Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 551 – 560

Procedia Computer Science

International Conference on Computational Science, ICCS 2013

Early experience on porting and running a Lattice Boltzmann code on the Xeon-Phi co-processor

G. Crimi[a], F. Mantovani[b], M. Pivanti[d], S. F. Schifano[e,1], R. Tripiccione[c]

[a] *Università di Ferrara, ITALY*
[b] *Facultät für Physik, Univesität Regensburg, GERMANY*
[c] *Dip. di Fisica e Scienze della Terra and CMCS, Università di Ferrara and INFN, ITALY*
[d] *Dip. di Fisica, Università di Roma* La Sapienza *and INFN, ITALY*
[e] *Dip. di Matematica e Informatica, Università di Ferrara and INFN, ITALY*

Abstract

In this paper we report on our early experience on porting, optimizing and benchmarking a Lattice Boltzmann (LB) code on the Xeon-Phi co-processor, the first generally available version of the new *Many Integrated Core* (MIC) architecture, developed by Intel. We consider as a test-bed a state-of-the-art LB model, that accurately reproduces the thermo-hydrodynamics of a 2D-fluid obeying the equations of state of a perfect gas. The regular structure of LB algorithms makes it relatively easy to identify a large degree of available parallelism. However, mapping a large fraction of this parallelism onto this new class of processors is not straightforward. The D2Q37 LB algorithm considered in this paper is an appropriate test-bed for this architecture since the critical computing kernels require high performances both in terms of memory bandwidth for sparse memory access patterns and number crunching capability. We describe our implementation of the code, that builds on previous experience made on other (simpler) many-core processors and GPUs, present benchmark results and measure performances, and finally compare with the results obtained by previous implementations developed on state-of-the-art classic multi-core CPUs and GP-GPUs.

Keywords:
Lattice Boltzmann, Many-core systems, Performance optimization

1. Introduction

The *multi-core* design approach is quickly becoming the preferred way to further improve processor performances in spite of the fact that current micro-electronic technologies put a practical upper limit on clock frequency at approximately 3 GHz. A multi-core processor is a single chip integrating two or more independent CPUs. The number of cores within one chip is quickly growing: processors with 100 or more cores are expected in the near future. The many-core approach allows processors to scale according to Moore's law, but it bears a great impact on application design, further moving the challenge of sustaining performance from hardware to algorithms and software.

In this paper we report on our early experience in implementing and partially optimizing a complex but highly parallelizable application on the Xeon-Phi processor, one of the earliest implementations of the *Many integrated*

*Corresponding author. Tel.: +39-053-297-4614 ; fax: +39-053-297-4614.
E-mail address: schifano@fe.infn.it.

Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science
doi:10.1016/j.procs.2013.05.219

Cores (MIC) architecture designed by Intel. A MIC processor integrates several tens of CPU cores; each core is a lightweight X86-compatible version of the Pentium architecture not including, for example, dynamic scheduling of instructions.

The first commercially available processor in this class is the *Knights Corner* (KNC), that should be released at the beginning of 2013. So far Intel has announced two versions of this processor, with 60 or 61 cores. Each core performs SIMD instructions on rather long data-vectors; this feature helps improve performance at the core level, but relies critically on the ability of programmers and compilers to use efficiently these instructions.

In this paper we focus on an early implementation, optimization and test of a production-grade Lattice Boltzmann (LB) code, that we have performed on pre-production samples of the KNC. We describe the structure of our program, that tries to map the available parallelism of the algorithm on the parallel features of the processor. To put our results in perspective, we compare with earlier implementations that we have optimized for not simpler many-core architectures and for GPUs. We have relied on the specific features of our algorithm to map a large fraction of the available parallelism onto the highly parallel data paths available on the processor; for this reason our approach does not easily relates to other codes relevant for the computational sciences; however, what we offer here is an early assessment of the potential for performance of a new class of High Performance Computing (HPC) processing architectures.

Our paper is structured as follows: after the present overview section, we describe the architecture of the MIC processor, discussing its potential for performance in HPC scientific codes; we then shortly describe the Lattice Boltzmann algorithm that we use as our test-bed and then present the structure of our MIC-based implementation. In a final section, we collect our performance results and our comparisons with other state-of-the-art architectures and present our concluding remarks.

2. The Intel Xeon-Phi co-processor

The Xeon-Phi co-processor is an accelerator for a more traditional host system, in much the same way as GPUs are used in conjunction with host CPUs. It is available as a standard Pci-express add-on card for traditional PCs. Data transfers from host to processor and vice-versa occur over 16 Pci-express (Gen. 2) data lanes.

The first production version of this system, expected to be available for the mass-market at the end of January 2013, has one *Knights Corner* (KNC) MIC processor and 8 GBytes of GDDR5 RAM. The KNC processor integrates up to 61 CPU cores, and runs at a frequency of ≈1 GHz. It connects to its private memory bank through 16 memory channels, delivering a theoretical peak bandwidth of ≈320 GByte/s. The theoretical peak bandwidth to the host-processor is ≈ 8 GByte/s.

Each core is based on the Pentium architecture, and includes 32 KB of L1 cache used for data and instructions, 512 KB of L2 data-cache, and a 512-bit wide vector Floating-Point Unit (FPU). The FPU engine performs one *fused-multiply-add* instruction per clock cycle, delivering a peak performance of ≈32 GFlops in single-precision, and ≈16 GFlops in double-precision, if all elements of the data-vector can be used at all clock cycles. In this case, the whole KNC is able to deliver a peak performance of around 2 and 1 Tflops respectively in single and double precision, respectively. Within the processor, the cores are connected through a high-speed bi-directional ring, and data items inside all L2-caches are shared by all cores. The KNC runs a modified lightweight version of the Linux operating system; each core supports the execution of up-to 4 Linux threads.

From a programming point of view, Phi systems can be programmed using two different approaches. The first approach – called *native* – consists in running programs directly on the KNC processor. In this case, the user compiles natively the program and logs onto the Linux system running on the KNC to execute it. The Intel compiler (the only available one, so far, supporting the KNC architecture) produces KNC native code if one enables the `-mmic` flag. A second approach – usually referred to as *accelerator* or *offloading* – consists in developing a hybrid program which runs on the host and on the KNC. Using the *offload* approach a program starts on the host and the compiler, instructed by appropriate `pragmas` directives, generates code that transparently transfers control to the MIC processor. Figure 1 shows a simple example of a program which, after initializing arrays `A, B, C` on the host, offloads the execution of the `vadd()` function onto the MIC accelerator. The code of `vadd` will be compiled both for the host and the MIC. In the case described on the figure, the code running on the MIC will execute the sum of vectors A and B, while the code running on the host will print a warning message.

```
void __attribute__((target(mic))) vadd ( double *A , double *B , double *C )

int main () {
  double A[N], B[N], C[N];
  vinit(double *A, double *B, double *C);
  #pragma offload target(mic:0) in(A,B:lenght(N)) inout(C:lenght(N))
  {
    vadd (A,B,C);
  }
  vprint(C);
}

void vadd (double *A, double *B, double *C) {
#ifdef __MIC__
  int i;
  for (i=0; i<N; i++)
    C[i] = A[i] + B[i];
#else
  fprint(stderr, "This code is running on the host\n");
#endif
}
```

Figure 1. Example of a MIC program using the offload approach: the execution of the `v_add` function is offloaded to the accelerator; a transfer of data from the processor to the accelerator and of results in the opposite direction is implied by the `in` and `inout` clauses.

The host version of `vadd` will be executed if the run-time system is not able to access the MIC card. The offloaded function can eventually spawn several threads to run on all available cores. The offload language available on the MIC software environment is integrated with several existing programming languages (C++, Fortran, Cilk, TBB, ...), and models (openMP and OpenCL [1, 2]).

In quest for performance, programs should offload kernel functions which are computationally expensive if executed on the host. On the other hand, the performance advantages of offloading a computationally intensive function to the co-processor must be confronted with the time necessary to move input and output data to and from the host using the PCI-express data link.

Conversely, if one chooses the *native* approach code can be compiled and run without any changes; however one encounters problems associated to the relatively small size of the memory banks directly connected to the co-processor while at the same time input-output operations can be more cumbersome. Moreover, the serial part of the code, like initialization of data structures, may have very poor performance on the MIC.

In both cases, performances heavily rely on the ability of programmers, compilers and run-time support to carefully exploit parallelism on *all* available hardware features. Optimizations areas relevant for performances are:

- core parallelism: the code running on the MIC processor should allow all cores to work in parallel exploiting MIMD or SPMD multi-task parallelism; in the first case the application is decomposed in several sub-tasks and each one is executed by a different core; in the latter case, that can be also combined with the previous one, the data-set is typically partitioned among the cores, and each core executes the same task on different portions of the data-set
- vector programming: each core processes the data-set of the application using vector instructions and exploiting streaming-parallelism (SIMD); the number of data-items that can be processed by any vector instruction is up to 16 for single-precision arithmetics and up to 8 for double-precision arithmetics. This approach is efficient for performance if a matching number of data words on which the same operation must be performed can be identified in the code.

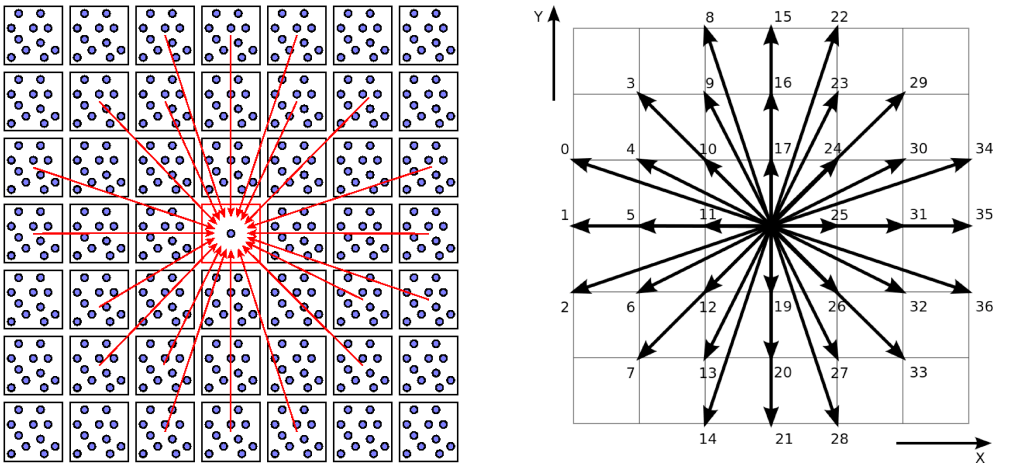

Figure 2. Velocity vectors for the LB populations in the D2Q37 model. Population labelling is arbitrary; populations associated to each site are stored at consecutive memory address in the order defined by their labels.

3. Lattice Boltzmann methods

The Lattice Boltzmann method is a computational approach that describes fluid dynamics by lattice discretization in position and momentum space [3]. On a discrete lattice (in $D = 2$ or 3 dimensions), the actual dynamics is replaced by a synthetic one, based on a set of lattice populations ($f_l(x,t)$); each population has a given fixed lattice velocity $\boldsymbol{c}_l$, associated to the fact that, at each time step, it drifts towards a nearby lattice site; populations evolve in (discrete) time according to the following equation:

$$f_l(\boldsymbol{x} + \boldsymbol{c}_l\Delta t, t + \Delta t) - f_l(\boldsymbol{x}, t) = -\frac{\Delta t}{\tau}\left(f_l(\boldsymbol{x}, t) - f_l^{(eq)}\right) \tag{1}$$

Macroscopic observables (density ρ, velocity $\boldsymbol{u}$ and temperature T) are defined in terms of the $f_l(x,t)$:

$$\rho = \sum_l f_l, \tag{2}$$

$$\rho\boldsymbol{u} = \sum_l \boldsymbol{c}_l f_l, \tag{3}$$

$$D\rho T = \sum_l |\boldsymbol{c}_l - \boldsymbol{u}|^2 f_l, \tag{4}$$

and the equilibrium distributions ($f_l^{(eq)}$) are themselves function of these macroscopic quantities [3].

LB models are available in many different versions, characterized by the dimension of the lattice, the number of populations that they use (they are usually labelled as DnQm, where n is the number of dimensions and m is the number of populations) and by the explicit functional form of the equilibrium distributions $f_l^{(eq)}$. Starting from early models (e.g., D2Q9 or D3Q19), able to describe an uncompressible fluids at constant temperature, recent LB algorithms describe the thermo-hydrodynamics of multiple-phases or correctly treat compressible fluids with realistic equations of state.

In the following we focus on a recently proposed D2Q37 model that correctly reproduces the equation of state of the fluid, regarded as a perfect gas ($p = \rho T$); see [4, 5] for full details. For this model, one shows that, after appropriate shift and re-normalization of the velocity and temperature fields, one recovers, via a Taylor expansion in Δt, the following set of thermo-hydrodynamical equations:

$$D_t\rho = -\rho\partial_i u_i^{(H)} \tag{5}$$

$$\rho D_t u_i^{(H)} = -\partial_i p - \rho g\delta_{i,2} + \nu\partial_{jj}u_i^{(H)} \tag{6}$$

$$\rho c_v D_t T^{(H)} + p\partial_i u_i^{(H)} = k\partial_{ii}T^{(H)}; \tag{7}$$

where superscripts H flag renormalized (physical) quantities, $D_t = \partial_t + u_j^{(H)}\partial_j$ is the material derivative and we neglect viscous heating; c_v is the specific heat at constant volume for an ideal gas, $p = \rho T^{(H)}$, and ν and k are the transport coefficients; g is the acceleration of gravity. In this model, population velocities are associated to moves on the lattice that reach points up to three lattice points away, see figure 2.

From the point of view of the present paper, LB algorithms are a nice test case, as they offer a huge degree of available parallelism, that can be immediately and easily identified: defining $\boldsymbol{y} = \boldsymbol{x} + \boldsymbol{c}_l\Delta t$, one rewrites the main evolution equation as:

$$f_l(\boldsymbol{y}, t + \Delta t) = f_l(\boldsymbol{y} - \boldsymbol{c}_l\Delta t, t) - \frac{\Delta t}{\tau}\left(f_l(\boldsymbol{y} - \boldsymbol{c}_l\Delta t, t) - f_l^{(eq)}\right) \tag{8}$$

One easily identifies the overall structure of the computation that evolves the system by one time step Δt; for each point $\boldsymbol{y}$ in the discrete grid one:

1. gather from neighboring sites the values of the fields f_l corresponding to populations that drift towards $\boldsymbol{y}$ with velocity $\boldsymbol{c}_l$ and then
2. perform all mathematical processing needed to compute (*in a completely local fashion*) the quantities appearing in the equation above. This step is slightly more complex if one wants to take into account reactive effects (combustion); indeed, in that case the divergence of the velocity field has to be explicitly computed. This means that a further gather operation must be performed midway in this (otherwise local) compute intensive step.

The key remark is that both steps above are completely uncorrelated for different points of the grid, so they can be parallelized according to any convenient schedule, as long as one makes sure that, for each and all grid points, step 1 is performed before step 2. In principle, the available parallelism is as large as the number of sites of the lattice (that easily grows to hundreds of millions of sites).

4. D2Q37 LBM implementation

A reference implementation of the LB algorithm repeatedly evolves the lattice-cells of the system for one time steps. For each point in the grid, data needed to compute the new value of each population at each site in the grid is gathered from nearby sites, and then a fully local processing step is performed. Here we describe the implementation of our LB algorithm, ported and optimized for the Xeon-Phi architecture. As already discussed, our D2Q37 model operates on a bi-dimensional lattice, and 37 populations are associated to each lattice site. The algorithm processes every grid-point by applying in order two critical computational kernels:

- *propagate()*: this phase of the computations gathers for each site 37 populations from neighbor lattice sites, according to the scheme of figure 2. This process does not perform floating-point computation; rather it performs memory copies accessing sparse memory addresses. It collects at each site the 37 populations that will interact at the next step during the execution of the *collide()* kernel. This step requires that each site accesses the populations of neighbor cells at distance up to 3 in the physical grid.
- *collide()*: this phase performs all the mathematical steps associated to equation 8 and needed to compute the population values at each lattice site at the new time step (this is called *collision*, in LB jargon). Input data for this phase are the populations gathered by *propagate()*. This step is the most floating point intensive section of the code; it uses only the population members of the site on which it operates, making the processing of different sites fully uncorrelated.

In our implementation the lattice is stored in column-major order, and we keep in memory two copies of it. Even if this solution allocates more memory than having a single copy it makes the implementation of the code much easier because each computational phase read inputs from one copy and writes results to the other; moreover, for the lattice sizes that we have in mind, memory size is not a critical resource on currently available machines. The lattice is stored in memory as an *Array of Structure* (AoS); we select this mapping as it is more efficient for the *collide* step, that is the most expensive part of the algorithm in our model. Indeed, in this case, populations associated to a given site are stored at contiguous memory address, and this improves data-locality and better exploits cache re-use, see for example [6].

```
struct {
  __m512d vp0;
  __m512d vp1;
  __m512d vp2;
  ...
  __m512d vp36;
} vpop_t;

vpop_t lattice[LX][LY];
```

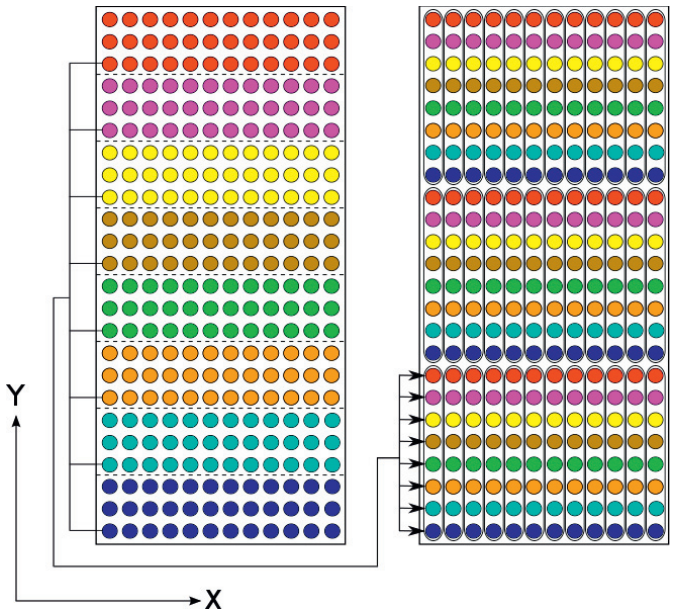

Figure 3. Left: data type definition used for the structure of vectors containing population data. Right: Data allocation among the threads and within the vector structures operated upon by each thread.

Our LB code is organized using the *offload* programming approach. The code segment running on the host initializes the lattice, downloads the lattice on the memory of the MIC, and starts the execution of a kernel on the MIC processor. When execution of the offloaded function is finished, it copies back the results on the memory on the host. In practice, a very large fraction of the code, including all compute-intensive sections is offloaded to the co-processor. Control goes back to the host only after several time-step iterations, when data has to be written to disk; the impact on performance is therefore negligible. This favorable state-of-affairs will have to be reconsidered when one tries to map the code on a parallel machine of many MIC systems, and data has to be moved to/from different processing nodes.

The code running on the MIC processor is organized as a multi-thread program; each thread processes a partition of the lattice. A lattice of size $L_x \times L_y$ is split on N_t threads along the X dimension. Each thread then processes a *sub-lattice* of size $(L_x/N_t) \times L_y$.

Within each thread, K sites are processed in parallel, in order to exploit the data-parallelism made available by vector instructions. In our case K is exactly the number of data words that can be packed into a 512-bit AVX vector; using double-precision, we have $K = 8$, while we set $K = 16$ if single-precision are used.

Streaming vector instructions can be automatically inserted by the compiler, or explicitly used by the programmer, by coding *intrinsic* functions. In the first case the program is a scalar code (all variables are scalar), and it is compiled enabling an appropriate *auto-vectorization* flag (e.g. `-O2` or `-O3` for the ICC compiler). The compiler automatically exploits data-parallelism enabling the use of streaming instructions if specific conditions are met. For example, if no data dependencies occur between iterations of a loop, the compiler can (partially) unroll it, and two or more iterations (according to the type of variables involved for which no dependencies occur), can be processed in parallel using vector instructions. This approach is a simple and fast option for the programmer, but efficiency can be limited by the ability of the compiler to identify all parts of the code on which vectorization can be applied.

A more cumbersome but potentially more efficient approach explicitly introduces vector variables and processes them by so called *intrinsic* functions which are mapped by the compiler directly onto the corresponding assembly instruction. For example a double precision sum on a vector of 8 elements can be performed by the code line `d = _mm512_fmadd_pd (a, b,c)` declaring `a, b, c, d` as vector variables of type `__mm512d`. In this case each variable holds 8 double-precision floating point numbers, and the intrinsic is directly mapped onto the `VFMADD132PD` assembly instruction.

In the implementation that we describe here, we have used the approach of using vector programming and intrinsic functions, based on our previous experience [7] on simpler vector machines for which auto-vectorization yielded sub-optimal performances; future work will compare both approaches also for this class of machines. We have divided the lattice in K strips along the Y dimension, and we have packed together populations of sites at distance L_Y/K as shown in figure 3. Using this approach our lattice is stored as an array of vector-sites, and each vector-site is stored as an array of 37 AVX vectors, each holding K populations.

5. Results and conclusions

In this section we report our preliminary performance results. We have tested the execution of the *propagate* and *collide* as two separate kernels; however production codes can merge the execution of the two kernels in a single step, applied in sequence to all cells of the lattice, saving access to the memory and improving performances. This optimization step is well known in literature, see for example [11]. We have run our kernels on two different versions of the MIC processor:

- we initially developed and debugged our codes on a prototype system (so called, step D0, software stack release Alpha 10) using a pre-production MIC chip, the 30-core *Knights Ferry* (KNF) processor, running at a frequency of 1.050 GHz, and a memory bank of just 2 GB. This earlier version of the MIC processor supports at full speed single-precision only; floating point single-precision peak performance is 1.075 GFlops, while double-precision runs eight times slower; memory interface bandwidth is limited to $\approx$ 80 GB/s. This system has been made available as a platform for code development only and very poor performance is to be expected.
- we then moved our program to a pre-production version of the Xeon-Phi (so called, step B0 software stack release 2.1.4346-16) using a 61-core *Knights Corner* (KNC) processor running at a frequency of 1.09 GHz. As already discussed, peak performances are $\approx$ 2 Tflops in single precision and $\approx$ 1 TFlops in double precision. The GDDR5 memory interface has a peak bandwidth of 320 GB/s.

In figure 4 we collect our performance results for the *propagate* and *collide* kernels, running on both versions of the MIC systems.

Let us focus separately on each of these kernels. For *propagate*, the following remarks are in order:

- the *propagate* kernel executes memory-to-memory copies with a rather sparse addressing pattern; for this reason performances are strongly correlated to the performance of the processor in computing addresses and performing memory accesses on short data sequences.
- On the KNF system, only up to 30 threads could be started; in this range the sustained bandwidth grows almost linearly with the number of threads, and reaches a peak of 8.5 GByte/s.
- The KNC processor is able to run a much larger number of threads; Intel suggest that using up-to 4 threads per core may be useful to increase performance. In our tests, we have used thread-affinity to define the map between threads and physical-CPUs; we do not use CPU-zero which is used by the operating system; for this reasons the number of available CPU-cores is 60. As we see from the plots, also in this case the sustained bandwidth scales almost linearly up to 120 threads, using one and two threads per core; running 3 or 4 threads improves performance marginally while using more than 4 threads results in bandwidth degradation. The highest sustained bandwidth is $\approx$ 51 GByte/s, obtained running 4 threads per core for a total of 240 threads; the corresponding efficiency is around 16% of peak.
- The efficiency figure cited above means that our benchmark is currently using only a small fraction of the available peak memory bandwidth. We believe that our results are limited by the heavy and complex set of memory addressing arithmetic involved in the kernel. To further understand this point we have measured the bandwidth for a memory-copy benchmark, that moves large sequences of contiguous data words. In this case, we expect that addressing has a minor impact on performance. This benchmark reaches a sustained bandwidth of approximately 135 GByte/s using 120 threads, that is approximately 42% of the peak. For a larger number of threads the bandwidth remains essentially constant. We see however that the bandwidth available to each thread quickly decreases – for reasons as yet unclear to us – as more and more threads compete to access memory;

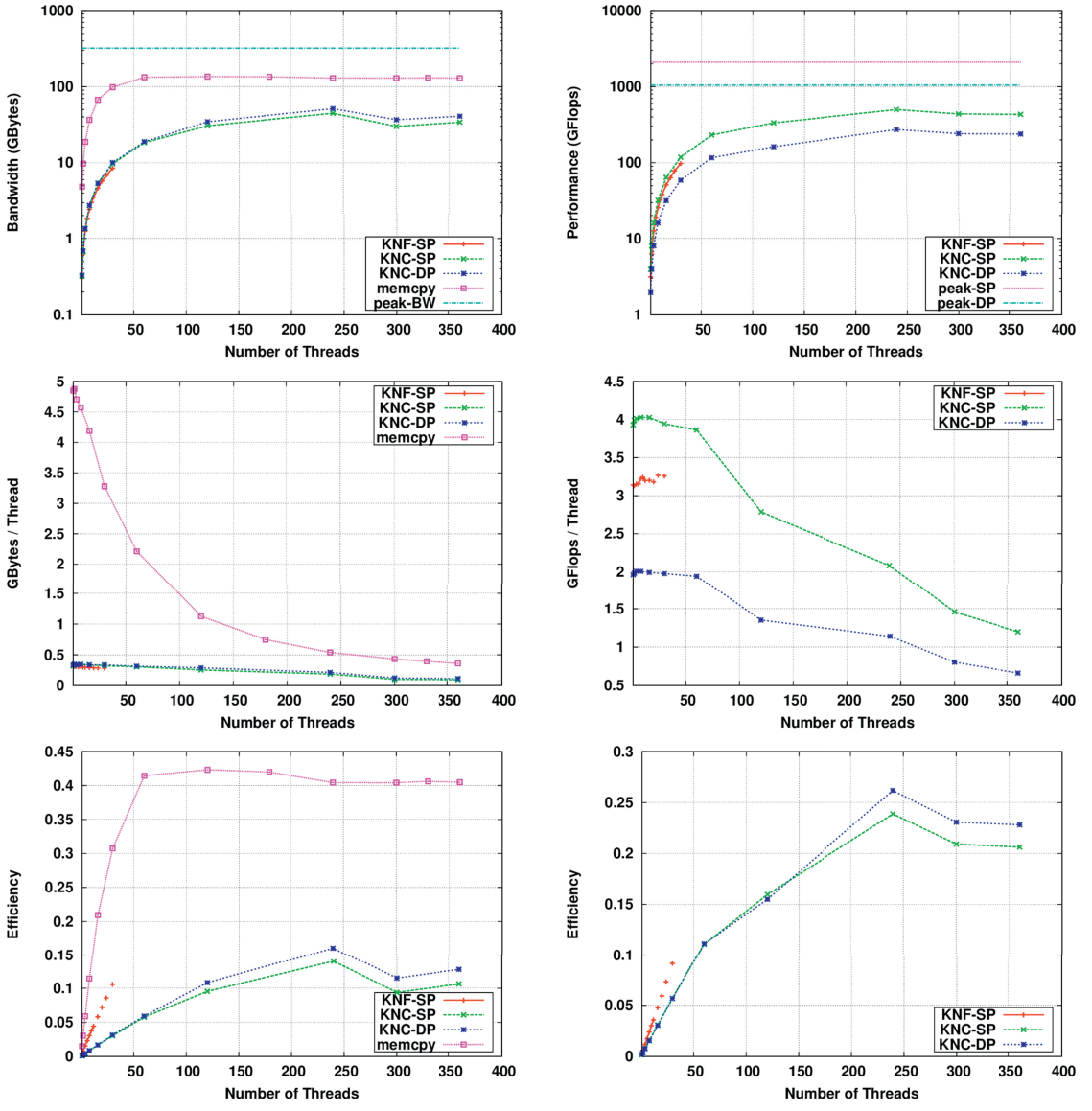

Figure 4. A summary of our benchmark results of the *propagate* (left) and *collide* (right) kernels. Starting from the top we plot the sustained performance (in GByte/s or GFlops), the performance per thread and the efficiency; all figures are a function of the number of threads on which the kernel has been mapped. For comparison we also plot the corresponding results for a mem-copy benchmark. Data is for both the KNF and KNC processors and for single precision (-SP) and double precision (-DP) arithmetics.

	C2050	2-WS	2-SB	KNC
propagate (GB/s)	84	17.5	60	52
$\mathcal{E}$	58%	29%	70%	16%
collide (GF/s)	205	88	220	274
$\mathcal{E}$	41%	55%	63%	27%
ξ (collide)	NA	1.19	1.27	0.52

Table 1. Performance comparison for *propagate* and *collide* among several architectures. C2050 is a NVIDIA Fermi GPU, 2-WS is a double-socket Westmere system, 2-SB is a double socket Sandybridge platform, and KNC is the Xeon-Phi board. For each kernel and each system we show sustained performance, efficiency and the ξ parameter (defined in the text), if applicable.

- Summing up, there seems to be problem in obtaining a large fraction of the theoretically available bandwidth, and allocating it efficiently to the cores. In our code this problem is largely hidden by the overhead associated to computing the addresses.

Contrary to *propagate*, the *collide* kernel is the most compute intensive part of our code. We make the following remarks:

- On the KNF processor, the sustained performance of our benchmark using 30 threads is $\approx$ 98 GFlops, corresponding to $\approx$ 10% of the peak, and performance grows linearly (or even super-linearly) with the number of threads. Given the repeated warnings that the KNF processor has not been designed for performance we do not further elaborate on these figures.
- On the KNC system the *collide* kernel scales almost linearly up-to 60 threads; going to 120 and 240 threads brings further sub-linear performance improvements, up to an overall sustained performance of 274 GFlops using 240 threads, corresponding to $\approx$ 27.5% of the peak in double-precision. Sustained performance in single precision follows the same pattern, with a top value $\approx$ 500 GFlops, corresponding to $\approx$ 24% of peak.
- As one uses more than 240 threads performance starts to drop in absolute value. We believe that this is due to some efficiency problem within the processor when switching from one to another thread on the *same* core. We base this belief on the observation that the same pattern of performance per thread occurs in single and double precision, so we can rule out that the performance drop is associated to insufficient memory bandwidth to support the processing rate.
- One of the reasons for the measured level of performance may be associated to the fact that our computational kernel implies a mix of mathematical operations that cannot be always cast into the multiply-add structure, so a significant loss of performance is expected. We are currently trying to identify other (if any) performance bottlenecks.

In order to put our results in perspective, table 1 provides a comparison of performance figures for the same algorithm, running on two more traditional many-core systems and on a state-of-the-art GPU.

We discuss performances for the same kernel codes measured on a 12-core system using two *Westmere* processors [8, 9] (with a 128-bit vector units), a 16-core system, based on the *Sandybridge* processor (with 256-bit vector units), and a NVIDIA C2050 card based on the Fermi processor [7, 10]. Inspecting this table, one might conclude that the KNC processor delivers more performance for this LB algorithm that all other implementation. Still, one has to remember that the actual efficiency is not very high, a situation that has to be better understood.

In an attempt to provide a fair comparison of performances across architectures with widely different number of cores and vector sizes, we define the ξ metric:

$$\xi = \frac{P}{N_c \times v \times f}$$

where P is the measured performance of a kernel code, N_c is the number of cores on which the kernel has been parallelized, v is the size of vector instruction used, and f is the operating frequency of the processor. The ξ parameter should allow to compare how well a given architecture is able to use all its parallel features, as well as its sheer clock speed, to deliver performance to a given application. We see that the more traditional Intel processors

have very similar figures for ξ while the MIC system has a significantly lower value. It will be interesting to see if more clever programming and optimization strategies may improve on these figures; work is in progress in this direction.

Acknowledgements

This work was performed in the framework of the COKA and Suma projects, supported by Istituto Nazionale di Fisica Nucleare (INFN). We would like to thank CINECA (Bologna, Italy) and the Jülich Supercomputer Center (Jülich, Germany) for allowing us to use their computing systems.

References

[1] Opencl, `http://www.khronos.org/opencl`.
[2] The openmp api, `http://www.openmp.org`.
[3] S. Succi, *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond*, Oxford University Press (2001).
[4] M. Sbragaglia et al., *Lattice Boltzmann method with self-consistent thermo-hydrodynamic equilibria*, J. Fluid Mech., **628** (2009) 299.
[5] A. Scagliarini et al., *Lattice Boltzmann methods for thermal flows: Continuum limit and applications to compressible Rayleigh-Taylor systems*, Phys. Fluids, **22** (2010) 055101.
[6] G. Wellein, et al., *On the Single Processor Performance of Simple Lattice Boltzmann Kernels* Computers & Fluids, **35** (2006) 910.
[7] L. Biferale et al., *An optimized D2Q37 Lattice Boltzmann code on GP-GPUs*, Computers and Fluids. (2012). Article in Press.
[8] L. Biferale et al., *Optimization of Multi-Phase Compressible Lattice Boltzmann Codes on Massively Parallel Multi-Core Systems*, Procedia Computer Science, **4** (2011) 994:1003.
[9] L. Biferale et al., *A multi-GPU implementation of a D2Q37 Lattice Boltzmann Code.* R. Wyrzykowski et al. (Eds.): PPAM 2011, Part I, LNCS 7203, pp. 640-650, Springer, Heidelberg (2012).
[10] A. Bertazzo et al., *Implementation and Optimization of a Thermal Lattice Boltzmann Algorithm on a multi-GPU cluster*, Proceedings of Innovative Parallel Computing (INPAR) 2012, May 13-14, 2012 San Jose, CA (USA).
[11] T. Pohl, et al., *Optimization and Profiling of the Cache Performance of Parallel Lattice Boltzmann Codes*, Parallel Processing Letters, 13(4) (2003) 549.