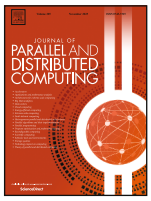
Journal of Parallel and Distributed Computing 207 (2026) 105188

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc



On the development of high-performance, multi-GPU applications on heterogeneous systems leveraging SYCL

Francisco J. Andújar [a], Rocío Carratalá-Sáez [b,*], Yuri Torres [a], Arturo Gonzalez-Escribano [a], Diego R. Llanos [a]

[a] Department of Computer Science, Universidad de Valladolid, Paseo de Belén 15, 47011, Valladolid, Spain
[b] Department of Computer Science, Universitat de València, Avinguda de la Universitat s/n, 46100, Burjassot, Spain

ARTICLE INFO

Keywords:
SYCL
CUDA
HIP
Finite Time Lyapunov Exponent
Performance evaluation
Development effort

ABSTRACT

Computational platforms for high-performance scientific applications are increasingly heterogeneous, incorporating multiple GPU accelerators. However, differences in GPU vendors, architectures, and programming models challenge performance portability and ease of development. SYCL provides a unified programming approach, enabling applications to target NVIDIA and AMD GPUs simultaneously while offering higher-level abstractions for data and task management. This paper evaluates SYCL's performance and development effort using the Finite Time Lyapunov Exponent (FTLE) calculation as a case study. We compare SYCL's AdaptiveCpp (Ahead-Of-Time and Just-In-Time) and Intel oneAPI compilers, along with different data management strategies (Unified Shared Memory and buffers), against equivalent CUDA and HIP implementations. Our analysis considers single and multi-GPU execution, including heterogeneous setups with GPUs from different vendors. Results show that, while SYCL introduces additional development effort compared to native CUDA and HIP implementations, it enables multi-vendor portability with minimal performance overhead when using specific design options. Based on our findings, we provide development guidelines to help programmers decide when to use SYCL versus vendor-specific alternatives.

1. Introduction

The complexity of scientific applications follows an increasing trend motivated by society's needs. Arising from many fields, computational applications require as much computational power as possible to contribute efficiently to scientific, commercial, and social progress. To accomplish this, high-performance computing (HPC) is vital. HPC relies on efficiently using the diversity of resources available in modern computational systems, which are becoming increasingly heterogeneous. This includes exploiting traditional multicore systems and devices such as Graphic Processing Units (GPU). In the particular case of GPUs, it has been proved that they offer excellent computational capabilities that can accelerate many computations by several orders of magnitude.

To take advantage of all the available hardware in a heterogeneous system, the first approach is usually to manually develop a specific solution for that particular hardware using vendor toolchains or parallel programming models. For example, CUDA [1] for NVIDIA GPUs, or HIP [2] for AMD GPUs. Thanks to efficiently managing the hardware resources, these tools and models have demonstrated great capabilities and versatility to obtain the best possible performance for those devices. Nevertheless, experts who do not belong to the HPC field, such as other engineers, physicists, or mathematicians, must deal with a non-negligible learning curve to take advantage of all these programming model capabilities. Moreover, using vendor-specific tools, the resulting applications are often not easily portable to alternative vendor devices, and additional programming efforts are needed to use different hardware.

In recent years, different approaches with an increasing level of abstraction have been presented for designing applications that can leverage the resources in heterogeneous systems with improved portability. OpenCL [3] is a good example of approaches that introduce a first layer of abstractions for dealing with the diversity of heterogeneous devices. It is an extension of the C/C++ programming language, capable of generating and running applications on different vendors' multiprocessors, FPGAs, and GPUs. However, OpenCL requires even more development effort than, for example, the use of vendor-specific programming models for GPUs, such as CUDA or HIP. Moreover, OpenCL requires explicitly managing the data transfers and synchronization using a low-level

* Corresponding author.
E-mail addresses: fandujarm@infor.uva.es (F.J. Andújar), rocio.carratala@uv.es (R. Carratalá-Sáez), yuri.torres@infor.uva.es (Y. Torres), arturo@infor.uva.es (A. Gonzalez-Escribano), diego@infor.uva.es (D.R. Llanos).

https://doi.org/10.1016/j.jpdc.2025.105188
Received 16 September 2024; Received in revised form 28 February 2025; Accepted 19 October 2025
Available online 24 October 2025

event model, further increasing the development effort if the programmer wants to perform asynchronous operations to overlap kernel executions and data transfers. For this reason, learning and using OpenCL is cumbersome for those who are not HPC experts but want to maximize their intensive-computation applications by exploiting the available resources in different heterogeneous environments.

In contrast, other proposals for higher-level heterogeneous programming simplify the code, require less explicit operations and cumbersome initialization, or even make operations such as data transfers transparent to the programmer. Some examples include SYCL [4], OpenMP [5], Kokkos [6], Raja [7], or other more academic approaches such as dOCAL [8] or CtrlEvents [9] that pursue a common objective: Offering higher-level abstractions that simplify and unify the programming of different computational resources transparently and effortlessly. While OpenMP is widely available in most modern compilers and the other alternatives previously cited have specific advantages, SYCL is becoming more and more popular as the available compiler implementations are becoming more mature, complete, robust, and efficient (see, e.g., AdaptiveCpp [10], or Intel oneAPI [11]). SYCL advocates a single-code approach, with automatic data-dependence analysis and data movements across memory hierarchies, which are easy to understand and to program by non-experts in low-level programming of heterogeneous devices. The SYCL community strives to make it the functional and performance portability baseline. As discussed in Section 9, several works compare the efficiency and portability between SYCL and other heterogeneous programming models for specific applications and platforms. Currently, it is highly relevant to investigate the efficiency and portability offered by the new SYCL implementations for real-world applications.

In this paper, we evaluate the current SYCL implementation, using a real-world application, from two different perspectives: The performance it offers when dealing with single or multiple GPU devices, from the same or different vendors, and the development effort required to implement the code. We compare the performance and the code with baselines programmed directly using CUDA or HIP technologies for NVIDIA and AMD GPUs, isolated or in combination. Moreover, we evaluate SYCL performance from three different design choices: The available SYCL compilers (AdaptiveCpp and Intel oneAPI), the compiler implementation (Single-source, single-compiler pass, and Single-source, multiple-compiler passes), and the Data Management (Unified shared memory and buffer model). These design choices will be explained in Section 2.1. In this comparison, we try to illuminate the advantages and limitations of the recent improvements introduced for this high-level programming model compared to traditional vendor-provided tools.

We have chosen as the case study the UVaFTLE [12] application, which computes the Finite Time Lyapunov Exponent (FTLE), to explore this development effort and performance evaluation. On the one hand, this application is formed by two conceptually very different kernels: One deals with larger data sets and memory accesses. At the same time, the other one focuses on solving a collection of linear algebra operations. This difference lets us explore whether the key aspects of most scientific applications (memory accesses and computations) are better addressed by native (vendor-provided) tools than by SYCL. On the other hand, we have not found any work in the literature that offers a recent and portable version of the FTLE solution, so we also provide the community with a novel portable and improved FTLE implementation, based on our previous work [12].

The main contributions of this work are:

- We offer a portable version of the UVaFTLE application using SYCL, with support to target multiple GPU devices simultaneously, even from different vendors.
- We present new baseline implementations of the UVaFTLE application. The first one uses CUDA. It increases the use of registers to minimize global memory accesses and a new kernel to implement the data preprocessing stage in GPU. The second baseline is a port of the same program using HIP to target AMD GPU devices. Both versions support the specific vendor's multi-GPU.
- We conduct an in-depth evaluation of the performance, in terms of execution time, offered by both the baseline implementations of the FTLE computation (based on CUDA and HIP) and the SYCL version, considering the main SYCL compiler, the compiler implementation, and the data management model.
- We compare the development effort required to implement the CUDA and HIP baselines with the SYCL version in terms of several classical development-effort metrics.
- Based on the evaluation conducted and its analysis, we provide a development strategy with recommendations on how and when to use SYCL or the native alternatives.
- This work contributes to open science. All our implementations are fully open-source and available by accessing the GitHub repository [13].

The rest of the paper is structured as follows: In Section 2, we provide a revision of the different SYCL implementations and the mathematical background of the FTLE; in Section 3 we describe the FTLE computation algorithm and our implementations, covering how we leverage CUDA, HIP, and SYCL; in Section 4 we describe how we ported UVaFTLE to SYCL and address some of the implementation decisions needed; in Section 5 we present an in-depth evaluation of the performance delivered by each implementation decision; in Section 6 we analyze the development effort associated with each implementation; in Section 7 we summarize the main findings of our evaluation and analysis; in Section 8 we provide development strategy recommendations and guidelines for using SYCL; in Sect. 9 we summarize the main existing works that use SYCL in their implementations and those related to the FTLE computation, comparing them to our work; and in Section 10 we summarize the main conclusions derived from this work and finalize by mentioning the future work.

2. Background

In this section, we first summarize the state of the art of SYCL, describing its different implementations highlightStartblueand the main features of each. Then, we describe the case study we utilize in this work: FTLE.

2.1. Heterogeneous computing and SYCL

In 2014, the Khronos Group presented SYCL [4], a standard model for cross-platform programming, to achieve both code and performance portability and lower the development effort. SYCL organizes the kernels using a task graph implicitly constructed by the SYCL runtime. This also allows implicitly managing the dependencies between the kernels and the data communications, although the developer can still manage them explicitly. In this work, we analyze the SYCL ecosystem through three main axes:

- **SYCL compilers**: The SYCL ecosystem has several implementations that rely on different compiler backends for different types of devices. Currently, the most widespread compilers are Intel's oneAPI [11] and AdaptiveCpp [10,14] (formerly known as hipSYCL). oneAPI supports Intel hardware (CPUs, GPUs, and FPGAs) and has two plugins developed by Codeplay to support NVIDIA and AMD devices using alternative backends. However, these backends are incompatible with the rest of Intel's hardware. AdaptiveCpp supports CPUs, AMD GPUs, NVIDIA GPUs, and Intel GPUs through OpenMP, HIP/ROCm, CUDA, and Level Zero, respectively. Other implementations are TriSYCL [15], which only supports CPUs and Xilinx FPGAs, and Codeplay's ComputeCPP [16], which supports CPUs and INTEL, NVIDIA, and AMD GPUs, but was discontinued after September 2023 [17]. For these reasons, the AdaptiveCpp and oneAPI compilers have been chosen for conducting this study.

- **Compiler design options:** There are multiple ways of implementing the SYCL compiler. According to the SYCL specification [4], there are three different choices:
  - **Library only-implementation**: It is possible to implement SYCL as a pure C++ library. For example, this approach is available in AdaptiveCpp to target NVIDIA GPUs.
  - **Single-source, single-compiler pass** (SSCP): The compiler parses the code only once, simultaneously compiling the host and the device code. The application binary can be used on different devices (e.g., two GPU models) without recompiling the code. AdaptiveCpp has recently presented the first version of an SSCP SYCL compiler [18]. Briefly, AdaptiveCpp uses LLVM at compile time to generate an intermediate and backend-independent representation. This representation is transformed at runtime into the format necessary for the backend driver.
  - **Single-source, multiple-compiler passes** (SMCP): The compiler parses at least twice times the code, one time for the host code and another for the device code. The device code is compiled once for each device to use the application on different devices. The application binary, also called fat-binary, contains all the device images. This is a usual approach, but it requires a higher compilation time.

  Since only AdaptiveCpp implements the SSCP model, we analyze the performance of SSCP and SMCP approaches using AdaptiveCpp. The library-only mode will not be investigated since only NVIDIA GPUs are supported. The main focus of this work is to study the performance/development effort of porting applications to SYCL, not using SYCL as a library for third-party compilers.
- **Data management**: SYCL has three abstractions to manage data:
  - **Unified Shared Memory** (USM) manages the data using a pointer-based approach based on C and C++ pointers. USM facilitates the migration of C/C++ codes to SYCL and is an ideal choice if our C/C++ code is pointer-based. However, not all the devices support this memory management. Data can be allocated to the host, the device, or both sides. We will study the FTLE application using these allocation modes:
    - *Device*: The data is allocated in the device; it is not accessible by the host, and the data movement is the programmer's responsibility. This allocation is made calling to *malloc_device()* function, which is equivalent to call *cudaMalloc()/hipMalloc()* in CUDA/HIP.
    - *Shared*: The data is allocated and accessible on both sides and automatically migrated between host and device when necessary. This allocation is made calling to *malloc_shared()* function, which is equivalent to call *cudaMallocManaged()/ hipMallocManaged()* in CUDA/HIP.

    From now on, we will use the terms *Device* and *Shared* to indicate the memory allocation mode, regardless of whether we are talking about CUDA, HIP, or SYCL codes.
  - **Buffers** provides a high-level abstraction to manage the data. The runtime manages the data storage and movement between different memory spaces. Thus, the programmer can skip this part of the data management tasks. However, using buffers requires more significant development effort on the programmer than USM, as new data abstractions (buffers and accessors) should be explicitly managed.
  - **Images** provides a high-level abstraction to manage image data. Its interface and data management are essentially the same as the *buffer* model, but this abstraction focuses on developing image and video applications.

Currently, AdaptiveCpp and oneAPI support all the data management models. However, FTLE is not an image/video application; therefore, employing *images* is out of the scope of this work. Thus, we will analyze the performance of *USM* (both device and shared allocations) and *buffer* models using both compilers.

2.2. Case of study: FTLE

Fluid dynamics is a widely explored field. In particular, the fluid particle trajectories in phase space, often referred to as *Lagrangian*, are of great interest. More specifically, calculating the *Lagrangian Coherent Structures (LCS)* [19] is key for several disciplines, such as cardiovascular engineering [20], aerodynamics [21], and geophysical fluid dynamics [22].

The fluid particle trajectories are defined as solutions of

$$\dot{\vec{x}} = \vec{v}(\vec{x}, t),$$

where the right-hand side is the fluid's velocity field in the absence of molecular diffusion. Solving this system of equations allows for the calculation of the LCS. The main interest in computing the LCS is that they let a better understanding of the flow phenomena since they can be broadly interpreted as *transport barriers* in the flow.

From the computational point of view, the extraction of LCS consists of two main steps: The flowmap computation and the resolution of the FTLE. We will focus on the second step, which is mathematically defined as

$$\Lambda_{t_0}^{t_1}(\vec{x}_0) = \frac{1}{t_1 - t_0} \log \sqrt{\lambda_n(\vec{x}_0)}$$

where λ_n is the maximum eigenvalue of the Cauchy-Green strain tensor C, defined as follows

$$C(\vec{x}_0) = \left[\nabla F_{t_0}^{t_1}(\vec{x}_0)\right]^T \nabla F_{t_0}^{t_1}(\vec{x}_0)$$

being F the flowmap [21].

The FTLE is a scalar field that works as an objective diagnostic for LCS: A first-order approach to assess the stability of material surfaces in the flow under study by detecting material surfaces along which infinitesimal deformation is larger or smaller than off these surfaces [19]. Although more reliable mathematical methods have been developed for the explicit identification of LCS, the FTLE remains the most used metric for LCS identification.

From the computational point of view, it is essential to highlight that the FTLE computation is applied to each particle of the flow independently of the other particles. Thus, it represents an embarrassingly-parallel problem [23]. We have already described, explored, and evaluated the FTLE computation in a previous work [12], where we presented UVaFTLE. This tool incorporates a CUDA-based kernel to use multiple NVIDIA GPUs in the FTLE computation.

3. Application description and implementation

In this section, we describe the FTLE algorithm. Next, we identify the regions of code suitable to be executed in GPUs; afterward, we present the native (CUDA and HIP) and the SYCL implementations of the GPU kernels; and, finally, we illustrate how to target multiple GPUs using SYCL. Note that the complete code of all versions is available in the UVaFTLE repository [13].

3.1. FTLE Algorithm

Provided the information of the mesh that defines the flow to study (namely the dimension, time instant when the FTLE will be computed, the mesh points coordinates and faces information, and the flowmap), the process of computing the FTLE (described in Algorithm 1) consists of the following steps performed over each point in the mesh:

1. Compute the gradients of the flowmap (see Algorithm 2). Note that the gradient calculation is based on the Green Gauss theorem [24].
2. Generate the tensors from the gradients and perform the matrix-matrix product of the previously generated tensors by their transposes (see Algorithm 2).

Algorithm 1 FTLE.

Require: $nDim, t\_eval, coords\_file, faces\_file, flowmap\_file$

1: $nVpF = (nDim == 2)\ ?\ 3 : 4$ ▷ Triangles or tetrahedrons
2: $\{nPoints, coords\}$ = read_coordinates(coords_file)
3: $\{nFaces, faces\}$ = read_faces(faces_file)
4: $flow$ = read_flowmap(flowmap_file)
5: $nFpP$ = create_nFacesPerPoint_vector(nPoints, nFaces, nVpF, faces)
6: FpP = create_FacesPerPoint_vector(nPoints, nFaces, nVpF, faces, nFpP)
7: **for** ip *in* $range(nPoints)$ **do**
8: **if** $nDim == 2$ **then**
9: $g1, g2$ = 2D_grad_tens (ip, nVpF, coords, flow, faces, nFpP, FpP)
10: max_eigen = max_eigenvalue_2D([g1, g2])
11: **else**
12: $g1, g2, g3$ = 3D_grad_tens (ip, nVpF, coords, flow, faces, nFpP, FpP)
13: max_eigen = max_eigenvalue_3D([g1, g2, g3])
14: **end if**
15: $result[ip] = log(sqrt(max\_eigen))/t\_eval$
16: **end for**
17: **return** $result[\ ]$

Algorithm 2 2D_grad_tens.

Require: $ip, nP, nVpF, coords[\ ], flow, faces[\ ], nFpP[\ ], FpP[\ ]$

1: $nFaces = (ip == 0)\ ?\ nFpP[ip]\ :\ nFpP[ip] - nFpP[ip-1]$
2: $left, right, below, above = GreenGauss(nFaces, FpP, nFpP, nVpF, coords)$ ▷ This provides the indices of the left, right, below, above closest points
3: $dx = coords[right \cdot nDim] - coords[left \cdot nDim]$
4: $dy = coords[above \cdot nDim + 1] - coords[below \cdot nDim + 1]$
5: $gra1[0] = (flow[right \cdot nDim] - flow[left \cdot nDim])/dx$
6: $gra1[1] = (flow[right \cdot nDim + 1] - flow[left \cdot nDim + 1])/dx$
7: $gra2[0] = (flow[above \cdot nDim] - flow[below \cdot nDim])/dy$
8: $gra2[1] = (flow[above \cdot nDim + 1] - flow[below \cdot nDim + 1])/dy$
9: $ftle\_m[0] = gra1[0] \cdot gra1[0] + gra1[1] \cdot gra1[1]$
10: $ftle\_m[1] = gra1[0] \cdot gra2[0] + gra1[1] \cdot gra2[1]$
11: $ftle\_m[2] = gra2[0] \cdot gra1[0] + gra2[1] \cdot gra1[1]$
12: $ftle\_m[3] = gra2[0] \cdot gra2[0] + gra2[1] \cdot gra2[1]$
13: $gra1[0] = ftle\_m[0]; gra1[1] = ftle\_m[1]$
14: $gra2[0] = ftle\_m[2]; gra2[1] = ftle\_m[3]$
15: $ftle\_m[0] = gra1[0] \cdot gra1[0] + gra1[1] \cdot gra1[1]$
16: $ftle\_m[1] = gra1[0] \cdot gra2[0] + gra1[1] \cdot gra2[1]$
17: $ftle\_m[2] = gra2[0] \cdot gra1[0] + gra2[1] \cdot gra1[1]$
18: $ftle\_m[3] = gra2[0] \cdot gra2[0] + gra2[1] \cdot gra2[1]$
19: **return** $ftle\_m$

3. Compute the maximum eigenvector of each resulting matrix (see Algorithm 3). Note that, as we are computing the eigenvalues of matrices of size 2x2 (2D) or 3x3 (3D), which in practice means respectively solving a 2nd- and 3rd-degree equation, we have directly implemented this computation, instead of calling mathematical libraries that perform this computation for generic matrices of any size.
4. Calculate the logarithm of the square matrix of the maximum eigenvalue and divide the result by the time instant to evaluate.

Note that we only present the algorithms for the 2D case here because the 3D case is straightforward.

In addition to the algorithms already described, it is also important to remark those utilized in lines 5 and 6 in Algorithm 1: *create_nFacesPerPoint_vector* (see Algorithm 4) and *create_FacesPerPoint_vector* (see Algorithm 5). Although they are part of the preprocessing and not the FTLE computation itself, they are needed to create the data structures called nFpP and FpP, which respectively contain the number of faces to which each mesh point belongs and the corresponding face identifiers. These data structures accelerate the process of computing the FTLE, because they establish the relationship between the different mesh points and faces, meaning that this is analyzed only once at the beginning of the code, instead of each time the Green Gauss algorithm is called.

3.2. GPU Kernels identification

The cost of computing the FTLE algorithm described in the previous section relies on two main procedures: The *create_facesPerPoint_vector* function and the linear algebra operations performed for each mesh point in each iteration of the *for* loop in line 7 of the Algorithm 1. As a consequence, this is what is worth it to be computed in the GPU; in other words, these are the two GPU kernels to build to accelerate the FTLE computation:

- **Preprocessing**: This kernel implements the *create_facesPerPoint_vector* function (see Algorithm 5). The create_facesPerPoint_vector kernel implements a memory-bound algorithm to determine the faces associated with each point within a mesh. The kernel iterates through all nFaces and checks nVertsPerFace vertices, resulting in O(nPoints × nFaces × nVertsPerFace) memory accesses, which are non-coalesced and lack shared memory optimization. This leads to high memory latency, making global memory access the dominant

Algorithm 3 max_eigenvalue_2D.

Require: M

$sq \leftarrow sqrt(M[21] * M[21] + M[10] * M[10] - 2 * (M[10] * M[21]) + 4 * (M[11] * M[20]))$
$eig1 \leftarrow (M[21] + M[10] + sq)/2$
$eig2 \leftarrow (M[21] + M[10] - sq)/2$
return $(eig1 > eig2)\ ?\ eig1 : eig$

Algorithm 4 create_nFacesPerPoint_vector.

Require: $nPoints, nFaces, nVpF, faces[\ ]$

for ip *in range*$(nPoints)$ **do**
  $nFpP[ip] = 0;$
end for
for $iface$ *in range*$(nFaces)$ **do**
  for ipf *in range*$(nVpF)$ **do**
    $ip = faces[iface \cdot nVpF + ipf]$
    $nFpP[ip] = nFpP[ip] + 1$
  end for
end for
for ip *in range*$(nPoints)$ **do**
  $nFpP[ip] = nFpP[ip] + nFpP[ip - 1]$
end for
return $nFpP$

Algorithm 5 create_facesPerPoint_vector.

Require: $nPoints, nFaces, nVpF, faces[\ ], nFpP[\ ]$

for ip *in range*$(nPoints)$ **do**
  $count = 0$
  $iFacesP = (ip == 0)\ ?\ 0 : nFpP[ip - 1]$
  $nFacesP = (ip == 0)\ ?\ nFpP[ip] : nFpP[ip] - nFpP[ip - 1]$
  while $(iface < nFaces)$ *and* $(count < nFacesP)$ **do**
    for ipf *in range*$(nVpF)$ **do**
      if $faces[iface \cdot nVpF + ipf] == ip$ **then**
        $FpP[ifacesP + count] = iface$
        $count = count + 1$
      end if
    end for
  end while
end for
return FpP

bottleneck rather than computation. The arithmetic workload is minimal, consisting mainly of integer comparisons and assignments, confirming that the limiting factor is memory bandwidth rather than operational intensity.

- **FTLE**: This kernel was already described in our previous work [12]; we presented a single CUDA-based kernel to compute everything described in Algorithms 2 and 3 (or their corresponding 3D versions), which means using the GPU device to compute lines 9–10 (2D case) or 12–13 (3D case) of the Algorithm 1. Note that this kernel has two variants: 2D and 3D. The gpu_compute_gradient_2D kernel builds upon the first algorithm by introducing floating-point operations for gradient calculations and eigenvalue extraction. However, the operational intensity is reduced due to its complex memory access patterns. Thus, the kernel is also memory-bound. The execution consists of three phases: neighbor search (O(nPoints × nFaces × nVertsPerFace) memory accesses), gradient computation (O(nPoints)), and FTLE matrix eigenvalue extraction (O(nPoints)). The non-coalesced accesses to faces, coordinates, and flowmap create significant memory stalls, limiting performance. Additionally, branch divergence in the neighbor selection further exacerbates execution time variability. Despite its higher arithmetic workload, the kernel's performance is still limited by memory latency rather than computation.
- **3D kernels**: Extending these algorithms to 3D versions (e.g., gpu_compute_gradient_3D) results in similar performance characteristics, as the additional spatial dimension only increases the memory access complexity while maintaining the same compute-to-memory imbalance. The search for neighboring points becomes even more expensive, scaling to O(nPoints × nFaces × nVertsPerFace) in three dimensions, further amplifying the impact of non-coalesced memory accesses and branch divergence.

In the following sections, we present details on implementing these kernels using CUDA or HIP (named native implementations) and SYCL.

3.3. Native implementations

Three different GPU kernels (*create_facesPerPoint_vector*, *gpu_compute_gradient_2D*, and *gpu_compute_gradient_3D*) have been developed corresponding to the algorithms described in previous sections. The *gpu_compute_gradient_2D* and the *gpu_compute_gradient_3D* kernels are improved versions of the CUDA-based implementation of our previous work, UVaFTLE [12]. Moreover, they have been appropriately ported to HIP to tackle AMD GPUs.

Whether they use CUDA or HIP, the three kernels perform the same two initial operations before starting the algorithm. The first operation

corresponds to the calculation of the thread global identifier. Each identifier corresponds to a mesh point. For code simplicity, we use one-dimensional `threadBlock` and `grid`, making it easier to calculate the global index of each thread and reducing the number of kernel instructions. The following instruction is executed to calculate the thread global identifier:

$$int th\_id = blockIdx.x * blockDim.x + threadIdx.x;$$

The second operation checks that the number of launched threads is not larger than the number of points in the mesh. For that, we insert the following condition wrapping each kernel implementation:

$$if(th\_id < numCoords)\{\dots\}$$

For each kernel, each thread of the GPU grid executes precisely the sequence of steps associated with the FTLE kernel described in Section 3.2. The implementation can currently leverage all the GPU devices available in a single node, as in our previous work [12]. Thus, we are deploying our multi-GPU executions in a shared-memory environment. We use the OpenMP programming model, instantiating as many threads as GPU devices to distribute the load among them. In particular, we have designed a static partitioning of the mesh points based on the number of GPU devices that take part in the execution.

In contrast to our previous work, pinned memory has been used to perform the data transfers of the results from the GPU to the host through *cudaHostAlloc* or *hipHostAlloc* primitives. Classical GPU reference manuals, such as [1], indicate that this kind of memory can be used when executions or asynchronous transfers are introduced, thus reducing the latencies in these data transfers.

Furthermore, the GPU community indicates that the best `threadBlock` size maximizes the streaming multiprocessor occupancy, such as 256, 512, and 1024. Since it is recommended, we have selected 512 as the `threadBlock` size. As this work does not intend to apply any tuning strategies, we have not evaluated additional sizes.

4. Porting UVaFTLE to SYCL: Implementation decisions

Based on the native implementations, the application has been ported to SYCL using the USM and the buffer models. Since the complete code of UVaFTLE is very large, we will illustrate the changes made in our application using a simpler code. Note that the complete SYCL code of the UVaFTLE can be found in our repository [13]. The example launches a simple kernel that, given an array A with n elements, calculates $A[i] = 2 \times A[i] + 1$ for each element i, being $0 \leq i < n$. Figs. 1, 3–6, show the code examples for SYCL buffers, CUDA (device mem.), SYCL USM-device, CUDA (shared mem.), and SYCL USM-shared versions, respectively. The background of both codes has been colored to help the reader identify the groups of lines in both codes with the same functionality. The parts with white backgrounds correspond to the host code, which has no differences between versions. Also, note that the HIP code was not included in the comparison since the differences between the CUDA and HIP versions are practically negligible.

4.1. Porting the application to SYCL buffers model

First, we focus on the SYCL buffer code (Fig. 1). The first step to making a SYCL application is choosing the device to execute the code (code with a blue background). For these purposes, SYCL employs a *queue*, an abstraction where the kernels executed on a single device are submitted. This is performed in line 4 of Fig. 1, where a new queue is created and attached to a GPU device. Note that, through the usage of *gpu_selector{}*, the kernel to be executed can be attached to any GPU in the system (usually the first GPU detected by the SYCL runtime).

```
1  using namespace sycl;
2  int main(){
3      // Set the execution queue by selecting a GPU
4      queue my_queue(gpu_selector{});
5      // Host memory allocation
6      int elements=100000;
7      float* h_array = (float*)malloc(elements*sizeof(float));
8      [...] // Host memory initialization
9      // Range declaration
10     range array_range{elements};
11     range block_range{512};
12     {   // Enters in a new scope
13     // Buffer declaration
14     buffer dev_buf{h_array, array_range};
15     // Submit the kernel to the queue
16     my_queue.submit([&](handler &my_handler) {
17     // Create the accessor to use the device array
18        accessor d_array{dev_buff, my_handler, read_write};
19         // Execute the kernel with a parallel for
20         my_handler.parallel_for(nd_range(array_range,
               block_range), [=](nd_item<1> i){
21             // Get the thread global identifier
22             int gpu_id = i.get_global_id(0);
23             // Kernel computation
24             if(gpu_id < elements)
25                 d_array[gpu_id]=d_array[gpu_id]*2+1;
26         }); // end parallel for
27     }); // end submit
28     } // Finish the scope to update host memory
29     [...] // Use the results of the kernel in the host
30     free(h_array);
31     return 0;
32 }
```

Fig. 1. Comparison between CUDA and SYCL buffer version. The lines with the same colors share the same purpose in all codes.

```
queue getHIPqueue(){
    auto platform = platform::get_platforms();
    for (int p=0; p < platform.size(); p++){
        if(!platform[p].get_info<info::platform::name>().
            contains("HIP")){
            auto devs= platform[p].get_devices();
            return queue(devs[0]);
        }
    }
}
```

Fig. 2. Example of a function for getting a queue attached to a HIP device in SYCL.

```
1  int main(){
2      cudaSetDevice(0); // Set the device
3      // Host memory allocation
4      int elements=100000;
5      float* h_array = (float*)malloc(elements*sizeof(float));
6      [...] // Host memory initialization
7      // Declare block and grid
8      dim3 block(512);
9      int numBlocks = (int) (ceil((double)elements/block.x)+1);
10     dim3 grid(numBlocks);
11     // Device memory allocation
12     float* d_array;
13     cudaMalloc(d_array, elements*sizeof(float));
14     // Synchronous Copy host memory to the device
15     cudaMemcpy(d_array, h_array, elements*sizeof(float),
           cudaMemcpyHostToDevice);
16     // Kernel launch
17     my_kernel<<<grid, block, 0, cudaStreamDefault>>>(d_array,
           elements);
18     // Asynchronous copy from device to host
19     cudaMemcpyAsync (h_array, d_array, elements*sizeof(float),
           cudaMemcpyDeviceToHost, cudaStreamDefault);
20     // Now we can do other things while data async. transfer
21     [...]
22     //But we need to sync to use the updated h_array
23     cudaDeviceSynchronize();
24     cudaFree(d_array);// Free device memory
25     [...] // Use the results of the kernel in the host
26     free(h_array);
27     return 0;
28 }
29 // Kernel declaration
30 __global__ void my_kernel (float* d_array, int elements){
31     // Get the thread global identifier
32     int gpu_id = blockIdx.x*blockDim.x + threadIdx.x;
33     // Kernel computation
34     if(gpu_id < elements)
35         d_array[gpu_id] = d_array[gpu_id]* 2 +1;
36 }
```

Fig. 3. Comparison between CUDA (with device memory) and SYCL. The lines with the same background colors share the same purpose in all codes.

However, the SYCL API offers methods to attach a GPU from a specific platform, model, etc. For example, Fig. 2 shows a function for creating a queue attached to a HIP device, getting at first the list of devices for the HIP platform. Attaching the queue to a CUDA device is also possible by simply comparing the string "CUDA" with the platform name.

After that, both CUDA and SYCL buffer codes allocate and initialize the host array. Next, the native implementation specifies the CUDA `numBlocks` and `grid` sizes (code with purple background). In SYCL, we must specify the range of our arrays (*array_range* in the example) and the range of the thread block (*block_range*). *array_range* will be used later to create the buffer. Both ranges will be necessary to launch the kernel. Therefore, we create the needed ranges to port our application to SYCL. Note that, for the simplicity of the example, we only use 1-dimensional ranges, but we can also specify 2-dimensional or 3-dimensional ranges.

The next step in CUDA is to allocate the device data and to copy the data from the host to the device (line 11 of Fig. 3, green-background code). In SYCL, buffers will be used to manage the data instead of allocating and copying it on the device. Buffers provide an abstract view of the memory accessible from the host and the devices. The buffers also allow the SYCL runtime to manage the memory transfers transparently to the programmer. On the contrary, in the native implementation, we manually allocate and free the device memory and manually manage the data transfers (both synchronous and asynchronous versions) between the host and devices or between devices. Therefore, the buffer

abstraction simplifies the memory management. For example, let's suppose three kernels: K_1 and K_2, which have no data dependencies, and K_3, which needs the results of the first two kernels to make their work. The SYCL runtime transparently transfers the host data to the devices running K_1 and K_2 using the buffer abstraction. Since both kernels have no data dependencies, both kernels can run concurrently on different devices. Once the kernels have finished, the SYCL runtime will transfer the necessary data to run K_3 in its device and finally transfer the resulting data to the host.

Thus, we must declare the buffers for managing the memory. The buffer declaration requires specifying the host memory to be managed and the buffer range (line 14 of Fig. 1). Therefore, to port UVaFTLE to SYCL, we have created the necessary buffers to manage all the application data.

After that, we specify the kernel declaration (code with dark red background) and its launch (code with light red background). In the native implementation, we should declare the kernel as a function (lines 30–36 in Fig. 3) and launch this function inside the host code using a specific syntax (line 17 in Fig. 3). In SYCL, the *submit()* method is used to submit the kernel in the desired queue (line 16 in Fig. 1). Using lambda functions, we perform the submission and define the kernel code. In the example, a *parallel_for* and *nd_range* kernel (lines 13–17 in Fig. 1)) are employed to perform the same work as the CUDA kernel, i.e., to launch a kernel with *elements* threads organized in blocks of 512 threads. Since the main purpose is not to describe the SYCL API, we will not go into more detail about the declaration of lambda functions. Please consult the reference guide [4] for further information.

However, the programmer does not directly access the buffers in the kernels. To read and write buffers, we must create an *accessor* object (line 18 in Fig. 1), specifying the accessed buffer and the access mode (read, write, or read_write). The kernel code is the same in both versions. If we appropriately name the accessors, making changes in kernel code is unnecessary. The only difference between native and SYCL kernel codes is how to obtain the global index to access the data (line 22, Fig. 1). Note also that the CUDA/HIP index ordering differs from SYCL index ordering. When we work with structures of more than one dimension, we must interchange the x- and y-index to exploit the data coalescence.

Finally, note that the buffer, kernel submit, etc., are created inside a new scope. A buffer updates the host memory when it is destroyed. Using a new scope, the host memory will be updated when the scope ends and destroys the buffer, avoiding explicitly transferring data and synchronizing the host and device. However, SYCL allows manually updating the host memory inside the scope if the programmer requires it.

4.2. Porting the application to SYCL USM-device model

As shown in Fig. 4, the only differences between buffers and USM-device models lie in memory management (code with green background). Queue declaration, range declaration, kernel declaration, and kernel launch are the same in both models. Using USM and device allocation, we must allocate the memory on the host (line 7, Fig. 4) and the device (line 13, Fig. 4). To port this to the SYCL USM-device model, we change the *cudaMalloc* call by a *malloc_device* call[1]. Since the array is allocated on the device, we must explicitly transfer the data from host to device before launching the kernel. This can be done by replacing the *cudaMalloc* call by the function *memcpy* of the queue class (line 14, Fig. 4).

After that, we launch the kernel (line 16, Fig. 4). Again, if we appropriately name the device arrays, making changes in kernel code is unnecessary. Note that *submit()* scope is not present in the code. Since the SYCL 2020 specification, the queues can directly use the *parallel_for()* without the *submit()* scope, therefore reducing the code lines.

[1] In the example, we have used C–style allocators, but C++–style and C++–allocator-style are also available. Please consult the reference guide [4] for more information.

Finally, we need to transfer the data from the device to the host calling again *memcpy* (line 22, Fig. 4). However, to use the data in the host code, the queue must finish all its work (kernel execution and data transfer). Then, we need to manually synchronize the host and device using *wait* function of the queue class (line 24, Fig. 4. After that, we can free the device memory (replacing *cudaFree* by *free*) and use the data in the host.

4.3. Porting the application to SYCL USM-shared model

Using the shared memory (named in CUDA as managed memory) allows us to use the same data structure both in the host and the device. The data transfer is transparent to the programmer, and the runtime migrates the data between the host and the device when necessary. This also reduces the lines of code required to write our application.

First, we will briefly compare the CUDA code using device memory (Fig. 3) and shared memory (Fig. 5). As can be seen, only one allocation using *cudaMallocManaged* is necessary (line 5, Fig. 5), and one deallocation using *cudaFree* (line 11, Fig. 5). No explicit data transfers or synchronizations are required to use the data in the host or the device.

Fig. 6 shows the SYCL USM-shared code. As in the previous SYCL models, queue declaration, range declaration, and kernel launch are the same. We only need to change the memory management. If we start from the shared-memory CUDA code, we must change the calls to *cudaMallocManaged* and *cudaFree* by *malloc_shared* and *free*, respectively. Starting from the device-memory CUDA code, we must replace the host memory allocation and deallocation with these SYCL functions and remove all the CUDA functions to manage the device memory. Appropriately naming the shared arrays means that changes in the code are minimal.

4.4. SYCL Porting process: Summary

We now summarize the steps to port UVaFTLE to SYCL, starting with the first version written in CUDA and using device memory.

SYCL using buffers model.

1. Create a *queue* attached to the desired GPU device.
2. Copy the original host code as the declaration and initialization of the host memory, management of the application's final results, etc., avoiding copying the kernel code and the calls to the CUDA API (as cudaMalloc(), cudaMemcpy, etc.).
3. Start a new scope and define the *buffers* to manage the data.
4. At the preprocessing kernel launch location in the CUDA code, submit this kernel to the *queue* using the *submit* scope.
   (a) Create the *accessors* with the appropriate names to avoid rewriting the kernel code.
   (b) Launch the kernel using an *nd-range parallel for*.
   (c) Copy the kernel code, changing the index calculation to SYCL syntax.
5. Repeat the step and sub-steps of step 4 for the FTLE kernel.
6. End the scope to update the host memory.

SYCL using USM-device model.

1. Create a *queue* attached to the desired GPU device.
2. Copy the original host code, including the calls to CUDA API, but not the kernel definitions and callings.
3. Change the *cudaMalloc()*, *cudaMemcpy* and *cudaFree()* calls for the SYCL functions *malloc_device()*, *queue.memcpy()* and *free()*.
4. At the preprocessing kernel launch location in the CUDA code, submit this kernel to the *queue* using the *nd-range parallel for* scope.
   (a) Copy the kernel code, changing the index calculation to SYCL syntax.
   (b) Wait for kernel completion using *queue.wait()* to use the correct data in the next step (or declare and ordered-queue on step 1.).
5. Repeat the step and sub-steps of step 4 for the FTLE kernel.

```
using namespace sycl;
int main(){
    // Set the execution queue by selecting a GPU
    queue my_queue(gpu_selector{});
    // Host memory allocation
    int elements=100000;
    float* h_array = (float*)malloc(elements*sizeof(float));
    [...] // Host memory initialization
    // Range declaration
    range array_range{elements};
    range block_range{512};
     // Mem. device allocation and initialization
    float* d_array=malloc_device<float>(elements, my_queue);
    my_queue.memcpy(d_array, h_array,  elements*sizeof(float));
    // Execute the kernel with a parallel for
    my_queue.parallel_for(nd_range(array_range,block_range),
        [=](nd_item<1> i){
        int gpu_id = i.get_global_id(0);
        if(gpu_id < elements)
            d_array[gpu_id]=d_array[gpu_id]*2+1;
    }); // end parallel for
    //Copy the final results to the host and wait for them
    my_queue.memcpy(h_array, d_array,  elements*sizeof(float));
    //wait for kernel and data transfer completion
    my_queue.wait();
    // Free device memory
    free(d_array, my_queue);
    [...] // Use the results of the kernel in the host
    free(h_array);
    return 0;
}
```

Fig. 4. Comparison between CUDA and SYCL USM version with device memory. The lines with the same background colors share the same purpose in all codes.

```
int main(){
    cudaSetDevice(0);// Set the device
    int elements=100000;
    float* sh_array; // Shared memory allocation
    cudaMallocManaged(sh_array, elements*sizeof(float));
    [...] // Memory initialization
    // Declare block and grid (no changes from CUDA device)
    [...]
    my_kernel<<<grid, block, 0, cudaStreamDefault>>>(sh_array,
        elements);
    [...] // Use the results of the kernel in the host
    cudaFree(sh_array); //Free the shared array
    return 0;
}
```

Fig. 5. Comparison between CUDA (main() with shared memory) and SYCL. The lines with the same background colors share the same purpose in all codes.

SYCL using USM-shared model.

1. Create a *queue* attached to the desired GPU device.
2. Copy the original host code as the declaration and initialization of the host memory, management of the application's final results, etc., avoiding copying the kernel code and the calls to the CUDA API (as cudaMalloc(), cudaMemcpy, etc.).
3. Change the *malloc()* and *free()* calls for the SYCL functions *malloc_shared()* and *free()*.
4. At the preprocessing kernel launch location in the CUDA code, submit this kernel to the *queue* using the *nd-range parallel for* scope.
   (a) Copy the kernel code, changing the index calculation to SYCL syntax.
   (b) Wait for kernel completion using *queue.wait()* to use the correct data in the next step (or declare and ordered-queue on step 1.).
5. Repeat the step and sub-steps of step 4 for the FTLE kernel.

To facilitate the reader's understanding of the main differences between models, we include Table 1, which shows the codes that perform the same functionality in CUDA/HIP, SYCL USM model, and SYCL buffer model. Since the CUDA code is practically the same as the HIP code, only changing the word "cuda" to the word "hip", the last one is not included in the table.

4.5. Targeting multiple GPUs and vendors with SYCL

At this point, UVaFTLE has been ported to SYCL and can be executed on NVIDIA and AMD GPUs. However, the application still does not support multi-GPU execution. From now on, we will use the term "*sub-kernel*" to refer to one part of a single kernel distributed across different devices, while the term "*kernel*" will refer to the execution of all the parts of the kernel. The native application uses OpenMP to instance multiple threads, and each thread performs a part of the computational work or sub-kernel using a different GPU device, as explained in

```
using namespace sycl;
int main(){
    // Set the execution queue by selecting a GPU
    queue my_queue(gpu_selector{});
    int elements=100000;
     // Memory allocation (accessible from host and device)
    float* sh_array = malloc_shared<float>(elements, my_queue);
    [...] // Host memory initialization
    // Range declaration
    range array_range{elements};
    range block_range{512};
    // Execute the kernel with a parallel for
    my_queue.parallel_for(nd_range(array_range,block_range),
        [=](nd_item<1> i){
        int gpu_id = i.get_global_id(0);
        if(gpu_id < elements)
            sh_array[gpu_id]=sh_array[gpu_id]*2+1;
    }); // end parallel for
    //wait for the kernel completion to use the data
    my_queue.wait();
    [...] // Use the results of the kernel in the host
    free(sh_array, my_queue);
    return 0;
}
```

Fig. 6. Comparison between CUDA and SYCL USM version with shared memory. The lines with the same background colors share the same purpose in all codes.

Table 1
Memory management in CUDA using device memory, CUDA using shared memory, SYCL USM using device memory (shown as S-USM device), SYCL USM using shared memory (shown as S-USM shared) and SYCL Buffers (shown as S-Buffers device).

Action	Language and model	Function
Allocate device memory	CUDA device	cudaMalloc(dev_array, mem_size)
	CUDA shared	cudaMallocManaged(shared_array, mem_size)
	S-USM device	dev_array = malloc_device<double> (num_elements, my_queue)
	S-USM shared	shared_array = malloc_shared<double> (num_elements, my_queue)
	S-Buffers	buffer buff_array{h_array, range{num_elements)}}
Access to device memory inside the kernel	CUDA device	Declare the array in the kernel prototype and
	CUDA shared	include dev_array/shared_array in the kernel invocation
	S-USM device	Use dev_array in kernel code
	S-USM shared	Use shared_array in kernel code
	S-Buffers	Create an accessor in kernel submit and use it in kernel code accessor acc_array {buf_array, my_handler, read_write}
Copy data between host and device	CUDA device (Sync)	cudaMemcpy(dst_array, src_array, mem_size, cudaMemcpyHostToDevice|cudaMemcpyDevicetoHost)
	CUDA device (Asycn.)	cudaMemcpyAsync(dst_array, src_array, mem_size, cudaMemcpyHostToDevice| cudaMemcpyDevicetoHost, cudaStream)
	CUDA shared	Implicitly done by CUDA runtime when shared_array is used
	S-USM device	my_queue.memcpy(dst_array, src_array, mem_size)
	S-USM shared	Implicitly done by SYCL runtime when shared_array (USM) or acc_array (Buffers) is used in a device kernel
	S-Buffers	
Sync. to ensure the host mem. is updated	CUDA device	cudaDeviceSynchronize() (only if asynchronous copy)
	CUDA share	Implicitly done by CUDA runtime when shared_array is used
	S-USM (both)	my_queue.wait()
	S-Buffers	Implicitly done by SYCL runtime when the scope of dev_buf ends
Free device memory	CUDA (both)	cudaFree(array)
	S-USM (both)	free(array, my_queue)
	S-Buffers	Implicitly done by SYCL runtime when the scope of dev_buf ends

Section 3.3. However, this solution is impossible since SYCL kernels can not be used inside OpenMP target regions [25].

Fortunately, we can do the same job instantiating as many SYCL queues as devices we need and attaching each queue to a different device. Moreover, the queue abstraction allows us to use GPUs from different architectures, such as NVIDIA and AMD. For example, the function shown in Fig. 2 could be easily modified to get a vector of queues with all the AMD GPUs attached to the current node, and Fig. 7 shows a function that returns a queue vector to use all the node's GPUs, regardless of their vendor or architecture. If the program was compiled targeting all the GPUs on the system using an SMCP compiler or with an SSCP compiler (see Section 2.1), the application kernels can be run on any device.

In contrast, targeting multiple GPUs from different vendors using CUDA or HIP requires compiling each native kernel implementation utilizing the specific compiler and developing a host code capable of supporting memory management, data transfers, and kernel launching. The host code is responsible for calling the correct compiled version of the code, depending on the targeted platform. This imposes a significant extra development effort compared to what is necessary with SYCL.

However, to distribute the computation of one kernel across all devices and to run all the sub-kernels concurrently, it is required that there

```
1  std::vector<queue> getAllQueues(){
2      auto devs = device::get_devices(info::device_type::gpu);
3      std::vector<queue> queues(devs.size());
4      for (int d=0; d < devs.size(); d++){
5          queues[d] = queue(devs[d]);
6      }
7      return queues;
8  }
```

Fig. 7. Example of a function for getting a vector of SYCL queues that attaches all the GPUs of the node.

are no data dependencies between sub-kernels; i.e., the range of the output data of each sub-kernel does not overlap any other sub-kernels' range. Using the buffer model, the SYCL runtime will serialize the execution of the sub-kernels after detecting the data dependencies, giving no advantage to using multiple GPUs. For example, let's suppose that the output of our kernel is an array of 1 000 elements, and we have two GPUs to execute the kernel. A non-overlapping data distribution could be the range $[0, 511]$ for the first GPU and $[512, 999]$ for the second, and the sub-kernels can run concurrently. An overlapping distribution of the data could be the range $[0, 511]$ for the first GPU and $[500, 999]$ for the second; in this case, the execution of the sub-kernels would be serialized. Using the USM model and overlapping ranges requires extra development effort to synchronize the data and to ensure the results are correct.

Focusing on the buffer model, the SYCL standard offers two ways to separate the data ranges: Ranged accessors and sub-buffers. A ranged accessor is built from a sub-range of a buffer, limiting the buffer elements that can be accessed. However, the ranged accessor creates a requisite for the entire buffer [26][2]. Therefore, since all the sub-kernels write the same buffer, their execution is serialized, although each sub-kernel writes a non-overlapping range. The sub-buffers are buffers created from a sub-range of a buffer previously created. If the ranges of two sub-buffers created from the same buffer, B_1 and B_2, do not overlap, the accessors created from them, A_1 and A_2, will not overlap. Therefore, if a kernel K_1 uses A_1 and a kernel K_2 uses A_2, both kernels can be concurrently executed. Unfortunately, AdaptiveCpp does not currently support the sub-buffer feature, and oneAPI supports them but also serializes the kernels.

The only solution is to create a buffer array with a separate buffer for each sub-kernel, ensuring their ranges do not overlap. Note the buffers must be explicitly initialized with a brace-enclosed expression or equivalent (aggregate initialization) in the array declaration. In another case, the compilation fails in the array declaration (e.g., using *buffer** followed by a *malloc*; or creating an empty *std::vector* of buffers and later adding the buffers). Moreover, the buffer cannot be created inside a *for* loop. Since each loop iteration creates a new scope, the SYCL runtime will create and destroy the buffer, serializing the kernels instead of concurrently executing them.

Therefore, creating one buffer for each possible sub-kernel is necessary, although the final number of executed sub-kernels is smaller. To illustrate this, Fig. 8 shows how the data is partitioned, assuming that there are three GPUs in the node (therefore creating three buffers) but using only two GPUs afterward. At first, two vectors are created to store the offsets and ranges. The vector size is the maximum number of devices (lines 9 and 10). After that, the values of the vector are initialized. When the device d is used, the offset and range are calculated such that the data among sub-kernels is equally distributed (lines 13–16). If the device d is not used, we must also initialize the offset and range (lines 17–21).

After that, we create an array of three buffers and explicitly initialize it with a brace-enclosed expression using the previously calculated offsets and ranges (lines 26–29)[3]. Although the third device is not used, the third buffer is always created (line 29). If the third buffer is wrongly initialized, the application will be aborted. Correctly initializing the buffers ensures that the application works for a maximum of three devices, independently of the number of used devices. In the example of Fig. 8, the ranges of *dev_buf[0]*, *dev_buf[1]* and *dev_buf[2]* are $[0, 49\,999]$, $[50\,000, 99\,999]$ and $[0, 0]$, respectively. Note that although the ranges of *dev_buf[0]* and *dev_buf[2]* overlap, the two sub-kernels can be concurrently executed since *dev_buf[2]* is never used and does not create data dependencies. Finally, the code starts a *for* loop with *usedDevices* iterations (line 31). At each iteration, the kernel is submitted to the queue *d*; an accessor is created using *dev_buf[d]* (line 34), and a *parallel_for* is launched using a range of *ranges[d]* elements (line 35).

Using the buffers this way allows distributing the computation between several GPUs, but it increases the development effort, as will be seen in Section 6. Note that the example of Fig. 8 only works for a maximum of three GPUs. An array of six buffers will be required in a six-GPU system. This extra development effort is more significant when the number of GPUs or data structures to distribute increases. This does not happen with the native versions, which can run with any number of GPUs without modifications. However, combining NVIDIA and AMD GPUs is easier using SYCL than combining the CUDA and HIP native versions, as explained at the beginning of the section.

In contrast, targeting multiple GPUs using the USM model is easier. We start considering the SYCL USM-device model. We only need to

1. Create the queue, the offset, and the range vectors.
2. Create one array per device of size *ranges[d]* (line 9, Fig. 9).
3. Copy *ranges[d]* items from address *h_array* + *offset[d]* to device array (line 10, Fig. 9).
4. Launch the kernels in queues[d] using a *parallel_for* with *ranges[d]* elements, as in the buffer model case (line 12, Fig. 9).
5. Copy *ranges[d]* elements from device array to address *h_array* + *offset[d]* (line 15, Fig. 9).

However, if our program has two or more kernels, as UVaFTLE, we must synchronize the first kernel finishing with the second kernel starting. In this case, we cannot call *queues[d].wait()* since this would serialize the execution of all kernels. This can be resolved in a simple way using an *in-order* queue (line 3, Fig. 9). This queue serializes all the actions submitted to *queues[d]*, but the actions of two different queues can run in parallel. Once all the kernels are submitted, we wait for the completion of all the submitted actions in all queues (line 18, Fig. 9). Note that *queues[d].wait()* must be called in its own *for* loop. Including these calls inside the main *for* loop would serialize the executions of each queue.

Using the SYCL USM-device model for targeting multiple GPUs is even simpler (Fig. 10). Since all the shared arrays are accessible by all

[2] Note that AdaptiveCpp has an extension that allows to run concurrent kernels [62] using non-overlapping sub-ranges. However, this extension does not comply with the SYCL 2020 specification and can not be used in other compilers, like oneAPI.

[3] To use *std::vector* is also possible simply replacing *buffer<int,1> dev_buf[3] = {...}* by *std::vector<buffer<int,1»dev_buf = {...}*. To use *std::array* is also possible.

```
[...]
int elements = 100000;//host memory declaration
float h_array [elements];
[...]//host memory initialization
//Get all the possible queues. Let's assume three queues
auto queues = getAllQueues();
int numMaxDevices =  queues.size(),  usedDevices = 2;
//Create the offset and range vectors
std::vector<int> offset(numMaxDevices);
std::vector<int> ranges(numMaxDevices);
int chunk = elements / usedDevices;
for(int d=0; d<numMaxDevices; d++){
    if(d < usedDevices){
    //Used buffer, calculate range and offset
        offset[d] = chunk*d;
        ranges[d] = chunk;
    }else{
    /*Ensure the unused buffer can be created. This will not
        affect the  kernels since the buffer is not used*/
        offset[d] = 0;
        ranges[d] = 1;
    }
}
//The last device will compute the padded elements
ranges[usedDevices - 1] += elements % usedDevices;
{//Start a new scope and create the buffers
buffer<int, 1> dev_buf[3]= {
    buffer(h_array + offset[0], range{ranges[0]}),
    buffer(h_array + offset[1], range{ranges[1]}),
    buffer(h_array + offset[2], range{ranges[2]})};
//submit the kernels
for(int d=0; d < usedDevices; d++)
 queues[d].submit([&](handler &my_handler){
//Create the accessor using the appropriate buffer
    accessor array{dev_buff[d], my_handler, read_write};
    my_handler.parallel_for(range{ranges[d]})[=](id<1> i){
        int gpu_id = i.get_global_id(0);
        [...]});
 });
} //end of scope and start the host code
[...]
```

Fig. 8. Distributing kernel work on multiple GPUs using SYCL buffer model.

```
[...]
//In Code of function getAllQueue()
queues[d] =
    queue(devs[d], property_list{property::queue::in_order()});
[...]
//create a vector of pointers (could also use std::vector)
float* d_array[usedDevices];
for(int d=0; d < usedDevices; d++){
  d_array[d] = malloc_device<int>(ranges[d], queues[d]);
  queues[d].memcpy(d_array[d], h_array+offset[d], ranges[d]);
  //Launch first kernel
  queues[d].parallel_for(range{ranges[d]})[=](id<1> i){[...]});
  //Launch the second kernel
  queues[d].parallel_for(range{ranges[d]})[=](id<1> i){[...]});
  queues[d].memcpy(h_array+offset[d], d_array[d], ranges[d]);
}
for(int d=0; d < usedDevices; d++)
  queues[d].wait();
[...]
```

Fig. 9. Distributing kernel submission on multiple GPUs using SYCL USM-device model.

```
1    //Launching a kernel
2    queues[d].parallel_for(range{ranges[d]})[=](id<1> i){
3      int gpu_id = i.get_global_id(0) + offset[d];
4      //kernel code
5      [...]  });
6    [...]
```

Fig. 10. Launching the kernels in multiple GPUs using SYCL USM-shared model.

devices, we only need to: i) create the in-order-queue, the offset, and the range vectors; ii) Launch the kernels in queues[d] using a *parallel_for* with *ranges[d]* elements, and iii) modify the *gpu_id* index calculation adding *offset[d]*.

However, a final consideration should be taken into account. Although SYCL supports simultaneously executing kernels in NVIDIA and AMD GPUs, the GPU drivers do not support transparently performing data transfers between both architectures. This can be solved in two ways: 1) manually transferring data from one device to another through the host, or 2) ensuring that there are no data dependencies between the devices of the different vendors. In our case, the second one is the best option since the data has already been distributed, avoiding data dependencies and thus ensuring the concurrent execution of all the sub-kernels. However, it does not work in all the models since UVaFTLE has several data arrays that the two kernels only read:

- In *buffers model*, the SYCL runtime copies the only-read arrays to each device. Since each GPU only has a disjoint set of items of the read-write arrays, there are no data transfers between different GPU architectures. Therefore, the application properly combines AMD and NVIDIA GPUs without code modifications.
- In *USM-device model*, all the GPUs work using their device memory, and the code does not require modification to run simultaneously in AMD and NVIDIA GPUs.
- In *USM-shared model*, when a *malloc_shared* is performed, SYCL internally calls to *cudaMallocManaged* or *hipMallocManaged*. Since each GPU architecture works with its own memory space, we cannot make data transfers between CUDA and HIP memory spaces. It will require duplicating all the common data arrays (one copy for AMD architecture and another for NVIDIA architecture) and splitting the read-write arrays. This supposes an extra development effort, so the SYCL USM-shared version has not been tested combining AMD and NVIDIA GPUs in Section 5.

4.6. Using multiple GPUs with SYCL: Summary

The steps to enable using multiple GPUs in the SYCL version of the UVaFTLE, assuming that our system has four GPUs, are the following:

1. Get a vector of *queues* to allow using all the GPUs. For both USM models, create the queues as *in-order* queues.
2. Calculate the range and offset of each *sub-kernel* for:
   (a) The output array of the preprocessing kernel (also used as an input in the second kernel).
   (b) The output array of FTLE kernel.
3. (Buffer model) Start a new scope, define and explicitly initialize two arrays of *buffers*, using the ranges and offsets previously calculated: *b_preproc* to manage the output array of the preprocessing kernel, and *b_flte* to manage the output array of the FTLE kernel.
4. Start a *for* loop with one iteration per used device. In iteration *d*:
   (a) Submit the preprocessing kernel, storing the *event ev* generated by the *queue[d]*:
      i. (Buffer model) Create the output *accessor* from *b_preproc[d]*.
      ii. Launch the kernel using an *nd-range parallel for* using the device range.
      iii. (USM-shared model) Change the index calculation by adding the offset of each device.
   (b) Submit the FTLE kernel:
      i. (Buffer model) Create the input accessor from *b_preproc[d]* and the output accessor from *b_ftle[d]*.
      ii. Launch the kernel using an *nd-range parallel for* using the device range.
      iii. (USM-shared model) Change the index calculation by adding the offset of each device.
5. End the *for* loop and:
   (a) (Buffer model) End the scope to update the host memory
   (b) (Both USM models) Wait for the kernel completion using *queue[d].wait()* inside a new *for* loop.

5. Evaluating the effects of porting decisions in terms of performance

5.1. Platform under test

The experiments have been conducted in a computing server property of the *Universidad de Valladolid*, which features two Intel(R) Xeon(R) Platinum 8160 CPU @ 2.10GHz, with 24 Core Processors and 48 physical threads each. The first socket has connected two AMD Vega 10 XT Radeon PRO WX 9100 GPUs with AMD 6.7.0 driver, while the second has two NVIDIA Tesla V100 PCIe 32 GB GPUs with NVIDIA 560.35.03 driver. The server is equipped with a Rocky Linux 9.3 operating system. The toolchains used are GCC 11.4, CUDA 12.4, ROCm 6.1.0, oneAPI 2024.1.0 and LLVM 17.0.6. This LLVM distribution has been used to compile AdaptiveCpp 24.02.0.

Different compilers activate different optimization chains and modules for the same optimization flags. Thus, the resulting codes may perform differently. This is the motivation to test different compilers instead of trying to generalize the results of a single one. The -O3 flag is the typical optimization flag that HPC programmers use by default. According to the documentation of the considered compilers, it activates a selection of optimization modules that at least include the same types of general techniques. Thus, we consider that the results using -O3 as the only optimization flag represent what a regular HPC programmer can expect from the compiler, presenting the fairest comparison scenario for the results obtained with different ones in an HPC environment.

The experiments of Section 5.3.1 to test the HIP-based application using shared memory XNACK have been conducted in LUMI Supercomputer [27]. The computing node is a 64-core AMD EPYC 7A53 “Trento” CPU with four AMD MI250X GPUs. The toolchains used are Cray Programming Environment 8.5.0 and ROCm 6.0.3.

5.2. Test cases

To conduct the performance evaluation, we have chosen two applications widely used in the literature when evaluating flowmap and FTLE computations: The Double–Gyre flow [28] for the 2D case and the Arnold–Beltrami–Childress (ABC) flow or Gromeka–Arnold–Beltrami-Childress (GABC) flow [29] for the 3D case. In particular, our evaluation in the 2D case uses a mesh composed of 10 000 000 points, and in the 3D case, a mesh consisting of 1 000 000 points. Table 2 reflects the details associated with each mesh geometry: The dimensions, the number of mesh points and mesh simplex (either triangles or tetrahedrons), the interval of interest at each axis, and the number of elements in the interval at each axis taken to define the mesh points.

Table 2
Description of the test cases used in our experiments.

	2D	3D
Dim	≈10 000 K (9 998 244)	1 000 K
nFaces	19 983 842	5 821 794
min-max(x, y, z)	(0–2, 0–1, 0-0)	(0–1, 0–1, 0–1)
length(x, y, z)	(3 162, 3 162, 0)	(100, 100, 100)

For each described FTLE test case, we evaluate the performance (in terms of execution time) by exploring six different axes: GPU vendor (NVIDIA, AMD), GPU count (one or two devices), programming model (CUDA, HIP, SYCL), compiler (nvcc, hipcc, clang, AdaptiveCpp, Intel oneAPI), compilation mode for SYCL (SSCP compiler, from now on *Just-in-time*, or *JIT* compiler; and SMCP compiler, from now on *Ahead-Of-time*, or *AOT* compiler), and data management strategy (device, shared or buffers, the last one only for SYCL). Fig. 11 details all the different options for each evaluated axis. From this, thirteen tests are conducted over NVIDIA GPUs and ten over AMD GPUs, using one and two devices with each vendor for each test. Note that we have indicated each configuration's name in gray to help you better understand the later result plots.

When opting for the AOT mode, each kernel of our program is compiled for each of the architectures specified during the build process. All the binary kernels are included in the final executable, and at runtime, the backend selects one kernel or another, depending on the target device. With JIT, contrarily, the compiler generates intermediate code for each of the kernels using LLVM; at runtime, the backend performs the compilation for the target device, regardless of its type. To avoid compiling at every program execution, a kernel cache is stored in the user's directory, so it will only be necessary to compile the kernel the first time the program is run (or if the backend detects that the program has changed).

AdaptiveCpp allows both AOT and JIT modes. Thus, we have tested both. In both cases, the programs can combine kernels executed on CPUs, AMD, and NVIDIA devices. The only restriction is that there should be no memory transfers between cards from different vendors (transfers between CPU and GPUs are not an issue). Thanks to this, we have included in our experiments an evaluation of the performance when using AdaptiveCpp with either AOT or JIT to target NVIDIA and AMD devices simultaneously. Nevertheless, in the case of oneAPI, we only test the AOT mode because JIT is not supported.

Regarding the different vendors, oneAPI includes a plugin for launching SYCL applications on NVIDIA cards. However, when using the recently released plugin for AMD with a profiler and two GPUs, the program crashes. Thus, when using oneAPI, we only experiment with NVIDIA GPU devices, while in any other case, we also test AMD GPUs.

In addition to these configurations, we also explore the multi-GPU performance using AdaptiveCpp and concurrently leveraging NVIDIA and AMD GPUs.

When a test is launched, the application is mapped to the socket connected to the tested GPU. In the experiments using both AMD and NVIDIA GPUs, the application was mapped to the AMD socket. Each test was repeated 30 times, and the results show the average of all of them. Note that when a kernel is executed using two or more GPUs, we take the longest execution time observed for all the sub-kernels; this is the one associated with the slowest sub-kernel execution.

Finally, we want to highlight that the preprocessing kernel takes more time to execute than the FTLE kernel. Thus, the execution time shown for the first kernel is reflected in seconds and milliseconds for the second.

In the following sections, we analyze in detail these results concerning each of the evaluated axes.

5.3. *Performance evaluation*

This section presents the performance evaluation of the tested implementations using different compilers and execution modes, incorporating the updated results shown in Figs. 12, 13, and 15. In Fig. 12, we illustrate the performance evaluation results of each kernel when targeting NVIDIA GPU devices for the 2D and 3D FTLE test cases. The same is done in Fig. 13 for the AMD GPU devices. The analysis focuses on execution times across various configurations and highlights the impact of different factors on performance.

5.3.1. *Impact of data management strategy*

A comparison between the three SYCL data management strategies shows that the USM-shared model obtains the worst performance. In NVIDIA GPUs, when the preprocessing kernel runs in only one device, there are no significant differences between the three memory models. However, splitting this kernel into two NVIDIA GPUs with USM-shared leads to performance degradation. The performance of the second kernel (FTLE) is severely degraded when using USM-shared with both one and two GPUs. The execution time of the 2D application is degraded to the point that it does not scale, and the time using two GPUs is higher than using one. The 3D application slightly scales with 2 GPUs, as the second kernel that is badly affected by the use of USM-shared has a much lower load than the preprocessing kernel. However, the buffer and USM-device versions continue to achieve better results.

In the AMD GPUs, the USM-shared code obtains systematically worse performance. This performance degradation is also observed using the native compilers (nvcc/hipcc) and clang. Note that the AMD Vega 10 XT GPUs do not support XNACK. XNACK allows AMD GPUs to migrate memory pages between the CPU and the GPU when a page fault occurs, improving the application performance when shared (or, in HIP terminology, managed) memory is used. To test the effects of shared memory in more modern architectures and the XNACK feature, we conduct a test using two AMD Instinct MI250X GPUs on LUMI supercomputer [27]. We run HIP-based FTLE applications considering three scenarios: i) using device memory, ii) using managed memory without XNACK, and iii) using managed memory activating XNACK. Fig. 14 shows the results. We observe that using managed memory severely degrades the application's performance. When XNACK is activated, the preprocessing kernel obtains the same performance as device memory if the kernel is executed on one GPU. However, the performance degradation on two GPUs is even more significant when XNACK is activated. Moreover, the second kernel always worsens its execution time, no matter the number of GPUs. Then, the problem of managed memory persists if the application has two or more kernels or is executed on several GPUs, even when XNACK is active.

Our experimentation shows that the USM-shared model systematically leads to performance degradation, regardless of the platform or programming language used. The second kernel (FTLE) always suffers from reduced performance, while the first kernel also experiences slowdowns when split between two GPUs. Using a fine-grain memory control could improve performance, at the cost of increasing the development effort.

Comparing the USM-device and buffer-based implementations, when AdaptiveCpp is used, the buffer-based management introduces a small overhead (1.5 %–3 %). This degradation is more noticeable in the 3D application on AMD GPUs. However, when the application is compiled with oneAPI, the buffer-based management obtains the best results. Although the improvement is slight in the 2D application (only 1 %), the 3D application speeds up by 20 %.

Given these findings, we recommend carefully evaluating USM-device and buffer-based strategies depending on the specific hardware and workload characteristics. Due to the current state of the shared memory implementation, it would be a good choice if our application has a single kernel and runs on a single GPU. Other cases would require tuning the memory access to improve performance by adding new code with hints declared in the language of the specific backend. This kind of tuning compromises portability. Thus, performance portability is simpler using the device memory model, which is the most common approach to implement GPU applications.

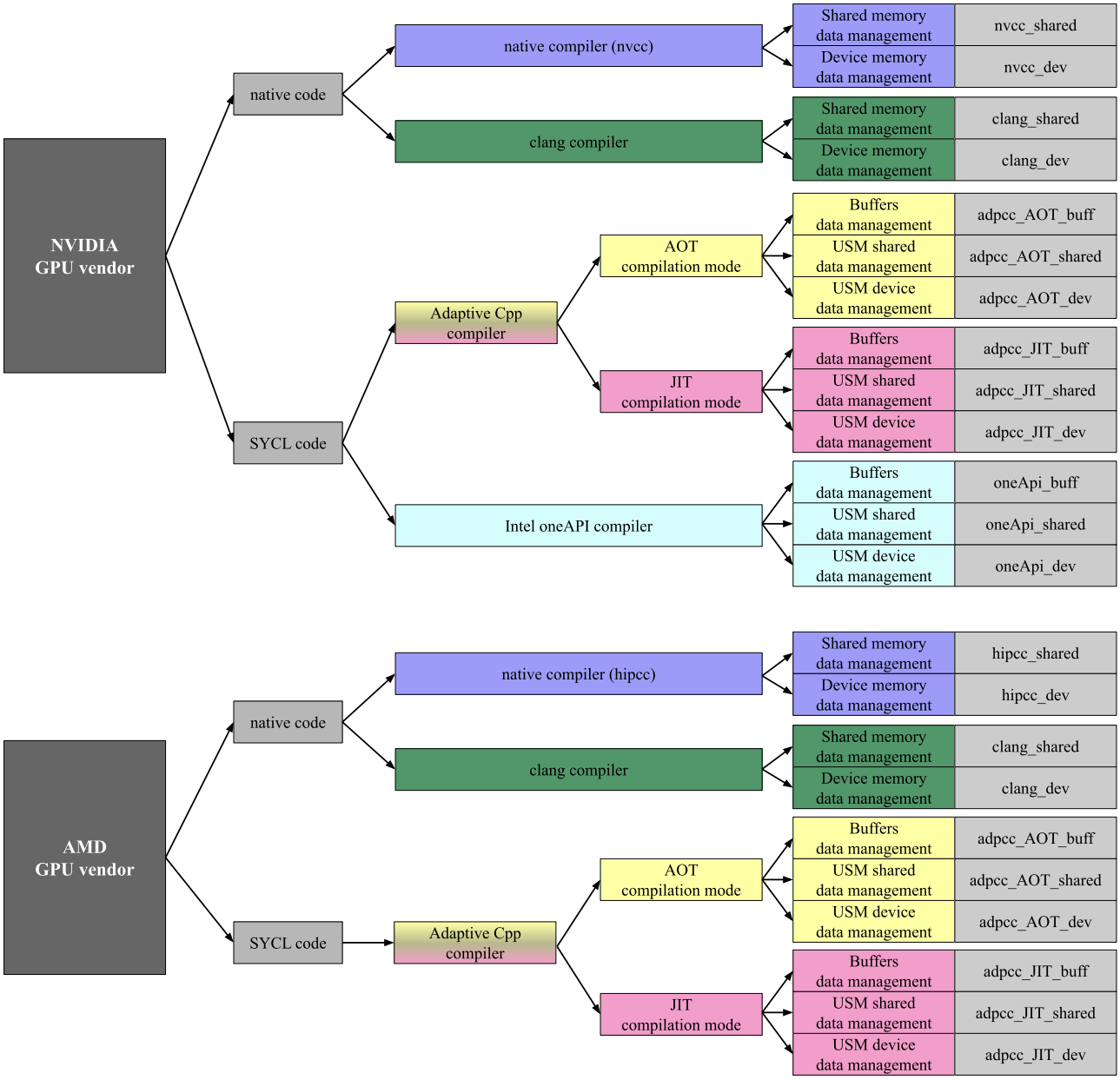


Fig. 11. Scheme of the different configurations tested for the performance evaluation. The gray boxes reflect each configuration's name used in the plots; the colors also correspond to those used in the plots.

From now on, the performance discussion in subsequent sections is centered on buffer and USM-device implementations.

5.3.2. *Effects of compiler choice*

Analyzing first the NVIDIA devices results (Fig. 12), we see that the workload against imposes non-negligible differences:

- With the preprocessing kernel, which has a greater workload, nvcc is always the best option. AdaptiveCpp and clang offer very similar results, and oneAPI is either similar to them or close to nvcc (in the 3D case).
- When the load is smaller, like in the FTLE kernel, surprisingly, the compilers used for SYCL outperform both nvcc (which offers the worst results) and clang. In particular, oneAPI combined with the buffer implementation is the best one.

Compiler differences are less pronounced for AMD GPUs. The preprocessing kernel delivers similar performance across all tested compilers, while AdaptiveCpp combined with the USM-device model slightly surpasses HIPCC for the 2D FTLE kernel and the 3D application. These results suggest that compiler selection significantly influences NVIDIA GPUs more than AMD GPUs. Therefore, the native compiler nvcc is preferable for high-workload kernels for NVIDIA GPUs. In contrast, SYCL implementations, particularly those compiled with AdaptiveCpp and oneAPI, provide competitive or superior performance in lower workload scenarios.

5.3.3. *Comparison between SYCL and native programming models*

A key objective of this study is to assess SYCL's performance relative to native programming models such as CUDA and HIP. For AMD GPUs, SYCL achieves performance levels comparable to native implementations across all cases. For NVIDIA GPUs, the performance comparison depends on workload complexity. High-workload kernels, such as preprocessing, reveal a significant performance gap, with nvcc maintaining a clear advantage over SYCL. However, this gap narrows in 3D

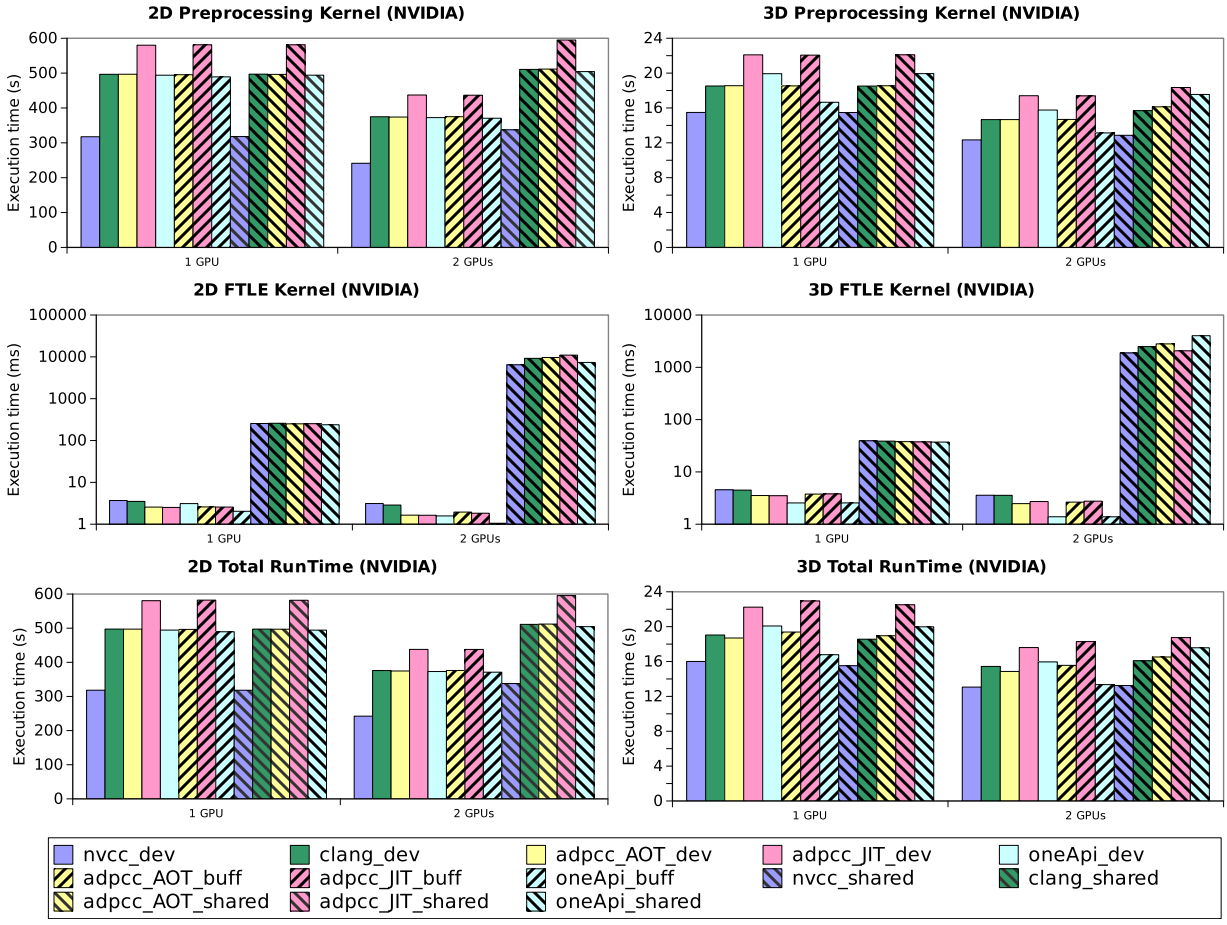


Fig. 12. Performance evaluation results of each kernel when targeting NVIDIA GPU devices for the 2D and 3D FTLE test cases.

test cases, where oneAPI compiled SYCL implementations nearly match nvcc. Notably, for low-workload kernels like FTLE, SYCL implementations actually outperform nvcc. These findings suggest that SYCL, when compiled with AdaptiveCpp or oneAPI, can serve as a competitive alternative to native programming models, offering strong performance along with portability advantages.

5.3.4. *Effects of SYCL compilation mode for AdaptiveCpp: AOT vs. JIT*

Finally, we discuss whether it is better to opt for AOT or JIT compilation modes when using AdaptiveCpp to compile SYCL codes. Although the general agreement is that JIT compiling can produce better optimizations, our results, and those presented by compiler designers (such as [18]), show that there are applications and situations where they lead to the opposite effect. With our chosen scenarios, there is only one case where the performance using AOT or JIT significantly differs: Using NVIDIA devices and running kernels with considerable workload (see Fig. 12, preprocessing kernel). In that case, AOT offers better results than JIT. In contrast, on AMD GPUs (see Fig. 13), JIT-compiled kernels run slightly better.

Consequently, the general recommendation would be to use AOT for NVIDIA GPUs and JIT for AMD GPUs. However, consider that the JIT compiler also achieves good performance, and it would be helpful when we want a completely portable application or when we do not know the architecture of the target device.

5.3.5. *Multi-GPU and multi-vendor performance*

Multi-GPU and multi-vendor performance evaluations provide further insights. Figs. 12 and 13 illustrate scalability trends when running test cases on up to two GPUs of the same vendor, demonstrating limited scalability due to the moderate workload of the kernels. Extending this evaluation, Fig. 15 presents results for SYCL compiled with AdaptiveCpp running on multiple GPUs from different vendors (NVIDIA and AMD).

The key takeaways from this analysis indicate that multi-vendor execution is feasible, showcasing SYCL's strength in heterogeneous environments. Unlike CUDA or HIP, which require vendor-specific frameworks and binaries, SYCL enables a unified codebase that effectively utilizes all available GPUs. The preprocessing kernel exhibits scalable performance, with improvements observed as the number of GPUs increases from two to four. However, scalability in the FTLE kernel stagnates due to its relatively low workload.

Additionally, using the four cards, AOT slightly outperforms JIT in most cases. Using only NVIDIA GPUs, AOT works better. On AMD GPUs, JIT works better. Nevertheless, the absolute gains of NVIDIA GPUs in the experimental platform are higher than the gains on the AMD GPUs. Thus, the overall result with the four GPUS is better using AOT.

Overall, these results emphasize SYCL's potential in multi-vendor, multi-GPU execution scenarios. Its ability to bridge performance gaps while maintaining portability makes it a promising alternative to proprietary solutions, particularly in heterogeneous computing environments.

6. Development effort

This section analyzes the differences in development efforts between CUDA, HIP, and SYCL codes of UVaFTLE. We consider four classical development effort metrics: The number of lines of code (LOC), the number of code tokens (TOK), McCabe's cyclomatic complexity (CCN) [30], and Halstead's development effort [31]. The first two metrics measure the code volume that the user should program. The third measures the rational effort required to program it, including code

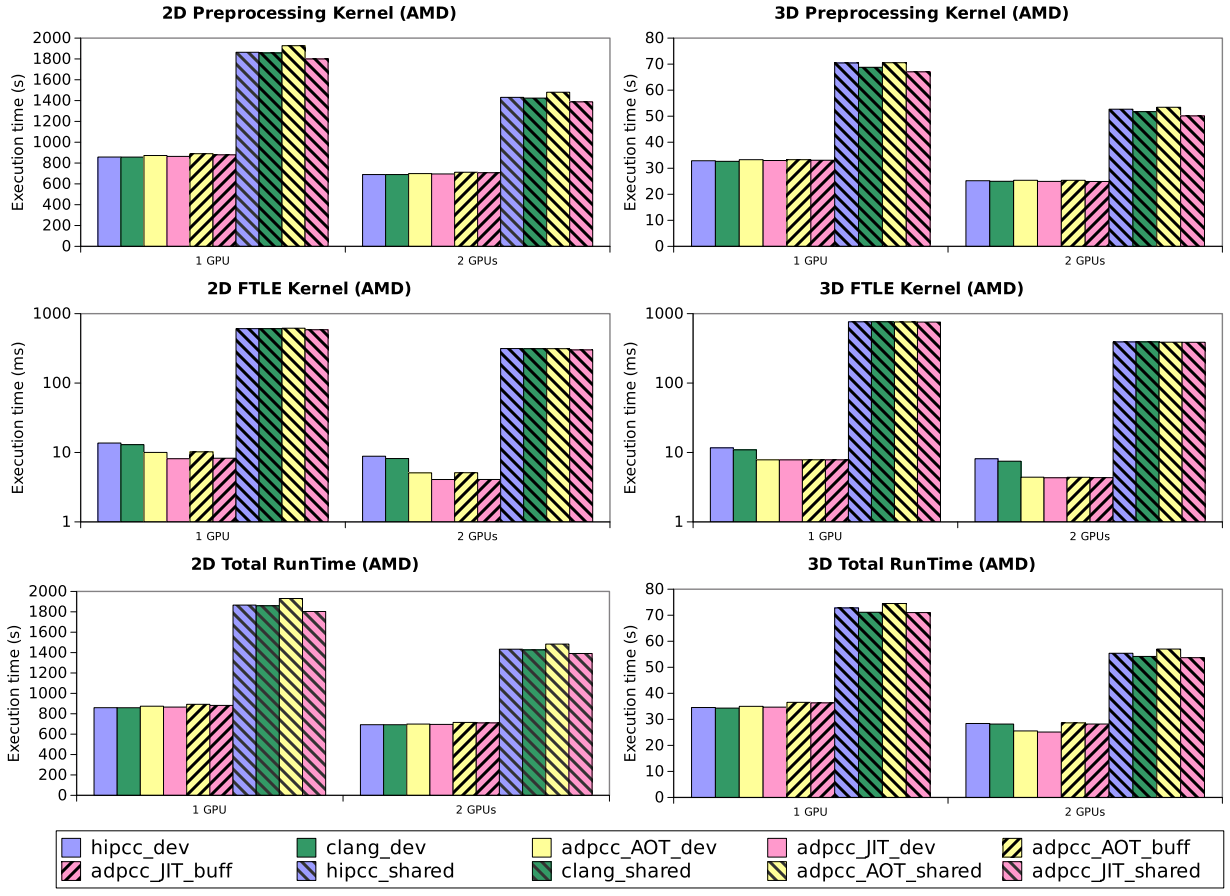


Fig. 13. Performance evaluation results of each kernel when targeting AMD GPU devices for the 2D and 3D FTLE test cases.

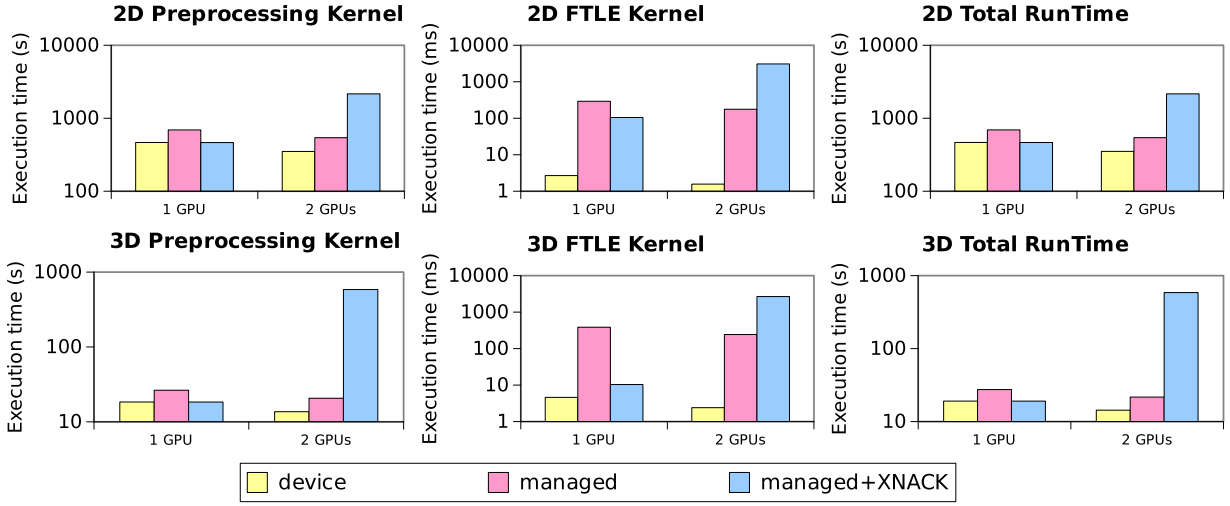


Fig. 14. Performance results targeting AMD Instinct MI250X on LUMI supercomputer using the HIP-based FTLE application using device memory, managed memory, and managed memory activating XNACK.

divergences and potential issues that should be considered when developing, testing, and debugging the program. The last metric measures code complexity and volume indicators, obtaining a comprehensive measure of the development effort.

The measured codes include the management of data structures, kernel definitions, and coordination host codes. For a fair comparison, each version is written in a single source code file and formatted according to the same criteria. The differences between codes are strictly necessary and are associated with the particularities of each programming model. For example, comparing the FTLE kernels in CUDA and SYCL, the main differences are how the thread global index is calculated, as explained in Section 4, and certain calls to perform mathematical operations, such as square root or cosine. The CUDA and HIP versions of the program support multiple GPUs of the corresponding vendor and the SYCL USM-based implementations. However, as we explain in Section 4.5, by enabling multi-GPU execution when the buffer model is used, the final SYCL code changes in volume depending on the maximum number of GPUs allowed. For this reason, we have compared four

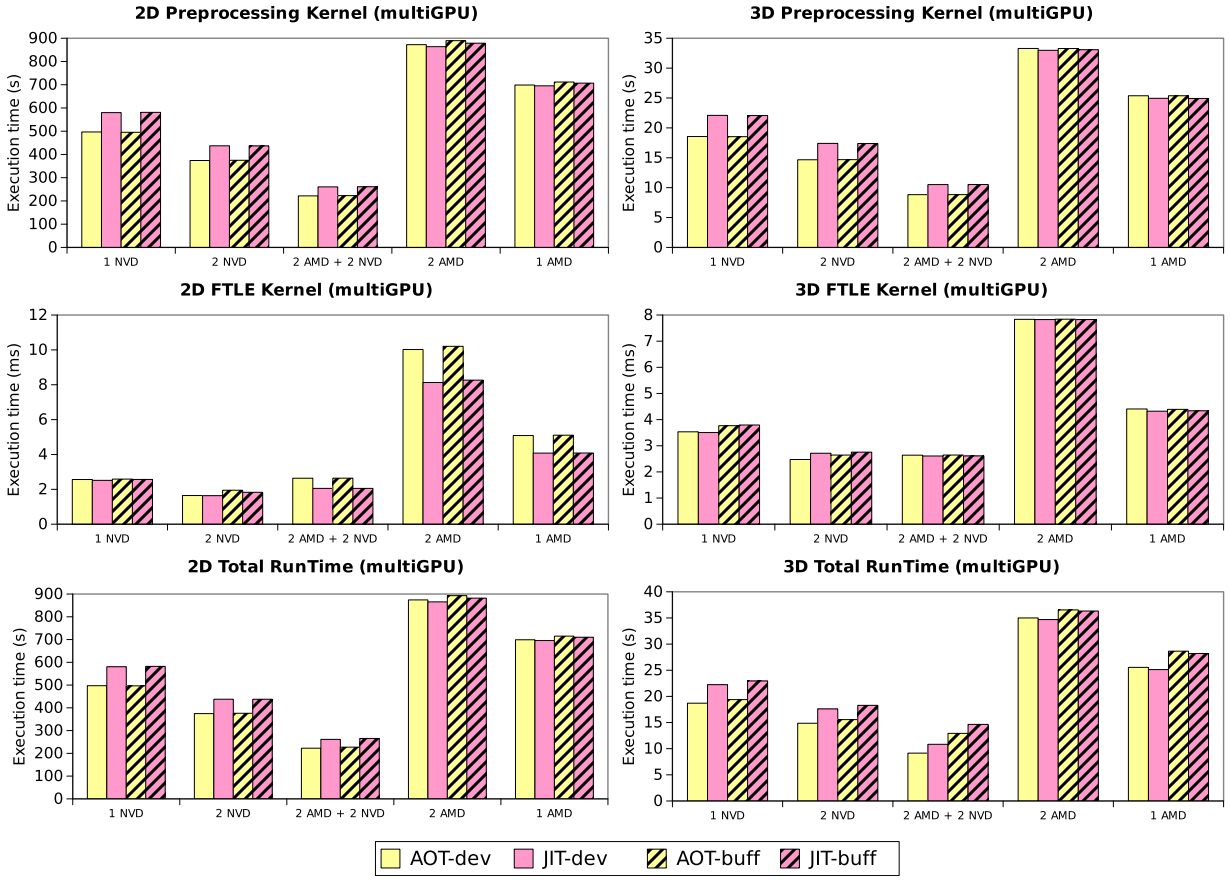


Fig. 15. AdaptiveCpp performance evaluation results of each kernel targeting AMD and NVIDIA GPUs simultaneously for the 2D and 3D FTLE test cases.

versions of the SYCL buffer code, allowing a maximum of 1, 2, 4, and 8 GPUs, respectively. The cleaned versions of both the SYCL programs and the CUDA and HIP versions can be found in our repository, in the folder *measure-codes* [13].

Table 3 reflects the measures of the four development-effort metrics for each one of the functions that present changes that depend on the programming model chosen. They include the three critical functions that have been transformed into kernels (preprocessing, and the 2D and 3D FTLE functions), the main function, which contains the memory management and kernel calls, and the whole program, including the previous code and other auxiliary functions and declarations independent of the heterogeneous programming model selected.

The metrics reveal that the development effort of CUDA and HIP versions regarding the kernels are almost the same, because their implementations are identical. Considering the SYCL version of the kernels, the values measured for the four metrics are higher than the CUDA/HIP results. Nevertheless, the CCN results present almost the same values as those observed for the native versions. These LOC and TOK higher values are mainly due to the *submit* lambda function, the creation of the *accessors* (in SYCL buffer code), and the *nd-range parallel for* lambda function (in all SYCL codes). The preprocessing kernel is the most affected one by this increase, as it is the smallest kernel, being its code lines increased by 31 % (with buffers) and 5 % (with USM), and its number of tokens by 65 % (with buffers) and 30 % (with USM). Halstead's development effort is around twice as much in SYCL than in the other two versions. This difference is less significant in the other two kernels: 7 % (buffers) and 1 % (USM) more lines, 17 % (buffers) and 6 % (USM) more tokens, and 17 % (buffers) and 15 % (USM) more Halstead's development effort for the 2D kernel; and 5 % (buffers) and 1 % (USM) more lines, 11 % (buffers) and 5 % (USM) more tokens, and 16 % (buffers) and 15 % (USM) more Halstead's development effort for the 3D kernel. These measures indicate that the increase in development effort is more significant with small kernels than with large kernels due to the minimum programming structures, declarations, and initialization needed in a SYCL kernel.

When analyzing the main function of the code, we see that the differences between CUDA and HIP are minor. In the particular case of CUDA/HIP devices, LOC is the same, while TOK is slightly smaller for HIP (0.5 % less) and Halstead (5 % less). The same applies to CUDA/HIP Shared. When comparing Shared against Device in the main function, Shared presents better results with around 13 % less LOC, 21 % less TOK, 17 % less CNN, and 26 % less Halstead values. All these differences in the main function are consequently also observed in the whole code for CUDA and HIP.

SYCL generally offers worse results than CUDA/HIP, with around 40 % higher CNN values. Taking as a reference the best CUDA/HIP case (Shared), when using SYCL with buffers, the LOC starts being 1 % higher for 1 GPU and reaches up to 14 % higher when using 8 GPUs; the TOK is 25 % higher with 1 GPU and reaches 57 % higher with 8 GPUs; and the Halstead metrics start at 34 % higher with 1 GPU, reaching more than double of the corresponding value for CUDA/HIP when using 8 GPUs. Note that, in modern systems with a vast number of GPUs, the development effort with SYCL using buffers would reach an impassable level of development (see the explanation in Section 4.5). Considering the SYCL USM versions, the Shared one requires less effort than the Device-based one. Compared with CUDA/HIP, the differences between those observed when using buffers are significantly reduced. Concretely, with SYCL USM Shared, the LOC is 2 % smaller, the TOK is 12 % higher with SYCL, and the Halstead metrics are 22 % higher.

Analyzing the whole code globally, it can be seen that the SYCL code has greater development effort metrics than native versions, even in a single GPU version, especially for the TOK and Halstead metrics. The

Table 3
Development effort metrics according to the programming model employed.

Function/Kernel	Code version	LOC	TOK	CCN	Halstead
Preprocessing	CUDA (Device and Shared)	19	190	8	23 908
	HIP (Device and Shared)	19	190	8	23 908
	SYCL Buffers	25	314	9	52 657
	SYCL USM (Device and Shared)	20	248	9	41 474
FTLE 2D	CUDA (Device and Shared)	134	1090	26	508 649
	HIP (Device and Shared)	134	1090	26	508 649
	SYCL Buffers	144	1273	27	596 583
	SYCL USM (Device and Shared)	136	1159	27	585 189
FTLE 3D	CUDA (Device and Shared)	194	1785	40	918 499
	HIP (Device and Shared)	194	1785	40	918 499
	SYCL Buffers	204	1982	41	1 070 756
	SYCL USM (Device and Shared)	196	1868	41	1 061 277
main	CUDA Device	178	1557	17	603 399
	CUDA Shared	154	1225	14	442 444
	HIP Device	178	1548	17	571 032
	HIP Shared	154	1216	14	418 225
	SYCL Buf. (1 GPU)	157	1511	20	594 992
	SYCL Buf. (2 GPUs)	163	1594	20	650 984
	SYCL Buf. (4 GPUs)	167	1694	20	718 834
	SYCL Buf. (8 GPUs)	175	1926	20	897 778
	SYCL USM Device	176	1782	20	787 955
	SYCL USM Shared	151	1372	20	542 507
Whole code	CUDA Device	625	5201	110	5 276 951
	CUDA Shared	601	4870	107	4 940 742
	HIP Device	625	5193	110	5 231 299
	HIP Shared	601	4861	107	4 895 645
	SYCL Buf. (1 GPU)	630	5660	116	6 716 551
	SYCL Buf. (2 GPUs)	636	5743	116	6 890 008
	SYCL Buf. (4 GPUs)	640	5843	116	7 093 358
	SYCL Buf. (8 GPUs)	648	6075	116	7 600 510
	SYCL USM Device	628	5637	116	6 763 383
	SYCL USM Shared	603	5227	116	6 123 110

only exception is the LOC value when using USM Shared with SYCL, which is slightly smaller than that for either CUDA or HIP.

In summary, in SYCL, the transparent management of buffers and memory movements for a single device and queue is more straightforward and comparable to orchestrating the equivalent asynchronous operations in CUDA or HIP. However, the elaborated syntax and declarations needed for kernels increase their complexity, especially for simple or small kernels. Moreover, in the SYCL host code with buffers, managing each extra device introduces more complexity. In contrast, managing an arbitrary number of devices can be easily abstracted in the CUDA and HIP versions. However, this SYCL problem could be solved in the future if the compilers include full support for sub-buffers (see Section 4.5). Considering the SYCL case when using USM, the development effort is notably smaller in general than that seen with buffers. However, it is still slightly greater than the equivalent measures for CUDA or HIP. Finally, using shared memory always obtains the lowest development effort metrics. However, the performance problems detected in current USM memory implementations (see Section 5) may discourage its use except for applications with only one kernel executed in one GPU, or if memory access is manually tuned.

7. Key findings and insights

Based on the experiments conducted and the analysis carried out, the main findings that can be extracted from this work are as follows:

- In general, the performance results reveal that, when using buffers or USM with device memory, there is not a remarkable overhead associated with SYCL usage in terms of the GPU kernel execution times compared to using kernel native implementations based on CUDA or HIP. This result is consistent when compiling it with either AdaptiveCpp or oneAPI. The only exception is when comparing the preprocessing kernel in the CUDA-based version compiled with nvcc against the one compiled with clang or the equivalent SYCL version. The first one is faster. When using USM and shared memory in SYCL (or managed memory in CUDA/HIP), the performance notably worsens. It would be advisable to use shared/managed memory: (1) To obtain an initial version of the application due to its lower development effort; (2) When the application has only one kernel and does not distribute the computation in two or more GPUs, or (3) When the programmer is eager to take the effort to manually optimize the application memory accesses by adding code with hints declared in the language of the specific backend, diminishing portability.
- We have evaluated two very different kernels in their nature: As explained in previous sections, the preprocessing kernel is much more memory-intensive than the FTLE one, which focuses on solving a collection of linear algebra operations and is much faster to complete. Although the kernels' typology is very different the scalability observed with the native versions and SYCL is equivalent.
- Regarding the multi-GPU programs with SYCL and AdaptiveCpp for four GPU devices, two NVIDIA, and two AMD, the first observation is that the code can indeed simultaneously leverage all of them because the application tested does not need communications across devices that require different backends. The performance results reflect that using the four GPU devices improves the results for the preprocessing kernel. However, this is not true for the FTLE kernel because the load is distributed by balanced blocks, not taking into account the different computational power of each device. Thus, the final load is unbalanced due to the lower computational power of the AMD GPUs in our experimental platform compared to the NVIDIA ones.
- The development effort measures indicate that using CUDA/HIP is slightly easier than programming in SYCL (both USM and buffer memory management models). The least complexity values regarding memory movements are observed when opting for shared memory, regardless of the programming model. The basic kernel syntax and the inner declarations needed with SYCL are slightly higher than for CUDA or HIP. Using buffers with SYCL increases the complexity of the kernels.

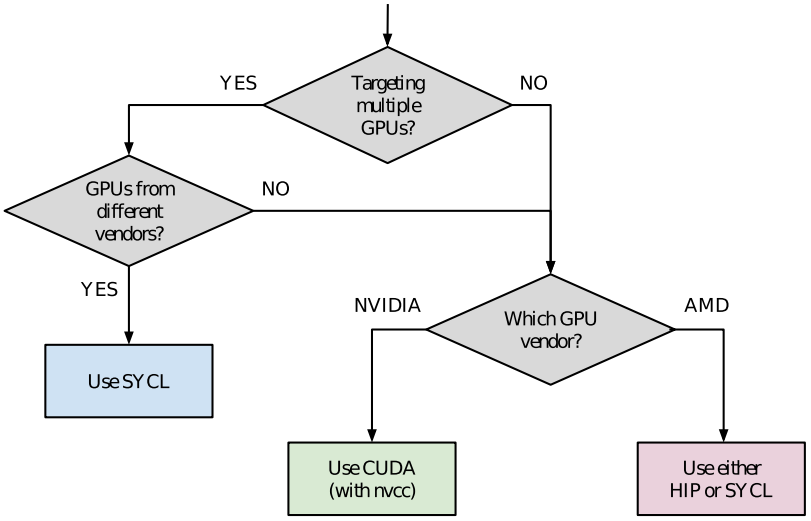


Fig. 16. Decision tree to assist in selecting the programming model, according to the number of GPUs to use and their vendor.

- With the current development status of the SYCL compilers, the development effort metrics reveal that the management of each extra device introduces more code complexity when opting for buffers. In contrast, managing an arbitrary number of devices can be easily abstracted in the CUDA and HIP versions. Using SYCL with USM is slightly more costly than opting for CUDA or HIP. Still, the complexity is constant, regardless of the number of devices targeted, contrary to the equivalent case when using buffers, where increasing the number of GPUs implies also increasing the complexity.
- Although the development effort is generally higher, the SYCL programs are more portable. They can run the application and distribute the computation in both NVIDIA and AMD GPUs, even combining the GPUs of the two vendors in the same execution. With vendor-provided models, this could only be done by combining them in a much more complicated code that should include the solutions in both models, adding some data communication across them.

8. Putting all together: A general development strategy

This section provides general insights to help choose the best programming model, compiler, and data management model. Although our findings have been obtained with a particular application there are general conclusions that can help guide the early stages of development of other parallel applications.

First, if the primary objective is to ensure code portability, it is evident that SYCL is a good choice. Of the different options considered in this study, it is the only one that allows the use of GPUs of different vendors, as far as there are no data communications across devices. Nevertheless, several other aspects must also be considered if the goal is to maximize performance. The first thing to consider is the number of GPUs and vendors to use. In Fig. 16, we illustrate schematically the decisions to select the most appropriate programming model (CUDA, HIP, or SYCL). If GPUs from different vendors are employed, SYCL is the best option; otherwise, if NVIDIA devices are used, CUDA offers the best performance, and for AMD devices, both SYCL and HIP are equally good options.

After choosing the programming model, the next step is to choose the most suitable compiler. Fig. 17 provides a schematic representation of the key considerations for this decision. In summary, to use CUDA, the best compiler option is nvcc; to use HIP, the recommended compiler is hipcc; with SYCL, for NVIDIA GPUs, oneAPI is the preferred choice, while for AMD GPUs, we would recommend using AdaptiveCpp.

Finally, it is also important to choose the data management strategy correctly. When nvcc or hipcc compilers are used, the best option

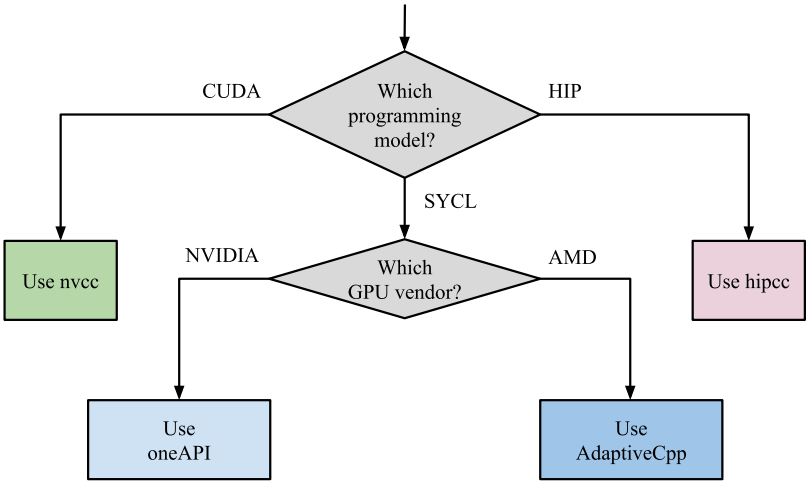


Fig. 17. Decision tree to assist in selecting the compiler, according to the programming model chosen.

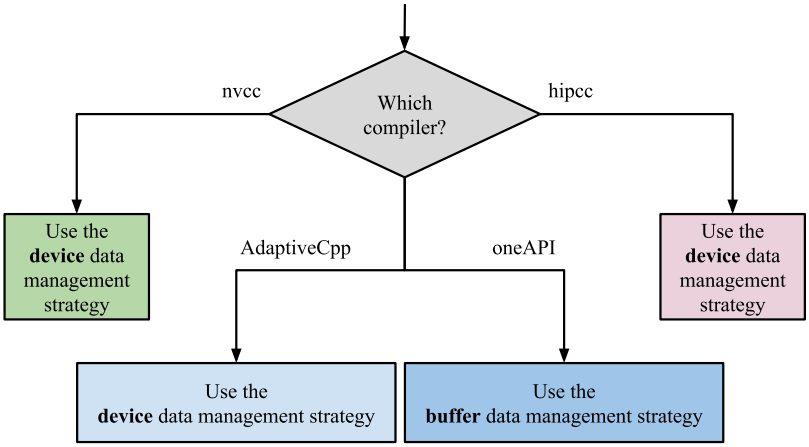


Fig. 18. Decision tree to assist in selecting the data management according to the programming model and compiler chosen.

in terms of performance is the USM-device data management strategy. With SYCL, if the AdaptiveCpp compiler is employed, also the USM-device data management strategy is the best option. With oneAPI, the buffer data management strategy is better. Fig. 18 provides a schema of these considerations.

9. Related work

In this section, we briefly describe the landscape of contributions that study the functional and performance portability of SYCL and its associated problems, as well as the works that focus on FTLE computation and their limitations.

9.1. SYCL Portability

Due to the growing interest in heterogeneous computing and SYCL, several works have used this standard and studied its portability. Some of these works are focused on code migration to SYCL from other languages like CUDA [32–34], OpenCL [35,36], or OpenMP [37], comparing the performance of both versions. Other papers present SYCL libraries to speed up and make portable other scientific works, such as machine learning [38], or neural network [39] algorithms, or present SYCL hand-tuned versions of a specific algorithm, comparing them with the state-of-the-art algorithms [40]. Other works extensively study the performance portability of SYCL across different device types for specific classes of applications (see e.g. [41]). There are also extensions to SYCL that explore the portability and efficiency of applications across multiple accelerators of different types using their own execution model, such as CHARM-SYCL [42].

Other works are focused on the performance evaluation of SYCL compilers. In [43], the authors made a comparative study of OpenCL, OpenMP, and TriSYCL in multiprocessors. However, TriSYCL currently does not support GPUs. In [44], a comparison using several benchmarks and the Intel LLVM-SYCL compiler against CUDA using Tesla V-100 is presented. However, AMD architecture is not studied. Other works compare several SYCL compilers [45–47] against multiple AMD and NVIDIA GPUs models.

Focusing on the SYCL memory management models, some works report that using the USM memory model with device allocation in discrete devices does not lead to performance penalties [48]. Other works point out that for other applications, using USM with shared (or managed) allocation implies significant performance degradation due to page fault handling and PCIe latencies [49]. The study of the performance portability of the SYCL memory management models depends on the communication structure of the application and the classes of different devices involved.

To the best of our knowledge, none of the existing works explore the possibilities offered by SYCL of using multiple GPUs of different vendors simultaneously (such as NVIDIA and AMD) while analyzing the development effort implications of coding in SYCL for several devices with varying models of memory allocation.

9.2. FTLE Computation

In the literature, previous works offer optimizations in the context of the FTLE computation. Some [50–54] focus on speeding up the calculations of the FTLE by applying some optimization techniques such as reducing I/O, optimizing the use of the memory hierarchy, or using multiple CPUs. Other authors [55–60] focus on exploiting GPU devices to accelerate FTLE computation. Another study proposes using an Accelerated Processing Unit (APU) to speed up the computation of FTLEs [61].

As we described in our previous work [12], the main problems of the existing proposals that leverage GPU devices to compute the FTLE are that most of them are old and based on outdated tools incapable of tackling modern devices. Besides, in general, a multi-GPU scheme is not supported. Moreover, neither an in-depth description of the GPU implementation nor the source code are provided. For these reasons, our previous work offered a competitive, open-source implementation of the FTLE computation (named UVaFTLE) equipped with a CUDA kernel capable of simultaneously using multiple NVIDIA GPU devices.

To the best of our knowledge, in the existing literature, there is a lack of updated proposals for FTLE computation that tackle heterogeneous environments provided with GPU devices from different vendors. To fill this gap, in this work, we re-design UVaFTLE to use SYCL in such a way that it can leverage any GPU device, regardless of its vendor. For completeness, we also present a novel UVaFTLE implementation that uses HIP instead of CUDA to tackle AMD GPU devices. Moreover, we evaluate the SYCL performance compared to the implementations based on HIP or CUDA.

10. Concluding remarks

Ensuring performance portability across heterogeneous GPU architectures remains a crucial challenge in high-performance computing. In this work, we show that SYCL can provide a viable alternative to vendor-specific programming models, enabling multi-GPU execution across different architectures with competitive performance. Our evaluation of the FTLE application shows that SYCL, particularly with USM-device memory, achieves performance close to CUDA and HIP while significantly improving portability. However, the shared (SYCL) or managed (CUDA/HIP) memory management model introduces performance penalties, particularly in multi-GPU setups.

The results highlight that while SYCL increases development effort compared to native programming models, it offers an important advantage: a unified programming approach across different GPU vendors. This makes it an attractive option for developers who prioritize portability without sacrificing significant performance.

Given these findings, future work should explore improved load-balancing techniques for SYCL multi-GPU applications, study the unsolved problem of communications across different backends in SYCL, and extend evaluations to additional architectures such as FPGAs. We also encourage researchers and developers to contribute to the ongoing refinement of SYCL implementations to further enhance performance portability. As SYCL continues to mature, it has the potential to become a standard tool for high-performance heterogeneous computing.

CRediT authorship contribution statement

Francisco J. Andújar: Software, Validation, Investigation, Formal analysis, Data curation, Conceptualization, Writing - review & editing, Resources, Visualization, Methodology, Writing - original draft; **Rocío Carratalá-Sáez:** Software, Investigation, Formal analysis, Data curation, Conceptualization, Writing - review & editing, Resources, Visualization, Methodology, Writing - original draft; **Yuri Torres:** Visualization, Formal analysis, Methodology, Data curation, Writing - original draft, Conceptualization, Validation, Writing - review & editing, Software, Investigation; **Arturo Gonzalez-Escribano:** Supervision, Investigation, Funding acquisition, Project administration, Conceptualization, Writing - review & editing, Resources, Validation, Methodology, Writing - original draft; **Diego R. Llanos:** Writing - review & editing, Supervision, Project administration, Conceptualization, Methodology, Writing - original draft, Funding acquisition, Visualization, Resources.

Data availability

All the codes and data are available in a GitHub repository included in our manuscript.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Francisco J. Andujar reports financial support was provided by Spain Ministry of Science and Innovation. Rocio Carratala-Saez reports financial support was provided by Spain Ministry of Science and Innovation. Yuri Torres reports financial support was provided by Spain Ministry of Science and Innovation. Arturo Gonzalez-Escribano reports financial support was provided by Spain Ministry of Science and Innovation. Diego R. Llanos reports financial support was provided by Spain Ministry of Science and Innovation. Francisco J. Andujar reports financial support was provided by Government of Castile and León. Rocio Carratala-Saez reports financial support was provided by Government of Castile and León. Yuri Torres reports financial support was provided by Government of Castile and León. Arturo Gonzalez-Escribano reports financial support was provided by Government of Castile and León. Diego R. Llanos reports financial support was provided by Government of Castile and León. Diego R. Llanos reports financial support was provided by Spain Ministry of Science and Innovation. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported in part by the Spanish Ministerio de Ciencia e Innovación and by the European Regional Development Fund's "A Way of Making Europe" (NATASHA project, Grant PID2022142292NB-I00, funded by MCIN/AEI/10.13059/501100011033), by Junta de Castilla y León FEDER Grant VA226P20 (PROPHET-2 Project), EuroHPC Joint Undertaking for awarding us access to LUMI at CSC, Finland (project EHPC-DEV-2024D07-079), and by Grant TED2021-130367B-I00, funded by MCIN/AEI/10.13039/501100011033, and by Next Generation EU – Plan de Recuperación, Transformación y Resiliencia.

References

[1] NVIDIA, CUDA Toolkit Documentation v12.5, 2024, http://docs.nvidia.com/cuda/, (accessed July 9, 2024).
[2] AMD, AMD ROCm Documentation, 2024, https://rocm.docs.amd.com/en/latest/ (accessed July 9, 2024).
[3] The Khronos Group Inc, Open Computing Language (OpenCL) Overview, 2020, http://www.khronos.org/opencl/ (accessed July 9, 2024).
[4] Khronos OpenCL working group, et al., SYCL 2020 Specification (revision 8), 2023, https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf (accessed June 26, 2024).
[5] OpenMP Consortium, OpenMP Specifications, 2021, https://www.openmp.org/specifications/, (accessed July 9, 2024).
[6] H.C. Edwards, C.R. Trott, Kokkos: enabling performance portability across manycore architectures, in: 2013 Extreme Scaling Workshop (xsw 2013), 2013, pp. 18–24. https://doi.org/10.1109/XSW.2013.7
[7] D.A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A.J. Kunen, O. Pearce, P. Robinson, B.S. Ryujin, T.R. Scogland, RAJA: portable performance for large-scale scientific applications, in: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), 2019, pp. 71–81. https://doi.org/10.1109/P3HPC49587.2019.00012
[8] A. Rasch, J. Bigge, M. Wrodarczyk, R. Schulze, S. Gorlatch, dOCAL: high-level distributed programming with OpenCL and CUDA, J. Supercomput. 76 (2020) 5117–5138. https://doi.org/10.1007/s11227-019-02829-2
[9] Y. Torres, F.J. Andújar, A. Gonzalez-Escribano, D.R. Llanos, Supporting efficient overlapping of host-device operations for heterogeneous programming with CtrlEvents, J. Parallel Distrib. Comput. 179 (2023) 104708, https://doi.org/10.1016/j.jpdc.2023.04.009
[10] A. Alpay, B. Soproni, H. Wünsche, V. Heuveline, Exploring the possibility of a hipSYCL-based implementation of oneAPI, in: Proceedings of the 10th International Workshop on OpenCL, IWOCL '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 1–12. https://doi.org/10.1145/3529538.3530005
[11] Intel Corporation, Intel oneAPI webpage, 2024, https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html (accessed July 9, 2024).
[12] R. Carratalá-Sáez, Y. Torres J. Sierra-Pallares S. López-Huguet D. R. Llanos, UVaFTLE: Lagrangian finite time Lyapunov exponent extraction for fluid dynamic applicationl, J. Supercomput. 79 (2023) 9635–9665. https://doi.org/10.1007/s11227-022-05017-x
[13] R. Carratalá, et al., Git Repository of UVaFTLE project, 2023, https://github.com/uva-trasgo/UVaFTLE (accessed July 9, 2024).
[14] AdaptiveCpp, Home of the AdaptiveCpp Project, 2024, https://github.com/AdaptiveCpp/AdaptiveCpp (accessed July 9, 2024).
[15] TriSYCL, The triSYCL Project, 2016, https://github.com/triSYCL/triSYCL (accessed Jun 20, 2024).
[16] A. Murray, E. Crawford, Compute Aorta: a toolkit for implementing heterogeneous programming models, in: Proceedings of the International Workshop on OpenCL, IWOCL '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 1–2. https://doi.org/10.1145/3388333.3388652
[17] Codeplay Software Ltd., The Future of ComputeCpp, 2023, https://codeplay.com/portal/news/2023/07/07/the-future-of-computecpp (accessed July 14, 2024).
[18] A. Alpay, V. Heuveline, One pass to bind them: the first single-pass SYCL compiler with unified code representation across backends, in: Proceedings of the 2023 International Workshop on OpenCL, IWOCL '23, Association for Computing Machinery, New York, NY, USA, 2023, pp. 1–12. https://doi.org/10.1145/3585341.3585351
[19] G. Haller, Lagrangian coherent structures, Annu. Rev. Fluid Mech. 47 (2015) 137–162. https://doi.org/10.1063/1.3690153
[20] S.S. Meschi, A. Farghadan, A. Arzani, Flow topology and targeted drug delivery in cardiovascular disease, J. Biomech. 119 (2021) 110307. https://doi.org/10.1016/j.jbiomech.2021.110307
[21] S. Brunton, C. Rowley, Modeling the unsteady aerodynamic forces on small-scale wings, in: 47th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition, 2009, p. 1127. https://doi.org/10.2514/6.2009-1127
[22] M. Serra, P. Sathe, F. Beron-Vera, G. Haller, Uncovering the edge of the polar vortex, J. Atmos. Sci. 74 (11) (2017) 3871–3885. https://doi.org/10.1175/JAS-D-17-0052.1
[23] M. Mikolajczak, Designing and building parallel programs: concepts and tools for parallel software engineering [book review], IEEE Concurr. 5 (2) (1997) 88–90. https://doi.org/10.1109/MCC.1997.588301
[24] D.J. Mavriplis, Unstructured grid techniques, Annu. Rev. Fluid Mech. 29 (1) (1997) 473–514. https://doi.org/10.1146/annurev.fluid.29.1.473
[25] Intel Corporation, C/C++ OpenMP and SYCL Composability, 2024, https://www.intel.com/content/www/us/en/docs/oneapi/programming-guide/2024-1/c-c-openmp-and-sycl-composability.html (accessed July 9, 2024).
[26] Khronos OpenCL working group, et al., Ranged accessors on SYCL 2020 Specification (revision 7), 2020, https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html#sec:accessors.ranged (accessed July 9, 2024).

[27] LUMI consortium, LUMI supercomputer webpage, 2025, https://lumi-supercomputer.eu/ (accessed February 20, 2025).
[28] C. Coulliette, S. Wiggins, Intergyre transport in a wind-driven, quasigeostrophic double gyre: an application of lobe dynamics, Nonlinear Process. Geophys. 7 (1/2) (2000) 59–85. https://doi.org/10.5194/npg-7-59-2000
[29] X.H. Zhao, K.H. Kwek, J.B. Li, K.L. Huang, Chaotic and resonant streamlines in the ABC flow, SIAM J. Appl. Math. 53 (1) (1993) 71–77. https://doi.org/10.1137/0153005
[30] T.J. McCabe, A complexity measure, IEEE Trans. Softw. Eng. SE-2 (4) (1976) 308–320. https://doi.org/10.1109/TSE.1976.233837
[31] M.H. Halstead, Elements of Software Science (Operating and Programming Systems Series), Elsevier Science Inc., 1977. https://doi.org/10.5555/540137
[32] Z. Wang, Y. Plyakhin, C. Sun, Z. Zhang, Z. Jiang, A. Huang, H. Wang, A source-to-source CUDA to SYCL code migration tool: Intel® DPC++ compatibility tool, in: International Workshop on OpenCL, IWOCL'22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 1–2. https://doi.org/10.1145/3529538.3529562
[33] L. Solis-Vasquez, E. Mascarenhas, A. Koch, Experiences migrating CUDA to SYCL: a molecular docking case study, in: IWOCL '23: Proceedings of the 2023 International Workshop on OpenCL, Association for Computing Machinery, New York, NY, USA, 2023, pp. 1–11. https://doi.org/10.1145/3585341.3585372
[34] G. Castaño, Y. Faqir-Rhazoui, C. García, M. Prieto-Matías, Evaluation of Intel's DPC++ compatibility tool in heterogeneous computing, J. Parallel Distrib. Comput. 165 (2022) 120–129. https://doi.org/10.1016/j.jpdc.2022.03.017
[35] Z. Jin, H. Finkel, A case study of k-means clustering using SYCL, in: 2019 IEEE International Conference on Big Data (Big Data), 2019, pp. 4466–4471. https://doi.org/10.1109/BigData47090.2019.9005555
[36] Z. Jin, V. Morozov, H. Finkel, A case study on the HACCmk routine in SYCL on integrated graphics, in: 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2020, pp. 368–374. https://doi.org/10.1109/IPDPSW50202.2020.00071
[37] Y. Faqir-Rhazoui, C. García, F. Tirado, Performance portability assessment: non-negative matrix factorization as a case study, in: Euro-Par 2022: Parallel Processing Workshops: Euro-Par 2022 International Workshops, Glasgow, UK, August 22-26, 2022, Revised Selected Papers, Springer-Verlag, Berlin, Heidelberg, 2023, p. 239–250. https://doi.org/10.1007/978-3-031-31209-0_18
[38] M. Goli, L. Iwanski, A. Richards, Accelerated machine learning using TensorFlow and SYCL on OpenCL devices, in: IWOCL 2017: Proceedings of the 5th International Workshop on OpenCL, IWOCL 2017, Association for Computing Machinery, New York, NY, USA, 2017, pp. 1–4. https://doi.org/10.1145/3078155.3078160
[39] R. Burns, J. Lawson, D. McBain, D. Soutar, Accelerated neural networks on OpenCL devices using SYCL-DNN, in: Proceedings of the International Workshop on OpenCL, IWOCL'19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 1–4. https://doi.org/10.1145/3318170.3318183
[40] Y. Faqir-Rhazoui, C. García, Exploring the performance and portability of the k-means algorithm on SYCL across CPU and GPU architectures, J. Supercomput. (2023). https://doi.org/10.1007/s11227-023-05373-2
[41] I.Z. Reguly, Evaluating the performance portability of SYCL across CPUs and GPUs on bandwidth-bound applications, in: Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis, SC-W '23, Association for Computing Machinery, New York, NY, USA, 2023, p. 1038-1047. https://doi.org/10.1145/3624062.3624180
[42] N. Fujita, B. Johnston, R. Kobayashi, K. Teranishi, S. Lee, T. Boku, J.S. Vetter, CHARM-SYCL: new unified programming environment for multiple accelerator types, in: Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis, SC-W '23, Association for Computing Machinery, New York, NY, USA, 2023, p. 1651-1661. https://doi.org/10.1145/3624062.3624244
[43] H.C. da Silva, F. Pisani, E. Borin, A comparative study of SYCL, OpenCL, and OpenMP, in: 2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW), 2016, pp. 61–66. https://doi.org/10.1109/SBAC-PADW.2016.19
[44] G.K. Reddy Kuncham, R. Vaidya, M. Barve, Performance study of GPU applications using SYCL and CUDA on Tesla V100 GPU, in: 2021 IEEE High Performance Extreme Computing Conference (HPEC), 2021, pp. 1–7. https://doi.org/10.1109/HPEC49654.2021.9622813
[45] T. Deakin, S. McIntosh-Smith, Evaluating the performance of HPC-Style SYCL applications, in: Proceedings of the International Workshop on OpenCL, IWOCL '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 1–11. https://doi.org/10.1145/3388333.3388643
[46] M. Breyer, A.V. Craen, D. Pflüger, Performance evolution of different SYCL implementations based on the parallel least squares support vector machine library, in: Proceedings of the 2023 International Workshop on OpenCL, IWOCL '23, Association for Computing Machinery, New York, NY, USA, 2023, pp. 1–12. https://doi.org/10.1145/3585341.3585369
[47] M. Breyer, A.V. Craen, D. Pflüger, A comparison of SYCL, OpenCL, CUDA, and OpenMP for massively parallel support vector machine classification on multi-vendor hardware, in: Proceedings of the 2022 International Workshop on OpenCL, IWOCL '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 1–12. https://doi.org/10.1145/3529538.3529980
[48] C.Q. Peralta, M.M. Trompouki, L. Kosmidis, Evaluation of SYCL's suitability for high-performance critical systems, in: Proceedings of the 2023 International Workshop on OpenCL, IWOCL '23, Association for Computing Machinery, New York, NY, USA, 2023, pp. 1–11. https://doi.org/10.1145/3585341.3585378
[49] H. Kim, H. Han, GPU Thread throttling for page-level thrashing reduction via static analysis, J. Supercomput. 80 (7) (2024) 9829–9847. https://doi.org/10.1007/s11227-023-05787-y
[50] F. Sadlo, A. Rigazzi, R. Peikert, Time-Dependent Visualization of Lagrangian Coherent Structures by Grid Advection, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 151–165. https://doi.org/10.1007/978-3-642-15014-2_13
[51] B. Nouanesengsy, T.Y. Lee, K. Lu, H.W. Shen, T. Peterka, Parallel particle advection and FTLE computation for time-varying flow fields, in: SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2012, pp. 1–11. https://doi.org/10.1109/SC.2012.93
[52] A. Kuhn, C. Rössl, T. Weinkauf, H. Theisel, A benchmark for evaluating FTLE computations, in: 2012 IEEE Pacific Visualization Symposium, 2012, pp. 121–128. https://doi.org/10.1109/PacificVis.2012.6183582
[53] C.M. Chen, H.W. Shen, Graph-based seed scheduling for out-of-core FTLE and pathline computation, in: 2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV), 2013, pp. 15–23. https://doi.org/10.1109/LDAV.2013.6675154
[54] F. Wang, L. Deng, D. Zhao, S. Li, An efficient preprocessing and composition based finite-time Lyapunov exponent visualization algorithm for unsteady flow field, in: 2016 International Conference on Virtual Reality and Visualization (ICVRV), 2016, pp. 497–502. https://doi.org/10.1109/ICVRV.2016.89
[55] C. Garth, F. Gerhardt, X. Tricoche, H. Hans, Efficient computation and visualization of coherent structures in fluid flow applications, IEEE Trans. Vis. Comput. Graph. 13 (6) (2007) 1464–1471. https://doi.org/10.1109/TVCG.2007.70551
[56] T.F. Dauch, T. Rapp, G. Chaussonnet, S. Braun, M.C. Keller, J. Kaden, R. Koch, C. Dachsbacher, H.J. Bauer, Highly efficient computation of finite-time Lyapunov exponents (FTLE) on GPUs based on three-dimensional SPH datasets, Comput. Fluids 175 (2018) 129–141. https://doi.org/10.1016/j.compfluid.2018.07.015
[57] M. Lin, M. Xu, X. Fu, GPU-Accelerated computing for Lagrangian coherent structures of multi-body gravitational regimes, Astrophys. Space Sci. 362 (2017) 1572–9460. https://doi.org/10.1007/s10509-017-3050-y
[58] M. Hlawatsch, F. Sadlo, D. Weiskopf, Hierarchical line integration, IEEE Trans. Vis. Comput. Graph. 17 (8) (2011) 1148–1163. https://doi.org/10.1109/TVCG.2010.227
[59] C. Garth, G.S. Li, X. Tricoche, C.D. Hansen, H. Hagen, Visualization of Coherent Structures in Transient 2D Flows, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 1–13. https://doi.org/10.1007/978-3-540-88606-8_1
[60] A. Sagristà, S. Jordan, F. Sadlo, Visual analysis of the Finite-Time Lyapunov Exponent, Comput. Graphics Forum 39 (3) (2020) 331–342. https://doi.org/10.1111/cgf.13984
[61] C. Conti, D. Rossinelli, P. Koumoutsakos, GPU and APU computations of Finite Time Lyapunov Exponent fields, J. Comput. Phys. 231 (5) (2012) 2229–2244. https://doi.org/10.1016/j.jcp.2011.10.032
[62] AdaptiveCpp, et al., Expose buffer page size as property for concurrent kernel buffer access, 2021, https://github.com/AdaptiveCpp/AdaptiveCpp/pull/513 (accessed February 1, 2025).

Francisco J. Andújar received the M.Sc. degree in Computer Science from the University of Castilla-La Mancha, Spain, in 2010, and the Ph.D. degree from the University of Castilla-La Mancha in 2015. He worked in the Universitat Politècnica de València under a post-doctoral contract Juan-de la Cierva, and currently works as Associate Professor on Computer Science at Universidad de Valladolid. His research interests include multicomputer systems, cluster computing, HPC interconnection networks, switch architecture, and simulation tools.

Rocío Carratalá-Sáez received a B.Sc. Degree in Computational Mathematics by Universitat Jaume I (UJI) of Castell ón (Spain) in 2015, M.Sc. Degree in Parallel and Distributed Computing by Universitat Politècnica de València (Spain) in 2016, and Ph.D. in Computer Science by UJI in 2021. She is currently an Assistant Professor at Universitat de València in the Department of Computer Science. Her main research interest is High-Performance Computing, focused on the parallelization of linear algebra operations and scientific applications. More information about her research can be found at http://rociocarratalasaez.es/

Yuri Torres de la Sierra received the B.S. degree in Computer Science and Engineering from University of Valladolid, Spain, in 2009. He received the M.S. degree in Information Communications in 2010, and the Ph.D. degree in Computer Science in 2014, both from the University of Valladolid, Spain. From 2014 to 2017, he was Associate Professor at Isabel I University, Burgos, Spain. He is currently Assistant Professor of computer science at the Universidad of Valladolid. His research interests include parallel and distributed computing, parallel programming models, and embedded computing. More information about his current research activities can be found at http://www.infor.uva.es/~yuri.torres.

Arturo Gonzalez-Escribano received his M.S. and Ph.D. degrees in computer science from the University of Valladolid, Spain, in 1996 and 2003, respectively. He is Associate Professor of computer science at the Universidad de Valladolid since 2008. He has participated in more than 100 scientific papers in journals and conferences. He has been principal researcher of national funded projects, lead several research contracts with enterprises, and participated in the committee of several international conferences. His research interests include parallel and distributed computing, parallel programming models, portability in heterogeneous systems, and embedded computing.

Diego R. Llanos received his MS and PhD degrees in Computer Science from the University of Valladolid, Spain, in 1996 and 2000, respectively. He is a recipient of the Spanish government's national award for academic excellence. Prof. Llanos is Full Professor of Computer Architecture at the Universidad de Valladolid, and his research interests include parallel and distributed computing, automatic parallelization of sequential code, and embedded computing. He is a Senior Member of the IEEE and Senior Member of the ACM, and has co-founded RDNest, a company that transfers to market research results in the field of Internet of Things and high-performance computing. More information about his current research activities can be found at http://www.infor.uva.es/~diego.