The Journal of Supercomputing
https://doi.org/10.1007/s11227-023-05884-y

Out of kernel tuning and optimizations for portable large-scale docking experiments on GPUs

Gianmarco Accordi[1] · Davide Gadioli[1] · Emanele Vitali[1,2] · Luigi Crisci[3] · Biagio Cosenza[3] · Andrea Beccari[4] · Gianluca Palermo[1]

Accepted: 23 December 2023

Abstract
Virtual screening is an early stage in the drug discovery process that selects the most promising candidates. In the urgent computing scenario, finding a solution in the shortest time frame is critical. Any improvement in the performance of a virtual screening application translates into an increase in the number of candidates evaluated, thereby raising the probability of finding a drug. In this paper, we show how we can improve application throughput using Out-of-kernel optimizations. They use input features, kernel requirements, and architectural features to rearrange the kernel inputs, executing them out of order, to improve the computation efficiency. These optimizations' implementations are designed on an extreme-scale virtual screening application, named LiGen, that can hinge on CUDA and SYCL kernels to carry out the computation on modern supercomputer nodes. Even if they are tailored to a single application, they might also be of interest for applications that share a similar design pattern. The experimental results show how these optimizations can increase kernel performance by 2×, respectively, up to 2.2× in CUDA and up to 1.9×, in SYCL. Moreover, the reported speedup can be achieved with the best-proposed parameterization, as shown by the data we collected and reported in this manuscript.

Keywords SYCL · CUDA · Parallel programming · Virtual screening · HPC · Performance · Optimization · GPU · Batch computation · Molecular docking

1 Introduction

Drug discovery aims to find a small molecule that has a beneficial effect on a target disease. It involves in vitro and in vivo stages that increase costs and duration and limit the number of candidates evaluated. Recent studies have shown that we increase the likelihood of finding a drug by introducing a in silico stage that selects which molecules to test in vitro [1, 16]. In this stage, we use *virtual screening*

Extended author information available on the last page of the article

Published online: 02 February 2024

software to estimate the strength of interaction between a drug candidate, a small molecule called *ligand*, and the target protein representing the disease. We can use this value to rank a chemical library of many ligands and forward only the most promising ones to the following stages of drug discovery.

It is possible to create a chemical library by simulating known chemical reactions. This method gives us access to a vast chemical space. Therefore, the size of the chemical library is limited only by the computational resources available for virtual screening. For this reason, supercomputers are the target system for a virtual screening campaign [6, 8]. If we look at the node architectures of the fastest according to the TOP500[1] list, we can see that they rely heavily on accelerators to increase their throughput. Although the accelerators are typically GPUs, they are from different vendors and usually have other native programming languages. In the context of urgent computing [14], where we want to reduce the social and economic impact of a pandemic, virtual screening software needs to benefit from all available computing resources. Due to the increasing cost of porting, testing, and maintaining different implementations, there is a growing interest in code portability. When targeting an HPC system, application performance is the key metric we want to improve. For this reason, much effort is put into analyzing and improving the computational kernels. It is common to find highly optimized libraries that solve well-known problems, for example, BLAS [2] for linear algebra computations. In addition, we can use various approaches to automatically tune kernel parameters to improve computational efficiency further [25] for the target architecture or workload.

Instead of focusing on the computational kernel, the main goal of this paper is to show how Out-of-kernel optimizations significantly impact an application's performance. As a demonstration, we applied them to LiGen, an extreme-scale virtual screening application [6]. In particular, we analyze how rearranging the kernel input data using architecture, kernel, and input features to execute them out-of-order can increase computational efficiency. Experimental results on a current HPC node show that this Out-of-kernel optimization can double the application's performance. Moreover, since LiGen has a CUDA and SYCL implementation of the computational kernels, we measured an advantage with both, showing that the approach is portable. Although the proposed optimizations and experimental results apply to LiGen, they might be of interest to other applications with a similar design.

In Sect. 2, we introduce the virtual screening problem and the LiGen application as a case study, while in Sect. 3, we discuss related work. Section 4 describes the proposed Out-of-kernel optimizations, along with their implementation for both SYCL 2020 and CUDA. The experimental results evaluating the approach are described in Sect. 5. Finally, Sect. 7 concludes the paper.

[1] https://www.top500.org/.

2 Background

This section introduces virtual screening in drug discovery and its use for HPC systems. We also give an overview of the GPU architecture and focus on the LiGen algorithm.

2.1 Virtual screening for drug discovery

Drug discovery aims to find effective drug candidates against a disease. Broadly speaking, drug candidates are small molecules called ligands, and one or more proteins may represent the disease. Based on domain knowledge, we can expect a beneficial effect if we find a molecule that has a strong interaction with target proteins. Virtual screening aims to evaluate the interaction strength of a chemical library composed of many ligands. This helps domain experts choose which ones to test in vitro. This is an embarrassingly parallel task, given the independence of each protein-ligand pair.

Virtual screening is a well-known problem in the literature, where many approaches have been proposed, implemented, and evaluated [19, 30]. To evaluate a protein-ligand pair, we have to perform three tasks. Since the ligand is much smaller than a protein in terms of the number of atoms, the first task is to identify one or more regions of the protein where we would like to place the ligand (*docking site*). The second task aims to estimate the 3D displacement of the ligand's atoms when it interacts with the target docking site: It *docks* the ligand into the protein. In addition to the rigid movement of the ligand, it is possible to change its shape. A subset of ligand bonds, called rotatable bonds, split the molecular atoms into two disjoint sets, allowing one set of atoms to rotate around the bond axis. This flexibility changes the geometric shape of the molecule, producing different *poses*, but does not alter its chemical and physical properties. Finally, the third task evaluates the strength of the interaction between the docked ligand and the docking site. This paper will focus on the second task: molecular docking.

2.2 HPC for virtual screening

Improvements in computing power and computational efficiency have made it possible to explore a large chemical space simultaneously and with greater accuracy [8]. In the context of urgent computing, [14], there are efforts to redesign existing software for scaling out and to use accelerators of modern heterogeneous HPC nodes. The goal is to achieve a significant performance improvement [6, 8, 17, 29]. The supercomputing centers extensively use external offloading accelerators to speed up computation. Indeed, the data parallelism available in virtual screening tasks is well suited to the architecture of GPUs. Examples from the literature (e.g., AutoDock-Vina [24], PLANTS [11], and LiGen [6]) have already demonstrated over 25× speedup after porting to GPUs.

Table 1 CUDA and SYCL terminology mapping

CUDA	SYCL
Streaming multiprocessor	Compute unit
Block	Work-group
Warp	Sub-group
Thread	Work-item

HPC infrastructure has already been used to perform molecular docking simulation for virtual screening [7], and it has already been proven successful in designing a drug against avian influenza viruses [13].

2.3 GPU architectures

NVIDIA was the first GPU vendor to recognize the benefits of offloading HPC computing onto the GPU. In the early 2000 s, NVIDIA introduced CUDA (Compute Unified Device Architecture). This prompted many developers to add CUDA support to their applications, including LiGen [26].

NVIDIA's GPUs consist of Streaming Multiprocessors (SMs). All SMs can access the same global memory and exchange data with the CPU's RAM. From a hardware perspective, each SM has a user-programmable cache called shared memory, a set of registers, and the hardware that performs the computation and memory operations. Each SM uses a SIMD, single instruction multiple data [9], scheme where a bundle of hardware threads called *warp* execute the same instruction on different data. Each SM can have a relatively high number of warps ready to execute, named active warps in CUDA jargon. Recent NVIDIA HPC architectures use 4 hardware scheduler for each SM. At each cycle, the latter selects which warp to run. The number of active warps depends on the kernel's hardware requirements and the GPU hardware availability. This way, it is possible to hide stalls due to memory access transparently for the programmer.

From a programming language perspective, it is possible to partition the computation into blocks. Each block consists of at least one warp. We have the guarantee that all warps belonging to the same block are executed by the same SM. Therefore, efficient synchronization is possible, and we can program the shared memory to avoid accessing the global memory as much as possible. All the blocks that make up the whole computation are called a grid.

The developers are responsible for choosing the number of threads per block and the number of blocks per grid. Although both are constrained by the algorithm and the input data size, they can usually be tuned to improve overall performance.

Implementing the computational kernels using SYCL is one approach to achieving functional portability. SYCL stands for Single-source C++ Heterogeneous Programming for OpenCL. In particular, it provides an abstraction using an offloading architecture abstraction. It allows everyone to write the implementation code once, using standard C++, and then each interested vendor will provide their backend targeting their accelerated architecture. LiGen has ported the CUDA kernels using

SYCL [4], in the context of the LIGATE European Project [20]. The SYCL terminology differs from CUDA but can be easily mapped using Table 1. In the following, we will use CUDA terminology as a reference.

2.4 Case study: LiGen

This paper aims to show how Out-of-kernel optimization can improve computational efficiency, tailoring them on LiGen, an application for extreme-scale virtual screening [6].

Algorithm 1 LiGen's Docking Algorithm

```
1:  for ligand in ligands do
2:      for docking_site in docking_sites do
3:          generate_init_pose(ligand.rotatable_bonds)
4:          align_ligand(ligand.atoms)
5:          for repetitions do
6:              for rotatable_bonds do
7:                  optimize_rotatable_bonds(ligand.atoms)
8:              end for
9:              refine_pose
10:         end for
11:         check_bumping(ligand.atoms)
12:         check_in_pocket(ligand.atoms)
13:     end for
14: end for
```

In Algorithm 1, we have described the LiGen docking algorithm. The inputs to this algorithm are the docking sites of the protein and the ligands from the chemical space to analyze. The algorithm docks each ligand to each docking site of the target protein. The algorithm outputs the list of poses that should be scored in the third task of virtual screening. In this work, we focus on the docking kernel because it is the most demanding one in terms of hardware requirements and computation effort. It accounts for 90% of the execution time.

The LiGen docking algorithm performs a gradient descent with multiple restarts [21]. This docking procedure is applied to all the target proteins' docking sites. Algorithm 1 describes the algorithm in more detail. An initial pose is generated using the internal flexibility provided by the rotatable bonds at the start of the iterations (line 3). After this translation, we use rigid rotation to find the best alignment in the docking site (line 4). Then, the shape of the ligand is optimized and refined based on the number of repetitions and rotatable bonds (lines 5 to 10). Finally, the algorithm checks that the pose does not collide with the protein (line 11) and is within the docking site (line 12). From the functions described in the algorithm, we notice that the computational complexity increases linearly with the number of rotatable bonds and atoms in the ligand [6]. Additionally, the evaluation of ligands in a docking site can be done in parallel without waiting for the result of other computations.

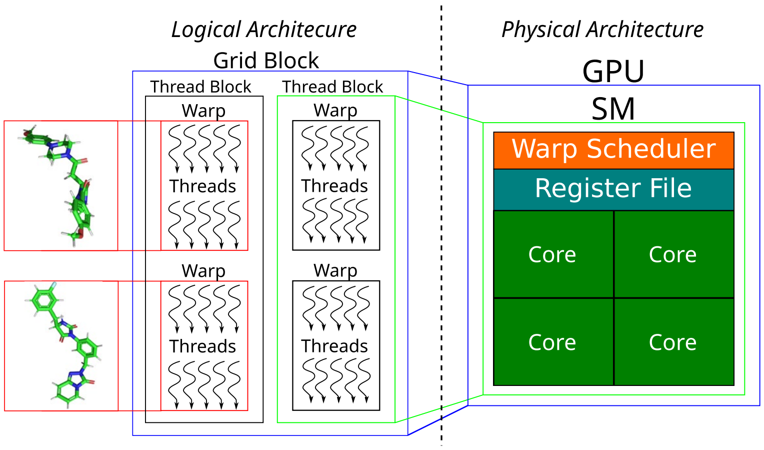


Fig. 1 The mapping of ligands' computation on the GPU logical and physical architecture

From a kernel implementation perspective, two main ways exist to take advantage of GPU parallelism in a virtual screening application. The most common is to spread the ligand-protein evaluation across the GPU to reduce the execution time as much as possible, which also increases the application throughput. We refer to this approach in literature as the latency one [12]. The AutoDock GPU and LiGen developers implemented this approach for an extreme virtual screening campaign against SARS-CoV-2 [6, 8]. Since the computation of a single ligand is independent of the others, we can hinge on data parallelism. The idea is to collect input data in a batch and then execute the whole batch on the GPU using a few threads to compute each input. Even if the computation time of a single ligand increases, the throughput might be higher since we are computing more input in parallel. Whether this batch approach yields a higher throughput than the latency one depends on the application. In the case of LiGen, it provides a 5× speedup with respect to the latency implementation [26]. This paper focuses on how we can manage the stream of ligands by reordering, packing, and organizing them. It also discusses how this affects the application's throughput. Rather than comparing the differences in the kernel implementations, as discussed in [14], the paper emphasizes the importance of efficient ligand management.

From an implementation point of view, LiGen uses multiple CPU threads, called *workers*, to launch the computational kernels. It uses a double buffering technique to hide data transfers to and from the device. When a worker receives a batch of ligands to compute, called a bucket, it starts copying the data to the first available buffer on the GPU. Then, it waits for the GPU to become available and executes the computation using the whole GPU. Finally, it copies the results to CPU memory and releases the buffer for another worker. It is possible to use this approach for any available GPU in the system. Figure 1 provides an overview of how the ligand computation is mapped on the CUDA abstraction. One warp carries out the computation of a single ligand. In particular, their atoms are computed in parallel by different GPU threads. This paper will focus on NVIDIA GPUs, using both CUDA and SYCL.

3 State of the art

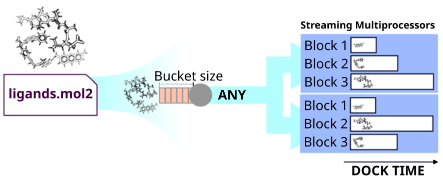
When we focus on the application level, improving the computation efficiency by executing a batch of input together has been successfully applied in various fields [26, 28, 31]. This trend is even more common in machine learning for accelerating training and inference [15, 18]. However, they target a more dynamic environment, considering the batch size as a tuning parameter [3]. This paper focuses on a classic HPC context, where the application and the architecture are known. For this reason, we can analyze in more detail how to hinge on these measures to optimize the computation.

When focusing on the GPU kernels, there are three main factors that we need to consider: data movement, thread divergence, and occupancy [12]. The latter is the ratio of active warps on the SM to the maximum number of warps that can run on an SM.[2]

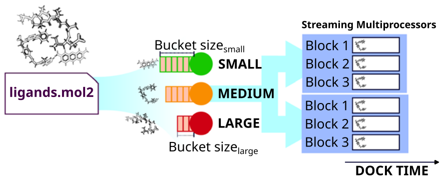
The GPU's memory hierarchy has three main layers: global memory, shared memory/caches, and registers. As a general rule, it is more efficient to use the shared memory and the registers, as they are closer to the GPU's computing resources[3] [5]. Moreover, the SIMD nature of GPU implies that control flow instructions may lead to thread divergence, decreasing computation efficiency inside a warp. For this reason, modern approaches borrowed from multi-core systems battle proven techniques [27] and enhanced them for GP/GPU systems [5]. These analyses are critical to identifying kernel bottlenecks and seizing optimization opportunities [10, 22]. However, this work focuses on Out-of-kernel optimizations and their impact on the application throughput. While the memory access pattern and thread prediction are tied to the kernel, we aim to increase the application throughput by improving occupancy. The kernel implementation using a batched design has been investigated in the literature [28, 29], also in an application domain similar to LiGen. However, these works focus more on in-kernel optimizations [10], while Out-of-kernel optimizations were either left to a static tuning phase or employed using a fixed strategy, not considering the target architecture. In this manuscript, we focus on Out-of-kernel methods to develop a more portable methodology. This follows a comparison between GPU-centered computation approaches, better detailed in a previous work [26]. Usually, developers expose software knobs to tailor the kernel to the execution environment. In literature, several autotuners can improve the application performance, leveraging these knobs [23, 25]. However, they target kernel-related parameters, such as the number of CUDA threads per block. This paper aims to show how input features, kernel, and architecture features can be used to rearrange the kernel input to improve the application performance.

[2] https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.html.

[3] https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html.



(a) Single input cluster



(b) Three input clusters

Fig. 2 Overview of how input ligands are mapped in different input clusters, according to their features, to enable out-of-order input execution. The goal is to balance the kernel time in the Streaming Multiprocessors

4 Out-of-kernel optimization

For embarrassing parallel applications, it can be beneficial to bundle input in a bucket and execute them in a batch fashion. This holds true for LiGen [26]. While the batched approach is fairly common and analyzed in literature [12], in this paper, we focus on how we bundle ligands together and their impact on performance. In particular, it must answer two different design choices: (i) which inputs to bundle into a batch, and (ii) its size. In the LiGen context, we want to optimize how to bundle ligands into a bucket. To increase code portability, we evaluated these Out-of-kernel optimizations for both SYCL and CUDA implementations. In particular, they use ligand characteristics, such as the number of atoms and rotatable bonds, to bundle them into different clusters. Then, for each cluster of ligands, we compute how many ligands a bucket should store using GPU hardware characteristics and kernel hardware requirements. In the remainder of this section, we will discuss implementation details.

4.1 Selecting the number of input clusters

In the batch approach, we hinge on the GPU parallelism by computing a bucket of ligands at the same time. This means the entire GPU is dedicated to the virtual screen of all the ligands in an input batch at any given time. The most straightforward implementation bundles all ligands in the same bucket, processing them without altering their order. Since the docking complexity depends on input features, the time required to dock all the ligands in a batch can differ, decreasing the computation efficiency. For this reason, we use the input features related to the application performance to cluster the ligands in different buckets. Figure 2 provides a schematic overview of this Out-of-kernel optimizations. The main goal that we want to achieve is to execute the input ligand out-of-order, processing together the ones that have a similar execution time, improving computation efficiency [12].

Looking at the algorithm complexity reported in Algorithm 1, we can see a linear dependency on the number of atoms and rotatable bonds. From the kernel implementation perspective, LiGen uses the warp threads to run parallel SIMD computations on the atoms. However, we need to process the rotatable bonds sequentially to preserve the ligand geometry. For this reason, ligands within a bucket must have similar numbers of atoms and fragments to have a similar execution time, balancing the computation. These two input features can be considered orthogonal as the number of atoms and fragments are loosely related.

A Cartesian product between the maximum number of atoms and the maximum number of fragments for ligands within that cluster defines the number of clusters. When considering the number of atoms, we used a multiple of the warp size to generate different clusters, i.e., with $32 \times n$ atoms with $n > 0$. For the number of rotatable bonds, we used non-uniform partitioning. We prefer a more fine-grained resolution toward the lower number of fragments since they lead to a higher relative difference. For example, we can bundle ligands with 0, 1, 2, 5, and 12 rotatable bonds in the same bucket. Section 5.1 reports the impact on performance when we change the number of clusters.

4.2 Selecting the bucket size

Since each ligand-protein pair can be computed independently, the bucket size is a parameter that we can tune. Ideally, we want a bucket with the number of ligands that maximizes computational efficiency, i.e., that uses the most GPU computational resources for the longest time. To achieve this goal, we must calculate how many ligands we can process in parallel on the GPU. In the remainder of this document, we will refer to this number as l. We can then use the largest multiple of l that fits into the GPU's memory as the bucket size.

Since LiGen uses one warp to compute a single ligand, we can use the number of CUDA threads t in a CUDA block to calculate the number of ligands in a CUDA block. Usually, t is a tuning parameter, but we set $t = 32$ since we focus on Out-of-kernel parameters in this paper. Thus, we compute a single ligand in each CUDA block.

The maximum number of active CUDA blocks depends on GPU hardware properties and kernel hardware requirements regarding registers and shared memory. We can use the CUDA API[4] to query how many blocks b can be run on the same Streaming Multiprocessors (SM) for any given kernel. In the LiGen context, we consider the kernel that docks a ligand since it is the most demanding one in terms of hardware requirements. Therefore, we can compute the bucket size l on the fly as follows:

$$l = b \times \mathrm{SM} \times \frac{t}{ws} \tag{1}$$

where *SM* is the number of SMs available on the GPU, and *ws* is the warp size.

Since one of the paper's goals is to have this optimization portable across a large set of GPUs, we adopted a similar approach within an SYCL porting of LiGen. The porting of the previous calculation on SYCL 2020[5] uses kernel bundle information from the standard. In particular, we can calculate the number of ligands l in a bucket as follows:

[4] CUDA function to query the number of active blocks on an SM for the given kernel `cudaOccupancyMaxActiveBlocksPerMultiprocessor`.

[5] https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html.

$$l = \frac{wgs}{t} \times \mathrm{CU} \times \frac{t}{sgs} \tag{2}$$

where *wgs* is the maximum number of work items we can run on a single Compute Unit (CU), given the target kernel. *CU* is the number of CUs, and *sgs* is the maximum subgroup size. Finally, *t* is the number of work-items in a work-group. In CUDA jargon, the work-items are CUDA threads, the CU is an SM, the work-group is a CUDA block, and the maximum sub-group size is the warp size. Although we can simplify *t*, we have written Eq. 2 to match the terms in Eq. 1. We can query the *wgs*, *sgs*, and *CU* values using the SYCL 2020 API. In particular, *wgs* is the property `kernel_device_specific::work_group_size`. The *t* parameter has the same value as in the CUDA implementation.

Unfortunately, not all SYCL compilers implement the entire standard yet. To overcome this problem, we need to calculate *wgs* by manually examining the compiled kernel. We look for its hardware requirements and compare them to the GPU resources. Due to differences in GPU architectures, especially when they are manufactured by different vendors, we cannot provide a one-fits-all equation to compute this term. However, since the dock kernel in LiGen is register bound, we set $t = 32$, and in this paper, we are targeting the NVIDIA A100 GPUs; we can calculate the number of active blocks as follows:

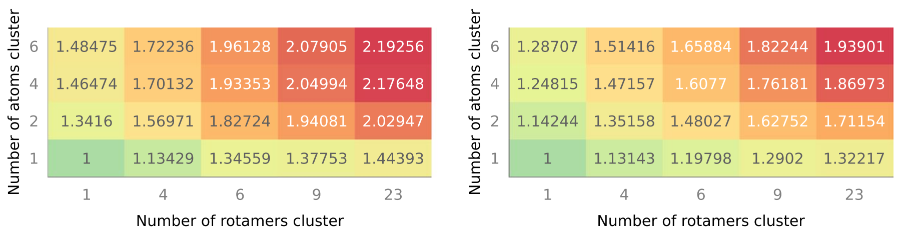
$$\frac{wgs}{t} = \left\lfloor \frac{maxr}{\lceil kr * ws \rceil_{256}} \right\rfloor_{4} \tag{3}$$

where *kr* is the number of registers used by a kernel implementation and *maxr* is the maximum number of registers a CUDA thread can use. Since our GPU does not allocate resources linearly, we must consider that it allocates blocks of 256 registers and that the number of active warps in each SM should be a multiple of 4. As the SYCL compilers mature, Eq. 3 will become obsolete as *wgs* can be retrieved at runtime using the standard SYCL API.

From a LiGen implementation point of view, the dock kernel has a non-type template parameter representing the maximum number of atoms it can process. This implementation choice allows the compiler to optimize the kernel for a specific number of atoms. However, it also changes the number of hardware registers used by the kernel. Therefore, depending on the maximum number of atoms, each ligand cluster may have a different *l* value.

5 Experimental results

This section evaluates the proposed Out-of-kernel optimizations' impact on the application performance. Therefore, in Sect. 5.2, we evaluate the impact on clustering ligands in different buckets according to input features, while in Sect. 5.3, we evaluate the impact of using kernel and GPU properties to size each bucket.



(a) CUDA implementation (b) SYCL implementation

Fig. 3 Application throughput speed-up by varying the number of clusters that target both the number of atoms and rotatable bonds in a ligand

5.1 Experimental setup

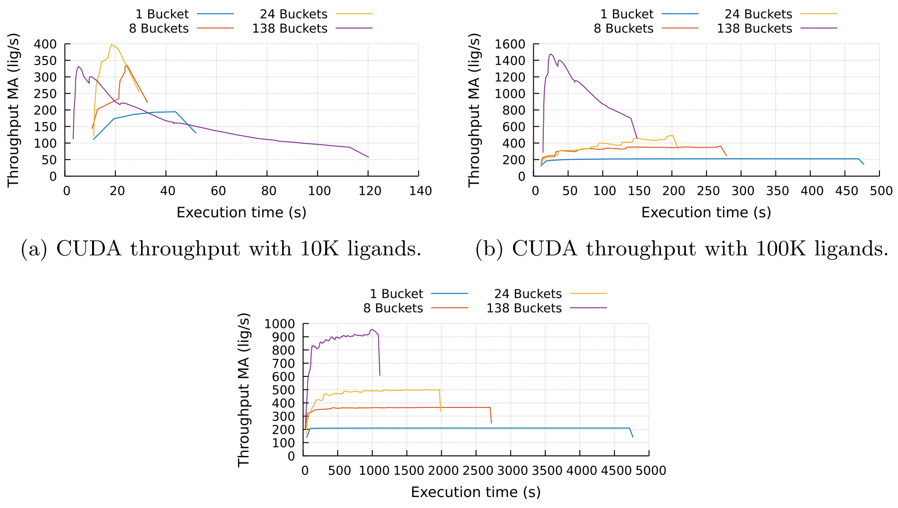
We run the experiment using a modern HPC compute node from the Karolina supercomputer at IT4I. A compute node uses two AMD EPYC 7763 with 64 cores paired with 1024 GB of RAM and eight NVIDIA A100 with 40 GB of VRAM each. We used CUDA 11.7 with GCC 11.3.0 as a software stack for compiling the CUDA kernels. We used Intel oneAPI 2023 as an SYCL compiler.

Since this is a typical HPC batch job, we want to evaluate the impact on the application throughput in terms of computed ligands per second. Unless stated otherwise, we report the average throughput, computed as the number of processed molecules divided by the elapsed wall time.

5.2 Measuring the clusterization impact

This experiment aims to measure how clustering the input into different buckets, according to ligand features, can affect the application's performance. We achieve this goal by performing a virtual screening campaign on a dataset of heterogeneous ligands. The input dataset contains 10 million different ligands with a number of atoms ranging from 20 to 120, having 0 to 20 rotatable bonds. In this experiment, we change the number of clusters for both input features. We set the bucket size for each cluster equal to l, as computed using Eqs. 1 and 2, according to the target implementation. The change in application performance is only due to how we bundle the ligands together in a batch.

Figure 3 shows the experiment results using CUDA and SYCL. On the x-axis, we report the number of clusters that we use to group the ligands according to the number of rotatable bonds. On the y-axis, we group them according to the number of atoms. The value reported in the heatmap is the normalized throughput achieved using as baseline the case where we bundle all ligands in the same batch, reported in the lower left cell. The upper right cell instead represents the finest-grained clustering, where we split the range of rotatable bonds into 23 clusters while we split the range of atoms into 6 clusters. In total, we divided the input ligands into 138 different clusters according to their input features.



(a) CUDA throughput with 10K ligands.

(b) CUDA throughput with 100K ligands.

(c) CUDA throughput with 1M ligands.

Fig. 4 LiGen CUDA throughput Moving Average, upon using different numbers of clusters

The application throughput shows how it is possible to double performance by using the highest number of clusters. In addition, the trend might suggest that increasing the number of clusters could further improve performance. However, when we increase the number of clusters, we also spread the input dataset over more buckets. Therefore, at the end of the computation, we will have a larger number of half-empty buckets. This limitation is not problematic for large virtual screening campaigns, as we have to evaluate a significant amount of ligands, but it becomes relevant for everyday use.

To measure this limitation, we performed a virtual screening by varying the number of clusters and the number of ligands in the input dataset. Figure 4 shows the execution trace of LiGen. On the x-axis, we plot the execution time, while on the y-axis, we plot the application's throughput over a moving observation window. Each line in the plot is an execution using 5 clusters for the number of atoms and 1, 8, 24, and 138 clusters for the number of rotatable bonds. We use LiGen to virtually screen a chemical library with a different number of ligands in each plot.

The experimental results show that the throughput is not constant, but we have a steep increase at the beginning and a steep decrease at the end. When we consider a large dataset, the effect of these two transients becomes negligible. However, when we use smaller datasets, it becomes more and more important. In particular, when we use 10K molecules (Fig. 4a), the configuration using more clusters becomes slower.

For this reason, the optimal number of clusters we want to use depends on the actual use case. If we target an extreme-scale virtual screening campaign [6], it is better to use many clusters. For everyday use, however, using a more coarse-grained classification of the input is better. We also need to consider that if we increase the

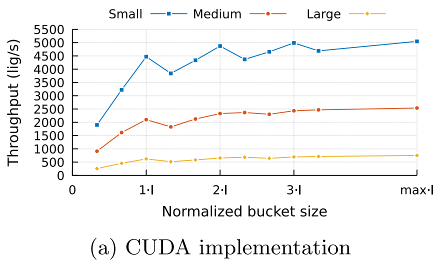


Fig. 5 LiGen average throughput by varying the number of ligands in a bucket for three classes of ligands

number of clusters, we must store more buckets in temporary storage, increasing the memory footprint.

5.3 Measuring the bucket size impact

We must choose their size when we bundle inputs together to hinge on data parallelism to use GPU parallelism. This section aims to show the impact of using the proposed methodology for selecting the number of inputs based on kernel and GPU features. In particular, we use Eq. 1 or Eq. 2 to find the minimum number of ligands we need to fill the GPU, called l. We can then choose its multiple according to the memory available on the host or device side.

Figure 5 shows the throughput of the application by varying the bucket size, using both the CUDA (Fig. 5a) and SYCL (Fig. 5b) implementations. To reduce the influence of different features, such as the number of rotatable bonds, we focus on three ligands: “small,” “medium,” and “large.” They have 17, 53 and 99 atoms, respectively. We measured a stable throughput by replicating each ligand 10 million times and docking the ligands into twelve docking sites of a target protein. Each line in the graph represents a different data set. The x-axis represents the number of ligands in each bucket. We normalize the size using l, which is the value suggested by Eq. 1 or Eq. 2, as reported in Table 2. We also evaluate application performance using the maximum number of ligands that fit into GPU memory, labeled “max$\cdot l$.”

The experimental results show two phases: before and after the value suggested by the equations. Before this value, the application throughput increases with the number of ligands. This is due to the increasing number of active warps that the SM scheduler can use to hide computational stalls. After this value, we have a drop in performance as we are processing the same ligand, and its computation will take a similar amount of time. The throughput increases as more active warps are created, repeating the pattern. The pattern is more evident for the SYCL and CUDA implementations with small ligands. For medium and large ligands, the change in throughput is smoother. This means that we choose how many and which ligands are evaluated on the GPU for each wave, but the GPU schedules the active warps.

To evaluate the impact of this Out-of-kernel parameter in a more realistic scenario, we performed the same experiment using the heterogeneous dataset described

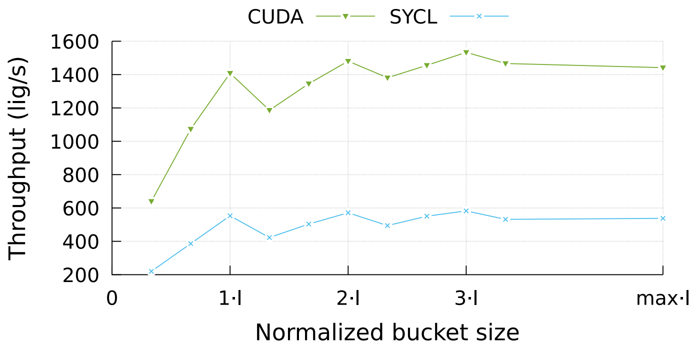


Fig. 6 LiGen average throughput with a dataset of heterogeneous ligands by varying the number of ligands in a bucket

Table 2 Comparison between the CUDA and SYCL kernel implementations, by varying the maximum number of atoms, on an NVIDIA A100 graphics card compiling with CUDA 11.7 and oneAPI 2023

Num atoms	CUDA		SYCL	
	Register	Bucket Size	Register	Bucket Size
32	102	1728	158	1296
64	103	1728	176	864
96	98	1728	176	864
128	102	1728	178	864
160	112	1728	176	864
192	124	1728	182	864

in Sect. 5.2. We bundle the input ligands into 128 different clusters to measure performance. Figure 6 shows the average throughput of the CUDA and SYCL implementations. While both follow the same trend, confirming the impact of Out-of-kernel optimization, the difference in throughput is larger than expected. We analyzed the docking kernel as it accounts for over 90% of the total execution time. Comparing their execution times, SYCL was only 12% slower than CUDA, while the gap in application throughput is more significant. This is due to the higher register pressure of the SYCL implementation, as reported in Table 2, which reduces the number of active blocks on the GPU. This means that SYCL can process fewer ligands in parallel, reducing its throughput. However, it can provide a significant speed-up over the CPU version, achieving a throughput of less than 50 ligands/s using all available cores.

The bucket size reported in Table 2 has been calculated using Eq. 1 for CUDA and Eq. 2 for SYCL. This number is subject to change depending on several factors. These factors include the template parameter used by the docking kernel, which affects the docking kernel register pressure used by the compiler, the docking kernel implementation, and the target offloading architecture. However, the bucket size is a multiple of 432 ligands on both implementations. The latter is the minimum increment in the size of the buckets due only to architectural characteristics. We can compute this increment step by multiplying the number of SMs per GPU by the number of hardware schedulers per SM. With our reference GPU, NVIDIA A100, we have 108 SMs and four hardware schedulers, which results in a bucket size step of 432.

6 Discussion

The main objective of the proposed work is to show how we can improve application throughput among different GPU architectures by using Out-of-kernel optimizations. In particular, we used input features, kernel requirements, and architectural features to rearrange the kernel inputs. Thus, we executed them out of order to improve the computation efficiency. We also wanted to demonstrate how these optimizations are not strictly related to the language used for the kernel description (i.e., CUDA or SYCL). Thus, Sect. 5.2 evaluates the impact of clusterization with SYCL and CUDA to demonstrate that they have a similar trend. Section 5.3 evaluates how the performance changes by varying the number of ligands to show how the trend is coherent between CUDA and SYCL.

Performing a direct comparison between CUDA and SYCL versions of the target application was out of the scope of this paper. However, we can draw a few general comments derived from our experience with this work. Different trade-offs exist when considering whether to use a CUDA or a SYCL implementation. CUDA enables better low-level access to the resources, thus granting the possibility to use the underlying hardware efficiently. SYCL has a higher level abstraction, supporting GPUs from different vendors. Their performances are not comparable on the target NVIDIA A100-based system used for the validation. This is mostly due to the different register pressure, as reported in Table 2, that limits the exploitable parallelism.

From the implementation point of view, the CUDA bucket dimensions are calculated by querying the CUDA runtime, making this information available for all the NVIDIA cards. On the other hand, SYCL is a cross-platform programming model with multiple implementations, which may support different (and partial) versions of the standard. Thus, to compute the bucket dimensions, we need to consider these differences, selecting the correct function call based on the compiler version.

7 Conclusions

With the increasing availability of heterogeneous nodes in HPC systems, there is a trend for re-designing scientific applications to hinge on accelerator computation power, typically GPUs. To achieve this goal, an application can follow two different strategies. On the one hand, it can spread the computation across the GPU, lowering the execution time and thus increasing its throughput. On the other hand, it can hinge on data parallelism by computing input batches.

This paper focuses on Out-of-kernel optimizations that application developers can use in batch approaches. In particular, we can use input features to group inputs that are expected to have similar execution times, while we can use the kernel and GPU features to compute the batch size.

We implemented this Out-of-kernel optimization on LiGen, an extreme-scale virtual screening application that can be scaled up an entire supercomputer. The experimental result shows that the data preparation phase can improve computational efficiency, doubling the application's throughput. This performance gain is consistent with both SYCL and CUDA implementations. However, a large or homogeneous input set is required to achieve this efficiency level. The latter is not a limitation since we need maximum efficiency only for extreme-scale virtual screening campaigns with a very large number of ligands. Even though the result refers to LiGen, the reported analysis may be of interest to applications with a similar pattern.

Acknowledgements We would like to thank Mauro Bisson and Massimiliano Fatica from NVIDIA Corp. for their support on the CUDA optimization. This project has received funding from EuroHPC-JU - the European High-Performance Computing Joint Undertaking - under grant agreement No 956137. The JU receives support from the European Union's Horizon 2020 research and innovation program and Italy, Sweden, Austria, Czech Republic, and Switzerland. We acknowledge EuroHPC Joint Undertaking for awarding us access to Karolina at IT4Innovations, Czech Republic (EHPC-DEV-2021D02-049).

Author Contributions G.A., D.G., E.V., and G.P. conceived the idea and wrote the paper, along with the data collection and analysis, and the review of the paper. B.C. and L.C. provided SYCL support for the data collection and analysis, as long as review for the SYCL content in the paper. A.B. contributed to the revision and critical analysis of the paper's content.

Funding Open access funding provided by Politecnico di Milano within the CRUI-CARE Agreement. EuroHPC-JU with grant agreement No 956137 (LIGATE), EuroHPC-JU with grant EHPC-DEV-2021D02-049.

Availability of data and materials All the datasets used in the experiments use ligands from the MEDIATE dataset, which is publicly available https://mediate.exscalate4cov.eu/. We use a target protein taken from the RCSV PDB, with id 1CVU, which is publicly available https://www.rcsb.org/structure/1cvu.

Declarations

Conflict of interest All authors declare that they have no conflict of interest.

Ethical approval Not applicable.

References

1. Allegretti M, Cesta MC, Zippoli M et al (2022) Repurposing the estrogen receptor modulator raloxifene to treat SARS-CoV-2 infection. Cell Death Differ 29(1):156–166
2. Blackford LS, Petitet A, Pozo R et al (2002) An updated set of basic linear algebra subprograms (BLAS). ACM Trans Math Softw 28(2):135–151
3. Crankshaw D, Wang X, Zhou G, et al (2017) Clipper: a low-latency online prediction serving system. In: NSDI, pp 613–627

4. Crisci L, Salimi Beni M, Cosenza B, et al (2022) Towards a portable drug discovery pipeline with SYCL 2020. In: International workshop on OpenCL
5. Ding N, Williams S (2019) An instruction roofline model for gpus. In: 2019 IEEE/ACM performance modeling, benchmarking and simulation of high performance computer systems (PMBS), pp 7–18
6. Gadioli D, Vitali E, Ficarelli F, et al (2022) Exscalate: an extreme-scale virtual screening platform for drug discovery targeting polypharmacology to fight SARS-CoV-2. IEEE Transactions on Emerging Topics in Computing pp 1–12
7. Ge H, Wang Y, Li C et al (2013) Molecular dynamics-based virtual screening: accelerating the drug discovery process by high-performance computing. J Chem Inf Model 53(10):2757–2764
8. Glaser J, Vermaas JV, Rogers DM et al (2021) High-throughput virtual laboratory for drug discovery using massive datasets. Int J High Perform Comput Appl 35(5):452–468
9. Hassaballah M, Omran S, Mahdy YB (2008) A review of SIMD multimedia extensions and their usage in scientific and engineering applications. Comput J 51(6):630–649
10. Hijma P, Heldens S, Sclocco A et al (2023) Optimization techniques for GPU programming. ACM Comput Surv 55(11)
11. Korb O, Stützle T, Exner TE (2011) Accelerating molecular docking calculations using graphics processing units. J Chem Inf Model 51(4):865–876
12. Lemeire J, Cornelis JG, Segers L (2016) Microbenchmarks for GPU characteristics: the occupancy roofline and the pipeline model. In: 2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), pp 456–463
13. Liu T, Lu D, Zhang H et al (2016) Applying high-performance computing in drug discovery and molecular simulation. Natl Sci Rev 3(1):49–63
14. López N, Debbio LD, Baaden M, et al (2021) Lessons learned from urgent computing in Europe: tackling the COVID-19 pandemic. In: Proceedings of the National Academy of Sciences, vol 118, pp 46
15. Ma S, Belkin M (2019) Kernel machines that adapt to GPUS for effective large batch training. In: Talwalkar A, Smith V, Zaharia M (eds) Proceedings of Machine Learning and Systems, pp 360–373
16. Matter H, Sotriffer C (2011) Applications and success stories in virtual screening. Wiley, chap 12, pp 319–358
17. Murugan NA, Podobas A, Gadioli D, et al (2022) A review on parallel virtual screening softwares for high-performance computers. Pharmaceuticals 15(1)
18. Nabavinejad SM, Reda S, Ebrahimi M (2022) Coordinated batching and DVFS for DNN inference on GPU accelerators. IEEE Trans Parallel Distrib Syst 33(10):2496–2508
19. Pagadala NS, Syed K, Tuszynski J (2017) Software for molecular docking: a review. Biophys Rev 9(2):91–102
20. Palermo G, Accordi G, Gadioli D et al (2023) Tunable and portable extreme-scale drug discovery platform at exascale: the lIGATE approach. In: Proceedings of the 20th ACM International Conference on Computing Frontiers, pp 272–278
21. Ruder S (2017) An overview of gradient descent optimization algorithms
22. Ryoo S, Rodrigues CI, Stone SS et al (2008) Program optimization carving for GPU computing. J Parallel Distrib Comput 68(10):1389–1401
23. Sethia A, Mahlke S (2014) Equalizer: dynamic tuning of GPU resources for efficient execution. In: 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, pp 647–658
24. Tang S, Chen R, Lin M et al (2022) Accelerating autodock vina with GPUS. Molecules 27(9):3041
25. Tillmann M, Karcher T, Dachsbacher C, et al (2014) Application-independent autotuning for GPUS. In: Parallel Computing: Accelerating Computational Science and Engineering (CSE). IOS Press, pp 626–635
26. Vitali E, Ficarelli F, Bisson M, et al (2024) GPU-optimized approaches to molecular docking-based virtual screening in drug discovery: a comparative analysis. J Parallel Distrib Comput 186(4)
27. Williams S, Waterman A, Patterson D (2009) Roofline. Commun ACM 52(4):65–76
28. Wu D, Zhang F, Ao N, et al (2009) A batched GPU algorithm for set intersection. In: 2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks, pp 752–756
29. Yu Y, Cai C, Zhu Z, et al (2022) Uni-dock: a GPU-accelerated docking program enables ultra-large virtual screening. American Chemical Society (ACS)
30. Yuriev E, Holien J, Ramsland PA (2015) Improvements, trends, and new ideas in molecular docking: 2012–2013 in review. J Mol Recognit 28(10):581–604
31. Zhou G, Feng Y, Bo R et al (2017) GPU-accelerated batch-ACPF solution for n-1 static security analysis. IEEE Trans Smart Grid 8(3):1406–1416

Authors and Affiliations

Gianmarco Accordi[1] · **Davide Gadioli**[1] · **Emanele Vitali**[1,2] · **Luigi Crisci**[3] · **Biagio Cosenza**[3] · **Andrea Beccari**[4] · **Gianluca Palermo**[1]

✉ Gianmarco Accordi
gianmarco.accordi@polimi.it

✉ Davide Gadioli
davide.gadioli@polimi.it

Emanele Vitali
emanuele.vitali@csc.fi

Luigi Crisci
lcrisci@unisa.it

Biagio Cosenza
bcosenza@unisa.it

Andrea Beccari
andrea.beccari@dompe.com

Gianluca Palermo
gianluca.palermo@polimi.it

1 Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milan, Italy

2 CSC - IT Center for Science, Espoo, Finland

3 Dipartimento di Informatica, Università degli studi di Salerno, Salerno, Italy

4 Dompé Farmaceutici S.p.A, Naples, Italy