Optimizing MILC-Dslash Performance on NVIDIA A100 GPU: Parallel Strategies using SYCL

1st Amanda S. Dufek
NERSC
Lawrence Berkeley National Lab
Berkeley, CA, USA
asdufek@lbl.gov

2nd Steven A. Gottlieb
Department of Physics
Indiana University
Bloomington, IN, USA
sg@iu.edu

3rd Muaaz Gul Awan
NERSC
Lawrence Berkeley National Lab
Berkeley, CA, USA
mgawan@lbl.gov

4th Douglas Adriano Augusto
Oswaldo Cruz Foundation
Rio de Janeiro, RJ, Brazil
daa@fiocruz.br

5th Jack Deslippe
NERSC
Lawrence Berkeley National Lab
Berkeley, CA, USA
jrdeslippe@lbl.gov

6th Brandon Cook
NERSC
Lawrence Berkeley National Lab
Berkeley, CA, USA
bgcook@lbl.gov

***Abstract*—MILC-Dslash is a benchmark that is derived from the MILC code which simulates lattice-gauge theory on a four-dimensional hypercube. This paper outlines a gradual progression in increasing the granularity of parallelism in the MILC-Dslash kernel using the SYCL programming model, transitioning from a simple to a fully parallel implementation. We explore the impact of various parallel strategies on the MILC-Dslash performance on an NVIDIA A100 GPU. This investigation encompasses different work-item index orders, work-group sizes, and memory access patterns that arise from these strategies. Examples of components intertwined with the parallel strategies include atomic memory operations, shared variables, divergent instructions, synchronization barrier, scenarios with and without dependencies between iterations, as well as versions with and without using the SYCL complex library (SyclCPLX) and the SYCLomatic tool. The best parallel strategy is twice as fast as the simplest strategy and shows a 10% improvement over the QUDA baseline, thanks to enhanced parallelism and the use of work-group local memory. This, along with other findings — such as optimizing GPU resource utilization even at the expense of concurrency, prioritizing the use of work-item indexing methods that favor more localized memory access patterns, and maximizing both the number of active work-items per warp and the sequence of successive active work-items — could provide valuable guidance for researchers and developers seeking to optimize parallel computing applications.**

***Index Terms*—MILC-Dslash, SYCL, SyclCPLX, SYCLomatic, QUDA**

I. Introduction

MILC-Dslash is a benchmark that is derived from the MILC code [1] which simulates four-dimensional hypercubic lattice quantum chromodynamics (QCD). The application `su3_rhmd_hisq` from the MILC code suite is one of the main applications used to generate gauge configurations with staggered quarks. It has been used in production for many years at U.S. Department of Energy (DOE) and U.S. National Science Foundation (NSF) supercomputer centers. It has also been used in benchmark requirements for new supercomputer acquisitions such as for NSF's Blue Waters, several generations of NERSC computers (including N10), and ORNL's Frontier and upcoming Discovery.

The so-called Dslash operator and its corresponding kernel is one of the most important ones for lattice QCD. Dslash describes the interaction of the quarks with the gluons. The name comes from a notation invented by Feynman in which a vector, say p, is Lorentz contracted with the Dirac gamma matrices would be denoted $\not{p}$, with an angled bar crossing the letter p. The Dirac equation for spin-1/2 particles involves a differential operator D, hence Dslash or $\not{D}$.

Many details of QCD and its applications can be found in Ref. [2]. This reference covers both the continuum and lattice versions of the theory. In lattice field theory, there are a number of common ways to describe the quarks. These include Wilson (or Wilson-clover), Domain Wall, and staggered. In the Wilson approach, there are four spin-components at each site, each of which is an SU(3) color vector. The stencil involves eight neighbor sites displaced by $\pm x$, $\pm y$, $\pm z$, and $\pm t$. In the Domain Wall approach, a fifth dimension is introduced. The quarks, that have improved chiral symmetry are on the edges of the domain. The gauge fields don't vary in this extra dimension, so there is a higher arithmetic intensity as quark fields for each value of the extra dimension interact with the same gauge fields.

The formulation that we consider here is called staggered, or Kogut-Susskind. It requires only one SU(3) color vector at each site. So, we don't benefit from applying the gauge field to each of four SU(3) vectors as in the Wilson case. Another difference with staggered quarks is that modern formulations involve terms with both first and third nearest neighbors, so it is a 16 point stencil rather

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00151

than eight. Because the arithmetic intensity of staggered quarks is low compared to the other two formulation, it is important to pay careful attention to memory traffic.

The Dslash operator is also a major component in other lattice QCD applications like Chroma [3], for example. In lattice QCD, the space-time continuum is discretized on a four-dimensional lattice, where quark "fields" contain complex numbers (similar to a electron wavefunction), and spinor and "color" components. Gluon fields connect neighboring lattice points and define the Dslash operator. The application of this operator is a key aspect of lattice QCD simulations (see Section II). It also represents a Stencil operation that is similar to those at the core of many computational differential equations. So, what we learn from optimizing the MILC-Dslash benchmark not only enhances lattice QCD numerical simulations but also can be applied to numerous other problems in the greater field of high-performance computing. For instance, this knowledge can be extended to computational fluid dynamics, real-space implementations of material science applications, and lattice-based simulations in other research domains like epidemiology and ecology.

Previous work [4] has demonstrated a comparative performance analysis between portable (SYCL and Kokkos) and accelerator-specific (CUDA and HIP) programming models of a parallel MILC-Dslash implementation across GPU architectures from three different vendors: NVIDIA, AMD, and Intel. The authors concluded that SYCL and Kokkos showed satisfactory performance portability across NVIDIA, AMD, and Intel GPUs, with speed-up values very close to those obtained by CUDA and HIP.

SYCL [5] has proved to be a parallel programming model effectively portable across multiple hardware architectures, vendors, and generations. It has experienced substantial growth in usage, gaining popularity in both the scientific and industry communities. SYCL stands as a high-level open-standard framework, providing a single-source solution for the development of modern C++ parallel applications designed for heterogeneous computing systems.

In the current work, we refine the SYCL parallel implementation of MILC-Dslash, further detailing the gradual progression in increasing its granularity of parallelism, going from a simple to a fully parallel implementation (see Section III). The parallel MILC-Dslash implementation reported in the former work [4] refers to just one of the multiple versions presented here. Furthermore, we examine how different parallel strategies, work-item index orders, and work-group sizes affect the MILC-Dslash performance on an NVIDIA A100 GPU (see Section IV). The findings highlight some good practices for optimizing GPU execution performance when programming with SYCL.

The tested parallel strategies include among other things assessing two recent additions to the SYCL ecosystem, SyclCPLX and SYCLomatic. SyclCPLX [6] provides enhanced support for complex numbers and complex number operations which has broad applications in scientific computing. However, in a performance-oriented code, it is important to understand how the use of a more general-purpose library impacts the performance of application kernels. SYCLomatic [7] [8] is a tool for converting CUDA applications to SYCL. The quality of implementations generated by this tool is of central interest. The use of this tool on an application with multiple high-quality "manual" implementations in various programming models allows for a direct assessment of its quality.

QUDA [9], [10] is likely the most popular library for running lattice QCD applications on GPUs. QUDA was initially developed at Boston University for use on NVIDIA GPUs, but it has recently been refactored to support HIP and SYCL back ends for use on AMD and Intel GPUs, respectively. To name just three of its features, QUDA supports gauge field compression, mixed-precision solvers, and auto-tuning to optimize the size of thread blocks and number of blocks launched simultaneously for each kernel. It also supports different quark formulations of lattice QCD, including domain-wall, staggered, twisted-mass, and Wilson-clover. QUDA's staggered approach is adopted here as a reference for comparing the multiple SYCL parallel implementations of MILC-Dslash proposed in this paper.

II. MILC-Dslash Benchmark

MILC-Dslash is a benchmark that is derived from the MILC code [1] which simulates four-dimensional SU(3) lattice-gauge theory, also known as lattice quantum chromodynamics. The Dslash operator, commonly employed in applications solving partial differential equations, represents a finite-difference approach to the partial differential operator found in the four-dimensional Dirac equation. The Dirac equation describes the movement of quarks in the presence of a gluon field. The Dslash operator acts on lattice quark fields, characterized by a three-component complex vector on every site s within a four-dimensional regular grid of size L^4, where $s = 0, \ldots, L^4$, and $L = 32$. Let $B_{j,s}$ and $C_{i,s}$ be quark fields with $i = j = 0, 1, 2$ at site s, then the discrete form of the operator equation $C = \text{Dslash} \times B$ involves a sum of generalized central differences as follows:

$$C_{i,s} = \sum_{k=0}^{3} \sum_{j=0}^{2} \left(U_{i,j,s,k} B_{j,s+\hat{k}} - U^{\dagger}_{i,j,s-\hat{k},k} B_{j,s-\hat{k}} \right), \quad (1)$$

where i and j represent the three color indices, k ranges over the four space-time dimensions, and the Us denote SU(3) matrices — square complex matrices of order three — that parametrize the gluon field. U and $U^{\dagger}$ will hereafter be referred to as U^l, where $l = 0, 1$. The dslash code consists of the more modern and commonly used version, which includes first- and third-nearest neighbor terms. For

implementation purposes, we store fat-links (U) and long-links ($U^\dagger$) along with their respective adjoints, which leads us to have $|l| = 4$ instead of $|l| = 2$.

In summary, the MILC-Dslash kernel essentially comprises the execution of five nested `for` loop structures to compute C. The outcome C vector is the product of multiple independent matrix-vector operations between U matrices and B vector.

III. Implementation Details

This section describes a gradual progression in increasing the granularity of parallelism in the MILC-Dslash kernel over the sections, with each new version building upon the previous one. It starts with coarse-grained parallelism (Section III-A), followed by two stages of medium-grained parallelism (Sections III-B and III-C), until achieving fine-grained parallelism of the MILC-Dslash kernel in Section III-D. For data management, we opted for Unified Shared Memory (USM) device allocations, ensuring explicit control over data movement between host and device memories. All parallel MILC-Dslash versions are written in C/C++ and SYCL, and are available at https://gitlab.com/NERSC/nersc-proxies/milc-dslash/-/tree/SYCL under the LBNL-modified BSD license.

A. One-loop Parallelism — 1LP

The first parallel version of the MILC-Dslash kernel focuses on the parallelism of the outermost loop, which is the most computationally costly loop, with the largest number of independent iterations over the target sites s^*, $s^* = 0, \ldots, \frac{L^4}{2}$, $L = 32$. For *One-loop Parallelism* (1LP), the $|s^*|$ iterations are distributed across $|s^*|$ work-items, i.e. each work-item corresponds to a single target site. Each work-item is responsible for $|l| \times |k| \times |i| \times |j|$ simple multiplication operations, where $|l| = |k| = 4$, and $|i| = |j| = 3$, as illustrated in Fig. 1. The global size is

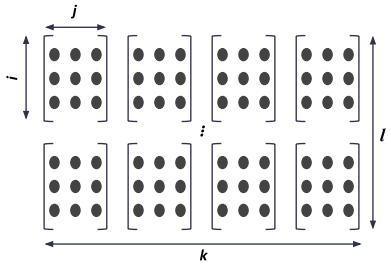


Fig. 1: Each target site s^* has $|l| \times |k|$ SU(3) matrices, $l = k = 0, 1, 2, 3$, and $i = j = 0, 1, 2$.

given by the total number of target sites, which in this case is $|s^*| = \frac{L^4}{2}$. The local size must be defined such that the division of global size by local size is exact, i.e. the number of work-groups is an integer value. The 1LP MILC-Dslash kernel is as follows:

```
1  queue.parallel_for(nd_range<1>{global_size,
   ↪  local_size}, [=](nd_item<1> item) {
2    int global_id = item.get_global_id(0);
3    int s = global_id;
4    for (int l = 0; l < nmat; l++) {
5      for (int k = 0; k < ndim; k++) {
6        // do some calculations
7        for (int i = 0; i < nrow; i++) {
8          for (int j = 0; j < ncol; j++) {
9            // do some calculations
10         }
11         // do some calculations
12       }
13     }
14   }
15 });
```

B. Two-loop Parallelism — 2LP

Two-loop Parallelism (2LP) increases the degree of parallelism by partitioning the i rows in U_k^l matrices among the processing elements of the compute devices, in addition to the target sites. By consequence, the total number of work-items increases by a factor of 3, while the amount of work done by each work-item is reduced by the same factor, where 3 is the number of rows in U_k^l matrices. In other words, there are three work-items per target site. That is, $|l| \times |k|$ rows — one row from each of U_k^l matrices — per work-item, totaling $|l| \times |k| \times |j|$ simple multiplication operations per work-item, as shown in Fig. 2. The global size is given by the product of the

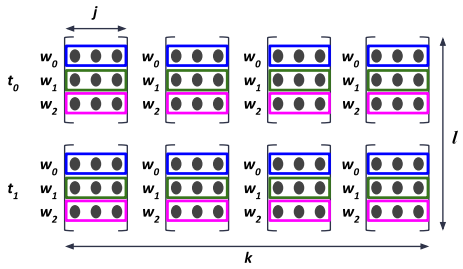


Fig. 2: *Two-loop Parallelism*: three work-items per target site with $|l| \times |k|$ rows, one row from each of U_k^l matrices, per work-item. For illustrative purposes, we assume $|l| = 2$.

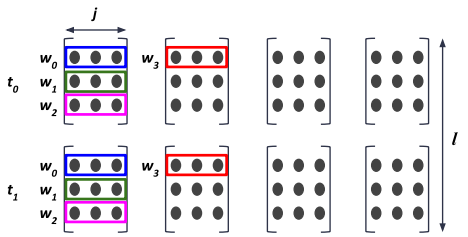
total number of target sites and the number of rows in U matrices, that is $\frac{L^4}{2} \times |i|$. The 2LP kernel is similar to the previous one, except for lines 3 and 7 that are replaced, respectively, by:

```
int s = global_id / nrow;
int i = global_id % nrow;
```

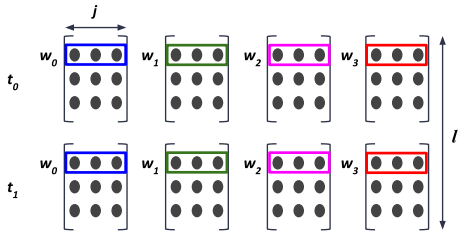
C. Three-loop Parallelism — 3LP

Three-loop Parallelism (3LP) consists of the parallel decomposition of three `for` loop structures: target sites s^*, matrix rows i, and U_k matrices. In this case, there

are 12 work-items per target site — four times more than the previous strategy — totaling $|l| \times |j|$ simple multiplication operations per work-item — four times less than the previous strategy, where four is the number of U_k matrices (see Fig. 3). The global size is given by the



(a) k-major order



(b) i-major order

Fig. 3: *Three-loop Parallelism*: 12 work-items per target site with $|l|$ rows, one row from each of U^l matrices, per work-item. Two different ways to order the work-item indices: k-major order and i-major order. For illustrative purposes, only four work-items are shown in the schematics and we assume $|l| = 2$.

product of the total number of target sites, the number of rows in U matrices, and the number of U_k matrices, that is $\frac{L^4}{2} \times |i| \times |k|$. There are two different ways to order the work-item indices in the parallel execution space: k-major order, work-items grouped by k index; and i-major order, work-items grouped by i index, as shown in Fig. 3.

Unlike the iteration-independent nature of the two previous strategies, the current strategy exhibits a data dependence carried by k-loop due to shared variable. For each target site s^* and matrix row i, their respective four work-items, associate with the four k indices, might require simultaneous access to update the same position of the output vector C.

Three different parallel implementations of the MILC-Dslash kernel are proposed to avoid a data race of shared variables. The three pieces of codes presented below pertain to the k-major order (Fig. 3a) and declare a structure data type named `double_complex`. This structure internally defines two doubles to represent complex numbers, along with arithmetic functions designed for manipulating complex numbers.

The first implementation (3LP-1) uses a local accessor to allocate the shared variable c in the device's local memory (line 2), and calls a barrier function to synchronize the work-items within the same work-group (line 17):

```
1  queue.submit([&](handler &h) {
2    auto c = local_accessor<double_complex,
     ↪  1>(local_size, h);
3    h.parallel_for(nd_range<1>{global_size,
     ↪  local_size}, [=](nd_item<1> item) {
4      int global_id = item.get_global_id(0);
5      int local_id = item.get_local_id(0);
6      int s = global_id / (ndim * nrow);
7      int i = global_id % nrow;
8      int k = (global_id / nrow) % ndim;
9      c[local_id] = {0.0,0.0};
10     for (int l = 0; l < nmat; l++) {
11       // do some calculations
12       for (int j = 0; j < ncol; j++) {
13         // do some calculations
14       }
15       // update c[local_id]
16     }
17     group_barrier(item.get_group());
18     if (k == 0)
19       // update C(i,s)
20   });
21 });
```

For more details of the first implementation in k-major order, the reader is referred to [4, Sections II-B].

The second parallel implementation (3LP-2) also uses local accessor to allocate the shared variable c in the work-group local memory, but uses atomic memory operations to update the output vector C (line 5) at the end of the code, i.e. it replaces lines 17–19 of the first implementation by:

```
1    if (k == 0)
2      // initialize C(i,s)
3    group_barrier(item.get_group());
4
5    atomic_ref<double, memory_order::relaxed,
     ↪  memory_scope::work_group,
     ↪  access::address_space::global_space>
     ↪  c_atomic(C(i,s));
6    c_atomic += c[local_id];
```

Finally, the third implementation (3LP-3) uses atomic operations instead of using local accessor (line 9):

```
1  queue.parallel_for(nd_range<1>{global_size,
   ↪  local_size}, [=](nd_item<1> item) {
2    int global_id = item.get_global_id(0);
3    int s = global_id / (ndim * nrow);
4    int i = global_id % nrow;
5    int k = (global_id / nrow) % ndim;
6    if (k == 0)
7      // initialize C(i,s)
8    group_barrier(item.get_group());
```

```
9     atomic_ref<double, memory_order::relaxed,
    ↪ memory_scope::work_group,
    ↪ access::address_space::global_space>
    ↪ c(C(i,s));
10    for (int l = 0; l < nmat; l++) {
11      // do some calculations
12      for (int j = 0; j < ncol; j++) {
13        // do some calculations
14      }
15      // update atomic variable c
16    }
17  });
```

For the correct computation of C, whose partial sums are temporarily stored in the shared variable c, the size of c, and consequently the local size, must be a multiple of $|i| \times |k| = 3 \times 4 = 12$ for k-major order, and $|k| = 4$ for i-major order. In addition, the remainder of global size upon division by local size must be zero.

The i-major order version (Fig. 3b) of the three parallel implementations differs from the k-major order (Fig. 3a) previously described in this section mainly in the declaration of i and k indices:

```
int i = (global_id / ndim) % nrow;
int k = global_id % ndim;
```

D. Four-loop Parallelism — 4LP

The highest amount of parallelism is achieved by adding the l-loop decomposition to the 3LP strategy, such that the number of work-items per target site is multiplied by $|l|$, and the amount of work per work-item is divided by $|l|$. There are two main approaches to fully parallelize the MILC-Dslash kernel. They differ in the way the work-item indices are oriented in the parallel execution space. In the first implementation (4LP-1), the work-items are first grouped by l index, followed by k index, in the k-major order (Fig. 5a), or i index, in the i-major order (Fig. 5b). The piece of code below refers to the MILC-Dslash kernel of Fig. 5a:

```
1  queue.submit([&](handler &h) {
2    auto c = local_accessor<double_complex,
   ↪ 1>(local_size, h);
3    h.parallel_for(nd_range<1>{global_size,
   ↪ local_size}, [=](nd_item<1> item) {
4      int global_id = item.get_global_id(0);
5      int local_id = item.get_local_id(0);
6      int s = global_id / (ndim * nrow * nmat);
7      int i = global_id % nrow;
8      int k = (global_id / nrow) % ndim;
9      int l = (global_id / (ndim * nrow)) %
   ↪ nmat;
10     if (l == 0)
11       // do some calculations
12     else if (l == 1)
13       // do some calculations
14       ...
15     for (int j = 0; j < ncol; j++) {
16       // do some calculations
17     }
18     // update c[local_id]
19     group_barrier(item.get_group());
20     if (l == 0)
21       // update c[local_id]
22     group_barrier(item.get_group());
23     if (l == 0 && k == 0)
24       // update C(i,s)
25   });
26 });
```

The kernel code of Fig. 5b is similar to the previous one, except for lines 7 and 8 that are replaced, respectively, by:

```
int i = (global_id / ndim) % nrow;
int k = global_id % ndim;
```

In the second implementation (4LP-2), the work-items are first grouped by k index, followed by l index, in the l-major order (Fig. 4a), or i index, in the i-major order (Fig. 4b). The main differences between the codebases of Figs. 4a and 5a concern the k and l declarations:

```
int k = (global_id / (nmat * nrow)) % ndim;
int l = (global_id / nrow) % nmat;
```

On the other hand, the codebases of Figs. 4a and 5b differs in the i, k, and l declarations:

```
int i = (global_id / nmat) % nrow;
int k = (global_id / (nmat * nrow)) % ndim;
int l = global_id % nmat;
```

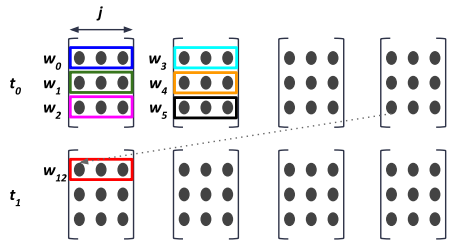
Both implementations contain two synchronization barriers to prevent data race of the shared variable c due to data dependence carried by k- and l-loops. Since the control flow diverges with U^l matrices, all warp threads take the path through the conditional branches (lines 10–14), one branch at a time, with a fraction of the warp threads masked off during the execution of the instructions within each branch. By following the same reasoning as in the 3LP strategy, the local size must be a multiple of $|i| \times |k| \times |l| = 3 \times 4 \times 4 = 48$ for all cases, and the remainder of the division of global size by local size must be zero. The global size is given by the product of the total number of target sites, the number of rows in U matrices, and the number of U_k and U^l matrices, that is $\frac{L^4}{2} \times |i| \times |k| \times |l|$.

IV. Performance Analysis

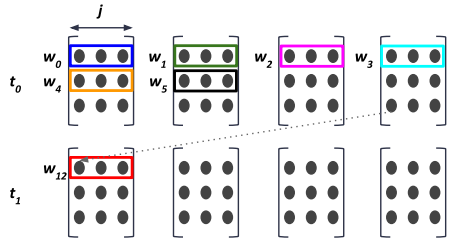
This section focuses on analyzing the impact of the different parallel strategies, work-item index orders, and local sizes on the performance of the MILC-Dslash kernel on an NVIDIA A100 GPU.

A. Computational Environment

The experiments were conducted on a single NVIDIA A100 GPU available on the Perlmutter (PM) supercomputer at the National Energy Research Scientific Computing Center (NERSC) located at Lawrence Berkeley

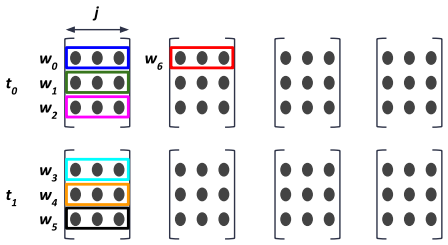


(a) k-major order

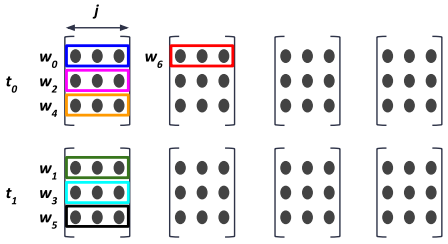


(b) i-major order

Fig. 4: *Four-loop Parallelism*: $|i| \times |k| \times |l|$ work-items per target site with one row per work-item. Two different ways to order the work-item indices: k-major order and i-major order. For illustrative purposes, only seven work-items are shown in the schematics and we assume $|l| = 2$.



(a) l-major order



(b) i-major order

Fig. 5: As Fig. 4, but for two other ways to order the work-item indices in the parallel execution space: l-major order and i-major order.

National Laboratory. NVIDIA A100 GPU is equipped with 40 GB of global memory and a 40 MB L2 cache for the entire GPU, along with 108 compute units. Each compute unit has 192 KB of shared L1 cache and local memory, with a maximum of 2,048 processing elements and 65,536 registers. It accommodates work-group sizes of up to 1,024 work-items, organized into warps of 32 work-items each.

The SYCL MILC-Dslash implementations were compiled using Intel oneAPI DPC++/C++ compiler 2024.1.0, with the optimization flag (`-O3`) enabled, and NVIDIA driver version 525.105.17. The version of the SYCLomatic tool was the release of October 16, 2023, available at https://github.com/oneapi-src/SYCLomatic; and the SyclCPLX library was based on commit 32684c7 from August 4, 2023, available at https://github.com/argonne-lcf/SyclCPLX. The CUDA implementation was compiled with CUDA 12.2 with the optimization flag (`-O3`) also enabled.

B. Parameters and Methodology

The performance analysis is carried out by evaluating the execution time of the SYCL MILC-Dslash kernel on a single NVIDIA A100 GPU. The mean kernel runtime is determined from a sample of 10 runs using the `clock_gettime()` function with the `CLOCK_MONOTONIC` option; each run comprises 100 kernel iterations and 1 warmup iteration. The monotonic clock guarantees that the runtime values provided can only increase over the course of its operation. A double-precision floating-point arithmetic has been employed to represent the complex matrices. The theoretical value of 600.8 million floating-point operations (FLOP) was adopted to compute performance in terms of FLOP/s.

The local sizes or work-group sizes have been defined to adhere not only to the parallel strategy constraints outlined in Section III but also to device constraints, that includes: not exceeding the maximum limit of 1,024 work-items per work-group, and being a multiple of warp size. For instance, the local sizes of 3LP-1, 3LP-2, 3LP-3, 4LP-1, 4LP-2 in k-major order that follow all established restrictions are: 96, 192, 384, and 768.

The Nsight Compute profiling tool [11] was used to provide additional information regarding the performance differences between parallel strategies and work-item index orders for the MILC-Dslash kernel on an NVIDIA A100 GPU, with a local size of 768 (256 for 1LP).

C. 3LP-1 versions

Given that 3LP-1 will end up being the best parallel strategy, as we will see in the next section, five additional implementations have come from minor changes in the original 3LP-1 version described in Section III-C:

1) The `double_complex` data structure, which is defined internally to represent complex numbers, is replaced by the SYCL complex library [6,

SyclCPLX], which means among other things that line 2 is rewritten as follows:

```
auto c = local_accessor<
↪  sycl::ext::cplx::complex<double>,
↪  1>(local_size, h);
```

2) A CUDA version was posteriorly developed to compare the CUDA and SYCL performances.

3) The CUDA version is compiled with the maximum number of registers per thread (`-maxrregcount`) set to 64.

4) A version provided by the SYCLomatic tool [7] [8] to migrate MILC-Dslash kernel automatically from CUDA to SYCL.

5) The SYCLomatic version after simplifying the expression to identify the work-item's global index. The expression in the original SYCLomatic version is given by:

```
int global_id = item.get_local_range(2) *
↪  item.get_group(2) +
↪  item.get_local_id(2);
```

It has been replaced with a call to the `get_global_id()` function:

```
int global_id = item.get_global_id(2);
```

D. Results

Fig. 6 shows the performance in terms of GFLOP/s achieved by the multiple SYCL implementations of MILC-Dslash on a single NVIDIA A100 GPU. For better data visualization and comprehension, y-axis is divided into two different scales that share the same x-axis. Each parallel strategy comprises its two respective work-item index orders, identified by filled and empty markers, except for 1LP and 2LP, which have only one work-item index order. Each color and marker represents a specific local size or work-group size, as shown by the legend on the right. Gray shaded area highlights the five additional 3LP-1 implementations. QUDA's `staggered_dslash_test` (gray horizontal dashed line) is plotted as a benchmark reference value (633.7 GFLOP/s). Furthermore, the profiling analysis using the Nsight Compute tool is presented in Table I.

1) Overview: According to Fig. 6, the performance (GFLOP/s) increases as the degree of parallelism increases from 1LP to 3LP-1, and thereafter it gradually decreases for 3LP-3, 3LP-2, 4LP-1, and 4LP-2. Table I shows that increasing the number of work-items from $\frac{L^4}{2}$ in 1LP to $|i| \times |k| \times \frac{L^4}{2}$ in 3LP-1 (row 2), along with the local memory usage by 3LP-1 (row 9), led to better utilization of GPU compute resources (row 3), with higher device occupancy (row 4), lower L1 cache miss rate (row 7), and fewer L1 tag requests by global memory (row 10). Consequently, 3LP-1 improved performance by approximately 2 times compared to 1LP. Given that the MILC-Dslash performance is bounded by memory bandwidth, this represents about 8% of the empirical peak performance of 7.6 TFLOP/s on A100 for 3LP-1, in contrast to just 4% for 1LP (row 5).

2) 3LP: Although 3LP-1, 3LP-2 and 3LP-3 share the same global size, they deal with the race condition in slightly different manners, leading to differences in performance. The 3LP-1 implementation uses work-group local memory for a collective update after a group wide barrier, while 3LP-2 atomically updates the shared variable, keeping everything else the same. Meanwhile, 3LP-3 does not use local memory at all and instead atomically updates a global variable. Despite the larger number of shared memory[1] bank conflicts (row 12), 3LP-1 outperforms the other two strategies due to the absence of atomic operations, which allows the GPU's latency-hiding nature to dominate. On the other hand, 3LP-2 and 3LP-3 have hundreds of work-items within the same work-group competing for an atomic region, which may result in a performance decrease of up to 8.4% and 7.4%, respectively. The peak performance is observed with the 3LP-1 strategy in its different variations: SYCL, SyclCPLX, SYCLomatic optimized, and CUDA, with particular emphasis on the last two versions, as will be explained in Section IV-D6.

3) QUDA: The QUDA library (1.1.0) [9] has been used for large scale lattice QCD applications running on NVIDIA GPUs. It has recently been extended to include support for HIP and SYCL in addition to CUDA. QUDA contains many advanced features including autotuning, compression of gauge fields (reducing required memory bandwidth), and mixed precision solvers, to name a few. The MILC-Dslash code using QUDA library was adopted here as a reference value for comparison with the parallel strategies proposed in this paper. The benchmark we use for comparison, `staggered_dslash_test`, does not include QUDA's gauge compression options as that is not a current feature of our SYCL implementation. If we run `staggered_dslash_test` without compression (option recon 18) the speed is 634 GFLOP/s. Using recon 12 and 9, the speed is 728 GFLOP/s and 825 GFLOP/s, respectively. As we can see in Fig. 6, all variations of the 3LP-1 strategy outperform the QUDA version without compression (gray horizontal dashed line), with a maximum improvement of 10.2%.

4) 3LP-1 CUDA: SYCLomatic optimized and CUDA baseline achieved equivalent performances. Moreover, a performance improvement of up to 3.6% after CUDA compiler optimization for register allocation reveals the importance of preventing the CUDA compiler from allocating either too many or too few registers. Beyond the results presented in Fig. 6, Dufek et al. [4] compared the 3LP-1 performance in k-major order ported to three parallel programming models: SYCL, CUDA, and

[1]Both local memory and shared memory are terms used to refer to data storage specific to a work-group.

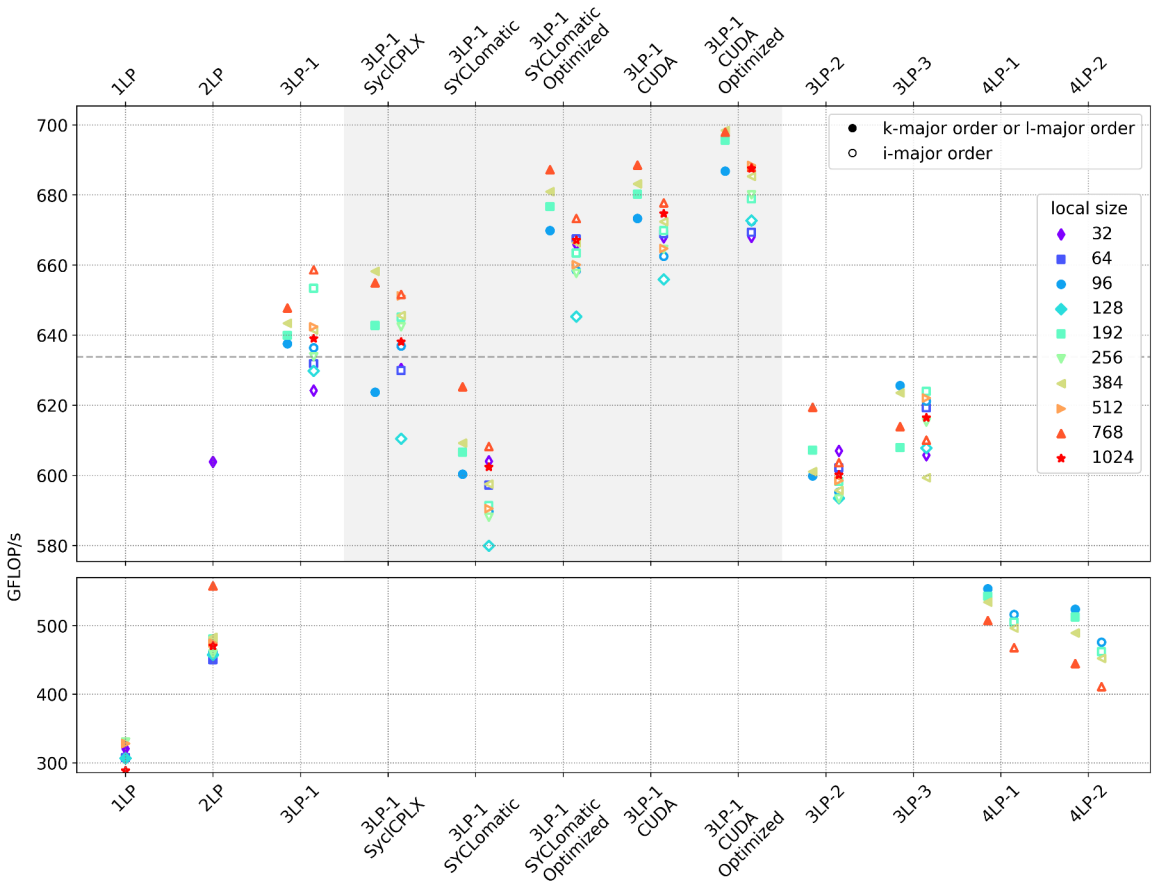


Fig. 6: Performance in terms of GFLOP/s achieved by the multiple SYCL implementations of MILC-Dslash on a single NVIDIA A100 GPU. For better data visualization and comprehension, y-axis is divided into two different scales that share the same x-axis. Each parallel strategy comprises its two respective work-item index orders, identified by filled and empty markers, except for 1LP and 2LP, which have only one work-item index order. Each color and marker represents a specific local size, as shown by the legend on the right. Gray shaded area highlights the five additional 3LP-1 implementations. QUDA's `staggered_dslash_test` (gray horizontal dashed line) is plotted as a benchmark reference value (633.7 GFLOP/s).

Kokkos, when running on GPU devices from two different vendors (NVIDIA and Intel). SYCL, CUDA, and Kokkos performed similarly on both devices.

5) 3LP-1 SyclCPLX: The SyclCPLX version reports positive and negative performance differences below 3% with 3LP-1, and these results may vary with system updates over time and across different systems and compilers, making them non-generalizable.

6) 3LP-1 SYCLomatic: In addition to the straightforward use of the SYCLomatic tool, it is worth noting that its optimized version exhibits a better performance than 3LP-1 for all local sizes. For instance, its performance value is 6.1% superior to that obtained by 3LP-1 when set up with a local size of 768 and k-major order. Unlike the default out-of-order queue in 3LP-1, the SYCLomatic tool explicitly creates an in-order SYCL queue, explaining the performance advantages ranging from 1.5% to 6.7% in favor of the SYCLomatic optimized version. A similar performance gain is obtained by non-optimized CUDA, where the CUDA stream also follows in-order semantics. These results indicate that out-of-order semantics might lead to performance loss attributed to scheduling overheads involved in managing multiple tasks and their dependencies, particularly when there is no opportunity for overlapping tasks [12]. The differences in execution times of the SYCLomatic version before and after optimization emphasize the importance of using the direct primary work-item indexing function to obtain information about the parallel index space, instead of deriving it from a combination of other functions, whenever possible. Otherwise, performance may be 10.0–12.2%

TABLE I: Profile information collected using the Nsight Compute tool for a single execution of the MILC-Dslash kernel (specifically, the second kernel launch) on NVIDIA A100 GPU, using a local size of 768 for all parallel strategies and work-item index orders, except for 1LP, which used 256. The exact metric names corresponding to rows 10, 11, and 12 are, respectively: `memory_l1_tag_requests_global`, `memory_l1_wavefronts_shared`, and the difference between `memory_l1_wavefronts_shared` and `memory_l1_wavefronts_shared_ideal`.

Description	1LP	2LP	3LP-1		3LP-2		3LP-3		4LP-1		4LP-2	
			k	i	k	i	k	i	k	i	l	i
1 - Duration (μs)	1821.3	1078.6	929.2	912.9	971.5	996.4	981.3	988.6	1187.3	1287.8	1353.5	1463.8
2 - Work-items (global size)	0.5M	1.6M	6.3M	6.3M	6.3M	6.3M	6.3M	6.3M	25.2M	25.2M	25.2M	25.2M
3 - Compute (SM) throughput (%)	4.4	11.0	12.7	12.9	10.8	11.2	10.2	10.6	30.6	27.9	34.2	27.9
4 - Achieved occupancy (%)	47.6	72.7	74.0	73.7	70.3	70.7	66.3	66.5	72.0	72.2	72.3	72.4
5 - Peak performance (%)	4	7	8	8	8	7	7	7	6	5	5	5
6 - L1/TEX cache throughput (%)	55.5	60.4	59.1	76.7	56.6	69.2	59.0	73.3	66.6	77.9	72.5	82.5
7 - L1/TEX miss rate (%)	37.4	31.9	26.9	25.4	28.7	26.3	32.6	30.7	24.0	23.0	23.5	22.9
8 - L2 miss rate (%)	31.2	38.6	51.1	49.8	47.1	47.3	42.5	41.9	56.9	57.5	56.3	57.2
9 - Dynamic shared memory per work-group (Kbyte/work-group)	0	0	12.3	12.3	12.3	12.3	0	0	12.3	12.3	12.3	12.3
10 - L1 tag requests global (sectors)	190M	121M	86M	101M	87M	101M	89M	103M	120M	140M	123M	124M
11 - L1 wavefronts shared (sectors)	0	0	4.7M	7.9M	1.6M	1.6M	0	0	21.0M	25.2M	26.2M	46.1M
12 - Excessive L1 wavefronts shared (sectors)	0	0	2.4M	5.5M	0.8M	0.8M	0	0	8.4M	12.6M	11.0M	30.9M
13 - Avg. divergent branches	0	0	0	0	0	0	0	0	5,461	5,461	7,281	7,281

less efficient. It may suggest that the mapping of work-item indices to data varies with the indexing functions employed, resulting in a more localized memory access pattern in the first case. Three other SYCLOMATIC versions were also examined: (i) one-dimensional instead of three-dimensional parallel index space, (ii) pass an explicit local memory fence argument to the barrier function (`sycl::access::fence_space::local_space`), and (iii) removal code processing error codes (`DPCT_CHECK_ERROR` and `CUCHECK`), but they do not affect performance.

7) Work-item index order: Let the U matrices be organized as $|l|$ arrays of $|i| \times |j|$ double-precision complex matrices, each array with a size of $L^4 \times |k|$. For a GPU cache line of 128 bytes, k- and i-major orders (Figs. 3 and 4) can merge requests from five and two consecutive work-items to the same cache line, respectively. In the former, all data in the cache line is utilized, while in the latter, 10 out of 16 8-byte words may remain unused. This results in a higher degree of memory coalescing and fewer memory transactions for k-major order, with a constant gap of two 8-byte words between two adjacent work-items, as opposed to eight for i major order. Although k-major order outperforms i-major order in 31 out of 36 cases, the performance differences are below 3% for most cases, except for 4LP-1, where the values range from 7.2% to 8.5%. As we can see in Table I (row 10), k-major order shows significant fewer L1 tag requests by global memory compared to i-major order, indicating better data locality that results in more coalesced memory accesses. Additionally, the number of excessive wavefronts in L1 from shared memory points to a higher number of bank conflicts in the shared memory when using the i-major order for both 3LP-1 and 4LP-1 (row 12). The l- and i-major orders relative to 4LP-2 will be discussed in Section IV-D8 below.

8) 4LP: Although 4LP presents greater concurrency than 3LP, it does not translate into higher performance. A quick look at Table I reveals large numbers of shared memory bank conflicts (row 12) and L1 tag requests (row 10), suggesting poor memory coalescing. Regardless of the better performance of k- and l-major orders, the overall metrics still indicate suboptimal resource utilization compared to 3LP. Another important factor that negatively impacts the parallel efficiency of 4LP is the large number of divergent branches (row 13). This results in only $|i| \times |k| = 12$ work-items in a 32-wide warp being active at any given time, concurrently executing the same SIMD instructions, while the remaining work-items and associated computing resources are in an idle state. Besides the aforementioned reasons, the 4LP strategy is generally unsuitable for the MILC-Dslash benchmark because it requires two synchronization barriers, a large number of work-items launched, and consequently, the creation of many warps. Since each warp takes a long time to complete due to increased bank conflicts and divergence, warp stalling occurs, thus reducing the opportunities for the GPU to exploit its latency-hiding capabilities. For instance, 4LP-1 shows a performance decline of 13.2–29.0% compared to 3LP-1, and it is almost equivalent to 2LP (Fig. 6). Unlike 4LP-1 with 12 consecutive active work-items (Fig. 4), the distribution of the 12 active work-items in 4LP-2 (Fig. 5) is intercalated with inactive work-items. In the case of l-major order, it alternates between a sequence of three active work-items and three inactive work-items, while i-major order alternates between one active work-item and

one inactive work-item. By consequence, l-major order outperforms i-major order by 8.2–11.0% for all local sizes in 4LP-2. It can further be observed in Fig. 6 that the 4LP performance degrades as the cluster size of active work-items within a warp decreases, with 4LP-2 in i-major order even underperforming 2LP by 3.9–26.3% in 3 out of 4 local sizes. The optimal work-item index order (Fig. 4a) can lead to performance improvements of 16.3–23.4% over the worst-performing one (Fig. 5b). The findings indicate that optimizing GPU execution performance involves maximizing not only the number of active work-items per warp but also the sequence of successive active work-items, thereby reducing branch divergence and increasing cache reuse.

9) Local size: Fig. 6 reveals that performance exhibits minimal variance with local size, albeit with a higher local size variability for 2LP. There is no unanimous consensus regarding the optimal local size across all SYCL implementations. However, the peak performance was obtained when configuring 3LP-1 in its different variations (SYCL, SyclCPLX, SYCLomatic, and CUDA) with a local size of 768. The performance differences between the optimal and suboptimal local sizes can vary from 1.6% to 34.2%, depending on the parallel strategy and work-item index order.

It is worth mentioning that the above-mentioned results are subject to changes based on the architecture and the choice of the following parameters: number of sites (L), matrix size (i and j), and number of matrices (k and l).

V. Conclusion

This study has explored the performance of different parallel strategies in the MILC-Dslash benchmark on a GPU using SYCL. Some specific features that emerge from these parallel strategies include atomic memory operations, shared variables, divergent instructions, synchronization barrier, and dependencies between iterations. Additionally, we evaluated the SYCL complex library (SyclCPLX) and the SYCLomatic conversion tool.

The SyclCPLX library has demonstrated its efficacy in simplifying coding. Similarly, the SYCLomatic tool has been acknowledged for its effectiveness in the present case. However, ongoing efforts towards code optimization are essential to address any inefficiencies and maximize its utility.

The comparative analysis of SYCLomatic performance before and after optimization underscores the significance of specific adjustments. For instance, employing the direct primary work-item indexing function in SYCLomatic was an important step in improving efficiency, otherwise incurring a penalty of approximately 12%. Our results illuminate the impact of such modification on overall performance, emphasizing the need for strategic code alterations.

Furthermore, an examination of the 3LP strategy shows that non-atomic memory operations, an in-order queue approach, and optimal work-item index order and local size setup collectively yield a performance improvement of about 15%. The peak performance is achieved by 3LP-1, which provide a 2x speedup over 1LP and a 10% improvement compared to the QUDA benchmark reference, thanks to enhanced parallelism and the use of work-group local memory.

The analysis of the 4LP strategy highlights some important lessons. It is often assumed that higher concurrency yields better performance. However, this study suggests that better utilization of GPU resources can be achieved even at the expense of concurrency. The benchmark under consideration is memory-bound and therefore did not benefit from the increased concurrency provided by 4LP. Instead, this approach led to suboptimal utilization of GPU's memory hierarchy and required frequent synchronizations and branching, which resulted in warp stalling.

In summary, our investigation sheds light on effective parallel strategies for the MILC-Dslash kernel, including also the use of a complex numbers library (SyclCPLX) and an automatic code conversion tool from NVIDIA CUDA to SYCL (SYCLomatic). The findings could contribute valuable insights into optimize not only parallel computing applications in general but also the QUDA library, which is widely used for lattice QCD applications in production environments, offering guidance for researchers and developers in their pursuit of more efficient implementations. As the ecosystem of massively parallel programming languages continues to evolve, ongoing refinement and optimization of parallel strategies, tools and libraries are crucial for maximizing performance in diverse applications.

Acknowledgment

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231.

References

[1] "MILC code for lattice QCD calculations," 2024. [Online]. Available: https://github.com/milc-qcd/milc_qcd

[2] F. Gross *et al.*, "50 Years of Quantum Chromodynamics," *Eur. Phys. J. C*, vol. 83, p. 1125, 2023.

[3] B. Joó and F. T. W. Robert G. Edwards, "Lattice quantum chromodynamics and chroma," in *Exascale Scientific Applications: Scalability and Performance Portability*, T. J. W. Tjerk P. Straatsma, Katerina B. Antypas, Ed. New York: Chapman and Hall/CRC, 2017.

[4] A. S. Dufek, R. Gayatri, N. Mehta, D. Doerfler, B. Cook, Y. Ghadar, and C. DeTar, "Case study of using kokkos and sycl as performance-portable frameworks for milc-dslash benchmark on nvidia, amd and intel gpus," in *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2021, pp. 57–67.

[5] J. Reinders, B. Ashbaugh, J. Brodman, M. Kinsner, J. Pennycook, and X. Tian, *Data Parallel C++: Programming Accelerated Systems Using C++ and SYCL*. Apress, 2023.

[6] T. Applencourt, B. Videau, J. Le Quellec, A. Dufek, K. Harms, N. Liber, B. Allen, and A. Belton-Schure, "Standardizing complex numbers in sycl," in *Proceedings of the 2023 International Workshop on OpenCL (IWOCL)*, 2023.
[7] Intel Corporation, "SYCLomatic Documentation," 2024. [Online]. Available: https://oneapi-src.github.io/SYCLomatic
[8] ——, "SYCLomatic: A New CUDA-to-SYCL Code Migration Tool," 2024. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/syclomatic-new-cuda-to-sycl-code-migration-tool.html
[9] M. Clark, R. Babich, K. Barros, R. Brower, and C. Rebbi, "Solving lattice qcd systems of equations using mixed precision solvers on gpus," *Computer Physics Communications*, vol. 181, no. 9, pp. 1517–1528, 2010.
[10] "Quda library for performing calculations in lattice qcd on gpus," 2024. [Online]. Available: https://github.com/lattice/quda
[11] "NVIDIA Nsight Compute Profiling Tool," 2024. [Online]. Available: https://docs.nvidia.com/nsight-compute/NsightCompute
[12] L. Crisci, L. Carpentieri, P. Thoman, A. Alpay, V. Heuveline, and B. Cosenza, "Sycl-bench 2020: Benchmarking sycl 2020 on amd, intel, and nvidia gpus," in *Proceedings of the 12th International Workshop on OpenCL and SYCL*, ser. IWOCL '24. New York, NY, USA: Association for Computing Machinery, 2024.