View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Centro de Servicios en Gestión de Información

Blocked All-Pairs Shortest Paths Algorithm on Intel Xeon Phi KNL Processor: A Case Study

Enzo Rucci[1], Armando De Giusti[1], and Marcelo Naiouf[2]

[1] III-LIDI, CONICET, Facultad de Informática, Universidad Nacional de La Plata
La Plata (1900), Buenos Aires, Argentina
Email: {erucci,degiusti}@lidi.info.unlp.edu.ar
[2] III-LIDI, Facultad de Informática, Universidad Nacional de La Plata
La Plata (1900), Buenos Aires, Argentina
Email: {mnaiouf}@lidi.info.unlp.edu.ar

Abstract. Manycores are consolidating in HPC community as a way of improving performance while keeping power efficiency. Knights Landing is the recently released second generation of Intel Xeon Phi architecture. While optimizing applications on CPUs, GPUs and first Xeon Phi's has been largely studied in the last years, the new features in Knights Landing processors require the revision of programming and optimization techniques for these devices. In this work, we selected the Floyd-Warshall algorithm as a representative case study of graph and memory-bound applications. Starting from the default serial version, we show how data, thread and compiler level optimizations help the parallel implementation to reach 338 GFLOPS.

Keywords: Xeon Phi, Knights Landing, Floyd-Warshall

1 Introduction

The power consumption problem represents one of the major obstacles for Exascale systems design. As a consequence, the scientific community is searching for different ways to improve power efficiency of High Performance Computing (HPC) systems [8]. One recent trend to increase compute power and, at the same time, limit power consumption of these systems lies in adding accelerators, like NVIDIA/AMD graphic processing units (GPUs), or Intel Many Integrated Core (MIC) co-processors. These manycore devices are capable of achieving better FLOPS/Watt ratios than traditional CPUs. For example, the number of Top500 [2] systems using accelerator technology grew from 54 in June 2013 to 91 in June 2017. In the same period, the number of systems based on accelerators increased from 55 to 90 on the Green500 list [1].

Recently, Intel has presented the second generation of its MIC architecture (branded Xeon Phi), codenamed Knigths Landing (KNL). Among the main differences of KNL regarding its predecessor Knights Corner (KNC), we can find the incorporation of AVX-512 extensions, a remarkable number of vector units increment, a new on-package high-bandwidth memory (HBM) and the ability

to operate as a standalone processor. Even though optimizing applications on CPUs, GPUs and KNC Xeon Phi's has been largely studied in the last years, accelerating applications on KNL processors is still a pending task due to its recent commercialization. In that sense, the new features in KNL processors require the revision of programming and optimization techniques for these devices.

In this work, we selected the Floyd-Warshall (FW) algorithm as a representative case study of graph and memory-bound applications. This algorithm finds the shortest paths between all pairs of vertices in a graph and occurs in domains of communication networking [14], traffic routing [12], bioinformatics [16], among others. FW is both computationally and spatially expensive since it requires $O(n^3)$ operations and $O(n^2)$ memory space, where n is the number of vertices in a graph. Starting from the default serial version, we show how data, thread and compiler level optimizations help the parallel implementation to reach 338 GFLOPS.

The rest of the present paper is organized as follows. Section 2 briefly introduces the Intel Xeon Phi KNL architecture while Section 3 presents the FW algorithm. Section 4 describes our implementation. In Section 5 we analyze performance results while Section 6 discusses related works. Finally, Section 7 outlines conclusions and future lines of work.

2 Intel Xeon Phi Knights Landing

KNL is the second generation of the Intel Xeon Phi family and the first capable of operating as a standalone processor. The KNL architecture is based on a set of *Tiles* (up to 36) interconnected by a 2D mesh. Each Tile includes 2 cores based on the out-of-order Intel's Atom micro-architecture (4 threads per core), 2 Vector Processing Units (VPUs) and a shared L2 cache of 1 MB. These VPUs not only implement the new 512-bit AVX-512 ISA but they are also compatible with prior vector ISA's such as SSEx and AVXx. AVX-512 provides 512-bit SIMD support, 32 logical registers, 8 new mask registers for vector predication, and gather and scatter instructions to support loading and storing sparse data. As each AVX-512 instruction can perform 8 double-precision (DP) operations (8 FLOPS) or 16 single-precision (SP) operations (16 FLOPS), the peak performance is over 1.5 TFLOPS in DP and 3 TFLOPS in SP, more than two times higher than that of the KNC. It is also more energy efficient than its predecessor [17].

Other significant feature of the KNL architecture is the inclusion of an in-package HBM called MCDRAM. This special memory offers 3 operating modes: *Cache*, *Flat* and *Hybrid*. In Cache mode, the MCDRAM is used like an L3 cache, caching data from the DDR4 level of memory. Even though application code remains unchanged, the MCDRAM can suffer lower performance rates. In Flat mode, the MCDRAM has a physical addressable space offering the highest bandwidth and lowest latency. However, software modifications may be required in order to use both the DDR and the MCDRAM in the same application. Finally, in the *Hybrid mode*, HBM is divided in two parts: one part in *Cache mode* and one in *Flat mode* [5].

From a software perspective, KNL supports parallel programming models used traditionally on HPC systems such as OpenMP or MPI. This fact represents a strength of this platform since it simplifies code development and improves portability over other alternatives based on accelerator specific programming languages such as CUDA or OpenCL. However, to achieve high performance, programmers should attend to:

- the efficient exploitation of the memory hierarchy, especially when handling large datasets, and
- how to structure the computations to take advantage of the VPUs.

Automatic vectorization is obviously the easiest programming way to exploit VPUs. However, in most cases the compiler is unable to generate SIMD binary code since it can not detect free data dependences into loops. In that sense, SIMD instructions are supported in KNL processors through the use of guided compilation or hand-tuned codification with intrinsic instructions [17]. On one hand, in guided vectorization, the programmer indicates the compiler (through the insertion of tags) which loops are independent and their memory pattern access. In this way, the compiler is able to generate SIMD binary code preserving the program portability. On the other hand, intrinsic vectorization usually involves rewriting most of the corresponding algorithm. The programmer gains in control at the cost of losing portability. Moreover, this approach also suggests the inhibition of other compiler loop-level optimizations. Nevertheless, it is the only way to exploit parallelism in some applications with no regular access patterns or loop data dependencies which can be hidden by recomputing techniques [6].

3 Floyd-Warshall Algorithm

The FW algorithm uses a dynamic programming approach to compute the all-pairs shortest-paths problem on a directed graph [7, 20]. This algorithm takes as input a $N \times N$ distance matrix D, where $D_{i,j}$ is initialized with the original distance from node i to node j. FW runs for N iterations and at k-th iteration it evaluates all the possible paths between each pair of vertices from i to j through the intermediate vertex k. As a result, FW produces an updated matrix D, where $D_{i,j}$ now contains the shortest distance between nodes i and j. Besides, an additional matrix P is generated when the reconstruction of the shortest path is required. $P_{i,j}$ contains the most recently added intermediate node between i and j. Figure 1 exhibits the naive FW algorithm.

4 Implementation

In this section, we address the optimizations performed on the Intel Xeon Phi KNL processor. First of all, we developed a serial implementation following the naive version described in Figure 1, as this implementation will work as baseline. Next, we optimized the serial version considering data locality and data level parallelism. Finally, we introduced thread level parallelism exploiting the OpenMP programming model to obtain a multi-threaded implementation.

4.1 Data Locality

To improve data locality, the FW algorithm can be blocked [19]. Unfortunately, the three loops can not be interchanged in free manner due to the data dependencies from one iteration to the next in the k-loop (just i and j loops can be done in any order). However, under certain conditions, the k-loop can be put inside the i-loop and j-loop, making blocking possible. The distance matrix D is partitioned into blocks of size $BS \times BS$, so that there are $(N/BS)^2$ blocks. The computations involve $R = N/BS$ rounds and each round is divided into four phases based on the data dependency among the blocks:

1. Update the block *k,k* ($D^{k,k}$) because it is self-dependent.
2. Update the remaining blocks of the k-th row because each of these blocks depends on itself and the previously computed $D^{k,k}$.
3. Update the remaining blocks of the k-th column because each of these blocks depends on itself and the previously computed $D^{k,k}$.
4. Update the rest of the matrix blocks as each of them depends on the k-th block of its row and the k-th block of its column.

In this way, we satisfy all dependencies from this algorithm. Figure 2 shows a schematic representation of a round computation and the data dependences among the blocks while Figure 3 presents the corresponding pseudo-code.

4.2 Data Level Parallelism

The innermost loop of FW_BLOCK code block from Figure 3 is clearly the most computationally expensive part of the algorithm. In that sense, this loop is the best candidate for vectorization. The loop body is composed of an *if* statement that involves one addition, one comparison and (may be) two assign operations. Unfortunately, the compiler detects false dependencies in that loop and is not able to generate SIMD binary code. For that reason, we have explored two SIMD exploitation approaches: (1) guided vectorization through the usage

```
1:  for k = 0..N-1 do
2:      for i = 0..N-1 do
3:          for j = 0..N-1 do
4:              if D_{i,k} + D_{k,j} < D_{i,j} then
5:                  D_{i,j} = D_{i,k} + D_{k,j}
6:                  P_{i,j} = k
7:              end if
8:          end for
9:      end for
10: end for
```

Fig. 1: Naive Floyd-Warshall Algorithm

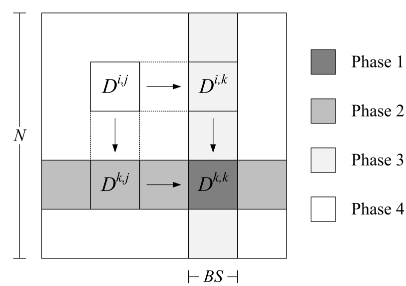

Fig. 2: Schematic representation of the blocked Floyd-Warshall Algorithm

```
1: function FW_BLOCK (D^1, D^2, D^3, P, base)
2:     for k = 0..BS-1 do
3:         for i = 0..BS-1 do
4:             for j = 0..BS-1 do
5:                 if D^2_{i,k} + D^3_{k,j} < D^1_{i,j} then
6:                     D^1_{i,j} = D^2_{i,k} + D^3_{k,j}
7:                     P_{i,j} = base+k
8:                 end if
9:             end for
10:        end for
11:    end for
12: end function
13: for k = 0..R-1 do
14:     b = k × BS
15:     # Phase 1
16:     FW_BLOCK(D^{k,k}, D^{k,k}, D^{k,k}, P^{k,k}, b)
17:     # Phase 2
18:     for j = 0..R-1, j ≠ k do
19:         FW_BLOCK(D^{k,j}, D^{k,k}, D^{k,j}, P^{k,j}, b)
20:     end for
21:     # Phase 3
22:     for i = 0..R-1, i ≠ k do
23:         FW_BLOCK(D^{i,k}, D^{i,k}, D^{k,k}, P^{i,k}, b)
24:     end for
25:     # Phase 4
26:     for i, j = 0..R-1, i,j ≠ k do
27:         FW_BLOCK(D^{i,j}, D^{i,k}, D^{k,j}, P^{i,j}, b)
28:     end for
29: end for
```

Fig. 3: Blocked Floyd-Warshall algorithm.

```
1: for k = 0..BS-1 do
2:     for i = 0..BS-1 do
3:         #pragma unroll
4:         #pragma omp simd
5:         for j = 0..BS-1 do
6:             if (D2[i][k] + D3[k][j] < D1[i][j]) then
7:                 D1[i][j] = D2[i][k] + D3[k][j];
8:                 P[i,j] = base+k;
9:             end if
10:        end for
11:    end for
12: end for
```

Fig. 4: Pseudo-code for FW_BLOCK implementation using guided vectorization

```
1:  for k = 0..BS-1 do
2:      bk = _mm512_set1_epi32(base+k);
3:      for i = 0..BS-1 do
4:          dik = _mm512_set1_ps(D2[i][k]);
5:          #pragma unroll
6:          for j = 0..BS-1 by 16 do
7:              dij = _mm512_load_ps (&D1[i][j]);
8:              dkj = _mm512_load_ps (&D3[k][j]);
9:              sum = _mm512_add_ps (dik, dkj);
10:             mask = _mm512_cmp_ps_mask(sum, dij, CMP_LT_OS);
11:             _mm512_mask_store_ps(&D1[i][j], mask, sum);
12:             _mm512_mask_store_epi32(&P[i][j], mask, bk);
13:         end for
14:     end for
15: end for
```

Fig. 5: Pseudo-code for FW_BLOCK implementation using intrinsic vectorization

of the OpenMP 4.0 *simd* directive and (2) intrinsic vectorization employing the AVX-512 extensions. The guided approach simply consists of inserting the *simd* directive to the innermost loop of FW_BLOCK code block (line 4). On the opposite sense, the intrinsic approach consists of rewriting the entire loop body. Figures 4 and 5 show the pseudo-code for FW_BLOCK implementation using guided and manual vectorization, respectively. In order to accelerate SIMD computation with 512-bit vectors, we have carefully managed the memory allocations so that distance and path matrices are 64-byte aligned. In the guided approach, this also requires adding the *aligned* clause to the *simd* directive.

4.3 Loop Unrolling

Loop unrolling is another optimization technique that helped us to improve the code performance. Fully unrolling the innermost loop of FW_BLOCK code block

was found to work well. Unrolling the i-loop of the same code block once was also found to work well.

4.4 Thread Level Parallelism

To exploit parallelism across multiple cores, we have implemented a multi-threaded version of FW algorithm based on OpenMP programming model. A *parallel* construct is inserted before the loop of line 13 in Figure 3 to create a parallel block. To respect data dependencies among the block computations, the work-sharing constructs must be carefully inserted. At each round, phase 1 must be computed before the rest. So a *single* construct is inserted to enclose line 16. Next, phases 2 and 3 must be computed before phase 4. As these blocks are independent among them, a *for* directive is inserted before the loops of lines 18 and 22. Besides, a *nowait* clause is added to the phase 2 loop to alleviate the thread idling. Finally, another *for* construct is inserted before the loop of line 26 to distribute the remaining blocks among the threads.

5 Experimental Results

5.1 Experimental Design

All tests have been performed on an Intel server running CentOS 7.2 equipped with a Xeon Phi 7250 processor 68-core 1.40GHz (4 hw thread per core and 16GB MCDRAM memory) and 48GB main memory. The processor was run in *Flat* memory mode and *Quadrant* cluster mode.

We have used Intel's ICC compiler (version 17.0.1.132) with the *-O3* optimization level. To generate explicit AVX2 and AVX-512 instructions, we employed the *-xAVX2* and *-xMIC-AVX512* flags, respectively. Also, we used the *numactl* utility to exploit MCDRAM memory (no source code modification is required). Besides, different workloads were tested: $N = \{4096, 8192, 16384, 32768, 65536\}$.

5.2 Performance Results

First, we evaluated the performance improvements of the different optimization techniques applied to the naive serial version, such as blocking (*blocked*), data level parallelism (*simd*, *simd (AVX2)* and *simd (AVX-512)*), aligned access (*aligned*) and loop unrolling (*unrolled*). Table 1 shows the execution time (in seconds) of the different serial versions when N=4096. As it can be observed, blocking optimization reduces execution time by 5%. Regarding the block size, 256×256 was found to work best. In the most memory demanding case of each round (phase 4), four blocks are loaded into the cache (3 distance blocks and 1 path block). The four blocks requires $4 \times 256 \times 256 \times 4$ bytes = 1024 KB = 1MB, which is exactly the L2 cache size.

As stated in Section 4.2, the compiler is not able to generate SIMD binary code by itself in the blocked version. Adding the corresponding *simd* constructs

Table 1: Execution time (in seconds) of the different optimization techniques applied to the naive serial version when N=4096.

naive	*blocked*	*simd*	*simd (AVX2)*	*simd (AVX-512)*	*aligned*	*unrolled*
602.8	572.66	204.52	100.47	36.95	33.28	22.95

to the blocked version reduced the execution time from 572.66 to 204.52 seconds, which represents a speedup of 2.8×. However, AVX-512 instructions can perform 16 SP operations at the same time. After inspecting the code at assembly level, we realized that the compiler generates SSEx instructions by default. As SSEx can perform 4 SP operations at the same time, the 2.8× speedup has more sense since not all the code can be vectorized. Next, we re-compiled the code including the *-xAVX2* and *-xMIC-AVX512* flags to force the compiler to generate AVX2 and AVX-512 SIMD instructions, respectively. AVX2 extensions accelerated the blocked version by a factor of 5.8× while AVX-512 instructions achieved an speedup of 15.5×. So, it is clear that this application benefits from larger SIMD width. In relation to the other optimization techniques employed, we have found that the *simd (AVX-512)* implementation runs 1.11× faster when aligning memory accesses in AVX-512 computations (*aligned*). Additionally, applying the loop unrolling optimization to the *aligned* version led to higher performance, gaining a 1.45× speedup. In summary, we achieve a 26.3× speedup over the naive serial version through the combination of the different optimizations described.

Taking the optimized serial version, we developed a multi-threaded implementation as described in Section 4.4. Figure 6 shows the performance (in terms of GFLOPS) for the different affinity types used varying the number of threads when N=8192. As expected, *compact* affinity produced the worst results since it favours using all threads on a core before using other cores. *Scatter* and *balanced* affinities presented similar performances improving the *none* counterpart. As the KNL processor used in this study has all its cores in the same package, *scatter* and *balanced* affinities distribute the threads in the same manner when one thread per core is assigned. Regarding the number of threads, using a single thread per core is enough to get maximal performance (except in *compact* affinity). This behavior is opposed to the KNC generation where two or more threads per core where required to achieve high performance. However, it should not be a surprise since the KNL cores were designed to optimize single thread performance including out-of-order pipelines and two VPUs per core.

It is important to remark that, unlike the optimized serial version, the parallel implementation used a smaller block size since it delivered higher performance. A smaller block size allowed a finer-grain workload distribution and decreased thread idling, especially when the number of threads was larger than the number of blocks in phases 2 and 3. Another reason to decrease block size was that the L2 available space is now shared between the threads in a tile, contrary to the single threaded case. In particular, BS=64 was found to work best.

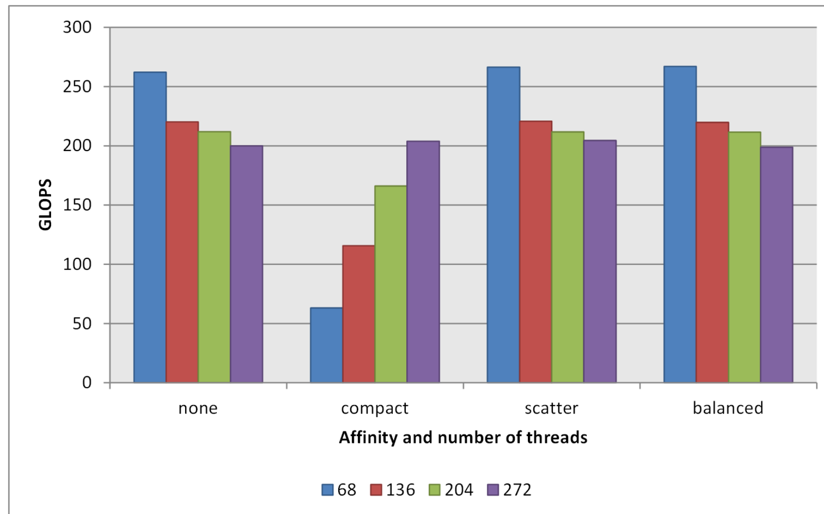

Fig. 6: Performance for the different affinity types used varying the number of threads when N=8192.

Figure 7 illustrates performance evolution varying workload and MCDRAM exploitation for the different vectorization approaches. For small workloads (N = 8192), the performance improvement is little ($\sim$1.1×). However, MCDRAM memory presents remarkable speedups for greater workloads, even when the dataset largely exceeds the MCDRAM size (N = 655536). In particular, MCDRAM exploitation achieves an average speedup of 9.8× and a maximum speedup of 15.5×. In this way, we can see how MCDRAM usage is an efficient strategy for bandwidth-sensitive applications.

In relation to the vectorization approach, we can appreciate that guided vectorization leads to slightly better performance than the intrinsic counterpart, running upto 1.03× faster. The best performances are 330 and 338 GFLOPS for the intrinsic and guided versions, respectively. After analyzing the assembly code, we realized that this difference is caused by the prefetching instructions introduced by the compiler when guided vectorization is used. Unfortunately, the compiler disables automatic prefetching when code is manually vectorized.

6 Related Works

Despite its recent commercialization, there are some works that evaluate KNL processors. In that sense, we highlight [18] that presents a study of the performance differences observed when using the three MCDRAM configurations available in combination with the three possible memory access or cluster modes. Also, Barnes et al. [3] discussed the lessons learned from optimizing a number of different high-performance applications and kernels. Besides, Haidar et al. [9]

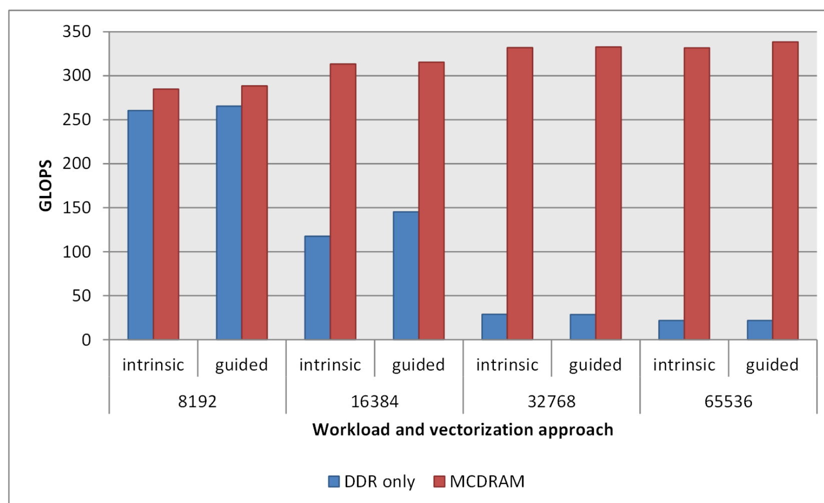

Fig. 7: Performance evolution varying workload and the MCDRAM exploitation.

proposed and evaluated several optimization techniques for different matrix factorizations methods on many-core systems.

Obtaining high-performance in graph algorithms is usually a difficult task since they tend to suffer from irregular dependencies and large space requirements. Regarding FW algorithm, there are many works proposed to solve the all-pairs shortest paths problem on different harwdare architectures. However, to the best of the authors knowledge, there are no related works with KNL processors. Han and Kang [10] demonstrated that exploiting SSE2 instructions led to 2.3×-5.2× speedups over a blocked version. Bondhugula et al. [4] proposed a tiled parallel implementation using Field Programmable Gate Arrays. In the field of GPUs, we highlight the work of Katz and Kider [13], who proposed a shared memory cache efficient implementation to handle graph sizes that are inherently larger than the DRAM memory available on the device. Also, Matsumoto et al. [15] presented a blocked algorithm for hybrid CPU-GPU systems aimed to minimize host-device communication. Finally, Hou et al. [11] evaluated different optimization techniques for Xeon Phi KNC coprocessor. Just as this study, they found that blocking and vectorization are key aspects in this problem to achieve high performance. Also, guided vectorization led to better results than the manual approach, but with larger performance differences. Contrary to this work, their implementation benefited from using more than one thread per core. However, as stated before, there are significant architectural differences between these platforms that support this behavior.

7 Conclusions

KNL is the second generation of Xeon Phi family and features new technologies in SIMD execution and memory access. In this paper, we have evaluated a set of programming and optimization techniques for these processors taking the FW algorithm as a representative case study of graph and memory-bound applications. Among the main contributions of this research we can summarize:

- Blocking technique not only improved performance but also allowed us to apply a coarse-grain workload distribution in the parallel implementation.
- SIMD exploitation was crucial to achieve top performance. In particular, the serial version run 2.9×, 6× and 15.5× faster using the SSE, AVX2 and AVX-512 extensions, respectively.
- Aligning memory accesses and loop unrolling also showed significant speedups.
- A single thread per core was enough to get maximal performance. In addition, *scatter* and *balanced* affinities provided extra performance.
- Besides keeping portability, guided vectorization led to slightly better performance than the intrinsic counterpart, running upto 1.03× faster.
- MCDRAM usage demonstrated to be an efficient strategy to tolerate high-bandwidth demands with practically null programmer intervention, even when the dataset largely exceeded the MCDRAM size. In particular, it produced an average speedup of 9.8× and a maximum speedup of 15.5×

As future work, we consider evaluating programming and optimization techniques in other cluster and memory modes as a way to extract more performance.

Acknowledgments The authors thank the ArTeCS Group from Universidad Complutense de Madrid for letting use their Xeon Phi KNL system.

References

1. Green500 Supercomputer Ranking, https://www.green500.org/
2. Top500 Supercomputer Ranking, https://www.top500.org/
3. Barnes, T., et al.: Evaluating and Optimizing the NERSC Workload on Knights Landing. In: Proceedings of the 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems. pp. 43–53. PMBS '16, IEEE Press, Piscataway, NJ, USA (2016)
4. Bondhugula, U., Devulapalli, A., Dinan, J., Fernando, J., Wyckoff, P., Stahlberg, E., Sadayappan, P.: Hardware/software integration for fpga-based all-pairs shortest-paths. In: 2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. pp. 152–164 (April 2006)
5. Codreanu, V., Rodrguez, J., Saastad, O.W.: Best Practice Guide - Knights Landing (2017), http://www.prace-ri.eu/IMG/pdf/Best-Practice-Guide-Knights-Landing.pdf
6. Culler, D.E., Gupta, A., Singh, J.P.: Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn. (1997)

7. Floyd, R.W.: Algorithm 97: Shortest path. Commun. ACM 5(6), 345– (Jun 1962)
8. Giles, M.B., Reguly, I.: Trends in high-performance computing for engineering calculations. Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences 372(2022) (2014)
9. Haidar, A., Tomov, S., Arturov, K., Guney, M., Story, S., Dongarra, J.: LU, QR, and Cholesky factorizations: Programming model, performance analysis and optimization techniques for the Intel Knights Landing Xeon Phi. In: 2016 IEEE High Performance Extreme Computing Conference (HPEC). pp. 1–7 (Sept 2016)
10. Han, S., Kang, S.: Optimizing all-pairs shortest-path algorithm using vector instructions (2005)
11. Hou, K., Wang, H., c. Feng, W.: Delivering parallel programmability to the masses via the intel mic ecosystem: A case study. In: 2014 43rd International Conference on Parallel Processing Workshops. pp. 273–282 (Sept 2014)
12. Jalali, S., Noroozi, M.: Determination of the optimal escape routes of underground mine networks in emergency cases. Safety Science 47(8), 1077 – 1082 (2009)
13. Katz, G.J., Kider, Jr, J.T.: All-pairs shortest-paths for large graphs on the gpu. In: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware. pp. 47–55. GH '08, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland (2008)
14. Khan, P., Konar, G., Chakraborty, N.: Modification of floyd-warshall's algorithm for shortest path routing in wireless sensor networks. In: 2014 Annual IEEE India Conference (INDICON). pp. 1–6 (Dec 2014)
15. Matsumoto, K., Nakasato, N., Sedukhin, S.G.: Blocked all-pairs shortest paths algorithm for hybrid cpu-gpu system. In: 2011 IEEE International Conference on High Performance Computing and Communications. pp. 145–152 (Sept 2011)
16. Nakaya, A., Goto, S., Kanehisa, M.: Extraction of correlated gene clusters by multiple graph comparison. Genome Informatics 12, 44–53 (2001)
17. Reinders, J., Jeffers, J., Sodani, A.: Intel Xeon Phi Processor High Performance Programming Knights Landing Edition. Morgan Kaufmann Publishers Inc., Boston, MA, USA (2016)
18. Rosales, C., Cazes, J., Milfeld, K., Gómez-Iglesias, A., Koesterke, L., Huang, L., Vienne, J.: A Comparative Study of Application Performance and Scalability on the Intel Knights Landing Processor, pp. 307–318. Springer International Publishing, Cham (2016)
19. Venkataraman, G., Sahni, S., Mukhopadhyaya, S.: A Blocked All-Pairs Shortest-Paths Algorithm, pp. 419–432. Springer Berlin Heidelberg (2000)
20. Warshall, S.: A theorem on boolean matrices. J. ACM 9(1), 11–12 (Jan 1962)