Non-Uniform Partitioning for Collaborative Execution on Heterogeneous Architectures

Gabriel Freytag[1], Matheus S. Serpa[1], João V. F. Lima[2], Paolo Rech[1], Philippe O. A. Navaux[1]
[1]Institute of Informatics, Federal University of Rio Grande do Sul (UFRGS) — Porto Alegre, Brazil
[2]Federal University of Santa Maria (UFSM) — Santa Maria, Brazil
{gfreytag, msserpa, prech, navaux}@inf.ufrgs.br, jvlima@inf.ufsm.br

***Abstract*—Since the demand for computing power increases, new architectures arise to obtain better performance. An important class of integrated devices is heterogeneous architectures, which join different specialized hardware into a single chip, composing a System on Chip - SoC. Within this context, effectively splitting tasks between the different architectures is primal to obtain efficiency and performance. In this work, we evaluate two heterogeneous architectures: one composed of a general-purpose CPU and a graphics processing unit (GPU) integrated into a single chip (AMD Kaveri SoC), and another composed by a general-purpose CPU and a Field Programmable Gate Array (FPGA) integrated into a single chip (Intel Arria 10 SoC). We investigate how data partitioning affects the performance of each device in a collaborative execution through the decomposition of the data domain. As a case study, we apply the technique in the well-known Lattice Boltzmann Method (LBM), analyzing the performance of five kernels in both architectures. Our experimental results show that non-uniform partitioning improves LBM kernels performance by up to 11.40% and 15.15% on AMD Kaveri and Intel Arria 10, respectively.**

***Index Terms*—Heterogeneous Architectures, Collaborative Execution, Non-Uniform Partitioning, FPGA, GPU, Lattice Boltzmann Method**

I. Introduction

The computational power currently available in high-performance computing systems makes it possible to perform extremely complex tasks in a few hours, minutes, seconds, or even in real-time. Such computational power usually results from the combination of a large number of devices of different architectures and, consequently, with different computational powers (such as CPU, GPU, and FPGA) [1], [2]. However, as the devices that make up these systems evolve and become more and more powerful, they require an increasing amount of energy that combined generates massive power consumption.

Despite the increasing power consumption of devices that make up high-performance computing systems, there are some devices whose power consumption is significantly lower. Devices with ARM or FPGA architectures typically have significantly lower power consumption than CPU and GPU devices, for instance, [3]. However, low-power devices usually also have lower computational power.

Although the heterogeneity of devices in high-performance computing systems is no longer a novelty, in the last few years some heterogeneous architectures have appeared that integrate into a single processor chip multiple architectures. By replacing external connections between different devices (usually PCIe connections) by significantly shorter and faster internal connections to the chip, it becomes possible to share the same memory space and, consequently, to eliminate data transfers between different memory addresses. These optimizations, together with a reduced area for the manufacture of multiple architectures where previously only one would be fabricated, make them low-power architectures [4], [5].

Some examples of currently available heterogeneous architectures are AMD Kaveri SoC [6], that integrates x86-64 CPU and Radeon R7 GPU processing units in the same chip, Intel Arria 10 SoC [7], that integrates ARM CPUs and an FPGA, and Xilinx Zynq SoC [8], that integrates ARM CPUs and an FPGA in the same chip. Although they are architectures with lower computational power, as in high-performance computing systems, to unlock its maximum computing power, it is necessary to distribute the workload of tasks between the devices available in the architecture in order to compute them collaboratively. However, since devices typically have different computing capabilities, it is necessary to find the optimal division of workload between the different devices.

In this paper, we investigate the performance impact of collaborative execution on two System-on-chip devices based in CPU+GPU and CPU+FPGA architectures using non-uniform data partitioning for each device. Our case of study is the five distinct kernels from the Lattice Boltzmann Method (LBM) in order to evaluate individual and collaborative execution performance of heterogeneous architectures. The main contributions of this paper are:

- We evaluate the performance of two heterogeneous architectures: AMD Kaveri SoC and Intel Arria 10 SoC;
- We analyze the performance of each device present in both architectures individually and in a collaborative way;
- We show that non-uniform partitioning on collaborative execution improves performance of LBM kernels;
- We present an OpenMP + OpenCL D3Q19 Lattice Boltzmann implementation for heterogeneous architectures.

The remainder of this paper are organized as follows. Section II discusses related work. Section III presents several details about the Lattice-Boltzmann Method (LBM) and Section IV describes our parallel implementation for two low-power heterogeneous platforms. Our experimental results are presented in Section V. Section VI presents the discussion, and finally Section VII presents conclusions and future work.

II. Related Work

Many of the related works used GPUs or FPGAs as an offload target to accelerate applications. The authors in [9] evaluated performance and power consumption of kernel computations on an Arria 10 FPGA over an Intel Xeon Phi KNL and an NVIDIA Tesla K80. In [10], the authors propose a novel approach that automatically optimizes task partitioning for different inputs and architectures. In [11] the authors evaluate the reliability behavior of AMD Kaveri aiming to find which configuration provides the lowest error rate or allows the computation of the highest amount of data before experiencing a failure. In [12], [13], [14] the authors accelerated deep learning networks using OpenCL and FPGAs. In [15] they designed computational intensive kernels of a tsunami simulator on FPGAs and GPUs using OpenCL. The authors in [16] evaluated the Rodinia benchmark over GPUs and FPGAs with OpenCL. In [17], [18] they combined spatial and temporal blocking to evaluate performance and power efficiency of stencil computations on FPGAs. In [19] they optimized geophysics models on GPUs. Regarding LBM kernels, the authors in [20] optimized for GPUs and in [21] they studied optimization strategies for accelerators such as GPUs and Intel Xeon Phi KNL. In [22] a memory-aware 2D LBM was implemented for Intel Xeon manycore processors.

Several authors have studied collaborative processing on heterogeneous devices. Hetero-Mark [23] is a benchmark suite for collaborative processing for CPU-GPU architectures with support to OpenCL. They analyzed experimental results on an AMD A10-7850K APU. Chai [24] is also a collaborative processing benchmark suite for integrated devices. It compared task and data partitioning strategies and has support to FPGAs and GPUs. In [25], they evaluated two FPGA systems with Chai benchmark and analyzed the task and data partitioning. The authors in [26] designed binarized neural networks to take advantage of FPGA systems, and compared performance and energy consumption over an NVIDIA Titan X GPU. In [27] they optimized a Binarized Neural Network and evaluated its performance in an Intel Xeon E5-2699v3 CPU, an NVIDIA GTX Titan X GPU, an Intel Stratix V and an Intel Arria 10 FPGA and a custom ASIC platform without using OpenCL architecture. The performance analysis of collaborative computing in two heterogeneous integrated systems using OpenCL was presented by [28]. They evaluated an AMD A10-7850K platform with CPU cores and GPU computing units integrated into the same chip and an E3-1240 v3 CPU chip connected through PCIe to an Intel Stratix V GX FPGA on a Terasic DE5-Net board. The results showed that in both platforms, the use of the two available devices in a collaborative way led to better performance compared to use CPU only or GPU/FPGA only.

Most of the works previously presented evaluate the performance of the different processing units individually. However, the use of these devices in collaborative processing can lead to significant performance improvements, as previously demonstrated by [28].

III. Use-case Application

A. Lattice-Boltzmann Method

The Lattice Boltzmann Method (LBM) is a numerical method for fluid flow simulations, and fluid physics modeling originated from discrete particle kinetics called Lattice Gas Automaton (LGA). The Lattice Gas Automaton is constructed as a simplified, fictitious molecular dynamic in which space, time, and particle velocities are all discrete [29]. Thus, in LBM space, time and velocity are also discrete.

Lattice-Boltzmann Method is often adopted as an alternative technique for computational simulations of Fluid Dynamics instead of conventional numerical schemes based on discretizations of macroscopic continuum equations as discrete Navier-Stokes equation solvers [29]. In LBM, a lattice is formed by discrete points, each with a fixed number of discrete displacement directions on which particles perform spatial displacements at each iteration, enabling simple simulations of physical properties of fluid flows [22].

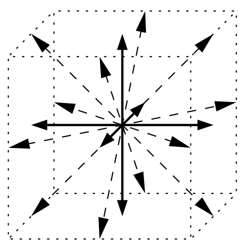
Fig. 1. D3Q19 lattice geometry.

In this paper, we used a three-dimensional lattice structure with nineteen propagation directions, as shown in Figure 1 and defined below [30]:

- A static point at coordinate $(0, 0, 0)$, where the particle has zero velocity. The value of ω_i in this case is $1/3$.
- Six nearest directions $(-1, 0, 0)$, $(+1, 0, 0)$, $(0, -1, 0)$, $(0, +1, 0)$, $(0, 0, -1)$ and $(0, 0, +1)$, with unity velocity and $\omega_i = 1/18$.
- Twelve diagonal line neighbors $(1, 1, 0)$, $(-1, 1, 0)$, $(1, -1, 0)$, $(-1, -1, 0)$, $(1, 0, 1)$, $(-1, 0, 1)$, $(1, 0, -1)$, $(-1, 0, -1)$, $(0, 1, 1)$, $(0, -1, 1)$, $(0, 1, -1)$ and $(0, -1, -1)$, with velocity $\sqrt{2}$ and $\omega_i = 1/36$.

To deal with the collisions against the boundaries of the structure we use a mechanism called Bounce-back which consists in the inversion of the speed vectors directions each time that a collision occurs against a static point preventing the forces leaving, returning them to the fluid [31]. For the experiments, we use a rectangular obstacle placed in the canal at the first third of the x-axis, as illustrated in Figure 2.

B. Domain Decomposition

To be able to use multiple devices simultaneously to compute the fluid dynamics of the Lattice-Boltzmann method in

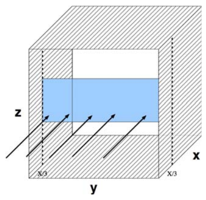

Fig. 2. Rectangular obstacle of the experiments.

parallel, we divide its domain into subdomains by dividing the original three-dimensional domain in the z axis. In this way, each device can apply the kernels of the method in parallel on each subdomain. However, dividing the domain into multiple subdomains leads to inconsistencies in macroscopic values of the fluid due to dependencies on the nineteen propagation directions of the neighbor's particles of each particle of the fluid according to the Figure 1. Therefore, we use ghost zones to deal with the inconsistencies arising from the division of the domain solving the problem completely.

Since each particle of the fluid has nineteen directions of propagation of the forces in the model D3Q19, as shown in Figure 1, as the fluid flows the forces of eighteen propagation directions of each particle are propagated through the neighboring particles in the fluid. With the division of the domain into subdomains, the existence of this data dependence makes it necessary, for particles located at the edges of each subdomain, to access data located in neighboring subdomains. However, the access to data located in other subdomains being manipulated by other devices can lead to concurrency in data access and consequently in inconsistencies. Therefore, to deal with these issues, we use ghost zones to keep a copy of the edges of the neighboring subdomains of each subdomain.

Beyond the division of the domain into subdomains to parallelize the routines of the method, the routines themselves were also divided into kernels. Each kernel has data dependencies which must be obeyed to ensure the consistency of the results of the method. In this manner, there is an order in which the kernels need to be executed by each device to simulate the flow of the fluid in its subdomain correctly. Besides that, when multiple devices are used collaboratively, the ghost zones of the neighbor subdomains of each device need to be updated. This ghost zone updates introduce a synchronization point in parallel execution of the method when using multiple devices.

Therefore, to deal with these issues, we use ghost zones to keep a copy of the edges of the neighboring subdomains of each subdomain. Figure 3 shows an illustration of the procedures for updating ghost zones of two subdomains in two devices. In this work, as in [32], we decompose the domain of the Lattice-Boltzmann method only in one dimension, in this case in dimension z. Besides that, as we use at most two devices to compute the method and use a circular strategy to simulate the flow of fluids in the method, meaning that the particles leaving one side of the domain return on the opposite side in order to preserve the macroscopic values of the fluid, there are only four ghost zones that correspond to the four faces at the z dimension of the subdomains. These four faces are first copied to a temporary buffer by each device, and then each device copies the adjacent faces from the temporary buffers to its local copy of the z dimension faces of it neighboring subdomain. As can be seen in Figure 3, the faces where the domain is divided are crossing copied, and the faces at the edges of the original domain are copied to the opposite side of the subdomains.

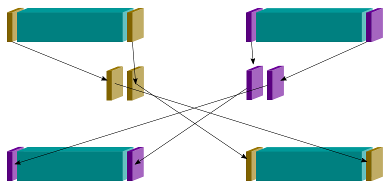

Fig. 3. LBM implementation with ghost zones.

C. Lattice-Boltzmann Kernels

The Lattice-Boltzmann method routines are composed of five well-defined kernels. The first kernel, called INITIALIZE, assigns an initial macroscopic value for each of the nineteen propagation directions of each particle of the fluid in the three-dimensional model used in our work. After initializing, the second kernel to be executed redistributes the forces of some of the propagation directions of each particle of the fluid and, therefore, is called *redistribute*. The third kernel, called PROPAGATE, is in charge of propagating the forces of the particles according to the flow of the fluid. After the propagation, the fourth kernel deals with the collision of the particles in the fluid with the barriers present in the domain, as described in Section III-A, and is called BOUNCEBACK. The fifth and last kernel relaxes the forces of each of the propagation directions of each particle in the fluid, and it is called RELAXATION.

```
1: Initialize parameters
2: for Each $N_x \times N_y \times N_z$ do
3:     INITIALIZE conditions
4: end for
5: for Each time step do
6:     for Each $N_x \times N_y \times N_z$ do
7:         REDISTRIBUTE
8:         PROPAGATE
9:         BOUNCEBACK
10:        RELAXATION
11:    end for
12: end for
```

Fig. 4. D3Q19 Lattice Boltzmann Method Algorithm

These five kernels must always be executed in the same sequence as described above. After the initialization of the macroscopic values in kernel INITIALIZE, the remaining four kernels need to be executed sequentially *t_max* times, as shown in Figure 4. When two devices are used, after the execution of the *redistribute* kernel, each device copies its edge faces to the temporary buffers and wait for both devices to finish. Then, each device copies the neighboring subdomain edge faces from the temporary buffers to its local copies and execute the remaining three kernels, and this cycle repeats *t_max - 1* times.

IV. Collaborative LBM Implementation

In this section, we describe the collaborative implementation of the Lattice-Boltzmann method developed for two low-power heterogeneous platforms: One is four CPU cores and eight GPU computing units integrated into a single chip, named *AMD Kaveri*; and two CPU cores and an FPGA with 660,000 logic blocks integrated in the same chip, named *Intel Arria 10*. In order to use both sets of processing units available in each platform in a collaborative way, we decompose the method's data domain into two subdomains. Besides that, to be able to evaluate the performance of each kernel of the method on each platform and find the domain decomposition that optimizes the performance of the method, we implement a variable domain decomposition to enable the assignment of different subdomain sizes for each set of processing units on each platform.

As both platforms have support for the OpenCL architecture, we can use the OpenCL language to implement our Lattice-Boltzmann method and use their heterogeneous processing units. However, while the AMD Kaveri platform is fully OpenCL capable, meaning that both CPU and GPU are OpenCL devices and, in this way, can run OpenCL code, the Intel Arria 10 is only partially OpenCL capable and only the FPGA can run OpenCL code. In this way, we implement the parallel heterogeneous version of the Lattice-Boltzmann method using the OpenMP Application Programming Interface (API) to run the method's kernels in the CPU cores of both platforms and the OpenCL language to run the method's kernels in the GPU and FPGA processing units of the AMD Kaveri and Intel Arria 10 platforms, respectively.

```
__kernel void redistribute(...)
   x = get_global_id(0);
   y = get_global_id(1);
   z = get_global_id(2);
   /* redistribute computation */
```

Fig. 5. OpenCL kernel code snippet.

In OpenCL language, each of the Lattice-Boltzmann kernels become OpenCL kernels as in Figure 5. The keyword *__kernel* represents OpenCL kernels. Function $get\_global\_id(dim)$ returns the unique global work-item ID value for dimension identified by dim. Initially, to be able to execute kernels in the devices of the OpenCL platform, it is needed to create an OpenCL context using the IDs of the devices. After, these kernels can be queued in OpenCL queues with First In First Out (FIFO) type data structure from which they are then delivered for the devices associated with the queue by the OpenCL runtime. Each queue can be associated with one or more OpenCL devices, and in our OpenCL implementation of the Lattice-Boltzmann method, we use an OpenCL queue for GPU / FPGA device.

```
void relaxation(...)
   #pragma omp parallel for collapse(3)
   for(x = 0; x < lx; x++)
     for(y = 0; y < ly; y++)
       for(z = 0; z < lz; z++)
        /* relaxation computation */
```

Fig. 6. OpenMP kernel code snippet.

In the CPU, however, as it is not an OpenCL device, we use the OpenMP API to parallelize the kernels of the method. Figure 5 shows a code snippet for the OpenMP implementation. As the kernels go through all particles in the three-dimensional domain, there are three nested for loops on which we use OpenMP for loop pragmas to parallelize the kernels in CPU. To achieve better performance, we collapse the three nested for loops using the OpenMP pragma *#pragma omp parallel for collapse(3)* on all kernels.

In this implementation, a kernel is queued in the FPGA device queue, and then the corresponding kernel is executed in the CPU. Only after executing this kernel in the CPU, we can queue the next kernel in FPGA device queue and then execute the corresponding kernel in the CPU. Thus, after queuing the *redistribute* kernel and executing the corresponding kernel in the CPU, queues a copy of the edge faces of the FPGA subdomain and copies its subdomain edge faces to the temporary buffers. After copied, the CPU waits for the copy of the FPGA subdomain edge faces and then copies it from the temporary buffer to its local copy, and the remaining kernels and kernels are executed. This cycle repeats *t_max - 1* times.

V. Experimental Results

This Section details the heterogeneous platforms, the methodology used in our experiments, and the results.

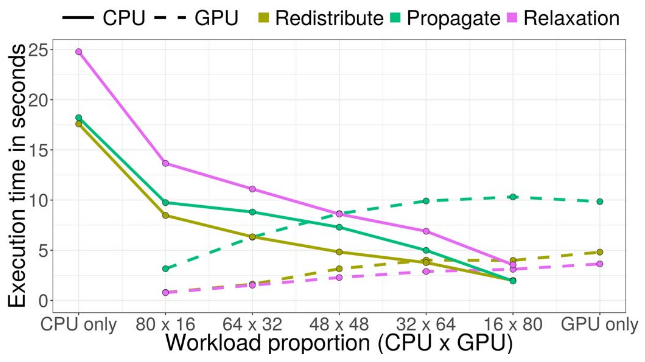

Fig. 7. Average execution time of the three most representative kernels on the CPU and GPU devices of the AMD Kaveri platform on each workload proportion using the OpenMP + OpenCL version with a domain of size $96 \times 96 \times 96$.

TABLE I
AVERAGE EXECUTION TIME OF ALL KERNELS ON THE CPU AND GPU DEVICES OF THE AMD KAVERI PLATFORM ON EACH WORKLOAD PROPORTION USING THE OPENMP + OPENCL VERSION WITH A DOMAIN OF SIZE $96 \times 96 \times 96$.

		80 x 16		**64 x 32**		**48 x 48**		**32 x 64**		**16 x 80**		
Kernel	**CPU only**	**CPU**	**GPU**	**CPU**	**GPU**	**CPU**	**GPU**	**CPU**	**GPU**	**CPU**	**GPU**	**GPU only**
Initialize	0.059	0.049	0.002	0.040	0.004	0.030	0.006	0.022	0.010	0.013	0.012	**0.006**
Redistribute	17.588	8.469	0.796	6.342	1.616	4.827	3.153	3.766	4.012	**1.987**	**3.989**	4.811
Propagate	18.212	9.749	3.157	8.811	6.307	**7.294**	**8.65**	4.99	9.908	1.934	10.313	9.846
Bounceback	1.544	0.925	0.024	0.732	0.048	0.593	0.076	0.445	0.102	0.227	0.122	**0.174**
Relaxation	24.785	13.659	0.772	11.104	1.518	8.603	2.279	6.895	2.884	**3.549**	**3.105**	3.636

A. *Platforms and Experimental Design*

We used the following two platforms. The first platform, called A10-7870K consists of 4 x86-64 CPU cores and, 8 Radeon R7 GPU computing units integrated on the same chip and is made by AMD. It has 6 GB of RAM, Ubuntu 14.04 (trusty) operating system, OpenCL Software Development Kit (SDK) version AMD APP SDK-3.0 and Clang C/C++ compiler version 7.0. The second platform, called A10SoCFPGA, consists of 2 ARM CPU cores and an FPGA with 660,000 logic blocks integrated into a single chip and made by Intel. It has 1GB of RAM, Angstrom 2014.10 operating system, Intel FPGA Runtime Environment (RTE) for OpenCL version 18.1 and ARM GCC C/C++ compiler version 4.7. From now, we will call the platforms AMD Kaveri and Intel Arria 10. Table II summarizes the environments.

TABLE II
CONFIGURATION OF EVALUATED PLATFORMS.

Platform	Parameter	Value
AMD Kaveri		
	CPU	AMD A10-7870K, 4 cores
	GPU	AMD Radeon R7, 8 cores
	Memory	6GB DDR3-2133
Intel Arria 10		
	CPU	ARM Cortex-A9, 2 cores
	FPGA	660 000 logic blocks
	Memory	1GB

Both architectures support the OpenMP [33] and OpenCL language [34]. However, only AMD Kaveri supports OpenCL on its two devices. In Arria 10 only the FPGA supports OpenCL code, which is converted automatically using a High-Level Synthesis compiler whose resulting binary is then used to program the FPGA by the host (CPU) through the OpenCL language. Thus, in both Arria 10 and AMD Kaveri, we evaluate the performance of the method in an OpenMP + OpenCL implementation (OpenMP in CPU and OpenCL in GPU / FPGA). The optimization flag used in both was *-O3*.

We performed experiments in both AMD Kaveri and Intel Arria 10 platforms using modified domain decomposition. The modified domain was decomposed in two subdomains, one for each device. The proportion of data assigned to each device varies from 0 to 96. 0 means that the kernel is executed individually in one device (CPU, GPU, or FPGA). For the collaborative execution, we decomposed the domain in z non-uniformly between the resources. We started from a subdomain of size 16 for the CPU to a subdomain of size 80, increasing by 16 the CPU proportion and consequently decreasing the GPU or FPGA proportion. For example, a domain proportion of 33.3% of a domain of size $96 \times 96 \times 96$ for the CPU and the remaining 66.7% of this domain for the FPGA corresponds to a subdomain of size $96 \times 96 \times 32$ being processed by the CPU and a subdomain of size $96 \times 96 \times 64$ being processed by the FPGA.

We measured the execution time of each device resource running the Lattice Boltzmann Method with different workload proportions. Besides, each experiment was executed at least ten times, with a confidence interval of 95% calculated with the Students t-distribution.

B. *Results*

In this section, we present the performance results of our collaborative execution strategy over the AMD Kaveri and Intel Arria 10 SoCs. Our case of study was the computing kernels of the LBM D3Q19 application. We performed experiments using CPU only, GPU only and CPU + GPU through a three-dimensional domain of size $96 \times 96 \times 96$. The domain size is limited by the Intel Arria memory which is 1 GB.

Table I presents the average execution time of the five kernels of the method in the OpenMP + OpenCL version on the AMD Kaveri platform. The first column is the kernels name. The following columns are different decomposition sizes of the method domain, from CPU only to GPU only. The bold cells are the ones that performed better, and consequently, had the best partitioning for each kernel. INITIALIZE and BOUNCEBACK kernels performed better in GPU only execution. Collaborative execution did not improve the performance of these kernels because their execution time was too short. The REDISTRIBUTE and PROPAGATE kernels performance was improved by 12.09% and 12.15%, respectively. The RELAXATION kernel had a low-performance improvement of 2.39%. Using CPU only the total execution time was 62.19 seconds, and using GPU only the total execution time was 18.47 seconds. However, collaboratively using both CPU and GPU and dividing the domain into five distinct proportions (GPU only, 16×80, 48×48, GPU only and 16×80), the

TABLE III
AVERAGE EXECUTION TIME OF THE ALL KERNELS ON THE CPU AND FPGA DEVICES OF THE INTEL ARRIA 10 PLATFORM ON EACH WORKLOAD PROPORTION USING THE OPENMP + OPENCL VERSION WITH A DOMAIN OF SIZE 96 × 96 × 96.

		80 x 16		64 x 32		48 x 48		32 x 64		16 x 80		
Kernel	**CPU only**	**CPU**	**FPGA**	**CPU**	**FPGA**	**CPU**	**FPGA**	**CPU**	**FPGA**	**CPU**	**FPGA**	**FPGA only**
Initialize	0.277	0.247	0.025	0.209	0.052	0.177	0.081	0.155	0.110	0.140	0.127	**0.139**
Redistribute	83.207	73.151	6.303	62.525	13.262	47.550	20.329	25.767	19.680	**14.409**	**24.666**	29.312
Propagate	146.685	117.726	19.007	93.066	38.473	73.195	57.700	**47.511**	**71.158**	22.480	84.810	83.020
Bounceback	14.754	12.636	0.379	10.385	0.629	6.06	0.9	2.028	1.222	**0.954**	**1.367**	1.691
Relaxation	205.069	175.321	13.223	142.31	27.726	107.32	42.587	67.172	48.677	**34.67**	**60.688**	72.078

shortest execution time achieved was 16.37 seconds using a non-uniform partitioning. Thus, non-uniform partitioning technique improves the entire method performance by 11.39%.

Moreover, looking more closely to Figure 7, where we show the execution times of each kernel on the seven different workload proportions, it is possible to observe that the shorter execution times of each kernel were not necessarily obtained with the same domain decomposition size. In this Figure, each solid line represents the CPU execution time for a specific kernel. The dashed ones represent GPU execution time. Each color is one of the three most representative kernels. The best execution time for a specific kernel is at the intersection of two lines for the same kernel. For instance, we have the RELAXATION kernel that was executed in 3.55 seconds in the CPU and 3.10 seconds in the GPU with a domain decomposition of one subdomain of size 96 × 96 × 16 for the CPU and another of size 96 × 96 × 80 for the GPU. Comparing the execution time of the PROPAGATE kernel using the same domain decomposition size that RELAXATION kernel performed better to the execution time of the PROPAGATE kernel using the domain decomposition of 96×96×48 for the CPU and 96×96×48 for the GPU, which provided the kernel shortest execution time, the execution time was 16.12% worst using RELAXATION domain decomposition size. The same happens with the other kernels. For the INITIALIZE kernel, the shortest execution time was achieved with a GPU only execution. For the REDISTRIBUTE kernel the shortest execution time was achieved with subdomains of size 96 × 96 × 16 and 96 × 96 × 80 for the CPU and GPU, respectively, and for the PROPAGATE kernel with subdomains of size 96 × 96 × 48 and 96 × 96 × 48 for the CPU and GPU, respectively. For the BOUNCEBACK kernel, GPU only execution achieved the shortest execution time.

Table III presents the average execution time of the five kernels in the OpenMP + OpenCL version on the Intel Arria 10 platform. As in Table I, the first column is the kernels name, and the following columns are different decomposition sizes of the method domain, from CPU only to GPU only. The bold cells are the ones that perform better, and consequently, had the best partitioning for each kernel. INITIALIZE kernel performed better in FPGA only execution. Thus, collaborative execution did not improve its performance because the execution time is short. BOUNCEBACK was better using a decomposition of 96 × 96 × 16 for the CPU and 96 × 96 × 80 for the FPGA. It represents a performance improvement of 19.16%. The best distribution for REDISTRIBUTE and RELAXATION was also 96 × 96 × 16 for the CPU and 96 × 96 × 80 for the FPGA. For these kernels, most of the work is executed in the FPGA. It is explained by the number of cores of Intel Arria CPU which is only two. The PROPAGATE kernel performance was improved by 14.29% using a distribution of 96 × 96 × 32 for the CPU and 96 × 96 × 64 for the FPGA. In this case, the CPU handle more data. Using CPU only execution, execution time was 449.99 seconds. Using FPGA only, it was 186.24 seconds. However, collaboratively using both CPU and FPGA and dividing the domain into five distinct proportions (FPGA only, 16 × 80, 32 × 64, 16 × 80 and 16 × 80), the shortest execution time achieved was 158.02 seconds. In the end, the LBM performance was improved by 15.15%.

Figure 8 presents the average execution time of the three most representative kernels of the method in the OpenMP + OpenCL version on the platform Intel Arria 10. The kernels are REDISTRIBUTE, PROPAGATE and RELAXATION. In axis x, we have seven different workload proportions from CPU only to FPGA only. The solid lines represent the CPU execution time and the dashed ones, GPU execution time. The colors represent each kernel. The best execution time is at the intersection

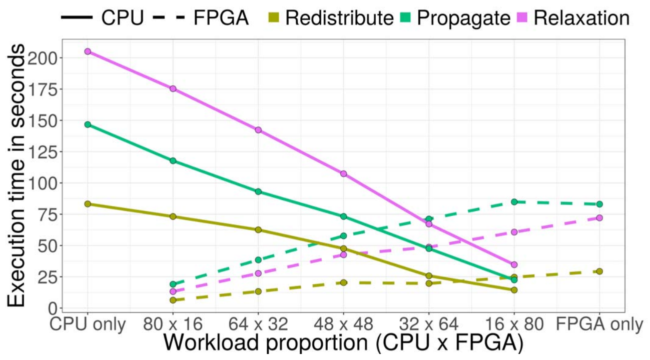

Fig. 8. Average execution time of the three most representative kernels on the CPU and FPGA devices of the Intel Arria 10 platform on each workload proportion using the OpenMP + OpenCL version with a domain of size 96 × 96 × 96.

TABLE IV
PERFORMANCE GAIN USING BEST NON-UNIFORM PARTITIONING FOR COLLABORATIVE EXECUTION OF THE ALL KERNELS ON AMD KAVERI AND INTEL ARRIA 10 PLATFORMS WITH A DOMAIN OF SIZE $96 \times 96 \times 96$.

	AMD Kaveri		Intel Arria 10	
Kernel	**Partitioning**	**Performance gain (%)**	**Partitioning**	**Performance gain (%)**
Initialize	0×80	0.00	0×80	0.00
Redistribute	16×80	17.09	16×80	15.85
Propagate	48×48	12.15	32×64	14.29
Bounceback	0×80	0.00	16×80	19.16
Relaxation	16×80	2.39	16×80	15.80

of two lines for the same kernel. With a three-dimensional domain of size $96 \times 96 \times 96$ and using only the CPU device (leftmost in the figure) the execution time of the kernels were 83.21, 146.68 and 205.07 seconds and using only the FPGA device (rightmost in the figure), the execution times were 29.31, 83.02 and 72.08 seconds. Nevertheless, collaboratively using both CPU and FPGA, the kernel's performance was improved by up to 19.16%. For instance, REDISTRIBUTE and RELAXATION performance was better with a distribution of $96 \times 96 \times 16$ for the CPU and $96 \times 96 \times 80$ for the FPGA. While PROPAGATE performance was better with $96 \times 96 \times 32$ for the CPU and $96 \times 96 \times 64$ for the FPGA. In the same way as for AMD Kaveri, in Intel Arria 10 the shorter execution times of each kernel were not necessarily obtained with the same domain decomposition.

VI. DISCUSSION

Our experimental results provide evidence that collaborative execution using non-uniform partitioning improve heterogeneous architectures performance. As a case of study, we show that LBM performance was improved by 11.39% and 15.15% in AMD Kaveri and Intel Arria 10, respectively.

Two points were essential in our approach to achieve that performance improvement. First, collaborative execution in heterogeneous architectures is possible due to the tight integration of the CPUs and the GPUs or FPGAs in these devices. It allows both devices working concurrently on the same workload, improving the overall system resources by employing both CPU threads and GPU or FPGA concurrency, thereby achieving higher performance. Second, non-uniform data partitioning is essential, which is a strategy that disjoint devices perform the same task on different subsets of the data.

From our experimental results, we make two major observations. First, as expected, it appears that each device is suitable or specialized for a specific kind of workload. That is, the performance of each computational kernel over a specific device depends on its workload. If the kernel is memory-bound, performance may be better if more workload is assigned to the CPU. On the other hand, if the kernel is CPU-bound assigning more workload to the GPU or FPGA may improve the overall performance. Second, choosing the optimal partitioning is one of the main challenges. Partitioning can be static, which a fixed fraction of workload is assigned to each device before execution, and dynamic that workload partitioning is defined at runtime.

Table IV summarizes the performance improvement of each LBM kernel on both SoC devices. The performance gain is calculated over the best individually performance which was GPU and FPGA for AMD Kaveri and Intel Arria 10, respectively. INITIALIZE and BOUNCEBACK performed better in GPU only executions. REDISTRIBUTE and RELAXATION performed better with a data partitioning of 16×80 in both CPU-GPU and CPU-FPGA. These kernels are more suitable to GPU and FPGA devices than CPU with a performance improvement of 17.09% in CPU-GPU and 15.85% in CPU-FPGA. PROPAGATE kernel, nonetheless, performed better for 48×48 and 32×64 data partitioning. It means that this kernel is suitable to both devices, having almost the same performance is both CPU-GPU and CPU-FPGA.

VII. CONCLUSION

In this work, we analyzed the performance impact of collaborative execution on two low-power heterogeneous architectures: an AMD Kaveri SoC with CPU x86-64 and Radeon R7 GPU devices integrated into a single chip, and an Intel Arria 10 SoC with ARM CPU devices and an integrated FPGA on a single chip. Our case of study was the D3Q19 Lattice Boltzmann Method application with five distinct kernels. We performed experiments with individual executions, CPU only, and GPU/FPGA only, and in a collaborative way through data decomposition of the data domain. Our experimental results suggest that collaborative processing reduces execution times and that non-uniform domain decomposition improves the kernel's performance by 11.39% and 15.15% on AMD Kaveri and Intel Arria 10, respectively.

Future works include an energy consumption analysis of our collaborative partitioning; the design of experiments using kernels from different applications; and the impact of a dynamic partitioning strategy on heterogeneous architectures.

ACKNOWLEDGMENT

This research received funding from the Petrobras project, grant n. 2016/00133-9, by CAPES (Brazil) - Finance Code 001 and by the project "GREEN-CLOUD" (#16/2551-0000 488-9), from FAPERGS and CNPq Brazil, program PRONEX 12/2014. The authors would like to thank Eduardo Roloff and

Francis B. Moreira, from UFRGS, Brazil, for their valuables feedbacks. We also thank Claudio Schepke from UNIPAMPA, Brazil, for his Lattice Boltzmann method figures.

REFERENCES

[1] D. B. Kirk and W.-m. Hwu, *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[2] K. Vipin and S. A. Fahmy, "Fpga dynamic and partial reconfiguration: A survey of architectures, methods, and applications," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, p. 72, 2018.

[3] K. O'Neal and P. Brisk, "Predictive modeling for cpu, gpu, and fpga performance and power consumption: A survey," in *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2018, pp. 763–768.

[4] K. Obrien, I. Pietri, R. Reddy, A. Lastovetsky, and R. Sakellariou, "A survey of power and energy predictive models in hpc systems and applications," *ACM Computing Surveys (CSUR)*, vol. 50, no. 3, p. 37, 2017.

[5] S. Mittal and J. S. Vetter, "A survey of cpu-gpu heterogeneous computing techniques," *ACM Computing Surveys (CSUR)*, vol. 47, no. 4, p. 69, 2015.

[6] D. Bouvier and B. Sander, "Applying amd's kaveri apu for heterogeneous computing." in *Hot Chips Symposium*, 2014, pp. 1–42.

[7] A. Akagic, E. Buza, R. Turcinhodzic, H. Haseljic, N. Hiroyuki, and H. Amano, "Superpixel accelerator for computer vision applications on arria 10 soc," in *2018 IEEE 21st International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*. IEEE, 2018, pp. 55–60.

[8] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart, *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media, 2014.

[9] Z. Jin and H. Finkel, "Power and performance tradeoff of a floating-point intensive Kernel on OpenCL FPGA platform," *Proceedings - 2018 IEEE 32nd International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2018*, pp. 716–720, 2018.

[10] K. Kofler, I. Grasso, B. Cosenza, and T. Fahringer, "An automatic input-sensitive approach for heterogeneous task partitioning," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ACM, 2013, pp. 149–160.

[11] G. P. Dávila, D. Oliveira, P. Navaux, and P. Rech, "Identifying the most reliable collaborative workload distribution in heterogeneous devices," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 1325–1330.

[12] J. Zhang and J. Li, "Improving the Performance of OpenCL-based FPGA Accelerator for Convolutional Neural Network," *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*, pp. 25–34, 2017.

[13] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An OpenCL(TM) Deep Learning Accelerator on Arria 10," in *International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 55–64.

[14] K. Ovtcharov, O. Ruwase, J.-y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating Deep Convolutional Neural Networks Using Specialized Hardware," *Microsoft Research*, pp. 3–6, 2015.

[15] F. Kono, N. Nakasato, K. Hayashi, A. Vazhenin, and S. G. Sedukhin, "Performance evaluation of tsunami simulation using OpenCL on GPU and FPGA," *Proceedings - IEEE 11th International Symposium on Embedded Multicore/Many-Core Systems-on-Chip, MCSoC 2017*, vol. 2018-Janua, pp. 106–113, 2018.

[16] S. Matsuoka, A. Smith, M. Matsuda, H. R. Zohouri, and N. Maruyamay, "Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs," *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, vol. 2016, no. November, p. 35, 2016.

[17] H. R. Zohouri, A. Podobas, and S. Matsuoka, "High-performance high-order stencil computation on FPGAs using opencl," *Proceedings - 2018 IEEE 32nd International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2018*, pp. 123–130, 2018.

[18] Zohouri, Hamid Reza and Podobas, Artur and Matsuoka, Satoshi, "Combined Spatial and Temporal Blocking for High-Performance Stencil Computation on FPGAs Using OpenCL," *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 153–162, 2018.

[19] M. S. Serpa, E. H. Cruz, M. Diener, A. M. Krause, P. O. Navaux, J. Panetta, A. Farrés, C. Rosas, and M. Hanzich, "Optimization strategies for geophysics models on manycore systems," *The International Journal of High Performance Computing Applications*, vol. 33, no. 3, pp. 473–486, 2019.

[20] J. Kraus, M. Pivanti, S. F. Schifano, R. Tripiccione, and M. Zanella, "Benchmarking gpus with a parallel lattice-boltzmann code," in *2013 25th International Symposium on Computer Architecture and High Performance Computing*. IEEE, 2013, pp. 160–167.

[21] C. Schepke, J. V. Lima, and M. S. Serpa, "Challenges on porting lattice boltzmann method on accelerators: Nvidia graphic processing units and intel xeon phi," in *Analysis and Applications of Lattice Boltzmann Simulations*. IGI Global, 2018, pp. 30–53.

[22] Y. Fu, F. Li, F. Song, and L. Zhu, "Designing a parallel memory-aware lattice boltzmann algorithm on manycore systems," in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Sep. 2018, pp. 97–106.

[23] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. Mccardwell, A. Villegas, and D. Kaeli, "Hetero-mark, a benchmark suite for cpu-gpu collaborative computing," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, Sep. 2016, pp. 1–10.

[24] J. Gmez-Luna, I. E. Hajj, L. Chang, V. Garca-Floreszx, S. G. de Gonzalo, T. B. Jablin, A. J. Pea, and W. Hwu, "Chai: Collaborative heterogeneous applications for integrated-architectures," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2017, pp. 43–54.

[25] S. Huang, L.-W. Chang, I. El Hajj, S. Garcia de Gonzalo, J. Gómez-Luna, S. R. Chalamalasetti, M. El-Hadedy, D. Milojicic, O. Mutlu, D. Chen, and W.-m. Hwu, "Analysis and modeling of collaborative execution strategies for heterogeneous cpu-fpga architectures," in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '19. New York, NY, USA: ACM, 2019, pp. 79–90. [Online]. Available: http://doi.acm.org/10.1145/3297663.3310305

[26] D. J. M. Moss, E. Nurvitadhi, J. Sim, A. Mishra, D. Marr, S. Subhaschandra, and P. H. W. Leong, "High Performance Binary Neural Networks on the Xeon+FPGA Platform," in *International Conference on Field Programmable Logic and Applications (FPL)*, vol. 27, 2017.

[27] E. Nurvitadhi, D. Sheffield, J. Sim, A. Mishra, G. Venkatesh, and D. Marr, "Accelerating binarized neural networks: Comparison of FPGA, CPU, GPU, and ASIC," *Proceedings of the 2016 International Conference on Field-Programmable Technology, FPT 2016*, no. c, pp. 77–84, 2017.

[28] L.-w. Chang, J. Gómez-Luna, I. El Hajj, S. Huang, D. Chen, and W.-m. Hwu, "Collaborative Computing for Heterogeneous Integrated Systems," *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering - ICPE '17*, pp. 385–388, 2017.

[29] S. Chen and G. D. Doolen, "Lattice Boltzmann Method for Fluid Flows," *Annual Review of Fluid Mechanics*, vol. 30, no. 1, pp. 329–364, 1998.

[30] C. Schepke, N. Maillard, and P. O. A. Navaux, "Parallel lattice boltzmann method with blocked partitioning," *International Journal of Parallel Programming*, vol. 37, no. 6, pp. 593–611, 2009.

[31] S. Cui, N. Hong, B. Shi, and Z. Chai, "Discrete effect on the halfway bounce-back boundary condition of multiple-relaxation-time lattice boltzmann model for convection-diffusion equations," *Physical Review E*, vol. 93, no. 4, p. 043311, 2016.

[32] G. Freytag, P. O. A. Navaux, J. V. F. Lima, L. H. S. Mello, Schnorr, and P. Rech, "Non-uniform domain decomposition for heterogeneous accelerated processing units," in *International Meeting on High Performance Computing for Computational Science (VECPAR)*, vol. 13, 2018.

[33] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[34] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science and engineering*, vol. 12, no. 3, p. 66, 2010.

.