2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

Exploiting Very-Wide Vectors on Intel Xeon Phi with Lattice-QCD kernels

Andreas Diavastos*†, Giannos Stylianou*, Giannis Koutsou*
*Computation-based Science and Technology Research Centre, The Cyprus Institute
{a.diavastos, g.stylianou, g.koutsou}@cyi.ac.cy
†Department of Computer Science, University of Cyprus
diavastos@cs.ucy.ac.cy

***Abstract*—Our target in this work is to study ways of exploring the parallelism offered by vectorization on accelerators with very wide vector units. To this end, we implemented two kernels that derive from the Wilson Dslash operator and investigate several data layout techniques for increasing the scalability of lattice QCD scientific kernels suitable for the Intel Xeon Phi. In parts of the application where real numbers are used for computation, we see a 6.6x increase in bandwidth compared to scalar code, thanks to the auto-vectorization by the compiler. In other kernels where arithmetic operations on complex numbers dominate, our hand-vectorized code outperforms the auto-vectorization of the compiler. In this paper we find that our proposed *Hopping Vector-friendly Ordering* allows for more efficient vectorization of complex arithmetic floating point operations. Using this data layout, we manage to increase the sustained bandwidth by approximately 1.8x.**

***Keywords*-Lattice QCD; Xeon Phi; many-cores; accelerators;**

I. Introduction

An emerging trend in the current design of supercomputers is the use of accelerators as the main source of computing power. This is evident by the fact that four of the top ten supercomputers [1] today use accelerators as co-processing units of the CPU. The increase in the use of accelerators is a consequence of the fact that they allow packing a higher floating point performance within a smaller power budget compared to regular CPUs and this is achieved by using special purpose hardware. This hardware though requires additional effort to make use of, such as CUDA cores which usually require re-factoring of compute kernels such that they are suitable for streaming floating point engines. Another example are the wide SIMD floating point units on the Xeon Phi that also require re-thinking of data layout to match the SIMD vector width.

In this work we focus on performance gains which can be achieved by wide vector units on accelerator-based systems and in particular systems with Xeon Phi coprocessors. To efficiently use these vector units requires either relying on the auto-vectorization by the compiler in cases where this is possible, or alternatively in using so-called intrinsic functions, i.e. vector operations exposed as functions to the programming language. To utilize the vector units of the Xeon Phi we use computational kernels arising from the field of lattice Quantum Chromodynamics (QCD). QCD is the part of the Standard Model of physics which describes the strong interactions, which in turn is responsible for the forces which hold the quarks within protons and neutrons. Phenomena which are dominated by the strong interaction include quark-gluon plasma, which was the state of the very early universe and the birth, life, and death of stars. Currently, the most well established way to obtain quantitative results from QCD is via simulation because of the vast size of data and the large computation intensity of QCD applications. The need for more computing power put the Lattice QCD researchers among the earliest adopters of innovative computing devices that offer more performance [2], [3], [4].

In this study, we present a bandwidth performance analysis of the very-wide vectors of the Intel Xeon Phi for single-precision and complex double-precision data using compiler auto-vectorization and hand-coded vector intrinsics. We evaluate several techniques involving reordering of data and scheduling of floating point operations, that help with the vectorization of lattice QCD kernels on the Intel Xeon Phi. Our results show that when operating on complex data, hand-coded vectorization produces more efficient code than compiler auto-vectorization. Furthermore we show that our *Hopping Vector-friendly ordering* technique is optimal for stencil kernels such as those in lattice QCD applications.

II. Intel MIC Architecture

The Xeon Phi coprocessor [5] (codename Knights Corner) is part of the Many Integrated Core (MIC) Architecture family from Intel. Each coprocessor is equipped with up to 61 processor cores connected by a high-performance on-die bidirectional ring interconnect. Each coprocessor includes 8 memory controllers supporting up to 16 GDDR5 channels (2 per memory controller) with a theoretical aggregate bandwidth of 352 GB/s. Each core is a fully functional, in-order core, that supports 4 hardware threads. The architecture of the Xeon Phi cores is based on the x86 Instruction Set Architecture (ISA), extended with 64-bit addressing and 512-bit wide vector instructions and registers that allow for significant increase in parallel computation. With a 512-bit vector set, a throughput of 16 double-precision or 32 single-precision floating point operations can be executed on each core per cycle when assuming a throughput of one fused multiply-add operation per cycle. Each core has a 32 KB L1

978-1-4673-8776-7/16 $31.00 © 2016 IEEE
DOI 10.1109/PDP.2016.116

data cache, a 32 KB L1 instruction cache and a 512 KB L2 cache. The L2 caches of all cores are interconnected with each other and the memory controllers via a bidirectional ring bus, effectively creating a shared, cache-coherent last-level cache of a total of 32 MB.

Although coprocessors in general are used as accelerators for offloading computation from host processes, the Xeon Phi is more flexible with the execution of applications [6]. The Xeon Phi offers three different modes of execution: the *Offload mode* where the host CPU offloads the computation to the coprocessor in the form of parallel threads, similar to a General Purpose GPU (GPGPU); the *Coprocessor-only mode* where the parallel application is launched directly on the coprocessor that works as an independent compute unit; and the *Symmetric mode*, where multiple processes of the same program can be launched on both the host CPU and the coprocessor similar to any other message passing model.

III. Benchmark Kernels

Lattice Quantum Chromodynamics (LQCD) [7] is a discrete formulation of QCD on a finite 4-dimensional space-time lattice that enables numerical simulation of QCD using Monte-Carlo methods. LQCD is the most well established method for solving the fundamental theory of strong interactions. Here we use components of the so-called Wilson Dslash operator [8] as the kernels we target for optimization. The Wilson Dslash operator applied to a vector ψ is given in Equation 1.

$$\sum_{x'} D[u](x,x')\psi(x') = \frac{1}{2\kappa}\psi(x) + \frac{1}{2}\sum_{\mu=0}^{3}[(I-\gamma_\mu)u_\mu(x)\psi(x+\hat{\mu}) + (I+\gamma_\mu)u_\mu^{-1}(x-\hat{\mu})\psi(x-\hat{\mu})] \quad (1)$$

where κ is a parameter of the theory, I is the 4×4 unit matrix, γ_μ are the four 4×4 gamma-matrices in some basis (e.g. the Dirac matrices), $\psi(x)$ is a so-called *spinor field*, x and x' are space-time (four components) coordinates, $u_\mu(x)$ represents a so-called *gauge field*, and μ is an index for the direction (e.g. $\mu = 0, 1, 2, 3$ may represent the t, x, y, z directions). Lattice QCD simulations typically spend 70-80% of the execution applying this operator, making it important to take advantage of any possible optimizations.

A. Stencil operator

The first kernel we implement is a nearest-neighbor 2D stencil operation. The idea is to emulate aspects of the data reuse required in the Wilson-Dirac equation. More precisely, the stencil we implement resembles a discrete 2D Laplacian operation applied to a field ϕ given by Equation 2.

$$\hat{\phi}(x) = \frac{1}{1+4\sigma}\left\{\phi(x) + \sigma[\phi(x+\hat{i}) + \phi(x-\hat{i}) + \phi(x+\hat{j}) + \phi(x-\hat{j})]\right\} \quad (2)$$

where x is a coordinate on a 2-dimensional grid with dimensions denoted by i and j and σ being a constant. This specific choice of normalization allows repeatedly applying the operator on a vector while keeping the numerical values of the elements to be of the same order. To calculate the element $\hat{\phi}(x)$ on the left-hand-side of Equation 2 we need to load the element $\phi(x)$ and the four neighboring elements ($\phi(x+\hat{i})$, $\phi(x-\hat{i})$, $\phi(x+\hat{j})$ and $\phi(x-\hat{j})$) from a 2-dimensional grid.

In we assume consecutive iterations of the stencil operation on an array with ϕ elements that is aligned in memory and elements are stored in lexicographic site order, then for an architecture with wide vector units, a number of following elements will not be aligned. Therefore, to vectorize this operation, shuffle operations would be required to align the appropriate elements at the right position of the vector register.

B. SU(3) Multiplication Kernel

The second kernel we implemented is the component of the Wilson Dirac equation which involves the multiplication with the gauge-links, which are elements of the special unitary group SU(3). As mentioned this is effectively a multiplication of an array of complex 3×3 matrices u, with an array of complex 3×4 ψ vectors (ψ fields are vectors in color space). Complex number types, such as the C99 complex type, are usually stored as a structure or array of two elements, with the real part in the first element and the imaginary part in the second element. A complex multiplication therefore involves cross-terms, i.e. multiplying the real part of the first operand with the imaginary part of the second operand and vice-versa. If this operation is to be vectorized, one therefore needs to reshuffle the operands' real and imaginary parts.

IV. Data Layout Optimizations

A. The Short Implementation

The fact that the gauge-links are elements of the SU(3) group has been exploited in lattice QCD applications to reduce their memory requirements. Namely, by the definition of an SU(3) matrix, that its Hermitian conjugate is its own inverse: $uu^\dagger = 1$, one can reconstruct the ninth element from the other eight. The advantage to this is that eight elements align in memory, and therefore every such truncated, or short, gauge-link will start on an aligned boundary. The disadvantage is that one has to recompute the ninth element of the gauge-link every time it is to be used. This data layout can only be applied to SU(3) matrices, thus it is implemented only for the SU(3) Multiplication kernel.

B. Data Padding

Padding is used regularly when requiring that every element of an array of structures be aligned in memory. We include a version with padding of the gauge-links for completeness. We note however that an architecture with potentially very large vector units, of a size multiple times

the size of a gauge link, would prohibit padding as an optimization technique. This is also SU(3)-specific implementation, thus it is only applied to the SU(3) Multiplication kernel.

C. The Vector-friendly Ordering Technique

The simplest way of reordering data to allow easier vectorization, is to change the order in which the indices run. In our case we have an array of gauge-links which are stored in memory with the color indices running fastest:

```
_Complex double u[NV][NC][NC];
```

where `NV` is the number of lattice sites and `NC`=3. Defining the lattice sites to run faster however:

```
_Complex double u[NC][NC][NV];
```

would allow easier vectorization, with the restriction that the number of lattice sites `NV` is a multiple of the vector register length. This layout is vector-friendly, because the inner-loop of the computation that works on lattice sites can be easily unrolled and vectorized. We call this the *Vector-friendly Ordering* (VFO) technique. The disadvantage is that for large `NV` this technique is not optimal in data reuse, since other data used in the computation must be retrieved three times from memory, once for each `NC` iteration.

D. The Hopping Vector-friendly Ordering Technique

The *Hopping Vector-friendly Ordering* (Hopping VFO) follows naturally from the requirement for cache locality of VFO above. The idea is to re-order the data such that part of the volume index `NV` runs faster. The extent by which it runs faster depends on the vector width. For instance one can define the data arrays as:

```
_Complex double u[NV/NW][NC][NC][NW];
```

where `NW` is the number of `_Complex double` elements which fit in a vector register (e.g. for the Xeon Phi `NW` = 4). The implications of this layout are more pronounced when one considers the 2D stencil operation. In this case, one would re-define the arrays from:

```
float phi[L][L];
```

to

```
float pho[L][L/NW][NW];
```

with the requirement that `L` is a multiple of `NW`. One then re-orders the entries as such:

```
float phi[L][L];
float pho[L][L/NW][NW];
for(y=0; y<L; y++)
  for(x=0; x<L/NW; x++)
    for(w=0; w<NW; w++)
      pho[y][x][w] = phi[y][w*L/NW + x];
```

This allows vectorizing the stencil operation along the `w` component:

```
for(w=0; w<NW; w++)
     pho_out[y][x][w] = a*pho_in[y][x][w]
              + b*(
                   pho_in[y+1][x][w] +
                   pho_in[y][x+1][w] +
                   pho_in[y-1][x][w] +
                   pho_in[y][x-1][w]);
```

Since the elements `pho_in[y][x][w]` for `w`=0,1,2,...,`NW-1` are maximally spaced apart in the x-direction, the `w` loop can be unrolled and vectorized across the fastest running index without requiring shuffles. The only exception is when `x` is on a boundary, i.e. the case when `x`=0 and `x`=`L/NW-1`. Only for these two cases does one require shuffle operations, to impose the periodic boundary conditions.

V. Experimental Evaluation

Our hardware setup was an Intel Xeon Phi 7120P with 61 cores and 4 threads per core, totaling 244 threads. We reserved one core for the operating system and thus used the remaining 60 cores for the application execution in *Coprocessor-only mode*. The coprocessor we used has a total of 16 GB of main memory and runs at 1.238 GHz. To cross-compile our benchmarks for the Xeon Phi we used the Intel *icc* v.14.0.1 compiler with the *-mmic* flag indicating the MIC architecture and the *-O3* optimization flag that includes auto-vectorization. For the GPU experiments we used an NVIDIA Kepler K20m card with 2496 CUDA cores. The total memory of the GPU card is 5 GB and the working frequency is 706 MHz. To compile for the GPU we used the *nvcc* NVIDIA compiler, v.5.5.0. The performance metric we quote for the two kernels is sustained bandwidth, that represents the number of bytes read and written during the total calculation, divided by the execution time. For performance reasons all our implementations use one-dimensional arrays for storing the fermion and gauge fields. The *baseline* shown in all the results of the two kernels is the original parallel implementation of the kernel with no vector optimizations.

In Figure 1, we present the results of the *2D Stencil operator*. Compared to the baseline, the auto-vectorization of the compiler manages around a 6.6× bandwidth increase. Our VFO technique seems to produce a slight overhead when reordering the data that cannot be eliminated for up to 60 threads. When we use the SMT capabilities of the Xeon Phi and increase the number of threads per core to 2 and 4, we manage to obtain as much performance as the auto-vectorization of the compiler. This is due to the higher I/O throughput achieved by hiding the memory latency through the use of multiple threads.

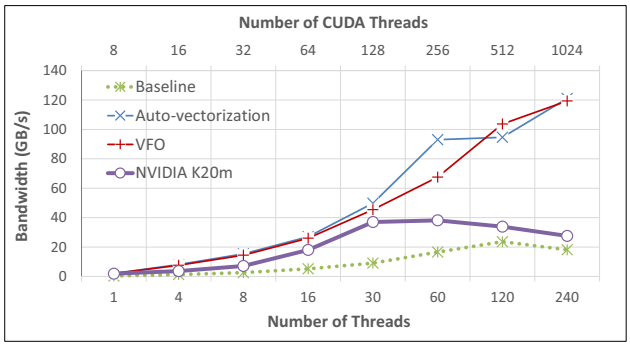

Figure 1. Performance of the 2D stencil with size L = 2048×2048 length vectors, that is the outer dimension of the gauge-link array.

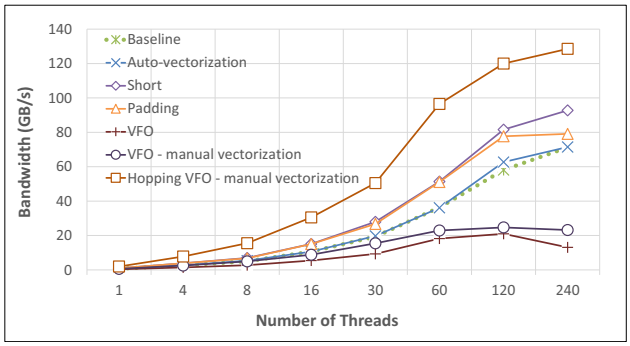

Figure 2. Performance for the SU(3) Multiplication kernel for the largest data set of our evaluation L = 3840×1024.

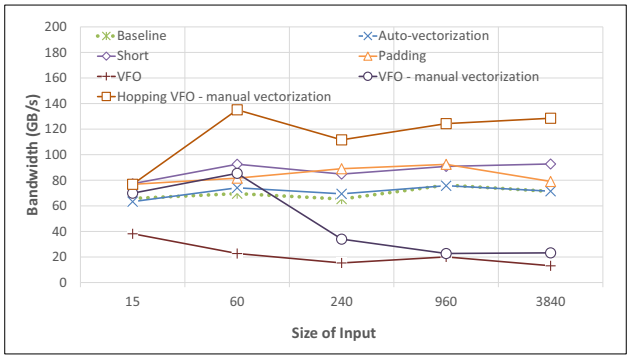

Figure 3. Performance for the SU(3) Multiplication kernel for 240 threads and varying the input size (L = Size of Input×1024).

For this kernel we also implemented a version with moderate optimizations for the GPU. The results show that for a small number of threads the GPU performance is close to the Xeon Phi but when we increase the number of threads on both coprocessors, the Xeon Phi significantly outperforms the GPU. A possible explanation is that the input size used for these tests is not large enough for the GPU to hide all memory latencies and also not large enough to efficiently utilize all the CUDA threads available.

The *SU(3) Multiplication kernel* uses complex floating point arithmetic, which means that each element of the matrix has a real and an imaginary part, with each one being a double-precision floating point number. In Figure 2, we present the performance results of this kernel on the Intel Xeon Phi. The auto-vectorization of the compiler in this case does not perform because of the data dependencies involved in complex arithmetic, which may hide auto-vectorization opportunities from the compiler. To overcome this, we implement and evaluate the data layout optimizations of Section IV while keeping the vectorization flag of the compiler `ON`.

In the *Short implementation*, where we remove one element, we vectorize all operations including the calculation of the missing element. We observe that this change leads to a bandwidth increases of ≈1.4×. Although this technique introduces additional operations re-computing the missing element, the increased performance attained by the vectorization seems to be sufficient to hide the extra computation. The same is observed for the *Padding implementation*, where we pad the end of each u matrix in memory such that every matrix begins on an aligned memory address. This method out-performs the baseline scenario, by approximately the same factor as the *short* implementation.

Applying the *VFO ordering technique* allows a more straight forward vectorization of the kernels, with a significantly smaller number of shuffle operations. This technique introduces a significant penalty, performing more than 3× worse than the baseline. When reordering the complex data of the SU(3) kernel within this technique we inevitably reduce data re-use, and effectively increase the number of cache misses, loosing both temporal and spatial locality of the data. This becomes more obvious in Figure 3, where we see the performance as a function of the problem size. We therefore conclude that the overhead imposed by the loss of locality in cache is greater than the gains achieved through use of the vector processing.

The *Hopping VFO technique* aligns the data in vector registers in such a way that distant data are packed within the same register. This data layout is intended to on one hand ease the vectorization of the codes by requiring less shuffles in the way VFO does, and on the other hand retain data locality such that the excessive cache misses observed in VFO are mitigated. Using this technique we get an almost 2-fold increase in the bandwidth compared to the baseline and this performance is stable across different input sizes as shown by the results of Figure 3.

VI. Related Work

There are various implementations of LQCD operators in the literature that target different architectures. In this section we concentrate on those relevant to accelerator-based architectures. M.A. Clark *et al.* in [9], [10], present the QUDA library which was used in this work, for performing

calculations in lattice QCD on GPUs, leveraging NVIDIA's CUDA [11] programming platform. In a continuation of this work, Ronald Babich *et al.* present the parallel implementation of the QUDA library for multi-GPU calculations [12], that manages to increase the performance of both single- and double-precision calculations on clusters with multiple GPUs.

Simon Heybrock *et al.* in [13], propose a domain decomposition for the LQCD Wilson-Clover operator with single- and half-precision operations on the Intel Xeon Phi coprocessor. Using domain-decomposition methods they manage to reduce the data movement from and to main memory as well as via the network. In [14], the authors present their approach of implementing the Wilson Dslash operator for the Intel Xeon Phi coprocessor while using cache-blocking techniques and block-to-core methods for increasing the performance, while an evolution of this work for multi-node Xeon Phi clusters is presented in [15]. In [16], the authors propose similar data layout transformation techniques to avoid the stream alignment conflict problem found in optimized implementations of stencil computations on short-vector SIMD architectures.

VII. Conclusions

In this work we show that the performance of LQCD-based applications can significantly increase on the Intel Xeon Phi if moderate effort is devoted in optimizing the data layout. Through our study we found that the compiler auto-vectorization was limited to non complex arithmetic, and was unable to automatically vectorize the code in complex arithmetic kernels. We evaluated various methods for manipulating the data to solve this problem and we found that the *Hopping Vector-friendly Ordering* technique allows one to efficiently vectorize the application. Using this technique we can achieve $\approx 1.8\times$ performance increase for a kernel that uses complex data structures.

Acknowledgment

A. Diavastos and G. Stylianou are supported by the Cyprus Research Promotion foundation, under project "GPU Clusterware"(ΤΠΕ/ΠΛΗΡΟ/ 0311(BIE)/09). The authors would also like to thank Dr. Pedro Trancoso for his valuable comments.

References

[1] (2015) Top500.org. [Online]. Available: http://top500.org/

[2] P. Boyle, C. Jung, and T. Wettig, "The QCDOC supercomputer: Hardware, software, and performance," *eConf*, vol. C0303241, p. THIT003, 2003.

[3] F. Bodin, P. Boucaud, N. Cabibbo, F. Calvayrac, M. Della Morte *et al.*, "The apeNEXT project," *Nucl.Phys.Proc.Suppl.*, vol. 106, pp. 173–176, 2002.

[4] G. I. Egri, Z. Fodor, C. Hoelbling, S. D. Katz, D. Nogradi *et al.*, "Lattice QCD as a video game," *Comput.Phys.Commun.*, vol. 177, pp. 631–639, 2007.

[5] The Intel Xeon Phi coprocessor. [Online]. Available: http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html

[6] R. Rahman, *Intel® Xeon Phi™ Coprocessor Architecture and Tools: The Guide for Application Developers*. Apress, 2013.

[7] K. G. Wilson, "Confinement of quarks," *Physical Review D*, vol. 10, no. 8, p. 2445, 1974.

[8] A. Kowalski *et al.*, "Implementing the dslash operator in opencl," *College of William and Mary Technical Report*, 2010.

[9] 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," *Computer Physics Communications*, vol. 181, no. 9, pp. 1517–1528, 2010.

[10] QUDA: A library for QCD on GPUs. [Online]. Available: http://lattice.github.io/quda/

[11] "NVIDIA, CUDA, programming guide," 2008.

[12] R. Babich, M. A. Clark, and B. Joó, "Parallelizing the QUDA library for multi-GPU calculations in lattice quantum chromodynamics," in *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*. IEEE, 2010, pp. 1–11.

[13] S. Heybrock, B. Joó, D. D. Kalamkar, M. Smelyanskiy, K. Vaidyanathan, T. Wettig, and P. Dubey, "Lattice qcd with domain decomposition on intel® xeon phi co-processors," in *SC14 International Conference*. IEEE, 2014, pp. 69–80.

[14] B. Joo *et al.*, "Lattice qcd on intel® xeon phitm coprocessors," in *Supercomputing*. Springer, 2013, pp. 40–54.

[15] K. Vaidyanathan, K. Pamnany *et al.*, "Improving communication performance and scalability of native applications on intel xeon phi coprocessor clusters," in *IPDPS 2014 International Conference*. IEEE, 2014, pp. 1083–1092.

[16] T. Henretty, K. Stock, L.-N. Pouchet, Franchetti *et al.*, "Data layout transformation for stencil computations on short-vector simd architectures," in *Compiler Construction*, ser. Lecture Notes in Computer Science, J. Knoop, Ed. Springer Berlin Heidelberg, 2011, vol. 6601, pp. 225–245.