Project Title	SCAlable LAttice Boltzmann Leaps to Exascale
Project Acronym	SCALABLE
Grant Agreement No.	956000
Start Date of Project	01.01.2021
	36 Months
Project Website	www.scalable-hpc.eu

D3.3 – Implementation and final report about scheduling and load-balancing

Work Package	**WP 3.3, Development of appropriate load-balancing**
Lead Author (Org)	**Raphael Kuate (CSGROUP)**
Contributing Author(s) (Org)	
Reviewed by	**Romain Cuidard (CSGROUP)**
Approved by	**Management Board**
Due Date	**30.06.2023**
Date	**17.07.2023**
Version	**V1.0**

Dissemination Level

X	PU: Public
	PP: Restricted to other programme participants (including the Commission)
	RE: Restricted to a group specified by the consortium (including the Commission)
	CO: Confidential, only for members of the consortium (including the Commission)

The SCALABLE project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 956000.

Versioning and contribution history

Version	Date	Author	Notes
0.1	29.06.2023	Raphael Kuate (CSGROUP)	TOC and V0.1
0.2	30.06.2023	Romain Cuidard (CSGROUP)	Review
1.0	17.07.2023	Corentin Lefevre (Neovia)	Edition of the final version approved by the MB

Table of Contents

The SCALABLE project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 956000.

List of Figures

The SCALABLE project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 956000.

Executive Summary

The main objective of SCALABLE for CSGROUP is the improvement of LaBS deployment in bigger clusters of thousands of cores, achieved by a transfer of performance technology from WaLBerla. Therefore, in the earlier work packages 3.1 [1], 3.2 [2] and 3.5 [3] many improvements were made to LABS scalability. In the previous reports on work package 3.5 and 3.2 we introduced works on LaBS targeting GPU clusters. In this report, we present the development of a prototyping GPU version of LABS.

1 Introduction

1.1 Context

Lattice Boltzmann methods (LBM) are nowadays trustworthy alternatives to conventional CFD methods, since it has been already shown in several engineering applications that they are faster than Navier-Stokes approaches in comparable scenarios. LBM methods can handle complex geometries and a wide range of Multiphysics applications that are of high industrial relevance. The main distinguishing feature of the LBM is its algorithmic locality stemming from an explicit time step. Thus, the LBM is especially well-suited to exploit advanced supercomputer architectures through vectorization, accelerators, and massive parallelization.

1.2 Objective

The main objective of SCALABLE for CSGROUP is the improvement of LaBS scalability, thus its deployment in bigger clusters of thousands of cores. Among massive parallelization techniques, GPU clusters have nowadays become more efficient and reliable for industrial applications than decades ago. LaBS future developments will thus target GPU clusters architectures as well as is does for CPU clusters. Initially, we didn't plan to develop a LaBS GPU version, but since we haven't deeply changed all our data structure as we initially planned (wp 3.1 [1]), we decided to work on a GPU version. This decision was mainly motivated by our exchanges with Marcus Holzer, Gabriel Staffelbach of waLBerla/ Cerfacs and by Jayesh Badwaik from Juelich university on aspects of GPU implementation.
However, to maintain as much as possible a single code for both CPU and GPU, we chosen the solution provided by the C++17 standard parallelism combined with the NVC++ compiler, a compiler able to provide GPU and CPU executables from the same C++ source code.

In the current document, we will present a LaBS GPU prototype developed for a simple test case, on uniform mesh.

2 LaBS architecture

LaBS code execution can be divided into three parts: the pre-processing phase, the solver phase and the post-processing phase.

2.1 Pre-processing

The SCALABLE project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 956000.

The aim of the pre-processing phase is to manage the user inputs, configure the data structure and the whole simulation workflow for the solver and post-processing. This workflow uses the concept of data flow programming where a schedule of tasks to be executed is built. As LaBS manages complex geometries for industrial test cases, it uses an unstructured cell design which results in a lightweight mesh compared to structured blocks widely used in many LBM software. So, the pre-processing performs a parallel octal mesher step with domain decomposition which enhances surface meshes files provided by the user, describing the geometry, refinement and boundary constraints. Among the important steps of the pre-processing phase, one has the load-balancing step, the sewing step capable of improving the quality of the mesh provided by the user, the scheduler step which generates the main input for the remaining code execution: the schedule of tasks from data (geometry) and physics. A field-mapping step manages initial conditions and other sources fields provided by the user in various forms for physics.

The entire pre-processing step launched by the CPU. We have however developed a in previous work (wp 3.5) an algorithm which manages in MPI multi-process parallelism a pre-processing on N CPU cores targeting P GPU cards, with P < N.

2.2 Solver phase

Once the pre-processing is achieved, the solver phase consists of reading the schedule and executing tasks of which the LaBS unstructured cell design of data is organized into *families* of cells where the same LBM computations functions are applied. Thus, the solver has two main parts: the *kernel* and the *hard library.* The *solver kernel* drives *the hard library* for physics computations and output calculations required within each timestep. Each function of the hard library consists of an API of two arguments defining the range of nodes on which its computations apply. The general conception of the LaBS GPU version is then based on the parallelization of the functions of the hard library such that it be compiled into compute kernels for GPU device.

2.2.1 NVIDIA nvc++ compiler and CUDA unified memory

To achieve the goal of calling physics computations functions (*hard library*) within either a CPU or GPU device without re-coding specifically the entire solver phase for GPU device, we have chosen a mean which needs less code changes and keeps the code being able to be compiled for CPU or GPU. The NVIDIA compiler nvc++ and its CUDA unified memory technology are designed for this purpose. Using the standard C++ 17 parallel algorithms, the loops written in parallel way using the standard C++ *std::for_each* algorithm are automatically converted into a CUDA callable function targeting a NVIDIA GPU device. Many other ways to write parallel part convertible to a GPU device code by the nvc++ compiler exists: openACC, openMP. The CUDA unified memory technology is a way of managing memory by NVIDIA nvc++ compiler such that data allocated being automatically transferred from CPU (**host**) to GPU (**device**) and back from device to host when needed. However, these NVIDIA tools are designed for (almost newer) NVIDA GPU cards only, no GPU card from other vendors can be used at this point.

The SCALABLE project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 956000.

2.2.1.1 Parallel loops

The functions of the *hard library* (physics computation functions) are mainly organized in two blocks of codes: the first block retrieves data and other parameters of the *kernel* needed for the computations and the second block is the loop on the range of nodes on which the computations are applied. The standard C++ algorithm *std::for_each* API needs a callback function and two iterators on objects to loop on as mandatory arguments. The two bounds of the range of nodes already used by the functions of the *hard library* are the base of the iterators needed by *std::for_each* API, since LaBS do not store all data of each node within a data structure, as it may be done in some other LBM codes. The second mandatory argument type, the callback function can be written in two main ways for nvc++: a lambda function or an object function, functions pointers are not supported by nvc++ for GPU target. We use the object function approach which has the main advantage of keeping almost unchanged parts of existing codes which are moved into a C++ class, whose *() operator* becomes the entry point. However, all functions called must be *inlined* if not, it won't be compiled by nvc++ for the device, but for the host. The main requirement for all data used on the device and thus being able to be managed by the CUDA unified memory, is its allocation on the heap; data on the stack being ignored or randomly handled by the device.

2.2.1.2 Data transfer between host and device

The main goal of the CUDA unified memory technology is the automatic handling of data transfer between host and device, since no explicit CUDA code must be written within the standard C++ code being compiled. NVIDIA nvc++ compiler requires that data must be allocated in the heap because the standard C++ memory allocator in this case is automatically replaced by NVIDIA own allocator implementation. However, a nvc++ compiler option exists allowing the user not to use CUDA unified memory allocator.

The first tests made with LaBS on GPU face an important memory overhead problem due to the CUDA unified memory policy adopted by NVIDA for efficient allocation/deallocation. It appears that all C++ standards containers used in LaBS were not always deallocated, leading to a continuous growing memory. For a specific executable, NVIDIA provides some environment variables for adjusting the bounds of the unified memory allocation. The later options could not be easily applicable in LaBS, since it requires that the user/programmer knows about the amount of memory required for each test case before its launch. Another option was to compile parts of the code not called by the device with the CUDA unified memory allocator disabled by the nvc++ compiler option. However, since no safe memory communication can be done between data managed by the CUDA unified memory allocator and the standard C++ memory allocator, the latter option required an entire refactoring of LaBS code for the separation of parts being handled by the two nvc++ compiler options used for this purpose with a communication API.

2.2.1.3 Hint

A hint has finally been found by using two memory allocators in the code: the CUDA uniform memory allocator and another defined allocator which skips CUDA unified memory allocator. In practice, variables used by the device or shared between the host and device are allocated in the C++ standard way, since CUDA unified memory allocator automatically replaces the standard malloc/free calls by its own allocator. The remaining variables are allocated with the

The SCALABLE project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 956000.

defined allocator which for example, calls indirectly malloc/free, but from a library compiled without CUDA uniform memory allocator.

The following summary shows the differences in terms of memory overhead using CUDA Unified memory allocator in different steps of the run. Test case of 630 508 fluid nodes

Runing step	Peak memory in Mega bytes per processor	
	CUDA unified memory on shared data + std allocator otherwise	CUDA Unified memory everywhere
Input	155.8	132.3
FluidLinks	944.0	2815.3
Migrate	1055.4	5339.8
FieldMapping	1071.4	5354.7
Poster	1364.4	6854.6

2.2.2 Kernels developments

As a first version of LaBS on GPU, we have focused our efforts on the development of a version for a simple test case, i.e., a uniform test case with the basic physics solved, using the HRR scheme and the D3Q19 LBM stencil. Thus, five *hard library* functions are coded for the GPU version: *Macroscopic, Collide, Propagate, Gradient* for physics and *Dimensionality* which handles computed variables for output. Many tests and profiling have been done with the objective of continuously improving the acceleration factor, using the NVIDA profiling tools nvprof and nsys.

In LaBS, computations are performed with a cache mechanism which limits the size of memory managed. Variables computed during the solver phase are stored in SOA (structure of array) way, however a temporary AOS (array of structure) data storage is used for some computations involving ***pdf*** (particle distribution functions). These tunings have proven performance and scalability on CPU.

The first GPU versions, very close to the usual LaBS CPU version, had very poor performances. The following changes were implemented and improved the GPU compiled version:

The **removal of cache mechanism** on the GPU compilation thus, using the largest *family/*range of nodes for computations, allowing more optimized computed kernels.

The **merging of functions**. We observed that the less one calls the device during a time step, the more computations are accelerated on GPU. So, we defined a simplified scheme for the purpose of the GPU version which uses mainly two functions: a physics computation function, which sequentially calls two GPU kernels: (*Propagate + Macroscopic*) and (*Gradient + Collide*) and a third kernel: *Dimensionality. A*ll hard library functions could not be merged. *Dimensionality* is being called depending on output times and other related user specifications; thus, it becomes an independent GPU kernel. We then merged (*Propagate + Macroscopic*) into a GPU kernel, and (*Gradient + Collide*) into another GPU kernel, since the

The SCALABLE project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 956000.

second step of computations (*Gradient + Collide*) needs computed *Macroscopic* values of the neighborhood of a node being computed, thus the first step (*Propagate + Macroscopic*) must have been completed entirely. We also call successively the two main GPU kernels inside the same function launched by the scheduler.

The following table shows differences between a version with five GPU kernels and an optimized version running only three kernels on double precision computations, no computation on outputs.

Mesh number of fluid nodes	Solver step performances in MLUPS	
	Five GPU kernels called by five *hard library* functions	Merged version with three GPU kernels called by two *hard library*
100x100x100 (1 M)	27.8	204.06
128x128x128 (2.09 M)	28.04	206.43

The **reduction of memory footprint**. Among the quantities analyzed using NVIDIA profiler tools, one can denote the size of memory and number of registers used by each GPU thread, the occupancy percentage of the GPU card and the size of the grid used by the GPU kernels. We have observed that the grid size is automatically adjusted by nvc++ at runtime such that at least two parallel launches be made for each kernel call. The parameter of which optimization improved the performances, and which depends on the kernel implementation was the size of memory used within the GPU threads. So, we optimized the amount of memory data occupies in the kernels by replacing static arrays stored within the object functions used as kernel, with C++ macros, and also by keeping at the most local level computed quantities usually stored at global scope: Gradients usually stored for all nodes in the CPU version for example, are now stored only for the current node being computed by the GPU kernel, since the same kernel computes the collision, which needs the gradient, immediately after.

The following table shows differences between a version and the optimized one with memory footprint reduced running in a single precision computation, no computation on outputs.

Mesh number of fluid nodes	Solver step performances in MLUPS	
	Without memory footprint optimization	Reduced memory footprint
100x100x100 (1M)	290.89	558.04
300x300x300 (27M)	309.74	609.31

An **optimization of the scheme**. Among the lots of tests made to identify the main bottom neck of performances on GPU, we added a loop on lots of arithmetic operations in the kernels, the latter not solving the physics but just to measure their behavior within the performances.

The SCALABLE project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 956000.

We observed that repeating about N times the loop on arithmetic operations added leads to about N/3 extra elapsed time. However, a similar test with la loop on the entire kernel (including memory operations) leads to about N extra elapsed time. Thus, since the *Propagate* function does not perform arithmetic operations, but simply pull from SOA and redispose in AOS ***pdf*** quantities. These ***pdf*** quantities when used in AOS storage were achieving best performances on CPU. We removed *Propagate* as standalone function and replaced the table of its pulled values by the exact memory operations earlier performed by the *Propagate* function, thus performing in-place propagation when needed. Even if this in-place propagation is done twice in the current simplified scheme *(Macroscopic, Collide*), this optimization gave one of the biggest gaps observed on performances.

The following table shows the difference between the versions with the in-place propagation, with computations on output: average density every 1000 iterations, thus four outputs over 4000 iterations; running in a single precision computation.

Mesh number of fluid nodes	Solver step performances in MLUPS	
	Without in-place propagation	With in-place propagation
100x100x100 (1M)	684.93	1010.9
300x300x300 (27M)	927.95	1428.05

The following table shows the difference between the versions with the in-place propagation, without outputs (deactivated) running in a single precision computation.

Mesh number of fluid nodes	Solver step performances in MLUPS, no output	
	Without in-place propagation	With in-place propagation
100x100x100 (1M)	1038.8	2255.08
300x300x300 (27M)	1182.1	2550.08

2.3 Post-processing

The post processing phase is performed at this point on CPU. However, some operations on data for outputs are computed within the scheduled operations at each timestep, depending on the configuration of output. The operations on output are separated from the *hard library.* The existing data operations on outputs were performed for each node on all its output, with data in AOS storage. The later way leads to very poor performances on GPU. We have thus improved the data operations algorithm such that it manages operations for each output, on all its nodes, thus with data stored in SOA way, which gives a better acceleration factor on GPU devices.

The SCALABLE project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 956000.

The following table shows differences when computations are performed on outputs: average density every 1000 iterations, thus four outputs over 4000 iterations; running in a single precision computation.

Mesh number of fluid nodes	Solver step performances in MLUPS	
	With unoptimized computations on output	With optimized computations on output
100x100x100 (1M)	12.45	684.93
300x300x300 (27M)		927.95

3 Illustrative test case

We have made many tests on performances improvements during the developments and verified that the physics were “correctly solved” on GPU. We present here some pictures on the behavior of the GPU version, compared to the usual LaBS CPU version. The test case is the so called COVO, with 1600x1600x4 nodes.

COVO is an isentropic vortex is simply moving to the east of a periodic square grid. With time. The flow is supposed inviscid Euler Equations. The following picture shows its initial state.

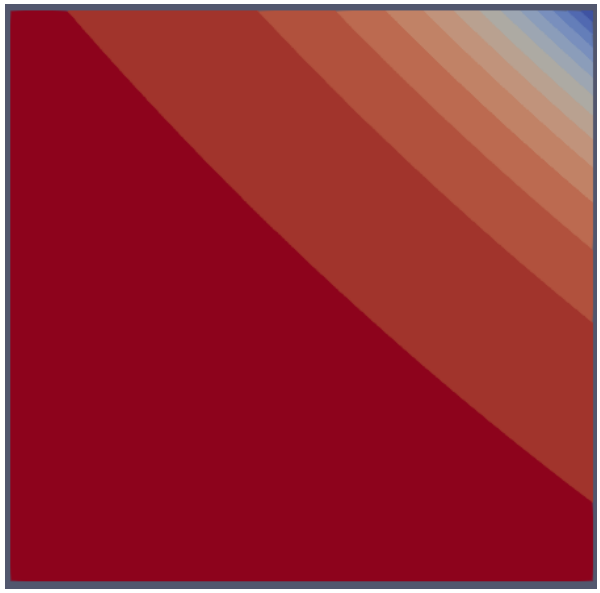

Figure 1. COVO test case, picture of the top of the mesh in z direction. Velocity in y direction at initialization.

The original test case is a 2D case, but for some 2D output matters on the base LaBS version extended to GPU, we had to turn into a 3D case by adding 4 nodes in z axis. The number of iterations is limited to 1000. The CPU code runed using 32 cores while the GPU was launched on a NVIDIA A100 GPU card with 40 GB of GPU memory. The tests have been done in double precision simulations. One can observe that the overall behavior of the simulation is quite the same on both CPU end GPU.

The SCALABLE project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 956000.

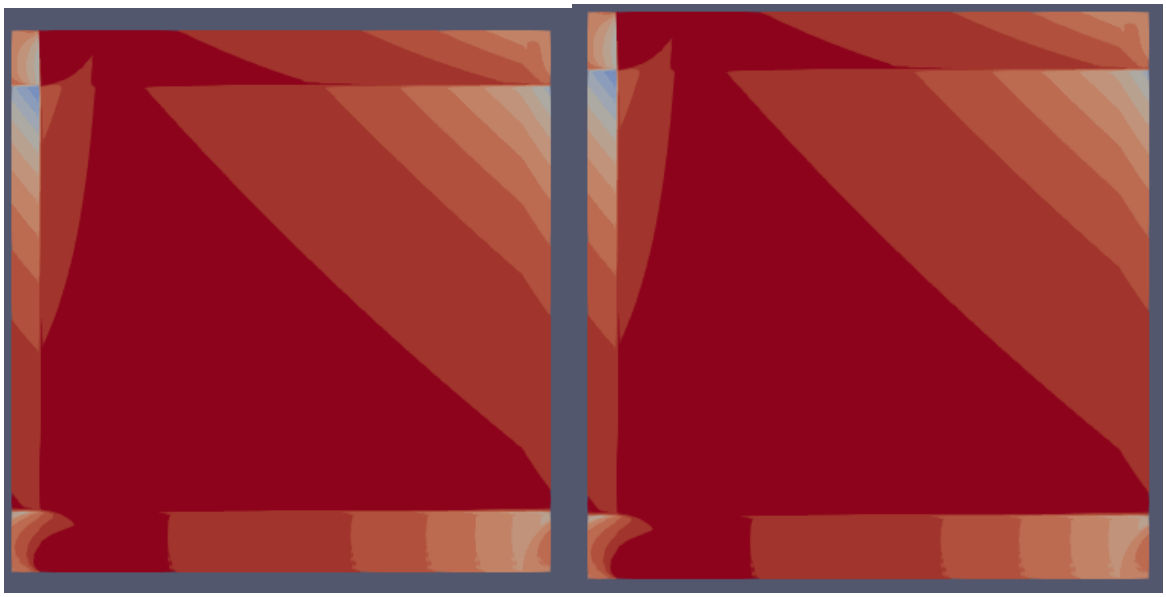

Figure 22. COVO test case, picture of the top of the mesh in z direction. Velocity in y direction at time 3/10. Left CPU results. Right GPU results

Figure 33. COVO test case, picture of the top of the mesh in z direction. Velocity in y direction at time 7/10. Left CPU results. Right GPU results

4 Perspectives

We have developed the first version of LaBS for GPU for a uniform test case, without complex physics. The remaining work is to handle the entire *hard library* into GPU kernels for complex physics, a task which needs prior work on refactoring and merging of some functions otherwise, small GPU kernels won't be efficient, as we have observed. The other part of the

The SCALABLE project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 956000.

remaining work is the handling of complex geometry such as meshes with refinements levels, porous and rotating domains which may widely affect the data organization into GPU device.

5 Acknowledgments

We thank the other SCALABLE partners for their help upon the achievement of this work. The technical meetings and exchanges we had with Jayesh Badwaik, Markus Holdzer, Gabriel Staffelbach and Radim Vavrik; gave us very helpful ideas on overall and technical aspects of GPU programming and profiling.

6 Bibliography

[1] R. C. Raphael Kuate, "D3.1 – Description of synthetic structured-unstructured data organization," SCALABLE project's public deliverables, https://scalable-hpc.eu/public-deliverables/, 2022.

[2] R. C. Raphael Kuate, "D3.2 – Report on scheduling and load balancing," SCALABLE project's public deliverables, https://scalable-hpc.eu/public-deliverables/, 2023.

[3] R. C. Raphael Kuate, "D3.5 – Pre-processing implementation and report," SCALABLE project's public deliverables, https://scalable-hpc.eu/public-deliverables/, 2022.

The SCALABLE project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 956000.