# OPTIMIZATION OF THE DOMAIN WALL DSLASH OPERATOR FOR INTEL XEON CPUS

Meifeng Lin
Computational Science Initiative
Brookhaven National Laboratory

The 34th International Symposium on Lattice Field Theory
University of Southampton, July 24 - 29, 2016

## COLLABORATORS

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

## OUTLINE

## INTRODUCTION

## 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 $\kappa$ 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 DSLASH OPERATOR

- 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.

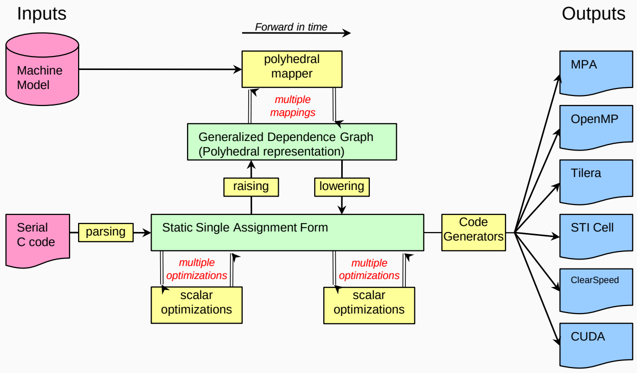


**Figure 1:** R-Stream workflow. Image from Papenhausen et al., VISSOFT15 Proceedings.

## R-STREAM TRANSFORMATION OF THE DW 4D DSLASH

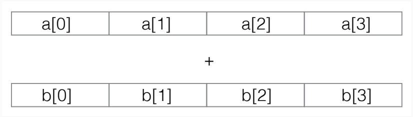
- The input code we used is the unoptimized `noarch` implementation of the Dslash in CPS.
- Most straightforward implementation, direct transcription of the Dslash definition.
- Some manual code transformation was needed to get R-Stream to parse the code:
  - **Delinearized array access**: 1D array $\longrightarrow$ 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.
- The rest of the talk will focus on the hand-tuning efforts.

## SINGLE-CORE OPTIMIZATION

## 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).
- 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.

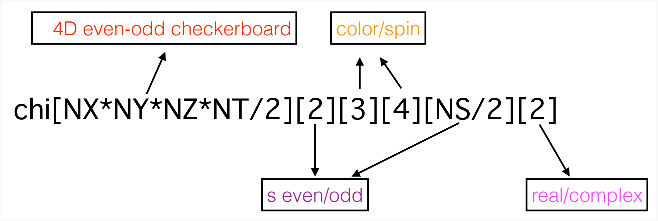
```
double a[4], b[4], c[4];
for (int n=0; n<4; n++) c[n] = a[n] + b[n];
```



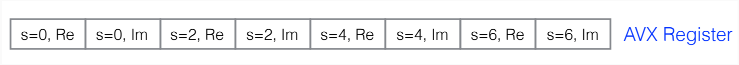
- There also cannot be any data dependencies among the SIMD data.
- In DWF 4D Dslash, the $s$ coordinates are completely independent. $\hookrightarrow$ Good place to vectorize.

## 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



|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| s=0, Re | s=0, Im | s=2, Re | s=2, Im | s=4, Re | s=4, Im | s=6, Re | s=6, Im |

AVX Register

- 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 |

## OPENMP OPTIMIZATION

## MULTITHREADING WITH OPENMP

- 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++)
   ......
  }
```

## OPENMP PERFORMANCE

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

- $8^4 \times 8$

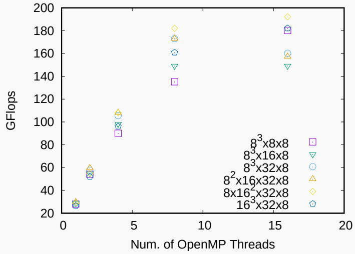
| Num. Threads | Simple Pragma | Compressed Loop | Explicit Dist. |
|---|---|---|---|
| 1 | 28.4 GF/s | 28.0 GF/s | 28.0 GF/s |
| 2 | 51.5 GF/s | 54.1 GF/s | 54.1 GF/s |
| 4 | 90.1 GF/s | 90.1 GF/s | 90.1 GF/s |
| 8 | 135.2 GF/s | 135.2 GF/s | 144.2 GF/s |
| 16 | 127.2 GF/s | 180.2 GF/s | 154.4 GF/s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

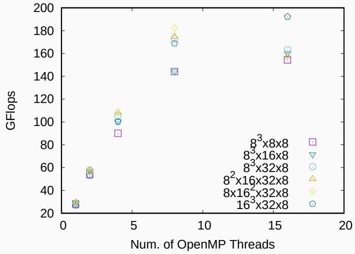
- $16^3 \times 32 \times 8$:

| 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 $\hookrightarrow$ possible indication of poor cache reuse.
- Volume comparison:
  Left - Compressed Loop. Right - Explicit Work Distribution.



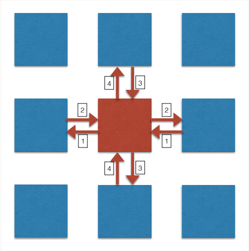


We also found that that binding OpenMP threads to the processors can improve the OpenMP performance a lot. With gcc, this is done through
`export OMP_PROC_BIND=true`

## MULTINODE/MPI OPTIMIZATION

## INTERNODE COMMUNICATION

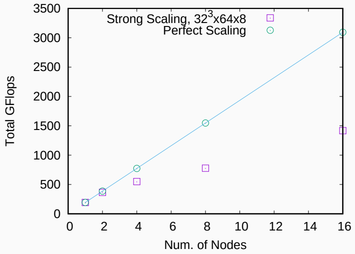
- We use QMP for communications between nodes.
- The communication pattern is illustrated in the following. There is blocking for each transfer sequence.

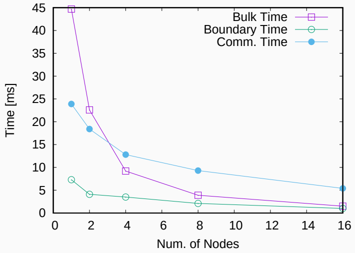


- The best performance is obtained with 2 MPI processes per node (1 MPI process per socket, improved data locality).
- With each MPI process, a number of threads equal to the number of compute cores are used.
- We dedicate one thread (the `master` thread) to do the communications, and the rest of the 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!





## CONCLUSIONS

## CONCLUSIONS

- To produce efficient Dslash code, optimizations in terms of data layout, SIMD, OpenMP scaling and internode communications have been studied.
- By vectorizing and changing the memory access pattern, we obtained 34% peak single-core performance in single precision.
  $\hookrightarrow$May still have poor cache reuse.
- On single node, OpenMP scaling deteriorates after 16 threads.
  $\hookrightarrow$ Further improvements possible.
- Multinode strong scaling is limited by the communication cost.
  $\hookrightarrow$ Better (higher-bandwidth) interconnects are critical.

## ACKNOWLEDGMENTS

## ACKNOWLEDGMENTS

- This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Nuclear Physics under Award Number DE-SC0009678.
- Results in this presentation were obtained using the high-performance Handy and LIRED computing systems at the Institute for Advanced Computational Science at Stony Brook University and the hpc1 computing cluster at the Brookhaven National Laboratory.

## BACKUP SLIDES

## RECAP OF LATTICE 2015 RESULTS

- Double precision, AVX2.
- Simple Pragma OpenMP implementation.

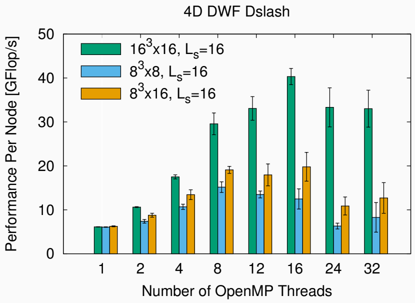


**Figure 2:** DWF 4D Dslash performance w.r.t. volume. Lin et al. PoS(LATTICE15)

.

- OpenMP scaling saturated at 16 threads. Limited by $NT$.
- Got 4-10X speedup in the current single-precision implementation.

## OVERLAPPING COMPUTATION AND COMMUNICATION

- Explicit work assignment with OpenMP.
- 1 master thread for communication, and worker threads for computation.
- Pseudo Code snippet.

```
#pragma omp parallel
  {
   int tid = omp_get_thread_num();
   if(tid == 0){ // master thread performs communication
    QMP_Start(x_multiple); // transfer along X direction
    QMP_Start(y_multiple); // transfer along Y direction
. . .
   } else { // worker threads
    int work = NT*NZ*NY*(NX/2)/nthreads;
    int start = (tid-1) * work;
    int end = tid * work;
    for(lat_idx = start; lat_idx < end; lat_idx++) {
     if(is_boundary(lat_idx))
      if(is_comm_finished())
       boundary_queue[tid].push(lat_idx);
       continue;
     // perform bulk computation
    } // end for loop
    wait_for_comm(); // wait for communication
    // process boundary points
    for(lat_idx = 0; lat_idx < boundary_queue[tid].size(); lat_idx++)
. . .
```