# miniLB: A Performance Portability Study of Lattice-Boltzmann Simulations

Luigi Crisci[1,*], Biagio Cosenza[1], Giorgio Amati[2] and Matteo Turisini[2]

[1]University of Salerno, Via Giovanni Paolo II 132, 80084, Fisciano, Italy
[2]CINECA, Via dei Tizii, 6b, 00185 Roma, Italy

**Abstract**

The Lattice Boltzmann Method (LBM) is a computational technique of Computational Fluid Dynamics (CFD) that has gained popularity due to its high parallelism and ability to handle complex geometries with minimal effort. Although LBM frameworks are increasingly important in various industries and research fields, their complexity makes them difficult to modify and can lead to suboptimal performance. This paper presents *miniLB*, the first, to the best of our knowledge, SYCL-based LBM mini-app. *miniLB* addresses the need for a performance-portable LBM proxy app capable of abstracting complex fluid dynamics simulations across heterogeneous computing systems. We analyze SYCL semantics for performance portability and evaluate *miniLB* on multiple GPU architectures using various SYCL implementations. Our results, compared against a manually-tuned FORTRAN version, demonstrate effectiveness of *miniLB* in assessing LBM performance across diverse hardware, offering valuable insights for optimizing large-scale LBM frameworks in modern computing environments.

**Keywords**

Lattice Boltzmann Methods, GPU, heterogeneous computing, SYCL

## 1. Introduction

In High-Performance Computing (HPC), *mini-apps* (or *proxy-apps*) are simplified codes that allow application developers to share and analyze important key features of large applications without forcing users to assimilate large and complex code bases. Mini-apps are often used as abstract models to evaluate performance and assess performance, portability, and performance portability ($\Phi$). Mini-app can also capture programming methods and styles that drive requirements for algorithms, compilers, and other toolchain elements. Developing mini-apps for relevant use cases is an important challenge in pushing the boundaries of HPC application performance. Important projects such as the Exascale Proxy Applications Project [1] aim to improve the quality of proxies produced by the Exascale Computing Project by defining standards for documentation, building and testing systems, performance models and evaluations, and templates and best practices for proxy developers to help meet these standards.

In recent years, the Lattice Boltzmann Method (LBM) has further strengthened its position as a valuable tool in the field of computational fluid dynamics [2]. LBM has attracted increasing interest in many industries and research organizations due to its high parallelism efficiency and ability to discretize complex geometries with little effort. This has led to the development of large frameworks, typically focused on specific LBM domains, with extremely complex and large code bases [3, 4, 5]. However, LBM frameworks have not evolved with the evolution of (massively parallel and distributed) computing systems, resulting in complex codebases that are very difficult to modify. Unfortunately, no mini-app for LBM can efficiently abstract the problem while providing hints for performance tuning and optimization.

This paper proposes the first mini-application for LBM, with an implementation in SYCL that allows not only performance evaluation but also performance portability on modern heterogeneous computing systems. Specifically, this paper makes the following contributions:

*BigHPC2024: Special Track on Big Data and High-Performance Computing, co-located with the 3$^{rd}$ Italian Conference on Big Data and Data Science, ITADATA2024, September 17 – 19, 2024, Pisa, Italy.*

*Corresponding author.
lcrisci@unisa.it (L. Crisci); bcosenza@unisa.it (B. Cosenza)

© 2024 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR-WS.org/Vol-3785/paper103.pdf

- The, to the best of our knowledge, first performance portable, tunable, SYCL-based Lattice-Boltzmann mini-app, translated from an original FORTRAN implementation, capable of targeting a wide range of multicore CPUs, GPUs, and accelerators.
- A performance portability analysis of SYCL semantics, including Unified Shared Memory, range, and ND-range kernels using a performance portability metric.
- An experimental evaluation of *miniLB* on NVIDIA V100S, AMD MI100, and Intel Max 1100 GPUs, using multiple SYCL implementations and different SYCL semantic combinations, compared to the manually-tuned FORTRAN version using different parallelism paradigms and compilers.

## 2. Background and Related Work

**Lattice Boltzmann Method** The Lattice Boltzmann Method (LBM) emerged in the late 1980s as an evolution of lattice gas cellular automata. It has since found numerous applications across various complex flow problems, from fully developed turbulence to micro and nanofluidics, and even quark-gluon plasmas.

The core concept of LBM is to solve a simplified Boltzmann kinetic equation for a set of discrete distribution functions, known as populations, $f_i(\mathbf{x}; t)$. These functions represent the probability of finding a particle at position $\mathbf{x}$ and time $t$, with a discrete velocity $\mathbf{v} = \mathbf{c}_i$. The discrete velocities are selected to ensure sufficient symmetry, thereby satisfying the mass, momentum, and energy conservation laws of macroscopic hydrodynamics and maintaining rotational symmetry. Figure 1 illustrates the lattices used for 2D LB simulations, featuring a set of nine discrete velocities (D2Q9). Instead of directly solving the Navier-Stokes equations ($\rho\vec{u}$), LBM solves a kinetic equation storing nine populations for each grid point, corresponding to different $\mathbf{c}$ velocity directions, including $\mathbf{c} = (0, 0)$. It is not necessary to store derived quantities like velocity and density.

In its compact form, the main LB equation is as follows:

$$f(\vec{x} + \vec{c_i}, t + 1) - f_i(\vec{x}; t) = -\omega(f_i(\vec{x}; t) - f_i^{eq}(\vec{x}; t)) + S_i, \quad i \in [0, b + 1] \tag{1}$$

where $\vec{x}$ and $\vec{c_i}$ are position and velocity vectors in ordinary space, $f_i^{eq}$ is the equilibrium distribution function, $S_i$ is a source term for the fluid interaction with external (or internal) sources. The local equilibria are provided by a lattice truncation, to the second order in the Mach number $M = u/c_s$, of the Maxwell-Boltzmann distribution, where $c_s$ is the lattice sound speed.

$$f_i^{eq}(\vec{x}, t) = w_i \rho (1 + u_i + q_i) \tag{2}$$

where $w_i$ is a set of weights normalized to the unit and,

$$u_i = 3\frac{\vec{u} * \vec{c_i}}{c_s} \qquad\qquad q_i = 9\ \frac{(\vec{u} * \vec{c_i})^2}{2c_s^2} - 3\ \frac{u^2}{2c_s^2} \tag{3}$$

where the left term is linear in velocity, the right one quadratic. The equation 1 represents two key processes: the *collision step* (right-hand side), where the populations locally relax towards equilibrium, and the streaming step (left-hand side), where the populations are propagated to neighboring locations at $\mathbf{x} + \mathbf{c}_i$ at time $t + 1$. This scheme can be demonstrated to reproduce the Navier-Stokes equations for an isothermal, quasi-incompressible fluid in terms of density and velocity [6].

In literature, several large Lattice Boltzmann frameworks exist, e.g. OpenLB[3], waLBerla[4], Palabos[5]. However, such frameworks usually present very large and complex code bases, which makes it difficult to experiment with for research purpose. Our work aims at staying as simple as possible while providing a playground for experimenting with SYCL-specific or LB-specific optimizations while providing good performance on multiple hardware. Other attempts to verify the performance of heterogeneous programming models on LB methods have been proposed in the literature: Ding et.al. [7] explore the performance of the SYCL and Kokkos programming model on an LB application, showing

performance pitfalls of both implementations. However, their work does not focus on evaluating single SYCL features like miniLB but rather the raw application performance. Moreover, their analysis only focuses on NVIDIA GPUs, while we examine performance portability across multiple vendors.

**SYCL** SYCL [8] is a royalty-free, cross-platform C++ abstraction layer that enables developers to write code for multiple heterogeneous devices, such as CPUs, GPUs, and FPGAs, in a convenient and performance-portable way. SYCL enhances the C++ programming language by adding abstractions for managing heterogeneous computing within ISO C++, aiming to align closely with the core language specifications. Originally designed to map onto OpenCL, the third revision of the SYCL 2020 specification allowed for custom backends, like NVIDIA CUDA, AMD HIP, OpenMP, and others. Key implementations of SYCL include Intel's OneAPI Data-Parallel C++ [9] and AdaptiveCPP [10], along with several other smaller-size implementations [11, 12, 13]. The versatility of SYCL has led to various extensions for specific heterogeneous computing scenarios, including distributed computing [14], real-time energy optimization [15], and approximate computing [16].

**Performance Portability** As HPC systems evolve with diverse hardware architectures, developing efficient, cross-device application code becomes crucial. This has led to the rise of "performance portability" in academic circles, which measures both an application's ability to meet performance benchmarks on specific platforms and its capacity to run across various hardware configurations.

However, a universally accepted definition is absent. In our research, we embrace the definition of performance portability by Pennycook et al. [17]: *"A measurement of an application's performance efficiency for a given task that can be successfully executed on all platforms within a specified set."* The formula to quantify performance portability is presented in Equation 4:

$$\Phi(a,p,H) = \begin{cases} \frac{|H|}{\sum_{i \in H} \frac{1}{e_i(a,p)}}, & \text{if platform } i \text{ is supported for all } i \in H \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

Here, $a$ represents the application, $p$ denotes the problem addressed by $a$, and $H$ signifies the set of target hardware. The performance portability metric $\Phi$ is defined as the harmonic mean of the application's performance efficiency $e_i(a,p)$ over the set of hardware $H$.

Pennycook et al. [17] highlight various methods for calculating application performance efficiency, specifically: *architectural efficiency*, which measures achieved performance as a fraction of peak hardware performance; and *application efficiency*, which measures achieved performance as a fraction of the best-observed performance against the most optimized native implementation.

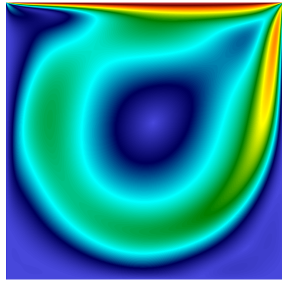
## 3. miniLB Overview

### 3.1. Computational Description

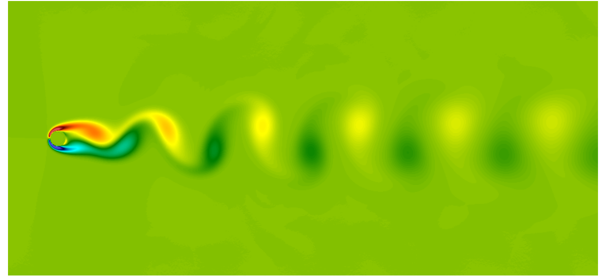
*miniLB* is a bidimensional computational fluid dynamic code for single-phase incompressible flows, with nine discrete velocities (D2Q9 using CFD jargon). It is a downsizing of a 3D `FORTRAN90 MPI+OpenACC` full application, developed by CINECA [18, 19]. *miniLB* is written in C++20 and SYCL, a single-source abstraction layer for heterogeneous computing. *miniLB* has no external dependencies and uses no SYCL compiler-specific feature, allowing it to run out-of-the-box on every platform with any SYCL compiler. The code is open-source and available on GitHub[1].

*miniLB* implements a *fused* approach [20, 21], where the *collision* and *streaming* operation are performed in a single kernel. In this approach, the app holds a pre-collision population $f^{pre}$ and a post-collision population $f^{post}$: at time $t$, input values read through a scattered read from $f^{pre}$, and post-collision results are written in $f^{post}$. Finally, the two populations are swapped at time $t+1$. Populations are

[1] https://github.com/Luigi-Crisci/miniLB



(a) Lid-Driven cavity



(b) Von Karman Street

**Figure 1:** Use case example outputs using the .vtk file with ParaView

stored using a Structure-of-Array (SoA) layout, with a unit-strided vector for each population $f_i$. *miniLB* has been designed to be highly and easily tunable to measure SYCL performance in a wide range of scenarios, with a total of 96 possible configurations, highlighted below in section 4.

#### 3.1.1. Numerical precision

*miniLB* supports four different numerical precisions to control how data is computed and stored:(i) `Single`: quantities are stored in single precision and all floating point operations are performed in single precision. (ii) `Double`: quantities are stored in double precision and all floating point operations are performed in double precision. (iii) `Mixed1`: quantities are stored in half precision and all floating point operations are performed in single precision. (iv) `Mixed2`: quantities are stored in single precision and all floating point operations are performed in double precision.

The `Single` precision option is the default one. A more comprehensive analysis of LBM numerical precision has been explored in [22].

### 3.2. FORTRAN-based Parallelization

The original FORTRAN app uses a *directive-based* parallelization that minimize code refactoring from CPUs to GPUs implementation. It implements several programming models: `OpenACC` pragmas, `OpenMP Offload`, and the FORTRAN built-in operator `DOCONCURRENT`.

While GPU vendors provide native compilation toolchains for some of those programming models, not all of them are supported by each vendor: for example, `OpenACC` supports only NVIDIA GPUs and AMD through the CRAY compiler, while `DOCONCURRENT` is not supported on AMD discrete GPUs. Furthermore, to achieve optimal performance on specific hardware, users must compile their programs using the proprietary vendor compiler (e.g. nvFORTRAN for NVIDIA, amdclang for AMD, ifx for Intel). This requirement increases fragmentation and adds complexity, as it necessitates testing with additional toolchains.

### 3.3. Use Cases

*miniLB* supports three classic CFD benchmarks: Lid-Driven Cavity (LDC), Taylor-Green Vortex, and Von Karman Street (VKS). The app also produces VTK output files for offline visualization with tools like ParaView [23]. In this paper, we focus on the latter and the former: LDC and VKS.

**Lid-Driven Cavity** The LDC problem involves a square or rectangular cavity closed on all sides. The top lid of the cavity moves at a constant velocity, while the other three walls remain stationary. This setup generates a complex flow pattern within the cavity, characterized by the following: *(a)* no-slip boundary conditions: all walls, including the moving lid, have a no-slip boundary condition, meaning the fluid velocity relative to the wall is zero; *(b)* driven flow: the movement of the top lid at a constant tangential velocity $u_0$ drives the flow within the cavity. Figure 1a shows an example output.

**Von Karman Street** The VKS occurs when fluid flows past a cylindrical object and the flow separates alternately from either side of the object, creating a pattern of vortices in the wake. This phenomenon is characterized by: *(a)* Periodic Vortex Shedding: alternating vortices are shed from opposite sides of the cylinder, creating a staggered array of vortices downstream. *(b)* Flow Regimes: the flow pattern depends on the Reynolds number, which is a dimensionless number representing the ratio of inertial forces to viscous forces in the flow. A visual example is given in figure 1b.

## 4. SYCL Porting

In this section, we analyze the principal SYCL features used during the porting and the available *miniLB* configurations.

### 4.1. Kernel Parallelism

SYCL uses `parallel_for` to declare parallel code regions. In particular, SYCL offers two variants of it: `range` and `NDrange`. The `range` is the simplest as it requires the user to specify only the iteration space. This allows the runtime to select the most appropriate number of threads depending on the target device without user intervention. On the other hand, `NDrange` allows to manually tweak the local iteration space (e.g. local workgroup size on OpenCL), allowing more fine-grained optimization but also requiring the user to manually optimize the local size to match the current device. *miniLB* kernels support both `range` and `NDrange`. The app defaults to `range` parallel for, but the latter can be activated by setting the `-DBGK_SYCL_ND_RANGE` compile-time parameter. Currently, the app only supports the tweaking of the collide and stream kernel size through the `-DBGK_SYCL_ND_RANGE_[X,Y]_SIZE` parameter at compile-time.

### 4.2. Data Management and Access

*miniLB* uses Unified Shared Memory for data management instead of the *Buffer-Accessor (BA)* paradigm as it shows more reliable and stable performance [24]. USM provides three allocation kinds: `malloc_device` are allocated directly on the target device, `malloc_host` allocates host page-locked memory accessible from both host and device, and `malloc_shared`, which are shared between devices using an automatic memory migration system.
*miniLB* implements two memory management backends, one based on `malloc_device` and `malloc_host` and one using `malloc_shared`, controllable via the compile-time parameter `-DBGK_SYCL_MALLOC_SHARED`. In the former, *miniLB* stores a device and host pointer for each population and manually migrates memory from the host and device. The host device is pinned because this increases the bandwidth with some GPU architectures [25]; in the latter, a single pointer is stored and the SYCL runtime handles the memory migration. In addition, the parameter `-DBGK_SYCL_ENALBE_PREFETCH` enables hints to prefetch memory on the host/device to the SYCL runtime.

Multi-dimensional data are defined with *MDspan* [26]. It is a lightweight, non-owning view that allows a piece of memory to be treated as a multi-dimensional entity. *MDspan* allows us to define the extents (i.e. the number of dimensions and sizes), the layout (e.g. row-major, column-major, etc.), and the data accessor (i.e. how to translate the pointer/index pair to a memory location). *miniLB* stores a two-dimensional *MDspan* for both host and device to reduce the view construction overhead. In addition, a compile-time parameter `-DBGK_SYCL_LAYOUT_[RIGHT|LEFT]` switches the data layout to row-major or column-major respectively. *miniLB* defaults to column-major as it is the default layout in the original FORTRAN application.

| Feature | Description | Options |
|---|---|---|
| Precision | Kernel's floating point precision | `Single, Double, Mixed1, Mixed2` |
| Queue | SYCL queue | `Out-of-order, In-order` |
| USM | SYCL USM data management backend | `Device, Shared, Shared + Prefetch` |
| Layout | Data representation layout | `Row-major, Column-major` |
| Ranges | SYCL kernel types | `Range, NDRange` |

**Table 1**
*miniLB* tuning configurations

### 4.3. Task Scheduling

To submit tasks, a SYCL user needs to create a `queue` that binds to a specific device. SYCL supports both out-of-order and in-order submissions. With the former, submitted tasks are executed without a defined order, allowing for parallel kernels execution. With USM, data dependencies between kernels must be explicitly tracked. With in-order submission, instead, kernels are executed in FIFO order. This hinders the ability to parallelize kernel executions, but it removes the overhead of dependency tracking. *miniLB* supports both queue configuration, controlled by the parameter `-DBGK_SYCL_IN_ORDER_QUEUE`.

## 5. Experimental evaluation

### 5.1. Experimental Setup

We evaluated *miniLB* on three GPUs from three principal GPU vendors, i.e. NVIDIA Tesla V100S, AMD MI100, and Intel Max 1100. We tested the app on two use cases: Lid-Driven Cavity (LDC) and Von Karman Street (VKS), using a $4096 * 4096$ grid with Reynolds number $Re = 10000$ and with 100.000 timesteps. As performance metric, we use MLUPs (Mega Lattice Update per second) that indicates how many millions of gridpoints are updated each second.

For the SYCL implementations, we chose AdaptiveCpp (commit sha a3c5c9d), and DPC++(commit sha ea0c067), where both support all three architectures. For AdaptiveCpp, we target the *generic* backend, which can target every hardware through an integrated JIT compiler. For the FORTRAN version, we used NVHPC 24.5 for NVIDIA, amdclang ROCM 6.0 for AMD, and IFX 2024.0.0 for Intel. To reduce the number of combinations in the tuning space, this paper explores all combinations except `Mixed 1` and `Mixed 2` precision, SYCL out-of-order queues, row-major layout, and the `Shared+Prefetch` configuration.

### 5.2. Use Case Evaluation

Figure 2 shows the achieved SYCL performance, while figure 3 shows the SYCL performance normalized to the best FORTRAN implementation on a given hardware (e.g. OpenACC on LDC-NVIDIA, OpenMP on AMD-VKS). For each use case and hardware combination, the result of the best SYCL configuration is shown (e.g. for DPC++ on the NVIDIA V100S with the LDC use case, we pick `NDrange + device` allocation). Both AdaptiveCpp and DPC++ show similar performance on every hardware on both use cases. The SYCL version outdo FORTRAN on almost every platform and precision with both compilers, achieving an average of 1.16x speedup on the NVIDIA V100S, 1.03x on Intel Max 1100, and 1.54x on AMD MI100. An interesting case is AMD: while the results are similar to the ones obtained on other platforms, the speedup over FORTRAN is way higher(e.g. 1.55x compared to 1.16x on NVIDIA with AdaptiveCpp). To find the cause of this speedup, we profiled both the FORTRAN and SYCL application on single precision using OmniPerf [27]. We found that `amdclang` performs an aggressive thread coarsening, reducing the iteration space by a factor of 30. This results in a significantly higher L1I and L1D cache misses by duplication, 26 miss per wave against the 0.04 miss per wave of the SYCL

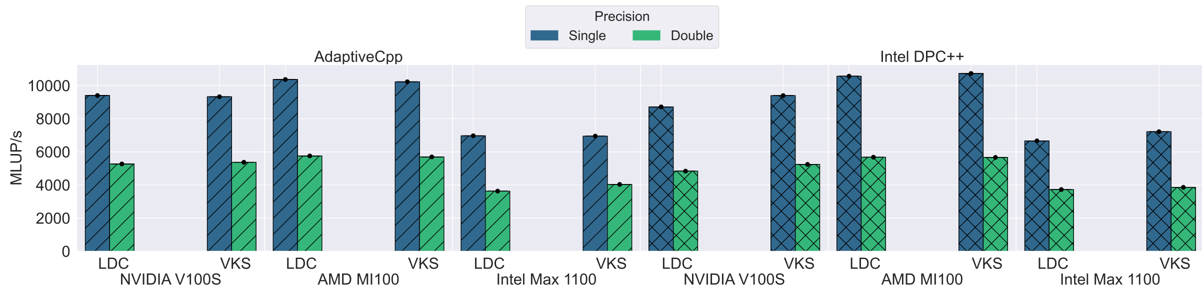



**Figure 2:** Best SYCL configuration Million Lattice Update per seconds (MLUPs) for Lid-Driven Cavity and Von Karmann Street use cases

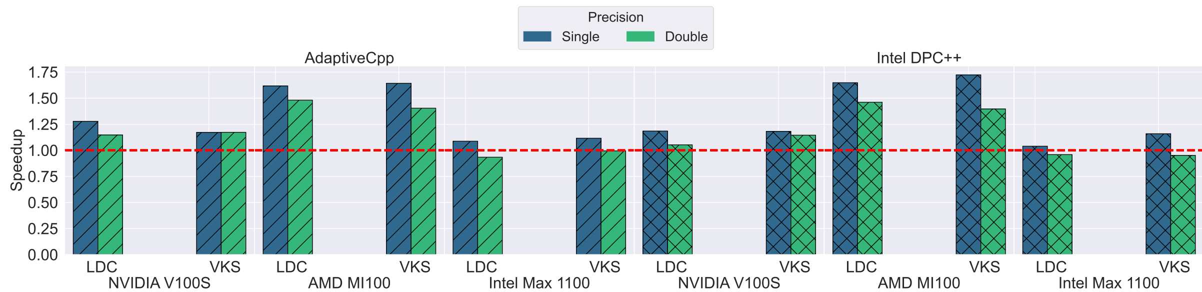



**Figure 3:** Best SYCL configuration speedup normalized over the Fortran best implementation

implementation. Moreover, the FORTRAN version achieves 53% more L2 cache misses, severely limiting application bandwidth. Similar considerations apply to the double precision too.

## 5.3. SYCL Feature Evaluation

SYCL provides a great variety of built-in constructs to parallelize an application on heterogeneous hardware. However, the performance of each construct heavily depends on the target use case [24]. Furthermore, different SYCL platforms could implement the same feature in different way, adding additional complexity. Figure 4 shows the performance of the Lid-Driven Cavity use case with each combination of USM allocations and SYCL kernel's type on every hardware and precisions, using both AdaptiveCpp and Intel DPC++. For `NDrange` kernels, the work group size is the biggest one possible on the target GPU (i.e. 1024 threads), organized as a block of 1x1024 threads. The values are normalized with the FORTRAN OpenMP offload backend, as it is the only one available on every hardware. In those benchmarks, we add a checkpoint at T = 50.000 to force data movement between host and device.

From the picture is clear that kernel performance in SYCL is heavily dependent on the adopted SYCL parallelization type. In particular, SYCL `range` are above the baseline in 33% of the cases. On the other hand, `NDrange` kernels beat the baseline in 85% of the configuration. However, some discrepancy between SYCL implementations arises: AdaptiveCpp `range` uses a work group size of 128 threads, organized in a 16x16 grid with 2-dimensional kernels. On NVIDIA GPUs, the gridsize is divided into 256x256 blocks. This results in 15% more uncoalasced global memory acccess compared to the `NDrange` version, where the work group size is unrolled along the y-axis (1x1024, or 1024x1 if in row-major). Similar considerations apply also to other architectures. On the other hand, DPC++ `range` kernels always select the largest possible work group size on GPU and put all the threads in one dimension (e.g. 1x1024 if the kernel is 2-dimensional). This heuristic performs well on both AMD and Intel, where `range` picks the same size as the one manually specified for `NDrange`. However, on NVIDIA GPUs, `range` kernels performance are detrimental, achieving only 15% of the `NDrange` performance. The difference in performance between hardware is due to a small, but significant change in the work group size definition heuristic on Nvidia hardware: while on AMD and Intel hardware, threads are placed on the first dimension (e.g. 1024x1x1), on NVIDIA hardware DPC++ place the

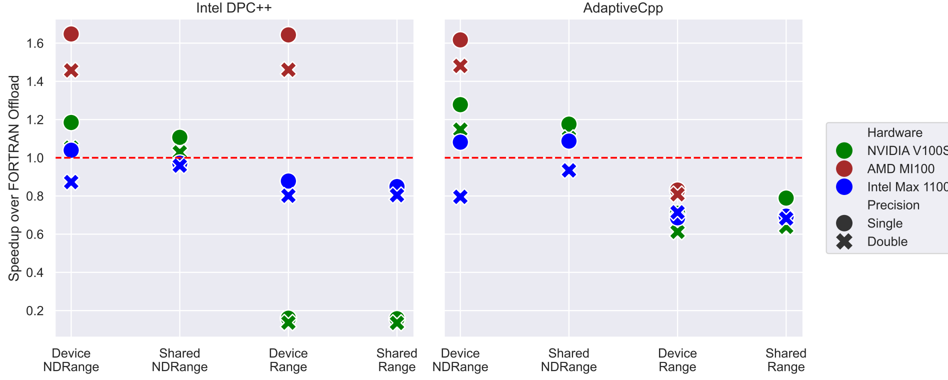



**Figure 4:** SYCL speedup over FORTRAN offload parallelism model by varying kernel and allocation type with column-major layout and in-order queues, on Lid-Driven Cavity

threads on the second dimension (1x1024x1). This results in 93% more uncoalasced access on the Tesla V100S compared to the other hardware. By switching to a `row-major` layout, NVIDIA performance improves but it crashes performance on AMD and Intel GPUs for the same reason. This discrepancy between work group size definitions severely limits the `range` performance portability across hardware.

Regarding data management, shared allocations are not shown on AMD hardware: on AMD GPUs, on-demand page migration between host and device memory relies on the XNACK feature, which is disabled by default. However, XNACK is known to be experimental and unstable. When we enabled it, we encountered random kernel failures and GPU hangs. Consequently, we disabled it for this analysis. Without XNACK enabled, *shared* allocations function like *host* allocations, meaning the data is allocated on the host and transferred to the device at each memory access, generating up to 1000x slowdown compared to the other implementation. Therefore, for this evaluation, we consider the AMD *shared* backend as not available. On average, `device` allocation beat the baseline in 60% of the configuration, while `shared` allocation only on 43% of the cases. However, `shared` allocations performance depends on hardware support: for example, on NVIDIA GPU they beat the baseline 50% of the time, while on the Intel GPU they only achieve better performance than the baseline on 37%. This variability, together with the unreliability of UVM on AMD, raises questions on its application to production scenarios.

## 5.4. Performance Portability evaluation

| SYCL Impl. | Hardware | USM | Kernel | $AI$ | $FR$ (GFlop/s) | $P$ (TFlop/s) | $e'$ |
|---|---|---|---|---|---|---|---|
| Intel DPC++ | NVIDIA V100S | Shared | Range | 0,38 | 125 | 0,43 | 29% |
| | | Shared | NDrange | 1.37 | 976 | 1.55 | 63% |
| | | Device | Range | 0.38 | 124 | 0.43 | 29% |
| | | Device | NDrange | 1.27 | 934 | 1.55 | 60% |
| | AMD MI100 (Est.) | Shared | Range | X | X | X | X |
| | | Shared | NDrange | X | X | X | X |
| | | Device | Range | 1.41 | 1132 | 1.60 | 70% |
| | | Device | NDrange | 1.41 | 1132 | 1.60 | 70% |
| | Intel Max 1100 | Shared | Range | 1.22 | 781 | 0.976 | 80% |
| | | Shared | NDrange | 1.22 | 782 | 0.976 | 80% |
| | | Device | Range | 1.21 | 772 | 0.968 | 79% |
| | | Device | NDrange | 1.21 | 775 | 0.968 | 80% |

**Table 2**
*miniLB* performance data for *col_MC* kernel, single precision w/ Intel DPC++

| SYCL Impl. | Hardware | USM | Kernel | $AI$ | $FR$ (GFlop/s) | $P$ (TFlop/s) | $e'$ |
|---|---|---|---|---|---|---|---|
| AdaptiveCpp | NVIDIA V100S | Shared | Range | 1,07 | 657 | 1.21 | 54% |
| | | Shared | NDrange | 1.32 | 926 | 1.49 | 62% |
| | | Device | Range | 1.05 | 588 | 1.18 | 49% |
| | | Device | NDrange | 1.32 | 885 | 1.49 | 59.2% |
| | AMD MI100 (Est.) | Shared | Range | X | X | X | X |
| | | Shared | NDrange | X | X | X | X |
| | | Device | Range | 1.14 | 601 | 1.38 | 44% |
| | | Device | NDrange | 1.13 | 976 | 1.36 | 72% |
| | Intel Max 1100 | Shared | Range | X | X | X | X |
| | | Shared | NDrange | 1.21 | 769 | 0.968 | 79% |
| | | Device | Range | 1.04 | 603 | 0.83 | 72% |
| | | Device | NDrange | 1.21 | 900 | 0.968 | 93% |

**Table 3**
miniLB performance data for *col_MC* kernel, single precision w/ AdaptiveCpp

To measure the application performance portability, we employ the Pennycook $\Phi$ metric [17].

However, we don't have a native optimized application version for each target hardware. In addition, calculating the application *architectural efficiency* can be challenging, as it requires the identification of relevant bottlenecks on each hardware. For those reasons, to calculate the performance portability we employ the *roofline efficiency*, which measures the distance between the application FLOP/s to the top of the roofline. Roofline efficiency has been demonstrated to successfully approximate architectural efficiency [28]. To calculate roofline efficiency, one needs to calculate the device peak performance

$$P = \min\left(FR_m, BW_m \times AI_k\right)$$

where $FR_m$ is the device floating point peak, $BW_m$ is the device bandwidth peak, and $AI_k$ is the arithmetic intensity for the application $k$, measured as the ratio between the application FLOP $FL$ and memory transferred. To capture those values, we used Nsight Compute on NVIDIA platform [29] and Intel Advisor on Intel [30]. However, we encountered two difficulties.

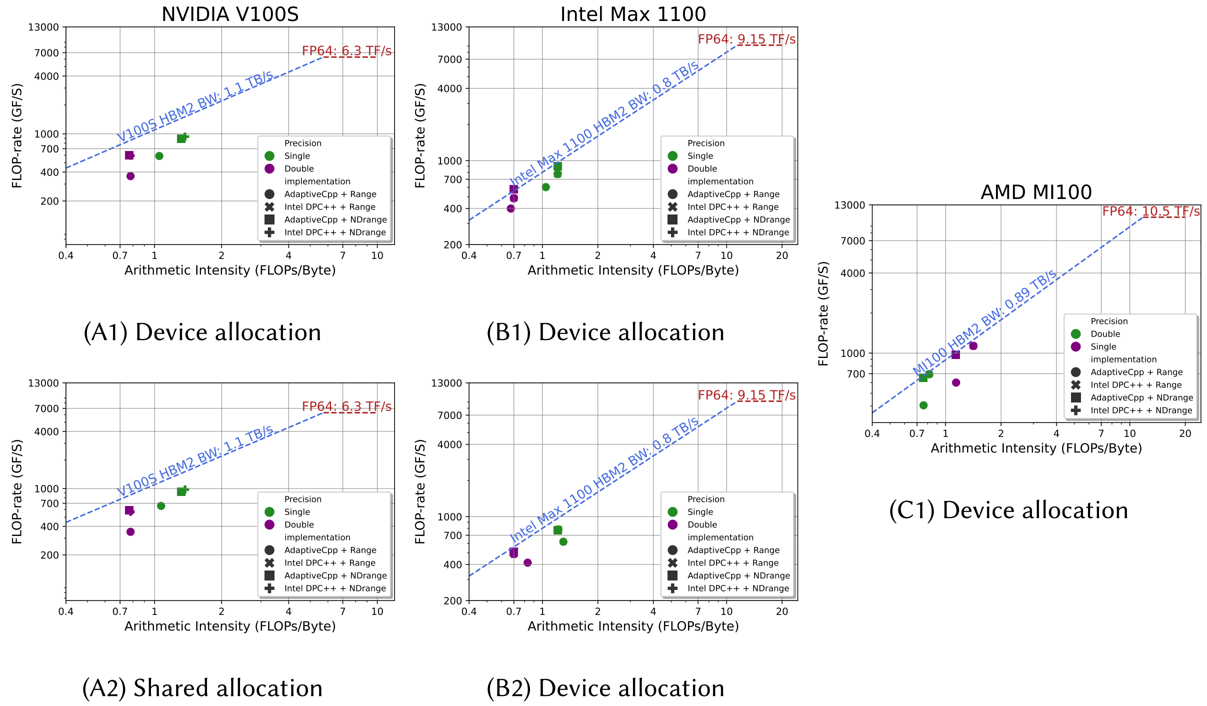
First, the AMD MI100 does not provide FLOP counters, therefore it is not possible to calculate the application FLOP/s. However, *miniLB* kernels do not have any device-dependent branch, therefore we expect the number of floating point operations to be the same on all three platform. To estimate the application FLOP/s, we use a similar methodology to the one defined in [28]:
$FR_{amd} = FR_{nvidia} * \frac{kernel_time_{amd}}{kernel_time_{nvidia}}$, where $FR_{nvidia}$ is the floating point ratio of the corresponding implementation on NVIDIA hardware, and $\frac{kernel_time_{amd}}{kernel_time_{nvidia}}$ is the ratio between the two application kernel performance.

The second problem is related to AdaptiveCpp on Intel. Profiling an AdaptiveCpp-compiled application with Intel Advisor results in a profiler's internal exception, therefore we couldn't measure the flop rate on Intel Max. Depending on the kernel type, we followed different procedures:

- `NDrange`: Both DPC++ and AdaptiveCpp use the same work group size, therefore we approximate the flop rate by multiplying DPC++ flop rate by the ratio between the AdaptiveCpp and DPC++ performance.
- `range` Because AdaptiveCpp uses a different work group size compared to DPC++, we can't approximate the kernel bandwidth with high precision, therefore we skip this configuration.

We measured the memory bandwidth of each hardware: our results showed a 1.1TB/s bandwidth on the NVIDIA V100S, 0.89TB/s on the AMD MI100, and 0.8TB/s on the Intel Max 1100. Table 2, 3, shows the performance value collected for the fused collide and stream kernel, called *col_MC* kernel. $e'$ indicates the distance from the roofline peak. The roofline results for both precision are shown in figure 5. As with every LB application, *miniLB* is bandwidth-bound, therefore the device peak depends on the device bandwidth and arithmetic intensity. For space constrain, we only show the results for single precision. Interestingly, *miniLB* achieves at least 62% of the device peak on every target hardware. We can see that on Intel Max we achieve the highest roofline efficiency, getting up to 91% of the peak on AdaptiveCpp with `NDrange`. While DPC++ and AdaptiveCpp show similar roofline efficiency for `NDrange` kernel, AdaptiveCpp `range` is 37% and 8% slower than DPC++ respectively on AMD and Intel `device` allocation. However, because of the previously mentioned uncoalesced access issue, DPC++ is 46% and 42% slower than AdaptiveCpp with `shared` and `device` allocation respectively on the NVIDIA V100S. This means that, while DPC++ could achieve better performance, on average AdaptiveCpp `range` heuristic is more portable among devices.

Finally, table 4 and 5 show the performance portability ($\Phi$) metric results for each precision. $\Phi'$ represents the value of performance portability considering only the current combination of data management backend and kernel type, while $\Phi$ is the maximum $\Phi'$ across all data management backend . Because we treated `shared` allocation as not available on AMD hardware, $\Phi$ is 0 for each `shared` configuration. *miniLB* achieves a minimum of 60% of performance portability among every precision, showing how SYCL can efficiently target any of the major vendor GPUs. `NDrange` achieve a medium portability of 78% among all precision, while `range` gets a medium portability of 62%. It is worth noting that, while `NDrange` required a tuning phase to find the best work group size for each hardware, `range` achieved such results without any user intervention.




(A1) Device allocation (B1) Device allocation (C1) Device allocation

(A2) Shared allocation (B2) Device allocation

**Figure 5:** *miniLB* roofline models per target hardware

| Precision | Kernel type | USM allocation | Ᵽ′ | Ᵽ |
|---|---|---|---|---|
| Single | Range | Device | 64% | 64% |
| | | Shared | 0 | |
| | NDrange | Device | 78% | 78% |
| | | Shared | 0 | |

**Table 4**
*miniLB* performance portability metric
for *col_MC* kernel, single precision

| Precision | Kernel type | USM allocation | Ᵽ′ | Ᵽ |
|---|---|---|---|---|
| Double | Range | Device | 61% | 61% |
| | | Shared | 0 | |
| | NDrange | Device | 78% | 78% |
| | | Shared | 0 | |

**Table 5**
*miniLB* performance portability metric
for *col_MC* kernel, double precision

## 6. Conclusion and Future Work

We presented *miniLB*, the first, highly tunable, SYCL-based lattice Boltzmann mini-app. We successfully ported the original FORTRAN application to C++ and SYCL, achieving a considerable speedup on every platform. We analyzed a subset of the 96 possible *miniLB* configuration settings to evaluate multiple combinations of SYCL features. We found that AdaptiveCpp and DPC++ portability can severely depend on the target SYCL feature, e.g. DPC++ `range` heuristic being less performance portable than AdaptiveCpp. Finally, we analyze *miniLB* performance portability using the well-known Ᵽ metric. Our results show that *miniLB* achieves high performance portability, with a Ᵽ value up to 78%.
As a future work, we plan to implement more SYCL features and optimizations, e.g. local memory, `Buffer-Accessors`, specialization constants, etc., as well as extending the app to multi-GPU systems, using both low-level MPI calls and high-level SYCL frameworks like Celerity. Furthermore, we would like to extend *miniLB* to the 3D case and measure the impact of mixed precision computation in SYCL, for both numerical stability and energy consumption.

## Acknowledgments

This project has received funding from the Italian Ministry of University and Research under PRIN 2022 grant No. 2022CC57PY (LibreRT project).

## References

[1] Exascale proxy applications project, https://proxyapps.exascaleproject.org/, 2024.

[2] K. V. Sharma, R. Straka, F. W. Tavares, Lattice boltzmann methods for industrial applications, Industrial & Engineering Chemistry Research 58 (2019) 16205–16234. URL: https://doi.org/10.1021/acs.iecr.9b02008. doi:10.1021/acs.iecr.9b02008.

[3] M. J. Krause, A. Kummerländer, S. J. Avis, H. Kusumaatmaja, D. Dapelo, F. Klemens, M. Gaedtke, N. Hafen, A. Mink, R. Trunk, J. E. Marquardt, M.-L. Maier, M. Haussmann, S. Simonis, Openlb—open source lattice boltzmann code, Computers and Mathematics with Applications 81 (2021) 258–288. URL: https://www.sciencedirect.com/science/article/pii/S0898122120301875. doi:https://doi.org/10.1016/j.camwa.2020.04.033, development and Application of Open-source Software for Problems with Numerical PDEs.

[4] M. Bauer, S. Eibl, C. Godenschwager, N. Kohl, M. Kuron, C. Rettinger, F. Schornbaum, C. Schwarzmeier, D. Thönnes, H. Köstler, U. Rüde, walberla: A block-structured high-performance framework for multiphysics simulations, Computers and Mathematics with Applications 81 (2021) 478–501. URL: https://www.sciencedirect.com/science/article/pii/S0898122120300146. doi:https://doi.org/10.1016/j.camwa.2020.01.007, development and Application of Open-source Software for Problems with Numerical PDEs.

[5] J. Latt, O. Malaspinas, D. Kontaxakis, A. Parmigiani, D. Lagrava, F. Brogi, M. B. Belgacem, Y. Thorimbert, S. Leclaire, S. Li, F. Marson, J. Lemus, C. Kotsalos, R. Conradin, C. Coreixas, R. Petkantchin, F. Raynaud, J. Beny, B. Chopard, Palabos: Parallel lattice boltzmann solver, Computers and Mathematics with Applications 81 (2021) 334–350. URL: https://www.sciencedirect.com/science/article/pii/S0898122120301267. doi:https://doi.org/10.1016/j.camwa.2020.03.022, development and Application of Open-source Software for Problems with Numerical PDEs.

[6] T. Krueger, H. Kusumaatmaja, A. Kuzmin, O. Shardt, G. Silva, E. Viggen, The Lattice Boltzmann Method: Principles and Practice, Graduate Texts in Physics, Springer, 2016.

[7] Y. Ding, C. Xu, H. Qiu, Q. Wang, W. Dai, Y. Lin, Y. Che, Evaluating performance portability of sycl and kokkos: A case study on lbm simulations, in: 2023 IEEE Intl Conf on Parallel and Distributed Processing with Applications, Big Data and Cloud Computing, Sustainable Computing and Communications, Social Computing and Networking (ISPA/BDCloud/SocialCom/SustainCom), 2023, pp. 328–335. doi:10.1109/ISPA-BDCloud-SocialCom-SustainCom59178.2023.00075.

[8] Sycl specification, https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html, 2023.

[9] B. Ashbaugh, A. Bader, J. Brodman, J. Hammond, M. Kinsner, J. Pennycook, R. Schulz, J. Sewall, Data parallel c++: Enhancing sycl through extensions for productivity and performance, in: Int. Workshop on OpenCL, 2020, pp. 1–2. doi:10.1145/3388333.3388653.

[10] A. Alpay, V. Heuveline, Sycl beyond opencl: The architecture, current state and future direction of hipsycl, in: International Workshop on OpenCL, 2020, pp. 1–1. doi:10.1145/3388333.3388658.

[11] A. Gozillon, R. Keryell, L.-Y. Yu, G. Harnisch, P. Keir, trisycl for xilinx fpga, in: Int. Conference on High Performance Computing and Simulation (HPCS), 2020.

[12] Y. Ke, M. Agung, H. Takizawa, Neosycl: A sycl implementation for sx-aurora tsubasa, in: The International Conference on High Performance Computing in Asia-Pacific Region, HPC Asia 2021, 2021, p. 50–57. doi:10.1145/3432261.3432268.

[13] P. Thoman, F. Knorr, L. Crisci, Simsycl: A sycl implementation targeting development, debugging, simulation and conformance, in: Proceedings of the 12th International Workshop on OpenCL and SYCL, IWOCL '24, Association for Computing Machinery, New York, NY, USA, 2024. URL: https://doi.org/10.1145/3648115.3648136. doi:10.1145/3648115.3648136.

[14] P. Salzmann, F. Knorr, P. Thoman, P. Gschwandtner, B. Cosenza, T. Fahringer, An asynchronous dataflow-driven execution model for distributed accelerator computing, in: IEEE 23rd Int. Symposium on Cluster, Cloud and Internet Computing (CCGrid), 2023, pp. 82–93. doi:10.1109/CCGrid57682.2023.00018.

[15] K. Fan, M. D'Antonio, L. Carpentieri, B. Cosenza, F. Ficarelli, D. Cesarini, Synergy: Fine-grained energy-efficient heterogeneous computing for scalable energy saving, in: International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2023. doi:10.1145/3581784.3607055.

[16] L. Carpentieri, B. Cosenza, Towards a sycl api for approximate computing, in: International Workshop on OpenCL, 2023, pp. 1–2. doi:10.1145/3585341.3585374.

[17] S. J. Pennycook, J. D. Sewall, V. W. Lee, A metric for performance portability, arXiv preprint (2016). doi:arXiv:1611.07409.

[18] G. Falcucci, G. Amati, P. Fanelli, V. K. Krastev, G. Polverino, M. Porfiri, S. Succi, Extreme flow simulations reveal skeletal adaptations of deep-sea sponges, Nature 595 (2021) 537–541. URL: https://doi.org/10.1038/s41586-021-03658-1. doi:10.1038/s41586-021-03658-1.

[19] G. Amati, S. Succi, P. Fanelli, V. K. Krastev, G. Falcucci, Projecting lbm performance on exascale class architectures: A tentative outlook, Journal of Computational Science 55 (2021) 101447. URL: https://www.sciencedirect.com/science/article/pii/S1877750321001289. doi:https://doi.org/10.1016/j.jocs.2021.101447.

[20] K. Mattila, J. Hyväluoma, J. Timonen, T. Rossi, Comparison of implementations of the lattice-boltzmann method, Computers and Mathematics with Applications 55 (2008) 1514–1524. URL: https://www.sciencedirect.com/science/article/pii/S0898122107006232. doi:https://doi.org/10.1016/j.camwa.2007.08.001, mesoscopic Methods in Engineering and Science.

[21] J. Latt, C. Coreixas, J. Beny, Cross-platform programming model for many-core lattice boltzmann simulations, PloS One 16 (2021) e0250306. URL: https://doi.org/10.1371/journal.pone.0250306. doi:10.1371/journal.pone.0250306.

[22] M. Lehmann, M. J. Krause, G. Amati, M. Sega, J. Harting, S. Gekle, Accuracy and performance of the lattice boltzmann method with 64-bit, 32-bit, and customized 16-bit number formats, Phys. Rev. E 106 (2022) 015308. URL: https://link.aps.org/doi/10.1103/PhysRevE.106.015308. doi:10.1103/PhysRevE.106.015308.

[23] J. Ahrens, B. Geveci, C. Law, ParaView: An end-user tool for large data visualization, in: Visualization Handbook, Elesvier, 2005. ISBN 978-0123875822.

[24] L. Crisci, L. Carpentieri, P. Thoman, A. Alpay, V. Heuveline, B. Cosenza, Sycl-bench 2020: Benchmarking sycl 2020 on amd, intel, and nvidia gpus, in: Proceedings of the 12th International Workshop on OpenCL and SYCL, IWOCL '24, Association for Computing Machinery, New York, NY, USA, 2024. URL: https://doi.org/10.1145/3648115.3648120. doi:10.1145/3648115.3648120.

[25] How to optimize data transfers in cuda, https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/, 2012.

[26] D. S. Hollman, B. Adelstein-Lelbach, H. C. Edwards, M. Hoemmen, D. Sunderland, C. R. Trott, mdspan in C++: A case study in the integration of performance portable features into international language standards, CoRR abs/2010.06474 (2020). URL: https://arxiv.org/abs/2010.06474. arXiv:2010.06474.

[27] X. Lu, C. Ramos, F. Zheng, K. W. Schulz, J. Santos, K. Lowery, N. Curtis, C. D. Pietrantonio, Amdresearch/omniperf: v2.0.1 (03 june 2024), 2024. URL: https://doi.org/10.5281/zenodo.7314631. doi:10.5281/zenodo.7314631.

[28] J. Kwack, J. Tramm, C. Bertoni, Y. Ghadar, B. Homerding, E. Rangel, C. Knight, S. Parker, Evaluation of performance portability of applications and mini-apps across amd, intel and nvidia gpus, in: Int. Workshop on Performance, Portability and Productivity in HPC (P3HPC), 2021, pp. 45–56. doi:10.1109/P3HPC54578.2021.00008.

[29] Nvidia profiling tools, https://developer.nvidia.com/tools-overview, 2023.

[30] Intel advisor homepage, https://www.intel.com/content/www/us/en/developer/tools/oneapi/advisor.html, 2024.