Journal of Physics: Conference Series **513** (2014) 052032 doi:10.1088/1742-6596/513/5/052032

Computing on Knights and Kepler Architectures

G Bortolotti[1], M Caberletti[1], G Crimi[2], A Ferraro[1], F Giacomini[1], M Manzali[1], G Maron[1], M Pivanti[3], D Salomoni[1], S F Schifano[4], R Tripiccione[3] and M Zanella[2]

[1] CNAF - INFN, ITALY
[2] Università di Ferrara, ITALY
[3] Dip. di Fisica, Università di Ferrara, and INFN, ITALY
[4] Dip. di Informatica e Matematica, Università di Ferrara and INFN, ITALY

E-mail: bortolotti@cnaf.infn.it, caberletti@cnaf.infn.it, gianluca.crimi@student.unife.it, ferraro@cnaf.infn.it, giacomini@cnaf.infn.it, manzali@cnaf.infn.it, maron@cnaf.infn.it, pivanti@fe.infn.it, salomoni@cnaf.infn.it, schifano@fe.infn.it, tripiccione@fe.infn.it, marc.zanella@student.unife.it

Abstract. A recent trend in scientific computing is the increasingly important role of co-processors, originally built to accelerate graphics rendering, and now used for general high-performance computing. The INFN *Computing On Knights and Kepler Architectures* (COKA) project focuses on assessing the suitability of co-processor boards for scientific computing in a wide range of physics applications, and on studying the best programming methodologies for these systems. Here we present in a comparative way our results in porting a Lattice Boltzmann code on two state-of-the-art accelerators: the NVIDIA K20X, and the Intel Xeon-Phi. We describe our implementations, analyze results and compare with a baseline architecture adopting Intel Sandy Bridge CPUs.

1. Introduction

Accelerators are quickly becoming key building blocks of HPC processors. Accelerators try to boost the performance of more traditional CPUs with an architecture based on (some combination of) a large number of cores, vector-SIMD processing, multi-threading. There is a large spectrum of computing architectures that one may devise using these building blocks, but – at present – there is a convergence on GPUs, that use a very large number of slim cores, and on MIC processors, recently introduced by Intel and integrating a relatively smaller number of largish cores; each core is a streamlined version of a traditional Intel processor relying on SIMD processing to increase performance. Accelerators today have peak performances of the order of 10^{12} *FLoating-point Operations Per Second* (Tflops); however for real codes it is not easy to extract a large fraction of the theoretically available performance, even if the parallelism that one must exploit is easily uncovered in the underlying algorithms. The difficulties arise for two different reasons. Firstly, there is a potential conflict between the parallelizing strategies that one must follow: for instance, building large vectors that can be operated on in SIMD mode is often difficult if one first has to partition the computation on a large number of cores. Secondly there are data transfer bottlenecks, associated to i) memory access to the accelerator memory in order to fetch the data items that must be processed and ii) bandwidth and latency issues in the transfer of data between host and accelerator.

Table 1. Hardware features of several multi core systems: *NVIDIA Tesla K20X* is based on the *Kepler* processor, *Intel Xeon-Phi* is based on the MIC architecture, while Xeon E5-2680 is a commodity processor that we use as a performance baseline.

	Intel Xeon E5-2680	NVIDIA Tesla K20X	Intel Xeon-Phi 7120P
#physical-cores	8	14	61
#logical-cores	16	2688	244
clock (GHz)	2.7	0.735	1.238
GFLOPS (DP)	172.8	1317	1208
SIMD	AVX 64-bit	N/A	AVX2 512-bit
cache (MB)	20	1.5	30.5
Mem BW (GB/s)	51.2	250	352
Watt	130	235	300

In this paper we offer a comparative assessment of these architectures, by describing the implementation issues and the performance results for a fluid-dynamics code that solves the Navier-Stokes equation for a 2D fluid using a recently developed Lattice Boltzmann (LB) method; this analysis extends results already presented in [1, 2, 3].

LB methods (see e.g. [4] for a detailed introduction) are discrete in both position and momentum spaces; they are based on the synthetic dynamics of (so called) *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 among one another; in this step they mix and their values change accordingly. We consider a 2D model that uses 37 populations and describes the dynamics of a fluid that follows the equation of state of a perfect gas (in LB jargon a method in x dimensions with y populations is labeled as $DxQy$, so we consider a $D2Q37$ model). In LB methods the macroscopic variables are functions of the populations f_l; at each time step each f_l drifts to a nearby grid site in a fixed direction, identified by its velocity vector $\boldsymbol{c}_l$. The master evolution equation is

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

where $f_l^{(eq)}$ is the local equilibrium distribution, depending on the local macroscopic variables. One sees immediately that each time step evolution can be performed independently on all lattice sites, offering a huge amount of available parallelism.

This paper is organized as follows: in section 2 we give a short description of the accelerator boards that we have used; in section 3 we describe the implementation of our code for both the K20X and Xeon-Phi systems, showing benchmark results that have guided our implementation choices; finally in section 4 we present performance results for our production codes and our concluding remarks.

2. The Xeon-Phi and K20X accelerator boards

The Xeon-Phi and K20X boards, see Table 1, are co-processors of a traditional host system, connected to the host via 16 PCIe lanes providing a peak bandwidth of 8 GB/s.

The Xeon-Phi board has one *Knights Corner* (KNC) processor, the first production chip based on the *Many Integrated Core* (MIC) architecture, and 8 GB of GDDR5 RAM. The KNC integrates up to 61 CPU-cores interconnected by a high-speed bi-directional ring, and runs at $\approx$1 GHz. It connects to its private external with a peak bandwidth of $\approx$320 GB/s. Each core is based on the Pentium architecture; it has 32 KB of L1 cache for data and instructions, 512 KB L2 data-cache, and a 512 bit vector Floating Point Unit (FPU). The FPU engine performs so called *fused-multiply-add* (FMA) instructions; in this operation one addition and one multiplication are executed together in one clock cycle. Consequently the peak performance is $\approx$32 (16) GFlops in single (double) precision, if all elements of the data vector are used at all clock cycles. In this case, the KNC delivers a peak performance of $\approx$2 (1) Tflops in single

(double) precision. Data on all L2 data-caches are shared among the cores through a coherency protocol running over the interconnect ring. The KNC runs a lightweight version of the Linux operating system and each core supports the execution of up-to 4 hardware threads. For more details see [5].

The K20X board has one *Kepler* processor and 6 GB of GDDR5 memory. The *Kepler* processor architecture is massively parallel, with 14 *Streaming Multiprocessors* (SMXs). Each SMX handles up to 2048 active threads and has 192 scalar cores to process them. Unlike typical CPU threads, *Kepler* threads are extremely lightweight: context switches between two threads happen on a cycle-by-cycle basis, so typically one thread processes just one element of the program data set. At each clock cycle the SMX schedules and executes *warps*, groups of 32 threads which are processed in SIMD fashion. The *Kepler* processor has a peak performance of ≈ 1 Tflops in double precision and more than 4 Tflops in single precision. On *Kepler* each thread addresses 256 32-bit registers and there are 65536 registers for each SMX. The memory controller has a peak bandwidth of 250 GB/s. For more details see [7].

3. Implementation of the LB code

In this section we describe the optimization of our LB code for single-host systems with either a *Xeon-Phi* or a *K20X* board; we also show benchmark results which have supported our choices.

The Xeon-Phi system is programmed using the *accelerator* or *offloading* approach, consisting in developing a hybrid program which runs on the host and on the KNC processor. The user writes a standard `C` or `C++` code and uses `#pragma offload` directives to identify the parts of the code to be offloaded and executed onto the MIC. The compiler generates code that transparently transfers control to the MIC processor. The offloaded function is a standard `C` or `C++` program, that can spawn several threads running on all available cores.

For GPUs, we use CUDA-C [6], the NVIDIA programming language for GPUs. A CUDA-C program contains one or more functions that run either on the host or on a GPU. Functions with no (or limited) parallelism run on the host, while those exhibiting a large degree of data parallelism run on the GPU. A CUDA-C program is a slightly modified `C` (or `C++`) program including keyword extensions defining data-parallel functions, called *kernels* on the GPU. Kernel functions 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.

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

For both implementations we adopt the *offload* approach, whereby execution is controlled by the host: the host first uploads the lattice onto the accelerator memory and then performs a loop over time steps; at each iteration it offloads the execution of several kernels, described in the following sub-sections.

In our codes, lattice data is stored in column-major order, and we keep two copies in memory. This choice uses more memory than really necessary, but makes it much simpler to handle many lattice sites in parallel, as we read input data from one copy and write results onto the other. The physical lattice is surrounded by 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 performance.

3.1. Optimizing for the Xeon-Phi

For the Xeon-Phi we adopt the Array-of-Structures (AoS) memory scheme, storing the populations of each site at contiguous memory addresses; in fact, the AoS scheme keeps all population data of each lattice site at contiguous addresses and better suits the cache structure of the KNC.

At the beginning of each iteration the host first offloads the execution of the `propagate_m` function, which performs the `pbc` and the `propagate` phases together. `pbc` enforces periodic boundary conditions

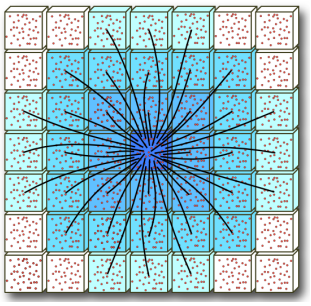

Figure 1. Move patterns for populations in the `propagate` phase of the LB D2Q37 method.

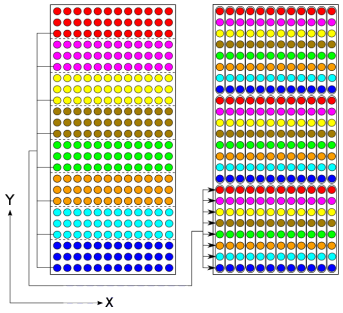

Figure 2. Data packing within AVX vectors of lattice data for the Xeon-Phi implementation.

along the X dimension; in our case this is simply a copy of fresh data to the halo columns. The `propagate` kernel moves populations of each site according to the pattern defined in Eq. 1 and visualized in Figure 1. 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. The `propagate_m` function is an OpenMP program which spawns N_t threads. The lattice is split among the threads along the X dimension, and each thread processes a *sub-lattice* of size $(L_x/N_t) \times L_y$. Two threads execute first the `pbc` phase 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. Within each thread, K sites are processed in parallel in order to exploit the data parallelism made available by vector instructions. In our case $K = 8$ is exactly the number of double-precision data words that can be packed into a 512-bit AVX vector. Streaming vector instructions are automatically inserted by the compiler, or explicitly invoked by the programmer, by coding *intrinsic* functions(see next paragraph for details). In the first case the program is a scalar code, compiled enabling *auto-vectorization* flags (e.g. `-O2` or `-O3` for the *Intel C/C++* compiler). The compiler automatically exploits data parallelism and uses streaming instructions if specific conditions are met. 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. For example, a double-precision sum on a vector of 8 elements is started by the code line `d = _mm512_fmadd_pd (a, b,c)` where `a, b, c, d` are 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. Our codes explicitly uses vector programming and intrinsic functions, based on our previous experience [8, 9] with Intel processors for which auto-vectorization yielded sub-optimal performances.

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. 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. In Figure 3 we show the bandwidth measured in several implementations of the `propagate` kernel for N_t values as large as four times the number of cores. We see that 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 to load onto the cache full data lines as we know that the full line will be updated within the loop (avx+storenrgo). In the same picture we also report the results of the STREAM memory benchmark [10], which attains a maximum bandwidth of ≈ 150 GB/s corresponding to $\approx 40\%$ of the peak. This is due to the limited bandwidth of the internal ring of the KNC which connects the cores to the memory controller. Under this constraints our implementation reached $\approx 65\%$ of the effective memory bandwidth.

After `propagate_m` completes, the host launches the `bc_m` kernel that applies the boundary conditions at the top and bottom of the lattice. This function is also an OpenMP program which runs several threads, each one operating only on the topmost and lower-most three rows of its slice. Since the computation of boundary conditions occurs only on a few lattice sites, the execution time of this phase is negligible.

The next step is the execution of `collide_m`, which performs the collision of populations gathered by the `propagate` step. This is the truly floating-point intensive part of the code. It performs approximately 7000 double-precision operations per site, offering in principle a degree of parallelism as large as the lattice size, as the processing of each site uses its own set of population variables.

The `collide_m` kernel is yet another OpenMP program which 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 Figure 5 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 $2\times$ improvement. Our best result is a performance of 360 GFlops, corresponding to an efficiency of 30% of the (double-precision) peak.

3.2. Optimizing for the K20X

For the GPU code we have adopted 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.

In this case each phase is performed by a CUDA kernel. Each block is configured as a unidimensional array of `N_THREAD` threads, processing populations allocated at successive locations in memory, in order to exploit data coalescing. The grid of blocks is a bi-dimensional array of $(L_y/\texttt{N\_THREAD} \times L_x)$ blocks. One drawback is that when we compute the `bc` kernel, many blocks are inactive, but, as underlined before, the impact of this kernel on performance is negligible.

At the beginning of each time step, the host runs `pbc` to enforce periodic boundary conditions by launching two asynchronous memory copies. All following steps, described in the previous subsection, are offloaded to the K20 device in sequential order.

In Figure 4 we show the effective bandwidth (with and without error correction, ECC) of our implementation as a function of the number of threads per block. The performance of this obviously memory-bound kernel depends strongly on the available memory bandwidth, which on the *Kepler* architecture is substantially constant for a number of threads-per-block larger than 64. With ECC enabled we measure a bandwidth of $\approx$ 160 GB/s. Disabling ECC, the bandwidth increases approximately by a

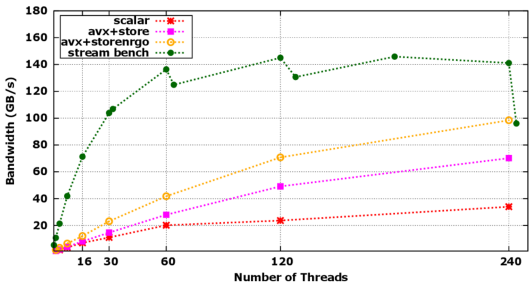

Figure 3. Performance of the `propagate` kernel on the Xeon-Phi. We include for comparison results of the STREAM memory benchmark.

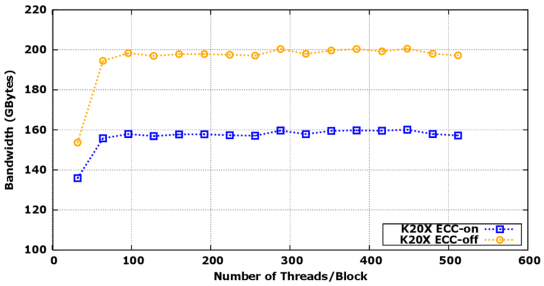

Figure 4. Performance of the `propagate` kernel on the K20X, with and without Error Correction (ECC).

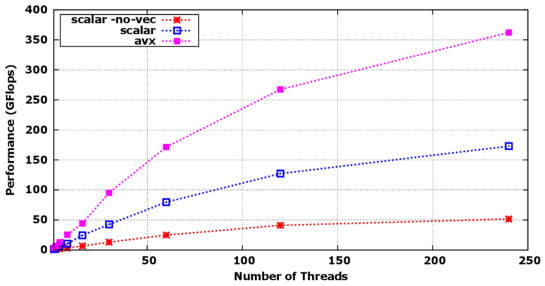

Figure 5. Performance of the `collide` kernel on the Xeon-Phi board.

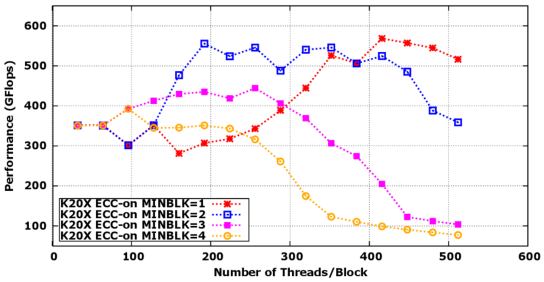

Figure 6. Performance of the `collide` kernel on the K20X board.

factor 1.25×, but we have not used this operation mode in order to make simulations robust against memory failures.

As in the previous case, `collide` executes after enforcing boundary conditions to the top and bottom of the lattice. We have profiled the code using the `nvprof` tool of NVIDIA. After compilation and optimization the `collide` kernel executes 6472 double-precision additions and multiplications for each lattice site; the processor executes this mix of operations as FMA instructions in $\approx 70\%$ of the cases, while the remaining 30% is executed as ADD (10%) and MUL (20%) instructions, slightly reducing the overall performance. Moreover, the kernel needs several constants which must be stored in the constant memory of the device. We implemented data prefetch to hide memory accesses and all loops accessing the thread-private prefetch array have been unrolled via `#pragma unroll`. This allows the compiler to keep the elements of the prefetch array in registers belonging to the very large register file on *Kepler*. We have experimentally tuned for best performance, which is a tradeoff between effective occupancy of the pipeline and register spilling, by varying the minimum number of blocks per SMX (`MINBLK`). This is easily done using `launch_bounds` [6]. Figure 6 shows the performance measured by our CUDA implementation as a function of the number of threads. We have benchmarked the kernel using several values of the `MINBLK` parameter. We find that `MINBLK=2` gives the best performance for a wide range of threads per block. Performance improves up to 256 threads per block reaching a value of ≈ 560 GFlops corresponding to $\approx 43\%$ of the peak; as we try to use a larger number of threads the performance drops again because the number of needed registers is larger than the available resources of the SMXs.

4. Results and conclusions

In Table 2 we compare the performance figures for the two presented implementations. We also include the performance of the same code developed and optimized for a dual-processor commodity system (dual

Table 2. Performance comparison of the `propagate` and `collide` kernels on accelerators and on a dual eight-core E5-2680 (Intel Sandy Bridge) processor running at 2.7 GHz. ε is the efficiency w.r.t. the available peak. For `collide` we also measure performance in *Mega Lattice Update per Second* (MLUPS), a user-friendly metric. Finally we list the energy needed to update one lattice cell.

	Intel dual E5-2680	Intel Xeon-Phi 7120X	NVIDIA K20X
propagate GB/s	60	98	155
ε	70%	28%	62%
collide GF/s	220	362	565
ε	63%	30%	43%
MLUPS	29	54	64
μJ / site	8.96	5.55	3.67

E5-2680), based on the Sandy Bridge architecture, see [8].

The `propagate` kernel is a memory-bound step which behaves like a memory-copy with very sparse memory addressing. On the *Kepler* architecture we reach $\approx 62\%$ of the available peak bandwidth, roughly the same as on the Sandy Bridge system; however the effective bandwidth – made possible by the GDDDR5 memories – is much higher. The *Xeon-Phi*, that uses the same memories, reaches a lower bandwidth, ≈ 100 GB/s, that is $\approx 30\%$ of peak. This is mainly due to the limited bandwidth (≈ 220 GB/s) of the internal ring, connecting cores and memory controllers.

The `collide` kernel is a strongly compute-bound step, requiring approximately 20 double-precision floating-point operations per byte. On the *Kepler* processor this kernel exploits more than 70% of the available FMA instructions and attains a maximum performance of $\approx 40\%$ of the available peak. The *Xeon-Phi* performance is lower, reaching approximately 30% of the available peak. All in all, the *Xeon-Phi* is faster by roughly $1.6\times$ with respect to the more traditional Sandy Bridge system; this speed-up figure grows to $2.6\times$ for the K20X.

Table 2 reports also the respective power consumptions, measured as energy required to update one lattice site. Accelerators have a better value than the dual Sandy Bridge system, with a significant improvement made possible by the latest generation of NVIDIA GPUs.

In conclusion, our application enjoys a $2\times - 3\times$ performance increase using accelerators; accelerators also help reduce the power budget. While these are valuable results, they were obtained with very careful handcrafted optimization work, tailored for the specific target architectures. This leads us to think that there is still a lot of architectural and software work ahead before accelerators become widely used in HPC architectures. From the software point of view, programming methodologies that can be shared across different accelerator technologies are necessary.

Acknowledgments

This work has been done in the framework of the COKA and SUMA projects of INFN.

References

[1] Crimi G et al 2013 Proc. Comp. Science **18** doi:10.1016/j.procs.2013.05.219
[2] Biferale L et al 2013 Comp. and Fluids **80** doi:10.1016/j.compfluid.2012.06.003.
[3] Kraus J et al *Benchmarking GPUs with a parallel Lattice-Boltzmann code* 2013 Proc. of SBAC-PAD (in press)
[4] Succi S *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond* Oxford University Press (2001).
[5] `http://goparallel.sourceforge.net/wp-content/uploads/2013/07/Intel_-Xeon-Phi-Core-Micro-architecture.pdf`
[6] `http://docs.nvidia.com/cuda/cuda-c-programming-guide`
[7] `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`
[8] Mantovani F et al 2013 Comp. and Fluids doi:10.1016/j.compfluid.2013.05.014 (in press)
[9] Mantovani F et al 2013 J. Phys.: Conf. Ser. **454** 012015 doi:10.1088/1742-6596/454/1/012015
[10] McCalpin J *The STREAM Benchmark: Computer Memory Bandwidth* `http://www.streambench.org/`