Editors: Volodymyr Kindratenko, University of Illinois, kindr@ncsa.uiuc.edu | Pedro Trancoso, Univ. of Cyprus, pedro@cs.ucy.ac.cy

Exploiting Very-Wide Vector Processing for Scientific Applications

Andreas Diavastos | Cyprus Institute and University of Cyprus
Giannos Stylianou and Giannis Koutsou | Cyprus Institute

The use of accelerators in compute-intensive scientific problems is growing rapidly: they pack a relatively large number of floating-point operations in a low-power profile, thus increasing the flop-to-watt ratio. In this work, we study ways of exploring the parallelism in scientific applications, using lattice quantum chromodynamics (QCD) compute kernels as our benchmark application. The parallelism available from the 512-wide vector units of the Xeon Phi accelerator is exploited by either using compiler auto-vectorization or by introducing hand-coded vectorization techniques. We see a 6.6× increase in bandwidth for certain parts of the application, thanks to the compiler's auto-vectorization, when compared to scalar code. In kernels where complex arithmetic operations dominate, hand-vectorized code outperforms the compiler's auto-vectorization and increases the sustained bandwidth by ≈1.8×.

Intel MIC Architecture

The Xeon Phi coprocessor (codename Knights Corner; www.intel.com/content/www/us/en/processors/xeon/xeon-phidetail.html) is part of Intel's Many Integrated Core (MIC) architecture family. Each coprocessor is equipped with 61 processor cores connected in a high-performance on-die bidirectional ring interconnect (see Figure 1). Each coprocessor includes eight memory controllers supporting up to 16 GDDR5 channels (two per memory controller) with a theoretical aggregate bandwidth of 352 Gbytes/s. Each core is a fully functional, in-order core that supports four hardware threads. To reduce hot-spot contention for data among the cores, a distributed tag directory is implemented such that every physical address is uniquely mapped through a reversible one-to-one address hashing function.

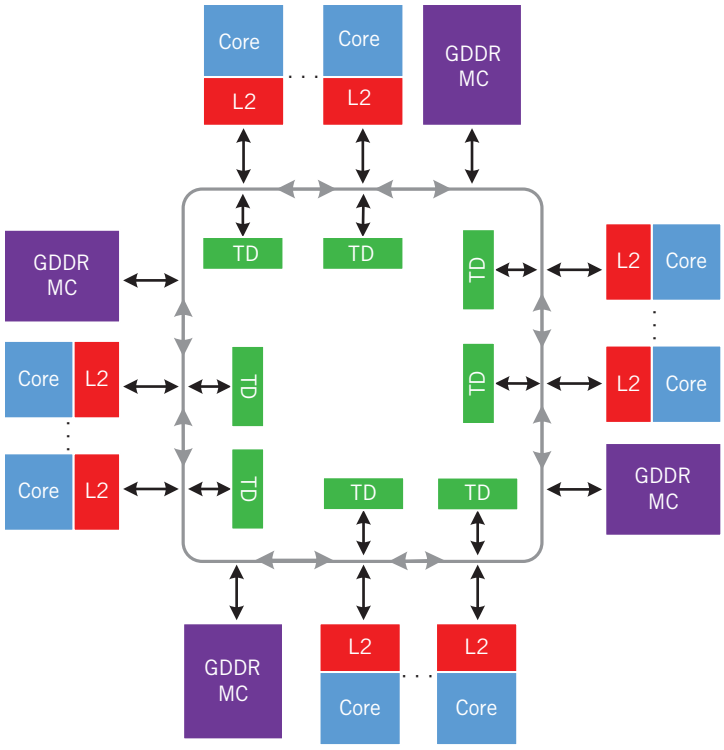

Figure 1. The Intel Xeon Phi top-level architecture. Each coprocessor is equipped with 61 processor cores connected in a high-performance on-die bidirectional ring interconnect.

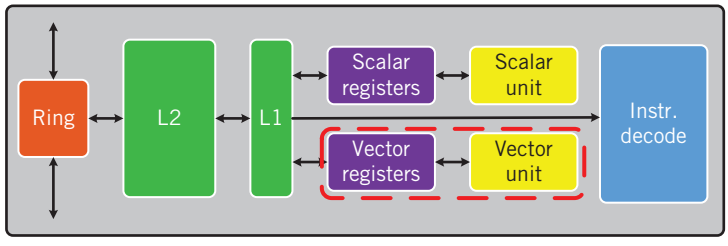

Figure 2. Top-level architecture of the cores in a Xeon Phi coprocessor. The dashed red line isolates the core's vector processing unit (VPU).

The working frequency of the Xeon Phi cores ranges between 1 and 1.3 GHz depending on the model, and their architecture 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 a twofold increase in parallelism compared to Advanced Vector Extensions (AVX) and a 4× compared to Streaming Single Instruction Multiple Data (SIMD) Extensions (SSE) technologies. The system used in this work (Intel Xeon Phi 7120P) is equipped with 61 cores working at 1.238 GHz, bringing the theoretical peak of double-precision floating-point performance to 1.208 Tflop/s. Each core has a 32-Kbytes L1 data cache, a 32-Kbytes L1 instruction cache, and a 512-Kbytes L2 cache. The L2 caches of all the 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 up to 32 Mbytes.

The Xeon Phi's vector processing unit (VPU) provides data parallelism using 512-bit wide-vector instructions, providing a throughput of 32 single-precision or 16 double-precision floating-point operations per cycle, on each core, assuming all operations are fused multiply-adds (FMA). The VPU is an extension to the P54C core on Xeon Phi and communicates with the core to execute the VPU ISA implemented in the coprocessor.[1] The core's arithmetic logic unit (ALU) feeds the VPU with instructions while receiving data from the L1 cache via a dedicated 512-bit bus. The VPU can read/write one vector per cycle from/to the vector register file or the data cache and it can do one load and one operation in the same cycle. The VPU instructions are ternary-operand with two sources and one destination, which can also act as a source for FMA instructions. This type of configuration provides approximately a 20 percent gain in performance over traditional binary-operand SIMD instructions. Figure 2 shows the top-level design of the core architecture in a Xeon Phi coprocessor.

The Xeon Phi's vector ISA is designed to address scientific, high-performance computing (HPC) applications. It supports both native 32-bit float and integer and 64-bit float operations in a coherent memory model in which both the Intel64 instructions and the vector instructions operate on the same address space. It also supports scatter/gather instructions to read or write sparse data in memory in or out of the packed vector registers. This feature of the vector architecture is intended to simplify code generation for the sparse data manipulations found in scientific applications.

Experimental Evaluation

Exploiting the parallelism available in the Intel Xeon Phi coprocessor requires efficient utilization of the vector registers. The data in memory and the arithmetic operations therefore need to be refactored such that loading and storing as well as operating on the data can be either vectorized by the compiler or hand-vectorized by explicitly hand-coding the vector intrinsics. As a benchmark, we use two components from the domain of lattice QCD—namely, a 2D stencil operation and a complex linear algebra operation. We execute both applications in coprocessor-only mode, in which the application

is launched explicitly only on Xeon Phi. In both cases, we measure the sustained bandwidth as a performance metric by knowing the effective number of bytes read and written during the total calculation and then dividing by the execution time.

We compare results when using the compiler optimization flag for auto-vectorization to results when we reorder the data in a vector-friendly way, using vector intrinsics to manually vectorize the two kernels. In addition, we include, as a baseline, the results with the parallel implementation of the kernels with no vector optimizations (scalar). The sources for the two kernels are openly available on Github (https://github.com/gpucw). We also compare the performance of two lattice QCD libraries—the QUDA library for GPUs[2] and the QPhix library for the Xeon processor family (https://github.com/JeffersonLab/qphix). The project's website provides more detailed information (http://clusterware.cyi.ac.cy).

2D Stencil Operator

Our first test is a 2D stencil operation that extends up to the nearest neighbor. 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 1:

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

where x is a coordinate on a 2D grid with dimensions denoted by i and j, and σ being a constant. This specific choice of normalization allows repeatedly applying the operator on ϕ while keeping the numerical values of the elements on the same order. Figure 3 shows the neighboring elements of a single element ϕ involved in the operation. To calculate the element $\hat{\phi}(x)$ on the left-hand side of Equation 1, 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 $\phi(x-\hat{j})$ in the 2D grid presented in Figure 3.

Figure 4 shows the results obtained from the 2D stencil operator. Compared to the baseline in our results, the compiler auto-vectorization increases the bandwidth by 6.6×. In the same graph, we see a manually vectorized version of the same kernel. This kernel requires reordering the data such that neighboring sites are scattered over different registers, allowing us to hand-vectorize the kernel with only a small number of shuffle operations required and to use vector intrinsics for the load, store, and floating-point operations.

The hand-vectorized code performs similarly to auto-vectorization—in some cases, it slightly underperforms. When using up to 60 threads, we pin one thread to a core. For the cases with 120 and 240 threads, we oversubscribe the cores, using two or four threads per core, respectively, enabling us to explore

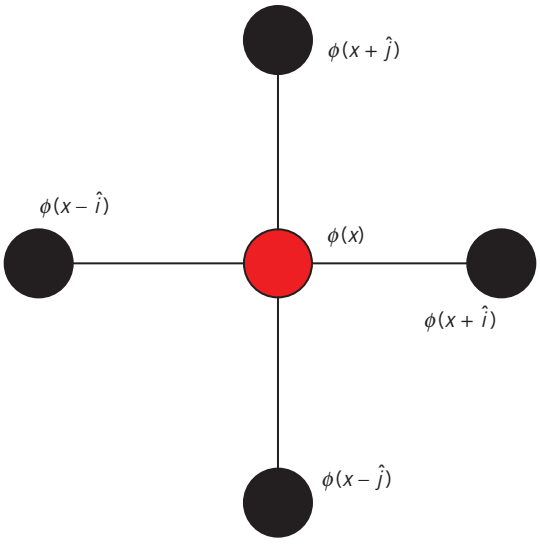

Figure 3. The elements involved in the stencil operation on site x of the 2D Laplacian. To calculate the element $\hat{\phi}(x)$ on the left-hand side of Equation 1, we need to load the element $\phi(x)$ and the four neighboring elements $\phi(x+\hat{i})$, $\phi(x-\hat{i})$, $\phi(x+\hat{j})$, and $\phi(x-\hat{j})$ in the 2D grid.

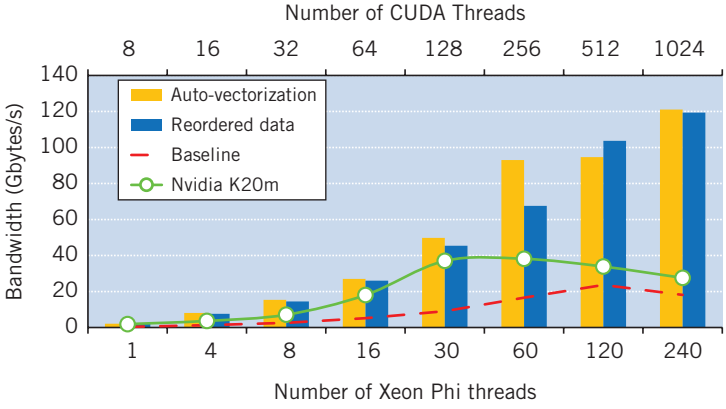

Figure 4. Bandwidth performance for the 2D stencil operation. The red dashed line shows the baseline, that is, with no vectorization. The yellow bars show Xeon Phi performance when auto-vectorization is enabled; blue bars show a vector-friendly reordering technique on the data in combination with auto-vectorization. The green circles show the stencil operation's performance on an Nvidia K20m GPU with the total number of CUDA threads on the upper *x*-axis.

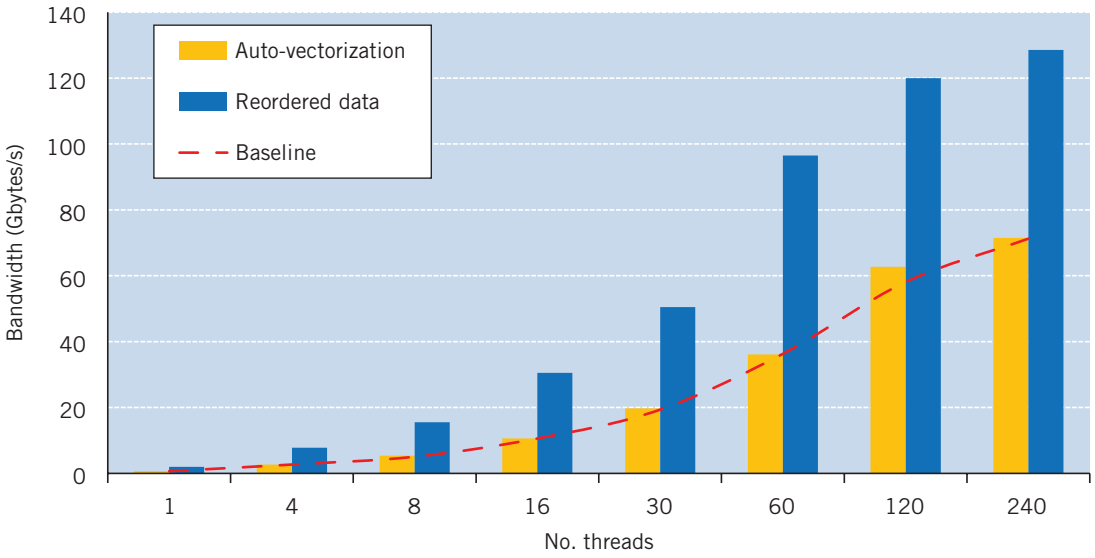

Figure 5. Bandwidth performance for the SU(3) multiplication kernel for varying number of threads. We obtain the same performance between the baseline and auto-vectorized case, an indication that auto-vectorization isn't optimizing the complex operations. Hand-vectorizing the kernel using our data reordering, on the other hand, leads to an almost twofold increase in the sustained bandwidth.

Simultaneous Multithreading's (SMT's) efficiency on Xeon Phi. In both the auto- and hand-vectorized cases, including more than one thread per core increases the sustained bandwidth, indicating that SMT on Xeon Phi is indeed beneficial for hiding I/O latencies. We note that we obtain the highest performance with 240 threads, exceeding 120 Gbytes/s. This number is consistent with the sustained bandwidth achieved when using simple stream benchmarks on Xeon Phi (https://software.intel.com/en-us/articles/optimizing-memorybandwidth-on-stream-triad).

For this kernel, we also implement a GPU version. Evaluation results show that for a small number of threads, GPU performance is close to Xeon Phi's, but when we increase the number of threads on both coprocessors, Xeon Phi significantly outperforms the GPU. The input size used (2,048 × 2,048 arrays) for these tests isn't large enough for the GPU to hide all memory latencies or to efficiently utilize all CUDA threads available. Even so, the GPU implementation outperforms the baseline implementation on Xeon Phi.

When comparing GPU performance to that of Xeon Phi, it's worth mentioning the programming effort devoted to each. For Xeon Phi, the data layout requires reordering to favor vectorization. For the GPU, the kernel involves copying to shared memory subblocks of the data within a thread block. In both cases, the source code requires a nonnegligible number of modifications. In addition, for the GPU, we must explicitly manage data communication between the CPU and the GPU; for Xeon Phi, we can maintain a single codebase for both, whether we use the compiler auto-vectorization or explicitly hand-code vector intrinsics. The fact that Xeon Phi's vector operations are exposed as functions that can be called in regular C code lets us reuse common segments between the CPU and Xeon Phi code and specify the architecture at compile time. This produces more maintainable and self-consistent source code than that of the GPU, where parallel sections must be rewritten in CUDA kernels.

In summary, although the programming effort for the GPU and Xeon Phi codes is in both cases moderate and approximately equivalent, we find better performance in our Xeon Phi implementation.

SU(3) Multiplication Kernel

The second kernel we test is a complex linear algebra operation that appears in the Wilson-Dirac operator in lattice QCD applications. Computationally, it includes the 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, that is, 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, we need to reshuffle the operands' real and imaginary parts. This kernel tests compiler auto-vectorization for complex types, as well as the improvements we can gain when hand-vectorizing complex operations. A subtler point however is the fact that a gauge link takes up 144 bytes in double precision. If we use registers wider than SSE—in this case, the 512-bit MIC registers, not every gauge link can be aligned in memory, with two possible solutions being either to pad consecutive gauge links or to load elements of multiple gauge links within the same vector unit. Both solutions have downsides: padding simplifies the number of shuffle operations required because each gauge link begins on an aligned memory boundary but reduces the effective bandwidth available. No padding requires scattering elements of one gauge link across vector registers, thus complicating the required shuffle operations but makes better use of memory bandwidth. A third solution is a reordering of the gauge links depending on the vector register width. This involves reordering the array of 3×3 matrices such that the array's index partially runs faster.

Figure 5 presents the performance results of this kernel comparing the baseline code, the code with the auto-vectorization enabled, and the code after we reorder the data and use explicit vector intrinsic calls. We obtain the same performance between the baseline and auto-vectorized case, an indication that auto-vectorization isn't optimizing the complex operations. Hand-vectorizing the kernel using our data reordering, on the other hand, leads to an almost twofold increase in the sustained bandwidth. As in the case of the 2D stencil operation, at the maximum number of threads, we achieve a sustained bandwidth of more than 120 Gbytes/sec, which is consistent with stream benchmarks.

Figure 6 evaluates a full lattice QCD implementation of the Wilson Dslash operator. These results are intended to demonstrate the effect of combining various vectorization techniques in a complete application, comparing a Xeon Phi implementation with an established GPU implementation. Specifically, we compare different input sizes for the QPhix implementation on Xeon Phi and the QUDA implementation on the Nvidia GPU, achieving an average of 9 and 10 percent peak performance on the Xeon Phi and the GPU, respectively, which is consistent with the performance reported in the literature.[3]

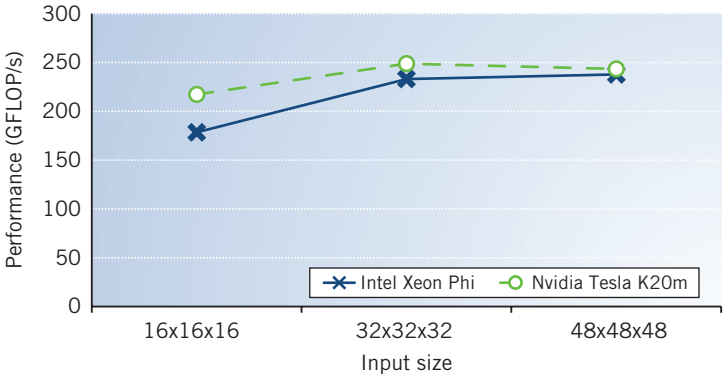

Figure 6. Comparison of sustained performance with maximum threads between an Intel Xeon Phi and an Nvidia GPU when running the same lattice QCD compute kernel, as a function of the problem size.

Although hand-coding introduces additional programming effort, the code is easy to maintain as a single source code for both CPUs and Xeon Phis by using macro directives to distinguish the architecture to be compiled for (scalar- or vector-based). At the same time, vectorizing for Xeon Phi gives a source code that also performs well on all Xeon-family processors and other architectures with wide vector units. ■

Acknowledgments

A. Diavastos and G. Stylianou are supported by the Cyprus Research Promotion foundation, under project "GPU Clusterware"(ΤΠΕ/ΠΑΗΡΟ/0311(BIE)/09).

References

1. R. Rahman, *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers*, Apress, 2013.
2. R. Babich, M.A. Clark, and B. Joo, "Parallelizing the QUDA Library for Multi-GPU Calculations in Lattice Quantum Chromodynamics," *Proc. IEEE Int'l Conf. High Performance Computing, Networking, Storage and Analysis* (SC), 2010, pp. 1–11.
3. B. Joo et al., "Lattice QCD on Intel Xeon Phi Coprocessors," *Supercomputing*, Springer, 2013, pp. 40–54.

Andreas Diavastos is a research assistant at the Cyprus Institute and a PhD candidate at the University of Cyprus. Contact him at diavastos@cs.ucy.ac.cy.

Giannos Stylianou is a research assistant at the Cyprus Institute. Contact him at g.stylianou@cyi.ac.cy.

Giannis Koutsou is an Assistant Professor at the Cyprus Institute. Contact him at g.koutsou@cyi.ac.cy.