PERFORMANCE PORTABILITY THROUGH SYCL: LBM AS A CASE STUDY

Youssef Mesri, Ouadie El Farouki

ParCFD2022
33^{rd} International Conference on Parallel Computational Fluid Dynamics
May 25-27 2022, Alba Italy

PERFORMANCE PORTABILITY THROUGH SYCL : LBM AS A CASE STUDY

Ouadie EL FAROUKI* **AND Youssef MESRI†**

* Mines ParisTech, Codeplay Software Ltd
R&D
e-mail: ouadie.elfarouki@codeplay.com

†Mines ParisTech
HPC & AI
Sophia Antipolis, France
e-mail: youssef.mesri@mines-paristech.fr

Key words: Parallel programming, Performance Portability, SYCL, LBM

Abstract. The present work falls under the umbrella of computational fluid dynamics from a software engineering perspective. The cornerstone of this study is to highlight functional and performance portability challenges rising from the software-hardware diversity we observe nowadays, along with the constraints and dependencies it brings. In order to lift this burden, we've adopted SYCL as a parallel programming language to leverage its hardware-agnostic property and implement an abstract parallelization of a 2D Lattice Boltzmann Method, able to compile and execute on a wide range of hardwares, including intel multi-core CPUs and GPUs and Nvidia GPUs.

1 INTRODUCTION

Computational methods have emerged from a cross-discipline intersection, including natural sciences such as physics, applied mathematics, and computer sciences. This set of techniques proved to be powerful and efficient, and made it possible for both researchers and industries to push further the limits of investigation and exploration of physical and chemical phenomena, and consequently solve to a high degree of efficiency and robustness many real world engineering problems. Computational Fluid Dynamics (CFD) gained particularly significant momentum thanks to the theoretical advances on one hand, and the considerable progress and innovations in computer sciences. This considerable leap in software engineering techniques was specifically made possible through the development of High Performance Computing frameworks and languages for GPGPU (General Purpose Computing on Graphical Processing units), besides the availability of highly efficient hardware accelerators.

The resulting diversity on both development stack and hardware architecture ends caused the emergence of some challenges, namely the stack-hardware dependency. In fact, developers can focus on a target hardware architecture and use an adequate tool (CUDA for Nvidia GPUs for instance) to implement highly efficient models, but will

find it impractical to reuse their code base on different platforms without making radical changes or even complete re-writings. Even when such a "functional-portability" is handled through adequate software-hardware combinations, performance portability is not always guaranteed.

We'll turn our focus throughout this paper, on the performance portability of one of the major computational techniques widely adopted nowadays : the Lattice Boltzmann Method (LBM). Similar studies[1] have been conducted to tackle this subject using libraries such as Kokkos to implement abstract unique code able to execute efficiently on multiple platforms such as x86 (intel, amd CPUs), Nvidia GPUs, and ARM. In the present work, we'll dive into another approach targeting the same challenge using SYCL as a parallel programming model for heterogeneous systems, along with some of its main implementations and libraries.

2 Background

2.1 SYCL in a Nutshell

SYCL [2] is an industry-driven open standard that brings data parallelism to C++ for heterogeneous systems. It's a "single-source" programming model and a cross-platform abstraction layer built on top of OpenCL's concepts, that ensures a high level of abstraction. SYCL promises improved maintainability, productivity, and ease of use, while offering the same degree of low level control and optimization through its underlying backends. SYCL is basically a pure C++ programming model, with no language extensions or special syntax like the one we find in CUDA Nvidia for instance and no special preprocessing directives (pragmas) such as OpenACC or OpenMP.

As a high level abstraction, SYCL uses dedicated backends to target specific hardware accelerators, such as OpenCL, Hip, and CUDA using adequate IRs (SPIR/SPIR-V, PTX etc...), and the list is growing with the additional SYCL implementations and their supported backends (ComputeCPP[3], OneAPI[4], HipSYCL[5] etc..).

The main construct of a SYCL programm is the SYCL Queue, to which actions are submitted in an asynchronous way to be executed on the underlying device. A queue incorporates a DIrected Acyclic Graph of Kernels and memory operations (Nodes), along with implicit or explicit dependencies (Edges), either implemented using explicit event management policies or implicitly inherited from the data buffers access dependencies. From a parallelization perspective, SYCL relies on a built-in abstract representation of the multi-level parallelization space denoted as ND-range, which can be mapped explicitly (by the user) or implicitly (by the SYCL implementation) to the underlying hardware parallelism(s) of the device.

2.2 Lattice Boltzmann Method

The Lattice Boltzmann Method falls under the recent computational methods introduced to solve modeling and simulation problems for CFD applications. It originated from Boltzmanns work on the kinetic theory of gazes where he formulated the equation named after him (Boltzmann Equation) to describe fluid dynamics using a distribution

function $f(x, \xi, t)$ representing normalized density of abstract groups of particle at a given point in space x, with velocity ξ at a given time t. The scale of such a modeling approach (mesoscopic) actually lies between two other well known methods : the Navier-Stokes formulation which stands as a macro description of fluid using macro variables such as temperature and density, and the molecular dynamic simulation (MDS) which applies basic Newton principles to individual particles at a microscopic scale.

After discretization of LBE into a velocity and physical space following a DdQq scheme, the numerical resolution becomes straightforward using the discrete-velocity distribution function $f_i(x, t)$ $i \in [0 : q]$ where the space vector x takes discrete values in the d-dimension physical grid and t corresponds to time-steps used for iterations. Following an initialization step of the system's macro-variables, these iterations consist of two physically intuitive steps: collision and streaming. The collision is a simple algebraic local operation. First, the density and the macroscopic velocity are calculated to find the equilibrium and the post-collision distributions. After collision, the resulting distribution is streamed to neighboring nodes.

Unlike legacy CFD methods that require complex implementations and rely on heavy solver operations, LBM turns out to be easier to implement and proved to be highly-parallelizable[6] given its relatively local operations dependencies when computing the distribution function states throughout the streaming and collision phases. Even though additional memory allocation is usually required in LBM compared to other computational methods, management of such computer resources remains feasible.

3 Experimental Settings and Implementation

3.1 Simulation model

For the sake of simplicity and demonstration purposes, a 2D single-phase fluid flow within a horizontal parallel pipe will be used as a simulation model. The LBM scheme is set to D2Q9, and the BGK (Bhatnagar-Gross-Krook) formulation will be used for the collision operator. An inflow condition is setting the velocity profile and an outflow condition simulates an infinite extension through simple values copy. Besides, periodic boundary conditions are imposed on the upper and lower pipe limits. The physical obstacle has been modeled as a no-slip circular surface to verify and visualize turbulence phenomena such as Karman vortex street.

In this setting, the discretization of LBE using BGK operator in the velocity space (9 discrete velocities), physical space (2 dimensions x and y), and time is formulated as follow for each discrete velocity direction $i \in [1 : 9]$:

$$f_i(x + c_i\Delta t, t + \Delta t) = f_i(x, t) - \frac{\Delta t}{\tau}(f_i(x, t) - f_i^{eq}(x, t)) \tag{1}$$

Where f_i^{eq} is the discrete distribution at equilibrium, $\frac{\Delta t}{\tau}$ being the relaxation frequency (also linked to the fluid's viscosity) and c_i the discrete velocity vectors.

The macroscopic variables, a.k.a mass density ρ and momentum density ρu at (x, t) are deduced as moments of f_i :

$$\rho(x,t) = \sum_i f_i(x,t) \qquad \rho u(x,t) = \sum_i c_i f_i(x,t) \tag{2}$$

The BGK operator relaxes the population to an equilibrium given by (after truncation):

$$f_i^{eq}(x,t) = w_i \rho \left(1 + \frac{u.c_i}{c_s^2} + \frac{(u.c_i)^2}{2c_s^4} - \frac{u.u}{2c_s^2}\right) \tag{3}$$

where the weights w_i specific to the chosen discrete velocity set. It's clear how $f_i^{eq}(x,t)$ only depends on local quantities (density and velocity at (x,t)) which can be obtained directly from equations (2) given that $u(x,t) = \rho u(x,t)/\rho(x,t)$.

Decomposition of LBM can be done to separate collision from streaming steps as follow :

- Collision :

$$f_i^*(x,t) = f_i(x,t) - \frac{\Delta t}{\tau}(f_i(x,t) - f_i^{eq}(x,t)) \tag{4}$$

- Streaming :

$$f_i(x + c_i\Delta t, t + \Delta t) = f_i^*(x,t) \tag{5}$$

3.2 Implementation overview

The implementation of this simple 2D simulation will be conducted using standard C++ combined with SYCL runtime library, and compiled in two seperate setups using DPCPP for Nvidia GPUs target through CUDA backends and ComputeCPP for intel CPU/GPU targets through OpenCL backends.

SYCL Buffers and accessors, supported by both DPCPP and ComputeCPP will be used to implicitly manage memory operations between host and device, including initial allocation and persistence of relevant variables on device memory and on-demand copy of macroscopic variables back to host for display/store to hard drive purposes. Different data layouts (namely $[n_x, n_y, n_{pop}]$ Vs. $[n_{pop}, n_x, n_y]$) and parallelization schemes (simple 1-D, 2-D with SIMD vectorization and 3-D ranges) will be evaluated for benchmarking purposes. A granular as well as aggregated profiling process was implemented to assess performance at kernel, copy and iteration levels.

Further details about the implementation will be published at a later stage, an initial Proof of Concept has been made public on our Github Repository[7].

REFERENCES

[1] Werner Verdier, Pierre Kestener, Alain Cartalade. 2020. Performance portability of lattice Boltzmann methods for two-phase flows with phase change.

[2] SYCL official website. Khronos, 2022. https://www.khronos.org/sycl.

[3] ComputeCPP developer guide. Codeplay, 2022. https://developer.codeplay.com/products/computecpp/ started.

[4] OneAPI official website. Intel, 2022. https://www.oneapi.io.

[5] HipSYCL github repository. 2022. https://github.com/illuhad/hipSYCL

[6] Timm Krger, Halim Kusumaatmaja, Alexandr Kuzmin, Orest Shardt, Goncalo Silva, Erlend Magnus Viggen. 2017. The Lattice Boltzmann Method : Principles and Practice.

[7] LBM On SYCL : Proof of Concept. Github repository. 2021. https://github.com/Ouadio/Parallel-Lattice-Boltzmann-SYCL