Future Generation Computer Systems 158 (2024) 44–59

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

Enabling performance portability on the LiGen drug discovery pipeline

Luigi Crisci [a,*], Lorenzo Carpentieri [a], Biagio Cosenza [a], Gianmarco Accordi [b], Davide Gadioli [b], Emanuele Vitali [c], Gianluca Palermo [b], Andrea Rosario Beccari [d]

[a] Department of Computer Science (DI), University of Salerno, Via Giovanni Paolo II 132, 84084, Fisciano, Italy
[b] Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB), Politecnico di Milano, Milan, Italy
[c] CSC - IT Center for Science, Espoo, Finland
[d] Dompé Farmaceutici S.p.A., Naples, Italy

ARTICLE INFO

Keywords:
Drug discovery
Molecular dynamics
Virtual screening
Performance portability
SYCL
HPC

ABSTRACT

In recent years, there has been a growing interest in developing high-performance implementations of drug discovery processing software. To target modern GPU architectures, such applications are mostly written in proprietary languages such as CUDA or HIP. However, with the increasing heterogeneity of modern HPC systems and the availability of accelerators from multiple hardware vendors, it has become critical to be able to efficiently execute drug discovery pipelines on multiple large-scale computing systems, with the ultimate goal of working on urgent computing scenarios. This article presents the challenges of migrating LiGen, an industrial drug discovery software pipeline, from CUDA to the SYCL programming model, an industry standard based on C++ that enables heterogeneous computing. We perform a structured analysis of the performance portability of the SYCL LiGen platform, focusing on different aspects of the approach from different perspectives. First, we analyze the performance portability provided by the high-level semantics of SYCL, including the most recent group algorithms and subgroups of SYCL 2020. Second, we analyze how low-level aspects such as kernel occupancy and register pressure affect the performance portability of the overall application. The experimental evaluation is performed on two different versions of LiGen, implementing two different parallelization patterns, by comparing them with a manually optimized CUDA version, and by evaluating performance portability using both known and ad hoc metrics. The results show that, thanks to the combination of high-level SYCL semantics and some manual tuning, LiGen achieves native-comparable performance on NVIDIA, while also running on AMD GPUs.

1. Introduction

The drug discovery process is of critical importance in the medical field and serves as a fundamental task in the identification of new drugs. This intricate and resource-intensive endeavor encompasses various stages, including *in silico*, *in vitro*, and *in vivo* phases. The initial step, known as the screening phase, marks the outset of drug discovery. Here, a collection of drug candidates undergo evaluation against specific targets. The outcome yields a subset of candidates advancing to subsequent pipeline stages. However, the increasing cost of *in vitro* and *in vivo* experiments constrains the number of drug candidates that can be evaluated in the drug discovery pipeline. Recent studies demonstrated that the introduction of an *in silico* stage, named virtual screening, to select which molecule test in a drug discovery pipeline, increases its success probability [1,2]. Unfortunately, the process of evaluating a drug candidate is a computationally intensive task. With candidates numbering in the millions or billions, it is necessary to run the calculations on High-Performance Computing (HPC) systems.

Recent years have seen a surge of interest in drug discovery, especially in light of the COVID-19 pandemic. Projects such as Exscalate4COV [3] or the COVID-19-HPC-Consortium [4] have emerged to rapidly identify effective drugs. When a novel disease appears, having fast and reliable drug discovery pipelines ready enables urgent computing scenarios, where the computational power of supercomputers around the world can be exploited immediately to respond quickly to the disease. In order to meet urgent computing requirements, a drug discovery pipeline needs to fully exploit modern supercomputers' hardware heterogeneity. Among the first ten positions of the Top500 list [5], nine supercomputers are GPU-accelerated, with GPUs from three different vendors. However, developing heterogeneous applications is challenging. Computing devices can differ in terms of execution model, optimal access pattern, and tuning, and require proprietary programming models (e.g., CUDA for NVIDIA GPUs, HIP for AMD GPUs, LevelZero for Intel GPUs) that undermine application portability across

* Corresponding author.
E-mail address: lcrisci@unisa.it (L. Crisci).

https://doi.org/10.1016/j.future.2024.03.045
Received 12 October 2023; Received in revised form 10 January 2024; Accepted 30 March 2024
Available online 16 April 2024

devices, requiring multiple native implementations to target all devices. A drug discovery software pipeline written in a native, device-specific language is not ideal because it is not portable, drastically undermining the ability to operate under urgent computational demands. To overcome the portability limitation, it is necessary to build the application source code so that it can run on a defined set of hardware without requiring multiple device-specific implementations. Furthermore, we also want to achieve the best performance on each of the hardware. In this regard, the term performance portability encompasses both aspects of portability and performance.

This work proposes a *performance-portable* implementation of LiGen [6], a high-throughput virtual screening application. LiGen is an industrial drug discovery software platform that consists of several implementations of the docking and scoring functionality: two main implementations called latency-oriented LiGen and throughput-oriented LiGen are provided [7], each with a C++ implementation for CPUs and a highly optimized CUDA C++ version targeting NVIDIA GPUs. We ported the CUDA code of both LiGen versions to SYCL 2020, exploiting several features such as sub-groups and group algorithms. To better understand the performance aspects of specific SYCL semantics, we developed two SYCL variants of each version, based respectively on the two SYCL memory models: the Unified Shared Memory and Buffer-Accessors paradigm. The new SYCL implementation can target multicore CPUs as well as GPUs from different vendors, and it has been tested on a broad range of hardware, including NVIDIA V100S and AMD MI100 GPUs.

Our work is not limited to the porting activities of the LiGen drug discovery platform but goes further by analyzing the performance portability of the resulting code. In particular, our analysis focuses on understanding how performance portability is affected by the high-level semantics provided by the SYCL programming model and how much low-level tuning is required to achieve a level of optimization that guarantees high performance on all target devices.

The paper makes the following contributions:

- A portable implementation of the LiGen industrial drug discovery platform, based on the Khronos SYCL 2020 standard and capable of targeting a wide range of multicore CPUs, GPUs, and accelerators.
- A performance portability analysis of SYCL 2020 semantics, including memory access models such as buffer-accessor and unified shared memory, group algorithms, and sub-groups, evaluated their impact on improving the portability of LiGen using the well-known performance portability metrics.
- An analysis of advanced portability issues that go beyond the SYCL language semantics such as occupancy, register pressure, and work-group size tuning.
- An experimental evaluation of the SYCL LiGen platform on NVIDIA V100S and AMD MI100 GPUs, performed with different SYCL implementations and memory access semantics, and compared to the manually optimized reference CUDA version. Before this article, no performance portability analysis of LiGen have been carried out.

The rest of this article is organized as follows. Section 2 provides a background on the drug discovery process, SYCL, and metrics for performance portability and roofline efficiency. The Ligen platform is described in detail in Section 3. Sections 4 and 5 focus on the performance portability aspects of the platform related to the SYCL language semantics and low-level details, respectively. Section 6 presents an extensive performance evaluation of the LiGen platform, while Section 7 further discusses and analyzes LiGen performance portability. Section 8 highlights the major lesson learned that emerged from the migration activity. Finally, Sections 9 and 10 conclude the article with related work and conclusions.

2. Background

This section presents the background of the paper on its main directions. Given the heterogeneity nature of the domain targeted, we split the section in four: state-of-the-art on virtual screening for drug discovery, background on SYCL, code efficiency using the roofline model, and the main definitions of performance portability.

2.1. Virtual screening

In the context of virtual screening, a drug candidate is a small molecule named *ligand*, with usually less than a hundred atoms. The goal of a virtual screening campaign is to rank a library of ligands according to their interaction strength against the target protein(s), which represent the disease. Domain experts will use this information to select which are the molecules that undergo *in vitro* experiments.

The evaluation of the interaction strength between a protein-ligand pair is composed of two tasks. The first one, named molecular docking, aims at estimating the 3D displacement of the ligand's atoms when they interact with a target protein. This is a complex task due to the number of degrees of freedom involved in the problem. Since the protein has tens of thousands of atoms, there are multiple areas of the target protein that a ligand can use for docking. Typically, they are cavities in the protein surface. In this article, we use the term *pocket* to identify each interaction area. Moreover, the ligand is not a fixed structure. A subset of the ligand's bonds, named rotatable bonds, split the molecule atoms into two disjoint sets that can rotate along the bond axis, without altering its chemical and physical properties. In this article, we will refer to each set of atoms derived by a rotatable bond as *fragment*. A ligand can have tens of rotatable bonds, that can be used to drastically change the molecule shape, named *conformer*. The second task of virtual screening uses a scoring function to estimate the interaction strength using the geometrical, chemical, and physical properties of both molecules. These are two well-known problems in literature that need to be solved for several purposes. For this reason, there is a wide range of solutions that cover the trade-off spectrum between performance and accuracy [8–11].

Recently, virtual screening applications have been accelerated by the processing power of GPUs [12–17]. AutoDock [18] has been ported in CUDA (AutoDock-GPU [19]) and deployed on the Summit supercomputer, where they docked over one billion molecules on two SARS-CoV-2 proteins in less than two days [20]. From this experiment, they derived a mini-app based on AutoDock-GPU to test different offloading schemes such as HiP and Kokkos [21]. In this article, we focus on *LiGen*, the virtual screening application of the EXSCALATE platform [3]. It has been designed from scratch to target supercomputers, not only to hinge on the overall number of GPUs available [7] but also to access the IO system efficiently [22]. Recently, LiGen has been deployed on Marconi100 at CINECA and HPC5 at ENI to virtual screen 72 billion of ligands against 15 pockets of 12 SARS-CoV-2 proteins, performing overall one trillion of protein-ligand evaluations in 60 h [6].

2.2. SYCL programming model

SYCL is a royalty-free, cross-platform C++ abstraction layer that allows developers to write applications that leverage multiple heterogeneous devices, including CPUs, GPUs, and FPGAs, in a convenient and performance-portable manner. SYCL extends the C++ programming language by introducing abstractions for handling heterogeneous computing within ISO C++ while striving to closely align with the core language's specification. While initially designed to be mapped onto OpenCL, revision 3 of the SYCL 2020 specification has opened the doors for additional custom backends [23], such as NVIDIA CUDA, AMD HIP, OpenMP, and more. Currently SYCL 2020 is at its 7th revision and it is heading towards the 8th revision by the end of 2023 [24]. The primary implementations of SYCL include OneAPI Data-Parallel C++,

developed by Intel [25], and AdaptiveCPP, developed at Heidelberg University [23], with several other minor implementations actively under development [26,27]. Furthermore, the flexibility of SYCL has given rise to various extensions targeting specific heterogeneous use cases, such as distributed computing [28], real-time energy optimization [29], and approximate computing [30]. Although SYCL allows for compiling a single-source code for multiple architectures, it does not handle performance portability automatically, and the efficiency of the same kernels can significantly vary across different hardware platforms. Nevertheless, SYCL provides a robust foundation where developers can write highly specialized code in a more accessible manner, aiming to achieve performance portability across a wide range of hardware architectures.

2.3. Roofline efficiency

The roofline model [31] is a two-dimensional graph that couples together the floating-point performance, data locality, and memory performance of an application in an intuitive way. It can tell if an application is either compute-bound or memory-bound, allowing for fine-grained optimizations. The roofline analysis uses the roofline model to quantify the performance of a target application with respect to the roofline peak performance, defined as *roofline efficiency* [32,33]. To compute this metric for a target application on a specific device m, one must first calculate the floating point peak ratio (FR_m) and maximum bandwidth (BW_m). The *balance point* is defined as the ratio of peak FP performance and bandwidth $B_m = FR_m \ / \ BW_m$, which represents the transition point from a memory-bound to a compute-bound application. When profiling the application k targeting a device m, one must compute the number of floating point operation FL_k and memory traffic TL_k which are used to compute the application *arithmetic intensity*. $AI_k = FL_k \ / \ TL_k$.

This value quantifies how many floating point operations the application does per transferred byte: a higher value ($AI_k > B_m$) indicates that the application is compute-bound, otherwise ($AI_k \leq B_m$) its memory-bound. Those values define the device peak floating-point performance P_k, defined as:

$$P_k = min \begin{cases} FR_m \\ BW_m * AI_k \end{cases} \tag{1}$$

Finally, the roofline efficiency E_k of a specific kernel is defined as the ratio of the kernel floating point rate FR_k over the device floating-point peak performance:

$$E_k = \frac{FR_k}{P_k} \tag{2}$$

The roofline efficiency allows for quantification of the application performance on a per-hardware basis using two simple metrics: floating point ratio and memory bandwidth.

2.4. Performance portability

HPC systems constantly evolve towards novel and different architectural designs. In particular, hardware heterogeneity has emerged as a key point. System architects mix different devices, such as CPUs and GPUs, to achieve the required functionality and performance. However, applications written and tuned for a specific target architecture cannot easily support different ones, especially with new emerging hardware not available at the developing time. As the range of hardware in modern HPC systems increases, it is crucial to have application code that can efficiently run on different devices. In this scenario, it is reductive to consider the efficiency of an application only for a specific hardware. It is essential to provide a quantitative metric that can include both the performance attained and the variety of devices on which the application can run. To this aim, the term performance portability has become increasingly popular in literature and encapsulates two aspects: Achieving some notional level of performance on the target platforms (performance); the ability to run an application across multiple hardware platforms (portability). The DoE [34] and ETP4HPC [35] have highlighted the importance of performance portability for HPC, but also recognized that there is not yet a universally accepted definition as it has different meanings among different application domains. In our study, we adopt the Pennycook et al. [36,37] performance portability definition: "A measurement of an application's performance efficiency for a given task that can be run successfully on all platforms in a given set". The metric to quantify performance portability is shown in Eq. (3):

$$\mathbf{\Phi}(a, p, H) = \begin{cases} \dfrac{\mid H \mid}{\sum_{i \in H} \dfrac{1}{e_i(a,p)}}, & \text{if } i \text{ is supported } \forall i \in H \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

where a a is the application, p the problem solved by a, and H is the set of target hardware. The $\mathbf{\Phi}$ metric is defined as the harmonic mean of application performance efficiency $e_i(a, p)$ with respect to a given set of hardware H.

Pennycook et al. [38] highlight various methods for calculating application performance efficiency, specifically: *architectural efficiency*, which measures achieved performance as a fraction of peak hardware performance; and *application efficiency*, which measures achieved performance as a fraction of the best observed performance against the most optimized native implementation. Determining the application's architectural or application efficiency can be complex, requiring the identification of relevant bottlenecks on each hardware or an optimized implementation for each platform. Interestingly, recent work has demonstrated that roofline efficiency can successfully approximate architectural efficiency [32]. To align the initial Pennycook et al. definition, for the rest of the manuscript we will refer to the roofline efficiency as $e'_i(a, p)$. In addition, we will use $e''_i(a, p)$ to refer to the *native efficiency*, a novel, ad-hoc efficiency metric which is introduced in Section 7.2.

3. The LiGen virtual screening application

An important feature of the virtual screening problem is that all the evaluations of ligand-protein pairs are independent. This is true at different levels of the processing pipeline. Given a specific protein pocket, each ligand and ligand conformation can be processed in parallel. Within a protein, multiple pockets can be processed independently, as independent could be the processing of multiple proteins and/or protein conformation. In fact, the embarrassingly parallel nature of the problem provides opportunities for parallelization at various levels, allowing for the processing of an enormous amount of data.

This section provides an overview of the three-stage processing of the LiGen application, focusing on the two GPU implementations optimized for latency and batch processing, respectively.

3.1. LiGen application overview

LiGen uses MPI to replicate on different nodes the same computational pipeline, where each pipeline processes its fraction of the input ligands [6]. It uses native threads to implement an asynchronous pipeline that reads, parse, evaluate, and write the results in the output files. The writing operation introduces the only synchronization among processes, to avoid overwriting each other. In this article, we focus on the evaluation stage, which is in charge of docking and scoring a ligand inside a target pocket. LiGen can have multiple targets, but they will be processed serially, thus we can focus on the single pocket case without losing in generality. Algorithm 1 shows the pseudo-code of the virtual screening algorithm for evaluating the interaction strength of a ligand-pocket pair. The LiGen's virtual screening algorithm is composed of the following three phases:

Algorithm 1: LiGen virtual screening algorithm

```
   Data: num_restart, num_iterations, max_num_poses
   Input: ligand, target
   Output: score
1  scores ← ∅, poses ← ∅;
2  for i ← 0 to num_restart do
3      pose ← initialize_pose(ligand, i);
4      pose ← align(pose, target);
5      for n ← 0 to num_iterations do
6          for fragment ← pose.fragments do
7              pose ← optimize(pose, fragment, target);
8          end
9      end
10     pose ← evaluate(pose, target);
11     poses ← poses ∪ {pose};
12 end
13 poses ← clip(sort(poses), max_num_poses);
14 for pose ← poses do
15     score ← compute_score(pose, target);
16     scores ← scores ∪ {score};
17 end
18 return max(scores)
```

1. **Docking phase** (lines 2–12): in this phase, we take as input the target pocket and ligand, and we output *num_restart* alternative poses of the ligand when it interacts with the target pocket. This is the only step that alters the displacement of the atom poses.
2. **Pose filtering** (line 13): this optional phase reduces the number of poses to score according to the *max_num_poses* parameter. The idea is that we want to score the most promising ones according to a simple geometric scoring function while promoting diversity.
3. **Scoring phase** (lines 14–18): The remaining poses are then evaluated using a chemical scoring function [39]. The score of the ligand is the maximum score among its poses.

Each ligand computation is independent, thus multiple ligands can be computed concurrently. Furthermore, pose computation and evaluation within the docking (line 2) and scoring (line 14) phases are also independent, exposing additional parallelism. In the following sections, we will describe in more detail each of these phases.

3.1.1. Docking phase

The LiGen docking algorithm is a gradient descent with multiple restarts. The number of poses that are generated for each input ligand depends on the number of restarts (line 2). At each restart, a pose is initialized using a deterministic heuristic that depends on the pose index (line 3). At first, LiGen aligns the pose to the target pocket using rigid roto-translations (line 4). Then, we take into account the ligand flexibility by optimizing all its fragments (lines 6–8). The user can configure the number of times that this optimization process is repeated, by setting the parameter *num_iterations*. Finally, we compute some properties of the pose that will be used later in the filtering phase (line 10) and append the pose to the set of computed poses (line 11).

3.1.2. Pose filtering phase

To reduce the computation effort of the scoring phase, LiGen keeps only the most promising ones, according to the *max_num_poses* parameter. To reach this goal, we first sort them according to a simple custom geometric score. Then, we keep track of the poses that have internal bumps or bump against the protein. Finally, we compute a clustering of the poses according to their similarity. Using this information we can sort the poses to have at lower indexes the best representative of each cluster. In the middle, we have all the poses that do not have any bump, while we leave at higher indexes the remaining poses. When we need to score only a subset of the poses, we start by considering the ones at lower indexes, which represent the most promising ones (line 13).

3.1.3. Scoring phase

In this phase we compute the actual score of the poses, using the LiGen scoring function [40] (line 14,15). Even if the scoring function evaluates several chemical and physical properties, the final score of a pose is reduced to a numerical value. The score of the ligand is equal to the highest that we were able to measure (line 18). Since all the algorithms are deterministic, we can reproduce a pose on demand. For this reason, there is no need to store the atom's displacement of the best pose.

3.2. GPU kernel implementations

When we focus on the CUDA implementation of the virtual screening computation kernels, there are two main ways of implementing them. The more straightforward approach aims at spreading the computation across the GPU cores, lowering the execution time. In this article, we name this approach *latency*, and it is the one that we used on the largest virtual screening campaign [6]. This is also the approach used by AutoDock-GPU [19]. When the ligand computation is not enough to use all the resources available in a GPU card, we use multiple software threads and CUDA stream to have more instances running in parallel. As a general rule, the kernel names reported in Table 1 match the name reported in Algorithm 1. However, a single step can be implemented using more than one kernel. This is because all the operations shall use a different mechanism to spread the computation across the GPU. For reference, *optimize_kernel_1* rotate a ligand's fragment and compute the gradient value at different angles. Then *optimize_kernel_2* perform a reduction to understand which is the best orientation, taking into account internal bumps among the ligand's atoms. Finally, the *score_kernel_1* computes the hydrogen bonding contribution.

In a recent work [7], we notice how a *batched* implementation yields $5x$ the throughput of the latency one. In this implementation, we restrict the ligand computation to a single warp. Then, we hinge on the GPU parallelism by evaluating a large number of them in parallel. While this approach is the fastest, it increases the application complexity since we need to manage bundles of ligands. In particular, by changing how we group ligands and how many ligands we bundle in a batch, we impact the application performance. For this reason, we need to have a method to automatically handle them at runtime, using a CUDA and SYCL implementation [41]. Moreover, the kernel design differs between the two implementation designs. Using the batched design we can aggregate more computation steps in a single kernel since they are all computed by a single warp. Therefore, the kernel names in Table 2 tend to refer to the three main phases of the virtual screening algorithm. They can be implemented using one or more kernels, according to the data structures required to carry out the computation. For reference, *score_kernel_1* is the pre-processing phase that computes per-atom topology properties to evaluate hydrogen bonding. The *score_kernel_2* computes the actual score value for all the poses. Regarding the pose filtering phase, the *filtering_kernel_1* sorts only the indexes of the poses, while *filter_kernel_2* performs the actual memory movement in the internal data structures.

4. Performance portability by SYCL semantics

This section presents a performance portability analysis of the LiGen application based on language semantics. Specifically, we will look at several SYCL 2020 semantics, including memory access models such as buffer-accessor and unified shared memory, group algorithms, and sub-groups.

Table 1
Register pressure (register per thread) for LiGen Latency compiled for Volta architecture (CUDA capabilities 7.0).

Compiler	*initialize_pose*	*align*	*optimize_kernel_1*	*optimize_kernel_2*	*score_kernel_1*
NVCC 12.1	52	44	88	36	72
DPC++ (SYCL Accessors)	72	82	121	38	96
DPC++ (SYCL USM)	70	78	90	28	80
AdaptiveCpp (SYCL Accessors)	72	82	76	38	93
AdaptiveCpp (SYCL USM)	70	82	92	38	81

Table 2
Register pressure (register per thread) for LiGen Batch compiled for Volta architecture (CUDA capabilities 7.0). Note that since the dock kernel is specialized depending on the ligand's number of atoms, the register usage shown is the arithmetic mean of all kernel specializations.

Compiler	*dock*	*score_kernel_2*	*filtering_kernel_1*	*filtering_kernel_2*	*score_kernel_1*
NVCC 12.1	~105	103	56	63	54
DPC++ (SYCL USM)	~159	122	40	28	44
DPC++ (SYCL Accessors)	~169	122	48	32	56
AdaptiveCpp (SYCL Accessors)	~163	180	40	29	76
AdaptiveCpp (SYCL USM)	~158	140	44	32	64

4.1. Memory access models

The SYCL programming model provides two ways of handling data: the *Buffer-Accessor* and the *Unified Shared Memory* (USM) memory management interfaces. With the former, the user wraps data in objects called buffers, specifying both size and dimensionality (1D to 3D). When a kernel needs to access these data, the user creates an *accessor* that specifies which elements of the specified buffers will be accessed and how that access will be performed, i.e., read-only or write-only. The access properties of the accessors are also used by the runtime to construct an optimized kernel task graph. USM is a lower-level mechanism in which memory allocations and deallocations follow the malloc/free C paradigm. With USM, the user can allocate memory exclusively on the host, on the device, or shared between the two using an automatic memory migration system. However, USM is an optional feature that may not be supported by all devices, and devices that support USM may not support all types of USM allocations. LiGen uses CUDA manual memory management to handle allocations on the device, which can be partially directly mapped on USM with minimal effort. On the other hand, the Buffer-Accessors interface requires slight code refactoring but severely reduces code complexity, removing all the unnecessary malloc/free, thus improving LiGen memory safety.

4.2. Group algorithms

SYCL has several functions that provide functionality related to groups of work items (such as group barriers and collective operations). These group functions act as synchronization points and must be encountered in converged control flow by all work items in the group. If one work item in a group calls a group function, then all work items in that group must call precisely the same function under the same set of conditions. All group algorithm functions take a group as the first argument that defines which group of work items executes the specified function. Even though features like atomic, work-group local memory, and barrier may be used to perform group algorithm functions directly in a program, many devices come with specialized hardware to speed up certain group operations. Calling a built-in function will often perform better than writing a general-purpose implementation since vendor-provided group function implementations are generally tailored for the device they are operating on, even when a device does not feature specialized hardware. LiGen makes heavy usage of manually written group functions: in particular, group shifts and reductions are exploited in the docking phase to measure the atoms pair interactions, while in the scoring phase group reductions are used to retrieve the best-evaluated pose. SYCL group algorithms simplify the ported code by substituting highly specialized custom functions with standard and well-known procedures while leveraging optimized native implementation provided by the SYCL implementations.

4.3. Sub-groups

The SYCL sub-group represents a collection of related work items within a work group that executes concurrently. Combining sub-groups with group algorithms functions helps in building performance portable applications. In CUDA, threads within a thread block are divided into groups of 32 threads called *warp*, which are executed in parallel. Parallel programs often use collective communication operations, such as broadcasts, parallel reductions, and scans. CUDA C++ supports such collective operations by providing warp-level primitives. SYCL implementations maps sub-group into CUDA warps, simplifying the porting process. Furthermore, SYCL offers several sub-group intrinsics that map to CUDA collective warp functions. LiGen exploits several CUDA warp-level features, which are directly mapped to SYCL with minimal effort. However, as the sub-group size depends on the target hardware, which is fixed to 32 on NVIDIA hardware, LiGen's SYCL kernels have been refactored to handle several sub-group sizes in order to be portable to multiple architectures. This enables for additional optimizations on architectures that support several sub-group sizes (e.g. Intel GPUs).

5. Advanced performance portability issues

Although the high-level abstractions of SYCL 2020 help build portable programs, achieving high performance also requires additional device-specific optimization and tuning. This section focuses on some of the performance issues we encountered while evaluating performance portability. In particular, we focused on the high register pressure of the main LiGen kernels and on work group size tuning.

5.1. Occupancy and register pressure

When launching a kernel, the GPU splits the threads into blocks and distributes them to the GPU computational units. Each unit comes with a limited number of physical resources that have to be shared between all threads which can limit the number of threads that can run concurrently on the GPU. On GPUs, a metric of how much a kernel is filling the available resources is called *occupancy*. It is defined as the ratio of the number of active sub-groups per multiprocessor to the maximum number of possible active sub-groups.[1] Although a higher occupancy is not always synonymous with higher performance, as in some cases you can trade occupancy for improving kernel resource usage [42], it gives a good overview of the kernel performance. Furthermore, a

[1] SYCL sub-groups are mapped to hardware-specific thread collections. E.g. sub-groups are mapped to *warps* on NVIDIA GPUs and to *wavefronts* on AMD GPUs.

Table 3
Hardware used in this evaluation.

Name	Vendor	Core count	Compute units	Memory processor clock	Memory clock	Measured FP64 rate	Measured bandwidth	Release date
V100S	NVIDIA	5120	80	1245 MHz	1107 MHz	6.3 TFLOP/s	1.01 TB/s	2019
MI100	AMD	7680	120	1000 MHz	1250 MHz	10.5 TFLOP/s	0.89 TB/s	2020

lower occupancy inhibits the ability to hide GPU memory latency, thus degrading performance.

One of the features that can inhibit GPU occupancy is register usage. Registers are small, on-chip storage locations that are accessed exclusively by a single thread during the kernel execution. Registers are very limited resources and GPU architectures usually put limits on the maximum number of registers per thread. For example, the NVIDIA Volta architecture has a register file size of 256 kB per Compute Unit, with a limit of 255 registers per thread. When a kernel uses more registers than the addressable one, the surplus data are stored in the GPU's local memory and accessed at increased latency. This process is referred to as *register spilling*. Because registers bind to a single thread and are not shared within the kernel block, they constitute a hard cap on achievable occupancy. The amount of registers required by a kernel is defined by the compiler and is referred to as *register pressure*.

In SYCL, the memory access model for handling memory allocations can significantly impact the register pressure of the applications. While USM employs raw pointers for handling device data, SYCL accessors are much more complex objects that carry several pieces of information such as the buffer dimension, sizes, and data offsets. Holding such information comes with a price, and thus SYCL accessors usually require more space than USM. In Tables 1 and 2, we show the differences in required registers between the CUDA implementation and the SYCL implementations using the USM and Buffer-Accessors memory models on an NVIDIA Tesla V100S. We show this analysis on the 5 most time-consuming kernels, which take up to 95% of LiGen Latency and 98% of LiGen Batch execution time.

In LiGen Latency, the USM implementation performs better on most of the kernel, allocating on average 18% fewer registers than the buffer-accessors backend using DPC++ and ∼1% on AdaptiveCpp. Conversely, with the *optimize_kernel_1* kernel DPC++ and AdaptiveCpp exhibit opposite behaviors, with DPC++ allocating 59% more registers on the buffer-accessors backend. In LiGen Batch, for the smaller kernel (*filtering_kernel_1*, *filtering_kernel_2*, and *score_kernel_1*) the register allocation varies between CUDA and SYCL primarily because of different allocation policies between NVCC and Clang, with the buffer-accessor backend allocating more registers compared to USM. With *dock* and *score_kernel_2* kernels, the two most register-intensive kernels, CUDA shows a better register allocation policy compared to both SYCL USM and buffer-accessors. In particular, the SYCL implementation of the *dock* kernel takes up to 60% more registers compared to the CUDA implementation, with the buffer-accessors backend requiring 10 more registers compared to USM.

5.2. Work group size tuning

Thread allocation can severely impact application performance when targeting a broad range of hardware. LiGen SYCL has been designed to target GPU systems, thus choosing a proper work group size for each platform is required to exploit the best performance from the underlying hardware.

LiGen Latency. In LiGen Latency, each ligand is computed individually on the GPU. Each kernel computes multiple poses, which are distributed to the available sub-groups. LiGen Latency is composed of several small kernels with low register requirements. Furthermore, runtimes are in the order of hundreds of nanoseconds, thus any scheduling overhead can severely impact the overall application performance. For those reasons, the selected approach was to schedule the maximum amount of threads per work group. In CUDA, the block size is defined at runtime via a command line parameter, potentially leading to sub-optimal resource usage. In SYCL, we query the kernel maximum work group size for the target device using a specific descriptor[2] available from SYCL 2020 with the *kernel bundle* feature. This ensures the allocation of as many threads as possible and reduces kernel scheduling overhead. If the SYCL compiler does not support the *kernel bundle*, we fall back to a predefined smaller work group size.

LiGen Batch. In LiGen Batch, ligands are grouped together and computed concurrently on the target device. Each ligand is assigned to a sub-group which computes all the ligand poses, with each group computing several ligands concurrently. CUDA uses a pre-defined block size of 256 threads. As shown earlier in this section, LiGen Batch kernels suffer from high register pressure, and thus work group sizes are severely limited. However, as each ligand computation within the sub-groups is independent of each other, we do not expect significant differences by varying the work group size. Consequently, we again query the maximum work group size for the kernel to schedule the maximum amount of threads possible. Furthermore, we focused on the *dock* kernel and tested several work group sizes on NVIDIA Tesla V100S to explore how different work group sizes can influence performance. The results are shown in Section 7.

6. Performance evaluation

In this section, we present the performance evaluation of the novel SYCL implementations of LiGen Latency and LiGen Batch. After presenting the experimental setup and dataset selection, we compare LiGen Latency with the manually optimized CUDA version; this is followed by a comparison of the most performance-portable LiGen Batch version with the equivalent manually-tuned CUDA version. The analyses also investigate important performance issues such as the SYCL compilation toolchain and register optimization.

6.1. Experimental setup

Our focus is on evaluating the performance of these implementations on various GPU architectures. Given that the baseline implementation of LiGen was designed for NVIDIA GPUs, we have chosen to assess the performance exclusively on GPUs, omitting evaluation on other hardware platforms such as CPUs or FPGAs. This choice aligns with the prevalent utilization of GPU-based clusters in modern HPC systems.

To ensure comprehensive coverage of major GPU producers, we perform our analysis on NVIDIA and AMD GPUs. Although Intel has recently entered the HPC scenario with dedicated GPU solutions, we do not have access to an HPC Intel GPU so far, thus we do not include Intel hardware in our evaluation. In this paper, we used an NVIDIA Tesla V100S and an AMD MI100:

- *NVIDIA V100S GPU node.* The NVIDIA node features an Intel Xeon Gold 5218 CPU running at 2.30 GHz with 64 cores, accompanied by an NVIDIA Tesla V100S connected via PCI Express 4.0. The GPU consists of 80 Streaming Multiprocessors, for a total of 5120 cores running at 1.245 GHz alongside 620 Tensor Cores. A Tesla V100S reaches 8.2 TFLOP/s FP64, 16.4 TFLOP/s with FP32, 32.2 TFLOP/s with FP16, and 130 TFLOP/s using FP16 Tensor cores. The GPU is equipped with 32 GB HBM2 memory that can reach up to 1132 GB/s bandwidth, 128 kB L1 Cache per SM, 6 MB L2 Cache, and 256 Kb registers per SM.

[2] *sycl::info::kernel_device_specific::work_group_size*.

- *AMD MI100 GPU node.* The AMD node is equipped with an AMD EPYC 7313 CPU operating at 3.7 GHz, comprising 16 × 2 cores, and an AMD MI100 GPU connected via PCI Express 4.0. The GPU comprises 120 compute units with 7680 cores, each running at 1.0 GHz. The MI100 can deliver up to 11.5 TFLOP/s FP64, 23.1 TFLOP/s FP32, and 184.6 TFLOP/s using FP16. It is equipped with 32 GB HBM2 memory, reaching up to 1129 GB/s bandwidth, 16 kB L1 Cache per Compute Unit, 8 MB L2 Cache, and 256 kB registers.

We measured the maximum flop-rate and memory bandwidth of each GPU using the Empirical Roofline Toolkit (ERT) [43]. In Table 3 we report a summary of the hardware used in this evaluation, alongside the computed peaks. For the evaluation of SYCL implementations, we have utilized the OneAPI Data-Parallel C++ (DPC++) compiler (commit 0e8ce8c) developed by Intel and the AdaptiveCpp (ACPP) compiler (commit 5e9c4e5) developed at Heidelberg University. These compilers offer native backends that facilitate targeting of NVIDIA, and AMD GPUs. The experimental setup described above allows us to conduct a comprehensive performance analysis of the SYCL implementations of LiGen on different GPU architectures.

6.2. Datasets selection

We have curated a diverse set of datasets, each consisting of 100,000 ligands, with varying characteristics. The purpose of these datasets is to comprehensively assess the performance of the LiGen implementations across different hardware platforms. To achieve this, we have considered various combinations of atoms and fragment sizes, which allow us to analyze performance in different operational scenarios. The number of atoms in a ligand directly impacts the number of operations that each sub-group must perform. We have selected four different atom sizes: 31, 63, 74, and 89. These sizes provide a range of ligand complexities to evaluate the implementations' efficiency. Additionally, the number of fragments influences the optimization rounds required by the algorithm to analyze the ligand's degrees of freedom. We have included four fragment sizes: 4, 8, 16, and 20, enabling us to explore the impact of different fragment sizes on performance. In total, we have created 16 datasets, encompassing all possible combinations of the selected atom and fragment sizes. This ensures a comprehensive evaluation of the implementations' performance while accounting for hardware-specific characteristics. Through the rest of the article, to refer to a dataset with a specific atom and fragment size, we will use the signature *d_atoms_fragments*.

6.3. Ligen Latency vs. manually-tuned CUDA

In Figs. 1 and 2, we present the performance of the LiGen Latency SYCL porting. The results are presented for varying the number of atoms and fragments in the ligand. Notably, altering the number of atoms has a milder impact on throughput reduction compared to increasing the number of fragments. This discrepancy arises from the fact that each additional atom heightens the workload of individual threads, while the escalation of fragments introduces new kernel calls, significantly reducing overall performance. The SYCL USM backend outperforms the accessor backend on both platforms when dealing with a small number of atoms. As the number of atoms increases, performance tends to stabilize. This behavior can be attributed to the constant overhead introduced by sycl::accessors due to multi-dimensional indexing and offsets, which are not present in the USM approach. The impact of this overhead is more pronounced at lower atom numbers, where kernels complete in a few nanoseconds. However, as the number of atoms grows and threads are assigned more computation tasks, this overhead becomes less prominent.

NVIDIA Tesla V100S. The native CUDA implementation performs the best across all implementations, achieving a median throughput of 603 ligand/s. DPC++ achieves the best performance between the SYCL compilers, obtaining a median throughput of 408 and 446 ligand/s with the accessors and USM backend respectively, and a maximum of 73% of the native CUDA performance. On the other hand, AdaptiveCpp performs poorly, achieving only 252 ligand/s median throughput on the accessor backend and 294 ligand/s using the USM one, reaching up to 48% of the native CUDA throughput. Overall, the USM backend performs slightly better than the accessor one, obtaining 9% higher median throughput with DPC++ and 17% increment with AdaptiveCpp.

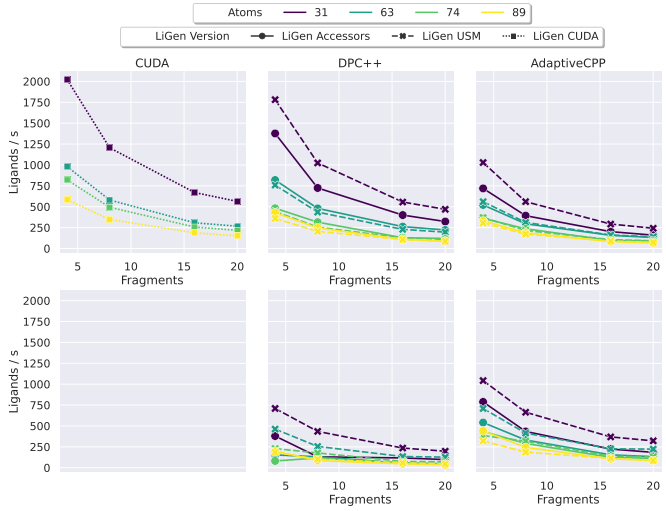
AMD MI100. AdaptiveCpp implementation shows the best performance between the SYCL compilers, achieving a median throughput of 352.9 and 289.2 ligand/s with the USM and accessors backend. DPC++ instead performs poorly, achieving 218.6 median throughput with the USM backend and 111 ligand/s with the accessor one.

6.3.1. DPC++ vs. AdaptiveCpp

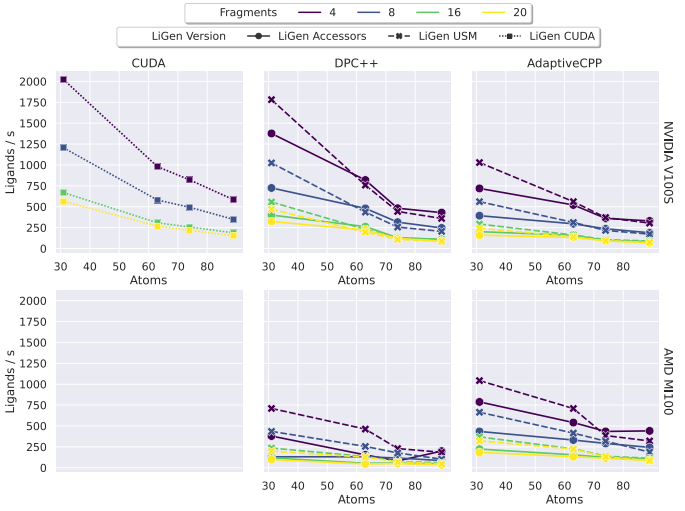
Both SYCL backends with AdaptiveCpp perform worse than the corresponding one built using DPC++ on NVIDIA V100S, which was unexpected as both compilers showed similar performance on NVIDIA hardware. To discern the reasons behind that significant performance disparity, we performed an in-depth analysis of the two implementations using *Nsight Compute*, an NVIDIA tool for CUDA application profiling. We profiled the *optimze fragment* kernel, which takes 90% of the overall execution time using the accessor's backend. Our analysis was performed on dataset *d_31_4*, which is the one that exhibits the higher performance difference, with a 1.91x slowdown. AdaptiveCpp takes on average 220 us for one iteration of the kernel, while the DPC++ version takes around 100 us, resulting in a 2.2x slowdown. The memory workload analysis showed that the AdaptiveCpp performs 285% more load/store instruction on L1/TEX Cache. In particular, it executes ∼500.000 local load and ∼250.000 local store, while DPC++ does not perform any local memory operations. We then analyzed the generated Parallel Thread Execution (PTX) code, the NVIDIA GPU intermediate representation, to highlight where those instructions are issued. We found out that the AdaptiveCpp version does not inline the kernel call, contrarily to DPC++: it creates the function stack frame by pushing all the kernel parameters in local memory and then passes to the actual kernel a pointer to them. This approach has the advantage of leveraging fewer registers, as the parameters are stored in a vector in local memory and loaded into the registers when necessary. On the other hand, it has several drawbacks: (i) the parameter packing introduces an initial overhead, which while negligible for heavy kernels, can be expensive for short ones; (ii) because parameters are stored in local memory as plain pointers, the compiler loose information about which kind of memory the pointer refers to (e.g. *global memory*, *local memory*, etc.). This forces the compiler to issue generic *load* and *store* instructions, which are resolved using generic addressing [44]. With this technique, the different memory spaces (*const*, *local*, and *shared*) are models as windows within the generic address space. When a generic *load/store* is executed, the hardware subtracts the window base of each memory state from the generic address and checks if it falls within that memory space window. This introduces a pointer subtraction overhead for each *load/store* operation performed; (iii) Local Memory in CUDA is stored in the device Global Memory in a way that consecutive 32-bit words are accessed by consecutive thread IDs to facilitate coalesced memory accesses. However, such load increases L1 Cache traffic and potentially affects the cache hit/miss ratio.

6.3.2. SYCL vs. manually-tuned CUDA

To discern the difference in performance between the CUDA and SYCL versions, we analyzed the application using the *Nsight Compute* profiler. Our analysis focused on the *optimize fragment* kernel from LiGen USM, using the dataset *d_31_4*, the one that shows the closest



(a) LiGen Latency throughput with a fixed number of atoms and variable fragments number



(b) LiGen latency throughput with a fixed number of fragments and variable atoms number

Fig. 1. LiGen latency throughput on NVIDIA and AMD hardware.

performance between the two versions. We found out that the SYCL version achieves −21% L1 cache hit and +81% L2 cache hit. This reduces memory throughput by 33% compared to the CUDA version. In addition, +69% branch instructions are issued in the SYCL version. We guess that the CUDA speedup is due to the use of *texture memory*, a special read-only memory that resides in device memory and is cached in texture cache, located on the same L1 cache and shared memory die, and optimized for spatial locality. In the *optimize fragment* kernel, which takes up to 90% of the overall time, LiGen CUDA stores the 3D pocket docking space in the texture memory. LiGen SYCL does not use texture memory as, at the time of writing, support is incomplete or absent in current SYCL implementations. We moved the pocket docking space to global memory and passed it to the kernel using a read-only accessor in LiGen Accessor and a const pointer in LiGen USM. However, accessing the pocket requires an expensive index calculation which contributes to the increased number of branching instructions. Moreover, we lose the data locality optimization from the texture cache, potentially increasing cache misses. This could also explain why performance drops when increasing the ligand's atoms number, as more access to the pocket space must be performed.

6.4. Ligen Batch vs. manually-tuned CUDA

In Figs. 3 and 4 we show the portability results achieved with LiGen Batch. The Batch version outperforms the latency implementation on all datasets, with both CUDA and SYCL backends respectively. Contrary to LiGen Latency, varying the number of atoms has a greater impact than increasing the number of fragments: this mostly happens because each fragment is computed in a for-loop within the *dock* kernel instead of issuing several kernel invocations. As LiGen Batch processes a higher number of ligands per kernel, the memory access differences between accessors and USM become minimal. Consequently, the two backends exhibit similar performance with both SYCL compilers.

NVIDA Tesla V100S. The CUDA native implementation again performs the best among all implementations, with a median throughput of 3043 ligands/s, achieving a 3.13x speedup compared to the corresponding LiGen Latency implementation.

Among the SYCL compilers, DPC++ delivers the highest performance. It achieves a median throughput of 1579.2 ligands/s with the accessors backend and 1619.4 ligands/s with the USM backend, reaching a maximum of 53% of the native CUDA performance. On the other hand, AdaptiveCpp achieves 1317.1 ligands/s and 1423.8 ligands/s

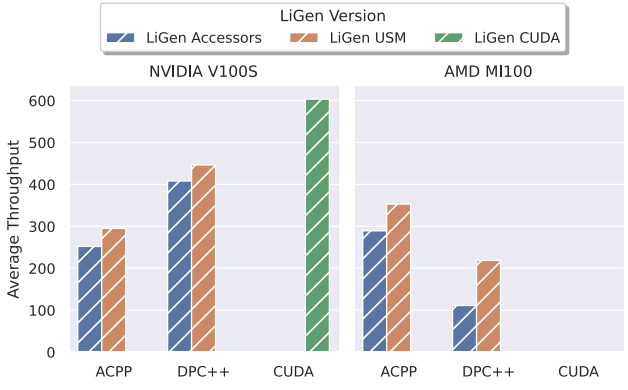


Fig. 2. LiGen Latency average throughput across all datasets, on NVIDIA V100S and AMD MI100.

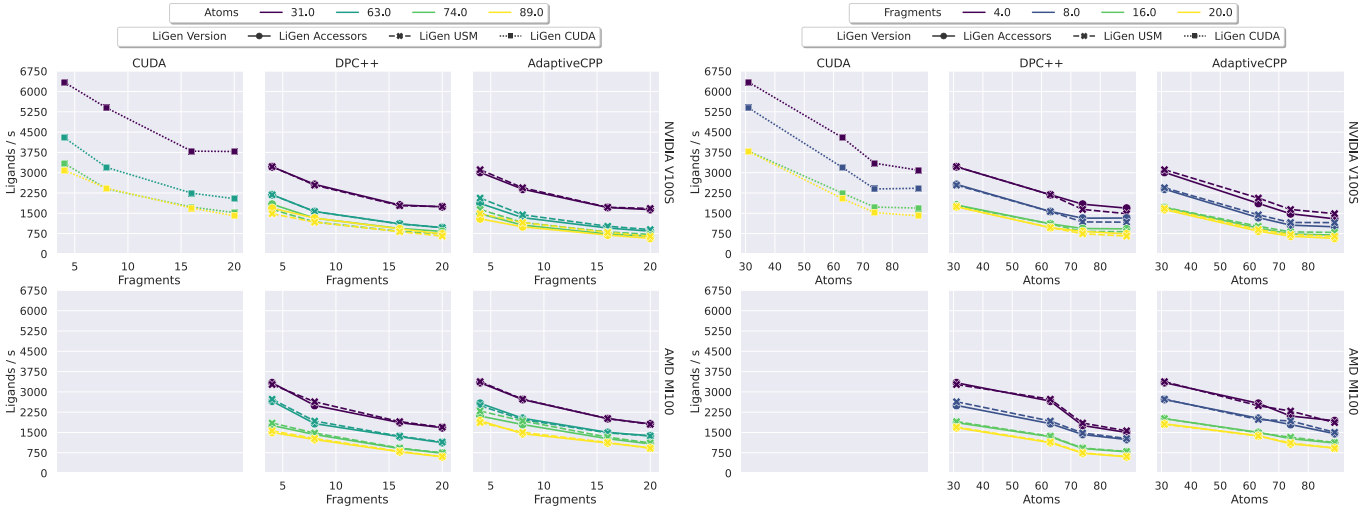
median throughput on the accessors and USM backend respectively, achieving 46% of the CUDA implementation.

AMD MI100. AdaptiveCpp again shows the best performance compared to DPC++, achieving a median throughput of 1815 ligands/s and 1832 ligands/s with the accessors and USM backend respectively. On the other hand, DPC++ gets 1579.2 ligands/s with the accessors backend and 1619.4 ligands/s with the USM backend. Contrary to LiGen Latency, this version achieves better performance compared to the corresponding one on NVIDIA hardware for each backend and compiler, mostly due to the increased number of compute units that allow more ligands to be computed concurrently.

6.4.1. Register optimization

The portability results obtained from the LiGen Batch and LiGen Latency applications exhibit varying degrees of performance portability when transitioning from CUDA to SYCL. Notably, LiGen Batch demonstrates poor portability, achieving only 53% and 46% of the native CUDA implementation performance.

Fig. 5 shows the runtime for the *dock* kernel, which takes up to 95% of the entire execution time of LiGen Batch. Dataset *d_31_20* exhibits the worst performance portability, achieving only 46% of the native implementation's performance with DPC++. Despite the SYCL



(a) LiGen Batch throughput with a fixed number of atoms and variable fragments number

(b) LiGen Batch throughput with a fixed number of fragments and variable atoms number

Fig. 3. LiGen Batch throughput on NVIDIA and AMD hardware.

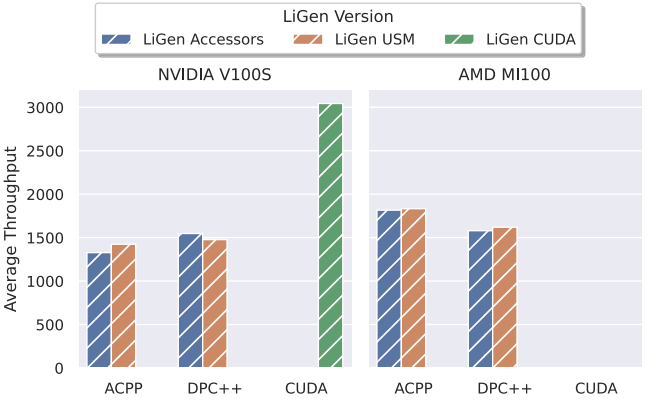


Fig. 4. LiGen Batch average throughput across all datasets.

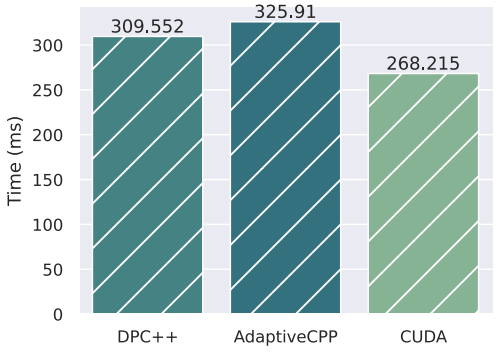


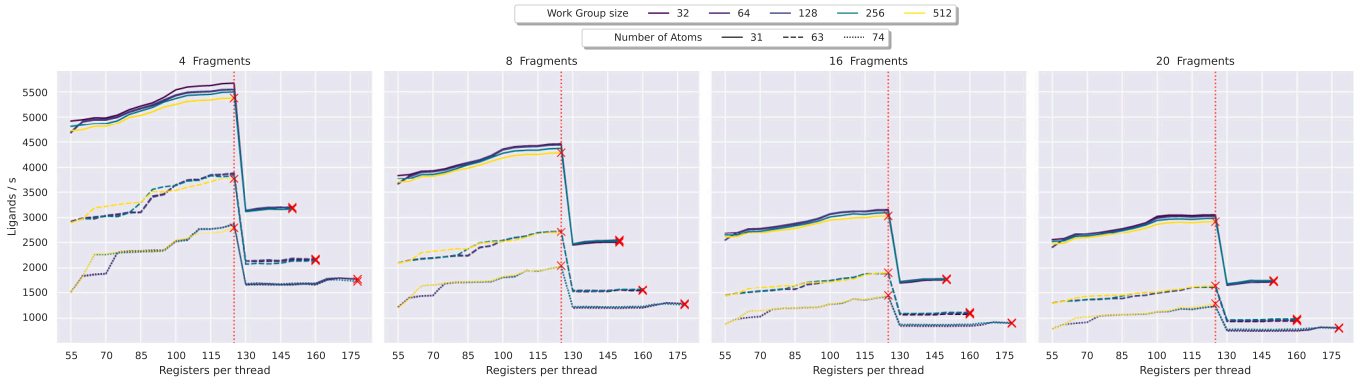
Fig. 5. Execution time of a single *dock* kernel run on a ligand with 31 atoms and 20 fragments, using SYCL accessors backend on the NVIDIA V100S.

implementations being up to 28% slower than the native one, these disparities in runtime do not fully account for the substantial gap in performance portability. As discussed in Section 5, register pressure imposes a hard limit on the achievable occupancy of modern GPUs. When considering the same dataset, the *dock* kernel in CUDA allocates 101 registers per thread. In contrast, the SYCL Accessors allocates 175 registers while SYCL USM uses 161 registers using DPC++. The Tesla V100S GPU comprises 80 Shared Multiprocessors, each equipped with 65 536 32-bit registers. For the purpose of this analysis, we will not consider shared memory capacity, as it does not affect the results.

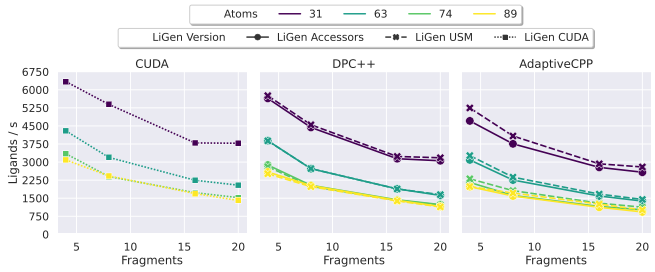
For LiGen Batch, the default work group size is set to 128 threads, with each sub-group computing 1 ligand (4 per work group). Based on these parameters, the GPU can concurrently compute up to $\lfloor \frac{65536}{128 \times 101} \rfloor \times 4 \times 80 = 4 \times 4 \times 80 = 1280$ ligands. However, the SYCL USM implementation with DPC++, being the least register demanding among all SYCL compilers, can only compute $\lfloor \frac{65536}{128 \times 161} \rfloor \times 4 \times 80 = 3 \times 4 \times 80 = 960$ ligands concurrently, resulting in 25% fewer ligands per batch. This discrepancy in occupancy severely impacts SYCL performance, as the GPU remains underutilized in comparison to the native CUDA implementation. Fortunately, NVIDIA GPUs allow developers to manually set an upper bound on the number of registers a kernel can allocate through the PTX assembler *ptxas*. This flexibility enables developers to make trade-offs between kernel performance and overall occupancy, potentially mitigating the performance degradation due to higher register allocation. Unfortunately, AMD GPUs do not allow to manually tuning register allocation, and thus we do not include this hardware in the analysis.

We run a hyperparameter optimization to find the best combination of register size and work group size for the LiGen Batch SYCL backend. We use the SYCL accessors backend together with the DPC++ compiler. We performed a Grid Search over 120 register sizes and 5 work group sizes, for a total number of 700 parameters. Since *dock* kernel is specialized over the ligand atoms number, we performed this analysis over all the kernel specializations, thus using the datasets with 31, 63, and 74 atoms together with all the fragment sizes. We exclude the datasets containing ligands with 89 atoms as that kernel specialization is covered by the dataset with 74 atoms.

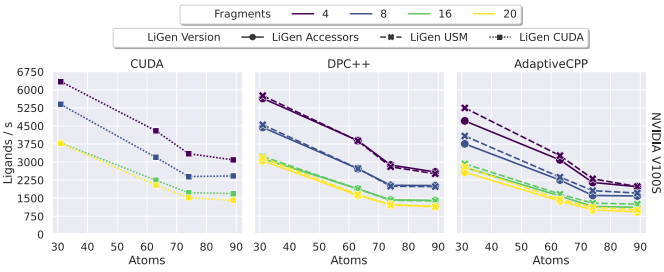
In Fig. 6(a) we show the results of our optimization campaign. For small register size, the kernel is significantly slower compared to the default configuration (*e.g.* 27% on *d_31_4* with 55 registers), but more ligands can be computed concurrently (*e.g.* 2560 ligands instead of 960 on *d_31_4* with 55 registers). Increasing the register size speeds up kernel time but reduces achieved occupancy. For all the datasets, the best register size is 125 registers per thread. This size ensures the best trade-off between kernel time and occupancy, with 1280 ligands computed concurrently. When computing ligands with few numbers of atoms, smaller work group sizes tend to perform slightly better than bigger sizes, exhibiting a maximum 5% speedup between 32 and 512 threads on dataset *d_31_4*. This advantage arises from the fact that each sub-group computes a single ligand, making all sub-group



(a) *dock* kernel throughput varying the number of registers per thread on an NVIDIA Tesla V100S.
The X points to the original SYCL register size for that atom size



(b) LiGen Batch throughput with a fixed number of atoms
and variable fragments number



(c) LiGen Batch throughput with a fixed number of fragments
and variable atoms number

Fig. 6. LiGen Batch throughput on NVIDIA V100S using optimal register and work group sizes.

computations independent of each other. Utilizing small work group sizes enables the GPU scheduler to allocate ligands on the GPU at the smallest granularity, effectively avoiding any potential workload imbalances that may occur when multiple ligands are computed within the same work group. However, as the number of atoms or fragments increases, the performance differences among various work group sizes become less pronounced. Consequently, for ligands with greater complexity, the choice of work group size becomes less critical, and the performance tends to converge. In light of these findings, we conducted additional experiments using LiGen Batch on the Tesla V100S, selecting a configuration with a maximum register size of 125 and 32 threads per work group. This particular setup consistently demonstrated better performance across all datasets. Figs. 6(b), 6(c) displays the achieved results. Both DPC++ and AdaptiveCpp outperform the default LiGen Batch configuration, with a 1.66x and 1.71x speedup respectively. The USM backend with DPC++ achieves the best results, with a median throughput of 2581 ligands/s, reaching 83% of the native CUDA performance. On the other hand, AdaptiveCpp USM reaches a throughput of 2270 ligans/s, 74% of the native CUDA performance.

7. Performance portability evaluation

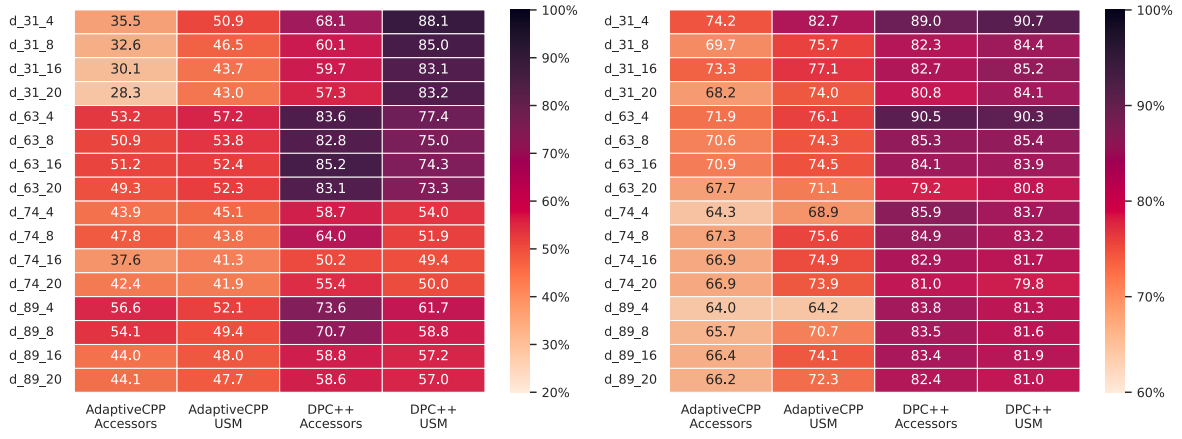
In this section, we will discuss LiGen performance portability across the selected hardware. We split our analysis into two phases: in the former, we evaluate the SYCL application efficiency on NVIDIA hardware. We first evaluate the SYCL application efficiency on NVIDIA hardware, and later evaluate the performance portability across different hardware.

7.1. SYCL vs. CUDA

To measure how close the SYCL implementation performances are to the native CUDA version, we use the *application efficiency*: this is defined as the ratio between the runtime of the current implementation and the runtime of the best native implementation on the same hardware. We show this analysis on a Tesla V100S, and we repeat that for each dataset and combination of SYCL compilers and backends.

LiGen Latency. Fig. 7(a) shows the achieved results. Generally, the application efficiency decreases by increasing the number of fragments: this happens because each fragment issues a new *optimize fragment* kernel invocation, each of them slightly slower than the CUDA native one. DPC++ USM performs on average better than the accessor's backend, with a median efficiency of 67.4% and 62% respectively. DPC++ USM outperforms the accessors backend on datasets with a low number of atoms, with a 56% improvement on *d_31_20*, while it performs slightly worse than the accessors backend on higher atoms number. AdaptiveCpp achieves a median efficiency of 44% and 47.8% on accessors and USM backend respectively, which is 18% and 19.6% less efficient than the corresponding backend using DPC++. This performance gap is mostly due to the AdaptiveCpp inlining issue that we addressed in the previous sections.

LiGen Batch. Fig. 7(b) shows the achieved results. As for LiGen Latency, the application efficiency decreases when increasing the number of fragments of the ligands. However, in LiGen Batch this does translate into additional kernel invocations, thus not introducing additional overhead. SYCL accessors and USM backend with DPC++ perform similarly, both achieving 83% median efficiency. AdaptiveCpp performs generally worse than DPC++ on both accessors and USM, achieving 67.5% and 74.2% of median efficiency respectively. As AdaptiveCpp compiles SYCL code using Clang CUDA without doing any major modification on source code, we also compared AdaptiveCpp performance against LiGen Batch built using Clang 17 CUDA backend. Because Clang CUDA allocates way more registers compared to NVCC, we applied the optimal register size found during the optimization phase. We found out that Clang CUDA achieves a median efficiency of 80% compared to NVCC, with AdaptiveCpp getting 92.5% and 84% median efficiency on USM and accessor respectively compared to Clang CUDA. On the other hand, DPC++ surprisingly outperforms Clang CUDA, probably due to a better PTX generation compared to vanilla Clang.



(a) LiGen Latency (b) LiGen Batch

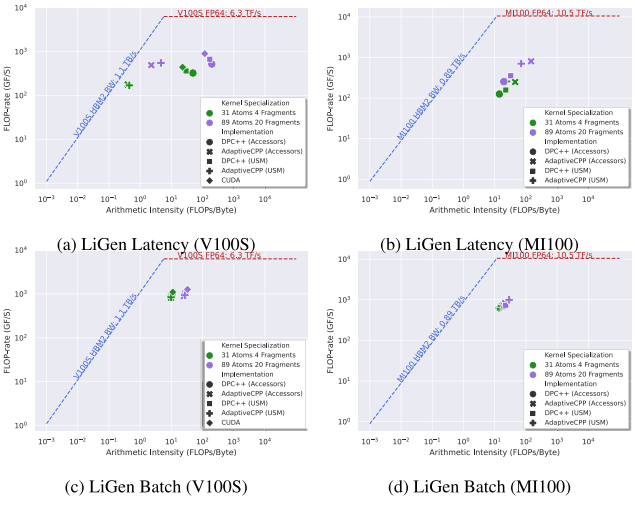
Fig. 7. LiGen application efficiency on NVIDIA Tesla V100S.

7.2. Kernel performance portability

While defining if an application is "performance portable" is highly subjective, and developers have diverging opinions on the topic, having formal metrics can help to quantify how an application performs across several hardware. In this section, we try to quantify how well the SYCL implementations perform on several hardware. We employ the Performance Portability Metric (Φ) [36] to define the degree of performance portability shown in Eq. (3). This metric is defined as the harmonic mean of the performance efficiency over the available platforms. However, calculating the precise performance efficiency can be challenging, as it requires either a deep understanding of the application performance bottleneck or an optimized native implementation for each platform. In LiGen, we have a native implementation targeting NVIDIA GPUs but we lack a direct comparison on the AMD platform. Furthermore, performing an in-depth bottleneck analysis for each combination of SYCL backend and compilers on both NVIDIA and AMD GPUs for both LiGen Latency and Batch would be extremely time-consuming.

In a context where we do not have access to a native implementation for each hardware, to approximate architectural efficiency we employ the *roofline efficiency* [32,45] analysis: it is defined as the ratio between the measured application performance (FL_k) and the target device peak performance (P_k). To collect the application performance data, we used specialized native profiling tools. For NVIDIA GPUs we employed NVIDIA Nsight [46], and for AMD MI100 the ROCm profiler [47]. While NVIDIA Nsight provides all the necessary information out-of-the-box, this was not the case for AMD as the MI100 does not provide a hardware flop counters. However, because LiGen Latency and LiGen Batch SYCL implementation does not have any device-dependent branch, the source code for both AMD and NVIDIA platforms is the same, thus we expect the number of floating point operations to be similar for both hardware. To estimate the kernel flop rate we employ different techniques for the two applications. We repeat our analysis on dataset *d_31_4*, which is referred to as *Light*, and on dataset *d_89_20*, referred to as *Heavy*.

Because the roofline efficiency can in some cases underestimate the application performance portability [32], we defined an ad hoc efficiency metric called *native efficiency* e_i'': it is calculated as the ratio between the target application roofline efficiency over the best native implementation roofline efficiency. This efficiency metric corresponds to calculating the Φ metric by moving the roofline top from the device peak efficiency to the native application peak efficiency, expressing portability in relation to the original native implementation. The ratio behind this efficiency metric is straightforward: because the SYCL implementations were defined starting from a native highly-optimized



(a) LiGen Latency (V100S) (b) LiGen Latency (MI100)

(c) LiGen Batch (V100S) (d) LiGen Batch (MI100)

Fig. 8. LiGen roofline models.

CUDA implementation, and the V100S and MI100 GPUs have similar hardware features, we expect the SYCL implementation to achieve a lower architectural efficiency compared to the native implementation. The novel efficiency metric uses a concept similar to application efficiency: we use the distance from the roofline top as an estimation of the application performance and then calculate the ratio between the native application efficiency and the ported one. Therefore, little variation is expected in the Φ values between the two metrics. The resulting formula is shown in Eq. (4):

$$\Phi''(a,p,H) = \begin{cases} \dfrac{|H|}{\sum_{i\in H}\dfrac{1}{e_i''(a,p)}} \\ 0 \end{cases} = \begin{cases} \dfrac{|H|}{\sum_{i\in H}\dfrac{e_b'(a_b,p)}{e_i'(a,p)}}, & \forall i \in H \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

where b is the hardware on which we have the best native implementation, a_b is the native application on hardware b, and $e_b'(a_b,p)$ is the roofline efficiency of the best native implementation.

7.2.1. LiGen Latency

We focus our analysis on the optimize_kernel_1, which takes ~90% of the entire application time. Table 4 and Fig. 8 show the roofline performance data.

Table 4
LiGen Latency roofline-based performance data for optimize_kernel_1.

Dataset	Hardware	LiGen backend	AI	FR (GFlop/s)	Bound	P (TFlop/s)	e'
Light	NVIDIA V100S	CUDA	22.83	443	Compute	6.3	8%
		DPC++ Accessors	48.96	325	Compute	6.3	5%
		AdaptiveCpp Accessors	0.4	178	Memory	0.440	39%
		DPC++ USM	29.6	362	Compute	6.3	5.7%
		AdaptiveCpp USM	0.45	170	Memory	0.504	33%
	AMD MI100 (Estimated)	DPC++ Accessors	14.15	126	Compute	10.5	1.2%
		AdaptiveCpp Accessors	45.22	249	Compute	10.5	2.3%
		DPC++ USM	22.57	158	Compute	10.5	1.5%
		AdaptiveCpp USM	23.36	257	Compute	10.5	2.4%
Heavy	NVIDIA V100S	CUDA	117	897	Compute	6.3	14.2%
		DPC++ Accessors	197	520	Compute	6.3	8.2%
		AdaptiveCpp Accessors	2.28	492	Memory	2.58	19%
		DPC++ USM	169	668	Compute	6.3	10.6%
		AdaptiveCpp USM	4.63	552	Memory	5.24	10.5%
	AMD MI100 (Estimated)	DPC++ Accessors	19.53	254	Compute	10.5	2.4%
		AdaptiveCpp Accessors	146	805	Compute	10.5	7.6%
		DPC++ USM	32.45	357	Compute	10.5	3.4%
		AdaptiveCpp USM	70.5	705	Compute	10.5	6.7%

Table 5
LiGen Batch roofline-based performance data for dock kernel.

Dataset	Hardware	LiGen backend	AI_k	FR (GFlop/s)	Bound	P (TFlop/s)	e'
Light	NVIDIA V100S	CUDA	11.1	1100	Register	6.3	17.3%
		DPC++ Accessors	10.44	961	Register	6.3	15.2%
		AdaptiveCpp Accessors	9.69	814	Register	6.3	12.9%
		DPC++ USM	10.43	960	Register	6.3	15.2%
		AdaptiveCpp USM	9.88	840	Register	6.3	13.3%
	AMD MI100 (Estimated)	DPC++ Accessors	14	632	Register	10.5	6%
		AdaptiveCpp Accessors	17.2	700	Register	10.5	6.6%
		DPC++ USM	13.2	626	Register	10.5	6%
		AdaptiveCpp USM	16	705	Register	10.5	6.7%
Heavy	NVIDIA V100S	CUDA	33.04	1273	Register	6.3	20%
		DPC++ Accessors	29	1098	Register	6.3	17.4%
		AdaptiveCpp Accessors	25.4	890	Register	6.3	14.1%
		DPC++ USM	29.7	1100	Register	6.3	17.4%
		AdaptiveCpp USM	26.7	936	Register	6.3	14.8%
	AMD MI100 (Estimated)	DPC++ Accessors	19.6	727	Register	10.5	7%
		AdaptiveCpp Accessors	21.7	826	Register	10.5	7.8%
		DPC++ USM	22.2	733	Register	10.5	6.9%
		AdaptiveCpp USM	28.9	1014	Register	10.5	10%

Table 6
Roofline efficiency and performance portability metrics for LiGen Latency's optimize_kernel_1 and LiGen Batch's dock kernel.

Application	Dataset	NVIDIA V100S	AMD MI100	Φ'	Φ''
LiGen Latency	Light	5.7%	2.4%	3.3%	41%
	Heavy	10.6%	7.6%	8.8%	61.9%
LiGen Batch	Light	15.2%	6.7%	9.3%	53.7%
	Heavy	17.4%	10%	12.7%	63.5%

NVIDIA V100S. The CUDA and SYCL implementations are mostly compute-bound, except for SYCL backends with the AdaptiveCpp compiler. Because of the kernel inlining issue with Clang PTX backend depicted in the previous sections, AdaptiveCpp accessor performs 285% more L1 cache load/store and achieves 52% less L1 cache hit rate, resulting in a 6655% higher memory throughput compared to DPC++ accessors when computing the *Light* dataset, thus drastically reducing the kernel Arithmetic Intensity and the device peak performance. Similar behavior can be observed with other SYCL backends and datasets. Both CUDA and SYCL implementations achieve relatively low levels of roofline efficiency on both datasets and hardware: this happens because a single ligand does not express a sufficiently high level of parallelism to efficiently fill the GPU resources. The only exception is AdaptiveCpp on NVIDIA hardware, which has lower peak performance and thus exhibits higher roofline efficiency as the roofline peak is lower than the peak device floating point ratio. For this reason, we ignore AdaptiveCpp in this analysis. DPC++ USM shows the best roofline efficiency between the SYCL implementations, with 5.7% and 10.6% on *Light* and *Heavy* datasets respectively, achieving 71% and 74% of the native CUDA implementation efficiency.

AMD MI100. To estimate the flop rate we follow the methodology depicted in [32] by multiplying the flop rate of the SYCL implementation on the V100S with the ratio of the V100S over the MI100 SYCL application runtime. The complete equation is shown in Eq. (5):

$$FR_{k\_mi100} = FR_{k\_v100s} * \frac{kernel\_time_{v100s}}{kernel\_time_{mi100}} \tag{5}$$

where FR_{k_v100s} is the flop rate of the corresponding SYCL implementation k on NVIDIA Tesla V100S. Both SYCL backends expose very low efficiency for the same reason we analyzed before for the V100S, as a single ligand is way too small to fill the GPU resources. We believe that the relatively lower performance on AMD GPUs is due to the highly experimental backend support of both DPC++ and AdaptiveCpp, which will improve as these compilation toolchains mature.

7.2.2. LiGen Batch

We focus our analysis on the *dock* kernel, which takes ~95% of the overall application time. We show the roofline performance data in Table 5 and Fig. 8.

NVIDIA V100S. Both CUDA and SYCL implementations are register-bound, severely limiting the number of ligands that can be concurrently computed on the GPU. The CUDA native implementation reaches

17.3% and 20% roofline efficiency on the *Light* and *Heavy* datasets respectively. The relatively low roofline efficiency is justified by the fact that LiGen, in addition to floating-point operations makes heavy usage of integer operations which are not captured by the roofline model, thus increasing the gap between the application and the roofline peak. SYCL with DPC++ is the closest to CUDA native, with both backends achieving similar efficiency, resulting in 15.2% and 17.4% roofline efficiency on the *Light* and *Heavy* dataset. On the other hand, AdaptiveCpp ∼13.2% and ∼14.5% efficiency on the two datasets, with the USM backend performing slightly better than the accessors one. Overall, DPC++ achieves the best roofline efficiency among the SYCL compilers, with 87% of native CUDA implementation efficiency on both datasets.

AMD MI100. The flop-rate estimation is less straightforward than in LiGen Latency. LiGen Batch automatically selects the best ligands batch size to fill the GPU resources: for the NVIDIA Tesla V100S is 1280 ligands, while on AMD MI100 is 1920 ligands. As each ligand is computed by a sub-group, the SYCL implementations spawn $1280 * 32 = 4096$ threads on the V100S and $1920 * 64 = 122880$ threads on the MI100, 4x more. We now try to quantify the amount of work computed by each sub-group. Each thread is assigned one or more atoms from the current ligand, depending if the atom size is bigger than the sub-group size (e.g. if the ligand has 50 atoms and the device's sub-group size is 32, thread 0 will have the atom 0 and 32, thread 1 will compute atom 1 and 33, etc.), and computes them sequentially. If the sub-group size is less than the number of atoms, $sub - group\_size - num\_atoms$ threads perform no useful work. Thus, the amount of work performed within a sub-group is proportional to the number of atoms in the ligand rather than the number of threads. As both testing datasets are uniform, i.e. all ligands have the same properties, we can assume that the SYCL implementations on AMD perform $\frac{1920}{1280} = 1.5$x more floating point operations per batch than the corresponding NVIDIA versions. Eq. (6) shows the complete formula:

$$FR_{k\_mi100} = \frac{FL_{k\_v100s} * \frac{batch\_size_{mi100}}{batch\_size_{v100s}}}{kernel\_time_{mi100}} \tag{6}$$

where FL_{k_v100s} is the number of floating point operations on the corresponding SYCL implementation k on NVIDIA Tesla V100S. As for the Tesla V100S, both SYCL backends are register-bound. AdaptiveCpp performs the best, achieving a maximum of 6% and 10% roofline efficiency on the *Light* and *Heavy* datasets respectively. The very low efficiency achieved on the first dataset derives from the low ligand number of atoms, which results in ∼50% unused threads per sub-group while increasing the number of atoms boosts roofline efficiency up to 49% on the Heavy dataset. On the other hand. DPC++ performs poorly, achieving 6% and ∼7.4% application efficiency on the two datasets. It is worth nothing that we were not able to tweak register usage as we did for NVIDIA hardware, thus such results are greatly influenced by Clang register allocation policies.

In Table 6 we show the performance portability values for both LiGen Latency and Batch. We show the results using both the roofline efficiency (Φ') and native efficiency (Φ''). We take the roofline efficiency of the best combination of SYCL backend and compilers for each platform and use it as an approximation of the platform's architectural efficiency. We do not take into account the native CUDA implementation as the analysis focuses on the SYCL kernel portability. Both LiGen versions show a relatively low level of Φ': this is justified by the low roofline efficiency reached by the original CUDA implementation on which the SYCL implementations are based , which achieves 8% and 14.2% roofline efficiency on the *Light* and *Heavy* dataset respectively in LiGen Latency, while on LiGen Batch it stops at 17.3% and 20% with the same datasets.. On the other hand, they achieve good Φ'', reaching 61.9% and 63.5% on the *Heavy* dataset. This means that both LiGen implementations get an efficiency close to the original native implementation.

8. Lesson learned

The SYCL programming language has faced a steady evolution process in the last few years. The last standard major release, SYCL 2020, added several new features to enable fast and efficient portability across a broad range of hardware. However, they lack validation of their applicability in a real-world scenario. Moreover, different compilers could expose very different performance and issues with the same feature. In this section, we will provide an overview of the principal lessons learned during the migration phase and how they impacted the porting development.

Memory access models influence performance. The Buffer-Accessors memory model allows to building of applications with fine-grained control over memory transfers without worrying about synchronization issues between kernels. Moreover, the RAII semantic frees the programmer from handling memory lifetime manually. However, it severely impacts over kernel's register pressure which can cause performance penalties on certain kinds of applications, while USM does not introduce any significant overhead. From our analysis, the buffer-accessors memory access model can impact the performance of register-bound applications due to its heavy register requirements, and manual register tweaking could be necessary to achieve good performance. Moreover, while DPC++ seems to generate code on par with USM when using buffer-accessors, AdaptiveCpp is still not able to optimize away all the abstractions from the memory access model, with generally worse performance compared to USM. However, compilers are actively evolving and such differences are going to shrink in the next future. The user should be careful in choosing the right memory model, considering also the target hardware.

Group algorithms reduce code complexity. SYCL 2020 introduces several work group and sub-group algorithms to write performance portable kernels without worrying about device-specific implementation details. LiGen heavily relies on custom reductions and warp-level primitives, written ad hoc for NVIDIA GPUs. We managed to easily map those functions to the most appropriate SYCL algorithms, with similar performance to the manually-tuned CUDA implementation and portable across different hardware. Moreover, the resulting code exposes a significantly lower code complexity, saving up to 22% lines of code on kernels with intra-group and across-group reductions.

SYCL application should be compiler-portable to achieve performance portability. As building reliable and efficient compiler toolchains takes time, SYCL compilers are still young and can expose significant performance discrepancies depending on the chosen SYCL feature or target device. DPC++ has shown remarkable performance on NVIDIA hardware, with a better code generation phase compared to AdaptiveCpp. On the other hand, the latter outperformed DPC++ on several benchmarks when targeting AMD hardware. Overall, to achieve the best performance portability is not only necessary to have code written in a device-independent language, but also choosing the most appropriate compiler for the target hardware is required. For those reasons, the SYCL application should avoid compiler-dependent branches as much as possible to make the code compiler-portable.

9. Related work

Modern computing systems are becoming increasingly heterogeneous. In this context, it is necessary to evaluate both the performance achieved by the application on each hardware and the number of devices on which the application can run (portability). However, the meaning of performance portability is still controversial and there is a lack of consensus on how to quantify performance portability [34]. Pennycook et al. define a new metric called Φ metric, that evaluates the performance portability of an application on a fixed problem size using the harmonic mean [36–38,48,49]. As Φ requirements are not readily addressable, several studies extended it by using the roofline

efficiency as an approximation for the architectural efficiency [45,45,50]: this allows to estimate the application performance portability without an in-depth analysis of the application bottleneck or an optimized native version for each hardware. However, if the target application is neither compute nor memory bound, the roofline efficiency can underestimate the Φ metric [32]. Differently, Marowka [51] proposes a metric based on the arithmetic mean rather than the harmonic mean. Daniel and Panetta [52] introduce a new metric that also considers how the performance portability of an application varies with the problem size. Over the past few years, various programming models have been defined to address the challenge of developing performance portable applications, such as Kokkos [53,54], OpenCL [55,56], RAJA [57,58], PACXX [59], and SYCL [60,61]. Even with support for heterogeneous programming models, there are many challenges to developing high-performance portable applications [62,63]. The increasing adoption of heterogeneous programming models to implement performance portable applications has led to the challenge of selecting the programming model that best fits the specific purpose. To facilitate this choice, several studies explore the performance portability of the most popular programming models across different architectures [64–66]. To speed up migrations from vendor-agnostic programming languages, several tools have been proposed for automatic translation from vendor-specific code to portable programming models [67–69] with promising results [70]. However, keeping the same performance while migrating the vendor-specific code to a heterogeneous programming model can be very challenging as vendor-specific features cannot be always directly mapped onto heterogeneous programming models, requiring additional manual tuning phases [71]. Solis-Vasquez et al. [72] leverage the DPC++ compatibility tool to migrate the AutoDock molecular docking application from CUDA to SYCL, showing that SYCL can reach native comparable performance with ad hoc tuning. However, it lacks an in-depth analysis of several SYCL 2020 new features, together with custom performance portability metrics. Moreover, their analysis is limited to the DPC++ compiler only.

10. Conclusions

In this work, we presented a performance-portable implementation of LiGen, a drug-discovery platform written in SYCL 2020. We ported the original CUDA implementations of two LiGen versions, specifically LiGen Latency and LiGen Batch, exploiting the most recent SYCL 2020 features, such as group algorithms and sub-groups, to efficiently map the computation to a broad range of GPU architectures. We measured how SYCL's memory access models, i.e. Unified Shared Memory and Buffer-Accessors, impact performance by evaluating both models on our LiGen implementations. Furthermore, we show how advanced portability issues such as work group size tuning and register pressure can severely limit SYCL performance. The results presented show that SYCL, with low porting effort and without any device-dependent optimization, can successfully run on both NVIDIA and AMD GPUs. On NVIDIA hardware our SYCL implementations, with minimal tuning can achieve comparable performance, with DPC++ outperforming AdaptiveCPP in almost every experiment. We evaluated the performance portability of both LiGen latency and batch using the Φ metric and an ad hoc efficiency metric called *native efficiency*. Both LiGen implementations show good portability, achieving good performance portability using the *native efficiency* metric, and showing performance close to the best native CUDA implementation. Finally, our results show that the LiGen Batch version achieves the best results on all performance and performance portability metrics compared to the LiGen Latency version.

CRediT authorship contribution statement

Luigi Crisci: Conceptualization, Investigation, Software, Writing – original draft. **Lorenzo Carpentieri:** Data curation, Software, Writing – original draft. **Biagio Cosenza:** Supervision, Writing – review & editing. **Gianmarco Accordi:** Investigation, Software, Validation, Writing – review & editing. **Davide Gadioli:** Resources, Software, Validation, Writing – review & editing. **Emanuele Vitali:** Resources, Software. **Gianluca Palermo:** Supervision, Writing – review & editing. **Andrea Rosario Beccari:** Project administration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The authors do not have permission to share data.

Acknowledgments

LiGen has been funded by the LIGATE project, receiving funding from the European High-Performance Computing Joint Undertaking (JU) [grant number 956137]. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Italy, Sweden, Austria, Czech Republic, Switzerland.

References

[1] H. Matter, C. Sotriffer, Applications and success stories in virtual screening, in: Virtual Screening, John Wiley & Sons, Ltd, 2011, pp. 319–358, http://dx.doi.org/10.1002/9783527633326.ch12, (Chapter 12).
[2] M. Allegretti, M.C. Cesta, M. Zippoli, A. Beccari, C. Talarico, F. Mantelli, E.M. Bucci, L. Scorzolini, E. Nicastri, Repurposing the estrogen receptor modulator raloxifene to treat SARS-CoV-2 infection, Cell Death Differ. 29 (1) (2022) 156–166, http://dx.doi.org/10.1038/s41418-021-00844-6.
[3] Exscalate4Cov official website, 2023, https://www.exscalate4cov.eu.
[4] J. Brase, N. Campbell, B. Helland, T. Hoang, M. Parashar, M. Rosenfield, J. Sexton, J. Towns, LLNL-JRNL-835693 - The Covid-19 High-Performance Computing Consortium, Tech. Rep., Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States, 2022, http://dx.doi.org/10.1109/MCSE.2022.3145608.
[5] Top500 chart, 2023, https://www.top500.org/.
[6] D. Gadioli, E. Vitali, F. Ficarelli, C. Latini, C. Manelfi, C. Talarico, C. Silvano, C. Cavazzoni, G. Palermo, A.R. Beccari, EXSCALATE: An extreme-scale virtual screening platform for drug discovery targeting polypharmacology to fight SARS-CoV-2, IEEE Trans. Emerg. Top. Comput. 11 (1) (2023) 170–181, http://dx.doi.org/10.1109/TETC.2022.3187134.
[7] E. Vitali, F. Ficarelli, M. Bisson, D. Gadioli, M. Fatica, A.R. Beccari, G. Palermo, GPU-optimized approaches to molecular docking-based virtual screening in drug discovery: A comparative analysis, 2022, arXiv:2209.05069.
[8] N.A. Murugan, A. Podobas, D. Gadioli, E. Vitali, G. Palermo, S. Markidis, A review on parallel virtual screening softwares for high-performance computers, Pharmaceuticals 15 (1) (2022) 63, http://dx.doi.org/10.3390/ph15010063.
[9] J. Biesiada, A. Porollo, P. Velayutham, M. Kouril, J. Meller, Survey of public domain software for docking simulations and virtual screening, Hum. Genom. 5 (5) (2011) 497–505, http://dx.doi.org/10.1186/1479-7364-5-5-497.
[10] E. Yuriev, J. Holien, P.A. Ramsland, Improvements, trends, and new ideas in molecular docking: 2012–2013 in review, J. Mol. Recognit. 28 (10) (2015) 581–604, http://dx.doi.org/10.1002/jmr.2471.
[11] N.S. Pagadala, K. Syed, J. Tuszynski, Software for molecular docking: a review, Biophys. Rev. 9 (2) (2017) 91–102, http://dx.doi.org/10.1007/s12551-016-0247-1.
[12] M. Fan, J. Wang, H. Jiang, Y. Feng, M. Mahdavi, K. Madduri, M.T. Kandemir, N.V. Dokholyan, GPU-accelerated flexible molecular docking, J. Phys. Chem. B 125 (4) (2021) 1049–1060, http://dx.doi.org/10.1021/acs.jpcb.0c09051, PMID: 33497567.
[13] B. Sukhwani, M.C. Herbordt, GPU acceleration of a production molecular docking code, in: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, 2009, pp. 19–27, http://dx.doi.org/10.1145/1513895.1513898.
[14] O. Korb, T. Stützle, T.E. Exner, Accelerating molecular docking calculations using graphics processing units, J. Chem. Inf. Model. 51 (4) (2011) 865–876, http://dx.doi.org/10.1021/ci100459b.
[15] Y. Fang, Y. Ding, W.P. Feinstein, D.M. Koppelman, J. Moreno, M. Jarrell, J. Ramanujam, M. Brylinski, GeauxDock: Accelerating structure-based virtual screening with heterogeneous computing, PLoS One 11 (7) (2016) e0158898, http://dx.doi.org/10.1371/journal.pone.0158898.

[16] I. Sánchez-Linares, H. Pérez-Sánchez, J.M. Cecilia, J.M. García, High-throughput parallel blind virtual screening using BINDSURF, BMC Bioinformatics 13 (SUPPL 14) (2012) http://dx.doi.org/10.1186/1471-2105-13-S14-S13.

[17] S. Tang, R. Chen, M. Lin, Q. Lin, Y. Zhu, J. Ding, H. Hu, M. Ling, J. Wu, Accelerating AutoDock vina with GPUs, Molecules 27 (9) (2022) 3041, http://dx.doi.org/10.3390/molecules27093041.

[18] G.M. Morris, R. Huey, W. Lindstrom, M.F. Sanner, R.K. Belew, D.S. Goodsell, A.J. Olson, AutoDock4 and AutoDockTools4: Automated docking with selective receptor flexibility, J. Comput. Chem. 30 (16) (2009) 2785–2791, http://dx.doi.org/10.1002/jcc.21256.

[19] S. LeGrand, A. Scheinberg, A.F. Tillack, M. Thavappiragasam, J.V. Vermaas, R. Agarwal, J. Larkin, D. Poole, D. Santos-Martins, L. Solis-Vasquez, A. Koch, S. Forli, O. Hernandez, J.C. Smith, A. Sedova, GPU-Accelerated drug discovery with docking on the summit supercomputer: Porting, optimization, and application to COVID-19 research, in: Proceedings of the 11th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics, Association for Computing Machinery, New York, NY, USA, 2020, http://dx.doi.org/10.1145/3388440.3412472.

[20] J. Glaser, J.V. Vermaas, D.M. Rogers, J. Larkin, S. LeGrand, S. Boehm, M.B. Baker, A. Scheinberg, A.F. Tillack, M. Thavappiragasam, A. Sedova, O. Hernandez, High-throughput virtual laboratory for drug discovery using massive datasets, Int. J. High Perform. Comput. Appl. 35 (5) (2021) 452–468, http://dx.doi.org/10.1177/10943420211001565.

[21] M. Thavappiragasam, A. Scheinberg, W. Elwasif, O. Hernandez, A. Sedova, Performance portability of molecular docking miniapp on leadership computing platforms, in: 2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), 2020, pp. 36–44, http://dx.doi.org/10.1109/P3HPC51967.2020.00009.

[22] S. Markidis, D. Gadioli, E. Vitali, G. Palermo, Understanding the I/O impact on the performance of high-throughput molecular docking, in: 2021 IEEE/ACM Sixth International Parallel Data Systems Workshop, PDSW, 2021, pp. 9–14, http://dx.doi.org/10.1109/PDSW54622.2021.00007.

[23] A. Alpay, V. Heuveline, SYCL beyond OpenCL: The architecture, current state and future direction of hipSYCL, in: Proceedings of the International Workshop on OpenCL, 2020, p. 1, http://dx.doi.org/10.1145/3388333.3388658.

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

[25] B. Ashbaugh, A. Bader, J. Brodman, J. Hammond, M. Kinsner, J. Pennycook, R. Schulz, J. Sewall, Data parallel C++: Enhancing SYCL through extensions for productivity and performance, in: Proceedings of the International Workshop on OpenCL, 2020, pp. 1–2, http://dx.doi.org/10.1145/3388333.3388653.

[26] A. Gozillon, R. Keryell, L.-Y. Yu, G. Harnisch, P. Keir, Trisycl for xilinx FPGA, in: Proceedings of the 2020 International Conference on High Performance Computing and Simulation, HPCS, IEEE, United States, 2020, The 2020 International Conference on High Performance Computing; Simulation, HPCS 2020 ; Conference date: 25-01-2021 Through 29-01-2021.

[27] Y. Ke, M. Agung, H. Takizawa, NeoSYCL: A SYCL implementation for SX-aurora TSUBASA, in: The International Conference on High Performance Computing in Asia-Pacific Region, in: HPC Asia 2021, Association for Computing Machinery, New York, NY, USA, 2021, pp. 50–57, http://dx.doi.org/10.1145/3432261.3432268.

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

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

[30] L. Carpentieri, B. Cosenza, Towards a SYCL API for approximate computing, in: Proceedings of the 2023 International Workshop on OpenCL, 2023, pp. 1–2, http://dx.doi.org/10.1145/3585341.3585374.

[31] S. Williams, A. Waterman, D. Patterson, Roofline: An insightful visual performance model for multicore architectures, Commun. ACM 52 (4) (2009) 65–76, http://dx.doi.org/10.1145/1498765.1498785.

[32] J. Kwack, J. Tramm, C. Bertoni, Y. Ghadar, B. Homerding, E. Rangel, C. Knight, S. Parker, Evaluation of performance portability of applications and mini-apps across AMD, Intel and NVIDIA GPUs, in: 2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC), 2021, pp. 45–56, http://dx.doi.org/10.1109/P3HPC54578.2021.00008.

[33] J. Kwack, T. Applencourt, C. Bertoni, Y. Ghadar, H. Zheng, C. Knight, S. Parker, Roofline-based performance efficiency of HPC benchmarks and applications on current generation of processor architectures, in: 2019 Cray User Group Meeting, Vol. 5, 2019.

[34] J. Neely, DOE Centers of Excellence Performance Portability Meeting, Tech. Rep., Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2016, URL https://asc.llnl.gov/doe-coe-mtg-2016.

[35] O. Aumage, P. Carpenter, S. Benkner, Task-based performance portability in HPC, 2021, URL https://www.etp4hpc.eu/news/273-task-based-performance-portability-in-hpc.html.

[36] S.J. Pennycook, J.D. Sewall, D.W. Jacobsen, T. Deakin, S. McIntosh-Smith, Navigating performance, portability, and productivity, Computing in Science & Engineering 23 (5) (2021) 28–38, http://dx.doi.org/10.1109/MCSE.2021.3097276.

[37] S.J. Pennycook, J.D. Sewall, Revisiting a metric for performance portability, in: 2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC), IEEE, 2021, pp. 1–9, http://dx.doi.org/10.1109/P3HPC54578.2021.00004.

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

[39] C. Beato, A.R. Beccari, C. Cavazzoni, S. Lorenzi, G. Costantino, Use of experimental design to optimize docking performance: The case of LiGenDock, the docking module of ligen, a new de novo design program, J. Chem. Inf. Model. 53 (6) (2013) 1503–1517, http://dx.doi.org/10.1021/ci400079k, PMID: 23590204.

[40] A.R. Beccari, C. Cavazzoni, C. Beato, G. Costantino, LiGen: A high performance workflow for chemistry driven de novo design, J. Chem. Inf. Model. 53 (6) (2013) 1518–1527, http://dx.doi.org/10.1021/ci400078g, PMID: 23617275.

[41] G. Accordi, E. Vitali, D. Gadioli, L. Crisci, B. Cosenza, M. Bisson, M. Fatica, A. Beccari, G. Palermo, Improving computation efficiency using input and architecture features for a virtual screening application, 2023, arXiv:2303.06150.

[42] V. Volkov, Better performance at lower occupancy, in: Proceedings of the GPU Technology Conference, GTC, Vol. 10, 2015.

[43] C. Yang, R. Gayatri, T. Kurth, P. Basu, Z. Ronaghi, A. Adetokunbo, B. Friesen, B. Cook, D. Doerfler, L. Oliker, J. Deslippe, S. Williams, An empirical roofline methodology for quantitatively assessing performance portability, in: 2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), 2018, pp. 14–23, http://dx.doi.org/10.1109/P3HPC.2018.00005.

[44] Generic addressing: PTX 8.2 specification, 2023, https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#generic-addressing.

[45] C. Bertoni, J. Kwack, T. Applencourt, Y. Ghadar, B. Homerding, C. Knight, B. Videau, H. Zheng, V. Morozov, S. Parker, Performance portability evaluation of OpenCL benchmarks across intel and NVIDIA platforms, in: 2020 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW, 2020, pp. 330–339, http://dx.doi.org/10.1109/IPDPSW50202.2020.00067.

[46] NVIDIA profiling tools, 2023, https://developer.nvidia.com/tools-overview.

[47] AMD ROCm profiler, 2023, https://docs.amd.com/en/docs-5.3.0/reference/compilers.html.

[48] S.J. Pennycook, J.D. Sewall, V.W. Lee, Implications of a metric for performance portability, Future Gener. Comput. Syst. 92 (2019) 947–958, http://dx.doi.org/10.1016/j.future.2017.08.007.

[49] J. Sewall, S.J. Pennycook, D. Jacobsen, T. Deakin, S. McIntosh-Smith, Interpreting and visualizing performance portability metrics, in: 2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), IEEE, 2020, pp. 14–24, http://dx.doi.org/10.1109/P3HPC51967.2020.00007.

[50] P. Grete, F.W. Glines, B.W. O'Shea, K-Athena: A performance portable structured grid finite volume magnetohydrodynamics code, IEEE Trans. Parallel Distrib. Syst. 32 (1) (2021) 85–97, http://dx.doi.org/10.1109/TPDS.2020.3010016.

[51] A. Marowka, Toward a better performance portability metric, in: 2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP, IEEE, 2021, pp. 181–184, http://dx.doi.org/10.1109/PDP52278.2021.00036.

[52] D.F. Daniel, J. Panetta, On applying performance portability metrics, in: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), IEEE, 2019, pp. 50–59, http://dx.doi.org/10.1109/P3HPC49587.2019.00010.

[53] H.C. Edwards, C.R. Trott, D. Sunderland, Kokkos: Enabling manycore performance portability through polymorphic memory access patterns, J. Parallel Distrib. Comput. 74 (12) (2014) 3202–3216, http://dx.doi.org/10.1016/j.jpdc.2014.07.003.

[54] H.C. Edwards, C.R. Trott, Kokkos: Enabling performance portability across manycore architectures, in: 2013 Extreme Scaling Workshop (Xsw 2013), 2013, pp. 18–24, http://dx.doi.org/10.1109/XSW.2013.7.

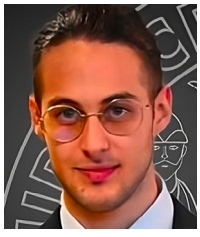
[55] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, H. Kobayashi, Evaluating performance and portability of OpenCL programs, in: The Fifth International Workshop on Automatic Performance Tuning, Vol. 66, 2010, p. 1.

[56] S.J. Pennycook, S.D. Hammond, S.A. Wright, J. Herdman, I. Miller, S.A. Jarvis, An investigation of the performance portability of OpenCL, J. Parallel Distrib. Comput. 73 (11) (2013) 1439–1450, http://dx.doi.org/10.1016/j.jpdc.2012.07.005.

[57] D. Beckingsale, R. Hornung, T. Scogland, A. Vargas, Performance portable C++ programming with RAJA, in: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, PPoPP '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 455–456, http://dx.doi.org/10.1145/3293883.3302577.

[58] 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, http://dx.doi.org/10.1109/P3HPC49587.2019.00012.
[59] V. Kucher, J. Hunloh, S. Gorlatch, Performance portability and unified profiling for finite element methods on parallel systems, Adv. Sci. Technol. Eng. Syst. J. 5 (1) (2020) 119–127, http://dx.doi.org/10.25046/aj050116.
[60] B. Johnston, J.S. Vetter, J. Milthorpe, Evaluating the performance and portability of contemporary SYCL implementations, in: 2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), IEEE, 2020, pp. 45–56, http://dx.doi.org/10.1109/P3HPC51967.2020.00010.
[61] 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, http://dx.doi.org/10.1145/3388333.3388643.
[62] A. Sedova, J.D. Eblen, R. Budiardja, A. Tharrington, J.C. Smith, High-performance molecular dynamics simulation for biological and materials sciences: Challenges of performance portability, in: 2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), 2018, pp. 1–13, http://dx.doi.org/10.1109/P3HPC.2018.00004.
[63] S.L. Harrell, J. Kitson, R. Bird, S.J. Pennycook, J. Sewall, D. Jacobsen, D.N. Asanza, A. Hsu, H.C. Carrillo, H. Kim, et al., Effective performance portability, in: 2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), IEEE, 2018, pp. 24–36.
[64] T. Deakin, S. McIntosh-Smith, J. Price, A. Poenaru, P. Atkinson, C. Popa, J. Salmon, Performance portability across diverse computer architectures, in: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), IEEE, 2019, pp. 1–13.
[65] W.-C. Lin, S. McIntosh-Smith, Comparing julia to performance portable parallel programming models for HPC, in: 2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, PMBS, 2021, pp. 94–105, http://dx.doi.org/10.1109/PMBS54543.2021.00016.
[66] M. Martineau, S. McIntosh-Smith, W. Gaudin, Assessing the performance portability of modern parallel programming models using TeaLeaf, Concurr. Comput.: Pract. Exper. 29 (15) (2017) e4117.
[67] A. Huang, Syclomatic compatibility library: Making migration to SYCL easier, in: Proceedings of the 2023 International Workshop on OpenCL, IWOCL '23, Association for Computing Machinery, New York, NY, USA, 2023, http://dx.doi.org/10.1145/3585341.3585349.
[68] M. Harvey, G. De Fabritiis, Swan: A tool for porting CUDA programs to OpenCL, Comput. Phys. Comm. 182 (4) (2011) 1093–1099, http://dx.doi.org/10.1016/j.cpc.2010.12.052.
[69] G. Martinez, M. Gardner, W.-c. Feng, CU2CL: A CUDA-to-OpenCL translator for multi- and many-core architectures, in: 2011 IEEE 17th International Conference on Parallel and Distributed Systems, 2011, pp. 300–307, http://dx.doi.org/10.1109/ICPADS.2011.48.
[70] 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, http://dx.doi.org/10.1016/j.jpdc.2022.03.017.
[71] Z. Jin, J.S. Vetter, Performance portability study of epistasis detection using SYCL on NVIDIA GPU, BCB '22, Association for Computing Machinery, New York, NY, USA, 2022, http://dx.doi.org/10.1145/3535508.3545591.
[72] L. Solis-Vasquez, E. Mascarenhas, A. Koch, Experiences migrating CUDA to SYCL: A molecular docking case study, in: Proceedings of the 2023 International Workshop on OpenCL, IWOCL '23, Association for Computing Machinery, New York, NY, USA, 2023, http://dx.doi.org/10.1145/3585341.3585372.

Luigi Crisci got his M.Sc. degree in Computer Science from the University of Salerno. From 2019 to 2020 he held a research scholarship related to the *EuroHPC LIGATE* euroepan project. He is a Ph.D. student in the department of Computer Science at the University of Salerno, Italy, since 2020. His research interests focus on High-performance and parallel computing, with a focus on performance portability on heterogeneous systems. interest.



Lorenzo Carpentieri received the master's degrees from the University of Salerno, Italy in 2022. He is now Ph.D. student in the Department of Computer Science at University of Salerno, Italy, under the supervision of Prof. Biagio Cosenza. His research interests include high-performance computing, compiler technology, and programming models having a particular interest in approximate computing techniques

Biagio Cosenza is a tenure-track Assistant Professor in the Department of Computer Science at the University of Salerno, Italy. From 2015 to 2019, he was Senior Research at the TU Berlin, Germany, where he was Principal Investigator for the DFG project Celerity. From 2011 to 2015, he was a Postdoctoral researcher at the University of Innsbruck, Austria, where he contributed to the Insieme Compiler and the DK-Plus multidisciplinary platform. Cosenza's research focuses on high performance computing and is currently funded by the EuroHPC Joint Undertaking (LIGATE project), the Italian Ministry of Research (LibreRT project), and several industrial projects..

Gianmarco Accordi received his Master of Science degree in Computer Engineering in 2022, while in the same year he started his Ph.D degree in Computer Engineering, from Politecnico di Milano (Italy). In the meanwhile, he is a Temporary Research associate at Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB) of Politecnico di Milano. His main research interests are in autonomic computing, approximate computing, and high-performance computing.

Davide Gadioli received his Master of Science degree in Computer Engineering in 2013, while in 2019 he received the Ph.D degree in Computer Engineering, from Politecnico di Milano (Italy).Currently, he is a postdoc at Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB) of Politecnico di Milano. In 2015, he was a Visiting Student at IBM Research (The Netherlands). His main research interests are in application autotuning, approximate computing, and high throughput molecular docking.

Emanuele Vitali graduated in 2015 from Politecnico di Milano (Italy) after completing his Master of Science in Computer Engineering, and in 2021 he received the Ph.D. degree from the same university. Currently, he is a postdoctoral researcher at CSC - IT Center for Science, Finland. In 2019, he has been Visiting Student at Dividiti (UK). His main research interests include GPGPU architectures and programming, application autotuning and high-throughput molecular docking.

Gianluca Palermo received his Master of Science degree in Electronic Engineering, in 2002, and the Ph.D. degree in Computer Engineering, in 2006, from Politecnico di Milano (Italy). He is currently a Full Professor at the Department of Electronics, Information, and Bioengineering (DEIB) at the same University. Previously, he was part of the Low-Power Design Group of AST - STMicroelectronics working on Network-on-Chip architectures, and a Research Assistant at the Advanced Learning and Research Institute (ALaRI) of the Universita' della Svizzera Italiana. His research interests include design methodologies and architectures for embedded and HPC systems, focusing on autotuning aspects and extreme-scale virtual screening. Since 2003, he published more than 150 scientific papers in peer-reviewed conferences and journals.

Andrea R. Beccari, Vice President EXSCALATE of Dompé farmaceutici SpA. Promoter of the open innovation project Italian Drug Discovery Network and co-founder of the Avicenna Alliance (Brussels) interest group. Promoter of two congress series: Computational Driven Drug Discovery and Italian Drug Discovery Summit. Active with the European Commission and Parliament in promoting the use of simulation in health. Coordinator of two European projects EXSCALATE4COV and LIGATE. Author of more than 50 publications and co-inventor in more than 15 patents.