VI International Conference on Computational Methods in Marine Engineering
MARINE 2015
F. Salvatore, R. Broglia and R. Muscari (Eds)

USING ACCELERATORS TO SPEED UP SCIENTIFIC AND ENGINEERING CODES: PERSPECTIVES AND PROBLEMS

E. CALORE*, S.F. SCHIFANO*, R. TRIPICCIONE*

*Università di Ferrara and INFN - Sezione di Ferrara
via Saragat 1, 44122 Ferrara (ITALY)
email: calore,schifano,tripiccione@fe.infn.it

Key words: Computational Fluid-dynamics, Accelerator computing, Many-core architectures

Abstract. Accelerators are quickly emerging as the leading technology to further boost computing performances; their main feature is a massively parallel on-chip architecture. NVIDIA and AMD GPUs and the Intel Xeon-Phi are examples of accelerators available today. Accelerators are power-efficient and deliver up to one order of magnitude more peak performance than traditional CPUs. However, existing codes for traditional CPUs require substantial changes to run efficiently on accelerators, including rewriting with specific programming languages.

In this contribution we present our experience in porting large codes to NVIDIA GPU and Intel Xeon-Phi accelerators. Our reference application is a CFD code based on the Lattice Boltzmann (LB) method. The regular structure of LB algorithms makes them suitable for processor architectures with a large degree of parallelism. However, the challenge of exploiting a large fraction of the theoretically available performance is not easy to met. We consider a state-of-the-art two-dimensional LB model based on 37 populations (a D2Q37 model), that accurately reproduces the thermo-hydrodynamics of a 2D-fluid obeying the equation-of-state of a perfect gas.

We describe in details how we implement and optimize our LB code for Xeon-Phi and GPUs, and then analyze performances on single- and multi-accelerator systems. We finally compare results with those available on recent traditional multi-core CPUs.

1 INTRODUCTION

The last decade has seen a significant change in the way in which computers deliver their computational performance. Today, clock frequencies have essentially reached a plateau, while processor architectures are becoming more and more parallel: in other words, processors are able to perform more and more tasks in a fixed amount of time rather than completing the same task in a shorter time span. This trend is due to basic physics limitations in the behavior of the electronic gates that make up computer systems, so it is going to stay in the foreseeable future.

An undesirable consequence is that moving a code from a given processor to a different one, which has a larger degree of parallelism and a larger peak performance, in most cases brings negligible gains: indeed, extracting a large fraction of the peak processor performance requires that the program use efficiently almost all available computing resources. Improved compilation techniques, able to identify parts of the original code that can execute in parallel and mapping them onto the processor, have tried to mitigate the problems. This approach has had limited success and is quickly becoming hopeless as recent high performance processors (usually referred to as *accelerators*) contains $\mathcal{O}(100)$ cores and are able to perform thousands of independent operations per clock cycle. In the (hopefully) near future, improved programming languages (e.g., OpenACC or OpenMP4), allowing programmers to explicitly identify the parallelism available in their codes, should become a viable and solid alternative; it is then up to the corresponding compilers to exploit available parallelism as widely as allowed by the target computer, granting – it is hoped – a fair balance of portability and efficiency. For the time being, writing a large production-grade, massively parallel, efficient scientific or engineering code is still a non trivial tasks, requiring, at some stage, to use proprietary languages (e.g., CUDA) specific for a given family of processors (NVIDIA GPUs, for the CUDA case).

In this work we share our experience in adapting and optimizing for accelerators a state-of-the-art Lattice Boltzmann code. Over the years, LB codes have been written and optimized for large clusters of commodity CPUs [1], for application-specific machines [2, 3] and even for FPGAs [4]. More recently work has focused on exploiting the parallelism of powerful traditional many-core processors [5], and of power-efficient accelerators such as GPUs [6, 7] and Xeon-Phi processors [8]. This paper focuses on a specific use case, but we think that our experience is a fair account of the problems (and of the strategies to overcome them) that arise as scientific and engineering codes are adapted and optimized for recent and future HPC architectures, so it can be interesting to a wider readership.

This paper is structured as follows: section 2 provides a short introduction to accelerator architectures and section 3 presents an overview of the Lattice Boltzmann approach to computational fluid-dynamics. Section 4 describes the optimization strategies used to implement our LB codes for different accelerators, and section 5 presents our results, assesses the performance gain that are to be expected by the use of accelerators and summarizes – as our concluding remarks – the balance between performance and portability.

2 ACCELERATOR ARCHITECTURES

In this section we sketch the main features of the accelerators we have used – the Intel Xeon-Phi and the NVIDIA K80 GPU – representative examples of the prevailing accelerator architectures. Both systems come as add-on boards, connected to the main processor by a PCIe interface (typical bandwidth is 8 GB/s) representing a significant bottleneck in the data exchange between CPU and accelerator.

The Xeon-Phi uses the *Knights Corner* (KNC) processor based on the *Many Integrated Core* (MIC) architecture, an extreme evolution of the multi-core approach, containing 61 cores interconnected by a high-speed ring. Each core has the Pentium architecture with 32 KB of L1-cache for data and instructions and 512 KB L2 data-cache; it also includes a vector Floating Point Unit (FPU), performing at each clock cycle the same operation on 16 (8) sets of operands in single (double) precision. Running at $\approx$1 GHz, the peak performance is $\approx$2 (1) Tflops in single (double) precision. A GDDR5 memory bank of 8 GB connects to the processor with a bandwidth of $\approx 320GB/s$, while all cores share their L2 caches. On these accelerators, performance dictates that applications are split in many concurrent threads (*thread* parallelism, mapped onto the cores) and, *at the same time*, on each thread the same operations are applied to many data items (*vector* parallelism). For more details see the Intel online documentation on the KNC architecture.

The K80 accelerator has two *Kepler* GPUs and 24 GB of GDDR5 memory. GPU operation is best abstracted in term of GPU-threads, each thread typically processing just one element of the program data set. Threads are executed by several *Streaming Multiprocessors* (SMXs). Each SMX handles up to 2048 active threads and has 192 scalar cores to process them. At each clock cycle groups of 32 threads executing the same instruction (so called *warps*) are active within each SMX. Large banks of registers within each SMX contain temporary data, while data transfer with the memory occurs at $\approx$ 250 GB/s. Peak performance is 4 (1) Tflops in single (double) precision. The thread-based approach to GPU programming is probably more elegant, since one has to consider just one level of parallelization at the conceptual level; in practice however the actual scheduling of threads to warps has several architectural limitations, so performance is strongly program dependent. For more details on Kepler architecture see the NVIDIA online whitepaper *"Kepler GK110"*.

3 LATTICE BOLTZMANN METHODS

Lattice Boltzmann methods (LB) are widely used in computational fluid dynamics, to describe flows in two and three dimensions. LB methods (see, e.g., [9] for an introduction) are discrete in position and momentum spaces; they are based on the synthetic dynamics of *populations* sitting at the sites of a discrete lattice. At each time step, populations hop from lattice-site to lattice-site and then incoming populations *collide*, that is, they mix and their values change accordingly.

Over the years many different LB models have appeared, solving flows in 2 and 3 dimensions with different degrees of accuracy and describing different situations such as multi-phase or complex flows; see [10] for a recent review. LB models in n dimensions with y populations are labeled as $DnQy$; here, we consider a state-of-the-art $D2Q37$ model that reproduces the thermo-hydrodynamical equations of motion of a fluid in two dimensions and enforces the equation of state of a perfect gas ($p = \rho T$) [11, 12]; this model has been extensively used for large scale simulations of convective turbulence (see e.g., [13, 14]).

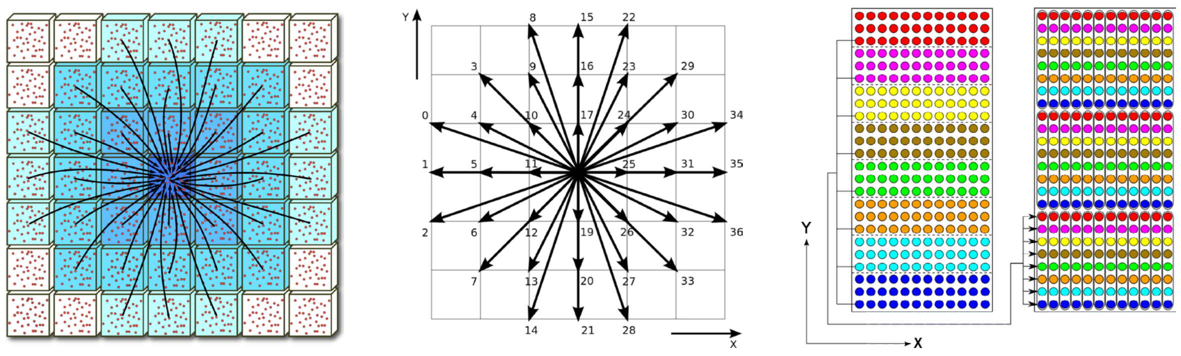

Figure 1: Left: LB populations in the D2Q37 model, hopping to nearby sites during the `propagate` phase. Center: populations f_l are identified by an arbitrary label; populations are stored in memory according to several possible layouts. Right: Data packing within AVX vectors of lattice data for the Xeon-Phi implementation.

From a computational point of view this very accurate scheme is more complex than simpler LB models, at the cost of more severe requirements for storage (each lattice points has 37 populations), memory bandwidth and floating-point throughput (each time step uses $\approx$ 7600 double-precision floating point operations per lattice point).

Populations ($f_l(x,t)$ $l = 1 \cdots 37$) are defined at the sites of a discrete and regular 2-D lattice; each $f_l(x,t)$ has a given lattice velocity $\boldsymbol{c}_l$; populations evolve in (discrete) time according to the following equation:

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

Macroscopic quantities, density ρ, velocity $\boldsymbol{u}$ and temperature T are defined in terms of the $f_l(x,t)$ and of the $\boldsymbol{c}_l$s (D is the number of space dimensions):

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

the equilibrium distributions ($f_l^{(eq)}$) are known function of these macroscopic quantities [9], and τ is a suitably chosen relaxation time. In words, Eq. 1 stipulates that populations drift from lattice site to lattice site according to the value of their velocities (*propagation*) and, on arrival at point y, they interact among one another and their values change accordingly (*collision*). One can show that, in suitable limiting cases and after appropriate renormalizations are applied, the evolution of the macroscopic variables defined in Eq. 2 obeys the thermo-hydrodynamical equations of motion of the fluid.

An LB code takes an initial assignment of the populations, in accordance with a given initial condition at $t = 0$ on some spatial domain, and iterates Eq. 1 for all points in the domain and for as many time-steps as needed; boundary-conditions at the edges of the integration domain are enforced at each time-step by appropriately modifying population values at and close to the boundaries.

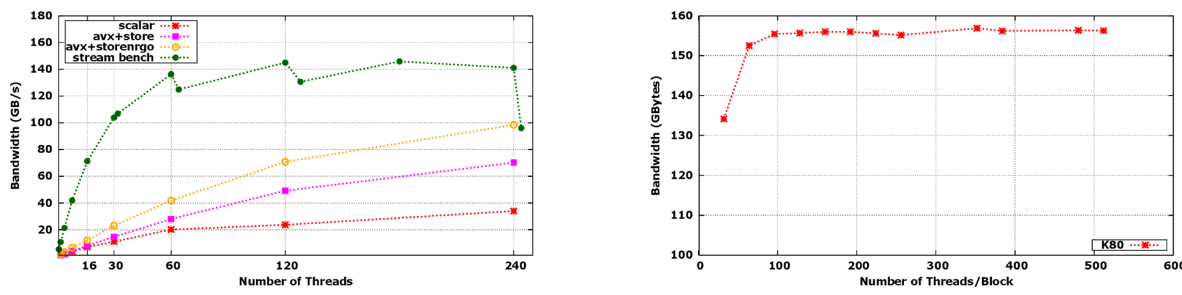

Figure 2: Performance of the `propagate` kernel on the Xeon-Phi (left) and on one GPU of the K80 board (right). For the Xeon-Phi, we show for comparison results of the STREAM memory benchmark.

The LB approach has a huge degree of easily identified parallelism. Indeed, Eq. 1 shows that the *propagation* step amounts to gathering the values of the fields f_l from neighboring sites, corresponding to populations drifting towards $\boldsymbol{y}$ with velocity $\boldsymbol{c}_l$; the following step (*collision*) then performs all mathematical processing needed to compute the quantities in the r.h.s. of Eq. 1, for each point in the grid. One sees immediately from Eq. 1, that both steps above are fully uncorrelated for different points of the grid, so they can be executed in parallel according to any suitable schedule.

In practice, an LB code executes a loop over time steps; each iteration foresees three kernels: `propagate`, `bc` and `collide`.

`propagate` moves populations across lattice sites according to the pattern of Figure1, collecting at each site all populations that will interact at the next phase (`collide`). In our model populations move up to three lattice sites per time step. Computer-wise, `propagate` moves blocks of memory locations allocated at sparse addresses, corresponding to populations of neighbor cells.

`bc` executes *after* propagation and adjusts populations at the edges of the lattice, enforcing appropriate boundary conditions (e.g., constant temperature and zero velocity at the top and bottom edges of the lattice). For the left and right edges, we usually apply periodic boundary conditions. This is conveniently done by adding *halo* columns at the edges of the lattice, where we copy the rightmost and leftmost columns (3 in our case) of the lattice before starting the `propagate` step. After this is done, points close to the boundaries are processed as those in the bulk.

`collide` performs all mathematical steps needed to compute the population values at each lattice site at the new time step, as per Eq. 1. Input data for this phase are the populations gathered by the previous `propagate` phase. This step is the most floating point intensive part of the code.

4 IMPLEMENTATION OF THE LB CODE

In this section we describe the implementation and optimization of our LB code for *Xeon-Phi* and *K80* accelerators. In both cases we adopt the *offload* approach: the host first uploads lattice data onto the accelerator memory and then performs a loop over time

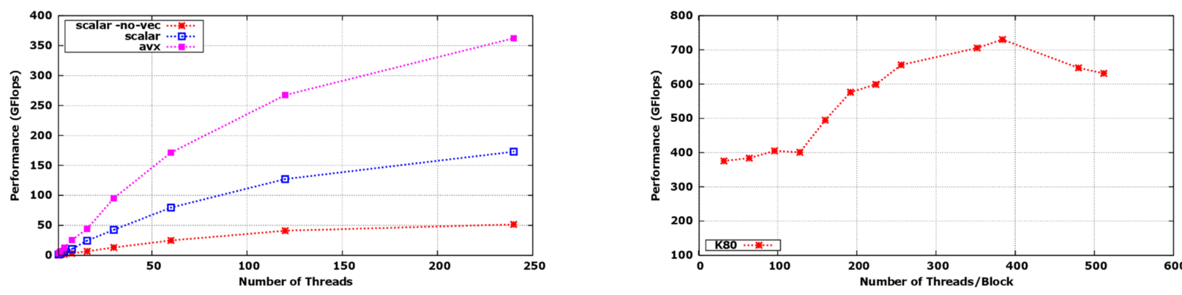

Figure 3: Performance of the `collide` kernel as a function of the number of threads for the Xeon-Phi accelerator (left) and for one GPU of the K80 board (right).

steps, and at each iteration executes several kernels.

Lattice data is stored in column-major order, and we keep in memory two copies; while wasting some memory space, this choice allows to process many lattice sites in parallel, as each kernel reads input data from one copy and write results onto the other. We surround the physical lattice with H_x halo-columns and H_y halo-rows; for a physical lattice grid of size $L_x \times L_y$, we allocate $N_x \times N_y$ lattice points, where $N_x = 2H_x + L_x$, and $N_y = 2H_y + L_y$. This makes the computation uniform for all sites and avoids control-flow divergences that negatively impact performances.

For the Xeon-Phi we write a hybrid program which runs on the host and on the KNC processor [8]. One starts from a standard `C` or `C++` code and uses `#pragma offload` directives to identify the parts of the code to be executed onto the KNC. The compiler generates code that automatically offload code and data to the KNC memory and starts all required threads.

For GPUs, we use CUDA-C , the NVIDIA programming language for GPUs [15]. CUDA-C is again a slightly modified `C` (or `C++`) program including keyword extensions defining data-parallel functions, called *kernels* and executed by GPUs. A CUDA-C program has code running on the CPU and kernels. Kernels typically generate a large number of threads and independent operations, that exploit data parallelism. Threads generated by a kernel are grouped into blocks which in turn form the execution *grid*. Blocks are arrays of threads which run on the same SMX and share data through a fast shared memory.

The two architectures use different programming tools, but the issues faced by programmers are similar: in order to exploit parallelism, one must ensure that all cores work in parallel, data is allocated in such a way that it can be fetched efficiently by memory controller and the code structure allows an efficient exploitation of vector parallelism.

4.1 OPTIMIZING FOR THE MIC ARCHITECTURE

For the Xeon-Phi we adopt the Array-of-Structures (AoS) memory layout, storing the populations of each site at contiguous memory addresses; this layout keeps all population data of each lattice site at contiguous addresses and better suits the cache structure of the KNC. Each iteration starts with the execution of the propagate_mic function, which performs the pbc and the propagate phases together. pbc enforces periodic boundary conditions along the X dimension; in our case this is simply a copy of fresh data to the halo Y columns.

The propagate kernel moves populations of each site according to the pattern defined in Eq. 1 and visualized in Figure 1 at left. This step does not perform any floating-point computation; it is basically a rearrangement of data in memory, implying memory accesses with sparse address patterns. propagate_mic spawns N_t threads, each threads handling a *sub-lattice* of size $(L_x/N_t) \times L_y$; first, two threads execute pbc to update the left and right halo columns, then all N_t threads apply the propagate step, each onto a different portion of the lattice.

To exploit *vector* parallelism, we divide the lattice in K strips along the Y dimension, and pack together populations of sites at distance L_y/K, see right-side of Figure1. In this approach our lattice is an array of vector sites, and each vector site is itself an array of 37 AVX vectors, each holding K populations. Within each thread, K sites are processed in parallel by vector instructions. We set $K = 8$, the number of double-precision data words that can be packed into a 512-bit AVX vector.

Streaming vector instructions can be automatically inserted by the compiler; this approach is a simple and fast option for the programmer, but efficiency is limited by the ability of the compiler to identify parts of the code on which vectorization can be applied. A potentially more efficient approach explicitly introduces vector variables and processes them by so-called *intrinsic* functions which are mapped directly onto the corresponding assembly instruction. We use vector programming and intrinsic functions, based on our previous experience [16, 17] with Intel processors for which auto-vectorization yielded sub-optimal performances.

In the left-side of Figure 2 we show the bandwidth measured in several implementations of the propagate kernel for N_t values up to four times the number of cores. The bandwidth obtained via an automatic vectorization is rather poor (scalar); bandwidth increases significantly (by a factor 2) as one uses AVX vectors through intrinsic functions (avx+store); a further significant gain is obtained using the STORENRNGO vector streaming instruction that does not waste time and bandwidth loading populations values into cache before updating them (avx+storenrgo). For comparison, we also show the results of the STREAM memory benchmark, which attains a maximum bandwidth of $\approx$ 150 GB/s corresponding to $\approx$ 40% of peak. Under this constraints our implementation reached $\approx$ 65% of the effective memory bandwidth.

After propagate_mic completes, the host launches the bc_mic kernel that applies boundary conditions at the top and bottom edges of the lattice. This function runs several threads, each one operating only on the three rows at the top and bottom of its lattice slice; the execution time of this phase has a minor impact on the overall performance.

The next step is the execution of collide_mic, which performs the collision of populations gathered by the propagate step. This is the truly floating-point intensive part of

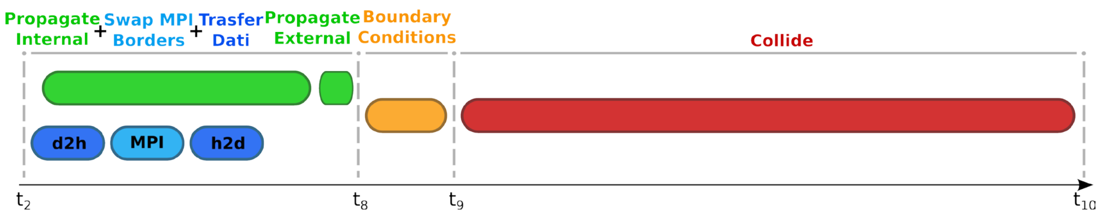

Figure 4: Program schedule allowing to overlap communications and processing of the `propagate` kernel.

the code. It performs approximately 7000 double-precision operations per site and offers in principle a degree of parallelism as large as the lattice size, as the processing of each site uses its own set of variables. The `collide_mic` kernel spawns several threads, up to 4 per core, each thread processing a slice of the lattice. We code `collide` using intrinsic functions and enforce SIMD parallelism explicitly, processing 8 lattice sites packed in an AVX vector. In the left-side of Figure 3 we show the performance of three different implementations, showing the performance gain obtained as more and more aggressive optimization steps are taken. One sees that automatic vectorization increases performance by a factor 3.4 over a basic non-vectorized version. A carefully handcrafted AVX-based optimization offers a further 2× improvement. Our best result is a performance of 360 GFlops, corresponding to an efficiency of 30% of the (double-precision) peak.

4.2 OPTIMIZING FOR THE KEPLER ARCHITECTURE

Our implementation for GPUs is described in detail in references [18, 17, 15]; here we recall the main results and update performance results including those measured on the recently introduced K80 accelerator.

A key point here is that a different data layout is needed: we adopt the Structure-of-Arrays (SoA) memory scheme, since it helps exploit the coalescing of global memory accesses, relevant to obtain a high memory bandwidth on these processors. To appreciate the relevance of this problem, consider that our tests have shown that adopting the AoS approach, we incur in a performance penalty as large as 5X for *propagate* and 2X for *collide*. Figure 2 shows the effective bandwidth as a function of the number of threads per block. This kernel is strongly memory-bound, and performance is substantially constant for a number of threads-per-block larger than 64, reaching a peak of 155 GB/s.

As in the case of Xeon-Phi, `collide` executes after enforcing boundary conditions to the top and bottom of the lattice. `collide` is strongly compute bound and executes 6472 double-precision additions and multiplications for each lattice site. We use data prefetch to hide memory accesses and all loops have been unrolled using `#pragma unroll` directive. Exploiting the large register file, this allows to keep population values on registers and avoid to load them several time from memory. Figure 3 shows the performance measured by our CUDA implementation as a function of the number of threads. Performance improves up to 384 threads per block reaching a value of $\approx$ 700 GFlops corresponding to $\approx$ 48% of peak.

```
// launch asynchrouns transfer from device to host (D2H)
#pragma offload_transfer: out( cf2[LEFT_HALO]  : REUSE into( send_L_buf ) ) signal( &send_L_buf )
#pragma offload_transfer: out( cf2[RIGHT_HALO] : REUSE into( send_R_buf ) ) signal( &send_R_buf )

// launch asynchronous execution of propagate kernel over BULK
#pragma offload: signal( &internal_prop_signal ){ propagate_m ( ... ); }

// wait end of d2h transfer
#pragma offload_wait: wait( &send_L_buf )
#pragma offload_wait: wait( &send_R_buf )

// execute halos SWAP
MPI_Sendrecv(send_R_buf to mpi_rank_R, TAG_RIGHT, recv_L_buf to mpi_rank_L, TAG_RIGHT);
MPI_Sendrecv(send_L_buf to mpi_rank_L, TAG_LEFT,  recv_R_buf to mpi_rank_R, TAG_LEFT);

// launch asynchrouns transfer from host to device (H2D)
#pragma offload_transfer: in( recv_L_buf : REUSE into(cf2[LEFT_HALO] )) signal( &recv_L_buf )
#pragma offload_transfer: in( recv_R_buf : REUSE into(cf2[RIGHT_HALO])) signal( &recv_R_buf )

// wait end of h2d transfer
#pragma offload_wait: wait( &recv_L_buf )
#pragma offload_wait: wait( &recv_R_buf )

// launch asynchronous execution of propagate over left- and right-columns
#pragma offload { propagate_m ( ... ); } signal(&external_prop_signal)

// wait end of propagate kernels
#pragma offload_wait: wait( &internal_prop_signal )
#pragma offload_wait: wait( &external_prop_signal )
```

Figure 5: Scheduling of operations on the MIC board to overlap communications and execution of `propagate`.

4.3 MULTI-ACCELERATOR IMPLEMENTATION

On multi-accelerator systems, accelerators are installed on the same host (typically up to 4 or 8 accelerators) or on multiple hosts that exchange data using commodity networks such as Infiniband. Here we discuss an implementation for a single-host with four accelerators.

Our implementation splits a lattice of size $L_x \times L_y$ on N_p accelerators along the X dimension, each handling a *sub-lattice* of size $L_x/N_p \times L_y$. This splitting is necessary to have continuous halo-columns allocated in memory, and avoids a further gather-step to collect halos on a contiguous buffer before doing communications. This allocation scheme implies a virtual ordering of the accelerators along a ring, so each one is connected with a previous and a next companion; at the beginning of each time-step, before starting `propagate`, accelerators must exchange data, since cells close to the right and left edges of the sub-lattice needs data allocated on the logically previous and next nodes.

When using multi-accelerator systems, the overlap of communications with computation is a key approach to scalability. In our case, the key point to consider is that `propagate` for the bulk of the lattice (all lattice points except for three columns at right and left) has no data dependency with `pbc` (while `propagate` on the edges depends on fresh data moved to the halos by `pbc`). Our strategy therefore tries to overlap as much as possible data transfers with the execution of `propagate` on the bulk.

We have scheduled operations as shown in figure 4: i) the host launches propagate on the bulk of the lattice; this is an asynchronous kernel which runs in parallel with the following steps; ii) the host starts the copy of the left and right column borders from device

```
// launch asynchronous execution on stream[0] of propagate over bulk
propagate <<<grid, block, 0, stream[0] >>> (f1+BULK_OFF, f2+BULK_OFF);

// execute MPI transfers
for(kk = 0; kk < NPOP; kk++){
  MPI_Sendrecv(
    f2_d+(kk*NX*NY)+(HX+SIZEX-3)*NY, 3*NY, MPI_DOUBLE, mpi_nxt, 0,
    f2_d+(kk*NX*NY),                 3*NY, MPI_DOUBLE, mpi_prv, 0,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);

  MPI_Sendrecv(
    f2_d+(kk*NX*NY)+HX*NY,           3*NY, MPI_DOUBLE, mpi_prv, 1,
    f2_d+(kk*NX*NY)+(HX+SIZEX)*NY,  3*NY, MPI_DOUBLE, mpi_nxt, 1,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

// launch asynchronous execution of propagate for left and right borders on stream[1] and stream[2].
propagate <<< propGridLR, propBlockLR, 0, stream[1] >>> (f1_d+LC_OFF, f2_d+LC_OFF);
propagate <<< propGridLR, propBlockLR, 0, stream[2] >>> (f1_d+RC_OFF, f2_d+RC_OFF);

// wait all kernel finish
cudaDeviceSynchronize()
```

Figure 6: Scheduling of operations on GPUs to overlap communications and execution of `propagate`.

to host (`D2H`); iii) as `D2H` finishes the host performs the corresponding MPI communications to exchange data with neighbor nodes; iv) as MPI receive operations are completed data are moved back on the accelerator (`H2D`); v) finally `propagate` executes on the left and right borders. When all the above steps are completed execution continues with `bc` and `collide` kernels. This schedule is the same for our MIC and GPU implementations.

On MIC systems MPI functions cannot access data allocated on the accelerator, so the code must explicitly move data between host and accelerator. All steps to overlap computation of `propagate` with `pbc` are detailed in Figure 5.

On GPUs MPI communication are integrated with CUDA through UVA (*Unified Virtual Addressing*), supporting a common address space between host and device. This means that MPI functions can address memory areas allocated on the GPU. Figure 6 shows how `propagate` and `pbc` are scheduled in this case.

5 RESULTS AND CONCLUSIONS

In Table 1 we compare performance figures of our code optimized for Xeon Phi, Kepler K80 and for a dual-processor commodity systems [17] (dual Intel E5-2630), based on the

Table 1: Performance comparison of the `propagate` and `collide` kernels on KNC and Kepler accelerators and on a dual 8-core E5-2630 (Intel Haswell micro-architecture) processor running at 2.4 GHz.

	Intel Xeon 7120				NVIDIA K80				E5-2630 v3	
# devices(s)	1	2	3	4	1	2	3	4	1	2
Propagate (GB/s)	85	161	225	274	154	266	393	520	40	88
S_r	1.0X	1.90X	2.65X	3.22X	1.0X	1.72X	2.55X	3.37X	1.0X	2.2X
Collide (GFs)	358	709	1056	1383	667	1359	2029	2706	111	222
S_r	1.0X	1.98X	2.95X	3.86X	1.0X	2.03X	3.07X	4.05X	1.0X	2.0X
Global P (MLUPS)	39	73	110	139	72	140	209	277	12	23
S_r	1.0X	1.95X	2.82X	3.56X	1.0X	1.93X	2.88X	3.84X	1.0X	1.98X

Haswell micro-architecture.

We first focus on comparing performances with just one accelerator. The `propagate` kernel is a memory-bound step corresponding to memory copies with sparse address patterns. On the *Kepler* architecture we reach ≈ 64% of the available peak bandwidth. The *Xeon-Phi*, that uses the same class of memories, obtains a lower bandwidth, ≈ 85 GB/s, that is ≈ 24% of peak. This is mainly due to the limited bandwidth (≈ 220 GB/s) of the internal ring, connecting cores and memory controllers. On the Haswell processor we measure ≈ 40 GB/s corresponding to ≈ 67% of the available peak.

The `collide` kernel is a strongly compute-bound step, requiring approximately 20 double-precision floating-point operations per byte. On *Kepler*, this kernel reaches a sustained performance of ≈ 46% of the available peak. The *Xeon-Phi* performance is lower, only approximately ≈ 30% of the available peak, while on the Haswell CPU we measure an efficiency of ≈ 29%. The last section (Global P) of the table shows the performance of the full code, measured in *Millions Lattice Update Per Second* (MLUPS). Comparing with the traditional eight-core CPU, the *Xeon-Phi* is ≈ 3X faster, while on one GPU of the K80 system, the speed-up is 6X.

Table 1 also shows scalability results (S_r). We see that the individual steps and the full code scale quite well meaning that communications have been hidden with computation; running with 4 GPUs the sustained performance of the full code is 1.7 TFLOPS.

In conclusion, our application running on one accelerator are up to 6X faster than a recent standard CPU, and also a good scaling on multi-accelerator systems. While these are valuable results, we underline that they were obtained with handcrafted optimizations tailored for each target accelerator. This is due to the still immature programing methodologies able to map the same code on different accelerator architectures. Portability of codes and also of performances is today a real issue that needs to be solved to make accelerators a widespread solution for HPC computing.

Acknowledgements

This work was done in the framework of the COKA, COSA and SUMA projects of INFN. We thank CINECA (Bologna, Italy), and the NVIDIA Jülich Application Lab (Jülich Supercomputer Center, Germany) for allowing us to use their computing systems.

REFERENCES

[1] Pohl, T., et al., Performance evaluation of parallel large-scale lattice boltzmann applications on three supercomputing architectures *Proc. of SC* (2004) doi:10.1109/SC.2004.37

[2] Biferale, L., et al., Lattice boltzmann fluid-dynamics on the QPACE supercomputer *Proc. Comp. Science* **1**(1) (2010) 1075–1082 doi:10.1016/j.procs.2010.04.119

[3] Pivanti, M., et al., An optimized lattice boltzmann code for BlueGene/Q *LNCS* **8385** (2014) 385–394 doi:10.1007/978-3-642-55195-6_36

[4] Sano, K., et al., FPGA-based streaming computation for lattice boltzmann method *Field-Programmable Technology* (2007) 233–236 doi:10.1109/FPT.2007.4439254

[5] Biferale, L., et al., Optimization of multi-phase compressible lattice boltzmann codes on massively parallel multi-core systems *Proc. Comp. Science* **4** (2011) 994–1003 doi:10.1016/j.procs.2011.04.105

[6] Biferale, L., et al., A multi-gpu implementation of a D2Q37 lattice boltzmann code *LNCS* **7203** (2012) 640–650 doi:10.1007/978-3-642-31464-3_65

[7] Bailey, P., et al., Accelerating lattice Boltzmann fluid flow simulations using graphics processors In: Parallel Processing (2009) 550–557 doi:10.1109/ICPP.2009.38

[8] Crimi, G., et al., Early Experience on Porting and Running a Lattice Boltzmann Code on the Xeon-phi Co-Processor *Proc. Comp. Science* **18** (2013) 551–560 doi:10.1016/j.procs.2013.05.219

[9] Succi, S., *The Lattice-Boltzmann Equation* Oxford university press (2001)

[10] Aidun, C.K., Clausen, J.R., Lattice-Boltzmann Method for Complex Flows *Annual Review of Fluid Mechanics* **42** (1) (2010) 439–472

[11] Sbragaglia, M., et al., Lattice Boltzmann method with self-consistent thermo-hydrodynamic equilibria *J. of Fluid Mechanics* **628** (2009) 299–309 doi:10.1017/S002211200900665X

[12] Scagliarini, A., et al., Lattice Boltzmann methods for thermal flows: Continuum limit and applications to compressible Rayleigh–Taylor systems *Physics of Fluids* **22**(5) (2010) 055101. doi:10.1063/1.3392774

[13] Biferale, L., et al., Second-order closure in stratified turbulence: Simulations and modeling of bulk and entrainment regions *Phys. Review E* **84**(1) (2011) 016305 doi:10.1103/PhysRevE.84.016305

[14] Biferale, L., et al., Reactive Rayleigh-Taylor systems: Front propagation and non-stationarity EPL (Europhysics Letters) **94**(5) (2011) 54004 doi:10.1209/0295-5075/94/54004

[15] Kraus, J., at al., Benchmarking GPUs with a parallel Lattice-Boltzmann code *Proc. of Computer Architecture and High Performance Computing* (SBAC-PAD) (2013) 160–167 doi:10.1109/SBAC-PAD.2013.37

[16] Mantovani F et al, Exploiting parallelism in many-core architectures: Lattice Boltzmann models as a test case. *J. Phys. Conf. Series* **454** 012015 (2013) doi:10.1088/1742-6596/454/1/012015

[17] Mantovani F et al, Performance issues on many-core processors: a D2Q37 Lattice Boltzmann scheme as a test-case. *Comp. & Fluids* **88** (2013) doi:10.1016/j.compfluid.2013.05.014

[18] Biferale L et al, An Optimized D2Q37 Lattice Boltzmann Code on GP-GPUs. *Comp. & Fluids* **80** (2013) doi:10.1016/j.compfluid.2012.06.003