gcc gave us the same performance boost as using intrinsics.
- **Improved data locality**:
  - We studied tiling to increase memory reuse, but didn't gain any performance.
  - We also explored using a space-filling curve, implemented as the Z-curve, to improve data locality, but the performance boost was minimal.
- **Prefetching:** Before the computation of each stencil operation, prefetch data needed for the next stencil. Led to 10% performance improvement.

- On Intel(R) Xeon(R) CPU E5-2690 v3 @ 2.60GHz processor (Haswell), with $8^4 \times 8$ lattice, we achieved 34% peak single-core performance in single precision.

Optimization	AVX2	Tiling	Z-Curve	Prefetching
time [ms]	0.86	0.92	1.0	0.76
Gflops	25.1	23.5	21.6	28.5

- Within the node, we use OpenMP for multithreading.
- Three strategies have been explored:
  - Simple Pragma: Thread the outer loop, usually the t loop.
    $\hookrightarrow$ Parallelism is limited by the t dimension size, won't scale well in many-core systems.
  - Compressed Loop: Compress the nested loops into one single loop.
  - Explicit Work Distribution: Similar to Compressed Loop, but explicitly assign work to each thread.

```
#pragma omp parallel
  {
   int nthreads = omp_get_num_threads();
   int tid = omp_get_thread_num();
   int work = NT*NZ*NY*(NX/2)/nthreads;
   int start = tid * work;
   int end = (tid+1) * work;
   for(lat_idx = start; lat_idx < end; lat_idx++)
   ......
  }
```

Performance was measured on LIRED, with dual-socket Haswell per node @ 2.6 GHz (24 cores).

- $8^4 \times 8$

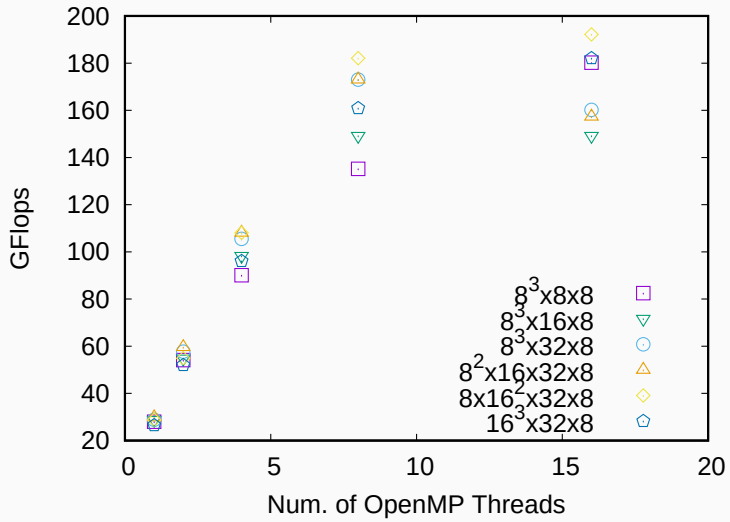
Num. Threads	Simple Pragma	Compressed Loop	Explicit Dist.
1	28.4 GF/s	28.0 GF/s	28.0 GF/s
2	51.5 GF/s	54.1 GF/s	54.1 GF/s
4	90.1 GF/s	90.1 GF/s	90.1 GF/s
8	135.2 GF/s	135.2 GF/s	144.2 GF/s
16	127.2 GF/s	180.2 GF/s	154.4 GF/s

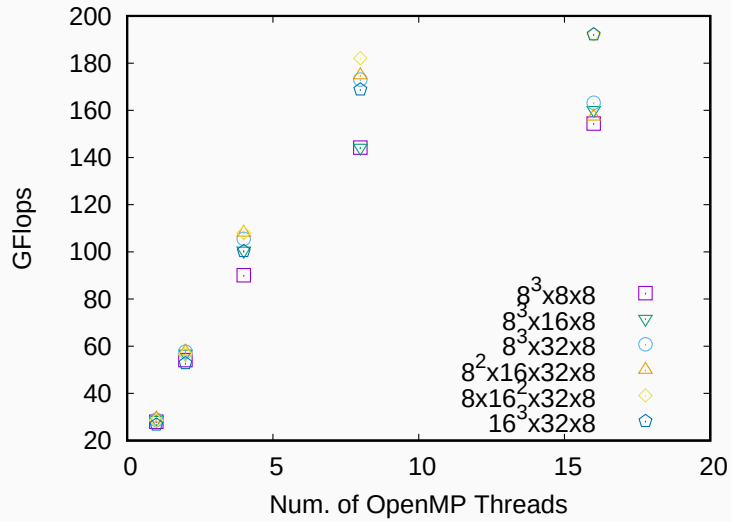
- $16^3 \times 32 \times 8$:

Num. Threads	Simple Pragma	Compressed Loop	Explicit Dist.
1	26.9 GF/s	26.5 GF/s	26.8 GF/s
2	54.5 GF/s	52.0 GF/s	52.8 GF/s
4	100.3 GF/s	96.1 GF/s	100.3 GF/s
8	168.8 GF/s	160.9 GF/s	168.8 GF/s
16	197.7 GF/s	182.1 GF/s	192.2 GF/s

OPENMP SUMMARY

- Three threading approaches result in similar performances, except when the problem size is small, Simple Pragma doesn't scale as well.
- Surprisingly, the performance does not deteriorate with a much larger lattice size
  ↪ possible indication of poor cache reuse.
- Volume comparison:
  Left - Compressed Loop. Right - Explicit Work Distribution.



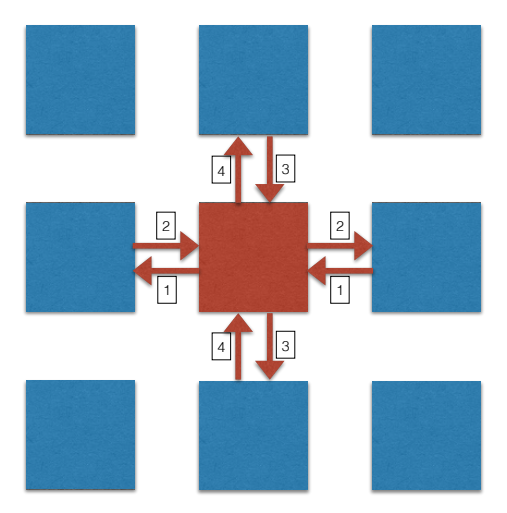


We also found that that binding OpenMP threads to the processors can improve the OpenMP performance a lot. With gcc, this is done through

```
export OMP_PROC_BIND=true
```

INTERNODE COMMUNICATION

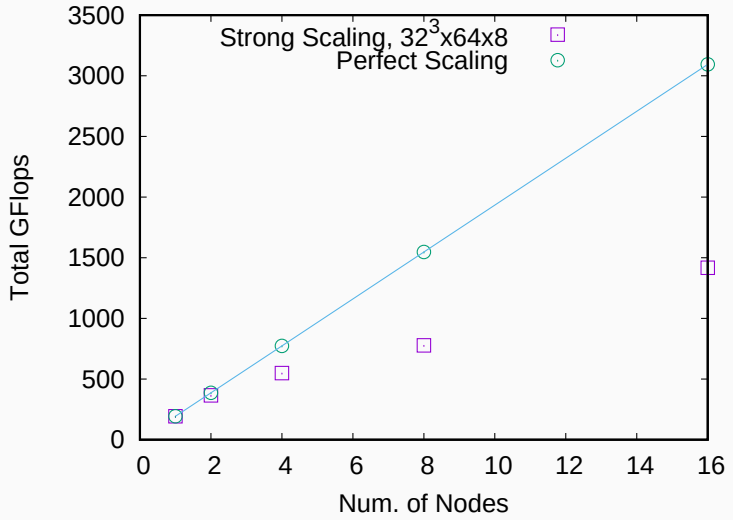
- We use QMP for communications between nodes.
- The communication pattern is illustrated in the following. There is blocking for each transfer sequence.

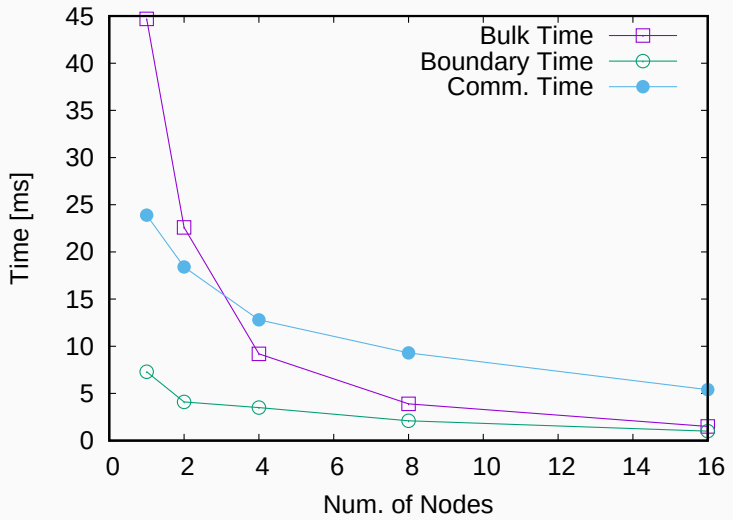


- The best performance is obtained with 2 MPI processes per node (1 MPI process per socket, improved data locality).
- With each MPI process, a number of threads equal to the number of compute cores are used.
- We dedicate one thread (the `master` thread) to do the communications, and the rest of the threads for computation.
- Do bulk computation first while waiting for the communication to complete, then do the boundary computation.

MULTINODE PERFORMANCE

- Strong scaling study of a $32^3 \times 64 \times 8$ calculation was performed on LIRED, with dual-socket Intel Haswell CPUs and Mellanox 56 Gigabit FDR interconnect.
- The performance scales well up to 4 nodes, and scales sublinearly from 8 to 16 nodes.
- After 4 nodes, the total time is dominated by the communication time.
- Bulk computation itself scales well with the number of nodes.
- Rediscovered the old truth: Communication is the bottleneck for strong scaling!





SOURCE-TO-SOURCE COMPILER SUMMARY

- It is difficult to produce portable high-performing code with one input code.
- With manual optimizations, we have gained good performance but lost portability (intrinsics).

OPENMP AND OPENACC

OVERVIEW

- Compiler directives provide another way to achieve (performance) portability.
- OpenMP and OpenACC are two programming directive standards that have introduced support for both CPUs and GPUs.
- OpenMP supports offloading calculations to accelerators through a `target` clause.
- OpenACC does this through a compiler target option (-ta=tesla, -ta=multicore, etc.).
- Prescriptive vs. Descriptive:
  - OpenMP is prescriptive: you tell the compilers exactly what to do.
  - OpenACC is descriptive: you tell the compiler what you want to do and the compiler does the optimization for you. Example, `kernels`.
- Data movement:
  - Both support `data` clauses to move data between host and devices.
  - OpenACC also supports *unstructured* data movement: `enter data`, `exit data`.

OPENMP VS OPENACC

```
extern void init(float*, float*, int);
extern void output(float*, int);
void vec_mult(int N)
{
 int i;
 float p[N], v1[N], v2[N];
 init(v1, v2, N);
 #pragma omp target data map(to:v1[0:N],
       v2[0:N]) map(from:p[0:N])
 #pragma omp parallel for private(i)
 for (i=0; i<N; i++)
   p[i] = v1[i] * v2[i];
 output(p, N);
}
```

Listing 1: OpenMP example.

```
extern void init(float*, float*, int);
extern void output(float*, int);
void vec_mult(int N)
{
 int i;
 float p[N], v1[N], v2[N];
 init(v1, v2, N);
 #pragma acc data copyin(v1[0:N],v2[0:N
       ]) copyout(p[0:N])
 #pragma acc kernels
 for (i=0; i<N; i++)
   p[i] = v1[i] * v2[i];
 output(p, N);
}
```

Listing 2: OpenACC example.

OFFLOAD COMPILER SUPPORT

- OpenMP:
  - GCC: v6.1 has full C/C++ support for OpenMP 4.5.
  - Intel: v16 has support for OpenMP 4.0.
  - Cray: supports OpenMP 4.0
  - Clang/LLVM: v3.8 supports some OpenMP 4.0 and 4.5
  - ...
- OpenACC:
  - PGI (NVIDIA)
  - Cray
  - GCC 6.1
  - Research compilers: OpenUH (U of Houston), OpenARC (ORNL)
- Targets/Architectures (to be) supported:
  - AMD and NVIDIA GPUs
  - Intel MICs and Xeons
  - IBM Power
  - ARM
  - FPGA
  - ...

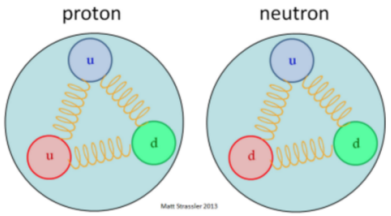
RECENT HACKATHON EXPERIENCE

- **Goal**: to make Grid[2] portable to GPUs
- **Approach**: use OpenACC
- **Venue:** University of Delaware
- **Team:** Chulwoo Jung, Zhihua Dong, Chris Kelly, ML, Mathias Wagner (NVIDIA), Mathew Colgrove (PGI)

Courtesy: University of Delaware

UDEL HACKTHON

5DSpeedsters



- Higgs Boson is the origin of mass in the Universe? WRONG! It's mainly **QCD**.
- QCD, Quantum Chromodynamics, is the theory of the **strong force**; as the name suggests, it's strong, much stronger than electromagnetism, gravity and the weak force.
- The QCD binding energy comprises over 99% of the mass to the proton and neutron (and basically every other composite particle).

[2] P. Boyle, A. Yamaguchi, A. Portelli, G. Cossu, arXiv:1512.03487

ISSUES

- Grid makes use of a lot of C++11 features, which caused some problems with the PGI compiler, even for CPUs.
  - std random number generator hangs. ↪ boost::random
  - std::thread not supported. ↪ Got rid of it.
  - Intrinsics not supported. ↪ Replaced with generic vector types.
- Offloading to GPUs is complicated by the user-defined data types. Any lattice wide operations (offloadable) will involve Spin Color Vectors.
- axpy example:

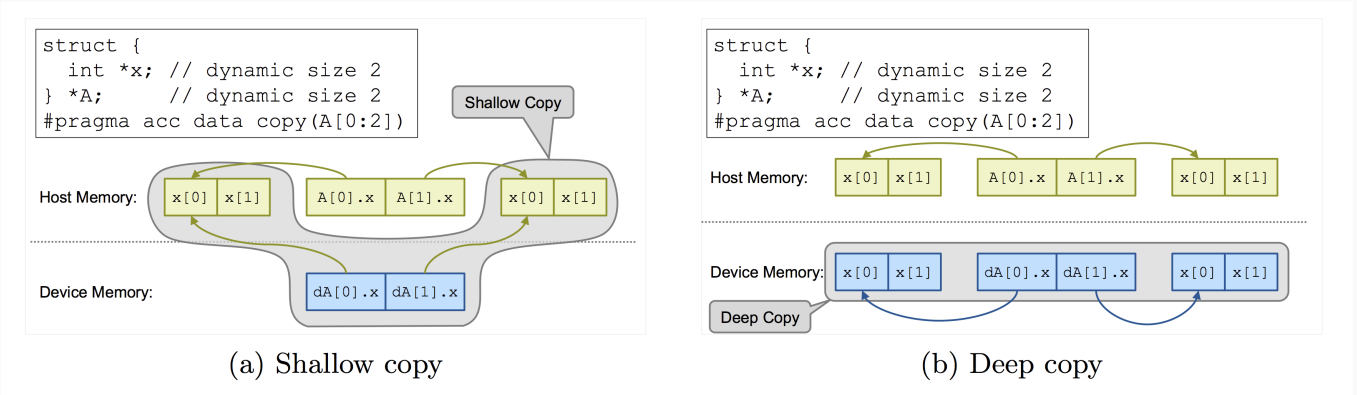
```
template<class sobj,class vobj> strong_inline
  void axpy(Lattice<vobj> &ret,sobj a,const Lattice<vobj> &x,const Lattice<vobj> &y){
 ...
    const vobj *xdata = x._odata.data();
    const vobj *ydata = y._odata.data();
    vobj *retdata = ret._odata.data();
 ...
#pragma acc data \
   copyin(a,xdata[0:size],ydata[0:size]) \
   copyout(retdata[0:size])
   {
   #pragma acc kernels
   for(int ss=0;ss<sites;ss++)    retdata[ss] = a*xdata[ss]+ydata[ss];
   }
```

- It runs on GPUs but gives the **incorrect** results. ↪ Issues with **deep copy?**

```
a=-1
xdata[size-1]S {V<4>{V<3>{<(-143.009,-141.73),(-121.144,-128.706)>,<(-131.458,-117.319),...}}}
ydata[size-1]S {V<4>{V<3>{<(-29.8989,-48.3114),(-57.2752,-29.1623)>,<(-43.3413,-41.3039),...}}}
--------------------
retdata[size-1]S {V<4>{V<3>{<(141.046,90.0025),(91.2382,113.036)>,<(110.604,72.2897),...}}}
```

DEEP COPY

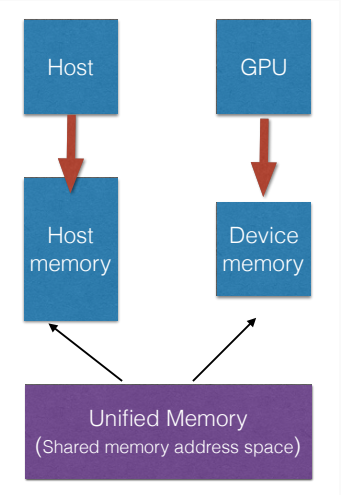
- Current OpenACC standard does not support *deep copy*.
- For user-defined data types (arrays of structures), simple copy will result in incorrect pointer dereferencing.



(a) Shallow copy (b) Deep copy

Source: www.openacc.org

- Work around: PGI compiler provides the ”managed” option to use NVIDIA unified memory.
- Pre-Pascal: The maximum size of the unified memory space is limited by the GPU memory.
- Pascal: Allows oversubscription of GPU memory. Unified memory can be as big as host memory.



MAYBE WE JUST WORKED ON THE WRONG APPLICATION!

- Took an hour to get CPS to run on GPUs simply by using `kernels` and -ta=tesla:managed.
- But of course the performance is poor. Spends 99% of time doing data movement.
- Not a fair comparison: CPS is mostly C, and doesn't have the same challenges as Grid.

```
Node 0: DiracOp::InvCg(V*,V*,F,F*): flops_per_site=11444
Node 0: DiracOp::InvCg(V*,V*,F,F*): True |res| / |src| = 7.977347e-08, iter = 70
Node 0: MatInv(V*,V*,F*):MatPcDat+InvCg(): 0.000000e+00 flops /1.153881e+01 seco
nds = 0.000000e+00 MFlops
Node 0: MatInv(V*,V*,F*):After InvCg(): 0.000000e+00 flops /4.665399e-02 seconds
 = 0.000000e+00 MFlops
Node 0: lattice:Convert(): 0.000000e+00 flops /8.998871e-03 seconds = 0.000000e+
00 MFlops
==45000== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 49.82%  2.78794s       630  4.4253ms  4.3899ms  4.5032ms  wilson_dslash_blk_dag
0_87_gpu
 49.03%  2.74364s       620  4.4252ms  4.3887ms  4.4955ms  wilson_dslash_blk_dag
1_65_gpu
  0.71%  39.635ms        21  1.8874ms  1.8857ms  1.8892ms  wilson_mdagm_175_gpu
  0.21%  12.004ms      5000  2.4000us  2.2080us  3.8080us  [CUDA memcpy HtoD]
  0.20%  11.022ms      5000  2.2040us  2.1120us  5.6000us  [CUDA memcpy DtoH]
  0.03%  1.8704ms         1  1.8704ms  1.8704ms  1.8704ms  wilson_mdag_97_gpu

==45000== Unified Memory profiling result:
Device "Tesla K20Xm (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
    2338  701.21KB  4.0000KB  3.0000MB  1.563473GB  927.6498ms  Host To Device
  385859  4.0000KB  4.0000KB  4.0000KB  1.471935GB   4.828694s  Device To Host

==45000== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 75.66%  5.60025s      5044  1.1103ms  1.6610us  4.5472ms  cuStreamSynchronize
 14.89%  1.10246s      1272  866.72us  13.291us  14.306ms  cuLaunchKernel
  4.50%  332.96ms         1  332.96ms  332.96ms  332.96ms  cuDevicePrimaryCtxRet
ain
  1.28%  94.855ms         1  94.855ms  94.855ms  94.855ms  cuDevicePrimaryCtxRel
ease
  0.93%  68.858ms         1  68.858ms  68.858ms  68.858ms  cuModuleLoadData
  0.59%  43.809ms       110  398.26us  31.218us  34.495ms  cuMemAllocManaged
  0.50%  37.167ms      5000  7.4330us  3.8970us  449.99us  cuMemcpyDtoHAsync
  0.46%  33.712ms         1  33.712ms  33.712ms  33.712ms  cuMemHostAlloc
  0.45%  33.381ms      5000  6.6760us  4.1240us  80.546us  cuMemcpyHtoDAsync
  0.18%  12.987ms      5000  2.5970us  1.7270us  73.095us  cuEventRecord
  0.17%  12.458ms     17886     696ns     188ns  983.57us  cuPointerGetAttribute
s
```

SUMMARY AND OUTLOOK

Summary

- Achieving performance portability is important for exascale.
- Ongoing efforts on using source-to-source compilers and programming directives to make existing LQCD codes performance portable.
- Challenges still remain to use compiler directives in C++ applications.

Outlook

- With Unified Memory and/or support for *deep copy*, it is possible to get existing C++ LQCD codes to run on GPUs.
- Should we design the next-generation of LQCD codes with performance portability in mind from the get-go?
- We should try to influence the directive standards to better support our needs.
  $\hookrightarrow$ BNL is now an OpenACC member organization.