Performance comparison of Intel Xeon Phi Knights Landing

Ishmail A. Jabbie[1], George Owen[2], Benjamin Whiteley[3]

[1]Department of Mathematics and Statistics, UMBC
[2]Department of Mathematics, Louisiana State University
[3]Department of Engineering & Aviation Sciences, University of Maryland, Eastern Shore

Advisor: Matthias K. Gobbert, gobbert@umbc.edu,
Department of Mathematics and Statistics, UMBC

Abstract

The Intel Xeon Phi is a many-core processor with a theoretical peak performance of over 3 TFLOP/s of double precision. We contrast the performance of the second-generation Intel Xeon Phi, code-named Knights Landing (KNL), to the first-generation Intel Xeon Phi, code-named Knights Corner (KNC), as well as to a node with two multi-core CPUs as baseline reference. The test code solves the classical elliptic test problem of the Poisson equation whose performance is prototypical for the computational kernel in many numerical methods for partial differential equations. The results show that the KNL can perform approximately four times faster than the KNC or than two CPUs, provided the problem fits into the 16 GB of on-chip MCDRAM memory of the KNL. The studies also confirm the nominal five times faster speed of the new high-performance MCDRAM memory in the KNL compared to the DDR4 memory of the node. We demonstrate the ease of porting code to the KNL by focusing on performance that was achieved by only re-compiling hybrid MPI+OpenMP code with a KNL flag.

Key words. Intel Xeon Phi, MPI, OpenMP, Poisson equation, Conjugate gradient method.

1 Introduction

Knights Landing (KNL) is Intel's code name for its second-generation many-integrated-core (MIC) Xeon Phi processor, that was announced in June 2014 [9] and began shipping in July 2016. The change in hardware represents a significant improvement over the first-generation Knights Corner (KNC), giving the KNL the potential to be even more effective for memory-bound problems. The Xeon Phi family is Intel's contribution to the trend towards the use of many-core processors in parallel computing. Already the first-generation Phi as co-processor in tandem with one or more host CPUs had an impact since its appearance in 2012, as exhibited by several of the highest-ranked clusters on the Top 500 list (www.top500.org) since then that use the Phi KNCs as accelerators.

The paper [20] by the chief designers of the KNL introduces the key features of the KNL. (i) The most fundamental change with the second-generation Phi is its ability to serve as stand-alone processor, i.e., without a CPU as host in a node. This means that there is potential for immediate impact on research as well as production codes, since source code can be ported to the Phi by simply compiling with an additional compile flag and without change in a first pass. (ii) A crucial improvement for performance of the second-generation KNL Phi is the 16 GB MCDRAM memory on board the chip [20]. The Phi can also access the DDR4 memory of the node, but MCDRAM is directly in the chip and is nominally 5x faster than DDR4 [20]. The MCDRAM is also nearly 50% faster than the GDDR5 memory, which is the memory on board the KNC. (iii) The bi-directional ring bus on the KNC that connects the computational cores to the on-chip memory has been replaced by a 2D mesh structure that allows for significantly more bandwidth since there

are more channels for cores to communicate. (iv) Finally, the KNL can have up to a maximum of 72 cores, while KNC has up to 61 cores.

We compare performance of three processors, namely an Intel Xeon Phi KNL, an Intel Xeon Phi KNC, and an Intel Xeon multi-core CPU. The KNL is a pre-production model with 68 cores. The KNC model contains 61 cores in a bi-directional ring bus structure with 8 GB of GDDR5 memory on the chip. We focus on native mode on the KNC for clearest comparison, which allows access to the GDDR5 memory on the chip only, but remark on the symmetric and offload modes. As baseline reference, we also tested on a compute node with two 8-core CPUs, giving it 16 cores total. The KNC and the CPU used here were the hardware that was available at the time of the experiments in Summer 2016 on the Stampede cluster at TACC and thus used as reference baseline, even though the equipment is older than the KNL obviously. Section 2 specifies the hardware in detail.

As test problem, we use the Poisson equation with homogeneous Dirichlet boundary conditions

$$\begin{aligned} -\Delta u &= f \quad \text{for } \mathbf{x} \in \Omega, \\ u &= 0 \quad \text{for } \mathbf{x} \in \partial\Omega, \end{aligned} \tag{1.1}$$

on the domain $\Omega = (0,1) \times (0,1) \subset \mathbb{R}^2$. Here, $\partial\Omega$ denotes the boundary of the domain Ω, and the Laplace operator in (1.1) is defined as $\Delta u = \frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2}$ for $\mathbf{x} = (x_1, x_2)^T \in \Omega$. The equation is discretized by the finite difference method on a $N \times N$ mesh and the resulting system of N^2 linear equations solved by the conjugate gradient method. The numerical method is parallelized in C using hybrid MPI+OpenMP code. Section 3 provides more implementation details.

The performance studies in Section 4 use essentially all available computational cores in each hardware, since that provides best total performance; for the KNC, this is discussed in [13], while for the KNL, it is discussed in detail in [4]. With 4 threads per core supported on both Phis, we use on the KNL with 68 cores a total of 272 threads; our tests indicate that there is no downside to this choice for this code for KNL. On the KNC with 61 cores, we report here results using a total of 240 threads (with 1 core reserved for the operating system in these tests), although also here tests indicated no distinct advantage to this choice for this code [13]. On the compute node with two eight-core CPUs, we use a total of 16 threads. We test combinations of MPI processes and OpenMP threads.

Section 5 summarizes our conclusions and suggests opportunities for more studies that would explore additional features of the KNL.

At the time of this writing, in Summer 2016, the KNL has just begun to be available to the general public, and only few performance comparisons are available. One paper is [17] that approaches the question by running established benchmark codes (Mantevo suite [6], NAS Parallel Benchmarks [1, 15, 22]) and sophisticated established research codes (WRF [18], LBS3D [16]) as test cases on the KNL. Another performance result of a formal benchmark is contained in the brief announcement [11] by Intel of results of the HPCG Benchmark that characterizes speedup of KNL over two 18-core CPUs (Intel E5-2697v4) as approximately 2x. The HPCG Benchmark [7] is essentially a more sophisticated implementation of a discretization of the 3-D version of (1.1) and designed to provide a modern reference for performance [3].

We complement these results here by using another well-established test problem, that is customary in contexts from numerical methods for PDEs to numerical linear algebra, but that is implemented in short hand-written code. Since the test problem is familiar to many researchers, we aim to demonstrate how researchers can readily implement the same or a version adjusted to their needs (e.g., different PDE, different discretization, and/or different solver) and thus can

achieve a benchmarking code for their research. We also want to show by remarks on compiling and running that one key benefit of the Intel Xeon Phi family of processors is that it only takes some compilation flags to port code to it, and excellent performance improvements are already possible. Demonstrating this for a test problem is the main point of this work; it shows how easy it is for a user to repeat the comparison for his/her own research code. The precise comparison of performance is only secondary and is necessarily limited by whatever hardware is available at the time, in this case Summer 2016. At the time of revision in 2017, one would expect state-of-the-art CPUs with, e.g., 16 cores per CPU to perform better.

2 Hardware

2.1 CPU Hardware

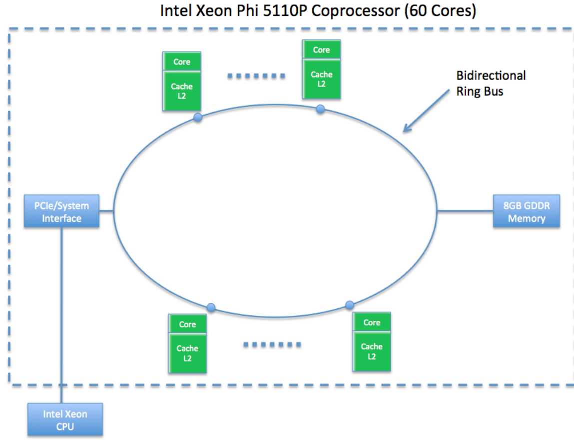
The CPU baseline results were obtained on Stampede at the Texas Advanced Computing Center (TACC). One compute node contains two eight-core 2.7 GHz Intel E5-2680 Sandy Bridge CPUs and 32 GB of DDR3 memory.

2.2 Intel Xeon Phi Knights Corner (KNC)

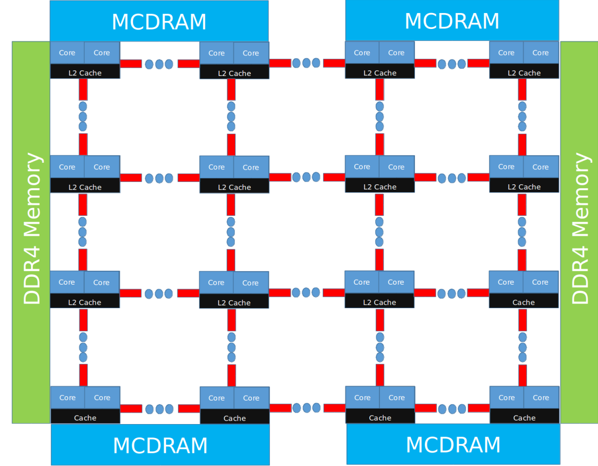
The first-generation Intel Xeon Phi is a many-core co-processor designed to compete with GPUs such as the Tesla cards by NVIDIA. The Phi was announced in June of 2011, and the first units were delivered Q4 2012. The cores on a Phi are more capable than the cores in a GPU, since each core of a Phi is x86 compliant and can run its independent instruction stream. GPU cores, such as a CUDA core found in a NVIDIA Tesla card, are quite low powered but numerous at 2496 cores in a NVIDIA K20, for example. By contrast, the Intel Xeon Phi, code-named Knights Corner (KNC), has cores more similar to that of a multi-core CPU, each of which is a full-fledged x86 compliant core. Production model KNCs are also known as Xeon Phi x100 series. Figure 2.1 (a) shows the schematic of a KNC with 60 cores; KNC processors can have up to 61 cores connected in a bi-directional ring bus [8]. Each core has 4 threads for a total of up to 244 threads. The processors in the KNC are based off the 22 nm Ivy Bridge architecture. The cores in KNC each have one 512-bit Vector Processing Unit (VPU). This VPU can do 16 single or 8 double floating point operations per cycle. Also connected to this bus is the 8 GB of GDDR5 memory, the same kind as used in GPUs. This bus also has a PCIe Gen 3 x16 slot that connects the KNC to the rest of the system. Most importantly the PCIe bus connects the KNC to the DDR3 memory of the node. The KNC only has a Linux micro-OS, this means it can only be run as a co-processor, and its use in a node requires a CPU as host, whether the node participates in the calculations or not. We use the KNC on Stampede in TACC that has 61 cores.

2.3 Intel Xeon Phi Knights Landing (KNL)

The new second-generation Intel Xeon Phi is code-named Knights Landing (KNL). The production models of KNL models are known as Xeon Phi x200 series. This new generation can have up to 72 cores. For our testing we had access to Grover, a KNL testing server, at the University of Oregon. Grover is a single node server with a Intel Xeon Phi 7250 pre-production model running as a standalone processor. The 7250 in Grover has 68 cores [10]. Figure 2.1 (b) shows the internal schematic of the KNL processor. These processors are based on a 14 nm Airmont process technology, similar to CPUs found in low power tablet computers. The processors are also now connected on



(a) KNC schematic



(b) KNL schematic

Figure 2.1: Schematic of the Intel Xeon Phi (a) Knights Corner (KNC) and (b) Knights Landing (KNL). Image sources (a) webpage of the UMBC High Performance Computing Facility (hpcf.umbc.edu) and (b) Samuel Khuvis.

a more complicated 2D mesh network instead of a bi-direction ring. This mesh is similar to the topology of a GPU and allows multitudes more connections between cores. The mesh arranges the 68 cores into 34 tiles with 2 cores each. The 2 cores on each of these tiles share an L2 cache. The `cpu_id` values of the cores are out of order, meaning that consecutive `cpu_id` values are not on the same tile. For example, `cpu_id`= 0 and = 1 are not on the first tile, but on the first and second tiles; the second core on the first tile is eventually reached by `cpu_id`= 34. This counting scheme is intended to desynchronize operations such as memory access, for better performance. Each core has 4 threads allowing a total of up to 272 threads in total.

The KNL has two different types of memory to access, DDR4 and MCDRAM. The DDR4 is the main memory of the cluster node. MCDRAM (Multi-Channel DRAM) is a new form of HMC (Hybrid Memory Cube) also known as stacked memory. MCDRAM allows speeds of up to 500 GB/s to 16 GB of RAM across 4 channels. This is 5 times faster bandwidth than DDR4 and nearly 50% more bandwidth compared to GDDR5 used in KNC and GPUs like the K20. The MCDRAM on Grover is configured in flat mode. This means that MCDRAM is used as addressable memory. MCDRAM may also be used as L3 cache in cache mode or as a combination of addressable memory and cache in hybrid mode. Another major improvement is the doubling of Vector Processing Units (VPUs), from one per KNC core to two per KNL core. Each VPU is 512 bits wide, allowing for 16 single or 8 double precision operations per clock cycle; thus, this allows 16 double precision additions to happen at the same time per core [19]. The KNL also runs a full Linux-based OS. This allows it to be run as either a processor or co-processor. As a co-processor the KNL functions similar to a KNC or GPU over a PCIe Gen 3 x16 connection. As a full processor, the KNL replaces a CPU in a node and runs through the LGA 3647 socket connection. Our studies use the Grover KNL as a standalone processor.

3 Test Problem

We consider the classical elliptic test problem of the Poisson equation with homogeneous Dirichlet boundary conditions in (1.1), as used in many Numerical Linear Algebra textbooks as standard example, e.g., [2, Section 6.3], [5, Subsection 9.1.1], [12, Chapter 12], and [21, Section 8.1], and researchers should be easily able to recreate the results. Our studies use $f(x_1, x_2)$ in (1.1) such that the analytical solution $u(x_1, x_2) = \sin^2(\pi x_1)\,\sin^2(\pi x_2)$ is known. Using $N + 2$ mesh points in each dimension, we construct a mesh with uniform mesh spacing $h = 1/(N+1)$. Specifically, define the mesh points $(x_{k_1}, x_{k_2}) \in \overline{\Omega} \subset \mathbb{R}^2$ with $x_{k_i} = h\,k_i$, $k_i = 0, 1, \ldots, N, N+1$, in each dimension $i = 1, 2$. Denote the approximations to the solution at the mesh points by $u_{k_1,k_2} \approx u(x_{k_1}, x_{k_2})$. Then approximate the second-order derivatives in the Laplace operator at the N^2 interior mesh points by centered difference approximations.

$$\frac{\partial^2 u(x_{k_1}, x_{k_2})}{\partial x_1^2} + \frac{\partial^2 u(x_{k_1}, x_{k_2})}{\partial x_2^2} \approx \frac{u_{k_1-1,k_2} - 2u_{k_1,k_2} + u_{k_1+1,k_2}}{h^2} + \frac{u_{k_1,k_2-1} - 2u_{k_1,k_2} + u_{k_1,k_2+1}}{h^2} \quad (3.1)$$

for $k_i = 1, \ldots, N$, $i = 1, 2$, for the approximations at the interior points. Using these approximations together with the homogeneous boundary conditions (1.1) as determining conditions for the approximations u_{k_1,k_2} gives a system of N^2 linear equations

$$-u_{k_1,k_2-1} - u_{k_1-1,k_2} + 4u_{k_1,k_2} - u_{k_1+1,k_2} - u_{k_1,k_2+1} = h^2\, f(x_{k_1}, x_{k_2}), \quad k_i = 1, \ldots, N, \quad i = 1, 2, \quad (3.2)$$

for the finite difference approximations u_{k_1,k_2} at the N^2 interior mesh points $k_i = 1, \ldots, N$, $i = 1, 2$.

For simplicity of coding, for reproducibility by others, and to provide a challenging benchmark for the hardware to be tested, we collect the N^2 unknown approximations u_{k_1,k_2} in a vector $u \in \mathbb{R}^{N^2}$ using the ordering of the mesh points with $k = k_1 + N(k_2 - 1)$ for $k_i = 1, \ldots, N$, $i = 1, 2$. In this ordering, neighboring x_1-coordinates are stored consecutively in memory. In terms of the two-dimensional representation u_{k_1,k_2}, this results in column-oriented storage; this is the storage scheme used in Matlab as well as BLAS/LAPACK, for instance, and thus natural to use to interface with such software later. We can state the problem as a system of linear equations in standard form $A\,u = b$ with a system matrix $A \in \mathbb{R}^{N^2 \times N^2}$ and a right-hand side vector $b \in \mathbb{R}^{N^2}$. The components of the right-hand side vector b are given by the product of h^2 multiplied by right-hand side function evaluations $f(x_{k_1}, x_{k_2})$ at the interior mesh points using the same ordering as the one used for u_{k_1,k_2}. The system matrix $A \in \mathbb{R}^{N^2 \times N^2}$ can be defined recursively as block tri-diagonal matrix with $N \times N$ blocks of size $N \times N$ each. Concretely, we have $A = \text{block-tridiag}(T, S, T) \in \mathbb{R}^{N^2 \times N^2}$ with the tri-diagonal matrix $S = \text{tridiag}(-1, 4, -1) \in \mathbb{R}^{N \times N}$ for the diagonal blocks of A and with $T = -I \in \mathbb{R}^{N \times N}$ denoting a negative identity matrix for the off-diagonal blocks of A; see, for instance, [13] for complete detail.

For fine meshes with large N, the conjugate gradient (CG) method is appropriate for solving this linear system and guaranteed to converge in exact arithmetic, i.e., by theory of numerical linear algebra, since the system matrix A is known to be symmetric positive definite [5]. We use a tolerance of 10^{-6} on the relative residual, the zero vector as initial guess, and run all reported cases to convergence. The convergence behavior of the CG method for this test problem has been tested for instance in [13,14]. The studies demonstrate that the number of iterations approximately doubles whenever the mesh size N doubles [14, Table 3.1]. We did not repeat these studies, but use their knowledge to anticipate the wall clock timings in the results tables in the next section.

In a careful implementation, the conjugate gradient method requires in each iteration exactly two inner products between vectors, three vector updates, and one matrix-vector product involving

the system matrix A. In fact, this matrix-vector product is the only way in which A enters into the algorithm. Therefore, a so-called matrix-free implementation of the conjugate gradient method is possible that avoids setting up any matrix, if one provides a function that computes as its output the product vector $v = A\,u$ component-wise directly from the components of the input vector u by using the explicit knowledge of the values and positions of the non-zero components of A, but without assembling A as a matrix.

Thus, without storing A, a careful, efficient, matrix-free implementation of the (unpreconditioned) conjugate gradient method only requires the storage of four vectors The memory required to run the code for $N = 8{,}192$ is estimated (and observed in [13]) at 2 GB. This fits comfortably in the memory of all hardware considered, however, doubling the resolution would not be possible within the 8 GB memory of the KNC any more. In a parallel implementation of the conjugate gradient method, each vector is split into as many blocks as parallel processes are available and one block distributed to each process. The data are split in the last, that is, the x_2-dimension. In this way, data in each block is consecutive in memory. Later, it is necessary to communicate one line of points between neighboring processes. The consecutive data storage then also applies to this line being communicated.

Each parallel process possesses its own block of each vector, and no vector is ever assembled in full on any process. The vector updates in each iteration can be executed simultaneously on all processes on their local blocks, because they do not require any parallel communications. The MPI function `MPI_Allreduce` is necessary to compute the inner product and vector norm required in the algorithm. The matrix-vector product $v = A\,u$ also computes only the block of the vector v that is local to each process. But since the matrix A has non-zero off-diagonal elements, each local block needs values of u that are local to the two processes that hold the neighboring blocks of u. More precisely, one line of points needs to be interchanged between neighboring processes, and the points in this line are consecutive in memory, which improves performance. The commands `MPI_Isend` and `MPI_Irecv` are used as non-blocking communication commands between neighboring processes and to interleave calculations and communications in the matrix-vector product. In the hybrid MPI+OpenMP implementation, all expensive `for` loops with large trip counts are parallelized with OpenMP. This includes the loops in the computation of the matrix vector product, axpby operations, and computation of dot products. These `for` loops are parallelized with the OpenMP pragma `#pragma omp parallel for` just before the outermost loop, instructing each MPI process to divide the iterations of the loop between the OpenMP threads and execute them in parallel. This strategy ensures that each thread takes advantage of the consecutive memory allocation of each thread's portion of the data.

4 Results

This section reports the performance studies for the solution of the test problem using hybrid CPU+OpenMP code on one CPU node in Section 4.1, on one hybrid CPU/Phi node with Intel Xeon Phi KNC in Section 4.2, and on one Intel Xeon Phi KNL in Section 4.3. Our studies that are presented here use essentially all available computational cores in each hardware, since that has been demonstrated to provide best total performance; see [13] for more detailed studies for the KNC. Compiler versions used are the most recent available versions on each platform at the the time of the runs.

We make one comment already here that applies to all tables and to set the expectations for the timing results in the following tables. The computational cost of the CG method is proportional to

the number of arithmetic operations the algorithm needs to reach convergence. On the one hand, the number of operations *per iteration* increases by a factor 4 whenever the mesh size N doubles, since the number of elements in the unknown vector increases by this factor. On the other hand, the convergence tests indicates that the number of iterations increases by a factor 2 approximately whenever N doubles. Thus, from one row to the next in tables of observed run times, you expect an increase of about a factor 8 for the unpreconditioned CG method used as test case here.

4.1 CPU Performance Studies on Stampede

The hybrid MPI+OpenMP code in C was compiled for CPUs on Stampede using the Intel compiler version 15.0 with flags `-c99 -Wall -O3 -openmp` (with OpenMP version 4.0) and Intel MPI version 5.0.3. We use default settings in the run script and use the standard run command `ibrun`.

Table 4.1 reports the baseline performance results using only the modern CPUs in one dual-socket node, for comparison with the Intel Xeon Phi studies. In the studies using 1 CPU node, we tested hybrid MPI+OpenMP code such that we utilized all cores on the node, so 2 CPUs with 8 cores per CPU for a total 16 cores, thus running 1 threads per core. These studies used the DDR3 RAM available on Stampede. Table 4.1 demonstrates that different choices of MPI processes and OpenMP threads per process can potentially make a difference in performance [13]. Specifically, it may be disadvantageous to only use OpenMP, reported in the first column of results in Table 4.1, since threads on one CPU may have to access data reachable only from the other CPU. This result confirms that using only shared-memory multi-threading may not be optimal, but that a combination of MPI and OpenMP may be needed.

Table 4.1: Observed wall clock times in units of MM:SS on one CPU node with two 8-core CPUs on Stampede using 16 threads and DDR3 memory.

CPU — Stampede — DDR3					
MPI proc	1	2	4	8	16
Threads/proc	16	8	4	2	1
1024×1024	00:02	00:01	00:01	00:01	00:01
2048×2048	00:35	00:20	00:20	00:20	00:20
4096×4096	05:11	02:42	02:42	02:42	02:42
8192×8192	36:18	21:57	21:58	22:02	21:48

4.2 KNC Performance Studies on Stampede

The hybrid MPI+OpenMP code in C was compiled for the KNC on Stampede using the Intel compiler version 15.0 with flags `-c99 -Wall -O3 -openmp -mmic` (with OpenMP version 4.0) and Intel MPI version 5.0.3. We use default settings in the run script except we use the normal-mic partition, `MIC_PPN` for the number of MPI processes, and `MIC_OMP_NUM_THREADS` for the number of OpenMP threads. The run command is `ibrun.symm -m` with the mic executable.

Table 4.2 collects the performance results using 1 Intel Phi KNC in native mode on Stampede that has 61 cores. In these studies, we tested the hybrid MPI+OpenMP code in native mode on 60 of the 61 KNC cores, leaving one core for the operating system. MPI processes times OpenMP threads are always equal 240, using all 60 cores with 4 threads per core. In native mode, we are restricted to the 8 GB of GDDR5 memory on board the KNC. We can see from Table 4.2 that for most combinations of MPI processes and threads, the KNC run times using GDDR5

are comparable to the CPU run times using DDR3 on Stampede. While the differences are not dramatic, it is noticeable that neither a pure MPI job (1 thread per process) nor a pure OpenMP job (1 MPI process) are optimal, but rather, a combination of them, with the optimal using around 4 or 8 threads per MPI process. This indicates the importance of incorporating both MPI and OpenMP in the code.

Table 4.2: Observed wall clock times in units of MM:SS on 1 KNC on Stampede in native mode using 240 threads and GDDR5 memory.

KNC — Stampede — GDDR5										
MPI proc	1	2	4	8	15	16	30	60	120	240
Threads/proc	240	120	60	30	16	15	8	4	2	1
1024×1024	00:03	00:02	00:02	00:02	00:02	00:02	00:02	00:02	00:04	00:13
2048×2048	00:17	00:16	00:16	00:15	00:15	00:15	00:15	00:15	00:18	00:26
4096×4096	01:53	01:48	01:46	01:45	01:57	01:45	01:51	01:48	01:52	02:33
8192×8192	28:24	28:20	27:51	23:08	23:06	23:00	22:24	22:45	22:43	25:37

The KNC is only a co-processor and needs to be used in a hybrid node that also contains a CPU as host. A possible and typical arrangement is that of having two CPUs and two KNCs in one node. In this arrangement, besides using one KNC in native mode, it can also be used in symmetric or in offload mode. With the KNC in symmetric mode, one can combine the resources of the two CPUs and of two KNCs. In Table 4.2 we focus on results in native mode for the KNC, since all runs on a standalone KNL are analogously restricted to its own cores. For the mesh resolution N = 8,192 problem, on a hybrid node using one 8-core CPU with one Phi KNC in symmetric mode run time was 17:55, which is both faster than the KNC by itself in Table 4.2 or two CPUs in Table 4.1. But the best performance achieved using two 8-core CPUs and two Phi KNCs is 09:51 [13, Table 2.7.1]. This performance in symmetric mode was achieved by running with 16 MPI processes and 15 OpenMP threads per process on each Phi and 16 MPI processes and 1 OpenMP thread per process on each CPU. Using all available resources on a hybrid node thus results in a 2.27x improvement in runtime over native mode on 1 KNC.

Remark: We mention that offload mode is a third mode of using the KNC, in which the two CPUs of a node only facilitate the MPI communication between the Phis, but all calculations take place on the Phis. The N = 8,192 case took 27:58 in offload mode, using both KNCs in the node [13, Table 2.7.1]. The poor performance of offload mode is due to the restriction of multithreading only, not MPI parallelism, in offload regions on the Phi. This explains why this performance in offload mode is consistent with the result for 1 MPI process and 240 threads per process in Table 4.2.

4.3 KNL Performance Studies on Grover

The hybrid MPI+OpenMP code was compiled on Grover using the Intel compiler version 16 with flag `-xMIC-AVX512` specifically for the KNL and other flags `-O3 -std=c99 -Wall -mkl -qopenmp` (with OpenMP version 4.0) and Intel MPI version 5.1. The code is run on Grover using the `numactl` command. Its flag `--membind=1` forces the code to use the MCDRAM memory, while the flag `--membind=0` has it use the DDR4 memory.

These are the performance studies for the KNL on Grover, using both the DDR4 RAM on the node and the MCDRAM on board the KNL. In the studies in Tables 4.3 and 4.4, we tested hybrid MPI+OpenMP code such that we utilized all 68 cores of the KNL. MPI processes times OpenMP threads equal 272, using 68 cores with 4 threads per core.

4.3.1 KNL using MCDRAM

In the studies in Table 4.3, we exclusively used the MCDRAM available on board of the KNL. Comparing corresponding mesh resolutions up to $N = 8{,}192$, the largest mesh resolution reported in the previous tables, Table 4.3 shows the KNL using MCDRAM is significantly faster than two CPUs in Table 4.1 and the KNC using GDDR5 RAM in native mode in Table 4.2. For instance with 8 threads per process for $N = 8{,}192$, Table 4.3 shows 05:12, while Table 4.1 shows 21:57, and Table 4.2 shows 22:24, showing a speedup of about 4x. The KNL using MCDRAM is also approximately 2 times faster than the best KNC performance of 09:51 in symmetric mode using two 8-core CPUs and two Phi KNCs [13, Table 2.7.1]. This is true across all combinations of MPI processes and threads on the KNL using MCDRAM in Table 4.3. This better performance allows us to add all results of the $N = 16{,}384$ mesh resolution in an additional row of Table 4.3 to the study, since it does not require excessive run times any more (note contrast to DDR4 memory in Table 4.4).

Table 4.3 also shows the results with two choices for thread pinning using the OpenMP option `KMP_AFFINITY`. Table 4.3 (a) shows results with `KMP_AFFINITY=scatter`, while Table 4.3 (b) with `KMP_AFFINITY=compact`. `Compact`, as the name implies, tries to keep the threads closer together. `Scatter` tries instead to distribute the threads to cores as evenly as possible. The cases of `KMP_AFFINITY=compact` and `KMP_AFFINITY=scatter` show no observable difference in run time behavior.

We observe that the run time is similar for all combinations of MPI processes and OpenMP threads. This is aided by the very systematic structure of the algorithm in the code, where MPI communication commands often appear in near-lockstep; in turn, this brings out the richness of connections between pairs of cores of the KNL enabled by the 2D mesh structure, which makes this good performance possible. Though the differences were not dramatic, we note that neither a pure MPI job (1 thread per process) nor a pure OpenMP job (1 MPI process) are optimal.

Table 4.3: Observed wall clock times in units of MM:SS on 1 KNL on Grover using all 272 threads and MCDRAM memory, with two settings of `KMP_AFFINITY`.

(a) KNL — MCDRAM — Scatter

MPI proc	1	2	4	8	16	17	34	68	136	272
Threads/proc	272	136	68	34	17	16	8	4	2	1
1024×1024	00:01	00:01	00:01	00:01	00:01	00:01	00:01	00:01	00:01	00:02
2048×2048	00:06	00:06	00:06	00:06	00:07	00:06	00:06	00:07	00:07	00:08
4096×4096	00:41	00:41	00:41	00:40	00:41	00:41	00:40	00:40	00:42	00:46
8192×8192	05:15	05:16	05:15	05:14	05:12	05:11	05:12	05:14	05:17	05:30
16384×16384	41:33	41:15	41:37	41:50	40:58	40:59	40:51	41:09	41:01	41:43

(b) KNL — MCDRAM — Compact

MPI proc	1	2	4	8	16	17	34	68	136	272
Threads/proc	272	136	68	34	17	16	8	4	2	1
1024×1024	00:01	00:01	00:01	00:01	00:01	00:01	00:01	00:01	00:01	00:02
2048×2048	00:06	00:06	00:06	00:06	00:06	00:06	00:06	00:08	00:07	00:08
4096×4096	00:41	00:41	00:41	00:41	00:41	00:41	00:41	00:40	00:42	00:46
8192×8192	05:15	05:14	05:13	05:13	05:12	05:11	05:13	05:13	05:18	05:30
16384×16384	43:46	41:14	41:09	41:11	40:55	40:59	40:55	40:52	41:32	41:38

4.3.2 KNL using DDR4 RAM

In the studies in Table 4.4, we exclusively used the DDR4 RAM of the KNL. Immediately, we observe that the KNL using DDR4 is significantly slower than the KNL using MCDRAM in Table 4.3. In the $N = 8{,}192$ case, MCDRAM is almost 5 times faster than DDR4 across the board. Since Grover is a shared resource, without a scheduler, we run only selected cases of significant run times. Based on the results of the MCDRAM 16,384 × 16,384 case, and the fact that the KNL accommodates 4 threads per core, we choose to run only the 1 MPI process case, thus using multithreading only, and the case with one MPI process per core, thus 68 MPI processes. In these cases, we observe run times over 3 hours that again demonstrate the almost 5x speedup of MCDRAM in Table 4.3 compared to DDR4 in Table 4.4. Omitted runs are indicated by (*) in Table 4.4. Though the DDR4 memory on the KNL is much larger than the 16 GB MCDRAM, we do not run larger problem sizes using the DDR4 due to the excessive run times required.

For the mesh resolutions $N = 8{,}192$, Table 4.4 shows that the KNL using DDR4 performance is comparable to the KNC using GDDR5 RAM in native mode in Table 4.2 and two CPUs in Table 4.1. But the best performance of the KNC in native mode using GDDR5 RAM and the best performance of the two CPUs both outperform the best KNL using DDR4. The KNL using DDR4 is also slower than 1 KNC in symmetric mode with 1 CPU [13, Table 2.7.1].

Table 4.4 also shows the results with two choices for thread pinning using the OpenMP option `KMP_AFFINITY`. Table 4.4 (a) shows results with `KMP_AFFINITY=scatter`, while Table 4.4 (b) with `KMP_AFFINITY=compact`. As in Table 4.3 using MCDRAM there is no difference in run times between `KMP_AFFINITY=scatter` and `KMP_AFFINITY=compact`.

We also observe that the run time is similar, but not identical, for different combinations of process and threads across each row of Table 4.4. As in previous cases, neither a pure MPI job (1 thread per process) nor a pure OpenMP job (1 MPI process) are optimal.

Table 4.4: Observed wall clock times in units of MM:SS on 1 KNL on Grover using 272 threads and DDR4 memory, with two settings of `KMP_AFFINITY`. (*) indicated runs omitted due to long run times.

(a) KNL — DDR4 RAM — Scatter										
MPI proc	1	2	4	8	16	17	34	68	136	272
Threads/proc	272	136	68	34	17	16	8	4	2	1
1024 × 1024	00:02	00:02	00:02	00:02	00:02	00:02	00:02	00:02	00:03	00:04
2048 × 2048	00:22	00:21	00:21	00:21	00:21	00:21	00:22	00:23	00:23	00:25
4096 × 4096	02:58	02:57	02:57	02:56	02:56	02:53	02:56	02:58	03:02	03:12
8192 × 8192	24:09	24:08	24:00	24:02	24:02	23:58	24:02	24:09	24:27	25:09
16384 × 16384	194:37	(*)	(*)	(*)	(*)	(*)	(*)	193:43	(*)	(*)

(b) KNL — DDR4 RAM — Compact										
MPI proc	1	2	4	8	16	17	34	68	136	272
Threads/proc	272	136	68	34	17	16	8	4	2	1
1024 × 1024	00:02	00:02	00:02	00:02	00:02	00:01	00:02	00:02	00:02	00:04
2048 × 2048	00:22	00:21	00:22	00:21	00:22	00:21	00:21	00:22	00:24	00:26
4096 × 4096	02:57	02:57	02:56	02:56	02:57	02:55	02:55	02:58	03:03	03:11
8192 × 8192	24:09	24:07	24:00	24:00	23:59	23:58	24:00	24:08	24:29	25:11
16384 × 16384	193:52	(*)	(*)	(*)	(*)	(*)	(*)	193:58	(*)	(*)

5 Conclusions and Outlook

The results in Section 4 provide a comparison of performance using essentially all available cores on CPUs, KNC, and KNL for the memory-bound test problem of the two-dimensional Poisson equation. We found that using MCDRAM on KNL is drastically faster than using CPUs, KNC (in all three modes of native, symmetric, or offload), or KNL with DDR4 in all cases for this test problem. Despite DDR4 being a slower form of memory than GDDR5, we found that KNL using DDR4 is comparable in most cases to KNC in native mode. Though the differences were not dramatic, using both MPI and OpenMP parallelism is vital to tune code for optimal performance, as opposed to using only MPI or only OpenMP. Table 5.1 summarizes the key comparisons of the results of observed wall clock time (in units of MM:SS = minutes:seconds) for the case of the $N \times N = 8{,}192 \times 8{,}192$ mesh, which is the largest case that fits in memory on the KNC in native mode and thus is the largest case that we can compare on all platforms. We observe that all rows of the sub-tables show that using different combinations of MPI processes and OpenMP threads can affect performance, but even for sub-optimal cases, we find that the KNL using the MCDRAM in Table 5.1 (c) is dramatically faster than the KNL using DDR4 memory, confirming the nominal 5x faster speed of MCDRAM memory. Despite DDR4 being a slower form of memory, the KNL using DDR4 in Table 5.1 (c) is comparable in most cases to KNC in native mode using its GDDR5 in Table 5.1 (b). The choice of `KMP_AFFINITY` had no effect on performance for this code on the KNL, which indicates the power of the 2D mesh structure in the KNL. In final analysis, with the best combination of MPI processes and OpenMP threads per process, the KNL can be approximately 4x faster than either KNC in native mode in Table 5.1 (b) or two CPUs in Table 5.1 (a). With KNC in symmetric mode, one can combine the resources of the two CPUs and of two KNCs and this results in much better performance than only CPUs in Table 5.1 (a) or KNC in native mode in Table 5.1 (b) or the KNL using DDR4 in Table 5.1 (c), but is still a factor of 2 slower than the KNL

Table 5.1: Comparison of observed wall clock times in units of MM:SS for $N \times N = 8{,}192 \times 8{,}192$ mesh on CPU node with two 8-core CPUs, KNC in native and symmetric modes, and KNL with 68 cores, accessing their respective memory, and a choice of `KMP_AFFINITY=scatter` or `KMP_AFFINITY=compact`.

(a) CPU node with DDR3 with total of 16 threads					
MPI proc	1	2	4	8	16
Threads/proc	16	8	4	2	1
DDR3	36:18	21:57	21:58	22:02	21:48

(b) KNC in native mode with GDDR5 and symmetric mode with total of 240 threads										
MPI proc	1	2	4	8	15	16	30	60	120	240
Threads/proc	240	120	60	30	16	15	8	4	2	1
GDDR5	28:24	28:20	27:51	23:08	23:06	23:00	22:24	22:45	22:43	25:37
KNC in symmetric mode: 1 CPU, 1 KNC — 17:55, 2 CPUs, 2 KNCs — 09:51										

(c) KNL with DDR4 and MCDRAM with total of 272 threads										
MPI proc	1	2	4	8	16	17	34	68	136	272
Threads/proc	272	136	68	34	17	16	8	4	2	1
DDR4–scatter	24:09	24:08	24:00	24:02	24:02	23:58	24:02	24:09	24:27	25:09
DDR4–compact	24:09	24:09	24:00	24:00	23:59	23:58	24:00	24:08	24:29	25:11
MCDRAM–scatter	05:15	05:16	05:15	05:14	05:12	05:11	05:12	05:14	05:17	05:30
MCDRAM–compact	05:15	05:14	05:13	05:13	05:12	05:11	05:13	05:13	05:18	05:30

using MCDRAM. This demonstrates the excellent performance improvements that are achievable for hybrid MPI+OpenMP code even without particular optimization for the new hardware. We believe that this is worth noting and may prove to be an important selling point of the KNL to many researchers, provided their problem fits into the KNL's 16 GB of on-chip MCDRAM memory, which is a significant amount of memory and sufficient for many applications.

In the process of running the simulations included here we also made some preliminary observations running the command `top` while simulations were running on the KNL. The resulting runs, in which `top` was run during the simulation, were approximately 20% slower than runs in which no other actions were executed on the KNL. Further study should be completed to asses the effect of leaving some cores available to the operating system and for tasks like running `top`.

These results give only an introduction to how to run code on the KNL and aim to demonstrate what performance is readily possible on it. The KNL has additional features that may be utilized to further improve performance, some of which are already tested in [17]. As mentioned in Section 2.3, each core of the KNL has 2 VPUs. As a result, proper vectorization of code is essential to good performance on the KNL. Even on the KNC which only had 1 VPU per core, [13, Table 1.6.2] shows a halving of runtime for a three-dimensional Poisson equation code by manually unrolling loops to improve vectorization. Also as mentioned in Section 2.3, the KNL may also be configured to use MCDRAM in cache and hybrid modes that increase the size of the cache but may increase latency of cache misses. The KNL may also be configured in different cache clustering modes. Grover was configured in all-to-all mode, meaning that memory addresses are uniformly distributed across the KNL. However, other modes that divide the KNL into hemispheres or quadrants are available that may improve cache locality and thus decrease latency of L2 cache misses. We have provided initial guidance on how to run memory-bound codes on the KNL, but these additional features provide further opportunities to tune codes for the KNL.

Acknowledgments

These results were obtained as part of the REU Site: Interdisciplinary Program in High Performance Computing (hpcreu.umbc.edu) in the Department of Mathematics and Statistics at the University of Maryland, Baltimore County (UMBC) in Summer 2016. This program is funded by the National Science Foundation (NSF), the National Security Agency (NSA), and the Department of Defense (DOD), with additional support from UMBC, the Department of Mathematics and Statistics, the Center for Interdisciplinary Research and Consulting (CIRC), and the UMBC High Performance Computing Facility (HPCF). HPCF is supported by the U.S. National Science Foundation through the MRI program (grant nos. CNS–0821258 and CNS–1228778) and the SCREMS program (grant no. DMS–0821311), with additional substantial support from UMBC. Co-author Ishmail Jabbie was supported, in part, by the UMBC National Security Agency (NSA) Scholars Program through a contract with the NSA. The authors thank both our team's graduate assistant Jonathan Graf and faculty mentor Dr. Matthias K. Gobbert for their support throughout the program and beyond. Graduate assistant Jonathan Graf was supported by UMBC. The authors would like to thank the project client and collaborator Samuel Khuvis of ParaTools, Inc. for his support and connection with the Performance Research Laboratory, University of Oregon, that provided access to the KNL Hardware. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI–1053575. We acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper.

References

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, *The NAS parallel benchmarks*, Int. J. Supercomputer Appl., 5 (1991), pp. 63–73.

[2] J. W. Demmel, *Applied Numerical Linear Algebra*, SIAM, 1997.

[3] J. Dongarra and M. A. Heroux, *Toward a new metric for ranking high performance computing systems*, Tech. Rep. SAND2013–4744, Sandia National Laboratories, June 2013. https://software.sandia.gov/hpcg/doc/HPCG-Benchmark.pdf, accessed on March 23, 2017.

[4] J. S. Graf, Parallel Performance of Numerical Simulations for Applied Partial Differential Equation Models on the Intel Xeon Phi Knights Landing Processor. Ph.D. Thesis, Department of Mathematics and Statistics, University of Maryland, Baltimore County, 2017.

[5] A. Greenbaum, *Iterative Methods for Solving Linear Systems*, vol. 17 of Frontiers in Applied Mathematics, SIAM, 1997.

[6] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, *Improving performance via mini-applications*, Tech. Rep. SAND2009–5574, Sandia National Laboratories, 2009.

[7] M. A. Heroux, J. Dongarra, and P. Luszczek, *HPCG technical specification*, Tech. Rep. SAND2013–8752, Sandia National Laboratories, October 2013. https://software.sandia.gov/hpcg/doc/HPCG-Specification.pdf, accessed on March 23, 2017.

[8] *Intel Xeon Phi coprocessor block diagram*. http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-block-diagram.html, 2012. Accessed March 23, 2017.

[9] *Intel re-architects the fundamental building block for high-performance computing — Intel newsroom*. https://newsroom.intel.com/news-releases/intel-re-architects-the-fundamental-building-block-for-high-performance-computing/, June 23 2014. Accessed March 23, 2017.

[10] *Intel Xeon Phi processor 7250 (16 GB, 1.40 GHz, 68 core) specifications*. http://ark.intel.com/products/94035/Intel-Xeon-Phi-Processor-7250-16GB-1_40-GHz-68-core, 2016. Accessed March 23, 2017.

[11] Intel Measured Results, High Performance Conjugate Gradients. www.umbc.edu/~gobbert/papers/KNL_UMBC.PNG, April 2016.

[12] A. Iserles, *A First Course in the Numerical Analysis of Differential Equations*, Cambridge Texts in Applied Mathematics, Cambridge University Press, second ed., 2009.

[13] S. Khuvis, Porting and Tuning Numerical Kernels in Real-World Applications to Many-Core Intel Xeon Phi Accelerators. Ph.D. Thesis, Department of Mathematics and Statistics, University of Maryland, Baltimore County, 2016.

[14] S. Khuvis and M. K. Gobbert, *Parallel performance studies for an elliptic test problem on the cluster maya*, Tech. Rep. HPCF–2015–6, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2015.

[15] NASA Advanced Supercomputing Division, *NAS parallel benchmarks*, 2016. http://www.nas.nasa.gov/publications/npb.html, accessed March 23, 2017.

[16] C. Rosales, *Porting to the Intel Xeon Phi: Opportunities and challenges*, in Extreme Scaling Workshop (XSW 2013), IEEE, 2013, pp. 1–7.

[17] C. Rosales, J. Cazes, K. Milfeld, A. Gómez-Iglesias, L. Koesterke, L. Huang, and J. Vienne, *A comparative study of application performance and scalability on the Intel Knights Landing processor*, in High Performance Computing: ISC High Performance 2016 International Workshops, ExaComm, E-MuCoCoS, HPC-IODC, IXPUG, IWOPH, P^3MA, VHPC, WOPSSS, Frankfurt, Germany, June 19–23, 2016, Revised Selected Papers, M. Taufer, B. Mohr, and J. M. Kunkel, eds., vol. 9945 of Lecture Notes in Computer Science, Springer-Verlag, 2016, pp. 307–318.

[18] W. C. Skamarock, J. B. Klemp, J. Dudhia, D. O. Gill, M. Barker, K. G. Duda, X. Y. Huang, W. Wang, and J. G. Powers, *A description of the advanced research WRF version 3*, tech. rep., National Center for Atmospheric Research, 2008.

[19] A. Sodani, *Intel Xeon Phi processor "Knights Landing" architectural overview*. https://www.nersc.gov/assets/Uploads/KNL-ISC-2015-Workshop-Keynote.pdf, 2015. Accessed March 23, 2017.

[20] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. C. Liu, *Knights Landing: Second-generation Intel Xeon Phi product*, IEEE Micro, 32 (2016), pp. 34–46.

[21] D. S. Watkins, *Fundamentals of Matrix Computations*, Wiley, third ed., 2010.

[22] F. C. Wong, R. P. Martin, R. H. Arpaci-Dusseau, and D. E. Culler, *Architectural requirements and scalability of the NAS parallel benchmarks*, in Proceedings of the 1999 ACM/IEEE Conference on Supercomputing, 1999, pp. 41–41.