# Optimizing the LiGen Drug Discovery Pipeline for Intel Max GPUs

Saleh Jamali Golzar, *University of Salerno*, Italy
Lorenzo Carpentieri, *University of Salerno*, Italy
Antonio De Caro, *University of Salerno*, Italy
Biagio Cosenza, *University of Salerno*, Italy
Davide Gadioli, *Politecnico di Milano*, Italy
Gianmarco Accordi, *Politecnico di Milano*, Italy
Gianluca Palermo, *Politecnico di Milano*, Italy
Federico Ficarelli, *CINECA*, Italy
Daniele Gregori, *E4 Computer Engineering*, Italy
Andrea R. Beccari, *Dompé Farmaceutici SpA*, Italy

***Abstract*—High-throughput virtual screening is a fundamental technique in modern drug discovery, enabling the identification of promising drug candidates by evaluating millions of ligand–protein interactions in silico. Achieving high performance and scalability in such workflows requires efficient exploitation of parallel architectures. LiGen is a high-performance virtual screening application designed to accelerate molecular docking and scoring computations. Initially implemented in CUDA to fully exploit NVIDIA GPUs, LiGen has been ported to SYCL to extend its support for heterogeneous architectures. In this work, we enhance the LiGen SYCL codebase through a performance-portable implementation to maximize the ligand throughput based on the new SYCL features introduced in oneAPI. Our optimized implementation leverages portable features that abstract the underlying hardware resources, enabling dynamic adaptation of the number of ligands processed per kernel to the characteristics of the target device. We further extended our implementation with architecture-specific optimizations for Intel GPUs, focusing on sub-group size tuning and efficient General Register File (GRF) utilization to maximize ligand throughput. Our experimental evaluation compares our optimized SYCL implementations against the manually-tuned CUDA and SYCL baselines. Results show that our version achieves up to 1.69× throughput compared to the SYCL baseline without requiring manual tuning, while the version with Intel-specific optimizations achieves a throughput of up to 2.42×.**

***Index Terms*—Drug Discovery, SYCL, Performance Portability, Programming Models.**

## I. Introduction

Drug discovery constitutes a fundamental process within the pharmaceutical industry, aiming to identify novel compounds with desirable properties. Computational approaches, such as virtual screening, mitigate the limitations of costly and time-intensive *in vitro* methods by enabling the evaluation of millions of small molecules (ligands) against target proteins. Recent works demonstrate how increasing the number of virtually screened ligands increases the probability of finding promising drug candidates [1]. The substantial computational demands of these methods underscore the critical role of High Performance Computing (HPC) as an integral component of contemporary drug discovery pipelines. The growing heterogeneity of HPC systems, together with the new vendors providing datacenter devices, increases the difficulty of developing pipelines that are both efficient and portable. As a result, vendor-locked applications are no longer practical or cost-effective. Proprietary programming models such as CUDA and HIP make this problem worse by limiting portability and adding development overhead. The SYCL programming model [2], standardized by the Khronos Group, offers a unified interface for heterogeneous programming in C++, facilitating both code and performance portability across diverse devices, architectures, and vendors. Nevertheless, achieving optimal performance across architectures often requires strategies that extend beyond mere portability. Different devices offer distinct hardware features, such as memory hierarchies, vector widths, and register configurations, which can greatly improve performance when properly optimized. SYCL provides this flexibility through backend interoperability mechanisms [3], enabling developers to access low-level hardware features. This allows performance-portable applications to effectively leverage architecture-specific features when advantageous. In this work, we focus on Intel datacenter GPUs, specifically the Intel Max Series, which introduces distinct architectural features such as configurable sub-group and register file. Additionally, different GPUs, even from the same vendor, typically require different tuning for low-level optimizations. Intel GPUs employ a hierarchical design comprising $X^e$ Stacks, Slices, and Cores, and support configurable SIMD widths via adjustable sub-group sizes (e.g., 16 or 32 work-items). Furthermore, they provide fine-grained control over the General Register File (GRF) configuration, a critical factor influencing register allocation and kernel occupancy. Maximizing performance through these mechanisms requires careful tuning of parameters such as sub-group size and GRF mode.

The extensibility of SYCL through extensions enables runtime querying of device properties and dynamic tuning of execution parameters. These extensions fall into two categories: device-portable features applicable across vendors, and oneAPI features tailored for Intel GPUs. This flexibility facilitates adaptive heuristics that promote performance portability while enabling architecture-specific optimizations.

This study optimizes the SYCL implementation of the LiGen drug discovery pipeline for Intel Max GPUs and compares the results against a manually tuned CUDA implementation. We propose a runtime heuristic that leverages SYCL and oneAPI capabilities to automatically adapt kernel configurations for performance portability, complemented by Intel-specific optimizations targeting sub-group size and GRF utilization. Our approach improves throughput by up to $2.42\times$ over the baseline SYCL implementation of LiGen, achieving performance comparable to manually-tuned CUDA and SYCL versions without the need for manual hardware-specific tuning.

The main contributions of this paper are as follows:

- *Performance-portable SYCL Optimizations*. We enhance the LiGen drug discovery pipeline by integrating advanced portable SYCL and oneAPI features to dynamically adapt the kernel workload to the target hardware by leveraging run-time heuristics, while removing the need for hardware-specific autotuning.
- *Architecture-specific Optimizations on Intel GPUs*. We explore architecture-specific optimizations exclusive to Intel GPUs, focusing on sub-group size tuning and GRF utilization to maximize performance on Intel GPUs;
- *Experimental Evaluation*. We conducted an extensive evaluation of LiGen by comparing the SYCL manually tuned baseline implementation (*SYCL-tuned*) with the same SYCL version implemented using the novel portable heuristic (*SYCL-heur*) and the Intel-optimized SYCL code (*SYCL-intel*). Furthermore, we provided a cross-platform roofline analysis that compares the CUDA manually tuned implementation executed on NVIDIA V100S with the *SYCL-intel* code executed on Intel Max 1100.

The remainder of this paper is organized as follows: Section II introduces the background and motivation, including the LiGen virtual screening pipeline, Intel GPU architecture characteristics, and the SYCL programming model. Section III details the proposed performance-portable heuristic and Intel-specific optimizations, illustrating how SYCL and oneAPI features are employed to dynamically adapt kernel configurations. Section IV describes the experimental setup and presents a comprehensive evaluation of the proposed optimizations, including comparisons with manually tuned CUDA and SYCL implementations and a roofline analysis. Section V reviews related work on virtual screening and performance portability. Finally, Section VI concludes the paper and outlines future research directions.

## II. Background and Motivation

### A. Virtual Screening with LiGen

Virtual screening [4] is a computational technique used in drug discovery to evaluate a large number of drug candidates, called ligands, against a target protein associated with a disease. The goal is to predict the interaction strength between each ligand and the protein, enabling researchers to prioritize the most promising candidates for experimental validation. In this context, a *ligand* is a small molecule, typically with fewer than a hundred *atoms*, whose structure can adapt when interacting with a protein. Each ligand may contain multiple rotatable *bonds*, which divide its atoms into fragments, i.e. subsets of atoms, that can rotate relative to one another without altering physical and chemical properties. The protein contains multiple potential binding regions, referred to as *pockets*, which are typically cavities on its surface where ligands can dock [5]. LiGen [6] performs virtual screening through a sequence of three computational phases for each ligand-pocket pair:

- *Docking phase*: LiGen generates multiple alternative poses of the ligand inside the target pocket using a multi-restart gradient descent approach. Each pose is initialized with a deterministic heuristic, aligned to the pocket via rigid roto-translations, and refined by optimizing the flexible *fragment*s;
- *Pose filtering phase*: To reduce computational effort, only the most promising poses are selected for scoring. Poses are ranked according to a simple geometric heuristic and clustered based on structural similarity to promote diversity. This filtering ensures that downstream scoring focuses on representative and high-quality poses;
- *Scoring phase*: Each retained pose is evaluated using LiGen's chemical scoring function, which reduces the interaction strength estimation to a single numerical score. The final score of the ligand corresponds to the highest value among its poses, reflecting the most favorable predicted interaction with the pocket.

### B. NVIDIA and Intel GPU Architectures

LiGen is a highly parallel application, making it well-suited for execution on GPU accelerators. The initial implementation of LiGen targeted CPU architectures [7], but the increasing computational demands of molecular docking and scoring motivated its migration to NVIDIA GPUs and, subsequently, to heterogeneous systems comprised of GPUs from various vendors, including AMD and Intel [8]. This work focuses on optimizing LiGen on Intel GPUs by leveraging specific Intel GPU architectural properties. Intel's GPU architecture adopts a hierarchical design that, while conceptually similar to NVIDIA's architecture, introduces its own terminology and organization. Over time, Intel has refined its naming conventions, and in this subsection, we refer to the most recent terminology, establishing a comparison with the corresponding NVIDIA concepts. An Intel GPU is composed of one or more $X^e$ *Stack*, which can be viewed as independent programmable GPU units, interconnected through High Bandwidth Memory (HBM). Each $X^e$ *Stack* contains multiple $X^e$ *Slices*, which share the same L2 cache and form the intermediate level of the hierarchy. An $X^e$ *Slice* serves a role comparable to NVIDIA's Graphics Processing Cluster (GPC), representing a top-level compute block that aggregates several mid-level processing units. An $X^e$ *Core* incorporates both $X^e$ *Vector* and *Matrix* engine, alongside a local L1/SLM cache. Each $X^e$ *Core* can compute simultaneously one or more work

groups (CUDA thread block). The $X^e$ *Vector Engine* is a multi-threaded SIMD processor capable of executing multiple work-items concurrently within a single hardware thread. The compiler generates SIMD code that maps several work-items (i.e., CUDA threads) to the vector lanes of a hardware thread, ensuring efficient parallel execution. Each hardware thread on the Intel Max 1100 features up to 256 registers and can schedule one or more sub-groups (i.e., CUDA warps), enabling simultaneous multithreading across the $X^e$ *Vector Engines*. Conceptually, the $X^e$ *Vector Engine* corresponds to NVIDIA's warp scheduler, representing the smallest thread-level execution unit responsible for dispatching and managing SIMT instructions. However, the execution models of the two architectures differ in how they realize this parallelism. NVIDIA GPUs implement SIMT execution through warp scheduling, whereas Intel GPUs achieve similar behavior through explicit SIMD vectorization. On NVIDIA architectures, the warp size is fixed at 32 threads, and both the compiler and hardware are optimized for this granularity. This fixed warp width dictates the SIMT execution behavior, instruction scheduling, and register allocation strategy. In contrast, Intel GPUs provide a configurable sub-group size (e.g. 16 and 32 work-items per sub-group for Intel Max 1100), depending on the hardware and compiler support. This flexibility allows developers to tune the effective SIMD width to better match the computational characteristics of a given kernel (i.e. SIMD vector width, register pressure, occupancy).

### C. SYCL Programming Model

To efficiently target multiple hardware backends, including Intel and NVIDIA GPUs, we employ the SYCL programming model, which enables developers to write standard C++ code for heterogeneous hardware such as CPUs and GPUs. It provides a high-level abstraction over backends such as OpenCL, CUDA, HIP, and Level Zero, enabling single-source C++ where host and device code coexist. At the core of SYCL's execution model is the concept of queues. A `sycl::queue` abstracts kernel submission to a target device. It encapsulates a `sycl::device` object, which abstracts the underlying hardware resource. The queue manages operation ordering, kernel submissions, and host–device data transfers. SYCL provides both in-order and out-of-order queue execution models, allowing developers to control task scheduling and dependency management for improved flexibility and performance. Kernels are submitted to a queue through command groups, which represent units of work to be executed on the device. Each command group can be submitted to a `sycl::queue` through the `submit` method. The SYCL runtime uses these command groups to manage dependencies and schedule tasks efficiently across heterogeneous devices. Parallelism is expressed in a command group by the `parallel_for` function, which accepts a C++ lambda function representing the kernel body. This lambda is executed in parallel across a range of work-items (i.e. CUDA threads). The iteration domain is defined through a `sycl::nd_range` object, which specifies both the total number of work-items and how work-items are organized in work-groups (i.e. CUDA thread blocks). This structure is analogous to the grid and block configuration of CUDA. Within a work-group, work-items are further organized into sub-groups (i.e. CUDA warps), which correspond to smaller execution units that are typically mapped to the hardware's SIMD (Single Instruction, Multiple Data) units. SYCL provides two main approaches for memory management and data access. The first is the buffer-accessor model, where data is encapsulated in `sycl::buffer` objects, and access is declared through `sycl::accessor` instances specified in the command group. This approach allows the SYCL runtime to automatically handle memory transfers and enforce dependency tracking between kernels, ensuring correct data movement and synchronization. The second approach is Unified Shared Memory (USM) which provides a pointer-based access to the data in a similar way to the one provided by CUDA.

### D. Towards SYCL Optimizations for LiGen

Over time, several optimizations have been applied to LiGen to utilize the increasing computational capabilities of GPUs. The first GPU-enabled implementation, referred to as LiGen-Latency, was developed using CUDA for NVIDIA GPUs [7]. In this version, each GPU kernel is responsible for the computation of a single ligand, fully parallelizing the internal operations of the ligand across GPU cores. This approach focuses on reducing the latency of individual ligand evaluations by distributing the computation of atomic interactions and gradient calculations across the GPU threads. When the computational load of a single ligand is insufficient to saturate all available GPU resources, multiple ligands are processed concurrently using software threads and CUDA streams. To improve GPU utilization further, a second version, namely LiGen-Batch [9], was developed. In this implementation, each kernel processes multiple ligands simultaneously, assigning the computation of each ligand to a CUDA warp (i.e., a sub-group). This batching strategy improves throughput by exploiting the inherent parallelism across ligands, significantly increasing occupancy and overall GPU efficiency. Both LiGen-Latency and LiGen-Batch were subsequently ported to SYCL to support execution on heterogeneous architectures. The SYCL implementation leverages SYCL 2020 features, including USM for pointer-based data management, buffer-accessor abstractions for dependency handling, and group algorithms and sub-groups for fine-grained parallelism. Among these, the SYCL LiGen-Batch version with USM has demonstrated the best performance and is used as the reference implementation in this work [8]. However, despite these optimizations, the SYCL LiGen-Batch implementation exhibits some limitations. *The first SYCL implementation of LiGen did not use features to query kernel characteristics or tune execution parameters for maximizing occupancy, as they were not yet defined in the SYCL.* As a result, achieving optimal performance often depended on manual parameter selection or tuning, which was not easily generalizable across different devices. Second, *the existing implementation does not account for architecture-*

*specific characteristics of Intel GPUs that can significantly affect the overall throughput*. Addressing these limitations is essential for achieving both performance portability and architecture-specific optimizations on Intel GPUs. For the rest of the paper, we refer to LiGen-Batch as SYCL-Baseline, as it performs the best among all earlier versions.

## III. Performance-portable and Intel-specific Optimizations

### A. Performance-Portable Optimizations with oneAPI

In the *SYCL-baseline* and *SYCL-tuned* implementations of LiGen, the number of ligands processed per kernel is determined by compile-time parameters. Setting these parameters typically requires detailed hardware knowledge and manual tuning to fully exploit the available GPU resources. The first SYCL version (i.e., *SYCL-baseline*) lacked the runtime mechanisms to query the information necessary for dynamically adapting workload sizes to the characteristics of the target hardware, as SYCL 2020 did not yet provide such features. To address this limitation, we introduce a portable runtime heuristic that leverages new SYCL and oneAPI features to automatically adapt the number of ligands processed per kernel based on the underlying device. This heuristic enables dynamic workload balancing and optimal resource utilization without requiring device-specific autotuning or user knowledge of the hardware. Although improving ILP is one of the ways to enhance performance [10], it also increases register pressure by raising the workload per thread. Since our baseline implementation already suffers from significant register pressure [8], we instead chose to maximize occupancy to achieve higher performance. Algorithm 1 presents the pseudo-code for our heuristic, referred to as *SYCL-heur*. The function takes as input the `sycl::device` object, which abstracts the target hardware, the `sycl::kernel` under optimization, the amount of Shared Local Memory (SLM) required per ligand (*slm_per_lig*), and the list of valid work-group sizes (*wgs_vec*) to evaluate. By using the `sycl::kernel_bundle` and the `kernel_device_specific::work_group_size` property, we can query the maximum work-group size supported by a kernel, considering resource constraints such as SLM usage and register availability. The minimum valid work-group size for LiGen is defined by the smallest available sub-group size, obtained from the device property `device::sub_group_sizes`, while the maximum value is defined by the SYCL feature `max_work_group_size`. The range between these minimum and maximum values forms the set of valid work-group sizes tested by the heuristic. In line 1 of Algorithm 1, the heuristic retrieves the default sub-group size used by the kernel through the `compile_sub_group_size` property. Then, in line 3, it determines the number of Compute Units (CUs) available on the device via the oneAPI extension `device::num_compute_units`, which abstracts the concept of NVIDIA Streaming Multiprocessors (SMs) and Intel $X^e$ *Cores*. For each candidate work-group size, the heuristic computes the number of ligands processed per kernel as [11]:

$$l = b \times CU \times \frac{wgs}{sgs} \tag{1}$$

where $b$ is the maximum number of active work-groups per compute unit, $CU$ is the number of compute units, $wgs$ is the number of work-items per work-group, and $sgs$ is the sub-group size (equivalent to a CUDA warp). The function `max_active_work_groups_per_cu()` (line 6) estimates $b$ by considering SLM usage, register pressure, and work-group size, providing an upper bound on the number of concurrently active work-groups for a given kernel configuration. It derives this bound from hardware thread limits on a single $X^e$ core and further restricts it based on barrier usage and shared local memory capacity, yielding a conservative estimate of achievable concurrency. The heuristic first identifies the configurations that maximize ligand throughput (lines 7–10). If multiple configurations yield the same throughput, it further refines the selection by minimizing register spilling (line 11), using the oneAPI property `kernel_device_specific::spill_memory_size` accessible through the `kernel_bundle`. This approach enables LiGen to automatically adapt its workload to different hardware devices while maintaining performance portability across GPU architectures.

**Algorithm 1:** *SYCL-heur for optimizing throughput*

**Input:** device, slm_per_lig, kernel, wgs_vec
**Output:** opt_config

1 $sgs \leftarrow kernel.get\_sub\_group\_size()$;
2 $max\_lig\_kern \leftarrow 0,\ configs \leftarrow \emptyset$;
3 $cu = device.num\_compute\_units()$;
4 **for** $wgs \in wgs\_vec$ **do**
5 $\quad slm\_wg \leftarrow slm\_per\_lig \cdot (wgs/sgs)$;
6 $\quad lig\_kern \leftarrow max\_wg\_per\_cu(wgs, slm\_wg, sgs) \cdot cu \cdot (wgs/sgs)$;
7 $\quad$ **if** $lig\_kern > max\_lig\_per\_kern$ **then**
8 $\quad\quad max\_lig\_per\_kern \leftarrow lig\_per\_kern$; $configs \leftarrow \{(kernel, wgs, sgs)\}$;
9 $\quad$ **else if** $lig\_kern = max\_lig\_kern$ **then**
10 $\quad\quad configs \leftarrow configs \cup \{(kernel, wgs, sgs)\}$;
11 $opt\_config \leftarrow \underset{c \in configs}{\arg\min}\ c.get\_spilled\_register()$;
12 **return** $opt\_config$;

### B. Intel-Specific Optimization

Although SYCL aims to achieve performance portability across heterogeneous architectures, hardware-specific features can still be exploited to achieve higher performance on a given platform. The SYCL specification allows extensions that are ratified by the Khronos SYCL group or supplied by individual vendors. In this context, oneAPI, as a SYCL implementation, provides both portable and vendor-specific extensions, such as GRF, enabling developers to fine-tune kernel execution

for specific hardware backends through backend interoperability [3]. Intel GPUs, in particular, expose several architectural parameters that can be adjusted to improve computational efficiency. Two key tunable parameters on Intel Max 1100 are the sub-group size and the GRF configuration. These parameters directly influence SIMD width, register allocation, and kernel occupancy, all of which play a crucial role in maximizing performance on Intel hardware. On Intel Max 1100, each $X^e$ *Core* can execute SIMD-16 or SIMD-32 instructions, corresponding to sub-group sizes of 16 and 32, respectively. Modifying the sub-group size affects how work-items are mapped to hardware threads, directly impacting occupancy and register utilization. Since each hardware thread has a fixed number of registers, using smaller sub-groups (e.g., 16) reduces register pressure and enables more active work-groups per $X^e$ *Core*. Conversely, larger sub-groups (e.g., 32) can increase parallelism but may decrease occupancy due to higher register demands. To exploit these properties, we extended the portable heuristic defined in Algorithm 1 to account for all available sub-group sizes supported by the target Intel GPU. Instead of querying only the default sub-group size (as in line 1 of Algorithm 1), the new version queries the full list of supported sub-group sizes using the `device::sub_group_sizes` property. The input kernel is then compiled for each sub-group size, and the heuristic evaluates their impact on occupancy and register spilling. In addition to sub-group tuning, Intel GPUs provide control over the GRF configuration. The GRF determines the number of registers allocated per thread, with three supported modes that on Intel Max 1100 correspond to the following configurations: *GRF-Small* allows for 128 registers per thread with 8 hardware threads for $X^e$ *Slice*; *GRF-Large* allows for 256 registers per thread with four hardware threads, which, compared to *GRF-Small*, can reduce register spilling but potentially lower occupancy since the number of hardware threads is halved; *GRF-Auto* allows the compiler to select a configuration between *GRF-Small* and *GRF-Large* based on its internal heuristics . From our experiments on LiGen, we observed that in the most computationally expensive kernel, the compiler heuristic does not always yield the optimal GRF and sub-group size configuration. To address this, the heuristic was further extended to compile and test all combinations of supported sub-group sizes and GRF modes. The loop in Algorithm 1 (lines 4-10) was thus enhanced with two additional nested loops iterating over the available sub-group sizes and GRF configurations. For each combination, the algorithm evaluates occupancy and register spilling by selecting the configuration that maximizes the ligand throughput. This enhanced version of the heuristic leverages the oneAPI extensions `intel::grf_size` for specifying the GRF mode and `intel::reqd_sub_group_size` for specifying the sub-group size. By systematically exploring the configuration space of sub-group size and GRF mode, our approach attempts to identify optimal kernel configuration for Intel GPUs, leading to significant throughput improvements without requiring manual tuning (Section IV-C).

### C. LiGen SYCL Implementations

The *SYCL-baseline* implements the LiGen pipeline where the number of ligands processed per kernel is determined using vendor-provided tools, such as the CUDA or Intel's GPU Occupancy Calculator[1]. Both the *SYCL-baseline* and the CUDA implementation use the same work-group and sub-group sizes. The *SYCL-heur* implementation introduces a dynamic heuristic that moves the computation of the number of ligands per kernel and the tuning of the work-group size directly inside the LiGen code. By using advanced SYCL and oneAPI features, this version can automatically adapt these parameters according to the number of atoms and fragments processed by each bucket, without requiring out of kernel tuning or multiple profiling runs. Furthermore, *SYCL-heur* remains portable across architectures, as it does not rely on hardware-specific features such as variable sub-group sizes or GRF registers. The *SYCL-intel* implementation builds upon *SYCL-heur* by incorporating Intel-specific optimizations within the heuristic. In particular, it exploits Intel GPU capabilities such as the GRF configuration and sub-group size tuning to further enhance performance on Intel architectures. Finally, the *SYCL-tuned* version represents the upper bound of achievable performance. In this implementation, we manually determine the best configuration for each combination of atoms and fragments by exhaustively exploring possible work-group sizes, sub-group sizes, and GRF settings through extensive profiling. Table I summarizes the configuration parameters used by all implementations. The *SYCL-baseline* employs a work-group size of 128, a sub-group size of 32, and determines the number of ligands per kernel using Intel's occupancy calculator together with the formula in Eq. 1. In the *SYCL-heur* implementation, both the work-group size and the number of ligands per kernel are selected dynamically at runtime by the heuristic, while architecture-specific parameters such as sub-group size and GRF mode remain fixed. The *SYCL-intel* implementation extends this approach by additionally exploring variable sub-group sizes (16 and 32) and different GRF configurations (small and large). Finally, the *SYCL-tuned* version represents the upper bound of optimization, as it selects the best configuration for each atom-fragment combination by exhaustively evaluating all possible parameter settings.

## IV. Experimental Evaluation

In this section, we describe the experimental setup and then discuss both portable and Intel-specific SYCL optimizations in detail, comparing them against the CUDA implementation.

### A. Experimental Setup

For all SYCL-based experiments, we used a system equipped with an Intel Max 1100 GPU at CINECA and the `icpx` compiler from the *oneAPI Base Toolkit* version 2025.1.0. Profiling was conducted using Intel UniTrace [12]. For the CUDA-based experiments, we utilized

[1]https://oneapi-src.github.io/oneAPI-samples/Tools/GPU-Occupancy-Calculator/

TABLE I: Overview of the tuning parameters for all SYCL LiGen implementations.

| Version | Work-group | Sub-group | GRF | Lig. per Kernel |
|---|---|---|---|---|
| ***SYCL-baseline*** | Fixed (128) | Default (32) | Compiler defined (auto) | Fixed (1120) |
| ***SYCL-heur*** | Heuristic defined (64 to 1024) | Default (32) | Compiler defined (auto) | Heuristic defined |
| ***SYCL-intel*** | Heuristic defined (64 to 1024) | Heuristic defined (16, 32) | Heuristic defined (small, large) | Heuristic defined |
| ***SYCL-tuned*** | Manually tuned (64 to 1024) | Manually tuned (16, 32) | Manually tuned (small, large) | Manually tuned (1000 to 4000) |

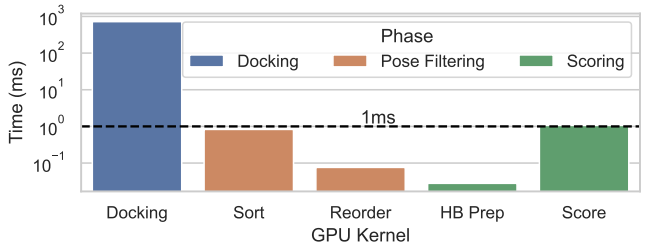



Fig. 1: Device time of the GPU kernels in the LiGen pipeline.

a separate system featuring an NVIDIA V100S GPU and the `nvcc` compiler from CUDA version 12.1. Profiling in this case was performed with NVIDIA Nsight Compute. The experiments were conducted using data with 31, 63, 74, and 89 atoms, and datasets comprising 100,000 ligands. Furthermore, fragment sizes of 4, 8, 16, and 20 were considered. Each experimental configuration was executed three times. The median coefficient of variation is less than one percent ($0.15\%$), indicating measurement stability. As illustrated in Fig. 1, the docking kernel accounts for more than 99% of the device execution time for a pipeline running on the NVIDIA V100S (CUDA). Consequently, our optimization efforts focused exclusively on the docking kernel.

### B. Performance-Portable Optimizations

Fig. 2 illustrates the throughput achieved by the four implementations of the LiGen docking kernel across different numbers of atoms and fragments. Among these four implementations, *SYCL-baseline* and *SYCL-heur* only consider SYCL features that are not architecture-specific, meaning that a default sub-group size of 32 (SG $=$ 32) and *GRF-Auto* are used. The heuristic described in Algorithm 1 predicts, at runtime, the optimal number of ligands processed by each docking kernel together with the corresponding work-group size. Ideally, these values should be close to those computed by the GPU-specific occupancy calculator, as defined in Eq. 1. *SYCL-baseline*, serving as a lower bound, uses a fixed work-group size of 128 and requires the user to manually determine the number of ligands per docking kernel using the occupancy calculator. In contrast, the proposed heuristic performs this computation automatically at runtime. As shown in Fig. 2, for atoms of 74 and 89, the heuristic successfully selects a more efficient configuration of work-group size and number of ligands ($l$), compared to *SYCL-baseline*. For instance, in the case of 89 atoms and 20 fragments, *SYCL-baseline* uses 256 as the work-group size and 1120 as the number of ligands per kernel, computed externally using the occupancy calculator (Table I). In contrast, the heuristic yields a work-group size of 512 and 1792 ligands per kernel. As a result, *SYCL-heur* achieves $1.68\times$ higher throughput.

### C. Intel-Specific Optimizations

*SYCL-intel* extends the device-portable *SYCL-heur* implementation by leveraging Intel PVC features and oneAPI SYCL extensions to enhance throughput. This improvement is achieved by trading thread occupancy per $X^e$ *Core* for a higher number of available registers per work-item. Consequently, the approach effectively alleviates the primary bottleneck of the baseline docking kernel [8], namely register spilling, thus improving the cache-hit ratio by preventing cache evictions induced by register spilling. Furthermore, reducing the sub-group size increases the share of each work-item with the configured GRF mode, albeit at the expense of reduced thread occupancy due to the overhead of scheduling more sub-groups at the hardware level. This aligns with the results shown in Fig. 2, where most of the best-performing configurations for *SYCL-intel* correspond to SG $=$ 32 and *GRF-Large*. The gray region in Fig. 2 illustrates the improvement obtained from these Intel-specific optimizations without modifying the kernel code. For example, with ligands consisting of 89 atoms and 16 fragments, the performance-portable implementation achieves $1.68\times$ the throughput of the baseline, while the Intel-optimized version reaches $2.42\times$, closely approaching the manually optimized upper bound of *SYCL-tuned*. Similarly, for ligands containing 63 atoms and 20 fragments, *SYCL-heur* employs 3000 ligands per kernel (*GRF-auto*), resulting in a $1.02\times$ throughput improvement over the baseline. In contrast, the heuristic in *SYCL-intel* uses 1792 ligands per kernel with *GRF-large*, achieving $1.69\times$ the baseline throughput (*SYCL-baseline*). The effect of forcing *GRF-large* is more evident in the case with 89 atoms and 20 fragments. For this configuration, *SYCL-baseline* uses GRF-auto, work-group size of 256, and 1120 ligands per kernel. In contrast, while both *SYCL-heur* and *SYCL-intel* use 1792 ligands per kernel, the use of *GRF-large* in *SYCL-intel* increases the speedup from 1.68 to 2.42 with respect to *SYCL-baseline*. Since the number of atoms in each ligand determines the amount of work assigned per sub-group, a higher throughput is achieved with SG $= 16$ for 31 atoms and 4 fragments. In this case, *SYCL-intel* achieves $1.35\times$ of the baseline throughput, while *SYCL-heur*, with a sub-group size of 32, delivers $1.02\times$. Thanks to the runtime heuristic and the additional Intel-specific optimizations, *SYCL-intel* closely matches the performance of *SYCL-tuned*, which serves as the upper bound configuration using the optimal parameter configurations determined by manual tuning of all the parameters defined in Table I. Notably, *SYCL-intel* achieves this performance without requiring any profiling.

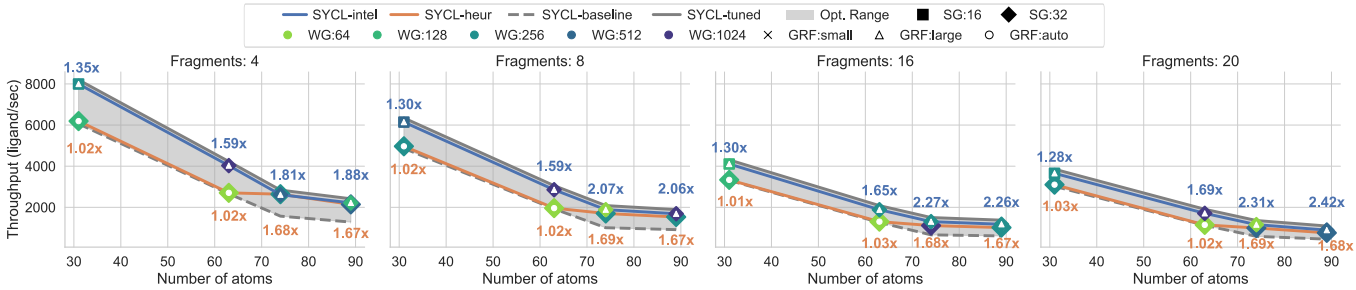
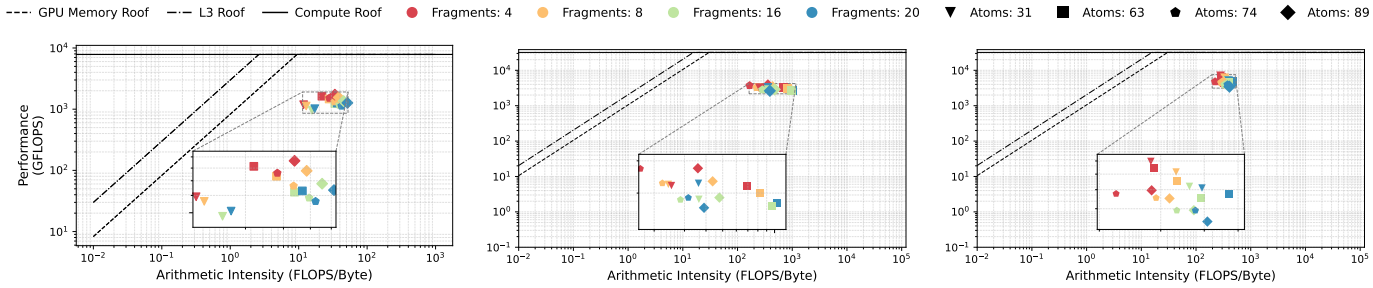



Fig. 2: Ligand per second of *SYCL-tuned* (SYCL-baseline with manual tuning and optimization), *SYCL-heur* (SYCL baseline with the portable heuristic), and *SYCL-intel* (*SYCL-heur* with Intel GPU-specific optimization). The speedups are with respect to the *SYCL-baseline*. *WG* and *SG* denote the work-group and sub-group sizes, respectively.




(a) CUDA on NVIDIA V100S (b) *SYCL-heur* on Intel Max 1100 (c) *SYCL-intel* on Intel Max 1100

Fig. 3: Roofline analysis of the CUDA, *SYCL-heur*, and *SYCL-intel* implementations.

### D. Comparison of the different versions

To gain a deeper understanding of how effectively the LiGen implementations utilize their GPUs, we conducted a roofline analysis of the CUDA version of LiGen on the NVIDIA V100S and of the *SYCL-heur* and *SYCL-intel* versions on the Intel Max 1100. Fig. 3 provides roofline models of three implementations of LiGen docking kernel for different numbers of atoms and fragments: CUDA on NVIDIA V100S, along with *SYCL-heur* and *SYCL-intel* on Intel Max 1100. Compared to *SYCL-heur*, a better throughput is achieved by *SYCL-intel*. The geometric mean speedup of *SYCL-heur* and *SYCL-intel* with respect to *SYCL-baseline* is $1.31\times$ and $1.76\times$, respectively. Furthermore, *SYCL-intel* (Fig. 3c) exhibits improved GPU utilization, as its kernel samples in the roofline model are positioned closer to the computational bound compared to those of *SYCL-heur* (Fig. 3b). To quantify how well these three implementations utilize their respective GPU devices, we use *roofline efficiency* [13], [14], defined as the ratio of the measured kernel performance over the peak device performance. The CUDA implementation on NVIDIA V100S achieves a roofline efficiency of $16.4\ \%$, while on Intel Max 1100, *SYCL-heur* and *SYCL-intel* achieve $11.7\ \%$ and $21.8\ \%$, respectively. Note that these values represent the geometric mean of the roofline efficiency ratios computed for all atom and fragment sizes.

## V. Related Work

Virtual screening [15], [16] is a key computational technique in modern drug discovery, used to evaluate millions of ligand–protein interactions to identify promising drug candidates. Given its computational intensity, GPU accelerators have become essential for implementing high-throughput virtual screening pipelines [17]–[20]. LiGen [21] was developed to address these challenges by providing a high-performance virtual screening pipeline optimized for GPU execution [7]. The application was initially implemented in CUDA and progressively optimized to take advantage of NVIDIA GPUs efficiently [9]. To extend its portability across heterogeneous systems, LiGen has been ported to SYCL, integrating advanced SYCL 2020 features such as USM, buffer-accessors, sub-groups, and group algorithm [22] to improve performance portability and maintain competitive efficiency on different GPU architectures [8], [23], [24]. The SYCL programming model, defined by the Khronos Group [2], proposes a performance-portable solution for heterogeneous computing. It is a single-source, modern C++ interface that enables developers to target CPUs, GPUs, and accelerators with a unified codebase. Recent research has focused on improving performance portability and programmability for SYCL through new features such as sub-groups, group algorithms, and kernel bundles [25], [26]. Several studies evaluated the performance of SYCL against vendor-specific programming models such as CUDA and HIP [13], [27]–[29], highlighting both its potential

for cross-platform performance and the remaining challenges in automatic workload adaptation and resource tuning. In this work, *we explored the new SYCL and oneAPI features to extend the SYCL implementation of LiGen* by introducing a performance-portable heuristic that dynamically adjusts the number of ligands processed per kernel to the target hardware, removing the need for manual tuning or hardware-specific expertise. Furthermore, *we extended the heuristic to encompass architecture-specific optimizations for Intel GPUs*, focusing on automatic tuning of sub-group size and GRF, to fully exploit Intel GPU architectures and maximize ligand throughput.

## VI. Conclusion

In this work, we improved the LiGen drug discovery pipeline with a portable heuristic that dynamically adapts the kernel workload at runtime by taking into account factors such as occupancy, register spilling, and work-group size tuning. We further extended this heuristic with Intel-specific optimizations, incorporating sub-group size tuning and GRF utilization to enhance performance beyond what can be achieved with portable techniques alone. Our findings show that the proposed portable heuristic (*SYCL-heur*) consistently outperforms the *SYCL-baseline* implementation without requiring manual tuning, while the Intel-optimized version (*SYCL-intel*) delivers additional performance gains of up to $2.42\times$ by demonstrating the importance of leveraging architecture-specific features.

## References


[1] F. Liu, O. Mailhot, I. S. Glenn, S. F. Vigneron, V. Bassim, X. Xu, K. Fonseca-Valencia, M. S. Smith, D. S. Radchenko, J. S. Fraser *et al.*, "The impact of library size and scale of testing on virtual screening," *Nature Chemical Biology*, pp. 1–7, 2025.

[2] Khronos Group, *SYCL 2020 Specification*, Khronos Group, 2020, accessed 2025-10-20. [Online]. Available: https://www.khronos.org/registry/SYCL/specs/sycl-2020/html/sycl-2020.html

[3] A. Alpay, T. Applencourt, G. Brown, R. Keryell, and G. Lueck, "Using interoperability mode in SYCL 2020," in *Proceedings of the 10th International Workshop on OpenCL*, 2022, pp. 1–1.

[4] N. A. Murugan, A. Podobas, D. Gadioli, E. Vitali, G. Palermo, and S. Markidis, "A review on parallel virtual screening softwares for high-performance computers," *Pharmaceuticals*, vol. 15, 2021.

[5] N. S. Pagadala, K. Syed, and J. A. Tuszynski, "Software for molecular docking: a review," *Biophysical Reviews*, vol. 9, pp. 91 – 102, 2017.

[6] C. Beato, A. R. Beccari, C. Cavazzoni, S. Lorenzi, and 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," *Journal of Chemical Information and Modeling*, vol. 53, no. 6, pp. 1503–1517, 2013.

[7] D. Gadioli, E. Vitali, F. Ficarelli, C. Latini, C. Manelfi, C. Talarico, C. Silvano, C. Cavazzoni, G. Palermo, and A. R. Beccari, "EXSCALATE: An extreme-scale virtual screening platform for drug discovery targeting polypharmacology to fight SARS-CoV-2," *IEEE Transactions on Emerging Topics in Computing*, vol. 11, no. 1, pp. 170–181, 2022.

[8] L. Crisci, L. Carpentieri, B. Cosenza, G. Accordi, D. Gadioli, E. Vitali, G. Palermo, A. R. Beccari *et al.*, "Enabling performance portability on the LiGen drug discovery pipeline," *Future Generation Computer Systems*, vol. 158, pp. 44–59, 2024.

[9] E. Vitali, F. Ficarelli, M. Bisson, D. Gadioli, G. Accordi, M. Fatica, A. R. Beccari, and G. Palermo, "GPU-optimized approaches to molecular docking-based virtual screening in drug discovery: A comparative analysis," *Journal of Parallel and Distributed Computing*, vol. 186, p. 104819, 2024.

[10] G. Shobaki, A. Kerbow, and S. Mekhanoshin, "Optimizing occupancy and ILP on the GPU using a combinatorial approach," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 133–144.

[11] G. Accordi, D. Gadioli, E. Vitali, L. Crisci, B. Cosenza, A. Beccari, and G. Palermo, "Out of kernel tuning and optimizations for portable large-scale docking experiments on GPUs," *J. Supercomput.*, vol. 80, no. 8, p. 11798–11815, Feb. 2024.

[12] Intel Corporation, *Intel/pti-gpu: UniTrace, a unified tracing and profiling tool*, 2025, accessed: 2025-10-20. [Online]. Available: https://github.com/intel/pti-gpu/tree/master/tools/unitrace

[13] J. Kwack, J. Tramm, C. Bertoni, Y. Ghadar, B. Homerding, E. Rangel, C. Knight, and 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)*. IEEE, 2021, pp. 45–56.

[14] C. Bertoni, J. Kwack, T. Applencourt, Y. Ghadar, B. Homerding, C. Knight, B. Videau, H. Zheng, V. Morozov, and S. Parker, "Performance portability evaluation of OpenCL benchmarks across Intel and NVIDIA platforms," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2020, pp. 330–339.

[15] J. Biesiada, A. Porollo, P. Velayutham, M. Kouril, and J. Meller, "Survey of public domain software for docking simulations and virtual screening," *Human Genomics*, vol. 5, no. 5, p. 497, 2011.

[16] E. Yuriev, J. Holien, and P. A. Ramsland, "Improvements, trends, and new ideas in molecular docking: 2012–2013 in review," *Journal of Molecular Recognition*, vol. 28, no. 10, pp. 581–604, 2015.

[17] S. Tang, R. Chen, M. Lin, Q. Lin, Y. Zhu, J. Ding, H. Hu, M. Ling, and J. Wu, "Accelerating AutoDock Vina with GPUs," *Molecules*, vol. 27, no. 9, p. 3041, 2022.

[18] L. Solis-Vasquez, E. Mascarenhas, and A. Koch, "Experiences migrating CUDA to SYCL: A molecular docking case study," in *Proceedings of the 2023 International Workshop on OpenCL*, 2023, pp. 1–11.

[19] O. Korb, T. Stutzle, and T. E. Exner, "Accelerating molecular docking calculations using graphics processing units," *Journal of Chemical Information and Modeling*, vol. 51, no. 4, pp. 865–876, 2011.

[20] Y. Fang, Y. Ding, W. P. Feinstein, D. M. Koppelman, J. Moreno, M. Jarrell, J. Ramanujam, and M. Brylinski, "GeauxDock: accelerating structure-based virtual screening with heterogeneous computing," *PloS one*, vol. 11, no. 7, p. e0158898, 2016.

[21] A. R. Beccari, C. Cavazzoni, C. Beato, and G. Costantino, "LiGen: A high performance workflow for chemistry driven de novo design," *Journal of Chemical Information and Modeling*, vol. 53, no. 6, pp. 1518–1527, 2013.

[22] Intel Corporation, *Intel/LLVM: SYCL and oneAPI Extensions Documentation*, 2025, accessed: 2025-10-20. [Online]. Available: https://github.com/intel/llvm/tree/sycl/sycl/doc/extensions

[23] D. Gadioli, G. Palermo, S. Cherubin, E. Vitali, G. Agosta, C. Manelfi, A. R. Beccari, C. Cavazzoni, N. Sanna, and C. Silvano, "Tunable approximations to control time-to-solution in an HPC molecular docking Mini-App," *J. Supercomput.*, vol. 77, no. 1, p. 841–869, Jan. 2021.

[24] G. Accordi, D. Gadioli, G. Palermo, L. Crisci, L. Carpentieri, B. Cosenza, and A. R. Beccari, "Unlocking performance portability on LUMI-G supercomputer: A virtual screening case study," in *Proceedings of the 12th International Workshop on OpenCL and SYCL*, 2024, pp. 1–4.

[25] L. Crisci, L. Carpentieri, P. Thoman, A. Alpay, V. Heuveline, and B. Cosenza, "SYCL-bench 2020: Benchmarking SYCL 2020 on AMD, Intel, and NVIDIA GPUs," in *Proceedings of the 12th International Workshop on OpenCL and SYCL*, 2024, pp. 1–12.

[26] T. Applencourt, B. Videau, J. Le Quellec, A. Dufek, K. Harms, N. Liber, B. Allen, and A. Belton-Schure, "Standardizing complex numbers in SYCL," in *Proceedings of the 2023 International Workshop on OpenCL*, 2023, pp. 1–6.

[27] A. Alekseenko and S. Páll, "Comparing the performance of SYCL runtimes for molecular dynamics applications," in *Proceedings of the 2023 International Workshop on OpenCL*, 2023, pp. 1–2.

[28] M. Breyer, A. Van Craen, and D. Pflüger, "Evaluation of SYCL's different data parallel kernels," in *Proceedings of the 12th International Workshop on OpenCL and SYCL*, 2024, pp. 1–4.

[29] I. Z. Reguly, "Evaluating the performance portability of SYCL across CPUs and GPUs on bandwidth-bound applications," in *Proceedings*

*of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, 2023, pp. 1038–1047.

[30] L. Carpentieri, M. D'Antonio, K. Fan, L. Crisci, B. Cosenza, F. Ficarelli, D. Cesarini, G. Accordi, D. Gadioli, G. Palermo *et al.*, "Domain-specific energy modeling for drug discovery and magnetohydrodynamics applications," in *Proceedings of the SC'23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*, 2023, pp. 1790–1800.