Early Experience on Using Knights Landing Processors for Lattice Boltzmann Applications

Enrico Calore[1,2], Alessandro Gabbana[1,2,3], Sebastiano Fabio Schifano[1,2](✉), and Raffaele Tripiccione[1,2]

[1] Università degli Studi di Ferrara, Ferrara, Italy
schifano@fe.infn.it
[2] INFN Ferrara, Ferrara, Italy
[3] Bergische Universität Wuppertal, Wuppertal, Germany

Abstract. The Knights Landing (KNL) is the codename for the latest generation of Intel processors based on Intel Many Integrated Core (MIC) architecture. It relies on massive thread and data parallelism, and fast on-chip memory. This processor operates in standalone mode, booting an off-the-shelf Linux operating system. The KNL peak performance is very high – approximately 3 Tflops in double precision and 6 Tflops in single precision – but sustained performance depends critically on how well all parallel features of the processor are exploited by real-life applications. We assess the performance of this processor for Lattice Boltzmann codes, widely used in computational fluid-dynamics. In our OpenMP code we consider several memory data-layouts that meet the conflicting computing requirements of distinct parts of the application, and sustain a large fraction of peak performance. We make some performance comparisons with other processors and accelerators, and also discuss the impact of the various memory layouts on energy efficiency.

Keywords: Lattice Boltzmann methods · Memory data layouts
Performance analysis · Knights Landing

1 Introduction

Hi-end processors commonly used in HPC computer systems, have seen a steady increase in the number of processing cores and operations per clock-cycle. This trend has been further pushed forward in accelerators, such as GPUs and Intel Xeon-Phi processors based on the *Many Integrated Cores* (MIC) architecture, offering large computing power together with a high ratio of computing power per Watt. However, the use of accelerated systems is not without problems. The link between host CPU and accelerator, usually based on PCIe interface, creates a data bottleneck that reduces the sustained performance of most applications. Reducing the impact of this bottleneck in heterogeneous systems requires complex implementations [1,2] with a non negligible impact on development and

© Springer International Publishing AG, part of Springer Nature 2018
R. Wyrzykowski et al. (Eds.): PPAM 2017, LNCS 10777, pp. 519–530, 2018.
https://doi.org/10.1007/978-3-319-78024-5_45

maintenance efforts. The latest generation of Xeon-Phi processor, codename *Knights Landing* (KNL), offers a way out of this problem: it is a self-hosted system, running a standard Linux operating system, so it can be used alone to assemble homogeneous clusters.

In this work we present an early assessment of the performance of the KNL processor, using as test-case a state-of-the-art Lattice Boltzmann (LB) code. For regular applications like LB codes, task parallelism is easily done by assigning tiles of the physical lattice to different cores. However, exploiting data-parallelism through vectorization requires additional care, and in particular a careful design of the data layout is critical to allow an efficient use of vector instructions. Our code uses OpenMP to manage task parallelism, and we experiment with different data-layouts trying to find a compromise between the conflicting requirements of the two main critical compute-intensive kernels *propagate* and *collide*. We then assess the impact of several layout choices in terms of computing and energy performance. Recent works have studied the performance of KNL [3–5] with several applications, but as far as we know none of these investigate the impact of data layouts on computing performance and energy efficiency of applications. Concerning data-layouts, [6–8] study optimal data structures for LB simulations. However, [6] analyses only the *propagate* kernel, while [7] does not take into account vectorization of the code. In [8] vectorization is exploited using intrinsic functions only. Conversely, in the present work we aim to allow efficient vectorization by the compiler for both *propagate* and *collide* steps for the KNL architecture. The rest of this paper is organized as following: Sect. 2 gives a short overview of the KNL architecture, highlighting the main features relevant for this work; Sect. 3 briefly sketches an outline of the Lattice Boltzmann method, while Sect. 4 presents the various options for data-layout that we have studied; Sect. 5 analyzes our results, and Sect. 6 ends with some concluding remarks.

2 Overview of Knights Landing Architecture

The Xeon-Phi codename *Knights Landing* (KNL) is the second generation of Intel processors based on the MIC architecture, and the first self-bootable processor in this family. It has an array of 64, 68 or 72 cores and four high speed memory banks based on the *Multi-Channel DRAM* (MCDRAM) technology providing an aggregated bandwidth of more than 450 GB/s [9]. It also integrates 6 DDR4 channels supporting up to 384 GB of memory with a peak raw bandwidth of 115.2 GB/s. Two cores form a tile and share an L2-cache of 1 MB. Tiles are connected by a 2D-mesh of rings and can be clustered in several NUMA configurations. In this work we only consider the *Quadrant* cluster configuration in which tiles are divided in four quadrants, each directly connected to one MCDRAM bank. This configurations is the recommended one to use the KNL as a symmetric multi-processor, as it reduces the latency of L2-cache misses, and the 4 blocks of MCDRAM appear as contiguous block of addresses. For more details on clustering see [10]. MCDRAM on a KNL can be configured at boot time in FLAT, CACHE or Hybrid mode. The FLAT mode defines the whole

MCDRAM as addressable memory allowing explicit data allocation, whereas CACHE mode uses the MCDRAM as a last-level cache between the L2-caches and the on-platform DDR4 memory. In Hybrid mode, the MCDRAM is used partly as addressable memory and partly as cache. For more details on memory configuration see [11]. In this work we only consider FLAT and CACHE modes. Parallelism is exploited at two levels on the KNL: *task parallelism* builds onto the large number of integrated cores, while *data parallelism* uses the AVX 512-bit vector (SIMD) instructions. Each core has two out-of-order vector processing units (VPUs) and supports the execution of up to 4 threads. The KNL has a peak theoretical performance of 6 TFlops in single precision and 3 TFlops in double precision. Typical thermal design power (TDP) is 215 W including MCDRAM memories (but not the Omni-Path interface). For more details on KNL architecture see [12].

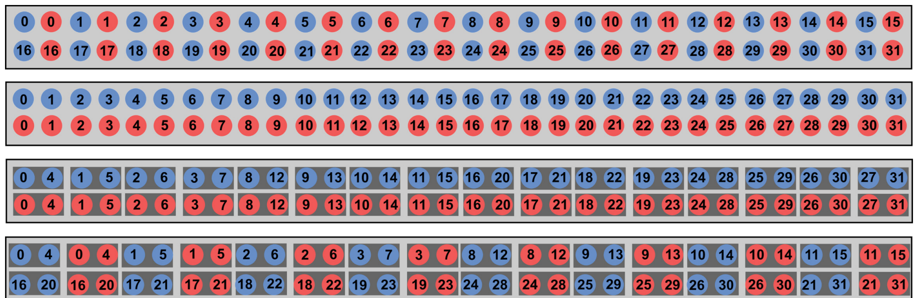

Fig. 1. Top to bottom, *AoS*, *SoA*, *CSoA* and *CAoSoA* data memory layouts for a 4×8 lattice with two populations (red and blue) per site. For *CSoA* and *CAoSoA* each grey-box is a cluster with $\mathrm{VL} = 2$. Memory addresses increase left-to-right top-to-bottom. (Color figure online)

3 Lattice Boltzmann Methods

Lattice Boltzmann Methods [13] (LBM) are widely used in computational fluid-dynamics, to describe fluid flows. They are used in science and engineering to accurately model single and multi-phase flows and can be easily accommodate irregular boundary conditions. This is why they are usually used in the oil&gas industry to study the dynamics of oil and shale-gas reservoirs and to maximize their yield. This class of applications, discrete in time and momenta and living on a discrete and regular grid of points, offers a large amount of available parallelism, so they are an ideal target for multi- and many-core processors. LBM are based on the synthetic dynamics of *populations* corresponding to (pseudo-)particles sitting at the sites of a discrete lattice. At each time step, populations *propagate* from lattice-sites to lattice-sites, and then *collide* mixing and changing their values accordingly. In these processes, there is no data dependency between different lattice points, so both the *propagate* and *collide* steps can

be performed in parallel on all grid points following any convenient schedule. A model describing flows in n dimensions and using m populations is labeled as $DnQm$. In this work we study a D2Q37 model, a 2-dimensional system with 37 populations associated to each lattice-site moving up to three lattice points away. This recently developed [14,15] LB model automatically enforces the equation of state of a perfect gas ($p = \rho T$). It has been recently used to perform large scale simulations of convective turbulence in several physics regimes [16,17].

4 Implementation and Optimization of D2Q37 LB Model

In LB applications, *propagate* and *collide* take most of the compute-time of the whole code, so optimization efforts have to target largely on these two kernels. The D2Q37 model is computationally more demanding than other simpler methods, because *propagate* is strongly memory-bound accessing 37 neighbor cells to gather all populations and generating sparse memory accesses, while *collide* is strongly compute-bound and executes ≈6600 double-precision floating point operations per lattice point. In this section we focus mostly on data memory-layouts which are becoming more and more important for exploiting vector performance on recent many-core processors. In the following we discuss several possible choices and we show that they have very large effects on computing and energy performances for the KNL processor. Here, we extend previous works [2,18], where additional details on other aspects of the code structure are available.

Array of Structures (*AoS*) and *Structure of Arrays* (*SoA*) are a starting points to implement more complex data memory organizations. In the *AoS* scheme, population data associated to each lattice site are stored one after the other at contiguous memory addresses. In this arrangement all data associated to one lattice point are at close memory locations, but same index populations of different lattice sites are stored in memory at non-unit stride addresses. To handle this, the compiler makes intensive use of `GATHER` and `SCATTER` SIMD instructions which are up to 10X slower than contiguous vector loads and stores (`VMOVE`) of 8 double-precision elements [19] resulting in poor data locality with many `L2 Misses` during the execution as shown in Table 1. Conversely, the *SoA* scheme stores same index populations of all sites one after the other. This is appropriate for vector SIMD instructions, as it allows to move several lattice sites – 8 for the KNL – in parallel. Figure 1 – first two designs from the top – visualize the *AoS* and *SoA* memory layouts, for a mockup lattice of 4×8 with two populations (red and blue) per site.

The *SoA* layout stores same index populations of all lattice-sites one after the other reducing the L2 miss-rate (see again Table 1), but introduces a potential inefficiency associated to unaligned memory accesses. In fact, the read-address for population values is computed as the sum of the address of the current site plus an offset, and the resulting address is in general not aligned to a 64 Byte boundary, preventing direct memory copies to vector registers. In order to circumvent this problem, we start from the *SoA* layout and, for a lattice of size

Table 1. Efficiency metrics measuring the impact of the different data-layouts on L2-CACHE and L2-TLB misses for *propagate* and *collide* kernels. The values are absolute numbers, and thresholds is a value suggested by Intel [19] to investigate code implementation if exceeded.

Metric	*AoS*	*SoA*	*CSoA*	*CAoSoA*	Threshold
propagate L2 CACHE miss rate	0.50	0.10	0.05	0.00	<0.20
collide L2 TLB miss overhead	0.00	0.21	1.00	0.00	<0.05

```
#define LYOVL (LY / VL)
typedef struct { double c[VL]; } vdata_t;
typedef struct { vdata_t s[LX*LYOVL]; } vpop_csoa_t;
vpop_csoa_t prv[NPOP], nxt[NPOP];
#pragma omp parallel for num_threads(NTHREAD) schedule(dynamic)
for ( ix = startX; ix < endX; ix++ ) {
  idx = (NYOVL*ix) + HYOVL;
  for( p = 0; p < NPOP; p++){
    for ( iy = 0; iy < SIZEYOVL; iy++ ) {
      #pragma unroll
      #pragma vector aligned nontemporal
      for(k = 0; k < VL;k++)
        nxt[p].s[idx+iy].c[k] = prv[p].s[idx+iy+OFF[p]].c[k]
} } }
```

Listing 1. Source code of `propagate` kernel for using the *CSoA* data layouts. `OFF` is a vector containing memory-address offsets associated to each population hop. `VL` is the size of a cluster.

$LX \times LY$, we cluster together `VL` elements of each population at a distance LY/VL, with `VL` a multiple of the KNL vector size. We call this data layout a *Cluster Structure of Array* (CSoA), see Fig. 1 – third design from top – for the case of $VL = 2$ corresponding to an hypothetical processor using vectors consisting of two 64-bit values. Using *CSoA*, *propagate*, whose main task is to read the same population elements at all sites and move them to different sites, is able to use vector instructions to process clusters of properly memory-aligned items. Listing 1 shows the corresponding `C` type definitions and code implementation for *propagate*.

The outer loop on X spacial direction is parallelized at a thread level using the OpenMP pragma parallel loop, making each thread to work on a slice of the lattice. The inner loop, copying elements of a cluster into another cluster, can be unrolled and vectorized since both read and write pointers are now properly aligned. This is confirmed by the compiler optimization report and by inspection of generated assembly code, now consisting of aligned load and store (`VMOVE`) vector instructions. A further optimization can in this case be applied with the use of non-temporal write operations saving time and reducing the overall memory traffic by 1/3 [2]. We instruct the compiler to do this through the pragmas `unroll` and `vector aligned nontemporal`. Figure 2 shows measured bandwidth for our data structures, using the FLAT mode, and using both

off-chip and MCDRAM memory, and the CACHE memory mode. The results refer to a 64 core Xeon-Phi 7230 running at 1.4 GHz.

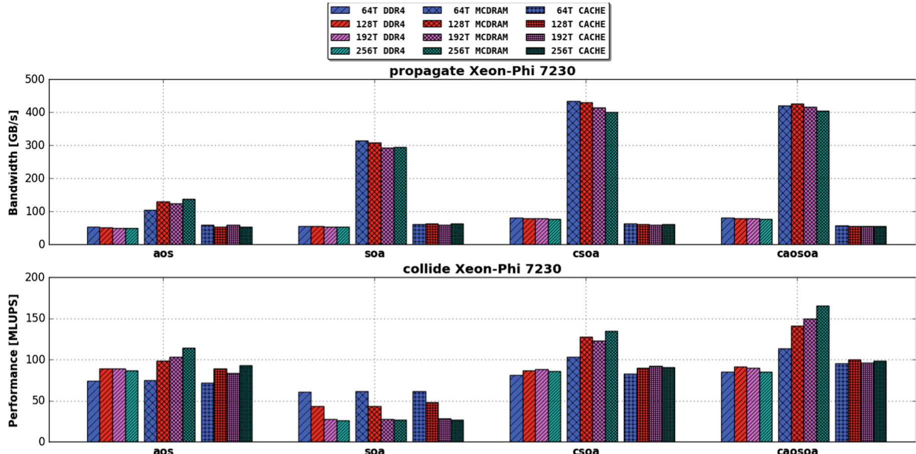

Fig. 2. Performance of *propagate* (top) is in GB/s and *collide* (bottom) is in MLUPS. All data for a 64 core Xeon-Phi 7230 running at 1.4 GHz. For the FLAT configuration we use a 2304×8192 lattice that fits into MCDRAM; for the CACHE configuration, the lattice is 4608×12288, twice the size of MCDRAM. For each layout, 3 groups of 4 bars correspond respectively to FLAT-DDR4, FLAT-MCDRAM and CACHE. Within each group, bars correspond respectively to 1, 2, 3 and 4 threads per core.

The *collide* kernel can be vectorized using the same strategy as of *propagate*, so one expects the *CSoA* layout should be an efficient choice. However, profiling the execution of this kernel, we found that a large number of L2-TLB misses are generated (see Table 1). This happens because different populations associated to each lattice site are stored at memory addresses far from each other, and several non-unit stride reads are necessary to load all population values necessary to compute the collisional operator. To reduce this penalty, we start again from the *SoA* layout, and for each population array, we divide each Y-column in `VL` partitions each of size `LY/VL`. All elements sitting at the ith position of each partition are then packed together into an array of `VL` elements called *cluster*. For each index i we then store in memory one after the other the 37 clusters – one for each population – associated to it. This defines a new data-structure called *Clustered Array of Structure of Arrays* (CAoSoA). The main improvement on *CSoA* is that it still allows vectorization of clusters of size VL, and at the same time improves locality of populations, keeping all population data associated to each lattice site at close and aligned addresses (see again Fig. 1 for a visual description). *CAoSoA* combines the benefits of the *CSoA* scheme, allowing aligned memory accesses and vectorization (relevant for the *propagate*) together with the benefits of the *AoS* layout providing population locality (relevant for

the *collide*). Taking into account that for KNL the cost of a L2 TLB miss is in the order of 100 clock cycles, these benefits can be quantitatively evaluated using the *L2 TLB Miss Overhead* metric reported in Table 1. For *CAoSoA* layout, usage of L2 TLB is as efficient as for the *AoS* case, whereas significant overheads are associated using to *SoA* and *CSoA* schemes.

5 Analysis of Results

In this section we present our performance results in terms of computing and energy. We also compare computing results with that we have measured on other multi- and many-core architectures.

5.1 Experimental Setup

We have run our tests on a desktop machine with a Xeon-Phi 7230 processor running at clock frequency of 1.3 GHz, and 128 MB of DDR4 memory. We have tested the FLAT and CACHE memory configurations. For the FLAT configuration we have allocated the data-domain of our application either on the 16 GB on-chip MCDRAM, or on the off-chip DDR4 memory. For the CACHE configuration we have used a lattice size that does not fit the 16 GB of on-chip memory. We have fixed the configuration of the cluster of cores to quadrant. This configurations is that recommended by Intel as it reduces the latency of L2-cache misses, the 4 blocks of MCDRAM appear as contiguous block of addresses, and the processor can be used as symmetric multi-processor. Tests are run launching one MPI process which spawns 1, 2, 3 and 4 threads per core.

5.2 Computing Performance Results

We summarize our performance results analyzing data reported in figure Fig. 2, where we report the measured performance for the *propagate* kernel measured in GB/s and the *collide* kernel – expressed in *Million Lattice UPdates per Second*, a common figure of performance for this operation – for all data-layouts considered so far. For both kernels we have analyzed the performance using the FLAT and CACHE memory configurations. For the *propagate* kernel, performance is almost independent from the number of threads per core, while the impact of the various data layouts is large. Indeed, using a FLAT MCDRAM configuration the measured bandwidth increases from 138 GB/s of *AoS* to 314 GB/s of *SoA* and to 433 GB/s of *CSoA*. This trend is similar using the DDR4 memory bank but performance is much lower, ranging from 54 GB/s of *AoS* to 56 GB/s of *SoA* and to 81 GB/s of *CSoA*. We have a similar behavior also with the CACHE configuration, measuring in this case a bandwidth of 59, 60 and 62 GB/s for the *AoS*, *SoA* and *CSoA* memory layouts for a lattice size that does not fit into MCDRAM. Using the *CAoSoA* layout, performance does not further improves, both for FLAT and CACHE configurations.

For *collide* kernel, using a FLAT configuration and MCDRAM, we obtain a good level of performance, 114 MLUPS, using the *AoS* layout with 4 threads per core. The *SoA* layout performance does not allow efficient vectorization, so performance goes down to 62 MLUPS with one thread per core, further decreasing if we use 2, 3 and 4 threads per core. Enforcing memory alignment with the *CSoA* layout, we obtain again a properly vectorized code and performance increases up to 135 MLUPS using 4 threads per core. Performances further improve with the *CAoSoA* layout removing the overhead associated to TLB misses, and reaching the level of 165 MLUPS with 4 threads per core, corresponding to a factor 1.4X and 1.2X w.r.t. the *AoS* and *CSoA* layouts. The *collide* kernels performs ≈6600 floating-point operations per lattice site. The KNL processor then delivers a sustained performance of approximately 1 TFlops using the *CAoSoA* layout, corresponding to ≈30% of the available raw peak. Using DDR4 results follows the same trend as in the MCDRAM case, but performances are harmed by memory bandwidth, reaching 89 MLUPS with the *CAoSoA* layout. The same is true with CACHE configuration where *collide* reaches a peak of 98 MLUPS for the *CAoSoA* layout.

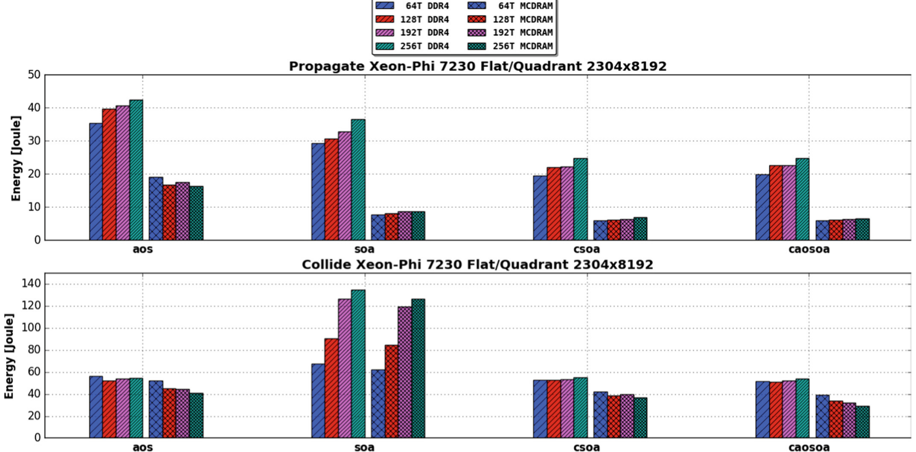

Fig. 3. Energy-to-Solution for *propagate* (top) and *collide* (bottom), for all data layouts, using the FLAT configuration. For each layout we plot two groups of bars corresponding to the use of either DDR4 off-chip memory or on-chip MCDRAM. Within each group the bars correspond respectively to 1, 2, 3 and 4 threads per core. All values are computed as the sum of the *Package* and *DRAM* RAPL energy counters, per iteration.

5.3 Energy Performance Results

We now consider energy efficiency for our code. We use data from the RAPL (Running Average Power Limit) register counters available in the KNL read through the custom library we have developed in [20]. Figure 3 shows the results

for FLAT configuration using both MCDRAM and DDR4, and assessing the impact of data-layouts on energy consumption. All figures refer to *Energy-to-Solution* (E_S) and are the sum of package and off-chip DDR4 contributions. For *propagate*, we see that the average power drain increase using MCDRAM (≈35%) compared to the use of off-chip DDR4, but E_S is lower since a slightly higher power gets integrated over a much shorter (≈4×) time. Also, the *CSoA* and *CAoSoA* data-layouts halve E_S w.r.t. the *AoS* and *SoA* as a result of their shorter execution times and slightly lower power drain. For the *collide* kernel the *SoA* has a rather low power drain (≈30% less than *CSoA* and *CAoSoA*) because vector units are not used. However, the code runs also much slower (≈3×), translating into the worst performance figure in terms of E_S. Conversely, the *CAoSoA* gives the best result in terms of energy efficiency, with E_S decreasing while increasing the number of threads per core, thanks to a constant power drain and an increasing performance. Using CACHE configurations, the average power drain is in between the values recorded for the DDR4 and MCDRAM cases. As shown in Fig. 2 performances are similar to the case of DDR4, with a slightly performance decrease for *propagate* and a slightly increase for *collide* when using *CSoA* and *CAoSoA* data-layouts. Thus, from the energy consumption point of view, using cache configuration leads to similar energy behaviors as using DDR4.

Table 2. Performance comparison among several processors. We consider the *propagate* and *collide* kernels and the full code (Global), using the *CAoSoA* data layout. We compare the KNL against the KNC, the NVIDIA GK210 and P100 GPUs, and the Intel E52697v4 CPU. The row labeled with *Global* report the performance of the full code.

	KNC 7120P	GK210	P100	E52697v4	KNL 7230 flat/quad	KNL 7230 cache/quad	KNL 7230 cache/quad
Lattice size	1024 × 8192						4608 × 12288
Memory footprint [GB]	≈4.6						≈30
T_{prop} [ms]	49.9	32.3	12.5	98.06	12.5	19.65	506.64
T_{coll} [ms]	180.9	71.1	24.1	173.42	50.3	51.42	550.25
Propagate [GB/s]	100	155	396	51	398	253	66
Collide [GF/s]	307	764	2253	320	1100	1079	680
Collide [MLUPS]	46	115	340	48	166	163	103
Global [MLUPS]	35	73	232	31	119	106	67

5.4 Comparison with Other Processors

We finally compare our performance results of our code running on KNL, with that we have measured on other recent multi- and many-core processors. Our

comparison is shown in Table 2 for both critical kernels and also for the complete code. We adopt the *CAoSoA* layout throughout, as it offers the best performance. Let first discuss the case of lattice size 1024×8192 requiring a memory footprint of ≈4.6 GB fitting the 16 GB on-chip MCDRAM. The data size also fits most other accelerator boards, so we can perform a meaningful comparison. Comparing the KNL in FLAT mode with the KNC [21], the previous generation Xeon-Phi processor, the performance for *propagate* and *collide* is respectively ≈4X and ≈3.5X faster. Comparing with NVIDIA GPUs [22,23], the execution time for *propagate* is ≈2.5X faster than on a GK2010 GPU (hosted on a K80 board), and the same as a P100 Pascal board. The execution time of *collide* is 1.4X faster than a GK210, and approximately 50% slower than a P100. Comparing performances with a more traditional Intel E5-2697v4 CPU [24], based on Broadwell micro-architecture, *propagate* is 7.8X faster and *collide* is 3.5X faster. Using the KNL in CACHE mode with a lattice that does not fit into MCDRAM, the performance of the processor is much slower. In the last column of Table 2 we see the results for a lattice using a memory footprint twice the size of MCDRAM. In this case, comparing with CPU E5-2697v4 for which the lattice 1024×8192 does not fit in the last-level cache, the performance of *propagate* is more or less the same, and that of *collide* is ≈2X faster.

6 Conclusions

In summary, based on our experience related to our application, some concluding remarks are in order: (i) the KNL architecture makes it easy to port and run codes previously developed for X86 standard CPUs. However performance is strongly affected by the massive level of parallelism that must necessarily be exploited, to avoid that the level of performance drops to the value of standard multi-core CPUs or even worst; (ii) for this reason data layouts plays a relevant role in allowing to reach an efficient level of vectorization. At least for LB applications, appropriate data structures are necessary to allow the different vectorization strategies necessary in different parts of the application; (iii) the KNL processor improves on the KNC – the previous generation Xeon-Phi processor – by a factor ≈$3 - 4$X; (iv) if application data fits within the MCDRAM, performances are very competitive with that of GPU accelerators. However, if this is not the case, performance drops to levels similar to those of multi-core CPUs, with the further drawback that codes and operations (editing, compilations, IO, etc.) not exploiting task and data parallelism run much slower.

In the future, we plan to further analyze the energy performance of KNL comparing with other processors, and to design and develop a parallel hybrid MPI+OpenMP code able to run on a cluster of KNLs, in order to investigate scalability.

Acknowledgements. This work was done in the framework of the COKA, COSA projects of INFN, and the PRIN2015 project of MIUR. We would like to thank CINECA (Italy) for access to their HPC systems. AG has been supported by the EU Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No. 642069.

References

1. Tang, P., et al.: An implementation and optimization of lattice Boltzmann method based on the multi-node CPU+MIC heterogeneous architecture. In: International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), pp. 315–320 (2016). https://doi.org/10.1109/CyberC.2016.67
2. Calore, E., et al.: Optimization of Lattice Boltzmann simulations on heterogeneous computers. Int. J. High Perform. Comput. Appl. 1–16 (2017). https://doi.org/10.1177/1094342017703771
3. 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. In: Taufer, M., Mohr, B., Kunkel, J.M. (eds.) ISC High Performance 2016. LNCS, vol. 9945, pp. 307–318. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46079-6_22
4. Li, S., et al.: Enhancing application performance using heterogeneous memory architectures on a many-core platform. In: International Conference on High Performance Computing Simulation (HPCS), pp. 1035–1042 (2016). https://doi.org/10.1109/HPCSim.2016.7568455
5. Rucci, E., et al.: First Experiences Optimizing Smith-Waterman on Intel's Knights Landing Processor. ArXiv e-prints, February 2017
6. Wittmann, M., et al.: Comparison of different propagation steps for the lattice Boltzmann method. CoRR abs/1111.0922 (2011)
7. Shet, A.G., et al.: Data structure and movement for lattice-based simulations. Phys. Rev. E **88**, 013314 (2013). https://doi.org/10.1103/PhysRevE.88.013314
8. Shet, A.G., et al.: On vectorization for lattice based simulations. Int. J. Mod. Phys. C **24**, 1340011 (2013). https://doi.org/10.1142/S0129183113400111
9. McCalpin, J.D.: Stream: sustainable memory bandwidth in high performance computers (2017). https://www.cs.virginia.edu/stream/
10. Colfax: Clustering modes in knights landing processors (2017). https://colfaxresearch.com/knl-numa/
11. Colfax: MCDRAM as high-bandwidth memory (HBM) in knights landing processors: developers guide (2017). https://colfaxresearch.com/knl-mcdram/
12. Sodani, A., et al.: Knights landing: second-generation Intel Xeon Phi product. IEEE Micro **36**(2), 34–46 (2016). https://doi.org/10.1109/MM.2016.25
13. Succi, S.: The Lattice-Boltzmann Equation. Oxford University Press, Oxford (2001)
14. Sbragaglia, M., et al.: Lattice Boltzmann method with self-consistent thermo-hydrodynamic equilibria. J. Fluid Mech. **628**, 299–309 (2009). https://doi.org/10.1017/S002211200900665X
15. Scagliarini, A., et al.: Lattice Boltzmann methods for thermal flows: continuum limit and applications to compressible Rayleigh-Taylor systems. Phys. Fluids **22**(5), 055101 (2010). https://doi.org/10.1063/1.3392774
16. Biferale, L., Mantovani, F., Pivanti, M., Sbragaglia, M., Scagliarini, A., Schifano, S.F., Toschi, F., Tripiccione, R.: Lattice Boltzmann fluid-dynamics on the QPACE supercomputer. Proc. Comput. Sci. **1**(1), 1075–1082 (2010). https://doi.org/10.1016/j.procs.2010.04.119
17. Biferale, L., et al.: Second-order closure in stratified turbulence: simulations and modeling of bulk and entrainment regions. Phys. Rev. E **84**(1), 016305 (2011). https://doi.org/10.1103/PhysRevE.84.016305

18. Calore, E., Demo, N., Schifano, S.F., Tripiccione, R.: Experience on vectorizing lattice Boltzmann kernels for multi- and many-core architectures. In: Wyrzykowski, R., Deelman, E., Dongarra, J., Karczewski, K., Kitowski, J., Wiatr, K. (eds.) PPAM 2015. LNCS, vol. 9573, pp. 53–62. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-32149-3_6
19. Jeffers, J., et al.: Intel Xeon Phi Processor High Performance Programming, 2nd edn, pp. 213–250. Morgan Kaufmann, Boston (2016). https://doi.org/10.1016/B978-0-12-809194-4.00010-7
20. Calore, E., et al.: Evaluation of DVFS techniques on modern HPC processors and accelerators for energy-aware applications. Concurr. Comput.: Pract. Exp. **29**, 1–19 (2017). https://doi.org/10.1002/cpe.4143
21. Crimi, G., et al.: Early experience on porting and running a lattice Boltzmann code on the Xeon-Phi co-processor. Proc. Comput. Sci. **18**, 551–560 (2013). https://doi.org/10.1016/j.procs.2013.05.219
22. Biferale, L., et al.: An optimized D2Q37 lattice Boltzmann code on GP-GPUs. Comput. Fluids **80**, 55–62 (2013). https://doi.org/10.1016/j.compfluid.2012.06.003
23. Calore, E., et al.: Massively parallel lattice Boltzmann codes on large GPU clusters. Parallel Comput. **58**, 1–24 (2016). https://doi.org/10.1016/j.parco.2016.08.005
24. Mantovani, F., et al.: Performance issues on many-core processors: a D2Q37 lattice Boltzmann scheme as a test-case. Comput. Fluids **88**, 743–752 (2013). https://doi.org/10.1016/j.compfluid.2013.05.014