Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200100

Porting a Lattice Boltzmann Simulation to FPGAs Using OmpSs

Enrico CALORE [a,1], Sebastiano Fabio SCHIFANO [a,b]
[a] *INFN Ferrara, Italy*
[b] *University of Ferrara, Italy*

Abstract. Reconfigurable computing, exploiting *Field Programmable Gate Arrays* (FPGA), has become of great interest for both academia and industry research thanks to the possibility to greatly accelerate a variety of applications. The interest has been further boosted by recent developments of FPGA programming frameworks which allows to design applications at a higher-level of abstraction, for example using directive based approaches.

In this work we describe our first experiences in porting to FPGAs an HPC application, used to simulate Rayleigh-Taylor instability of fluids with different density and temperature using Lattice Boltzmann Methods. This activity is done in the context of the FET HPC H2020 EuroEXA project which is developing an energy-efficient HPC system, at exa-scale level, based on Arm processors and FPGAs. In this work we use the *OmpSs* directive based programming model, one of the models available within the EuroEXA project. *OmpSs* is developed by the Barcelona Supercomputing Center (BSC) and allows to target FPGA devices as accelerators, but also commodity CPUs and GPUs, enabling code portability across different architectures. In particular, we describe the initial porting of this application, evaluating the programming efforts required, and assessing the preliminary performances on a Trenz development board hosting a Xilinx Zynq UltraScale+ MPSoC embedding a 16nm FinFET+ programmable logic and a multi-core Arm CPU.

Keywords. FPGA, OmpSs, EuroEXA, HPC, Lattice Boltzmann

1. Introduction

Reconfigurable computing using *Field Programmable Gate Arrays* (FPGA) is attracting lot of attention from the scientific community for its potential to accelerate a large variety of applications with interesting performance-energy ratios. However, the complexity of programming such devices has been one of the major issues preventing FPGAs to become widely adopted in scientific software communities. In fact, FPGAs have been commonly programmed using *Hardware Description Language* (HDL) such as VHDL and Verilog, which allow to describe arbitrary circuitry at *Register Transfer Level* (RTL). This approach is too low level for many application programmers, and has restricted the use of FPGA mainly to electronic engineering experts.

[1]Corresponding Author: Enrico Calore, INFN Ferrara, Via Saragat 1, 44121 Ferrara, Italy; E-mail: enrico.calore@fe.infn.it.

Despite of this, FPGAs have been successfully used in the past to boost the computing performance of several scientific applications. As an example: COPACOBANA [1] for code breaking; RIVYERA [2] for bioinformatics applications; EXTOLL [3] for communications and Janus [4,5] for spin-glasses simulations. These are all relevant projects using FPGAs as processing elements for HPC and scientific applications in general [6]. However, they required strong customization of applications, using data structures and implementations, very far from that used on commodity CPUs.

Recently, the resources available on FPGA chips have increased a lot, integrating high-end interfaces (e.g., PCIe, DDR3, GbitE, etc...) large static memories, large number of DSPs and also CPUs cores. The last feature in particular has raised the interest of many researchers for enabling FPGA-accelerated computing. A CPU is integrated on the device together with the programmable logic, and is able to run a full operating system, commonly based on GNU/Linux, allowing to start applications on the CPU to later offload specific functions on the FPGA.

In the last years, also the programming environments have been extensively developed. Languages such as OpenCL and High-Level Synthesis (HLS) frameworks based on *pragmas* directives are now available for different FPGAs, allowing to describe applications at algorithmic level [7]. This can be then interpreted and transcompiled by automatic software tools into a Register-Transfer Level (RTL) and finally synthesized to the gate level. In particular, for applications already developed for ordinary CPUs and accelerators, directive approaches allow to just annotate legacy C and Fortran codes with *pragmas* that guide the compilers in the synthesis process. Clearly, this approach is more abstract compared to a low level manual programming of the HDL code, and can results in less optimized designs in terms of timing and FPGA resources usage. Despite of this, the reduced programming effort required, combined with a faster design space exploration and a much higher software portability, make this approach very attractive and usable by larger application developers communities.

All of these factors contributed to the possibility of using FPGAs as computing accelerators, and many projects are now following this path. One of this is the EuroEXA project [2], a H2020 project funded by the EU, that following an hardware/software co-design approach, aims to port a rich mix of applications to its architecture. One of these applications consist in the simulation of fluids using Lattice Boltzmann Methods (LBM). The increasing popularity of LBM comes from its flexibility, allowing to study complex geometries and different types of boundary conditions, and from being particularly suitable for highly scalable implementations on massively parallel architectures [8].

In the EuroEXA project Xilinx FPGAs are adopted. Xilinx provides the VivadoHLS Design Suite to annotate C codes with proprietary HLS directives, allowing to offload a specific function to an FPGAs, automatically managing the host code compilation and the synthesis of the function to be offloaded. Anyhow, in this work we include a further level of programming abstraction over VivadoHLS and in particular we use the *OmpSs* directive based programming model in conjunction with its *OmpSs@FPGA* extension [9]. *OmpSs* allows to annotate the application code with directives to compile and offload a kernel on FPGAs, enabling accelerated computing, but given the same source code, it is also able to target other devices, such as GPUs or multi-core CPUs, easily enabling code portability [10].

[2] https://euroexa.eu/

The remainder of this contribution is organized as follows: in the next Section we introduce the EuroEXA project, in Sec. 3 we describe our LBM application, and in Sec. 4 we briefly overview *OmpSs@FPGA*. Sec. 5 describes our code porting to FPGA, Sec. 6 reports our results, and finally Sec. 7 highlights some concluding remarks.

2. The EuroEXA Project

The *Co-designed innovation and system for resilient exascale computing in Europe: from application to silicon* (EuroEXA) is a H2020 FET HPC project funded by the EU commission with a budget of $\approx$ 20M€. The aim of the project is to develop a prototype of an *exascale* level computing architecture suitable for both compute- and data-intensive applications, delivering world-leading energy-efficiency. To reach this goal this project proposes to adopt a cost-efficient, modular integration approach enabled by: novel iner-die links; FPGAs to leverage data-flow acceleration for compute, networking and storage; an intelligent memory compression technology; a unique geographically-addressed switching interconnect and novel Arm based compute units. As main computing elements are going to be adopted multi-core Arm processors combined with Xilinx UltraScale+ FPGAs, to be used both as compute accelerators and to implement an high bandwidth and low-latency interconnect between computing elements.

Form the software platform point of view, EuroEXA provides five high-level programming frameworks that enable FPGA-accelerated computing: Maxeler MaxCompilerMPT [3], OmpSs@FPGA [9], OpenStream [11], SDSoC or SDAccel [4] with OpenCL, and Vivado High Level Syntesis [5]. These frameworks are used to implement several key HPC applications across climate/weather, physics/energy and life-science/bioinformatics scientific domains. More details about the EuroEXA project can be obtained from its website: https://euroexa.eu.

In this work we describe our early steps towards the porting of our application within the EuroEXA Project using the OmpSs programming model. In preparation for the EuroEXA prototype, we are working on a Trenz TE8080 development board where we have developed our early implementations and performed preliminary performance measurements.

3. The Lattice Boltzmann Application

In this contribution we address CFD simulation applications based on the Lattice Boltzmann Method (LBM), a class of CFD solvers able to describe efficiently the physics of complex fluid flows, through a mesoscopic approach. LBM are stencil-based algorithms, discrete in space, time and momenta, operating on regular lattice grid. A set of synthetic pseudo-particles called *populations* are sitting at the edges of the lattice, and evolved for several time steps. At each time step, populations *propagate* from lattice-site to lattice-site, and then *collide* among each other updating their physical parameters. These two steps are the most compute intensive parts of actual LBM codes. In both rou-

[3] https://www.maxeler.com/solutions/low-latency/maxcompilermpt/
[4] https://www.xilinx.com/products/design-tools/all-programmable-abstractions.html
[5] https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

tines, there are no data dependencies between different lattice points, so they can execute in parallel on as many lattice sites as possible and according to the most convenient schedule. Boundary conditions, specific to each particular problem could be applied but have a truly minor computational impact on simulation time. A model labeled as *DxQy* describes a fluid in x dimensions using y populations per lattice point.

The specific model used in this work, the D2Q37, has been extensively used for convective turbulence studies [12], but has been also deeply optimized and used as a benchmarking application for several programming models and HPC hardware architectures. In the D2Q37 model the *propagate* function gather at each lattice site populations values from neighbors at distance up to 3 in the grid, generating a large number of sparse memory accesses and resulting to be strongly memory-intensive. The *collide* kernel performs ≈ 6600 double precision floating-point operations on data local to each lattice site, and is strongly compute-intensive with an arithmetic intensity greater than 10 [13]. Different implementations of this code have been designed and implemented, adopting several directive based languages to address both multi-core CPUs [14] and accelerators [8], such as GPUs [15] and also many-core devices [16].

4. The OmpSs Programming Model

In this work we have implemented our application using the *OmpSs* directives based programming model, developed at the BSC, and its *OmpSs@FPGA* extension [9].

OmpSs is very similar to the widely known OpenMP, and in fact it is a forerunner of OpenMP, where new features are introduced and developed before possibly getting pushed in the OpenMP standard. *OmpSs* is one of the tools selected to be used in the framework of the EuroEXA project to exploit FPGAs as accelerators, and allows to define task functions to be offloaded to such devices. It provides an automatic generation of a wrapper code handling data copies to and from the FPGA device, and manages flow dependencies and synchronizations. These data dependencies can be specified by the programmer using directives, as shown in Listing 1, where an example function is decorated with *pragmas* in order to be offloaded to an FPGA. Different buffers are set as input and outputs, giving also their sizes, in order to be copied in and out as needed. The function body to be actually offloaded onto FPGA, once processed by the *OmpSs* source to source toolchain, gets transformed into a bitstream by the VivadoHLS synthesis tool, allowing the programmer to add also proprietary HLS directives in the source code.

Thanks to the *OmpSs* directives, simply changing the offload target (which directly affect the final compiler to be used), the same source code can be compiled for several architectures, possibly targeting different accelerators. Interestingly enough, the use of *OmpSs* allows us also to exploit a wider set of tools developed at BSC, meant for performance analysis. In particular, we used Extrae [17], a tracing tool allowing to collect information during the execution of an application, such as: hardware counters; calls to MPI, OpenMP and *OmpSs* libraries; etc. To later utilize the acquired traces, we used also Paraver [18], a performance analysis tool which loading traces generated by Extrae provides a visual interface to analyze them. The traces can be displayed as timelines of the execution, as shown later, but can also be used to perform much more complex statistical analyses [19,20].

Listing 1: Example of a function performing the dot product operation, decorated with *OmpSs* directives, in order to be offloaded on a FPGA accelerator.

```
#pragma omp target device(fpga)
#pragma omp task in([BSIZE]v1, [BSIZE]v2) inout([1]result)
void dotProduct(float *v1, float *v2, float *result) {
  int resultLocal = result[0];
  for (size_t i = 0; i < BSIZE; ++i) {
    resultLocal += v1[i]*v2[i];
  }
  result[0] = resultLocal;
}
```

5. Implementation

Our application, when exploiting accelerators, is commonly implemented allocating the whole data domain into the device memory once for all, at the beginning of the simulation, and then performing several iterations of the algorithm following a double buffering approach [8]. Despite of this, when using FPGAs, the on-board memory is quite limited (although faster) with respect to other accelerators, thus we developed a blocking implementation, allowing to move slices of the whole lattice, to the FPGA device, in order to compute one slice at a time. To slice the lattice, gather and scatter operations are required in order to move just contiguous memory buffers, in and out from the FPGA BRAMs. The host part of this implementation is shown in Listing 2. Here we can see an outer loop over the iterations and an inner loop over different blocks of the lattice. Once the gather operation is completed on the host side, the *lbmBlocking* task function is called, which automatically handles the copy in and out of buffer arguments thanks to the OmpSs directives shown in Listing 3.

As described in Sec. 3, for each iteration, two different functions are commonly implemented: *propagate* and *collide*. In a first ported version, we directly implemented these two as inline functions, calling one after the other inside the *lbmBlocking* one. This requires a temporary intermediate buffer allocated in the FPGA's BRAMs which could be easily removed by merging the two function in a single one, saving about one third of the BRAM required.

Using proprietary VivadoHLS directives, as the ones shown in Listing 3, we have also optimized the placement of arrays in the BRAMs (using *HLS array_partition* directive), allowing for the concurrent access of multiple data items. Using *HLS pipeline* and *HLS unroll* directives we have been able also to achieve pipelining or unrolling of the loops performing the *collide* operation, increasing the performance as reported in Sec. 6.

On other parallel accelerators, such as GPUs, this application is commonly parallelized computing several lattice sites at the same time, exploiting the independence between the loops over the lattice sites [8]. On an FPGA this would translate to the unrolling and replication of the whole function body, which is not possible with the available resources on our development board. On the other side, to achieve a certain level of resources reuse, one may pipeline the execution over the lattice sites, allowing to start the computation of a new lattice site every few clock cycles. Unfortunately, at this stage, the resources of our target FPGA should be enough to pipeline the execution over the

Listing 2: Core of the LBM application showing the call to the function to be offloaded on the FPGA and computing one iteration on one block of the lattice.

```
for (i=0; i<NITER; i++) {
  for ( ix = HX; ix < HX+LX ; ix+=BCOL ) {

    // Gathering
    Bprv[...] = f1_soa[...];

    lbmBlocking( Bnxt, Bprv, param );
    #pragma omp taskwait

    // Scattering
    f2_soa[...] = Bnxt[...];

  } // Block loop
} // Iter loop
```

lattice sites, but the high routing congestion, due to the *collide* operation complexity, did not allowed us to produce a working bitstream.

From the portability point of view, interestingly enough, when compiling the application for architectures not using VivadoHLS, these directives are just ignored. In particular we compiled exactly the same code to run on the Arm cores of the Cortex A53 processor available in the same Trenz development board. Other directives could be added in the future, exploiting other directive based languages, to target other architectures.

6. Results

From the portability point of view, a first result is that we now have a single implementation able to be compiled for a multi-core CPU, just selecting *smp* as device target, or to offload the most time consuming part of our LBM application to FPGAs, selecting *fpga* as target device.

In this work, to test the application exploiting the FPGA offload on an actual hardware device, we have used a Trenz TE8080 development board [6], which hosts a Xilinx UltraScale+ ZU9 MPSoC. The FPGA in our Trenz board is much smaller, both in terms of resources and capabilities, wrt the one that will be used in the EuroEXA project, nevertheless, it is supported by *OmpSs* and it allows us to test and run our preliminary ported code and measure initial results. From the performance point of view, in fact, results are still very preliminary and even on this testbed a wide range of further optimization could be still explored. We report in Tab. 1 results measured with different implementations of our code showing the percentage of the ZU9 FPGA resources utilized, and the corresponding overall execution time divided by the lattice size (set to 256×256), giving the execution time required for each lattice site. We underline this is one of the handiness of using high level synthesis tools, which allow to easily test different implementations, possibly changing just *pragma* directives.

[6] https://shop.trenz-electronic.de/en/TE0808-04-9BE21-AS-TE0808-04-9BE21-AS-Starter-Kit

Listing 3: Sketch of the kernel function to be offloaded onto the FPGA, with *OmpSs* and HLS directives, corresponding to the last column implementation in Tab. 1.

```
#pragma omp target num_instances(1) device(fpga)
#pragma omp task out([BS]Bnxt) in([BH]Bprv, [PS]param)
void lbmBlocking(data_t Bnxt[BS],data_t Bprv[BH],data_t param[PS]) {

  #pragma HLS array_partition variable=param block factor=37

  for ( ix = 0; ix < BCOL; ix++) {
    for ( iy = HY; iy < (HY+LY); iy++) {

      #pragma HLS pipeline II=32
      #pragma HLS loop_flatten
      #pragma HLS expression_balance
      #pragma HLS dependence variable=Bnxt inter false

      data_t localPop[NPOP];

      #pragma HLS array_partition variable=localPop complete

      // PROPAGATE
      localPop[0] =  Bprv[              idxh - 3*NY + 1];
      localPop[1] =  Bprv[  1*popoffh + idxh - 3*NY    ];
      ...
      localPop[36] = Bprv[ 36*popoffh + idxh + 3*NY - 1];

      // COLLIDE
      for (p = 0; p < NPOP; p++) { Ops on localPop[] and param[] };
      ...
      for (p = 0; p < NPOP; p++) { Ops on localPop[] and param[] };

      Bnxt[] = ...;
      }
  }
```

The first version, on the leftmost column, refers to the original code annotated just with *OmpSs* directives, giving an execution time per site of $\approx 61\mu$sec. In the second version, we have added also HLS directives, pipelining or unrolling the inner loops involved in the *collide* operator, increasing the performance by a factor $\approx 5\times$, and reducing the time per lattice-site to $\approx 12.6\mu$sec In a third version we attempted to optimize for resources, merging the *propagate* and *collide* functions, and applying just the pipelining of the loops in the *collide* region. Moreover we also partitioned an array of constant parameters, splitting its content across different BRAMs, to allow to access different data items during the same clock cycle. As shown in Tab. 1, this results in a reduction of resources usage, especially for DSPs and BRAMs, without a negative impact on the time per site, which is even slightly better. In the last version we attempted to pipeline over the lattice sites (i.e., the outer loops). In this case, as reported by Vivado, the computing resources of the ZU9 FPGA should be enough to fit the design, if the *Initiation Interval* (`II`) – corresponding to the number of clock cycles to wait before filling in the pipeline a new lattice site – is kept greater or equal to 32, as shown in Listing 3. Anyhow, unfortunately,

Table 1. FPGA resources utilization and achieved overall execution time per lattice site at 200MHz, for different implementations of the offloaded function.

	First version	**Pipeline/Unroll Collide loops**	**Merged Funcs and Partition**	**Pipeline over sites**
DSP48E	2.8%	16.9%	4.8%	20.83%
BRAM_18K	29.2%	35.7%	11.4%	31.74%
LUT	9.1%	34.7%	38.28%	68.17%
FF	5.5%	15.1%	12.44%	58.66%
Exec. Time per lattice site	$60.62\mu s$	$12.65\mu s$	$12.4\mu s$	$\approx 0.16\mu s$ (Estimated)

the high amount of computing resources involved and the code complexity, result in high routing congestion levels and consequently in a failure of the final bitstream synthesis. Further increasing the `II` value allows to reduce the computing resources usage, but the routing congestion still impairs the synthesis, unless using `II` values in the same order of the pipeline depth. Neglecting routing issues, taking into account the FPGA clock frequency – set to 200MHz – and the minimum `II` value that allows to fit FPGA computing resources, we can estimate an execution time of $\approx 0.16\mu$sec per lattice site. This corresponds to a performance speed-up of about one order of magnitude. This is relevant in prospective, giving us an estimation of what we could expect to obtain e.g. on the larger Xilinx VU9 UltraScale+ FPGA, which could be adopted by the EuroEXA project.

To make a point of reference, we can compare the results achieved with the ones measured on a processor with a similar power envelop. In particular, running the same code on the Arm Cortex A53 embedded in the same MPSoC, we measure a value of $9.26\mu s$ per lattice site. This result is very close to the one measured on the ZU9 FPGA, but as already highlighted, using the larger FPGA available in the EuroEXA computing nodes, we expect to significantly increase this performance value.

Another interesting result is that *OmpSs* can be combined with a set of performance profiling tools, such as Extrae and Paraver. Extrae allows to collect execution traces that can be then visualize using Paraver. In Fig. 1 we show the execution traces of several launches of our code, corresponding to the different slices (or blocks) of the lattice. running on the Trenz board. We clearly see 5 different timelines, one for each core of the Arm CPU and one for the FPGA. The threads (in green) spawns the tasks (in red) which offload the *lbmBlocking* kernel that is executed on FPGA (in blue).

7. Conclusion and future works

Using *OmpSs* programming model and *OmpSs@FPGA* extension, after the initial setup of a working tool-chain involving OmpSs, Xilinx VivadoHLS SDK and Xilinx Petalinux (to generate bootable images for the Trenz board), we have been able, with minimal code modifications, to allow an actual HPC Lattice Boltzmann application to exploit a Xilinx FPGA as an accelerator. Interestingly, the same code can be compiled targeting different architectures, such as x86 and Arm multi-core CPUs.

Adding proprietary HLS directives, we have been able to increase the performance by $5\times$ wrt the initial version, without introducing major changes to the actual C code,

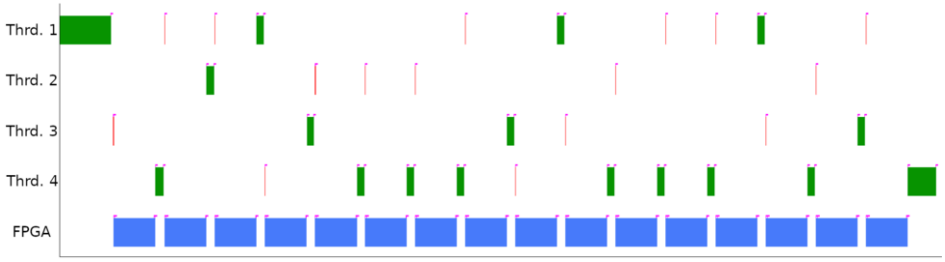

Figure 1. Paraver view of the execution timeline. The 4 top rows represent the 4 Arm cores performing the lattice initialization and the gather/scatter operations (Green) and managing the function offload to the FPGA (Red). In the bottom row is represented the execution timeline in the FPGA, each Blue box represent the fused Propagate-Collide computation on one lattice block.

starting from the original implementation of the two main functions of the LBM algorithm. With some modifications (i.e., merging *propagate* and *collide*) we were able to save some BRAM memory, allowing to increase the lattice slice we could process. On the Xilinx ZU9 FPGA we have used in this work the result achieved is similar to that measured running the code on the Arm cores of the same MPSoC; however we have estimated that using a larger FPGA, e.g the Xilinx VU9, we can speed-up the execution time by at least one order of magnitude. In particular, on the VU9 we expect to be able to pipeline over the lattice sites, thanks to the increased routing resources, and to reduce the minimum `II`, thanks to the higher amount of computing resources.

As future works, we plan to re-organize the loops involved in the *collide* kernel, to help to parallelize reduction operations involved in the inner loops. On the host side we aim to avoid gathering and scattering operations and succeed to overlap data transfers with computations. In particular, we aim to develop a multi-FPGA implementation, keeping one slice of lattice stored on each FPGA (e.g., in the VU9 on-board UltraRAM will be available), and moving back and forth from the host-DRAM only the lattice-slice halos for communications with neighbours. We also expect to work soon on a Xilinx VU9 in order to verify our estimations.

Acknowledgments: E.C. has been supported by the European Union's H2020 research and innovation programme under EuroEXA grant agreement No. 754337. This work has been done in the framework of the EuroEXA EU project and of the COKA, and COSA, INFN projects.

References

[1] Güneysu, T., Kasper, T., Novotný, M., Paar, C., Rupp, A.: Cryptanalysis with copacobana. IEEE Transactions on Computers 57(11), 1498–1513 (Nov 2008), doi:10.1109/TC.2008.80

[2] Wienbrandt, L.: Bioinformatics Applications on the FPGA-Based High-Performance Computer RIVYERA, pp. 81–103. Springer New York, New York, NY (2013), doi:10.1007/978-1-4614-1791-0_3

[3] Fröning, H.: Extoll and data movements in heterogeneous computing environments. In: Resch, M.M., Bez, W., Focht, E., Kobayashi, H., Patel, N. (eds.) Sustained Simulation Performance 2014. pp. 127–139. Springer International Publishing, Cham (2015), doi:10.1007/978-3-319-10626-7_11

[4] Belletti, F., Guidetti, M., Maiorano, A., Mantovani, F., Schifano, S., Tripiccione, R., Cotallo, M., Perez-Gaviro, S., Sciretti, D., Velasco, J., Cruz, A., Navarro, D., Tarancon, A., Fernandez, L., Martin-Mayor, V., Munoz-Sudupe, A., Yllanes, D., Gordillo-Guerrero, A., Ruiz-Lorenzo, J., Marinari, E., Parisi, G.,

Rossi, M., Zanier, G.: Janus: An FPGA-based system for high-performance scientific computing. Computing in Science and Engineering 11(1), 48–58 (2009), doi:10.1109/MCSE.2009.11

[5] Baity-Jesi, M., Baños, R., Cruz, A., Fernandez, L., Gil-Narvion, J., Gordillo-Guerrero, A., Iñiguez, D., Maiorano, A., Mantovani, F., Marinari, E., Martin-Mayor, V., Monforte-Garcia, J., Sudupe, A.M., Navarro, D., Parisi, G., Perez-Gaviro, S., Pivanti, M., Ricci-Tersenghi, F., Ruiz-Lorenzo, J., Schifano, S.F., Seoane, B., Tarancon, A., Tripiccione, R., Yllanes, D.: Janus II: A new generation application-driven computer for spin-system simulations. Computer Physics Communications 185(2), 550 – 559 (2014), doi:10.1016/j.cpc.2013.10.019

[6] Vanderbauwhede, W., Benkrid, K.: High-performance computing using FPGAs, vol. 3. Springer (2013), doi:10.1007/978-1-4614-1791-0

[7] Nane, R., Sima, V., Pilato, C., Choi, J., Fort, B., Canis, A., Chen, Y.T., Hsiao, H., Brown, S., Ferrandi, F., Anderson, J., Bertels, K.: A Survey and Evaluation of FPGA High-Level Synthesis Tools. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 35(10), 1591–1604 (Oct 2016), doi:10.1109/TCAD.2015.2513673

[8] Calore, E., Gabbana, A., Kraus, J., Pellegrini, E., Schifano, S.F., Tripiccione, R.: Massively parallel lattice-Boltzmann codes on large GPU clusters. Parallel Computing 58, 1 – 24 (2016), doi:10.1016/j.parco.2016.08.005

[9] Filgueras, A., Gil, E., Alvarez, C., Jimenez, D., Martorell, X., Langer, J., Noguera, J.: Heterogeneous tasking on SMP/FPGA SoCs: The case of OmpSs and the Zynq. In: 2013 IFIP/IEEE 21st International Conference on Very Large Scale Integration (VLSI-SoC). pp. 290–291 (Oct 2013), doi:10.1109/VLSI-SoC.2013.6673293

[10] Bosch, J., Filgueras, A., Vidal, M., Jimenez-Gonzalez, D., Alvarez, C., Martorell, X.: Exploiting parallelism on GPUs and FPGAs with OmpSs. In: Proceedings of the 1st Workshop on AutotuniNg and aDaptivity AppRoaches for Energy efficient HPC Systems. p. 4. ACM (2017), doi:10.1145/3152821.3152880

[11] Pop, A., Cohen, A.: Openstream: Expressiveness and data-flow compilation of openmp streaming programs. ACM Transactions on Architecture and Code Optimization (TACO) 9(4), 53 (2013), doi:10.1145/2400682.2400712

[12] Biferale, L., Mantovani, F., Sbragaglia, M., Scagliarini, A., Toschi, F., Tripiccione, R.: Second-order closure in stratified turbulence: Simulations and modeling of bulk and entrainment regions. Physical Review E 84(1), 016305 (2011), doi:10.1103/PhysRevE.84.016305

[13] Calore, E., Gabbana, A., Schifano, S.F., Tripiccione, R.: Evaluation of DVFS techniques on modern HPC processors and accelerators for energy-aware applications. Concurrency and Computation: Practice and Experience 29(12), 1–19 (2017), doi:10.1002/cpe.4143

[14] Mantovani, F., Pivanti, M., Schifano, S.F., Tripiccione, R.: Performance issues on many-core processors: A D2Q37 Lattice Boltzmann scheme as a test-case. Computers & Fluids 88, 743 – 752 (2013), doi:10.1016/j.compfluid.2013.05.014

[15] Calore, E., Gabbana, A., Kraus, J., Schifano, S.F., Tripiccione, R.: Performance and portability of accelerated lattice Boltzmann applications with OpenACC. Concurrency and Computation: Practice and Experience 28(12), 3485–3502 (2016), doi:10.1002/cpe.3862

[16] Calore, E., Gabbana, A., Schifano, S.F., Tripiccione, R.: Early experience on using knights landing processors for lattice boltzmann applications. In: Parallel Processing and Applied Mathematics: 12th International Conference, PPAM 2017, Lublin, Poland, September 10-13, 2017. Lecture Notes in Computer Science, vol. 1077, pp. 1–12 (2018), doi:10.1007/978-3-319-78024-5_45

[17] Servat, H., Llort, G., Huck, K., Giménez, J., Labarta, J.: Framework for a productive performance optimization. Parallel Computing 39(8), 336–353 (2013), doi:10.1016/j.parco.2013.05.004

[18] Pillet, V., Labarta, J., Cortes, T., Girona, S.: Paraver: A tool to visualize and analyze parallel code. In: Proceedings of WoTUG-18: transputer and occam developments. vol. 44, pp. 17–31 (1995)

[19] Mantovani, F., Calore, E.: Multi-node advanced performance and power analysis with paraver. In: Parallel Computing is Everywhere. Advances in Parallel Computing, vol. 32, pp. 723–732 (2018), doi:10.3233/978-1-61499-843-3-723

[20] Calore, E., Mantovani, F., Ruiz, D.: Advanced Performance Analysis of HPC Workloads on Cavium ThunderX. In: 2018 International Conference on High Performance Computing Simulation (HPCS). pp. 375–382 (July 2018), doi:10.1109/HPCS.2018.00068