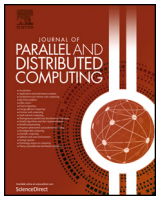
Journal of Parallel and Distributed Computing 186 (2024) 104819

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

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



GPU-optimized approaches to molecular docking-based virtual screening in drug discovery: A comparative analysis

Emanuele Vitali [a,c], Federico Ficarelli [b], Mauro Bisson [e], Davide Gadioli [c,*], Gianmarco Accordi [c], Massimiliano Fatica [e], Andrea R. Beccari [d], Gianluca Palermo [c]

[a] CSC - IT Center for Science, Espoo, Finland
[b] Dipartimento di Ingegneria dell'Energia Elettrica e dell'Informazione - Università di Bologna, Bologna, Italy
[c] Dipartimento di Elettronica, Informazione e Bioingegneria - Politecnico di Milano, Milano, Italy
[d] Dompé Farmaceutici S.p.A, Napoli, Italy
[e] NVIDIA Corp., Santa Clara, USA

ARTICLE INFO

Keywords:
High Performance Computing
High throughput virtual screening
GPU acceleration
CUDA
Batch

ABSTRACT

Finding a novel drug is a very long and complex procedure. Using computer simulations, it is possible to accelerate the preliminary phases by performing a virtual screening that filters a large set of drug candidates to a manageable number. This paper presents the implementations and comparative analysis of two GPU-optimized implementations of a virtual screening algorithm targeting novel GPU architectures. This work focuses on the analysis of parallel computation patterns and their mapping onto the target architecture. The first method adopts a traditional approach that spreads the computation for a single molecule across the entire GPU. The second uses a novel batched approach that exploits the parallel architecture of the GPU to evaluate more molecules in parallel. Experimental results showed a different behavior depending on the size of the database to be screened, either reaching a performance plateau sooner or having a more extended initial transient period to achieve a higher throughput (up to 5x), which is more suitable for extreme-scale virtual screening campaigns.

1. Introduction

Drug discovery is a long and costly process that aims at finding new drugs. Typically, this process involves several *in silico*, *in vitro* tasks (ranging from chemical design to toxicity analysis), and *in vivo* experiments. Virtual screening is one of these tasks, which has to be performed at the beginning of the drug discovery process in the exploratory research phase. This task aims at reducing the number of candidate drugs from billions of molecules to a number that can be managed with costly chemical experiments. Molecular docking represents but one stage of this step [2,19]. It aims to estimate a given molecule's three-dimensional pose, the *ligand*, when it interacts with the target protein. 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 molecular docking algorithm positions the *ligand* inside the *docking site* in the most suitable place. The algorithm must perform translation and rotation operations on the target *ligand*. This flexibility changes the geometric shape of the molecule, producing different *poses*, but does not alter its chemical and physical properties. Furthermore, it is possible to identify a subset of bonds – named *rotamers* or *rotatable bonds* – that split the *ligand* into two disjoint fragments when they are removed. These *rotamers* can be rotated without changing the chemical properties of the *ligand*. Therefore, the algorithm must also consider the different shapes of the *ligand* that can be generated from the rotation of all its *rotamers*.

An efficient implementation of the virtual screening phase has two positive effects. On one hand, it reduces the time to wait for the screening phase. On the other hand, it permits the enlargement of the input chemical space, thus increasing the number of molecules to be evaluated. While these benefits are clear, they become even more apparent when a pandemic breaks out, as was the case with the recent COVID-19 pandemic. Indeed, when the pandemic started, several efforts have been made to find a therapeutic cure for the SARS-CoV-2 infection.

* Corresponding author.
E-mail address: davide.gadioli@polimi.it (D. Gadioli).

https://doi.org/10.1016/j.jpdc.2023.104819
Received 3 August 2023; Received in revised form 6 December 2023; Accepted 12 December 2023
Available online 19 December 2023

Example in this direction are the COVID-19 HPC Consortium,[1] the EXSCALATE4CoV project,[2] and the current LIGATE project.[3] The workload in the case of screening a large set of molecules is embarrassingly parallel since each ligand-protein pair can be processed in parallel to the others. This makes the use of large supercomputer infrastructure the most suitable target [27,43] for urgent computation in the case of a pandemic, given the possibility to have a simple data splitting across the nodes with lighter synchronization for I/O accesses [22,40]. Similar thoughts can be done when considering resources within the node. In particular, current supercomputers are mainly accelerated with multiple GPU cards, and the workload can be further split for each card.

In this paper, we analyze two different GPU implementations of a high-throughput *in silico* virtual screening application, LiGen [1], to compare their behaviors given the different parallelization strategies. Despite the experimental results being related to the specific code implementation, within the paper, we focus more on analyzing the parallel computation patterns and their mapping on the target architectures. Both of these implementations target NVIDIA GPU and are written in CUDA. However, they have an orthogonal approach where the first implementation is a synchronous, latency-oriented one, while the second is an asynchronous implementation that uses a batched approach. In the first implementation, called *latency implementation* from now on, we exploit the GPU parallelism to shorten the computation time required to dock a single ligand by evaluating the different poses and different atoms in parallel. This is the classic approach for the acceleration of molecular docking applications, used in AutoDock [33], and in previous versions of LiGen [48]. In the second implementation, we approach the problem of parallelizing the computation from a different perspective: we exploit the GPU parallelism to evaluate several ligands in parallel, and a single warp of threads always evaluates a single ligand. The warp is a collection of threads, 32 in current implementations, that are executed simultaneously by a Streaming Multiprocessor, SM; therefore, this is considered the basic unit of execution on a GPU. For this reason, we will define this version as *batched implementation*.

The contributions of this work are:

- comparison between the latency and batch virtual screening implementation for drug discovery;
- demonstrates that while the latency version is the best solution for small-scale experiments, the batched version widely outperforms the other version for extreme-scale virtual screening campaigns;
- The batch approach performance benefits by rearranging the input data using architecture, kernel, and input features.

Although the analysis has been carried out on a single GPU, it can be generalized on a multi-GPU case since both implementations scale almost ideally in a virtual screening scenario. Multi-node optimizations are orthogonal to the current work and are widely described in a previous work [12]. The remainder of the document is organized as follows: Section 2 briefly describes the state of the art in the field and related approaches applied to virtual screening. Section 3 describes the target application and the two different implementations under analysis. Section 4 reports the experimental results obtained by the two implementations, highlighting the performance characteristics and limitations, together with a deep profiling analysis on the use of the resources. Finally, Section 5 concludes the paper.

2. State of the art

High throughput virtual screening has been widely applied in recent years during the early stage of drug discovery. Indeed, this helped find some novel drugs [3,21,34]. Several steps are required to perform a virtual screening campaign [13]; however, we will focus on the molecular docking step in this work.

Many pieces of software have been created during the latest years to this end, both open sources [6,26] and commercial [10,35]. There are two main approaches to the molecular docking step: the first is a deterministic approach, while the second favors a random-based approach. Random-based approaches use well-known techniques to create different poses of a *ligand* and measure their interactions with the protein *docking sites*. Examples of these are MolDock [41] and Gold [16] where genetic algorithms are used, or Glide [10], and MCDock [20] where the technique used is Monte Carlo simulation. However, this approach has a significant drawback since its results may not be entirely reproducible. This drawback may be a blocking issue for some pharmaceutical companies that refuse to start the expensive in-vitro and in-vivo phases without a reproducible result. For this reason, sometimes a deterministic approach is required. Examples of deterministic approaches are BIGGER [29], DOCK [6], LiGen [1], and Flexx [35]. These approaches use deterministic algorithms that can modify the shape of the ligands by leveraging their torsional bonds. Many molecular docking applications were born as single workstation applications; however, given the amount of complex elaboration that has to be performed, they quickly evolved into High-Performance Computing (HPC) applications. As we can see from this survey [5], different techniques were studied to improve the capabilities of this software and scale them to HPC machines. There are prominent approaches, such as scaling with MPI [52], and more complex solutions, such as developing ad hoc scripts to wrap the main kernel and deploy it to different nodes with different data [51]. In recent years, we have seen the rise of heterogeneous clusters in HPC, where several GPUs are used as accelerators next to the CPU. For this reason, some of these molecular docking applications have been modified in order to be able to exploit these co-processors [7,8,17,32,38,39]. In particular, MedusaDock [7] achieves a 1.54× overall speedup, and GeauxDock [8] has a 3.5× speedup thanks to the GPU porting. Other applications show better behavior on the GPU and have double digits speedup, such as PIPER [38] with a 17× speedup, AutoDock-Vina [39] with a 50× speedup, and PLANT [17], that reported a 60× speedup. The latest GPU porting of AutoDock [18,33,37] has been optimized for running on the Summit supercomputer [14] to support COVID-19-related research. A new Autodock development was recently released: Uni-Dock [50]. Uni-Dock increases the accuracy compared to the Autodock and VINA GPU versions, making the execution ten times faster thanks to batching. Uni-Dock tries to use heuristics, based on the type of architecture used, to execute a batch of inputs likely to fill the entire memory of the GPU. Other approaches have improved performance by using dedicated hardware for matrix computation. For example, Autodock has been accelerated using NVIDIA's Tensor Core [36]. Using this approach, they have achieved a 4-7× speedup in reduction operations, with an overall reduction of 27% in docking time. Much attention has been given to using HPC software on a cloud-available platform. An example using Autodock on clouds has been reported using Kubernetes and Apache Airflow [23]. This approach enables virtual screening campaigns on a more available cloud basis while taking advantage of heterogeneous platforms. In this paper, we focus on the efficient GPU porting of the LiGen application by describing and analyzing two different parallelization approaches considering the peculiarities of the target workload and GPU devices. LiGen is an MPI application that distributes the workload across different nodes of a supercomputer [11]; for its embarrassingly parallel nature, we can consider multi-node optimizations orthogonal to the current work. The target LiGen code in this *latency implementation* has been used for the largest virtual screening campaign ever run (> 70 billion ligands and 12 viral proteins) during the first wave of the COVID-19 pandemic [12]. A recent interesting parallel investigation on LiGen is about its performance portability across multiple GPU architectures and vendors, adopting different high-level languages

[1] https://covid19-hpc-consortium.org/.
[2] https://www.exscalate4cov.eu/.
[3] https://www.ligateproject.eu/.

Algorithm 1: LiGen virtual screening algorithm.

```
   Data: max_num_ligands
   Input: ligand_library, target
   Output: top_candidates
1  candidates ← ∅;
2  foreach ligand ← ligand_library do
3  |  candidates ← candidates ∪ dock(ligand, target);
4  end
5  return top_n(candidates, max_num_ligands)
```

Algorithm 2: LiGen dock algorithm.

```
   Data: num_restart
   Input: ligand, target
   Output: best_pose
1  poses ← ∅;
2  for i ← 0 to num_restart do
3  |  pose ← init_pose(ligand, i);
4  |  pose ← align(pose, target);
5  |  pose ← optimize(pose, target);
6  |  pose.validity ← is_valid(pose, target);
7  |  if pose.validity then
8  |  |  pose.score ← score(pose.atoms, target);
9  |  else
10 |  |  pose.score ← −∞;
11 |  end
12 |  poses ← poses ∪ {pose};
13 end
14 return max_score(poses)
```

[28]. This path is out of the scope of this paper, which focuses only on NVIDIA GPUs.

3. Application description

LiGen [1] is a molecular docking application designed to run on High-Performance Computers and adapted for extreme-scale virtual screening campaigns [12]. Algorithm 1 reports the pseudo-code for virtual screening an input ligand library against a target docking site. The output lists the ligands with the highest interaction strength with the target. The procedure is straightforward; we must dock each ligand from the input library to estimate its interaction strength using a scoring function. When we have more than one docking site, repeating the whole procedure for another target is possible, generating a different set of best candidates. Domain experts will combine the data to select a global set of candidates to test *in-vitro* (or further *in-silico*). Thus, we can focus on the single-target scenario without losing generality.

Algorithm 2 describes in more detail all the steps that LiGen uses to dock a ligand inside a target. The overall algorithm is a gradient descent with multiple restarts [30]. At each restart, we generate an initial pose (line 3) by rotating the ligand's rotamers using a heuristic that maximizes the distance among the iterations in the conformation space of the molecule. The gradient descent procedure is composed of two operations. The first considers the molecule to be a rigid body to align with the docking site (line 4). In contrast, the second uses the internal molecule flexibility to optimize its shape for the target and performs a local minimization (line 5). We use a geometric score to define the gradient that drives the docking. To select the most suitable pose, we need to re-score the poses using a scoring function that considers physical and chemical properties (line 8). To avoid useless computation, we do not compute the scores of molecules (line 10) that clash internally or with the protein (lines 6,7). Finally, among all the ligand's poses, we are interested only in the one that yields the highest score (line 14).

From the algorithm description, we can notice how a pose evaluation is independent of the others. We can generate many ligands by simulating known chemical reactions, making the virtual screening problem embarrassingly parallel. LiGen uses these properties to distribute the input ligand library across different nodes [12] and to offload the computation to GPUs. In this paper, we explore two strategies to implement the algorithm in CUDA that use drastically different design choices to hinge on hardware parallelism. In this section, we introduce the main concepts of the CUDA architecture and how the two implementations map the algorithm in its computation model.

3.1. CUDA architecture

Since most of the world's supercomputers make extensive use of GPUs, we target NVIDIA's GPUs to accelerate computation, and we use the CUDA language to exploit the maximum potential of the architecture.

GPUs implement a SIMT (single instruction multiple threads) architecture. In particular, NVIDIA's GPUs organize threads in warps, which are groups of 32 threads [45]. A warp of threads follows the same execution flow. Conditional branching can cause divergence, which introduces overhead because the instructions on the two paths are not executed in parallel. Full efficiency is, therefore, achieved when all threads in a warp agree on the execution path. In CUDA, *threads* are arranged hierarchically, whereby they are grouped into *blocks* and then launched in *grids* [45]. These *threads* and *blocks* may be grouped in a three-dimensional structure, allowing for efficient organization and execution of instructions. Different structures can be utilized to achieve maximum occupancy or maximum parallelism [46]. A SM is made of up to 4 dual-issue warp schedulers. This means that it will select potentially two instructions to be executed. As with threads, GPU memory is also organized in a hierarchy. All the data must be close to where they are needed. The NVIDIA GPU's memory is organized so that different memory is intended for different granularity accesses. There are three different levels of memory on the board. The main *global memory* can be accessed by each thread independently from its position in the block/grid. It allows communication between all threads and between the host and the GPU, but is the slowest. To get closer to the threads, one can preload data into the *shared memory*, which, as the name suggests, is shared by the threads of the same block. It is faster than *global memory*, but its size is limited per block so that it can limit the occupancy of the GPU. The closest type of memory to threads is *registers*, which are very few and are allocated to threads in a fixed amount. They are the fastest memory a thread can use, but since they are limited per SM, registers limit the number of concurrent threads that can run on an SM.

3.2. Latency implementation

The first implementation we will analyze is the *latency implementation*. This approach aims to keep a synchronous interface, where a single *ligand* is docked on the GPU in every host call to the dock function. This approach is the same as the previous implementation of LiGen [12,48] and allows us to focus only on the acceleration of the kernels without having to modify the whole application structure, thus purely following Algorithm 1 for screening a ligand library. On the GPU, we distribute the operation that we have to perform as much as possible, trying to make each kernel as parallel (and fast) as possible to execute (See Fig. 1). This approach is the most straightforward and traditional one, and it is the same followed by most of the GPU porting for molecular docking (e.g., AutoDock-GPU [18,33]).

Fig. 2 provides an overview of the approach in terms of the main parallelism exploited and execution phases. The idea is to parallel execute all the iterations for the loop at line 3 of Algorithm 2. To reach this goal, we must perform each step of the computation on all the poses as depicted in Fig. 2. The CUDA grid is set over the different ligand poses. We implemented all the steps using at least one kernel to increase the exposed parallelism. In this way, we can execute in parallel the internal loops required to carry out the computation. In particular, for the *init_pose* step, each CUDA thread updates the position of a single atom. For the *align* step, we use two kernels. The first one evaluates all the rigid transformations for all the poses in parallel, where

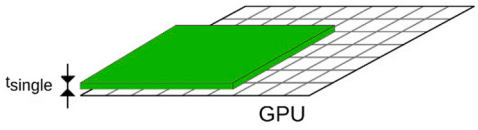


Fig. 1. Latency implementation GPU usage.

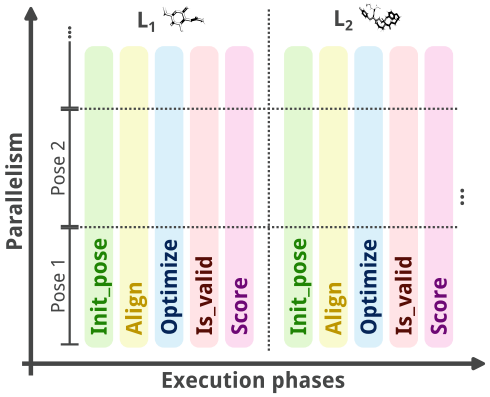


Fig. 2. Logical mapping on how the latency approach hinges on GPU parallelism to accelerate the execution time. Each step is implemented using at least one dedicated kernel.

each CUDA thread updates the atoms' displacement and computes the gradient value. The second kernel performs a reduction to find the optimal alignment for the ligand and updates the pose atoms displacement accordingly. In the *optimize* step, we need to evaluate each rotamer sequentially to preserve the ligand geometry. We use two kernels to evaluate a single rotamer using an approach similar to the *align* step. Besides rotating and computing the gradient value, the main difference is that each CUDA thread needs to check if the rotamer's angle leads to an internal clash. For the *is_valid* step, we use two different kernels to check whether there is a clash with the protein or an internal one. In both cases, the distance between each atom pair has to be calculated. To limit the computation effort, we perform an early escape when we detect a bump between atoms, thus determining an invalid pose.

By computing poses using the parallelism at grid level, these kernels have a very short execution time and aim at freeing the hardware resource for other kernels. To maximize the GPU utilization, we rely on a multi-threaded approach to instantiate several different kernels (on different streams). Every *ligand* will be tied to a host thread tied to an asynchronous queue (CUDA stream) and a reserved space in the GPU memory. The reservation of the space at the thread level instead of at the ligand level allows us to allocate and deallocate that memory only once in the thread's lifetime. This first optimization saves a lot of memory operations since this memory space is not linked to the docking of a single ligand but is linked to the application's lifetime. The drawback of this approach is that we need to allocate the *worst case* space, which must be known at compile time. This introduces a limitation on the maximum size of the processed ligands. However, this is not a real issue for the application since it can be changed at compile time. Moreover, some data structures (such as the one that represents the target pocket) can be shared among all the threads using the same GPU: this can be done since they are read-only data structures, not modified in the docking process. The access to the pocket does not follow a coalesced pattern, but the access point is given by the x, y, and z coordinates of the atom and, for this reason, has a random pattern. Random accesses in memory are a costly operation in GPU since they disable the coalesced access mechanism that allows for providing data to all the threads in a warp with a single read operation. However, there is a feature in CUDA that allows for improving the performance in these situations, which is the texture cache. Texture caches allow organizing data in 2D or 3D

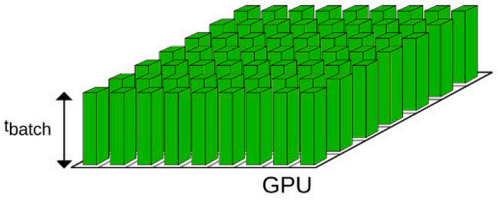


Fig. 3. Batched implementation GPU usage.

spaces and are optimized for semantic data locality. This means that accessing points in the space close to the previous ones is usually faster since they should already be cached. We expect rotations and translations in the 3D space will not place atoms "too far" across the different iterations. For this reason, we use the texture cache to store the protein pocket values.

On the other hand, when multi-dimensional arrays are needed and have to be accessed from different thread blocks, it is very important to organize the data to allow the reads to be coalesced. For this reason, we extensively use CUDA-pitched arrays in storing temporary values that are needed across kernels. Pitched multi-dimensional arrays are an instrument provided by CUDA. They are allocated with rows padded to a size that ensures that each row starts at an address that meets the alignment requirements for coalescing.

3.3. Batched implementation

The second version of the application is the *batched implementation*. This implementation follows a different paradigm from the *latency* one. Instead of using the whole GPU to process a single ligand at a time, we pack it with as many ligands as possible that are processed in parallel (using fewer resources per ligand) as depicted in Fig. 3. This approach follows a paradigm similar to the one described in [15], used within the NAS [31] benchmark suite to estimate the upper achievable limits of floating-point performances on a system since it requires almost no communication to process the data. This approach is possible since the amount of data per ligand is limited (up to 20 KB input - 1 MB output), and thus we can upload on the GPU a huge number of them.

With this approach, the time to process a single ligand t_{batch} will be greater than the time required by the *latency* implementation $t_{latency}$; however, many more ligands will be processed in parallel during the time t_{batch}. As long as the size of the batch of ligands processed in parallel is greater than $t_{batch}/t_{latency}$, this implementation is expected to deliver higher throughput than the *latency* one since reduces to the minimum the number of synchronization points.

When we focus on the kernel design, we must take a completely different approach. The main idea is to parallelize the loop at line 2 of Algorithm 1 and to implement all the steps depicted in Algorithm 2 sequentially in the same kernel. Following CUDA's thread hierarchy, we use 32 CUDA threads to process the ligand's atoms in parallel. A CUDA thread may process more than one atom when the molecule has more than 32 atoms. The spare CUDA threads are not used if the molecule has less than 32 atoms. So, in the following part of the paper, we intend a warp to be a single block of 32 CUDA threads. We launch the kernel over a number of ligands that are enough to cover the GPU parallelism. Fig. 4 provides a schematic view of the logical mapping. It is important to notice that using a single warp to compute a ligand implies that we can use CUDA cooperative groups to perform reductions and early escape in pose evaluations.

This implementation forces us to redesign the whole application approach because we must first load several ligands in a single batch and then launch the processing kernels when the batch is full. We adopted an asynchronous paradigm where different CPU threads managing the upstream phases push ligands in the batch, and another CPU thread is in charge of launching the GPU kernels when the batch is full. In this implementation, external parallelism (CUDA grid) is addressed by

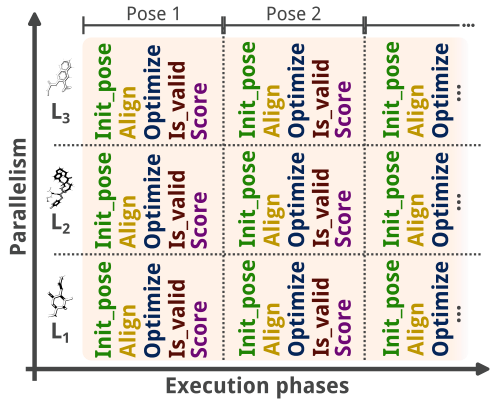


Fig. 4. Logical mapping on how the batched approach hinges on GPU parallelism to accelerate the execution time. All the steps are implemented using a single kernel.

docking multiple ligands simultaneously, while thread-level parallelism is addressed by distributing the set of operations to be performed on the atoms of a single ligand over a single warp.

To make this approach to be successful, we need to address some criticalities. The obvious one is that we need a large number of ligands to fully utilize the GPU. This is not a concern since, as mentioned in Section 1, the virtual screening task we are targeting considers a large chemical space (up to millions or billions of candidate molecules). The second one is that, since we are processing batches of ligands concurrently, the overall kernel time will be dictated by the slowest warp of the grid, i.e., the warp assigned to the ligand that requires more operations. Since the processing is data-dependent, we need to balance the size of the ligands that are collected in a single batch. It is also important to efficiently use registers and shared memory, two precious and scarce resources in the GPU. To make the batched kernels run as fast as possible, the ligand data used often (i.e., atom coordinates and fragments indices) are kept in registers and shared memory to be accessed more efficiently. Since this requires defining at compile time the resources used by the kernels, balancing the sizes of the ligands in the batches allows for maximizing the usage of those statically allocated resources. For this reason, we have clustered the ligands in 5 different batches according to their number of atoms: (0, 32], (32, 64], (64, 96], (96, 128], (128, 160]. The number of ligands accumulated in each batch before being processed by the GPU depends on the maximum number of atoms in its range. For each range, we used kernels compiled to reserve a precise number of registers per thread such that each warp can hold, at most, a number of atoms equal to the upper limit of the range. Thus, the size of each batch is set equal to the maximum number of warps that can be concurrently active on all GPU's SMs with the respective kernel. We determined this number by using Equation (1).

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

In Equation (1), we compute the size of each batch l, where SM is the number of SMs available on the GPU, ws is the warp size, and b is the number of blocks that can run on the same Streaming Multiprocessors (SM) for any given kernel.[4] Section 4 uses an NVIDIA A100 GPU card to validate the approach. However, since we compute the number of ligands l for each bucket using a query to the CUDA runtime, the proposed methodology is agnostic about the target architecture. Indeed, we efficiently deployed LiGen on systems also equipped with V100 and H100 NVIDIA cards. The proposed methodology was able to adapt the number of ligands accordingly.

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

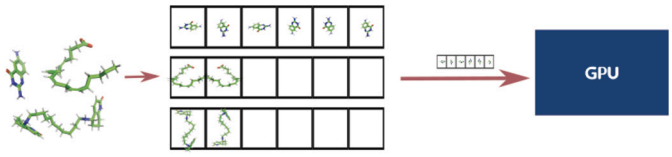


Fig. 5. Graphical representation of the batch creation process: all incoming ligands are divided into batches according to their characteristics, and only when a bucket is full is sent to the GPU.

Moreover, since the *optimize* step needs to process the ligand's rotamers sequentially, we can introduce a strong imbalance if we bundle in the same batch ligands with a different number of rotamers. For this reason, we also need to cluster the *ligands* by their number of fragments. We decided to group them by four (i.e., ligands with 0-3 fragments clustered in one batch, while ligands with 4-7 fragments in another, and so on). This decision is a compromise between having the ligands as similar as possible and avoiding the explosion of the number of different batches. Considering all of these divisions, we have a matrix of buckets where we collect ligands with similar features. This aims at reducing the disparity between the ligands that need to be processed in a batch to improve the efficiency of the computations. A graphical representation of this process is provided in Fig. 5.

The kernels developed for this implementation declare the array parameters as *const _restrict_* so that the compiler can automatically use cached loads for them. Moreover, since we read them only once to copy their content in the register/shared memory, we use regular allocations instead of pitched ones.

4. Experimental results

In this section, we will compare the two implementations in terms of application throughput (end-to-end) on different datasets and conditions. Given that the target molecular docking application has a highly data-dependent throughput, we performed four types of different analyses by changing the characteristics and size of ligand libraries.

The first one takes into consideration more uniform datasets (*preprocessed datasets*), where the ligands to be processed have been clustered according to their characteristics in terms of the number of atoms and fragments (see Subsection 4.2). This analysis has been done to show the different performance trends without the possible noise introduced by the different sizes and flexibility of the target molecules.

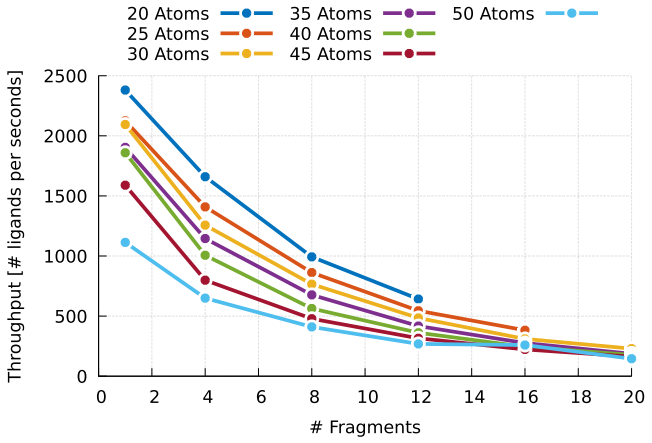
The second analysis regards the scaling of the throughput of the application according to the size of the dataset (see Subsection 4.3). In this case, we want to know if one of the implementations is always optimal or (and this is the expected behavior) if it depends on the dataset size. In this second circumstance, we are particularly interested in finding the dataset size that triggers the optimality change. This analysis has been performed on preprocessed and real-world datasets, where the molecule size and flexibility are unknown *a priori*.

The third analysis (see Subsection 4.4) refers to the performance of both implementations on real-world public datasets, taken from the MEDIATE initiative [24]. The datasets are characterized by a large size and molecule characteristics variability and can be seen as the target of an actual virtual screening campaign.

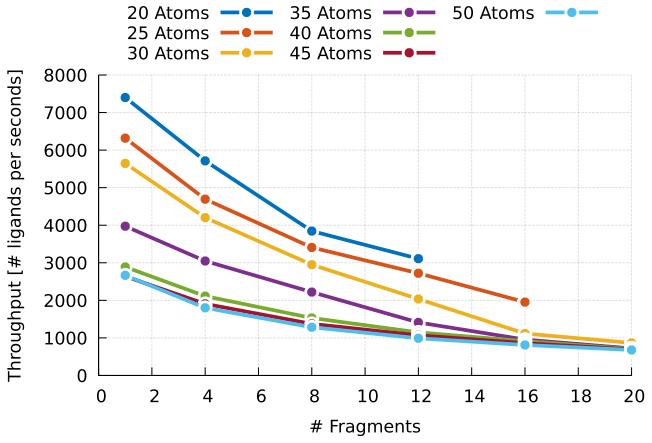
In Subsection 4.5, we report an in-depth analysis of the workload using the *instruction roofline methodology*. This analysis has been performed to better understand the different resource utilization for the two implementations.

4.1. Experimental setup

To perform the docking experiment, we target a machine that resembles an HPC node, equipped with 2 CPU AMD Epyc 7282 2.80 GHz 16 core and one NVIDIA A100 GPU, connected with PCI-E 4.0.

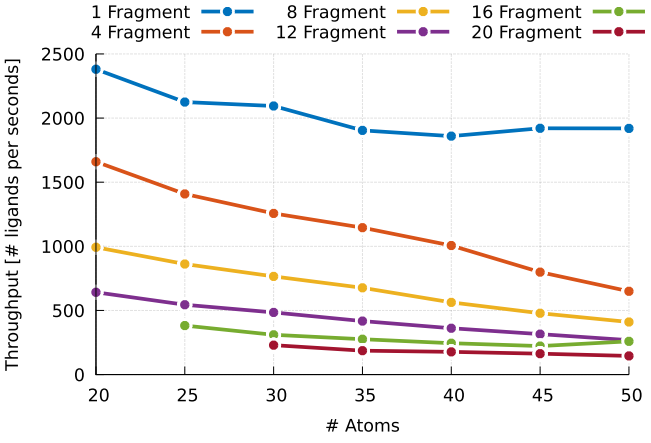


(a) Latency

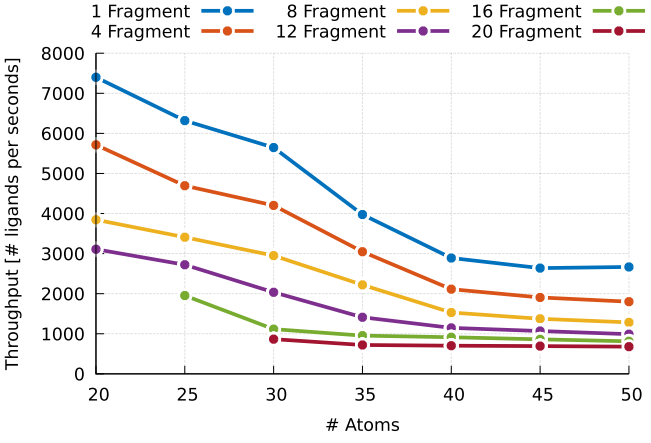


(b) Batched

Fig. 6. Throughput of the two implementations with the different datasets, organized by the number of atoms and increasing the number of fragments on the X axis.



(a) Latency



(b) Batched

Fig. 7. Throughput of the two implementations with the different datasets, organized by the number of fragments and increasing the number of atoms on the X axis.

4.2. Preprocessed datasets

The first set of experiments wants to show the throughput of the two implementations when we are running at the best of the application capabilities (i.e. we recorded the average throughput, which is the total number of items processed since the application launch divided by the total execution time of the application, when its value reaches a stable value.). We have docked several uniform datasets of 50 K ligands, each with fixed characteristics in heavy atoms and fragments. The range is between 20 and 50 heavy atoms and 1 and 20 fragments. In this context, we define every non-hydrogen atom that is part of the molecule as a heavy atom. We need to point out that for the batched implementation, having the same number of heavy atoms does not mean that all the *ligands* belong to the same batch since LiGen groups them according to the total number of atoms, which also includes the hydrogens. The ranges of heavy atoms and fragments for the molecules have been selected considering the ones available in commercial databases.

Fig. 6 and Fig. 7 report the throughput reached by the two implementations for each uniform dataset from two different perspectives.

Fig. 6 plots the varying throughput according to the change in the number of fragments (x-axis) while considering the number of heavy atoms fixed. We can see that this data feature heavily impacts the throughput. The two implementations show similar behavior, going from a high throughput value with 1 fragment and slowing down more with the increase of the number of fragments. However, if we look at the y-axis, we can notice that the batched implementation is much faster than the latency one, on average, by three times.

In Fig. 7, we plot the variation in the throughput at the change of the number of atoms (plotted on the x-axis) while keeping the number of fragments constant. We can notice that, in this case, the behavior is different. The latency implementation has a negligible throughput degradation when we change the number of atoms with a constant number of fragments, while the batched implementation has a more significant throughput loss. However, since it starts from a higher throughput, it still performs better than the latency implementation, in the worst case, by 1.37×.

To conclude this analysis, we can see in Fig. 8 the heatmap of the speedup obtained by the batched implementation compared to the latency implementation, with several datasets of 50 K ligands. As we can see, the batched implementation is always better than the latency one, given this dataset dimension on a single GPU. However, we can notice that the amount of speedup changes according to the characteristics of the *ligands*: the batched implementation behaves dramatically better with fewer atoms and a higher number of fragments.

4.3. Scaling analysis

With this experiment, we aim to find the minimal database size to reach throughput optimality with both implementations, and we are interested in seeing the impact of the dataset composition on this size. For this analysis, we used two different datasets with homogeneous and heterogeneous ligands. We considered a set of molecules with 35 Heavy Atoms and 12 Fragments within the first dataset. This dataset has been selected considering average values for ligand size and flexibility from

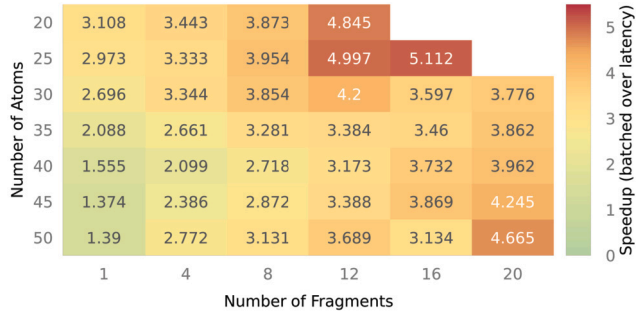


Fig. 8. Speedup Heatmap of the batched version against the latency one for the different homogeneous datasets of 50 K ligands with the same characteristics.

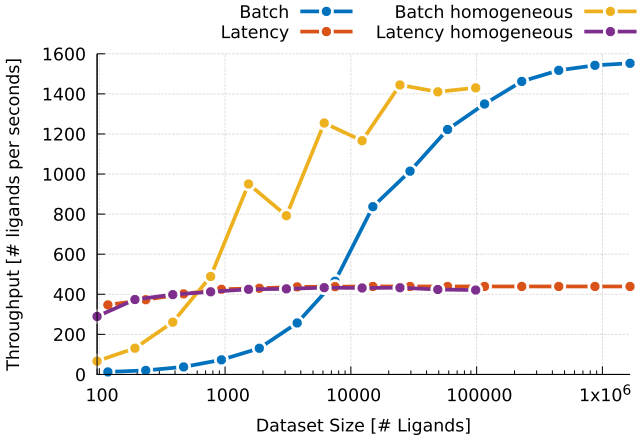


Fig. 9. Scaling analysis.

the ligands considered in the previous section. The second dataset includes a heterogeneous mix of ligands from all previously considered libraries.

Fig. 9 reports the growth of the throughput (y-axis) at the varying of the dataset size (x-axis). As we can see, with small datasets, the latency implementation outperforms the batched implementation. This happens because the batched implementation waits until the batch size is reached and distributes the computation on different CUDA warps. If the dataset is too small and does not reach the size of the batch, we are going to underutilize the GPU, and this explains why in these circumstances, the latency implementation performs better. However, after a certain threshold, we can see that the batched implementation overtakes the latency implementation (with almost exponential growth) until it reaches its saturation point (with a total speedup of around 3.5×). This behavior is observed in both the homogeneous dataset (purple and yellow lines) and the heterogeneous one (blue and red lines). The only difference between the two is when the batched implementation overtakes the latency one, and this happens for the homogeneous dataset one order of magnitude earlier. We can also notice that the growth phase of the batching application when using a mixed dataset ends at almost 10^6 ligands. This means that to get the maximum out of this implementation, we need to dock a very large dataset with at least 10^6 ligands for each GPU involved in the computation. On the other hand, for the homogenous dataset, 20 K ligands are enough to stabilize the throughput. Finally, it is interesting to notice the fluctuations of the throughput in the yellow line (homogeneous batched implementation). As mentioned, the batches are created according to the total number of atoms, including hydrogens, and some *ligands* are processed in different batches. When many buckets are processed, even if they are not completed (i.e., small batches), the resources are not well used, resulting in a performance loss. The higher the number of ligands, the lower the probability of falling in this situation, which can be noticed by the fluctuation reduction while increasing the dataset size.

Table 1
MEDIATE dataset characterization. For each library, its size and the average values (± standard deviations) for the number of heavy atoms and rotatable bonds have been reported.

Library	Size	#Heavy Atoms	#Rot. Bonds
Comm. MW330-	1.9M	18.06 ± 4.05	3.65 ± 1.79
Comm. MW330-500	2.8M	28.12 ± 3.70	5.71 ± 2.11
Comm. MW500+	250 K	38.46 ± 4.83	8.35 ± 3.52
Drugs	8.8 K	29.04 ± 12.89	6.87 ± 5.66
Foods	65.5 K	51.06 ± 18.88	37.91 ± 20.45
Natural Products	263.5 K	30.94 ± 13.03	6.35 ± 6.10
Peptides 2AA	400	20.07 ± 3.33	7.60 ± 1.77
Peptides 3AA	8 K	29.05 ± 4.07	11.40 ± 2.16
Peptides 4AA	160 K	37.40 ± 4.71	15.20 ± 2.51

4.4. Real world datasets

Finally, we want to evaluate the performance of the two implementations on real-world datasets. These datasets come from the MEDIATE [24,44] initiative and contain libraries including ligands from different categories: commercial compounds, natural products, drugs, and peptides.

The “Commercial” category represents the space of currently purchasable compounds libraries [42]. In particular, this set is clustered in three libraries where molecules are selected according to their molecular weight (MW). The first one contains ligands with a molecular weight lower than 330 (MW330-), the second set has ligands with a molecular weight between 330 and 500 (MW330-500), and the last contains all the ligands with a molecular weight higher than 500 (MW500+). The “Drugs” category contains known drugs, including the set of safe-in-man drugs, commercialized or under active development in clinical phases. The “Natural” category contains two sets of molecules: Foods and Natural Products. They are taken from the FooDB online database [9]. FooDB is the world’s largest and most comprehensive resource on food constituents, chemistry, and biology. It provides information on many constituents that give foods flavor, color, taste, texture, and aroma.

Finally, “Peptides” were generated by mixing in a combinatorial way all 20 natural amino acids. They are collected in three files according to the number of amino acids that compose the peptide. In particular, 2AA contains dipeptides (peptides formed by two amino acids), 3AA contains tripeptides, and 4AA contains tetrapeptides. All peptides have been constructed with an extended structure and optimized with MOPAC 2016 [25]. They have been protected with acetylation of the N-terminal end and the addition of amide in the C-terminal one. The total amount of peptides is quite low and not evenly distributed. This is because they are a combination of the 20 amino acids found in nature.

To better contextualize the different sets concerning the analysis done in the previous subsections, a characterization of them in terms of size of the ligand library, number of heavy atoms, and rotatable bonds has been reported in Table 1.

Fig. 10 reports the throughput of the different implementations on the several files composing the mediate dataset. We can immediately notice that the batched version strongly outperforms the latency implementation on the largest files (the Commercial with the different molecular weight). This is expected since we have 5 million molecules here, which heavily exceeds what we have found to be the cross-over point (Subsection 4.3). However, the remaining files are smaller. There are, in particular, two datasets (Drugs and Peptides2AA), where the batched version is unable to reach its optimal performances and a throughput good enough to be better than the latency implementa-

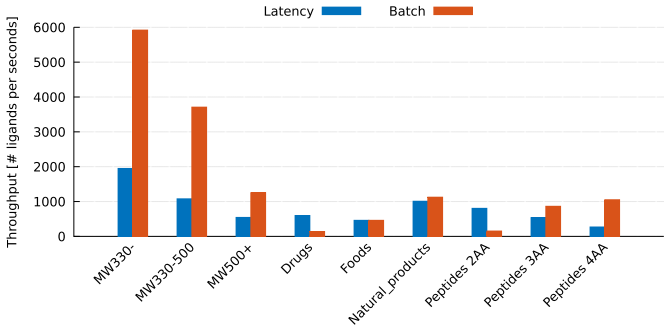


Fig. 10. Throughput comparison on the Mediate dataset.

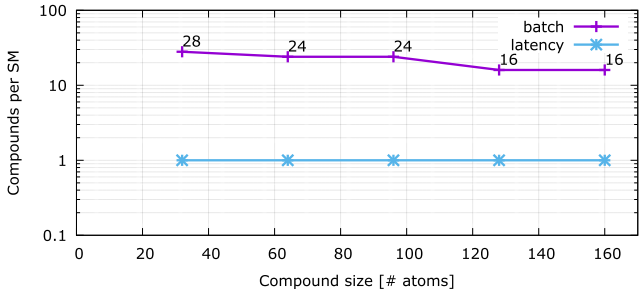


Fig. 11. Static ligands allocation per SM.

tion. The first dataset has 14 K ligands, which should be enough for the scaling analysis to exceed the latency implementation's performance at least. However, it cannot reach a good throughput because it is heavily unbalanced. Thus, in the runtime, it forces the execution of several almost empty batches, which is detrimental to the performances. On the other hand, the Peptides_2AA is a very small dataset, and even if it is quite uniform, it still does not have enough data to outperform the latency implementation. In all the remaining libraries, the batched implementation performs closely or better than the latency but cannot reach its peak performance.

4.5. Workload analysis

The previous analysis shows that the batched implementation has a slow start but a better overall throughput. Now, we want to analyze the two implementations more in-depth to find the reason behind this result, given the performance improvement of the batched implementation goes beyond the reduction in the grid level synchronization. To reach this goal, we will characterize both workloads in terms of execution profiles, applying the *instruction roofline methodology* [4], using GIPS(Giga Instructions Per Second) to assess and measure performance, on an input dataset constructed to be representative of different molecule categories from real-world datasets [24].

We now consider the dimensions that affect the computational complexity of the workload, namely the number of atoms and rotatable bonds. We cannot analyze all possible combinations of atoms and fragments. Therefore, we analyze the application's performance with three clusters of molecules. The characteristics of the clusters have been chosen in an attempt to highlight different levels of complexity. A sample molecule was randomly selected from the test dataset for each of these clusters and then duplicated. The duplicate of the molecule in each cluster coincides with the suggested batch size, as described in Section 3.3. A uniform input dataset allows for homogeneous execution paths across all warps involved in a single kernel grid, especially for the batched implementation where each warp handles different input ligands. The results of this analysis would be the same if, instead of an artificial dataset composed by a duplicated molecule, we use a dataset composed by different ligands referring to the same batch. The test molecule clusters have been defined as:

- *Small*: $(0, 64]$ atoms, 1 rotatable bond, batch of 1920 molecules;
- *Medium*: $(64, 96]$ atoms, 12 rotatable bonds, batch of 1600 molecules;
- *Large*: $(96, 160]$ atoms, 20 rotatable bonds, batch of 960 molecules.

Since we want to know why the two implementation throughput is so different, we focus our analysis on the CUDA bottleneck kernel of each implementation. For the *latency* version, this is the kernel that performs the ligand's fragment optimization (lines 10-16 in Algorithm 2, accounting for 92% of the overall docking pipeline's runtime).

4.5.1. Resources allocation

We first analyzed static resource allocation to understand the consequences of different design principles between the two approaches.

In Fig. 11, the maximum amount of ligands allocated on a single SM is shown. While the *latency* version dedicates all the resources within an SM to a single ligand, the *batch* version allocates multiple ligands to a single warp, allowing for multiple concurrently running ligands in a single SM. In the latter implementation, the registers per thread are the limiting factor for the ligands allocation to an SM. Therefore, the number of ligands assigned to an SM decreases with the increment of their complexity.

This has two evident consequences: on the one hand, the latency implementation has a more consistent behavior that does not depend on the ligand size; on the other hand, the batched implementation is strongly influenced by the data size. It has an optimal behavior with small ligands and degrades, increasing the size of the ligand.

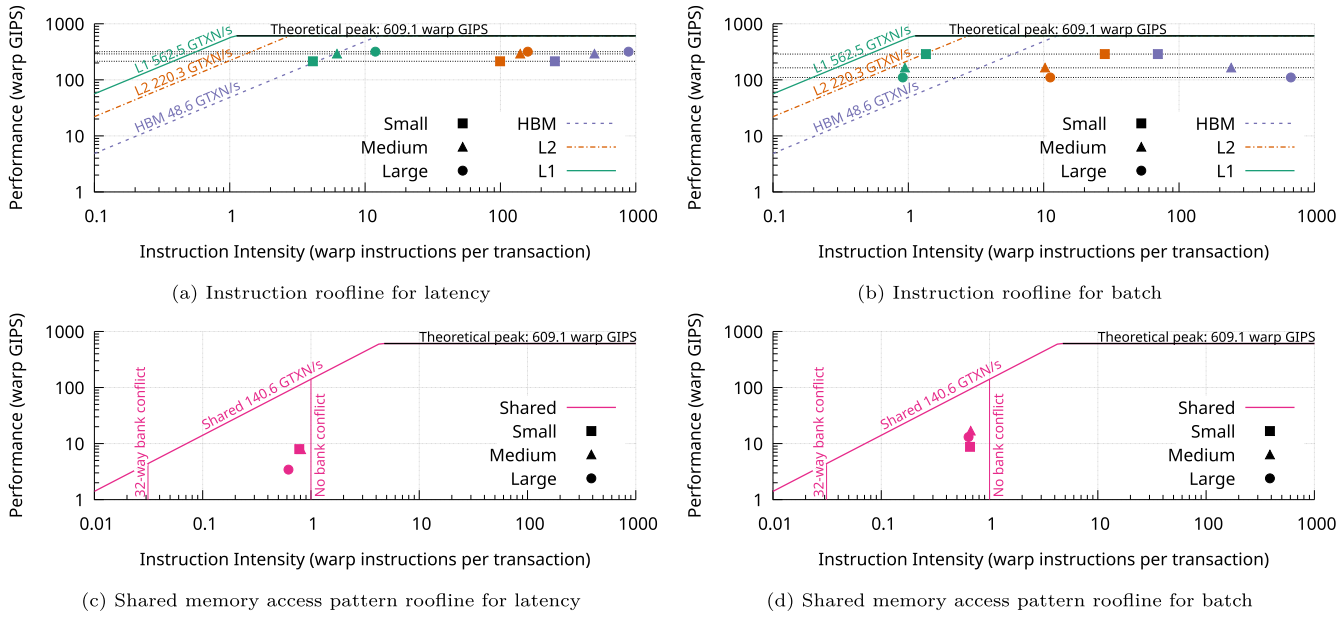
Moreover, we can see that the batched implementation can process more ligands per SM, which allows it to reduce the overheads when launching the kernels since it will have a smaller amount of kernels to launch. Indeed, while in the latency implementation, we have to launch at least one kernel per ligand, we process between 960 and 1920 ligands with a single kernel in the batched one.

4.5.2. Roofline analysis

In this section, we compare the use of computing resources for the two implementations to understand if this could be the reason for the performance differences. In this analysis, we are more interested in the difference between the two implementations rather then their absolute values. We present a comparison between different roofline plots [4] produced by measuring both implementations' execution behaviors via NVIDIA NSight profiler [47] in Fig. 12.

In particular, in Fig. 12a and Fig. 12b, we report the instruction issued roofline. These rooflines are obtained by considering all kinds of warp-level instructions issued. From these two graphs, we can say that both application implementations are not memory-bound. Moreover, we use the GPU appropriately since we are close to the roof. We can notice a difference in the two implementations if we look at their behavior on the size of the different molecules. On one hand, in the *batch* implementation, the amount of GIPS decreases with bigger molecules; on the other hand, in the *latency* implementation, the GIPS value increases with bigger molecules. This is expected due to respective scaling design choices: on the latency version, we improve the number of instructions because the efficiency of the kernel is constant; thus, with bigger molecules, the amount of data to feed the GPU increases. On the other hand, in the batched implementation, we are using more registers to store bigger ligands, and in this way, we have fewer active threads per SM, which decreases the number of instructions issued.

Another insight given by these two plots is the cache reuse: the horizontal distance between points of the same molecule class represents the ability of the cache to satisfy a request. The larger the distance between two points, the higher the reuse of data present in the highest level memory (i.e., the distance between L1 and L2 caches represents the ability of the L1 cache to serve the read request).



(a) Instruction roofline for latency

(b) Instruction roofline for batch

(c) Shared memory access pattern roofline for latency

(d) Shared memory access pattern roofline for batch

Fig. 12. Roofline analysis comparison between *latency* (left) and *batch* (right) on instruction performance (Fig. 12a and Fig. 12b) and shared memory access pattern (Fig. 12c and Fig. 12d).

The *latency* implementation (Fig. 12a) shows regular cache reuse across molecule classes, and we can notice that the reuse of the L2 cache increases with the size of the ligands; we can see from the image that the distance between the red symbols (L2) and purple symbols (HBM) are greater when comparing squares (Small ligands) with circles (Large ligands). On the other hand, the *batched* implementation (Fig. 12b) has a high L1 reuse for *Small*, but the L1 arithmetic and instruction intensities are $\sim 10\times$ lower than L2 and HBM values. However, larger molecule classes begin to rely heavily on L2 cache: this can be seen by the fact that the HBM arithmetic and instruction intensities are $\sim 100\times$ higher than L1 and L2 values. This also strengthens the idea that the batched implementation has better behavior with small molecules but degrades with the data size growth.

The second set of images reports the *shared memory roofline* (Fig. 12c and Fig. 12d). They are obtained by measuring both warp-level load/store instructions issued and shared memory transactions performed). The x-axis indicates within the interval between no bank conflict and 32-way bank conflicts how efficient the kernel is in terms of shared memory access. It is the ratio between the number of shared load and store instructions issued by warps and the effective number of shared memory transactions. For example, in case of no conflict, we can accommodate the load/store operations of all warp's threads in one shared memory transaction; on the contrary, we need to serialize all of them. The y-axis represents the number of shared memory load/store instructions per second. Both implementations show little to no impact due to shared memory bank conflicts and, thus, an efficient access pattern.

From this analysis, the two implementations look similar, with the batched one showing a slightly better utilization of the GPU for small ligands. At the same time, the latency one uses the resources better with large ligands. However, this analysis is unable to explain the speedups that we have found from the experiments done in Subsection 4.2, Subsection 4.4 and Subsection 4.3.

4.5.3. Execution profiling

In this section, we want to investigate the execution profiles of the two implementations in order to gather more insight about them. The results of this analysis are reported in Fig. 13.

Fig. 13a reports the *occupancy*, defined as the ratio between sustained and peak percentage of active warps per SM. Occupancy is one of the factors that can be used to improve performance. However, there are others since it is possible to reach optimal performances by decreasing the occupancy and having more registers per thread [49]. For this reason, we are not interested in the absolute value in this graph, but we are looking at the comparison between the two implementations. Both implementations show a comparable degree of SM occupancy. We can notice that while for *batch*, it decreases with an increasing molecule complexity (more registers used), for *latency*, the behavior is uniform. This analysis does not provide insight into the difference in throughput but helps explain why the advantage of using the batched implementation decreases with larger molecules.

Fig. 13b reports the *efficiency*, defined as the degree of thread predication across all the instructions executed in a single *SM Sub Partition*. Both implementations show high execution efficiency and thus low degrees of thread predication. Slightly lower efficiency in *batch* is ascribed to molecule sizes not being a multiple of the warp size. This plot demonstrates how both implementations are quite efficient in the use of resources.

Finally, Fig. 13c reports the *instruction mix*, defined as the percentage of instructions executed in a single SMSP grouped by instruction type:

- `fp`: floating point instructions (any precision, including scalar, FMA, and tensor),
- `int`: integer instructions (any integer data type),
- `mem`: memory operations (load/stores),
- `cf`: control flow operations,
- `comm`: inter-thread communication and synchronization,
- `misc`: everything else including bit-wise operations and casts

There are two interesting pieces of information in this figure. The first one is that the largest part of the operation done is integer arithmetic. This is expected since they comprehend index calculations, and the *Score* function used to select the best pose is a sum over integer values. Moreover, if we look at the latency implementation, it has a large (20 to 40%) of `comm` instructions that almost completely disappear in the batched implementation. These `comm` instructions are mostly due to the design of the latency kernel, as shown by the algorithm's pseudocode in Section 3. As we already mentioned, we need to process all the fragments sequentially in the pose optimization phase. To analyze the impact on the performance of the *check_bump* kernels, we have run

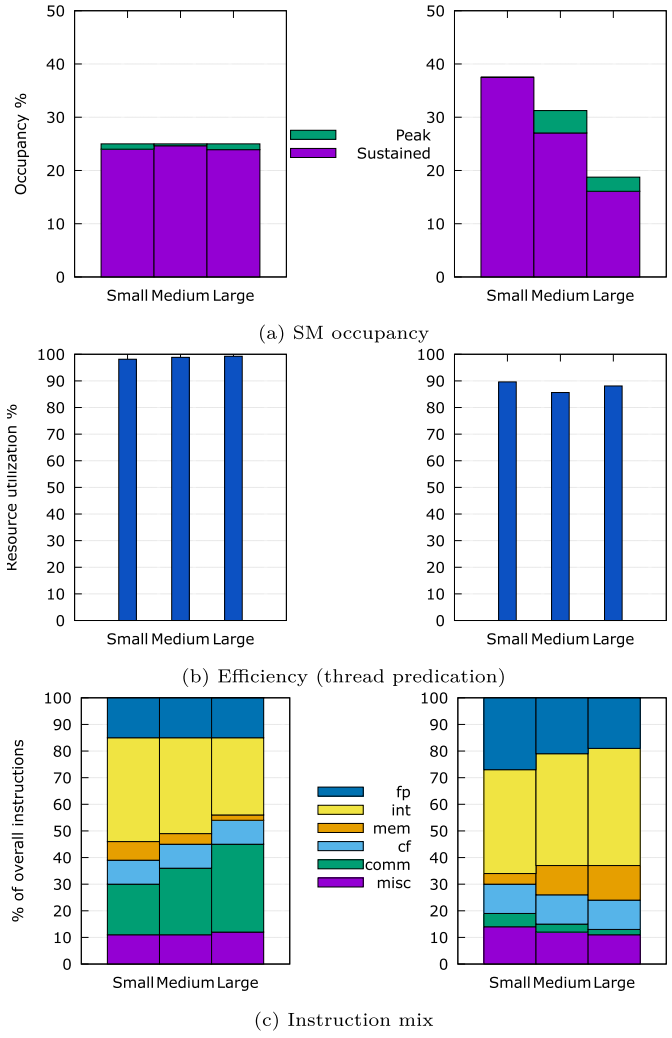


Fig. 13. Comparison between *latency* (left) and *batch* (right) on peak and sustained active warps (Fig. 13a), efficiency (or thread predication, Fig. 13b) and instruction mix (Fig. 13c).

both implementations without the early escape from this loop and report the result in Fig. 14. The advantage in terms of speedup in using the batched approach has been reduced a lot and reached a maximum value of 2× only for very small ligands. This is expected since the previous analysis shows that the batched approach is more efficient for small molecules. On other molecule dimensions, i.e., larger in terms of atoms and fragments), the speedup is slightly above 1, including a small slowdown for the bottom left corner. This analysis confirms that the management of the early exit condition is the tie-breaker between the two implementations since, in the batched version, we can use it without introducing much synchronization overhead.

The *latency* implementation demonstrates consistent performance across molecule classes regarding performance, occupancy, and instruction throughput. This is due to its design principle of scaling computing resources based on the complexity of the input ligand.

On the other hand, the *batch* implementation uses a fixed amount of computing resources allocated to a batch of input ligands and deals with the increasing molecules' complexity by increasing the amount of work a single warp must carry out. Moreover, this second implementation has its best behavior with small molecules, and its performances have a slight degradation when increasing the data size because fewer compute resources are used since we need more registers for the data, thus decreasing the number of active threads.

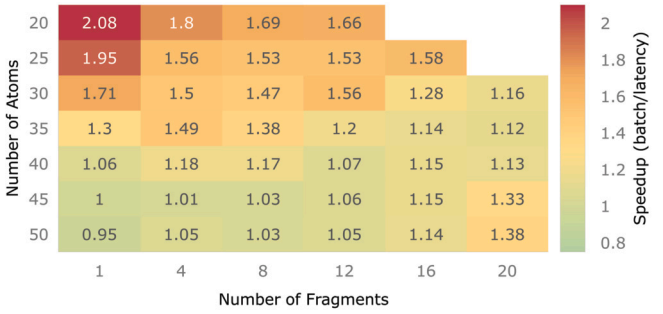


Fig. 14. Speedup Heatmap of the batched version against the latency one for the different homogeneous datasets without the early exit from the *check_bump* function. Both throughputs are taken with large enough datasets.

To summarize this discussion, we have seen that a batched method provides a significant benefit, primarily because processing a ligand with a warp can eliminate most synchronization issues among warps in the same SM. This is fundamental in the *check_bump* function because it allows the exploitation of the early exit condition without introducing too much overhead.

5. Conclusion

In this paper, we have presented the problem of virtual screening a large set of molecules. We have seen that it is usually tackled by performing molecular docking of the candidate molecules in the protein pocket, a process done using large computer simulations. We have presented two optimized implementations of a molecular docking application designed for virtual screening that uses the GPU as a hardware accelerator for the docking procedure. While the first version refers to the classical *latency* approach that spreads the computation of a ligand-protein pair across the device, the second one focuses more on the throughput of a virtual screening campaign. In this second version, we process a *batch* of ligand-protein pairs across the device, increasing the latency of a single evaluation but improving the throughput of the whole screening. The *batch* version required a redesign of the application to pack and carefully cluster similar ligands for more efficient resource usage. We compare the different ideas behind the two approaches, and thanks to an extensive experimental section, we evaluated the two implementations to search for their limits and advantages.

While this paper reports the results only on NVIDIA GPUs, the extension of the analysis on other GPU vendors using different high-level programming languages (i.e., SYCL, HIP) is ongoing.

CRediT authorship contribution statement

Emanuele Vitali: Conceptualization, Data curation, Methodology, Software, Validation, Writing – original draft. **Federico Ficarelli:** Investigation, Software, Writing – original draft. **Mauro Bisson:** Conceptualization, Methodology, Software, Validation, Writing – review & editing. **Davide Gadioli:** Conceptualization, Investigation, Methodology, Software, Writing – original draft, Writing – review & editing. **Gianmarco Accordi:** Methodology, Software, Writing – review & editing. **Massimiliano Fatica:** Conceptualization, Methodology, Supervision, Writing – review & editing. **Andrea R. Beccari:** Funding acquisition, Supervision, Writing – review & editing. **Gianluca Palermo:** Conceptualization, Funding acquisition, Methodology, Supervision, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Emanuele Vitali, Federico Ficarelli, Davide Gadioli, Andrea R.

Beccari, Gianluca Palermo reports financial support was provided by Horizon 2020 Programme under Grant Agreement 101003551 (EXSCALATE4CoV). Emanuele Vitali, Federico Ficarelli, Davide Gadioli, Andrea R. Beccari, Gianluca Palermo reports financial support was provided by EuroHPC Joint Undertaking under Grant Agreement No 956137 (LIGATE).

Data availability

The data that has been used is confidential.

Acknowledgments

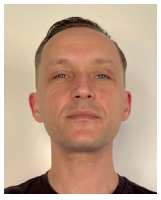
This work has received funding from EuroHPC Joint Undertaking under grant agreement No 956137 (LIGATE) and from the Horizon 2020 Programme under grant agreement No 101003551 (Exscalate4CoV). We acknowledge EuroHPC Joint Undertaking for awarding us access to Karolina at IT4Innovations, Czech Republic (EHPC-DEV-2021D02-049).

References

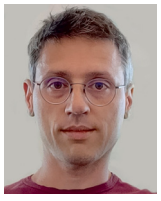
[1] A.R. Beccari, C. Cavazzoni, C. Beato, G. Costantino, Ligen: a high performance workflow for chemistry driven de novo design, J. Chem. Inf. Model. 53 (6) (2013) 1518–1527.
[2] A.R. Beccari, M. Gemei, M.L. Monte, N. Menegatti, M. Fanton, A. Pedretti, S. Bovolenta, C. Nucci, A. Molteni, A. Rossignoli, L. Brandolini, A. Taddei, L. Za, C. Liberati, G. Vistoli, Novel selective, potent naphthyl trpm8 antagonists identified through a combined ligand- and structure-based virtual screening approach, in: Scientific Reports, 2017.
[3] D.E. Clark, What has virtual screening ever done for drug discovery?, Expert Opin. Drug Discov. 3 (8) (2008) 841–851, https://doi.org/10.1517/17460441.3.8.841, pMID: 23484962.
[4] N. Ding, S. Williams, An instruction roofline model for GPUs, in: 2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), IEEE, Denver, CO, USA, 2019, pp. 7–18, https://ieeexplore.ieee.org/document/9059264/.
[5] D. Dong, Z. Xu, W. Zhong, S. Peng, Parallelization of molecular docking: a review, Curr. Top. Med. Chem. 18 (2018), https://doi.org/10.2174/1568026618666180821145215.
[6] T.J. Ewing, S. Makino, A.G. Skillman, I.D. Kuntz, Dock 4.0: search strategies for automated molecular docking of flexible molecule databases, J. Comput.-Aided Mol. Des. 15 (5) (2001) 411–428.
[7] M. Fan, J. Wang, H. Jiang, Y. Feng, M. Mahdavi, K. Madduri, M.T. Kandemir, N.V. Dokholyan, Gpu-accelerated flexible molecular docking, J. Phys. Chem. B 125 (4) (2021) 1049–1060, https://doi.org/10.1021/acs.jpcb.0c09051, pMID: 33497567.
[8] Y. Fang, Y. Ding, W.P. Feinstein, D.M. Koppelman, J. Moreno, M. Jarrell, J. Ramanujam, M. Brylinski, Geauxdock: accelerating structure-based virtual screening with heterogeneous computing, PLoS ONE 11 (7) (2016) e0158898.
[9] Foodb: the largest and most comprehensive resource on food constituents, www.foodb.ca.
[10] R.A. Friesner, J.L. Banks, R.B. Murphy, T.A. Halgren, J.J. Klicic, D.T. Mainz, M.P. Repasky, E.H. Knoll, M. Shelley, J.K. Perry, D.E. Shaw, P. Francis, P.S. Shenkin, Glide: a new approach for rapid, accurate docking and scoring. 1. method and assessment of docking accuracy, J. Med. Chem. 47 (7) (2004) 1739–1749, https://doi.org/10.1021/jm0306430, pMID: 15027865.
[11] D. Gadioli, G. Palermo, S. Cherubin, E. Vitali, G. Agosta, C. Manelfi, A.R. Beccari, C. Cavazzoni, N. Sanna, C. Silvano, Tunable approximations to control time-to-solution in an hpc molecular docking mini-app, J. Supercomput. 77 (1) (2021) 841–869.
[12] D. Gadioli, E. Vitali, F. Ficarelli, C. Latini, C. Manelfi, C. Talarico, C. Silvano, C. Cavazzoni, G. Palermo, A.R. Beccari, Exscalate: an extreme-scale virtual screening platform for drug discovery targeting polypharmacology to fight sars-cov-2, IEEE Trans, Emerg. Topics Comput. (2022) 1–12, https://doi.org/10.1109/TETC.2022.3187134.
[13] E. Glaab, Building a virtual ligand screening pipeline using free software: a survey, Brief. Bioinform. 17 (2) (2016) 352–366, https://doi.org/10.1093/bib/bbv037, https://europepmc.org/articles/PMC4793892.
[14] J. Glaser, J.V. Vermaas, D.M. Rogers, J. Larkin, S. LeGrand, S. Boehm, M.B. Baker, A. Scheinberg, A.F. Tillack, M. Thavappiragasam, et al., High-throughput virtual laboratory for drug discovery using massive datasets, Int. J. High Perform. Comput. Appl. (2021) 10943420211001565.
[15] C. Gong, J. Liu, J. Qin, Q. Hu, Z. Gong, Efficient embarrassingly parallel on graphics processor unit, in: 2010 2nd International Conference on Education Technology and Computer, vol. 4, 2010, pp. V4–400–V4–404.
[16] G. Jones, P. Willett, R.C. Glen, A.R. Leach, R. Taylor, Development and validation of a genetic algorithm for flexible docking, J. Mol. Biol. 267 (3) (1997) 727–748.
[17] O. Korb, T. Stützle, T.E. Exner, Accelerating molecular docking calculations using graphics processing units, J. Chem. Inf. Model. 51 (4) (2011) 865–876, https://doi.org/10.1021/ci100459b.
[18] S. LeGrand, A. Scheinberg, A.F. Tillack, M. Thavappiragasam, J.V. Vermaas, R. Agarwal, J. Larkin, D. Poole, D. Santos-Martins, L. Solis-Vasquez, et al., Gpu-accelerated drug discovery with docking on the summit supercomputer: porting, optimization, and application to covid-19 research, in: Proceedings of the 11th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics, 2020, pp. 1–10.
[19] E. Lionta, G. Spyrou, D.K. Vassilatis, Z. Cournia, Structure-based virtual screening for drug discovery: principles, applications and recent advances, Curr. Top. Med. Chem. 14 (16) (2014) 1923–1938.
[20] M. Liu, S. Wang, Mcdock: a Monte Carlo simulation approach to the molecular docking problem, J. Comput.-Aided Mol. Des. 13 (5) (1999) 435–451.
[21] A. MacConnachie, Zanamivir (relenza®) — a new treatment for influenza, Intensive Crit. Care Nurs. 15 (6) (1999) 369–370, https://doi.org/10.1016/S0964-3397(99)80031-7.
[22] S. Markidis, D. Gadioli, E. Vitali, G. Palermo, Understanding the i/o impact on the performance of high-throughput molecular docking, in: 2021 IEEE/ACM Sixth International Parallel Data Systems Workshop (PDSW), 2021, pp. 9–14.
[23] D. Medeiros, G. Schieffer, J. Wahlgren, I. Peng, A gpu-accelerated molecular docking workflow with kubernetes and apache airflow, in: A. Bienz, M. Weiland, M. Baboulin, C. Kruse (Eds.), High Performance Computing, Springer Nature Switzerland, Cham, 2023, pp. 193–206.
[24] Mediate: Molecular docking at home, https://mediate.exscalate4cov.eu/.
[25] Mopac2016, http://openmopac.net/home.html.
[26] G.M. Morris, R. Huey, W. Lindstrom, M.F. Sanner, R.K. Belew, D.S. Goodsell, A.J. Olson, Autodock4 and autodocktools4: automated docking with selective receptor flexibility, J. Comput. Chem. 30 (16) (2009) 2785–2791.
[27] N.A. Murugan, A. Podobas, D. Gadioli, E. Vitali, G. Palermo, S. Markidis, A review on parallel virtual screening softwares for high-performance computers, Pharmaceuticals 15 (1) (2022), https://doi.org/10.3390/ph15010063, https://www.mdpi.com/1424-8247/15/1/63.
[28] G. Palermo, G. Accordi, D. Gadioli, E. Vitali, C. Silvano, B. Guindani, D. Ardagna, A.R. Beccari, D. Bonanni, C. Talarico, F. Lughini, J. Martinovic, P. Silva, A. Bohm, J. Beranek, J. Krenek, B. Jansik, B. Cosenza, L. Crisci, P. Thoman, P. Salzmann, T. Fahringer, L.T. Alexander, G. Tauriello, T. Schwede, J. Durairaj, A. Emerson, F. Ficarelli, S. Wingbermühle, E. Lindhal, D. Gregori, E. Sana, S. Coletti, P. Gschwandtner, Tunable and portable extreme-scale drug discovery platform at exascale: the ligate approach, in: Proceedings of the 20th ACM International Conference on Computing Frontiers, CF '23, Association for Computing Machinery, New York, NY, USA, 2023, pp. 272–278.
[29] P.N. Palma, L. Krippahl, J.E. Wampler, J.J. Moura, Bigger: a new (soft) docking algorithm for predicting protein interactions, Proteins, Struct. Funct. Bioinform. 39 (4) (2000) 372–384.
[30] S. Ruder, An overview of gradient descent optimization algorithms, arXiv:1609.04747, 2017.
[31] S. Saini, D.H. Bailey, Nas Parallel Benchmark (Version 1.0) Results 11-96, NASA Ames Research Center, 1996.
[32] I. Sánchez-Linares, H. Pérez-Sánchez, J.M. Cecilia, J.M. García, High-throughput parallel blind virtual screening using BINDSURF, BMC Bioinform. 13 (SUPPL 14) (2012), https://doi.org/10.1186/1471-2105-13-S14-S13.
[33] D. Santos-Martins, L. Solis-Vasquez, A.F. Tillack, M.F. Sanner, A. Koch, S. Forli, Accelerating autodock4 with gpus and gradient-based local search, J. Chem. Theory Comput. (2021).
[34] J.R. Schames, R.H. Henchman, J.S. Siegel, C.A. Sotriffer, H. Ni, J.A. McCammon, Discovery of a novel binding trench in hiv integrase, J. Med. Chem. 47 (8) (2004) 1879–1881, https://doi.org/10.1021/jm0341913, pMID: 15055986.
[35] I. Schellhammer, M. Rarey, Flexx-scan: fast, structure-based virtual screening, Proteins: Struct. Funct. Bioinform. 57 (3) (2004) 504–517.
[36] G. Schieffer, I. Peng, Accelerating drug discovery in autodock-gpu with tensor cores, in: J. Cano, M.D. Dikaiakos, G.A. Papadopoulos, M. Pericàs, R. Sakellariou (Eds.), Euro-Par 2023: Parallel Processing, Springer Nature Switzerland, Cham, 2023, pp. 608–622.
[37] L. Solis-Vasquez, D. Santos-Martins, A.F. Tillack, A. Koch, J. Eberhardt, S. Forli, Parallelizing irregular computations for molecular docking, in: 2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3), 2020, pp. 12–21.
[38] B. Sukhwani, M.C. Herbordt, Gpu acceleration of a production molecular docking code, in: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, 2009, pp. 19–27.
[39] S. Tang, R. Chen, M. Lin, Q. Lin, Y. Zhu, J. Ding, H. Hu, M. Ling, J. Wu, Accelerating autodock vina with gpus, Molecules 27 (9) (2022) 3041.
[40] M. Thavappiragasam, V. Kale, O. Hernandez, A. Sedova, Addressing load imbalance in bioinformatics and biomedical applications: efficient scheduling across multiple gpus, in: 2021 IEEE International Conference on Bioinformatics and Biomedicine (BIBM), 2021, pp. 1992–1999.
[41] R. Thomsen, M.H. Christensen, Moldock: a new technique for high-accuracy molecular docking, J. Med. Chem. 49 (11) (2006) 3315–3321.
[42] H. van Vlijmen, J.-Y. Ortholand, V.M.-J. Li, J.S. de Vlieger, The European lead factory: an updated hts compound library for innovative drug discovery, Drug Discov.

Today 26 (10) (2021) 2406–2413, https://doi.org/10.1016/j.drudis.2021.04.019, https://www.sciencedirect.com/science/article/pii/S1359644621002063.
[43] J.V. Vermaas, A. Sedova, M.B. Baker, S. Boehm, D.M. Rogers, J. Larkin, J. Glaser, M.D. Smith, O. Hernandez, J.C. Smith, Supercomputing pipelines search for therapeutics against covid-19, Comput. Sci. Eng. 23 (1) (2021) 7–16, https://doi.org/10.1109/MCSE.2020.3036540.
[44] G. Vistoli, C. Manelfi, C. Talarico, A. Fava, A. Warshel, I.V. Tetko, R. Apostolov, Y. Ye, C. Latini, F. Ficarelli, G. Palermo, D. Gadioli, E. Vitali, G. Varriale, V. Pisapia, M. Scaturro, S. Coletti, D. Gregori, D. Gruffat, E. Leija, S. Hessenauer, A. Delbianco, M. Allegretti, A.R. Beccari, MEDIATE - molecular DockIng at homE: turning collaborative simulations into therapeutic solutions, Expert Opin. Drug Discov. 18 (8) (2023) 821–833, https://doi.org/10.1080/17460441.2023.2221025, pMID: 37424369.
[45] NVIDIA CUDA Guide, https://docs.nvidia.com/cuda/.
[46] NVIDIA CUDA Occupancy Calculator, https://docs.nvidia.com/cuda/cuda-occupancy-calculator.
[47] NVIDIA Nsight Compute Kernel Profiling Guide, https://docs.nvidia.com/nsight-compute/ProfilingGuide.
[48] E. Vitali, D. Gadioli, G. Palermo, A. Beccari, C. Cavazzoni, C. Silvano, Exploiting openmp and openacc to accelerate a geometric approach to molecular docking in heterogeneous hpc nodes, J. Supercomput. 75 (7) (2019) 3374–3396.
[49] V. Volkov, Better performance at lower occupancy, in: Proceedings of the GPU Technology Conference, GTC, vol. 10, San Jose, CA, 2010, p. 16.
[50] Y. Yu, C. Cai, J. Wang, Z. Bo, Z. Zhu, H. Zheng, Uni-dock: gpu-accelerated docking enables ultralarge virtual screening, J. Chem. Theory Comput. 19 (11) (2023) 3336–3345, https://doi.org/10.1021/acs.jctc.2c01145, pMID: 37125970.
[51] S. Zhang, K. Kumar, X. Jiang, A. Wallqvist, J. Reifman, Dovis: an implementation for high-throughput virtual screening using autodock, BMC Bioinform. 9 (1) (2008) 1–4.
[52] X. Zhang, S.E. Wong, F.C. Lightstone, Message passing interface and multithreading hybrid for parallel molecular docking of large databases on petascale high performance computing machines, J. Comput. Chem. 34 (11) (2013) 915–927.

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



Federico Ficarelli graduated in Computer Science at the University of Milan in 2008. He is currently a senior HPC software engineer in the High Performance Computing dept. at Cineca where he leads several co-design and development activities in the industrial R&D team. He is involved in several research projects as responsible for application and hardware-software co-design activities focusing on novel computing architectures, programming paradigms for heterogeneous platforms and compiler technologies.



Mauro Bisson graduated in Computer Science in 2006 and received his PhD degree in Computer Science in 2011, both from the Sapienza University of Rome, Italy. He is a software engineer in the High Performance Computing and Benchmarks team at NVIDIA. His research interests include graph processing, image processing, scientific and high performance computing and code optimization. He has been an ACM Gordon Bell finalist four times, received an honorable mention in 2011 and has received a number of IEEE awards for his works on graph processing.

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

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

Massimiliano Fatica graduated in Aeronautical Engineering and received a PhD degree in Theoretical and Applied Mechanics, both from the Sapienza University of Rome, Italy. He is the Director of the High Performance Computing and Benchmarks team at NVIDIA. His research interests include computational fluid dynamics and parallel and high performance computing. He has been a finalist in the ACM Gordon Bell prize multiple times, received an honorable mention in 2011 and won the prize in 2018 for applying an exascale-class deep learning application to extreme climate data and breaking the Exaop computing barrier for the first time with a deep learning application.

Andrea R. Beccari is currently responsible for the Drug Discovery Platform of Domp´e Farmaceutici SpA and leader of the EXSCALATE team. Since 2015, responsible of the Joint Bioinformatics Groups at the IBP Institute of the National Research Council of Italy. He was promoter and coordinator of the open innovation initiative: Italian Drug Discovery Network and co-founder and member of the board of the Avicenna Alliance (Brussel). He was the originator and chairman of the Computational Driven Drug Discovery and Italian Drug Discovery Summit series of meetings. He has co-organized several initiatives with the European Commission and parliament promoting the use of in-silico simulation to increase the awareness towards the potentiality of high-performance computing in healthcare. He is project coordinator of the H2020-EXSCALATE4CoV and EuroHPC-LIGATE projects. He published more than 20 publications in peer review journals and co-author for 7 patents.

Gianluca Palermo received his Master of Science degree in Electronic Engineering, in 2002, and the PhD degree in Computer Engineering, in 2006, from Politecnico di Milano (Italy). He is currently a Full Professor at the Department of Electronics, Information, and Bioengineering (DEIB) at the same University. Previously, he was part of the Low-Power Design Group of AST - STMicroelectronics working on Network-on-Chip architectures, and a Research Assistant at the Advanced Learning and Research Institute (ALaRI) of the Universita' della Svizzera Italiana. His research interests include design methodologies and architectures for embedded and HPC systems focusing on autotuning aspects. Since 2003, he published more than 100 scientific papers in peer-reviewed conferences and journals.