Performance portability of a Wilson Dslash Stencil Operator Mini-App using Kokkos and SYCL

Bálint Joó*, Thorsten Kurth†, M. A. Clark‡, Jeongnim Kim§,
Christian R. Trott¶, Dan Ibanez¶, Dan Sunderland¶, Jack Deslippe†
*Jefferson Lab
Newport News, VA, U.S.A
Email: bjoo@jlab.org

†NERSC
Berkeley, CA, U.S.A
Email: tkurth@lbl.gov, jrdeslippe@lbl.gov

‡NVIDIA
San Jose, CA, U.S.A
Email: mclark@nvidia.com

§Intel Corporation
Hillsboro, OR, U.S.A
Email: jeongnim.kim@intel.com

¶Sandia National Laboratories
Albuquerque, NM, U.S.A
Email: crtrott@sandia.gov, dsunder@sandia.gov, daibane@sandia.gov

***Abstract*—We describe our experiences in creating mini-apps for the Wilson-Dslash stencil operator for Lattice Quantum Chromodynamics using the Kokkos and SYCL programming models. In particular we comment on the performance achieved on a variety of hardware architectures, limitations we have reached in both programming models and how these have been resolved by us, or may be resolved by the developers of these models.**

***Index Terms*—Portability, Performance, Kokkos, SYCL , Lattice QCD, Wilson Dslash**

I. Introduction

While performance portability has always been desirable, applications could often get away with maintaining multiple codebases to target multiple architectures. A common approach to targeting multiple systems in numerical lattice quantum chromodynamics (Lattice QCD or LQCD) (e.g. [1]) has been to utilize a layered software design and access crucial algorithms (primarily linear solvers) implemented in libraries optimized directly to the underlying hardware. An example is the *Chroma* code [2] which uses a data-parallel layer called *QDP++* which has a canonical CPU reference implementation as well as versions of it [3] which provide better vectorization on CPUs and run on NVIDIA GPUs utilizing JIT compilation from the LLVM compiler suite [4]. In turn high-performance solvers are accessed from architecture-specific libraries such as QUDA [5]–[7] for NVIDIA GPUs and the QPhiX and MG-Proto libraries for Intel CPUs [8]–[10].

With the recent announcement of the next round of pre-exascale and exascale systems in the United States, the strategy of maintaining an Intel AVX multicore CPU and NVIDIA GPU code stacks and libraries to cover most bases will no longer be successful, since the three announced systems will feature different accelerators. The Aurora system to be installed in Argonne Leadership Computing Facility (ALCF) will feature Intel X^e technology, the Frontier System at Oak Ridge Leadership Computing Facility (OLCF) will be powered by GPUs from AMD Corporation, while the Perlmutter System at NERSC will utilize NVIDIA GPUs. Maintaining separate code stacks to cover all the architectures would rapidly outpace available manpower efforts. As such, performance portable approaches are crucial in order to successfully target all of these systems.

In this paper we present Mini-Apps for LQCD [11], [12] implementing the Wilson Dslash Operator, developed in the Kokkos [13] programming model and ported to the SYCL programming model [14] which, in turn, will form the basis

This research is funded by the Exascale Computing Project and the Scientific Discover through Advanced Computing program of the U. S. Department of Energy, by the Offices of Advanced Scientific Computing Research (ASCR) and Nuclear Physics (NP). Authored by Jefferson Science Associates, LLC under U.S. DOE Contract No. DE-AC05-06OR23177. The U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce this manuscript for U.S. Government purposes.

978-1-7281-6003-0/19/$31.00 ©2019 IEEE
DOI 10.1109/P3HPC49587.2019.00007

of Data Parallel C++ in the OneAPI effort from Intel. The Wilson-Dslash kernel is a particular nearest-neighbor stencil operator which is used at the heart of iterative solvers in LQCD calculations. In order for a programming model to be viable for encoding lattice QCD applications, being able to develop an efficient single process Wilson-Dslash kernel using the model is a minimum requirement in our view.

Our primary contributions are as follows: we provide open source, publicly available mini-apps for Wilson-Dslash in both Kokkos and SYCL for the future use of the community. In addition by gathering performance data on a variety of hardware platforms we give some idea of performance portability attained using these models as regards these mini-apps specifically.

This paper is structured as follows: in Sec. II we will detail some relevant background such as the structure of the Wilson-Dslash operator and recap a basic performance model for the algorithm. We will also outline briefly the features of the Kokkos and SYCL programming models relevant to this paper. Finally we will also touch on some related work of which we are aware. In Sec. III we will discuss the salient features of our Kokkos implementation including numerical results, and we will consider the SYCL implementation in Sec. IV. We will discuss our experiences further in Sec. V and conclude in Sec. VI.

II. Background

A. Wilson Dslash Operator

The Wilson-Dslash operator is a finite-difference operator used in Lattice QCD to describe the interactions of quarks with gluons. Let us consider space-time as being discretized onto a 4-dimensional hypercubic lattice, the sites of which will be labeled by index x. The quarks are denoted by a complex-valued spinor field $\psi^{\alpha a}(x)$ where $\alpha \in [0,3]$ are the the so called *spin*-indices, and $a \in [0,2]$ are the *color*-indices. The gluons are represented by the gauge matrices $U_\mu^{ab}(x)$ which are ascribed to the links of the lattice emanating from site x in space-time direction $\mu \in [0,3]$. The U are members of the gauge group $SU(3)$ and are represented as unitary 3×3 complex matrices with respect to color indices and have unit-determinant. With these basic definitions the Wilson-Dslash operator is defined as (suppressing spin and color indices):

$$\not{D}\psi(x) = \sum_{\mu=0}^{3} P_\mu^- U_\mu(x)\psi(x+\hat{\mu}) + P_\mu^+ U_\mu^\dagger(x-\hat{\mu})\psi(x-\hat{\mu}) \quad (1)$$

where $\not{D}$ is the so called "D-slash" operator acting on the spinor at site x. In the definition above, $P_\mu^\pm = \frac{1}{2}(1 \pm \gamma_\mu)$ are spin-projection matrices (matrices in spin-space) with γ_μ being members of a Dirac algebra, and $\psi(x \pm \hat{\mu})$ refers to the value of ψ from the neighboring sites in the forward/backward μ direction.

A common trick in evaluating $\not{D}\psi$ is to note that the result of applying $P^\pm$ to $\psi(x)$ has only two independent spin components. As a result the matrix multiplication in the color degrees of freedom needs to be carried out only twice, rather than four times. A basic algorithm for the Wilson-Dslash operator is given in Fig. 1, where the spin-projection trick is indicated by the calls to `projectPlus()`, and `projectMinus()` to project onto a *half spinor* (a spinor with index $\alpha \in [0,1]$). Once the matrix-vector multiplications are carried out (lines 14 & 23 of Fig. 1), the four components in the original spin basis are restored with `reconstructPlus()` and `reconstructMinus()` respectively. For more details we refer the reader to e.g. [1], [5], [15]. It is fairly standard to pack the gauge fields when setting up the Dslash operator, so that they can be accessed efficiently (e.g. in a uniform access fashion on CPUs or with coalesced access on GPUs).

```
1  void dslash( Spinor out[Nsites], const Gauge u_in[Nsites][4][2],
2                    const Spinor s_in[Nsites] )
3  {
4   forall(int site=0; site < Nsites; ++site) {
5     HalfSpinor tmp1,tmp2;
6
7     out[site] = 0; // Initialize
8
9     for(int mu=0; mu < 4; ++mu) { // XYZT directions
10
11       // P^{-}_{mu} U_mu(x) psi( x + hat(mu));
12       tmp1 = projectMinus(mu, neighborPlus(site, mu, s_in));
13       for( spin=0; spin < 2; ++spin) {
14         // matrix multiply in color space
15         tmp2[spin] = u_in[site][mu][FORWARD] * tmp1;
16       }
17       out[site] += reconstructMinus(mu,tmp2);
   // accumulate
18
19       // P^{+}_{mu} U^\dagger psi( x - hat(mu))
20       tmp1 = projectPlus(mu, neighborMinus(site, mu, s_in));
21       for( spin=0; spin < 2; ++spin) {
22
23         // matrix multiply in color space
24         tmp2[spin] = adj(u_in[site][mu][BACKWARD]) * tmp1;
25       }
26       out[site] += reconstructPlus(mu,tmp2);
   // accumulate
27     } // mu
28   }// site
29 }
```

Fig. 1. Pseudocode for Wilson-Dslash Operator

B. Basic Performance Model

Not counting sign flips or multiplications by $\pm i$, the arithmetic carried out per output site is $8 \times (2 \times 66 + 12) + 7 \times 24 = 1320$ floating point operations (FLOP). The 12 FLOP term is the arithmetic cost of the spin-projection operation (6 complex adds), the factor of 66 FLOPs is the cost of a single complex 3×3 matrix by 3-vector multiplication of which we must do two for each spinor. The factor of 8 comes from having to do the above for all 4 forward and backward directions in space-time and the 7×24 FLOPs term is the cost of summing 8 complex 12-component vectors. In terms of data movement we must read in the 8 neighboring spinors ($24F$ Bytes (B) each where F is the floating point size), the 8 gauge links ($18F$ B each) and write out the result ($24F$ B). If we do not have non-temporal writes, we must bring the output into cache first which may cost an additional $24F$B of reading. However, some of the spinors read may be reused from cache (there is no reuse of the gauge fields in our scheme). If we denote by R

the number of reusable neighbors a simple model of arithmetic intensity is [16]:

$$A = \frac{1320}{(8 \times 18 + 24(8 - R) + 24r + 24)F} \text{ FLOP/Byte} \quad (2)$$

where the $(8 - R)$ term reflects reuse and $r = 0$ ($r = 1$) if the output is written without (with) reading first. The naive value of A with no reuse, for 32-bit floating point numbers with $r = 0$ is $A = 0.92F/B$. If due to some clever lattice traversal scheme such as 3-1/2 dimensional blocking with scanlines [16], or tiling [8] or the use of space filling curves [17]–[20], one can achieve that 7 out of 8 neighbors are reused the arithmetic intensity can be as high as $A = 1.72F/B$. While we did not employ it here, one can also employ *gauge compression* as in [5], [9], for example by storing only 2 rows of the gauge field and reconstructing the 3rd on the fly using properties of $SU(3)$. If we do not count the decompression FLOP-s, such 12 compression can push the intensity as high as $A = 2.29F/B$ in single precision. On architectures such as NVIDIA GPU and Intel Xeon Phi Knights Landing (KNL) as well as most currently available CPUs, $A = 1.72F/B$ indicates that our kernels will still be *memory bandwidth bound*, leaving plenty of "free" FLOPs to carry out the compression/reconstruction.

C. SIMD Parallelism

Each output site can be computed independently of the others, and the main source of parallelism is over the lattice sites. However to fully utilize the available hardware we may need to utilize SIMD vector registers such as AVX2 and AVX512 on Intel Xeon and Xeon Phi CPUs. Further, on accelerators such as the ones from NVIDIA, thread warps may also be treated in a SIMD manner. In this work we consider exploiting SIMD in two ways: first we consider (in our Kokkos implementation) a Dirac operator acting on multiple vectors simultaneously. This is referred to as a 'multi right hand side' (MRHS) approach. We evaluate

$$\chi_i = \not{D}(U)\psi_i \quad (3)$$

reusing the same gauge field U for all the ψ_i. This is a trivial kind of SIMD parallelism and is straightforward to implement.

The second approach we consider is described in [20], [21] called Virtual Node SIMD. The idea of this approach is that the SIMD lanes can be considered as virtual computer processors, arranged in a hypercubic topology. A SIMD register which can accommodate 2^d complex numbers can be considered as a four dimensional virtual node grid (VNG) with extent 2 in d of its dimensions and extent 1 in the others. We can lay out the original lattice on the VNG, by dividing the extent of each of its dimensions by the corresponding dimension of the VNG, yielding a SIMD-ized lattice with a SIMD vector of 2^d fields on each site and link. Fields on the SIMD-ized (outer) lattice can be indexed by their corresponding site x_o and the lane index x_l within the (outer) site, which can be computed from the original x by using division by 2 and modulo operations. We illustrate the scheme in Fig. 2. In

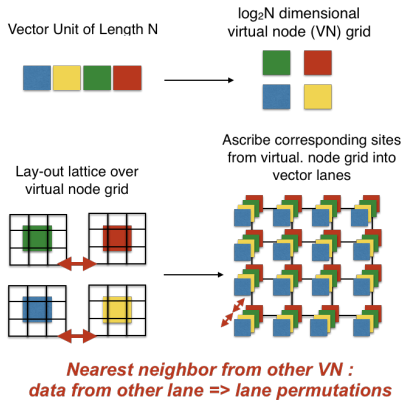

Fig. 2. The Virtual Node Scheme for SIMD-ization

this approach most lattice-wide arithmetic operations can be carried out by replacing the previous scalar arithmetic with identical arithmetic using SIMD vectors. An exception to this is dealing with neighboring sites when wrapping around the edge of the SIMD-ized lattice. As can be seen in Fig. 2 accessing a neighbor in such a case moves it from a neighboring virtual node to ours and such a movement between virtual nodes corresponds to a shuffle between the vector lanes. In our *Dslash* implementation, when we compute the index of our neighbor spinor, we can simultaneously set a flag as to whether such a permutation is needed. When the field vector from the neighboring (outer) site is loaded for spin projection, the necessary lane permutation swizzle can be carried out. It is also possible to define load-swizzle operations as suggested to us by P. Boyle from his implementation in the Grid [20] LQCD code, where threads corresponding to vector lanes can load data from suitably swizzled addresses within a warp while maintaining coalescing. As a final note, we mention that the method can be adapted to SIMD lengths longer than 16 complex numbers while maintaining a 4-dimensional lattice. In this case we extend the dimensions of the virtual node grid appropriately. For example, a SIMD length of 32 complex numbers could be accommodated with a $2^3 \times 4$ virtual node grid, and a SIMD length of 64 could be accommodated using, say a $2^2 \times 4^2$ virtual node grid and so forth.

D. Kokkos

Kokkos [13], [22] is a programming model developed at Sandia National Laboratories to enable performance portability for C++ programs. It is developed in Modern C++ and offers a conceptual model of a compute node as a collection of Memory and Execution spaces. Parallel kernels are encoded as C++ functors or *lambda expressions* and can be dispatched using constructs controlled by policies. Kokkos offers *parallel_for*, *parallel_reduction* and *parallel_scan* dispatches as well as a tasking interface based on asynchronous task handles known as *futures*. The constructs can be bound to execution spaces and otherwise customized using policies, for example *parallel_for* can be run on a CPU with OpenMP or a GPU

with CUDA by selecting the appropriate execution space policy. The iteration space is also controlled by policies, for example a *forall* can be executed over a simple *range*, using nested parallelism, or it can cover a potentially blocked multi-dimensional index space range using *MDRangePolicy*.

To facilitate efficient memory access, Kokkos defines a multi-dimensional array type called a *View*. A View is a handle to underlying memory and allows us to manage that using semantics identical to *std::shared_ptr* in C++. The handle is copyable, allowing it to be captured in lambda expressions. Views provide for arbitrary data-layouts (index ordering operations) by defining *Layout* policies. Two common layouts are *LayoutLeft* where indices run left-fastest and *LayoutRight* where indices run rightmost fastest. Kokkos supports the concept of vectorized inner loops in the nested parallel-for dispatch. On CPUs this is implemented by decorating the inner loop with `#pragma ivdep`, while in the CUDA backend, thread blocks are created with the `blockDim.x` being equal to the requested vector length, whereas the y-index of the `threadId` is used for threading otherwise. Since generally thread ID-s are bound to the leftmost index, `LayoutLeft` usually gives coalesced access on NVIDIA GPUs whereas `LayoutRight` will usually give cache line friendly access on CPUs such as KNL. When a vector range is present, the fastest index is the vector lane (`threadIdx.x` on NVIDIA GPUs) and so `LayoutRight` is preferable for both CPU and GPU.

Kokkos provides for portability by implementing a variety of backends. The current most most mature ones are the OpenMP back end to target CPUs and the CUDA one to target NVIDIA GPUs. Other backends for forthcoming systems are in development including a ROCm backend (using HIP) [23] to target AMD and a SYCL back end which is not yet mature enough to have been included in this study. In addition an OpenMP-target offload backend is also in development. Kokkos is mostly a header library for C++ and is quite small. A standard C++ compiler which is capable of driving the underlying hardware is needed. In particular the GPU backend can be utilized either using NVIDIA *nvcc* or the *clang* compilers.

E. SYCL

SYCL [14] is a standard for heterogeneous computing from the Khronos group, originally designed to allow a single source C++ approach to programming OpenCL [24]. It has a model of a node as a collection of *devices* on which one can create work *queues*. Parallel kernels can be submitted to the queues, using functors and lambdas in a similar way to Kokkos. Parallel dispatches can occur over up to 3-dimensional index ranges. Memory handling on the other hand is quite different from Kokkos. Users must create *buffer* objects which manage underlying memory. To access these buffers users first get *accessor* objects either on the host, or within a queue, and must declare their intent as to whether they intend to read or write to the buffer. This leads to safe memory accesses, and in addition the runtime can create a dependency graph of buffer accesses and automate underlying data movement.

SYCL implementations can offer performance portability in a variety of ways, for example, by leveraging Khronos group standards related to OpenCL [24]. A typical approach is that a dedicated SYCL compiler extracts the source code for the kernels from the single source C++ program, after which the kernels are compiled into an intermediate representation (IR) such as Khronos SPIR (Standardized Portable Intermediate Representation), LLVM IR, or PTX byte-code and compiled into the application. The IR can then be Just-In-Time compiled for the hardware device by the device driver to execute on the device. There are several efforts at producing SYCL compilers. Codeplay Ltd. offers a community edition of its ComputeCPP compiler [25] which is what we used for most of the SYCL work in this paper, and which has proved to be very flexible. It can compile kernels into a variety of intermediate byte-code formats (SPIR, SPIRV, PTX and LLVM-IR) which it stores in *integration-headers*. These are then compiled into the final executable. In this way ComputeCPP can target a diverse range of hardware from NVIDIA GPUs and KNL (using the POCL OpenCL driver [26], [27]), as well as Intel CPUs and Intel HD Graphics GPUs through their OpenCL Runtime for Intel CPUs [28] and the Intel(R) Graphics Compute Runtime for OpenCL(TM) (also known as the NEO driver) [29]. However, not every combination of byte-code and driver work well. For example to use NVIDIA GPUs, we found the PTX64 byte-code and the the NVIDIA OpenCL driver to be an unstable combination, whereas the POCL driver would work only with SPIRV and not SPIR in our tests.

There are several other SYCL compilers in the community including one being developed in Clang/LLVM by Intel [30] on which they will base their Data Parallel C++ in the OneAPI initiative. The HIPSYCL [31] effort aims to generate HIP [32] code which can then be compiled directly into ROCm [23] and CUDA in order to target AMD and NVIDIA GPUs respectively. The TriSYCL [33] effort provides yet another open source compiler.

F. Related Work

Many LQCD codes are pursuing performance portability. We have already mentioned Chroma [2] and performance portability through QDP-JIT [3]. We have also mentioned the Grid code [20], a C++ expression template based framework similar in spirit to QDP++ which focuses on vectorization and performance portability between CPU and GPU systems currently. The QUDA library [5] after its most recent restructuring can also launch its kernels on CPUs although the focus as regards performance is still primarily on NVIDIA GPUs. The `hipify` tool in AMD ROCm aims to allow quick initial conversion of CUDA based codes to the HIP programming model. OpenCL was investigated for performance portability for LQCD in the CL^2QCD application [34] where it was noted that while OpenCL proved portable it was not immediately performance portable. It should be noted here that performance portability is likewise not immediately guaranteed

by any of the portability frameworks, rather their merit lies in the degree to which they reduce the difficulty of writing performance portable software for the programmer. OpenMP [35] is another language standard for performance portability. It is being actively investigated for performance portability of C++ expression templates (specifically in reference the Grid code) e.g. in [36]. RAJA [37] is an alternative peformance portability layer, however it does not offer a View abstraction like Kokkos. Rather memory access and management responsibilities are relegated to other packages such as the Copy Hiding Application Interface (CHAI) [38] and UMPIRE [39]. We are not aware of performance portability investigations of LQCD in Raja at the time of writing this paper.

III. Kokkos Wilson-Dslash Implementation

We implemented the Wilson Dslash in the *KokkosDslash* [11] mini-app in two ways: The first approach was a *naive* implementation without any vectorization for the Single-Right-Hand-Side (SRHS) case, and later the second approach implemented the Virtual Node SIMD technique. We localized our policy decisions as regards layout and kernel launch policies in single header file, which could be customized to suit our architecture by build system configuration options. For complex numbers we used the `Kokkos::complex` type which mirrors the standard library `std::complex`, although to allow us to switch to other custom complex types later we aliased this to `MGComplex` (where MG comes from the Multi-grid heritage of the mini-apps).

A. Naive Single and Multi-Right Hand Sides Implementation

Our basic implementation fields targeted NVIDIA GPUs and Intel Xeon and KNL using the CUDA and OpenMP backends of Kokkos respectively. We used Kokkos `View`-s to hold data for our lattice-wide Spinor and Gauge field types. We templated these containers on a contained type. For spinors the contained type could be either a scalar complex number type (for SRHS), or a SIMD-type in order to encode the multiple right hand spinor fields for the MRHS case. We bound one thread in Kokkos to a single lattice site. To evaluate Dslash, in each such thread we needed to hold a full spinor to accumulate the results for the site, and two temporary half spinors. We held this data in automatic thread-local fixed-sized arrays, referred to as `SiteView`-s, which we intended for the compiler to registerize on GPUs. We streamed through the gauge field and so it did not require a `SiteView`.

We created a `SIMDComplex` type as the contained type for the `View`-s used in lattice-wide MRHS spinors. However, the corresponding `SiteView` had to be a little elaborate, due to the KNL and GPU dealing with vectorization differently. For CPUs, the `SiteView` could also be templated on `SIMDComplex` with each OpenMP thread now processing a full SIMD vector. Kokkos, however performed vectorization in the CUDA backend by assigning lanes to the x-dimension of the thread block, but each thread still being scalar. To overcome this we defined a `GPUThreadSIMDComplex` type which internally held just a scalar value. The full interface for accessing the lanes of the `SIMDComplex` was implemented, but lane-accesses only touched the scalar value. The idea was that in a Kokkos `ThreadVectorRange` each GPU thread would need to access data only using its own `threadIdx.x`. We illustrate the idea in Fig. 3. Further, through appropriate templating, we overloaded the complex arithmetic functions to be able to deal transparently with both the scalar and the vector types. As a result the Single and Multi-Right Hand sides implementations differ only in terms of the template parameters to the Dslash functor. In order to allow vectorization, we had to dispatch the Dslash kernel using a `ThreadExecPolicy` having set the chosen SIMD Length.

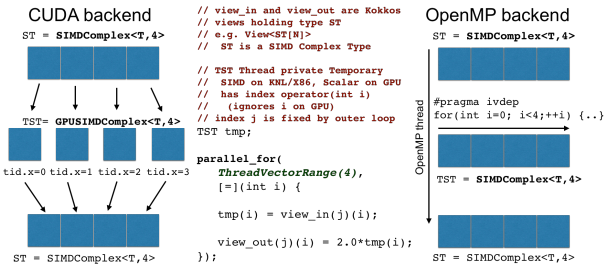

Fig. 3. Utilizing Templating encoding portable SIMD

1) Benchmarking Setup: We measured the performance of our naive SRHS operator on both Intel Xeon Server (SKX) CPU, Intel Xeon Phi Knights Landing (KNL) and NVIDIA GPU systems.

The Skylake Xeon Server platform (referred to as SKX from here on) used was a single node of the Cori-GPU development Cluster at NERSC. The node features 2 sockets of Xeon(R) Gold 6148 CPUs with 20 cores each running at 2.4 GHz. Hyper-threading was enabled on the nodes allowing for 40 OpenMP threads per socket. A single socket has 6 channels to DDR4 memory running at 2666MHz, giving a theoretical maximum DDR memory bandwidth of 127.96 GB/sec. In practice on would expect an efficient code to be able to exhaust around 80% of this theoretical maximum giving practical memory bandwidth limits of around 100 GB/s.

The nodes also feature 8 NVIDIA Tesla V100 SXM2 (Volta) GPUs which were used to generate our GPU results for the Tesla V100 architecture. The V100 compute units feature up to 15 TFLOPS of single precision floating point performance, with High Bandwidth HBM that can run at a theoretical maximum of 900 GB/sec. We also measured performance on V100 SXM2 GPUs on a node of the Summit Supercomputer housed at Oak Ridge Leadership Computing Facility (OLCF) which features six V100 units per node with POWER9 CPUs as hosts. Since we were also able to measure SYCL results on Cori-GPU but not on Summit, we will generally show the Cori-GPU Kokkos results going forward

unless otherwise stated, to allow for consistent comparison. Summit and Cori-GPU results were similar in magnitude, with Summit results typically being a little slower than the Cori-GPU ones. The discrepancies for SRHS were at most 17 GFLOPS (for Kokkos) and 10 GFLOPS (for QUDA) which is less than 2% (Kokkos) and sub-percent (QUDA) respectively of the absolute performances measured on Summit.

For our SKX testing we used the Intel C++ compiler, from Intel Parallel Studio 19.0.3. For the CUDA back-end we compiled with the GNU C++ compiler version 8.3 and CUDA toolkit version 10.1. Our V100 tests, used a single one of the 8 available GPUs on the node. Our CPU tests restricted running to a single socket using the `numactl` tool which was also used to select a local memory allocation policy. This was done to avoid distortion due to NUMA effects.

The KNL system used in our study was a node of the Jefferson Lab 18p cluster featuring Xeon Phi 7250 CPUs with 68 cores running at 1.4GHz. The system runs CentOS 7.4 and we used the version 19.0.0.117 of the Intel C++ compilers. The KNL system features 16 GB on-package high speed MCDRAM which can theoretically deliver over 450 GB/sec memory bandwidth. However, in the quad-cache configuration which we used, the maximum attainable memory bandwidth for the streams triad from MCDRAM is 345GB/sec [40].

Finally, we also ran tests on an NVIDIA Tesla K80 system at Jefferson Lab using the GNU C++ compiler g++ version 6.3.0 which we built from source and CUDA-toolkit version 10.0. We used a single CUDA-device to run on (so only 1 of the 2 accelerators in a K80). This somewhat older system features GDDR GPU memory capable of a maximum GPU memory bandwidth of 240 GB/sec per accelerator in the unit.

In all cases, single right hand side (SRHS) tests used a lattice of size 32^4 sites. On the SKX and KNL platforms we compared performance to the legacy `cpp_wilson_dslash` code [41] with Git revision ID 838efd95 compiled with and without SSE2 optimizations, and against the highly optimized QPhiX library [9] (Git revision acce97c8), On the GPU platforms we used the *dslash_test* program from the QUDA library [7] with Git ID 553c60c.

2) Baseline Performance SRHS: We show our baseline performance in Fig. 4 and can see from the figure that performance even in the naive implementation was reasonable on the GPU systems: we could achieve $\approx 77\%$ of the performance of the QUDA library on the K80 system, and about $87-88\%$ on the V100 device. In terms of absolute performance memory analysis from the NVIDIA visual profiler on Summit showed that the QUDA Dslash kernel is drawing a high amount of memory bandwidth (772 GB/s out of a theoretical maximum of 900 GB/s) justifying our use of QUDA as the comparison point. In turn, a similar analysis for the Kokkos Dslash on Summit showed it drawing 753 GB/sec of HBM memory bandwidth.

Unfortunately on the SKX and KNL systems the naive performance is very low compared to both the legacy *cpp_wilson_dslash* and the QPhiX codes. The primary reason for this turned out to be lack of vectorization which prompted

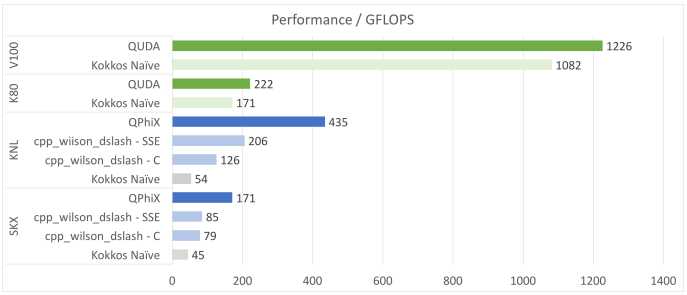

Fig. 4. Baseline Performances compared to standard codes.

us to try the Virtual Node SIMD vectorization. Before we describe that, however, we will first consider the MRHS case:

3) Multi-Right Hand Side Vectorization: As discussed earlier, by allowing the right use of templates the Naive Dslash operator could be used also to encode the MRHS operator. We summarize these performances in Fig. 5. One can see that the Naive Kokkos implementation, using `ThreadVectorRange` gave poor performance on KNL, and indeed (although we do not show it here) it also performed substantially slower than QUDA on the GPUs. On the KNL vectorization did not occur and on the GPUs the range checking of the vector loop caused issues. The situation could be resolved on both CPU and GPU by defining a SIMD vector type (VecType in the figures). On KNL we specialized the arithmetic operations for the vector type `SIMDComplex<float,8>`; which contained 16 floating point values; with AVX512 intrinsics and the performance immediately improved. On GPUs we made two optimiztions. First, we dealt directly with the CUDA `threadIdx.x` index in our vector type and eliminatd the loop as our the entire SIMD range was expressed by the x-dimension of the thread block. Second, we wrote our own complex-number class, deriving from the CUDA *float2* type to assist with aligned and coalesced memory access.

We show the resulting performances in Fig. 5. Since on KNL and SKX we were using a SIMD complex vector of length 8 per site, we opted for a lattice size that was 1/8th of our SRHS case to keep the memory footprint similar between the tests. Thus on KNL we used a lattice size of $16^3 \times 32$ sites. While we do not have a proper multi-right hand side operator to compare with directly we see that we get similar performances now to the QPhiX (SRHS) performance on KNL, whereas we get an improvement compared to QPhiX on SKX, which we attribute to the large Level 3 cache on SKX.

On GPUs we used a vector length of 16 complex numbers, to fill out the warps and correspondingly we divided our original lattice size by 16 to keep the memory footprint similar giving us a lattice of size 16^4 sites. On the GPUs the QUDA library provides a Domain Wall fermion operator (DWF-4D) the diagonal part of which is a multi-right hand side Wilson-Dslash operator and that is what we used here for performance comparison. It is implemented slightly differently in that the right hand sides are not vectorized over, but it still benefits

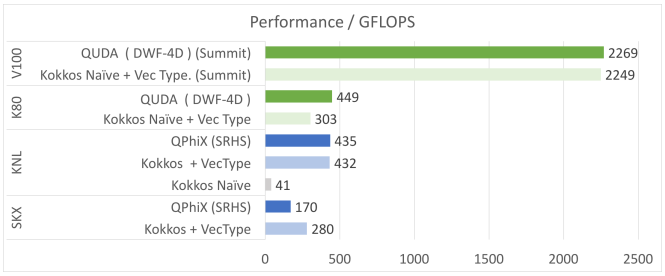

Fig. 5. Multi-Right Hand Side performances

from the gauge field reuse in the same way as our MRHS operator. We note that the Kokkos performances on the K80 are very respectable, and are excellent on the Summit V100 where we reach $\approx 99\%$ of the performance of the QUDA comparison.

4) Virtual Node Mode Single Right Hand Sides: Encouraged by the good performance we saw in the MRHS case with a vector type, we implemented the virtual node SIMD vectorized version of our SRHS operator using our vector types in an attempt to bridge the performance gap remaining specifically on KNL. We implemented the necessary lane permutations to `SIMDComplex<float,8>` using AVX512 shuffle intrinsics. Further we moved to the Kokkos *MDRange* dispatch which allowed us to get a cache blocked lattice traversal for free. However, this optimization had a knock on effect for the GPU side of the code, which was that it forced the use of a SIMD length of 1, as when we switched from the Team dispatch to MDRange we could no longer set a SIMD length (a feature supported currently only by the Team dispatch). Setting a vector length with MDRange is a feature that will need to be implemented in Kokkos in the future. In addition we would need to implement either the necessary swizzles with the CUDA shuffle intrinsics or load-permute functions as mentioned previously, in our own vector type or in a forthcoming Kokkos SIMD type which again, we will leave for future work. Nonetheless, despite working with a SIMD length of 1, the GPU code benefited from the MDRange traversal whose block sizes could now become tunable.

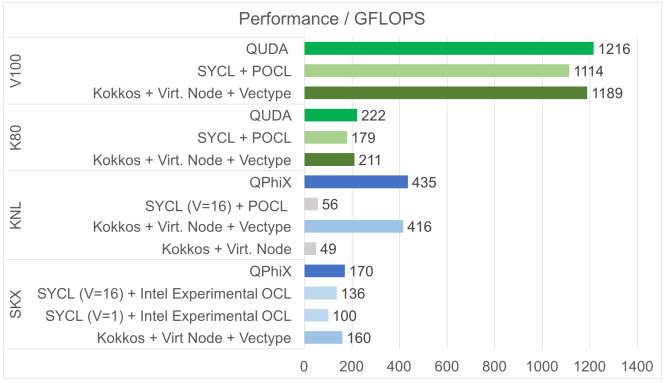

Fig. 6. Virtual-Node SIMD SRHS Operator performances for Kokkos and SYCL

We show the resulting performance data in Fig. 6, again for a lattice of size 32^4 sites. We see that without the vector type performance is still poor on KNL and SKX, however, with the vector type, performances are excellent, and are comparable with QPhiX both for KNL and SKX. Thus using this vectorization scheme and the optimized vector types we could successfully bridge the performance gap between the QPhiX reference implementation and the Kokkos implementation after the block sizes were (auto) tuned. A VTune measurement found that the Kokkos Dslash sustained an MCDRAM bandwidth of 327 GB/sec on KNL. The CUDA performances have also improved primarily from the coalesced access of the vector type and possibly the MDRange dispatch may have captured more locality.

B. Discussion of the Kokkos Dslash

Our experience with Kokkos has been very positive overall. However, for performance we did need to add some optimizations, most specifically the SIMD vector type. Kokkos developers are working on adding standardized SIMD types to Kokkos with a similar envisaged behavior to the one we described for the GPU, namely that a 'thread-local' type must accompany the Vector type, allowing a generalized implementation that should work well on accelerators. Already progress has been made on the `Kokkos::complex` type which when decorated with an *alignas* keyword gets appropriately aligned, obviating the need for our custom type derived from `float2`. A very rough estimate based on previous *sloccount* measurements is that the KokkosDslash mini-app consists of about 4500-5000 lines of code, excluding unit tests. Of this about 274 lines of code are AVX512 specific, a similar amount are AVX2 specific. On the GPU side the `GPUThreadSIMD` specializations using `threadIdx.x` take up about 210 lines whereas the length 1 SIMD specializations take up about 275 lines. So currently this is close to about 1100 lines of architecture specific code. Switching to a standard Kokkos SIMD type could essentially eliminate these lines of code from our mini-app. On the other hand, the fact that Kokkos is embedded into standard C++ and that such specializations are available to the developer if necessary, is an attractive feature of Kokkos.

IV. SYCL Wilson-Dslash Implementation

Our SYCL implementation was a fairly straightforward port of the Kokkos one, however, we implemented only the Virtual Node SIMD SRHS operator, motivated by the apparent vector type support in the SYCL standard via the `vec` template. In terms of compilers we focused on the Clang-SYCL compiler from Intel [30] and the ComputeCPP compiler from Codeplay Software [25]. In our CMake [42] based build system we could treat the Clang SYCL compiler as a regular C++ compiler with an additional need to link against OpenCL. With ComputeCPP there are some special compiler invocations needed to deal with SYCL, and to enable these we used the CMake scripts available in the ComputeCPP SDK [43] which provided the necessary CMake integration.

A. SYCL Vector Type and SYCL View

1) SYCL Vector Type: SYCL provides the template `vec<T,N>` where the type `T` is the type of the vector elements and `N` is the length of the vector, with `N` $\in [0, 15]$. In addition a rich set of swizzle operations is supported. We used this to develop two kinds of Complex SIMD: a) a Fortran-like version: `vec<std::complex<T>, N>` which holds 8 complex numbers with alternating real and imaginary parts and b) a more vector-like version: `std::complex< vec<T,N> >` where the real parts and imaginary parts are held in separate vectors. Here we support a vector length up to 16, so that on a KNL, say, in principle the SIMD vector could be stored 16 complex numbers in 2 AVX512 registers.

However, as it turns out these vector types do not necessarily match to the underlying hardware SIMD. In particular, on both Intel HD Graphics and NVIDIA GPUs SIMD is best implemented at the 'sub-group' (Intel OpenCL Subgroup Extension) [44], [45] or Warp (CUDA) level which these types do not implement. As such on all the platforms we have tried, with the exception of SKX and KNL, the best performance was obtained with $N = 1$, although surprisingly, as we shall show, there were still differences in terms of performance between a) and b) which in this limit should result in the same data layout .

2) Views in SYCL : SYCL differs from Kokkos, in that data is stored in *buffers* which cannot be accessed directly, rather one needs to use *accessors* which are returned by a buffer following calls to their *get_access* methods. In kernel scope this method requires the *command group handler (CGH)* from the command queue. Further, while *accessors* can support multi-dimensional indexing, currently only up to 3-dimensions are supported, which is sufficient for spinors (site,spin,color indices), however, for the gauge field 4 are needed (site, direction, and 2 color indices).

Our implementation of the `View` followed the pattern of accessors in SYCL . We defined a `ViewAccessor` template, which contained within it a SYCL accessor. Two `get_access` methods in `View`, one with a CGH parameter and one without (for host access) allow the creation of the `ViewAccessor`-s. The `ViewAccessor` classes are templated on a `Layout` class, the number of dimensions (indices) and the contained type in the `View`. The `Layout` provides indexing functions which can linearize the indices into an offset index to pass to the underlying SYCL accessor, or can also transform such offset indices back into coordinates. Following Kokkos, we implemented *LayoutLeft* and *LayoutRight* indexing with the leftmost and rightmost indices running fastest, respectively. Finally, due to the RAII (Resource Acquisition is Initialization) nature of initializing SYCL *buffers*, currently our `View` classes do not allow deferred allocation, and the buffers are defined and allocated at `View` object construction. These requirements from SYCL make it less than straightforward to implement Kokkos Views without changing their interface in a manner similar to the one presented here. However, the proposed Unified Shared Memory (USM) extensions [46], if adopted into the SYCL standard would solve this difficulty.

B. SYCL Experimental Setup

Our first target for the SYCL implementation was to see if it can target something other than a regular CPU or NVIDIA GPU, so we chose an Intel Embedded GPU. Our Intel GPU system was an Intel Next Unit of Computing (NUC) containing a Skylake CPU (i7-6770HQ running at 2.6 GHz) with an embedded Intel HD Graphics P580 GPU (running at 950 MHz). The system is running Ubuntu 18.04 TLS for which the Intel Graphics Compute Driver (NEO) is available as a set of precompiled Debian packages. The driver we used here had version 19.36.14103. We used the ComputeCPP Community Edition v1.0.4 (which was packaged specifically for Ubuntu v16.04 on the Codeplay web-site, but ran fine on Ubuntu 18.04 as well). The Intel Clang/SYCL compiler was evolving rapidly during the time of writing this paper. The results we show were obtained using the compiler with GitID: 51a6204c0.

Next we aimed to check performance portability to NVIDIA GPUs on our K80 system. First we tried this with the NVIDIA OpenCL driver available in CUDA-10.0, however this driver does not support consuming SPIR or SPIRV and so we used the *ptx64* target of ComputeCPP. Here we used ComputeCPP Community Edition v1.1.4 for CentOS as our K80 and system was running CentOS 7.4. The combination ran, but was fairly unstable and resulted in failures in our unit tests and a segmentation fault in the timing benchmark. We then tried using the POCL portable OpenCL driver version 1.3. In order to build POCL, we needed the source of Clang/8.0.1 as well as a built version of it with the NVPTX back end enabled (which contained information about the build configuration). Finally we also had to build the *spirv-llvm* translator package from the Khronos Group [47] which needed to be built with the same version of Clang and LLVM sources as POCL. Once all the software had been assembled we could run our SYCL Dslash code on the K80 system using the *spirv64* SYCL target of ComputeCPP. Further, using the POCL driver, it was possible to profile the resulting code using the standard NVIDIA profiling tools *nvprof* and *nvvp* (which no longer support NVIDIA's own OpenCL implementation).

Since the K80 architecture is rather old, we also ran on the V100 nodes of Cori-GPU. Cori GPU is a node supplied by Cray and we did not have super-user access on it. However, we could still use the same approach as for our K80, but needed to also build our own version of an OpenCL-ICD-Loader. The ICD mechanism allows OpenCL to choose amongst multiple OpenCL platforms on a system. It provides an OpenCL interface library and forwards OpenCL calls to the actual OpenCL drivers. The drivers themselves are shared libraries, whose locations are maintained in so called *.icd* files, usually located in the directory `/etc/OpenCL/vendors`. The OpenCL-ICD-Loader we used was from the Khronos-Group [48] and allowed us to use the environment variable `OCL_ICD_VENDORS` to point to a different, user accessible location for the *.icd* file for POCL which we could use to

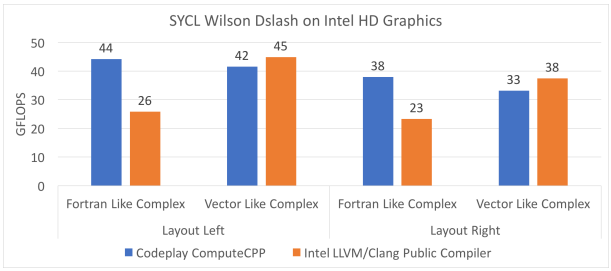

Fig. 7. Performance of SYCL Dslash on Intel-HD Graphics, using the Intel/Clang and Codeplay ComputeCPP compilers

point to our own user installed version of the POCL driver. Fortunately for us, the binary distribution of ComputeCPP for Ubuntu, worked fine on the Cori-GPU node and we were able to run V100 benchmarks with the above setup.

The Cori-GPU nodes allowed us to also run SYCL benchmarks on its SKX sockets. To do this we built the Intel/Clang compiler and downloaded the experimental Intel CPU OpenCL driver [49]. These were fortunately distributed in a `.tar.gz` file as a collection of shared libraries, and we could install them to a user accessible location and add the requisite `.icd` file to our own `OCL_ICD_VENDORS` location.

Finally following our success with CoriGPU, we tried also to run the benchmark on our KNL node. Intel Compute SDK for OpenCL applications currently does not support KNL, and so once again we turned to POCL (as well as ComputeCPP CEv1.1.4 CentOS). To this end we rebuilt POCL for KNL in the same way as for the GPU systems, although this time we did not enable the NVPTX back end, and likewise we rebuilt *spirv-llvm*. The resulting code compiled and ran, but for reasons we have not yet understood this time we had to use the *spir64* target rather than the *spirv64* target of ComputeCPP.

C. Performance Results

1) Intel HD Graphics: The performance results for our SYCL Dslash, on a lattice of 32^4 sites, are shown in Fig. 7. We show all four variations of layout index order and complex number vector implementation. We can see that on our NUC we achieved a maximum performance of 44-45 GFLOPS. Since we do not have a comparison code on the GPU we profiled the application with VTune. We chose the ComputeCPP build which ran at 44 GFLOPS. The profiling overhead caused a slight drop in performance down to 41.4 GFLOPS and VTune indicated that we sustained a read bandwidth of around 30 GB/sec and the average write bandwidth is 3.9 GB/sec. The profile showed that the L3 cache to GTI (Graphics Technology Interface) bandwidth is 32.8 GB/sec, very close to our total GPU memory bandwidth suggesting that nearly all our memory bandwidth comes from either the uncore or the DRAM. The DRAM speed is 2.4 GT/s per channel with 2 channels and a transfer size of 8 bytes giving us a maximum DRAM bandwidth of $2.4 \times 8 \times 2 = 38.4$ GB/s. Additionally, we ran the BabelStream [50] SYCL streams benchmarks which gave the bandwidths shown in Tab. I, where we also show the achieved VTune cross checks of the BabelStream results. We

Function	Bandwidth (MB/sec)	VTune Read (GB/sec)	VTune Write (GB/sec)	VTune Total (GB/sec)
Copy	32883.792	16.0	15.2	31.2
Mul	29302.054	14.5	13.9	28.4
Add	27440.823	17.6	8.5	26.1
Triad	28968.301	18.4	8.9	27.3
Dot	25823.097	28.0	0	28.0

TABLE I
RESULTS OBTAINED FROM THE BABELSTREAM BENCHMARK

can see that BabelStream sustains between 26 GB/sec and 31.2 GB/sec depending on the benchmark in question. Compared with these results, our attained 30GB/sec read + 3.9 GB/sec write bandwidth indicates an excellent level of performance. By applying our earlier performance model from Eq. 2 for 41.4 GFLOPS, we estimate that we are seeing a reuse factor $R = 4$ or $R = 5$ depending on whether nontemporal stores (RFO or Read for Output) are used. The $R = 4$ case predicts a bandwidth of 30 GB/s read bandwidth (without RFO) while the R=5 case predicts the same read bandwidth with RFO. Both cases predict a write bandwidth of 3.0 GB/s.

Our results show a surprising feature, which we cannot yet explain. ComputeCPP seems to achieve higher performance for the Fortran-like implementation of SIMD complex, while in the case of the Intel Clang public compiler, it appears to be the vector-like SIMD complex implementation which is more performant.

2) NVIDIA GPUs: The performances for K80, and V100 GPUs are shown in Fig. 6. We can see that the SYCL code running on top of POCL is slightly slower than the Virtual node Kokkos implementation, but by comparing with Fig. 4 we can see that the SYCL performance is comparable to the baseline naive Kokkos implementation. Specifically the SYCL performances are 1114 GF (179GF) while the naive Kokkos performances are 1070GF (171GF) for the V100 (K80) GPUs respectively.

In the case of the K80 we examined *nvprof* profiles and found that the SYCL code achieves 166.4 GB/sec device memory bandwidth which is actually a little higher than the 150 GB/sec. attained by the (Virtual node SIMD-ized) Kokkos Dslash. The Kokkos Dslash uses 146 registers whereas the SYCL one uses 142, which are both much higher than the 80 registers used by the *WilsonGPU* kernel in QUDA. Interestingly we note that the SYCL code working via ComputeCPP and POCL uses only add and multiply instructions on K80 as opposed to fused multiply adds (FMAs), whereas the Kokkos Variant uses only Adds and FMAs. QUDA in turn uses all three.

As mentioned before on NVIDIA GPU systems, we needed to run the SYCL Dslash code over the POCL OpenCL driver. We show the results for various indexing orders and our two SIMD complex-number implementations in Tab. II for K80. Similarly to Intel HD Graphics we see that performance varies depending on the particular configuration of index order and SIMD Complex implementation. It is unambiguous that LayoutLeft is the preferred order

	Layout Left (GFLOPS)	Layout Right (GFLOPS)
Fortran-like Complex	179	41
Vector-like Complex	149	22

TABLE II
SYCL DSLASH PERFORMANCE ON K80 GPU AND POCL, USING COMPUTECPP IN VARIOUS CONFIGURATIONS.

	LayoutLeft (GFLOPS)	Layout Right (GFLOPS)
Fortran-like complex (V=1)	16	36
Fortran-like complex (V=8)	44	55
Vector-like complex (V=1)	11	28
Vector-like complex (V=16)	39	56

TABLE III
SYCL DSLASH PERFORMANCE ON INTEL KNL USING POCL

for NVIDIA GPUs, which would be the expectation from the point of view of coalesced access if the GPU thread ID's are bound to the leftmost index. For K80 we get the best performance when we use Fortran-like complex numbers (`vec<std::complex<float>,N>`) as opposed to the *vector-like* (`std::complex<vec<float,N>>`) case, even tho once again the vector length is 1.

3) Intel KNL: On the Intel KNL system, we once again had to rely on POCL. We have run the usual benchmark with 32^4 sites on our KNL system having varied once again both the Layout and the complex-number type. In addition we attempted to switch the vector length to see if we could get any benefit from the `vec` template in SYCL. We chose the vector length to match the hardware length of 512-bit vectors. In the Fortran-like order this corresponded to 8 complex numbers (in 1 register) whereas for the vector-like layout it corresponded to 16 complex numbers in 2 registers, storing the vectors of the real and imaginary parts respectively. Our results are shown in Tab. III. Looking at the table we see that `LayoutRight` is clearly preferred and that utilizing the vector types helps a little, but we are still in the ballpark of the naive Kokkos implementation and roughly a factor of 8 away from our comparison of QPhiX and the Kokkos virtual node operator using AVX512 intrinsics (which both sustained over 400 GFLOPS). It may be possible to tune these numbers further using POCL environment variables which we leave for potential future work.

D. Intel Xeon Server (SKX)

The performance results for the SKX system are also shown in Fig. 6. We show the maximum performance obtained after varying the parameters for both layout and vector length. One interesting difference here, compared to the other systems is that changing the vector length does appear to help. We suspect this is a feature of using the Intel Clang Compiler in combination with the Intel Experimental OpenCL runtime for SYCL rather than any other feature of the node. Using V=16 and Vector-like complex numbers we achieve up to 136 GF / socket, which is about 80% of the corresponding QPhiX performance and 85% of the corresponding virtual node Kokkos performance.

V. DISCUSSION

We achieved excellent performance on our target architectures with Kokkos (KNL and NVIDIA GPU) and likewise with SYCL on the Intel HD Graphics GPU with the NEO driver.

We achieved reasonably good performance with SYCL on the NVIDIA GPU systems using POCL and on SKX using the Intel Clang toolchain and experimental OpenCL CPU driver. In these cases the best SYCL performances were a little below the best Kokkos SRHS performances but in the case of the GPUs were comparable with the naive Kokkos performances. The main difference between the naive and virtual-node SIMD-ized Kokkos implementations is the use of the `MDRange` dispatch for the virtual node implementation, as opposed to the flat team based dispatch of the naive one. This suggests that by adding a similar blocked dispatch, the SYCL version may actually catch up to the best Kokkos implementation by capturing locality in the caches. This may be especially beneficial for the SKX implementation which has a large L3 cache.

On the down side, the results from KNL were not nearly as good as the others and in the GPU cases using Compute CPP with the NVIDIA OpenCL driver proved somewhat unstable, either due to to the NVIDIA driver itself or due to the PTX64 bitcode produced by ComputeCPP. This raises an important point, that in order for SYCL to be successful it requires both good compiler and driver support. The CUDA system, where NVIDIA maintains and develops both the *nvcc* and the device drivers, is a good example of how this can work well. Intel's commitment to the OpenCL drivers for its HD Graphics and OpenCL Runtime for their Xeon processors also results in a combination that works well. However, when hardware is no longer well supported one needs to resort to other publicly available solutions such as POCL which work well in some cases and perhaps less well in others. At the same time we are cognizant of the fact that support for products will always be a market driven decision by the vendors.

It has become clear that clarity and guidance is required to best be able to adequately exploit SIMD vectorization possibilities on accelerators via SYCL. It was frustrating to find a general vector type, and to invest effort in coding up the necessary swizzles only to discover that on the accelerators the ideal SIMD length from the application is actually 1. While on SKX using a larger SIMD length seemed to help, the gain was not very big. On the KNL system explicit SIMD-coding with the `vec` template had an effect, but it certainly was not as effective as the AVX512 vector type was in the case of Kokkos. The proposed sub-group extensions to OpenCL and to SYCL by Intel [44], [45] may be a good step towards remedying this situation

The SYCL way of managing memory through buffers and accessors may be safe, but feels somewhat cumbersome to us and may create difficulties interfacing with non-SYCL external libraries in an efficient way. Further, often it is desirable to have explicit control over where the data is rather than

delegating the management of memory to the SYCL runtime. The Unified Shared Memory extensions proposed for SYCL [46] which allow the allocation of memory explicitly on the host or the device address these concerns, but need to be accepted into the SYCL standard and implemented by the various compiler vendors in order to be successful.

VI. Conclusions and Future Work

In this paper we presented two Wilson Dslash mini-apps encoded in the Kokkos and SYCL programming models. We described the structure and optimizations involved in the mini apps and explored their performance on a wide variety of hardware (Intel HD Graphics, NVIDIA GPUs, SKX and KNL). In order to do this we utilized the Intel Clang public SYCL compiler and the Codeplay ComputeCPP compiler in combination with the Intel NEO Compute Graphics runtime, the Intel Experimental OpenCL runtime for CPUs and the Portable Open Compute Language (POCL) runtime. We achieved good performance on our accelerator targets with both Kokkos and SYCL but were only successful on the KNL platform using Kokkos combined with a manually added SIMD type. Our future work can range in many directions: it would be worthwhile to extend our work to AMD GPUs over ROCm using either POCL or the HIPSYCL compiler. Likewise we aim to port the KokkosDslash mini-app using the currently developing Kokkos HIP backend as it matures. This would be helpful in planning for performance portability to the forthcoming Frontier System at OLCF. We plan to integrate the SIMD type(s) now in development in Kokkos into the Kokkos Dslash to see if we can eliminate our own architecture specific SIMD types while still maintaining our current level of performance portability. The approach may also allow us to rapidly port to new targets, such as the POWER9 or ARM CPUs.

An issue we have not investigated in this paper but which is important for the future would be communications between devices and how best they can be carried out in the context of these programming models (for example, whether communications can be initiated from within a kernel and whether the programming models need any extensions to facilitate this) and we aim to extend these benchmarks to support multiple devices in the future.

VII. Acknowledgement

This work was funded by the U.S. Department of Energy under the Exascale Computing Project by the Office of Advanced Scientific Computing Research and through the Scientific Computing Through Advanced Discovery (SciDAC) program of the U.S. Deparment of Energy Offices of Nuclear Physics and Office of Advanced Scientific Computing Research (ASCR). B. Joó gratefully acknowledges the NERSC Exsascale Scientific Applications Program (NESAP) of NERSC for supporing a Summer Associateship at NERSC to work on the material presented in this paper. We gratefully acknowledge use of computer systems at NERSC, Jefferson Lab, Argonne Leadership Computing Facility and Oak Ridge Leadership Computing Facility for development and benchmarking during this work. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Nuclear Physics under contract DE-AC05-06OR23177. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy?s National Nuclear Security Administration under contract DE-NA-0003525.

References

[1] T. Straatsma, K. Antypas, and T. Williams, *Exascale Scientific Applications: Scalability and Performance Portability*, ser. Chapman & Hall/CRC Computational Science. CRC Press, 2017, ch. Chapter 16 Lattice Quantum Chromodynamics and Chroma (B. Joo, R. G. Edwards, F. T. Winter). [Online]. Available: https://books.google.com/books?id=rGQ-DwAAQBAJ

[2] R. G. Edwards and B. Joo, "The Chroma software system for lattice QCD," *Nucl. Phys. Proc. Suppl.*, vol. 140, p. 832, 2005, [,832(2004)].

[3] F. T. Winter, M. A. Clark, R. G. Edwards, and B. Joó, "A framework for lattice QCD calculations on GPUs," in *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, ser. IPDPS '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 1073–1082. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2014.112

[4] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673

[5] M. A. Clark, R. Babich, K. Barros, R. C. Brower, and C. Rebbi, "Solving Lattice QCD systems of equations using mixed precision solvers on GPUs," *Comput. Phys. Commun.*, vol. 181, pp. 1517–1528, 2010.

[6] R. Babich, M. A. Clark, and B. Joó, "Parallelizing the QUDA library for multi-GPU calculations in lattice quantum chromodynamics," *ACM/IEEE Int. Conf. High Performance Computing, Networking, Storage and Analysis, New Orleans*, 2010.

[7] M. Clark and R. Babich, "QUDA: A library for QCD on GPUs," http://lattice.github.io/quda/.

[8] B. Joó, D. Kalamkar, K. Vaidyanathan, M. Smelyanskiy, K. Pamnany, V. Lee, P. Dubey, and W. Watson, "Lattice QCD on Intel(R) XeonPhi(TM) Coprocessors," in *Supercomputing*, ser. Lecture Notes in Computer Science, J. Kunkel, T. Ludwig, and H. Meuer, Eds. Springer Berlin Heidelberg, 2013, vol. 7905, pp. 40–54.

[9] Joó, B., "`qphix` package web page," http://jeffersonlab.github.io/qphix.

[10] B. Joó, "`mg_proto`: a prototype multi-grid library for QCD," https://github.com/jeffersonlab/mg_proto.

[11] B. Joo, "A Wilson-Dslash MiniApp written in Kokkos," https://github.com/bjoo/KokkosDslash.git, 08 2019.

[12] ——, "A Wilson-Dslash MiniApp written in SYCL," https://github.com/bjoo/SyCLDslash.git, 08 2019.

[13] H. C. Edwards and D. Sunderland, "Kokkos Array Performance-portable Manycore Programming Model," in *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, ser. PMAM '12. New York, NY, USA: ACM, 2012, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/2141702.2141703

[14] T. K. O. W. G. S. Subgroup, "The sycl(tm) specification version 1.2.1," https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdfa, April 2019.

[15] B. Joó, M. Smelyanskiy, D. D. Kalamkar, and K. Vaidyanathan, "Chapter 9 - Wilson Dslash Kernel From Lattice QCD Optimization," in *High Performance Parallelism Pearls Volume Two: Multicore and Many-core Programming Approaches*, J. Reinders and J. Jeffers, Eds. Boston, MA, USA: Morgan Kaufmann, 2015, vol. 2, pp. 139 – 170. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780128038192000239

[16] M. Smelyanskiy, K. Vaidyanathan, J. Choi, B. Joó, J. Chhugani, M. A. Clark, and P. Dubey, "High-performance lattice QCD for multi-core based parallel systems using a cache-friendly hybrid threaded-MPI approach," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11, 2011, pp. 69:1–69:11.

[17] A. Pochinsky, "Writing efficient QCD code made simpler: QA(0)," *PoS*, vol. LATTICE2008, p. 040, 2008.

[18] O. Kaczmarek, C. Schmidt, P. Steinbrecher, and M. Wagner, "Conjugate gradient solvers on Intel Xeon Phi and NVIDIA GPUs," in *Proceedings, GPU Computing in High-Energy Physics (GPUHEP2014): Pisa, Italy, September 10-12, 2014*, 2015, pp. 157–162.

[19] O. Kaczmarek, C. Schmidt, P. Steinbrecher, S. Mukherjee, and M. Wagner, "HISQ inverter on Intel Xeon Phi and NVIDIA GPUs," *CoRR*, vol. abs/1409.1510, 2014. [Online]. Available: http://arxiv.org/abs/1409.1510

[20] P. Boyle, A. Yamaguchi, G. Cossu, and A. Portelli, "Grid: A next generation data parallel C++ QCD library," 2015.

[21] P. Boyle, "The BlueGene/Q supercomputer," *PoS*, vol. LATTICE2012, p. 020, 2012. [Online]. Available: http://pos.sissa.it/archive/conferences/164/020/Lattice%202012_020.pdf

[22] H. Carter Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *J. Parallel Distrib. Comput.*, vol. 74, pp. 3202–3216, Dec. 2014. [Online]. Available: https://doi.org/10.1016/j.jpdc.2014.07.003

[23] "ROCm, a New Era in Open GPU," https://rocm.github.io/.

[24] K. Group, "Khronos OpenCL registry, which contains specifications for the core API; Khronos- and vendor-approved extensions to the core API; the OpenCL C and C++ languages; and the OpenCL SPIR-V Environment," https://www.khronos.org/registry/OpenCL.

[25] "Codeplay ComputeCPP website," https://www.codeplay.com/products/computesuite/computecpp.

[26] V. Korhonen, "Portable OpenCL Out-of-Order Execution Framework for Heterogeneous Platforms," https://dspace.cc.tut.fi/dpub/handle/123456789/22636?show=full, 12 2014.

[27] "Portable Computing Lanugage Web Site," http://portablecl.org/.

[28] Intel, "OpenCL(TM) Runtimes for Intel Processors," https://software.intel.com/en-us/articles/opencl-drivers.

[29] ——, "Intel(R) Graphics Compute Runtime for OpenCL(TM)," https://github.com/intel/compute-runtime.

[30] "Intel Public SYCL Compiler," https://github.com/intel/llvm.

[31] "HIP SYCL GitHub Project," https://github.com/illuhad/hipSYCL.

[32] AMD, "It's HIP to be Open whitepaper," https://gpuopen.com/wp-content/uploads/2016/01/7637_HIP_Datasheet_V1_7_PrintReady_US_WE.pdf.

[33] "TriSYCL GitHub web page," https://github.com/triSYCL/triSYCL.

[34] O. Philipsen, C. Pinke, A. Sciarra, and M. Bach, "CL^{2}QCD - Lattice QCD based on OpenCL," *PoS*, vol. LATTICE2014, p. 038, 2014.

[35] OpenMP Architecture Review Board, "OpenMP Application Program Interface," 2011. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP3.1.pdf

[36] P. A. Boyle, M. A. Clark, C. DeTar, M. Lin, V. Rana, and A. V. Aviles-Casco, "Performance Portability Strategies for Grid C++ Expression Templates," *EPJ Web Conf.*, vol. 175, p. 09006, 2018.

[37] R. D. Hornung and J. A. Keasler, "The raja portability layer: Overview and status," 9 2014.

[38] L. L. N. Laboratory, "Copy-hiding array abstraction to automatically migrate data between memory spaces," https://github.com/LLNL/CHAI.

[39] ——, "An application-focused API for memory management on NUMA & GPU architectures," https://github.com/LLNL/Umpire.

[40] D. Doerfler *et al.*, "Applying the Roofline Performance Model to the Intel Xeon Phi Knights Landing Processor," in *High Performance Computing. ISC High Performance 2016*, ser. Lecture Notes in Computer Science, M. Taufer, B. Mohr, and J. Kunkel, Eds. Springer, Cham, 2016, vol. 9945, no. "DOI: https://doi.org/10.1007/978-3-319-46079-6_24".

[41] "A C++ Wilson Dslash Operator," https://github.com/jeffersonlab/cpp_wilson_dslash.git.

[42] Kitware, "CMake Build System Website," https://cmake.org/.

[43] "Collection of samples and utilities for using ComputeCpp, Codeplay's SYCL implementation," https://github.com/codeplaysoftware/computecpp-sdk.

[44] A. Bader and Intel, "SYCL(TM) Proposals: Sub-groups for NDRange Parallelism," https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/SubGroupNDRange/SubGroupNDRange.md.

[45] B. Ashbaugh and B. George, "Modeling Explicit SIMD Programming With Subgroup Functions," in *Proceedings of the 5th International Workshop on OpenCL*, ser. IWOCL 2017. New York, NY, USA: ACM, 2017, pp. 16:1–16:4. [Online]. Available: http://doi.acm.org/10.1145/3078155.3078178

[46] J. Brodman and Intel, "SYCL(TM) Proposals: Unified Shared Memory," http://https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/USM/USM.adoc.

[47] K. Group, "LLVM framework with SPIR-V support," https://github.com/KhronosGroup/SPIRV-LLVM.

[48] ——, "The OpenCL ICD Loader project," https://github.com/KhronosGroup/OpenCL-ICD-Loader.

[49] "Experimental Intel(R) CPU Runtime for OpenCL(TM) Applications with SYCL support version (Linux)," https://github.com/intel/llvm/releases/download/2019-09/oclcpuexp-2019.8.8.0.0822_rel.tar.gz.

[50] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, "Evaluating attainable memory bandwidth of parallel programming models via Babel Stream," *Internatiional Journal of Computer Science and Engineering*, vol. 17, no. 3, pp. 247–262, 2018.