Lattice QCD on Intel Xeon Phi

Bálint Joó[1], Dhiraj D. Kalamkar[2], Karthikeyan Vaidyanathan[2], Mikhail Smelyanskiy[3], Kiran Pamnany[2], Victor W Lee[3], Pradeep Dubey[3], and William Watson III[1]

[1] Thomas Jefferson National Accelerator Facility, Newport News, VA, U.S.A
[2] Parallel Computing Lab, Intel Corporation, Bangalore, India
[3] Parallel Computing Lab, Intel Corporation, Santa Clara, CA, U.S.A

Abstract. The Intel Xeon Phi architecture from Intel Corporation features parallelism at the level of many x86-based cores, multiple threads per core, and vector processing units. Lattice Quantum Chromodynamics (LQCD) is currently the only known model independent, non perturbative computational method for calculations in theory of the strong interactions, and is of importance in studies of nuclear and high energy physics. In this contribution, we describe our experiences with optimizing a key LQCD kernel for the Xeon Phi architecture. On a single node, our Dslash kernel sustains a performance of around 280 GFLOPS, while our full solver sustains around 215 GFLOPS. Furthermore we demonstrate a fully 'native' multi-node LQCD implementation running entirely on KNC nodes with minimum involvement of the host CPU. Our multi-node implementation of the solver has been strong scaled to 3.6 TFLOPS on 64 KNCs.

1 Introduction

Lattice Quantum Chromo-dynamics (QCD) is a computationally challenging problem that solves the discretized Dirac equation in the presence of an SU(3) gauge field. Its key operation is the multiplication of a matrix vector, known as the Dslash operator, with a vector.

The current trend in high performance computing is to couple commodity processors with various types of computational accelerators, which offers dramatic increases in both compute density, memory bandwidth and energy efficiency.

In this paper we describe implementations and tuning of LQCD for Intel's recently released Intel® Xeon Phi™ coprocessor, codenamed Knights Corner (KNC). Our implementation exploits the salient architectural features of KNC, such as large caches, inter-core communication as well as hardware support for irregular memory accesses. By using KNC-friendly lattice layout together with a blocking algorithm, our implementation of Dslash kernel on KNC sustains 280 GFLOPS on a single node which corresponds to nearly 80% of achievable performance. Furthermore, we demonstrate fully 'native' multi-node LQCD implementation running entirely on KNC nodes, with minimum involvement of CPU host. Our multi-node implementation of Dslash and full solver have been strong scaled to 4.5 TFLOPS and 3.6 TFLOPS, respectively, on 64 KNCs

2 Background

2.1 Lattice QCD

In this section we provide relevant details of LQCD for our work. For further details, please refer to many excellent references e.g [1]. LQCD operates on an $N_d = 4$ dimensional space time lattice, with $V = L_x L_y L_z L_t$ sites, where L_x, L_y, L_z and L_t are the dimensions of the lattice in X,Y,Z and T respectively. Quark fields are ascribed to the sites of the lattice while gluon fields are ascribed to the links between sites. The interaction of quarks and glouns is given by the Fermion matrix M. In the Wilson [2] formulation of quarks M is given by:

$$M = (N_d+m) - \frac{1}{2}D, \text{with } D = \sum_{\mu=1}^{4} \left((1-\gamma_\mu) \otimes U_x^\mu \, \delta_{x+\hat{\mu},x'} + (1+\gamma_\mu) \otimes U_{x-\hat{\mu}}^{\mu\dagger} \, \delta_{x-\hat{\mu},x'} \right) \tag{1}$$

where D is the *Wilson-Dslash* operator (WD). In Eq. 1 the sum is over directions μ, m is a quark mass parameter, U_x^μ is the gauge link matrix connecting sites x with its neighbor in the μ direction, and γ_μ are elements of a Dirac spin-algebra. The propagation of quarks in a gluon field is given by the Dirac equation: $M\psi = \chi$ where ψ and χ are spinors, which at each site x are complex matrices carrying a spin index $\alpha \in [0, 1, 2, 3]$ and color index $a \in [0, 1, 2]$. Applying D to a spinor can be viewed as a nearest neighbour stencil operation. To facilitate the solution of M typically an even-odd preconditioning is used, wherein a lattice is divided (typically in the x direction) into even and odd sub-lattices and one solves the Schur complement system $\tilde{M}_{oo}\tilde{\psi}_o = \tilde{\chi}_o$ only on one checkeboard (in this case 'odd'). Here $\tilde{M}_{oo}$ is the Schur complement of M after checkerboarding given by:

$$\tilde{M}_{oo} = (N_d + m)I_{oo} - \frac{1}{4(N_d + m)} D_{oe} D_{eo} \tag{2}$$

where subscripts oe, eo indicate that the operator maps odd sites to even, even to odd respectively. The system is large and sparse and is typically solved by an interative solver such as Conjguate Gradients (CG) [3] or BiCGStab [4].

The key operation is the sparse matrix vector multiplication $D\psi$. The naive arithmetic intensity of the Dslash operator is 1320 flops / 1440 bytes = 0.92 flops/byte in single precision, however, due to nearest neighbor nature of the Dslash operator, there is substantial reuse amongst spinors. Properly exploiting this reuse with caches, can almost double the arithmetic intensity [5]. In some cases one can perform 2-row gauge field compression, thus trading flops for bandwidth by making use of the $SU(3)$ nature of the links and storing only two rows of the matrix and reconstucting the third row on the fly [6].

2.2 Intel® Xeon Phi™ coprocessor architecture

The recently released Intel® Xeon Phi™ coprocessor architecture features many in-order cores on a single die. Each core has 4-way hyper-threading support to help hide

memory and multi-cycle instruction latency. In addition, each core has 32 vector registers, 512 bits wide, and its vector unit executes 16-wide (8-wide) single (double) precision SIMD instructions in a single clock, which can be paired with scalar instructions and data prefetches. KNC has two levels of cache: single-cycle 32 KB first level data cache (L1) and larger globally coherent second level cache (L2) that is partitioned among the cores. Each core has a private 512 KB partition. KNC also has hardware support for iregular data accesses and features several flavors of prefetch instructions for each level of the memory hierarchy.

KNC is physically mounted on a PCIe slot and has dedicated GDDR memory. Communication between the host CPU and KNC is therefore done explicitly through message passing. However, unlike many other coprocessors, it runs a complete Linux-based operating system, with full paging and virtual memory support, and features a shared memory model across all threads and hardware cache coherence. Thus, in addition to common programming models for coprocessors, such as OpenCL, KNC supports more traditional multiprocessor programming models such as pthreads and OpenMP.

3 QCD Implementation on KNC

Our library is written in C++ and threading is carried out using OpenMP threads. The library implements the Wilson Dslash and the even-odd preconditioned Wilson operators and a Conjugate Gradients solver. We consider the code in two parts: a high level part which is concerned with parallelizing over threads, and performs the loop structure for our cache blocking strategy, and a 'back end' part, which takes care of working on a vector of lattice sites.

To achieve high performance on KNC, one must take full advantage of its vector capabilities. A SIMD friendly, *partial structure of arrays* (SOA) layout such as described in [5] can run with high vector efficiency, however it requires that a *scanline* (line of sites in X) be a multiple of the vector unit length (vec). This can then restrict the application of the code to problems where $L_{xh} = L_x/2$ the X-width of a checkerboard is a multiple of vec. The larger vec is, the more restrictive this becomes. Our first KNC implementation was written this way. However, we have developed a more general approach, described below, to address this limitation. Specifically, we allow the inner array length soa of the SOA (or SOA length) to be a factor of vec. This has the advantage of allowing more general problem sizes, but since it mixes X and Y dimensions it can complicate communications in those directions. Our code is templated on floating point type, vector length vec and SOA length soa. The back end codes are generated using a code generator which we will describe below, and are hooked into the main library using template specialization. We focused specifically on the cases of $vec = 16$ ($vec = 8$), $soa = 4, 8, 16$ ($soa = 4, 8$) for Xeon Phi (Xeon E5-AVX).

Our primary data structures are $SU(3)$ gauge fields, 4-spinors as discussed earlier. Each of these fields is associated with a lattice site. In the case of the gauge fields we associate with a site the 8 links emenating from it (forward and backwards in each of 4 directions). We then split the lattice into blocks of sites. In the case of spinors we have $V_B = L_{xh}L_yL_zL_t/soa$ such blocks per checkerboard. We require that soa divide L_{xh} exactly, and that vec divide L_yL_{xh} exactly. In this way our back end kernels can

process $soa \times ngy$ spinor blocks where $ngy = vec/soa$. A single processing step can thus process a full vec sites worth of data, made up of spinors of length soa from ngy different y coordinates. The situation is illustrated in Fig. 2 on the right panel. The gauge fields could be packed similarly, however since they are typically used in several dslash applications, it is worth repacking the ngy blocks of length soa into a single block of length vec up-front, allowing them to be read as a single vector. Hence the gauge field has $V_{BG} = L_{xh}L_yL_zL_t/vec$ blocks. We show the types of the block data in figure 1. To

```
typedef float SU3MatrixBlock[8][3][3][2][vec];
typedef float FourSpinorBlock[3][4][2][soa];
```

Fig. 1: The structures of the fields used for a single block of computation.

reduce the effects of associativity conflict misses, we pad our array, by adding Pad_{xy} blocks onto the end of every X-Y plane and Pad_{xyz} blocks onto the end of every XYZ time-slice. Hence a spinor site with coordinates (x, y, z, t) is indexed as follows: We locate the X-Y plane for the z and t indices using the formula $xyBase = t\ Pxyz + z\ Pxy$ with $Pxy = (L_{xh}L_y) + Pad_{xy}$, and $Pxyz = L_zPxy + Pad_{xyz}$. Within the X-Y plane, we first split the x coordinate into an x-block index; $xb = x/soa$ and an index within the block: $xi = x \mod soa$. The offset to the block is now $xb + nsoa * y$ with $nsoa = L_{xh}/soa$. Given an array `spinor` of objects of type `FourSpinorBlock` as defined in Fig. 1, with V_B array elements, the site would be indexed as `spinor[ xb + nvecs*y + xyBase ][c][s][r][xi]` where `c`, `s` and `r` are indices for the color, spin and complex component respectively. In our code, we compute $xyBase$ and then separate offsets in terms of floating point numbers to be used by gathering loads. Indexing gauge fields is slightly different, since ngy lines of y are packed together. In this case the block offset would be computed as $xb + (nvec * y + xyBase)/ngy$.

We wrote a simple code generator to overcome two programming challenges: having a portable abstraction for producing intrinsics to generate code for both Xeon Phi, and AVX. The second challenge was to intermingle software prefetch instructions with the main computation. The generation of L1 prefetch instructions is part of the code generating routines. The L2 prefetches are generated in a separate pass and the two instruction streams are intermixed. The current version of the generator supports single precision vector instructions for Xeon Phi and AVX. It is straightforward to extend the generator to support double precision and other vector architectures. Gathering (scattering) spinors from (to) their $soa \times ngy$ XY blocks is supported both via the gather intrinsics on Xeon Phi, or via a sequence of masked load-unpack, or pack-store instructions. To take advantage of streaming stores we use in-register shuffles to pack full cache lines, which are then streamed to memory using nontemporal store instructions.

Our code implements a variant of the usual 3.5D blocking [7], however, we have a sufficiently large number of cores, that we cannot divide the Y-Z planes amongst them, without local Y-Z blocks becoming too small and leading to excessive redundant memory traffic. Instead, we consider our cores as making up a 3-dimensional grid of cores with dimensions (C_y, C_z, C_t). The T-dimension is effectively blocked with a block size

$B_t = L_t/C_t$, (with some cleanup work if C_t does not divide L_t exactly). To block in the YZ plane, we introduce blocking factors B_y and B_z and require that B_yC_y divide L_y exactly and similarly that B_zC_z divide L_z exactly. Each core then works on an integer number of blocks of size $B_y \times B_z$ sites in the YZ plane, and we can tune B_y and B_z to maximize cache reuse. We show semantically the blocking in Fig. 2. As a result, cores working on the same block of T constructively reuse spinor data across adjacent YZ planes. Moreover, cores working on adjacent By x Bz sites will share boundary data via inter-core communication, without going back to memory. However, in order to ensure constructive sharing, it is important to maintain some degree of synchronization between these cores, which is achieved via infrequent inter-core barriers amongst groups of $Cy \times Cz$ cores.

Within each core we treat the available SIMT threads as a grid of threads with dimensions $S_y \times S_z$. Our lattice traversal looping strategy for any given thread is as follows: First, the core and SIMT coordinates (c_y, c_z, c_t) and (s_y, s_z) of the thread are computed, and then one identifies the origin of the first YZ tile which that thread will process. The thread then loops over the Y and Z blocks, leaping over those done by the other cores. In each block, the threads stream through the local T portion. The local Y and Z plane of the block is split between the SIMT threads on the core. Since each SIMT thread processes a $soa \times ngy$ block looping over x and y is done in units of blocks of length soa in X, and in increments of $nyg * S_y$ in Y respectively. The innermost loop can then use the information from the loop indices and the origins computed at the outset to index the neighbour blocks to be read, the output block to be written, and the neighbours of the successive block for L2 cache prefetching.

On Xeon Phi, the optimal block size was $(B_y, B_z) = (4, 4)$ which gave the best tradeoff between cache efficiency and redundant memory traffic on the edges of the blocks. We note also that for computing Dslash, we always used all the available SIMT threads on a core, and threads were indexed in a SIMT major order, ie: thread ID tid is defined as $tid = s_y + S_y * (s_z + S_z * (c_y + C_y * (c_z * C_z * c_t)))$.

In order to implement a Conjugate Gradients solver, one needs several Level-1 BLAS like streaming linear algebra operations. These operations require few floating point operations and are heavily bound by memory bandwidth. If there is a chance to increase reuse between them it is worth fusing several successive kernels, for example in a situation where one can compute a residual vector, and compute its norm at the same time. We coded these operations in vector intrinsics with explicit software prefetching. We found we can achieve different levels of throughput depending on how many threads are per core. We show in table 1 the various kernels used along with the (possibly fused) operatins they perform, and as an illustration, the memory bandwidths achieved by the kernels from a particular timing measurement. We see that in most kernels one or two threads per core performed optimally. In our timing runs we have auto-tuned the number of threads in our BLAS kernels for efficiency.

3.1 Multi-Node considerations

It is typical to parallelize LQCD problems onto multiple nodes to increase performance. Further, memory limitations can also force the calculation onto multiple nodes. Consequently we paralellized our code onto multiple Xeon Phis (and also SNB-EPs). The

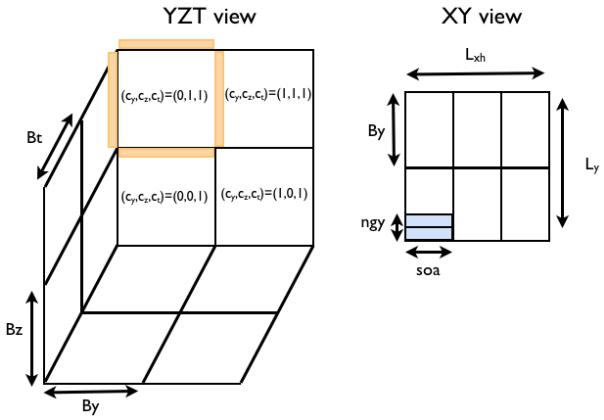


Fig. 2: Blocking the lattice. Left: YZT view showing B_y, B_z and B_t, and core coordinates for the top block. The boundary layer of the YZ plane for one of the blocks is shown in orange, although (for the last T-slice) typically it would wrap around. Right: XY-view, showing blocks of length soa in X, and ngy lines in Y. In this instance $ngy = 2$ and $vec = 2soa$

Routine Name	operation	B/W 1 thread (GB/s)	B/W 2 threads (GB/s)	B/W 3 threads (GB/s)	B/W 4 threads (GB/s)
`aypx2`	$y \rightarrow \alpha y + x$	172	164	152	148
`xmyNorm2`	$r \rightarrow x - y$ $\rho \rightarrow \|\|r\|\|^2$	150	145	135	128
`norm2`	$\rho \rightarrow \|\|r\|\|^2$	154	157	154	156
`rmammpNorm2rxpap`	$r \rightarrow r - \alpha q$ $\rho \rightarrow \|\|r\|\|^2$ $x \rightarrow x + \alpha p$	165	141	124	115
`copy`	$x \rightarrow y$	155	155	149	149

Table 1: BLAS Like Kernels and the operations they perform. Here x, y, r, p, q are lattice spinors, α and ρ are scalars, and in a Conjugate Gradients solver we have that $q = M^\dagger M p$. We also quote memory bandwidths observed in these kernels as a function of threads per core from a single timing measurement on a Xeon Phi 7110P device in a node of the Endeavor cluster, rounded to the nearest GB/sec.

multi-node implementation overlaps computation of the body with the communication of faces. Faces are projected to two spinors to reduce data to be communicated. Once the body computation and communications are complete, the faces are multiplied appropriately with gauge links, and their contribution to Dslash is accumulated as is usual in QCD implementations (e.g. [8, 9]). Our face processing routines were also written with the code generator, but are not yet optimized to the same extent as the body computation.

A novel feature of the Xeon Phi architecture and software ecosystem, is that native implementations can make direct MPI [10] message passing calls, freeing up the user from orchestrating data transfers between host and the co-processor as would be needed in an offload model. This allows the programmer to treat a cluster of KNCs as a regular cluster of homogenous MPI nodes and thus improves the ease of programming. Our code initially used the QMP message passing layer over MPI [11].

On the other hand, although communication across the KNCs using MPI directly is optimized for latency, the achievable peak bandwidth is quite low due to hardware issues unrelated to Xeon Phi. As we scale LQCD in a cluster, the boundaries that are exchanged with the neighbors vary between 256KB and several MBs for large problems such as one would schedule on a Xeon Phi and for these message sizes the bottleneck is typically the communication bandwidth.

In particular, experimentation has shown that only about 1 GB/s bandwidth can be achieved using MPI directly, and this was insufficient for strong scaling to a large number of nodes. To overcome this issue, we implemented a reverse proxy that relays the large message network data to CPUs, which in turn sends the data to the destination CPU, similar to the design proposed in [12]. We use this proxy to handle the bandwidth limited nearest neighbour communications via the Xeon-E5 CPU. We assign a CPU core to process requests from local KNCs for extracting the data from local KNC memory to host memory via DMA and send the data to destination CPU. Similarly, at the destination, the CPU core receives the data and copies the data from host memory to KNC memory. The whole process is performed in a pipelined manner by splitting the application data into several small chunks. The chunk sizes for a given application message are also chosen dynamically since smaller chunk sizes can amortize the startup overheads but at the cost of lower bandwidth while larger chunk sizes give good bandwidth but may expose startup overheads. We use a memory mapped request and response queue model [13] control message handshake between CPU and Xeon Phi. We stress that this proxy is very lightweight and does not require much in the way of CPU resources.

4 Hardware Setup and Experiments

Our numerical experiments consist of two kinds of measurements: the performance of our Dslash operator and the performance of the Conjugate Gradients solver. We have also measured performance of both, for comparison, using SNB-EP as well as NVIDIA Kepler K20m GPUs.

Two different kinds of Xeon Phi systems were used: nodes from the Endeavor Xeon Phi Cluster operated by Intel and nodes from from the Jefferson Lab (JLab) 12m cluster.

An Endeavor cluster node is comprised of dual socket Intel Xeon E5-2670 (SNB-EP) CPUs with 8 cores per socket, and two Xeon Phi co-processors with 61 cores each running at 1.1 GHz. The Xeon Phi has SKU B1PRQ-7110P (7110P from here on), using B1 stepping silicon, running Intel MPSS version 2.1.3552-1. The 7110P has 7936 MB of GDDR memory running at 2.75GHz giving a GDDR speed of 5.5GT/s. These nodes are connected with an FDR Infiniband interconnect. For compilation we used Intel Composer XE version 13.0.0 and Intel MPI version 4.1.0.027. Nodes of the JLab 12m cluster contain dual socket Xeon E5-2650 CPUs running at 2.0 GHz. They also contain four Xeon Phi 5110P co-processors (5110P from here on), each of which has 60 cores running at 1.053GHz. The 12m nodes run MPSS version 2.1.4346-16 (Gold) and are connected by FDR Infiniband, tho we have not run multi-node Xeon Phi tests on these systems. The node runs CentOS 6.2. In our tests the Xeon Phi nodes were booted with icache snooping turned off.

Our Kepler K20m measurements were made on a node of the Jefferson Lab 12k cluster. The base nodes (chassis, motherboard, CPU, fabric) are the same as the 12m cluster node described previously. We used the gcc-4.4.6 and CUDA Toolkit v5.0 for compilation, and ran using version 304.54 of the CUDA driver. Our reference measurements used the publicly available QUDA [6] software package for lattice QCD on GPUs[4] in a pure single precision mode. We have also measured the performance using Kepler K20c and the results were within experimental noise, indentical to the Kepler K20m results.

We have chosen 5 volumes, on which to run our timing tests. We were driven in our choice by the spatial sizes of 24^3, 32^3, 40^3 and 48^3 sites for the first four of these, aiming to make the temporal direction as large as we could fit on the device, in order to mimic a capacity mode of operation, where as few nodes as possible are used to perform a calculation. However due to memory limitations we had to vary the T-extent and in one case the Z extent. The volumes thus chosen were $24^3 \times 128$, $32^3 \times 128$, $40^3 \times 96$ and $48^2 \times 24 \times 64$ sites respectively. Our fifth volume, of $32 \times 40 \times 24 \times 96$ was chosen to allow maximum performance on 60 cores without having to split the time direction over the cores. Using 4×4 blocks in the Y and Z directions, this volume can ideally use 60 cores as a plane of 10×6 cores.

5 Single-Node Results

Our single node measurements for Wilson Dslash are shown in figure 3. We show both the case when gauge fields are compressed and when they are uncompressed. We show performance for all 4 systems considered. The different colored bars correspond to different volumes.

We can see that Sandy Bridge system seems not to benefit much from the gauge field compression. It does, however benefit both Xeon Phi platforms and the NVIDIA k20m platform as expected. We note that in the $V = 32 \times 40 \times 24 \times 96$ case, performance hits 300 GFLOPS on the 7110P, but recall that this is an 'ideal volume', for this 61 core device. Indeed we do not see this level of performance for our 60 core part.

[4] We used git commit-ID: 541c66ba1a0eca11eb555dc8de6686cd54383c6c, master branch, Feb-04, 2013, available from https://github.com/lattice/quda

We point out that a dual socket Xeon E5-2680 (2.6GHz) CPU can sustain performances that are approximately half the speed of a the Xeon Phi 5110 – roughly 120 GFLOPS in the case of the Sandy Bridge, vs roughly 250 GFLOPS in the case of the Xeon Phi 5110 with compression enabled.

The performances seen on the Xeon Phi and the NVIDIA K20m are very close to each other. In the case Wilson Dslash with compressed gauge fields, the K20m is in the same ballpark as the XeonPhi 5110P, with some volumes running slightly faster on one, and some running slightly faster on the other. The 61 core Xeon Phi 7110P appears the fastest amongst the systems tested, from the point of view of Dslash.

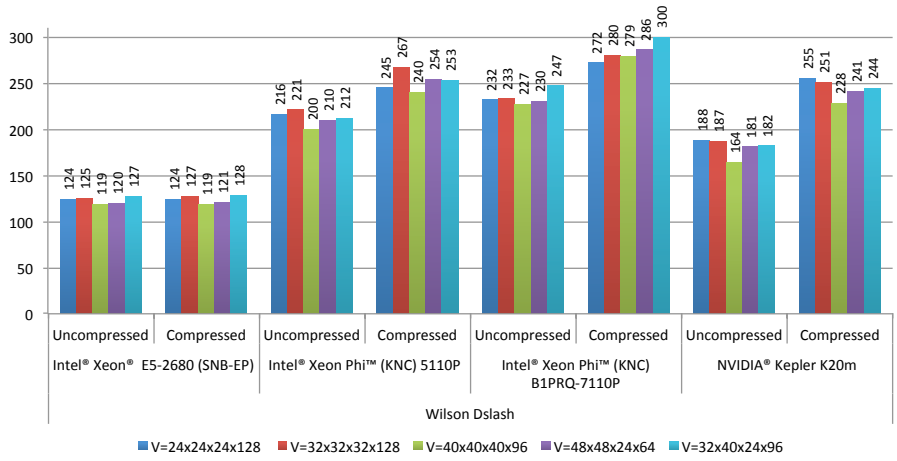


Fig. 3: Performance of Wilson Dslash for various volumes, for Xeon E5 (Sandy Bridge), XeonPhi 5110P, XeonPhi 7110P and NVIDIA K20m, Vertical axis shows performance in GFLOPS

The performances of the Conjugate Gradients algorithm are shown in figure 4. We consider the same volumes as for the Dslash. Here we find that for uncompressed gauge fields the K20m and the Xeon Phi 5110P are nearly identical in terms of speed. However, when using compressed gauge fields, the K20m performs faster than our Xeon Phi 5110P and rivals the 7110P. Again, we observe roughly a factor of 2 in performance between the dual socket Sandy Bridge EP system and the Xeon Phi 5110P.

To understand these results let us consider the following: with perfect spinor reuse, using gauge compression (but not counting the extra flops it incurs) the memory-bound performance of Wilson-Dslash is $\frac{1320}{4*(24*2+12\times8)/B_m}$ GFLOPS [5], where B_m is the memory bandwidth in GB/sec. The STREAMS triad bandwidth on KNC is $\sim$150 GB/s using four threads, similar to `aypx2` in Tab. 1. Hence, the peak achievable Dslash performance is $\sim$354GFLOPS of which we sustain 280GFLOPS. This is 80% of what is achievable. The 20% deviation from this idealized model is due to redundant memory traffic from the block boundaries resulting from blocking the volume in three dimen-

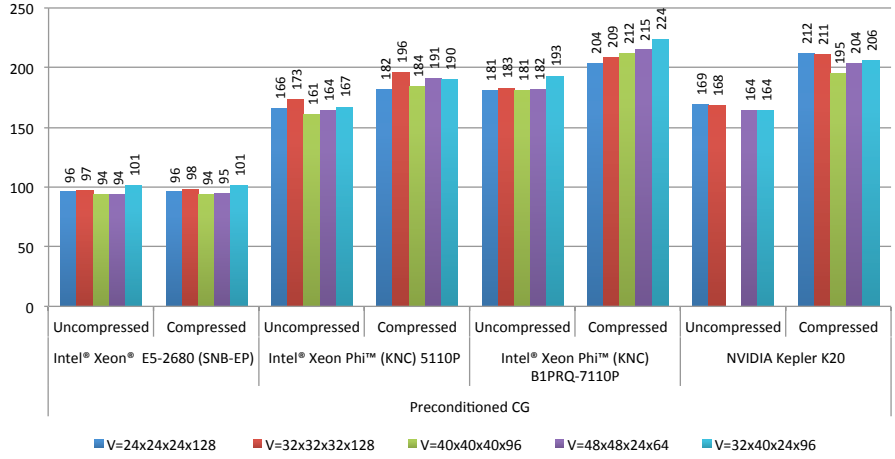


Fig. 4: Performance of Conjugate Gradient solver for various volumes, for Xeon E5 (Sandy Bridge), XeonPhi 5110P, XeonPhi 7110P and NVIDIA K20m. Vertical Axis shows performance in GFLOPS

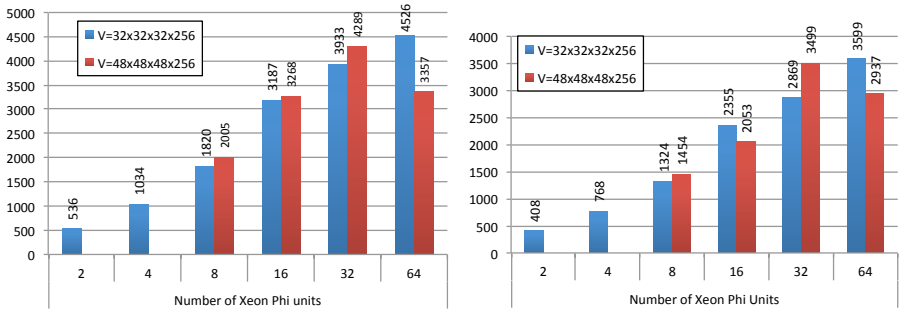
sions and our inability to use all the cores in cases where grid dimensions are not divisible by core grid dimensions. This is mildest for the $32\times40\times24\times96$ site volume where, as a result, we achieve 300GFLOPS. On the SNB-EP, memory bandwidth is about half that of KNC, while on the NVIDIA K20m it is comparable to KNC. Hence the SNB-EP runs at roughly half the speed of, while the NVIDIA K20m performs similarly to KNC.

6 Multi-Node Results

Our multi-node experiments were carried out on up to 64 nodes of the Endeavor cluster described in sec. 4. In our tests we used one KNC processor per cluster node.

Figure 5 demonstrates the strong scaling performance of our code on lattices of size $V = 32^3 \times 256$ and $V = 48^3 \times 256$ sites. We note that our $V = 32^3 \times 256$ scaling study was carried out with the first version of our code, which required L_{xh} to be a multiple of 16 sites but allowed for full 3-dimensional communications patterns, while our $V = 48^3 \times 256$ study was carried out with the more general version described in Sec. 3, which allows for different values of soa, but is restricted at present to at most 2-dimensional communication.

The strong scaling performance in Fig. 5 indicates that we do scale up to 16 nodes on the $32^3 \times 256$ problem. However, the efficiency drops to 44% on 32 nodes and 25% on 64 nodes demonstrating 3.9 TFLOPS and 4.5 TFLOPS, respectively. This is due to the local problem decreasing to a sufficiently small size, that the overhead of copying, communicating and processing the boundaries is exposed. We measured these overheads for 32 node strong scaling experiment for the $V = 32^3 \times 256$ workload. We observed that 50% of the time was spent in computing the internal volume and 25%



(a) (b)

Fig. 5: Strong Scaling multi-node Performance on the Endeavor cluster: (a) Wilson Dslash, (b) Conjugate Gradients. The vertical axis shows the performance in GFLOPS

of the time was spent in computing the boundaries. Although the communication was overlapped with computation, 13% of the communication was still exposed since the body computation finished earlier than the communication. The rest of the overhead (12%) was due to face projection and boundary copying.

In turn, the Conjugate Gradients algorithm sustained a performance of up to 3.6 TFLOPS for the $32^3 \times 256$ lattice. We also see that the CG solver has a roughly 20% slowdown as compared to the performance of the Wilson Dslash kernel of 4.5 TFLOPS. This slowdown is due to the additional computation incurred by the linear solver in our linear algebra operations and the synchronization overheads incurred in global reduction operations (global sums) performed across the MPI cluster. Although beyond the scope of this work, we expect we could improve our solver performance using a variant of pipelined CG [14].

7 Related Work

There are several implementations of LQCD Dslash implementations in the literature, many of them targeting novel architectures (of the time of their writing) such as GPUs. Optimizing for the Xeon architecture is described by us in [5], whereas recent efforts to optimize for BlueGene/Q architecture are presented in [9,15]. GPU implementations using CUDA are presented in [6,16], while an OpenCL version is in [17,18]. Autotuned blocking schemes for Wilson Dslash have been investigated on GPUs in [19].

Historically the implementations on various architectures are too numerous to list. We will content ourselves with mentioning the implementation of [20] on the QCDSP supercomputer and of [21] on the BlueGene/L, since these contributions both won Gordon Bell prizes for cost effective supercomputing in 1998 and 2006.

Code generators for writing efficient assembly code are described in [22] and [23]. Reverse offload style communications are described in [12] for the Road Runner Supercomputer, and in [24] for GPU applications.

8 Conclusion

We have detailed our approach to implementing the Wilson Dslash operator of LQCD, and have presented performance results for both single and multi-node settings. Our single node Dslash operator sustains ~ 250 GFLOPS and ~ 280 GFLOPS on Xeon Phi 5110P and 7110P devices respectively. Our multi-node implementation has been run on up to 64 Xeon Phi devices and has been strong scaled up to 4.5 TFLOPS for Wilson-Dslash, and 3.6 TFLOPS for the CG solver on a lattice of $32^3 \times 256$ sites.

We compared our single node results with performances from the QUDA library on NVIDIA K20m GPUs, running purely in single precision to have like for like tests, and found them to be competitive. We note, however, that QUDA can gain additional performance on the GPUs by employing 16-bit precision in mixed precision solvers. We are considering implementing this approach as potential future work.

To achieve these performances required attention to expose parallelism on core, SIMT thread and vector instruction levels on a single KNC (Xeon) as well as cache-blocking techniques. To achieve the multi-node performance we used a reverse communication proxy. In order to utilize fully the vector capabilities of the architecture and to mimimize memory latency through software prefetching, we wrote a simple code-generator. A notable outcome of our work is an infrastructure which could be retargeted to other vector formats. In particular, we sustained excellent performance on Xeon E5-s ($\approx$ 125 GFLOPS single precision in the Dslash) simply by re-targetting the code-generator to emit AVX intrinsics, and re-tuning our blocking factors. This demonstrates a performance portability aspect of our infrastructure.

Our future work will include improving our blocking strategy, further optimization of our code for multiple nodes, implementing other formulations of LQCD, and investigating the potential of a hybrid code using both the host CPU and the Xeon Phi(s).

9 Acknowledgements

B. J. gratefully acknowledges funding through U.S. DOE Grants: DE-FC0 2-06ER41440 and DE-FC02-06ER41449 (USQCD SciDAC project) Notice: 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.

References

1. Creutz, M.: QUARKS, GLUONS AND LATTICES Cambridge, Uk: Univ. Pr. (1983) 169 P. (Cambridge Monographs On Mathematical Physics).
2. Wilson, K.G.: Quarks and Strings on a Lattice. In Zichichi, A., ed.: New Phenomena in Subnuclear Physics. Plenum Press, New York (1975) 69
3. Hestenes, M.R., Stiefel, E.: Methods of Conjugate Gradients for Solving Linear Systems. Journal of Research of the National Bureau of Standards **49**(6) (December 1952) 409–436
4. van der Vorst, H.A.: Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems. SIAM Journal on Scientific and Statistical Computing **13**(2) (1992) 631–644

5. Smelyanskiy, M., Vaidyanathan, K., Choi, J., Joó, B., Chhugani, J., Clark, M.A., Dubey, P.: 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. SC '11 (2011) 69:1–69:11
6. Clark, M.A., Babich, R., Barros, K., Brower, R.C., Rebbi, C.: Solving Lattice QCD systems of equations using mixed precision solvers on GPUs. Comput. Phys. Commun. **181** (2010) 1517–1528
7. Nguyen, A.D., Satish, N., Chhugani, J., Kim, C., Dubey, P.: 3.5-d blocking optimization for stencil computations on modern cpus and gpus. In: SC. (2010) 1–13
8. Babich, R., Clark, M.A., Joó, B.: Parallelizing the QUDA Library for Multi-GPU Calculations in Lattice Quantum Chromodynamics. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. SC '10 (2010) 1–11
9. Boyle, P.: The BlueGene/Q supercomputer. PoS **LATTICE2012** (2012) 020
10. : MPI: A Message-Passing Interface Standard. Mar 1994
11. Joó, B.: SciDAC-2 software infrastructure for lattice QCD. Journal of Physics: Conference Series **78**(1) (2007) 012034
12. Pakin, S., Lang, M., Kerbyson, D.J.: The reverse-acceleration model for programming petascale hybrid systems. IBM Journal of Research and Development **53**(5) (sept. 2009) 8:1 –8:15
13. Heinecke, A., et al.: Design and Implementation of the Linpack Benchmark for Single and Multi-Node Systems Based on Intel(R) Xeon Phi(TM) Coprocessor. In: Proceedings of IPDPS Conference, accepted for publication. (2013)
14. Strzodka, R., Göddeke, D.: Pipelined mixed precision algorithms on FPGAs for fast and accurate PDE solvers from low precision components. In: IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2006). (April 2006) 259–268
15. Doi, J.: Peta-scale lattice quantum chromodynamics on a blue gene/q supercomputer. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. SC '12, Los Alamitos, CA, USA, IEEE Computer Society Press (2012) 45:1–45:10
16. Alexandru, A., Lujan, M., Pelissier, C., Gamari, B., Lee, F.X.: Efficient implementation of the overlap operator on multi-GPUs. (2011)
17. Bach, M., Lindenstruth, V., Philipsen, O., Pinke, C.: Lattice QCD based on OpenCL. (2012)
18. Kowalski, A., Shen, X.: Implementing the Dslash Operator in OpenCL. College of William and Mary Technical Report (2010)
19. Clark, M., Babich, R.: High-efficiency lattice qcd computations on the fermi architecture. In: Innovative Parallel Computing (InPar), 2012. (may 2012) 1 –9
20. Chen, D., et al.: QCDSP machines: design, performance and cost. In: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM). Supercomputing '98, Washington, DC, USA, IEEE Computer Society (1998) 1–6
21. Vranas, P., et al.: The BlueGene/L supercomputer and quantum ChromoDynamics. In: Proceedings of the 2006 ACM/IEEE conference on Supercomputing. SC '06, New York, NY, USA, ACM (2006)
22. Boyle, P.A.: The bagel assembler generation library. Computer Physics Communications **180**(12) (2009) 2739 – 2748 40 YEARS OF CPC: A celebratory issue focused on quality software for high performance, grid and novel computing architectures.
23. Pochinsky, A.: Writing efficient QCD code made simpler: QA(0). PoS **LATTICE2008** (2008) 040
24. Chen, J., Watson, W., Mao, W.: GMH: A Message Passing Toolkit for GPU Clusters. In: Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on. (dec. 2010) 35 –42