Exploiting OpenMP & OpenACC to Accelerate a Geometric Approach to Molecular Docking in Heterogeneous HPC Nodes

**Emanuele Vitali · Davide Gadioli ·
Gianluca Palermo · Andrea Beccari ·
Carlo Cavazzoni · Cristina Silvano**

Abstract In drug discovery, molecular docking is the task in charge of estimating the position of a molecule when interacting with the docking site. This task is usually used to perform screening of a large library of molecules, in the early phase of the process. Given the amount of candidate molecules and the complexity of the application, this task is usually performed using High-Performance Computing (HPC) platforms. In modern HPC systems, heterogeneous platforms provide a better throughput with respect to homogeneous platforms.

In this work, we ported and optimized a molecular docking application to a heterogeneous system, with one or more GPU accelerators, leveraging a hybrid OpenMP and OpenACC approach. The target application focuses on the virtual screening phases in the drug-discovery process and it is based on geometric transformations of the target ligands. We prove that our approach has a better exploitation of the node compared to pure CPU/GPU data splitting approaches, reaching a throughput improvement up to 25% while considering the same computing node.

Emanuele Vitali · Davide Gadioli · Gianluca Palermo · Cristina Silvano
Dipartimento di Elettronica, Infomazione e Bioingegneria, Politecnico di Milano, Italy.
E-mail: name.surname@polimi.it

Andrea Beccari
Dompe' Farmaceutici SpA Research Center, L'Aquila, Italy.
E-mail: andrea.beccari@dompe.com

Carlo Cavazzoni
Cineca Supercomputing Innovation and Applicatioon Department, CINECA, Bologna, Italy.
E-mail: c.cavazzoni@cineca.it

1 Introduction

The drug discovery process involves several tasks performed *in-silico*, *in-vitro*, and *in-vivo*. While the output of this process is a single valid solution, typically it starts considering a huge set of candidate molecules. In the early stages of drug discovery, the focus is on finding a small set of candidate molecules, named *ligands*, which have a strong interaction with the binding site of a target molecule, named *pocket*. These stages are performed *in-silico* and they usually leverage molecular docking algorithms, which estimate the strength of the interaction between the target *pocket* and the evaluated *ligand*.

The strength of the interaction depends on the three-dimensional displacement of the *ligand* when it interacts with the target *pocket*. Therefore, a docking algorithm must estimate the correct *ligand* pose to test if the *ligand* has to be discarded. This is a complex task due to the high number of involved degrees of freedom. Besides the six degrees of freedom for placing a rigid body in a three-dimensional space (three translational and three rotational), it is possible to change the geometrical shape of a *ligand* without altering its chemical properties. Indeed, a subset of bonds between the *ligands* atoms, named *rotamers*, split the *ligand* into two disjoint sets of atoms that are able to rotate independently along the bond axis, called *fragments*. Thus, each *rotamer* introduces two degrees of freedom in the problem. Given that a *ligand* might have more than one hundred of *fragments*, estimating the strength of the interaction between a *ligand* and the target *pocket* is a complex task. If we consider that pharmaceutical companies would like to consider billions of *ligands* to increase the probability to find the best candidates, we require a large computational power. For this reason, pharmaceutical companies usually rely on High-Performance Computing to address the early phases of the drug discovery process.

The Top500 list [16] ranks HPC platforms according to their computational power in terms of throughput. If we consider the most powerful platforms, as in November 2018, they use heterogeneous computation units. Given that the performance of a platform is limited by the power consumed, energy efficiency is becoming a key factor in the HPC context. Depending on the application algorithm, hardware accelerators such as GPU or Xeon-phi might significantly improve the application throughput with respect to general purposes CPUs, considering the same power consumption. From the pharmaceutical company point of view, the benefits of an increment of the throughput are twofold. On one hand, it might reduce the monetary cost of the drug discovery process. On the other hand, it might increment the number of evaluated *ligands* , therefore increasing the probability of finding a good candidate.

Given that each *ligand* evaluation is independent from the others, the first stages of the drug discovery process are data parallel. Moreover, the evaluation of each three-dimensional displacement of a *ligand* is still independent of the others. Given the high amount of independent computations, this process matches the GPUs computation paradigm. In the typical heterogeneous programming model, the application is still executed in a general purpose CPU,

named *host*. Only the hot-spots of the application are offloaded to accelerators, named *devices*. Typically, by using this approach application developers are facing two problems: (1) managing data transfers between host and device, and (2) implementing the hot-spots of the application using a different language for a different architecture.

In our previous work [18], we investigated the benefits and limitations of using the OpenACC [4] language extension in a molecular docking application, to mitigate the second problem. In this work, we implemented a hybrid version by using OpenMP [2] and OpenACC to leverage the processing elements on a heterogeneous node. In particular, given the limitation analyzed in the previous work, we aim at mapping each phase of the application on the most suitable processing element. To summarize, the contributions of this work are the following:

– We propose an OpenACC version of the geometric docking algorithm targeting GPU architectures;
– We propose a Hybrid CPU/GPU version of the geometric docking algorithm capable to fully exploit the node heterogeneity;
– We analyze the resource utilization of the different solutions to find the best configuration also in presence of multi-GPU nodes;
– We discuss the obtained results, comparing them with the sequential CPU application and the GPU implementation of our previous work;

In Section 2 we describe previous work in molecular docking. We focus on how they dealt with the complexity of the problem, to introduce and examine languages and techniques to accelerate and optimize computations. Section 3 describes the original algorithm of the target molecular docking application, designed for general purpose CPU. After this background, Section 4 describes how we used OpenACC and OpenMP to leverage the heterogeneity of the platform. We evaluate the benefits and limitation of the enhanced application in Section 5. Finally, Section 6 summarizes the overall findings and concludes the paper.

2 State of the art

Molecular docking is an important application domain in High-Performance Computing. Due to the complexity of the problem, it is not possible to evaluate all the poses of a *ligand* when it interacts with the target *pocket*. Therefore, all the algorithms proposed in literature aim at finding a *good enough* solution within a reasonable amount of time.

A common approach for performing molecular docking employs stochastic algorithms to sample the space of possible *ligand* poses. For example, MolDock [15] and Autodock [10] leverage genetic algorithms to converge to an optimal solution. While works such as Glide [5] and AutoDock Vina [17] use a Monte Carlo algorithm. Using a different approach, it is possible to use geometrical and chemical properties of the target *pocket* and of the evaluated *ligand* to

drive the docking process following a heuristic. For example, Dock 4.0 [3], FLEXX [8], and Surflex-Dock 2.1 [6] belong to this category. While stochastic algorithms converge to an optimal pose, the heuristic employed in deterministic approaches might fail to dock the *ligand* in the target *pocket*. However, the reproducibility of the results might become a domain requirement since the later stages of a drug discovery process require a heavy monetary effort. In this work, we focus on LiGenDock [1], which is a deterministic docking application that might be used to perform two tasks. On one hand, it aims at performing an accurate docking, by using chemical and physical information. This task might be used to accurately estimate the three-dimensional pose of a *ligand* to forward to later stages of the process. On the other hand, it aims at performing a fast docking, using only geometrical information. This task might be used to perform a fast virtual screening of a huge library of *ligands*.

From the implementation point of view, previous works propose a different approach to leverage the available computation resources. Frameworks such as LiGen [1] and DOCK [3], support the MPI paradigm to enable multi-node scaling. Other frameworks, such as Glide[5], MolDock[15] or Autodock Vina[17], prefer to split the data across different nodes. Therefore, each instance of the application executes in a different node, processing a fraction of the ligand database. A post-processing phase is then required to unify and to evaluate the results.

From the single node perspective, DOCK[3] and Autodock Vina[17] have been designed to leverage the computational power of a homogeneous node. Differently, Glide[5] and MolDock[15] also address heterogeneous nodes.

In this work, we use as baseline an application [1] that is already able to scale horizontally. We focus on vertical scaling addressing heterogeneous nodes with at least one GPU. In this direction, two are the main approaches for offloading computation to GPUs: using dedicated languages and directive-based programming models.

In the first case, it is possible to write the source code of the application, using specific languages such as CUDA [11] or OpenCL [14]. Usually, they separate the source code executed by the CPU (host side), from the code executed by the accelerator (device side). Therefore, application developers need (1) to write the offloaded kernel considering the device memory model and parallelization scheme. Moreover, they need (2) to explicitly handle the data transfer between host and device; and they need to (3) write the boilerplate code for the initialization. However, with this approach application developers have the finest control on the application source code.

The second approach leverages directive-based languages, like OpenACC [4] and OpenMP [2]. In these languages, the application developer uses compiler directives to annotate the source code. The toolchain transforms and compiles the offloaded kernels, and generates the code to transfer the data between host and device. Moreover, it automatically generates the initialization code. The benefit of this approach is the ease of use. The programmer writes the entire application with a single language independently from the actual target, i.e. CPU host or the accelerator. With this approach, a single source code can be

executed on different hardware, thus enabling functional portability. However, the application developer is still in charge of writing an algorithm suitable for the device memory model and parallelization scheme. Despite the multiplatform approach of those languages, the kernels must be tuned according to the target platform, since performance portability is still an open problem [13]. Indeed, one of the conclusions of our previous work [18] was that the code modified and optimized for the GPU was not efficient when compiled and run on the CPU as the original code.

To improve computation efficiency, our intention in this paper is to optimize the exploitation of all the computational resources available in an HPC node. We will continue to leverage MPI for the inter-node communication. In modern systems, the HPC node typically includes several CPUs and GPUs. The multi-GPU problem has been investigated in literature. For example, the approaches proposed in [23, 22] suggest to extend OpenMP to support multiple accelerators in a seamless way. OpenACC has runtime functions to support the utilization of multiple GPU, however, lacks GPU to GPU data transfer, in single node [21] and multinode [9]. A previous work in literature [20] investigates a hybrid approach with OpenMP and OpenACC. It proposes the usage of OpenMP to support a multi-GPU OpenACC application, assigning each GPU to an OpenMP thread. In this context, each OpenMP thread performs the data transfer between the host and the target device, without performing any other computation.

In this paper, we extend our previous work [18] by suggesting a new approach that offloads to the GPU the most compute-intensive kernels, using a hybrid approach of OpenMP and OpenACC. In this way, we exploit multi-GPU nodes offloading to accelerators only the kernels that maximize the advantage of being run on the GPU. Differently from previous approaches, we rely on CPU to compute kernels that are not suitable for GPUs. Moreover, to maximize the utilization of the resources, we dedicate more than one OpenMP thread for each GPU. In this way, we split the workload across the compute units, according to the characteristic of the compute units and of the algorithm.

3 Target Application

This section introduces the target application by describing its algorithm and by providing timing analysis. We omit implementation details that are not required to understand the proposed approach, for simplicity and clarity reasons.

3.1 Application Algorithm

This work targets the *LiGenDock* [1] molecular docking application. *LiGenDock* uses a two-level approach for screening a large library of *ligands*. First, it

Listing 1: Pseudo-code of the original algorithm.

```
 1   load(pocket);
 2   for (ligand : ligands)
 3   {
 4      for (pose_id=0; pose_id<N, pose_id++){
 5         generate_starting_pose(pose_id , ligand);
 6         align_ligand (ligand , pocket)
 7         for(rep = 0; rep <num_repetitions; rep++){
 8            optimize_pose(ligand , pocket);
 9         }
10      }
11   }
```

performs molecular docking considering only geometrical features, to filter out the *ligands* that are not able to fit in the target *pocket*. Then, it simulates the actual physical and chemical interaction of the most promising *ligands* to find a more accurate estimation of their three-dimensional pose when interacting with the target *pocket*.

In this paper, we focus on a *LiGenDock* mini-app performing only geometrical transformations, named *GeoDock*. *GeoDock* attempts to capture key computation kernels of the molecular docking application for the drug discovery implemented in LiGenDock. It performs the docking using only geometrical features, such as the position of the *ligand* atoms with respect to the *pocket* and shape of the *ligand* fragments. In particular, *GeoDock* explores the conformer space using operators such as *ligand* roto-translations and fragment rotations.

By developing *GeoDock* in parallel with the new version of LiGenDock, application developers can work with system architects and domain-experts to evaluate alternative algorithms to better satisfy the end-user constraints or to better exploit the architectural features. *GeoDock* enables us a faster performance analysis and the optimization of the key kernels.

Listing 1 shows the pseudocode of *GeoDock* algorithm. In the initial phase, it reads the target *pocket* and the *ligand* library (lines 1–2). The algorithm that performs the molecular docking between a *ligand-pocket* pair is the body of the outermost loop (lines 4–10). Given the size of the solution space, an exhaustive exploration is unfeasible. Therefore, *GeoDock* uses a greedy heuristic based on gradient descent with multiple restarts (lines 4–5) to process the solution space (lines 6–9). In particular, multiple restarts are composed of different starting poses. To generate the starting poses, *GeoDock* performs uniform sampling in the conformer space of the original molecule considering only the most important fragments. The docking algorithm is composed of two phases. In the first phase *GeoDock* considers a *ligand* as a rigid body and it aims at finding the best orientation that fits the *pocket* (line 6). The best alignment is evaluated with a scoring function, that is called at every rotation. In this work, we did not modify the original scoring function used in LiGen that it is using an in-house developed empiric scoring function reported in [1] and it is based on the work of Wang [19]. The second phase addresses the internal degrees of freedom
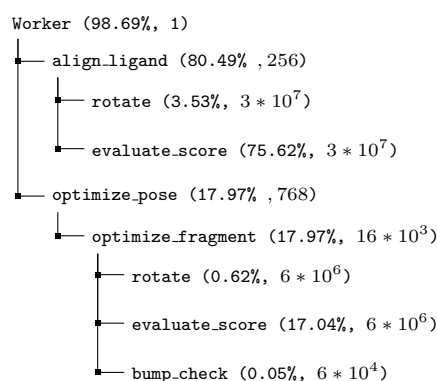
```
Worker (98.69%, 1)
│
├── align_ligand (80.49% , 256)
│      │
│      ├── rotate (3.53%, 3 * 10⁷)
│      │
│      └── evaluate_score (75.62%, 3 * 10⁷)
│
└── optimize_pose (17.97% , 768)
       │
       └── optimize_fragment (17.97%, 16 * 10³)
              │
              ├── rotate (0.62%, 6 * 10⁶)
              │
              ├── evaluate_score (17.04%, 6 * 10⁶)
              │
              └── bump_check (0.05%, 6 * 10⁴)
```

Fig. 1: The profiling of the application, in terms of percentage of execution time and number of visits.

of the *ligand*, and it aims at optimizing the displacement of its atoms (line 8). In this phase, each *fragment* of the *ligand* is sequentially rotated to optimize the shape of the *ligand* inside the target pocket. The complexity of the second phase depends on the structure of the *ligand*. *GeoDock* can manage *ligands* with an arbitrary number of fragments, however, the number of fragments influences the execution time. The optimization procedure is repeated to refine the optimization of the pose (lines 7–9).

3.2 Application Profiling

To better understand the application complexity and to identify hotspots, we profile *GeoDock* by using the Score-P [7] framework. Figure 1 shows the result of this analysis for the most significant functions. In particular, it reports the percentage of time spent in the evaluated function and the number of times that the function is called.

From the results, we noticed how the scoring function is the bottleneck of the application. This function is rather simple and it is already optimized for leveraging CPU architecture features, such as vector units. In fact, the call to the function consumes less than $100ns$. The problem lies in the number of times that we need to call the function to dock a *ligand* inside the *pocket* (10^7). Indeed, the scoring function evaluates "how good" is the current position of the *ligand* with respect to the *pocket*. Therefore, we need to call the scoring function after every change of the *ligand* shape.

From the profiling report, we might also notice how the functions that actually rotates the *ligand* atoms (`Rotate`) or that tests whether a pose is valid (`Bump_check`), have a negligible impact on the overall execution time, since they are also able to exploit the vector units of the CPU (Rotate) or are called less frequently (Bump_check). Indeed, being a greedy algorithm, the

bump_check function is called only when the score of the pose is better than the actual best score, otherwise the pose is immediately discarded.

The two main kernels, i.e. the alignment kernel (`align_ligand`, line 6 in Listing 1) and the pose optimization kernel (`optimize_pose`, line 8 in Listing 1), includes all these functions. In the evolution of the listings we will only show these two function (`align_ligand` and `optimize_pose`) to simplify the discussion. The internal function will not be shown again.

From the timing analysis, we might conclude how the performance of the application is not limited by a single complex function. The bottleneck of the algorithm is due to the algorithm complexity for computing the *ligand* pose, that needs to evaluate a high number of alternatives.

Moreover, since the algorithm is greedy, we need to perform multiple restarts to reduce the probability of settling with a local minimum. Therefore, since all of these operations are independent, it seems that the algorithm fits the parallel nature of the GPU. The main problem is that we do not have a single kernel to offload to the GPU, but we need to consider the whole docking algorithm for the single *ligand*.

4 GPU Accelerated Versions

In this section, we describe the proposed approach to accelerate *GeoDock* using GPUs. First, we describe in Section 4.1 how we accelerate the whole algorithm with GPUs, relying on OpenACC. Then, we describe in Section 4.2 the analysis that lead us to implement the hybrid OpenMP/OpenACC solution. In this second case, the idea is to allocate the workload on the heterogeneous resources according to different hardware capabilities, to improve the overall performance of the application.

4.1 OpenACC Implementation

This paper extends our previous work [18], where we developed a pure GPU version of the algorithm, where both the main kernels (alignment and pose optimization) are offloaded to the GPU. In particular, starting from the profiling analysis, we implemented a first version of the algorithm that aims at minimizing the data transfer while maintaining the application structure.

Figure 2 shows a graphical representation of the algorithm described in Section 3, highlighting sections of the algorithm that are independent, by using different boxes. We might consider each restart of the docking algorithm as a different initial pose. Given that every initial pose might proceed independently, we have the first level of parallelism to map on the GPU. Given an input *ligand*, it is possible to generate and to dock the *ligand* initial poses on the device side. All the phases of the docking algorithm are performed in parallel, on different data, and we only extract the result at the end of the algorithm. In this way, we transfer data only at the beginning and at the end of the docking algorithm (i.e. once in the lifetime of the *ligand*).
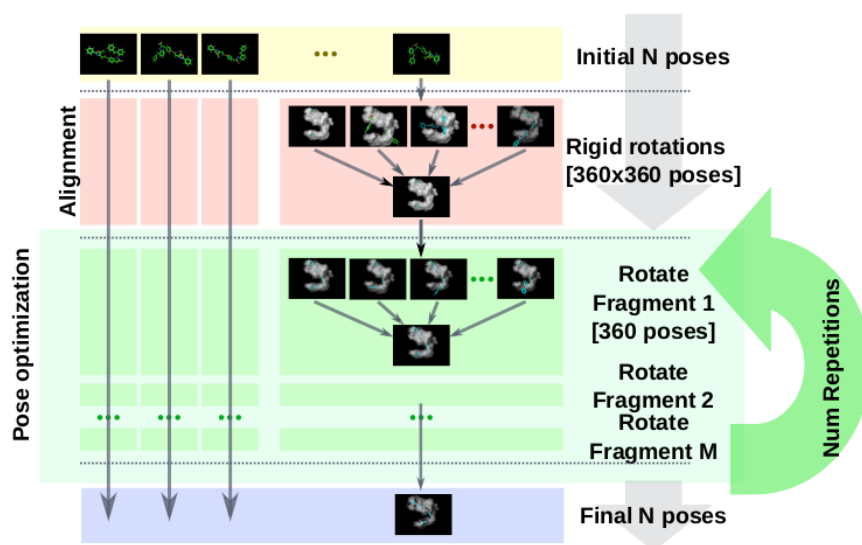
Fig. 2: Overview of the GPU implementation of the docking algorithm. Each box represents a computational part of the application that might be executed independently. The optimization phase must evaluate each *fragment* sequentially and the whole procedure might be repeated to refine the final result.

From the implementation point of view, we use OpenACC to avoid rewriting the application source code in a different language. OpenACC is able to operate with data structures that are resident on the GPU and usable across different kernels. However, we need to introduce the following changes in the source code of the application to be compliant with the OpenACC standard [12]. First of all, the data structures that interact with the offloaded kernels must manage data transfers in the constructor and destructor. In particular, the constructor allocates memory on the device side and it copies the initialized data into device memory. The destructor must free both the device and the host memory. Moreover, it is mandatory to mark each function called inside the offloaded region with the OpenACC `routine` directive.

Since our plan is to parallelize the computation over the initial poses, we need each pose in a different memory region to schedule all the iterations independently on the GPU. The OpenACC language provides the `private` keyword to express this concept. However, the system runtime available on our platform was not able to support this feature[1].

Therefore, we had to manually duplicate the data. With this modification, we were able to generate a correct binary. However, the GPU application was

---

[1] The *GeoDock* execution triggered an illegal accesses to the GPU memory when trying to transfer the private data structure.

Listing 2: Pseudo-code of the GPU algorithm.

```
1   load(pocket);
2   for (ligand : ligands)
3   {
4     ligand_t ligand_arr[N];
5     #pragma acc parallel loop
6     for (pose_id = 0; pose_id < N, pose_id++){
7       ligand_arr[pose_id] = ligand;
8     }
9     #pragma acc parallel loop gang
10    for (pose_id = 0; pose_id < N, pose_id++){
11      generate_starting_pose(ligand_arr[pose_id]);
12      #pragma acc worker
13      align_ligand(ligand_arr[pose_id], pocket);
14      #pragma acc loop seq
15      for(rep = 0; rep <num_repetitions; rep++){
16        #pragma acc worker
17        optimize_pose(ligand_arr[pose_id], pocket);
18      }
19    }
20  }
```

slower than the baseline version running on the CPU. We profiled *GeoDock* to identify the cause of the slowdown since all the memory used by the application is resident on the GPU, therefore data transfers introduce a negligible overhead. The result shows how the implementation is not able to exploit the parallelism. In fact, the number of multiple restarts of the algorithm does not expose enough computation to the GPU. In particular, using a single data structure to memorize the position of the *ligand* atoms in the alignment function limits the amount of exposed parallelism.

To obtain an advantage from the use of the accelerator, we had to rework the source code to find (and expose) more parallel computation. To achieve the desired result, we unified the rotation and scoring functions to avoid to store any temporary *ligand* poses. With this modification, that is not shown in the listing as explained in Section 3.2, we were able to expose more parallelism, since the rigid rotations are no more sequentially executed on a shared data structure. After the computation, we schedule a reduction to retrieve the *ligand* orientation with the best score. Finally, we rotate the *ligand* data structure accordingly, to forward to the optimization phase. Therefore, we are able to replicate the parallelism depicted in Figure 2. We applied the same technique to further expose parallel computation also in the pose optimization phase. However, the exposed parallelism is limited by two factors. On one hand, we rotate a *fragment* along with a 1-dimensional axis instead of a free rotation in a 3-dimensional space. On the other hand, we must optimize each *fragment* in sequence to save the *ligand* consistency. Therefore, we exploit again the pattern of parallel evaluation followed by a reduction, but the number of data is smaller: we need to perform a reduction at the end of every fragment evaluation. This phase is repeated num_repetitions times to improve the pose estimation.

Listing 2 describes the pseudocode of this GPU implementation. With respect to the baseline algorithm, we replicated the original *ligand* according to the number N of multiple restarts of the algorithm (lines 4–8). Once we initialize the memory on both the device and the host side, we evaluate each starting pose in the parallel region (lines 9–19) offloaded to the GPU. It is possible to notice how in the pseudocode there are no pragmas for transferring data between host and device. All the data transfers are managed with constructors and destructors of the data structures, according to the OpenACC standard. To leverage all the levels of parallelism available in the GPU, we inserted different levels of parallelism in the code as well. OpenACC offers three levels of parallelism: vector, worker, and gang. Vector level parallelism is the SIMT (Single Instruction, Multiple Threads) level on GPU. Gang level is the outer-most parallelism level, where all the elements are independent and the communication between gangs is forbidden. Worker is an intermediate level used to organize the vectors inside a gang. In particular, the vector and worker levels are the dimensions of a CUDA block, while the number of gangs is the CUDA grid. Therefore, we split the initial poses at gang level, since all of them are independent (line 9–10). We set all the internal functions (not shown in Listing 2) that change the position of the atoms at vector level. The intermediate functions (i.e. `align_ligand` and `optimize_pose`) are set at worker level (lines 12–13 and 16–17). The pose optimization loop (lines 15–18) is marked with a `loop seq` pragma. This is mandatory to force the compiler to execute that loop in a sequential way.

To evaluate the performance of this GPU version, we profiled the application and compared the results with the CPU baseline. In the alignment phase, we obtained a good speedup (16x). We noticed that the pose optimization was less suitable for GPU acceleration since too few operations per kernel were possible. Indeed, the sequentiality of the fragments and the control flow operations, inserted by the correctness checks, limit the reached speedup over the baseline CPU version. The final speedup for this kernel was of only 2x. Moreover, the profiling results show how the bottleneck of the application is changed. With the GPU version, approximately the 70% of the time is spent in the Optimize Pose kernel, while the Alignment takes less than 30%. This is a different result with respect to the profiling done on the baseline application on CPU.

Furthermore, we performed a parameter space exploration and tuning of the size of gangs, workers, and vectors to optimize the usage of the GPU, taking into account NVIDIA recommended best practices. From experimental results, the best size configuration for each function is:

– Alignment (`align_ligand`): 8 workers and 128 vector length.
– Pose optimization (`optimize_pose`): 64 workers with 1 as vector length.

This information confirmed us the hypothesis we made beforehand: the second kernel is less GPU-friendly than the first one. More in-depth analyses of this version are described in previous work [18].
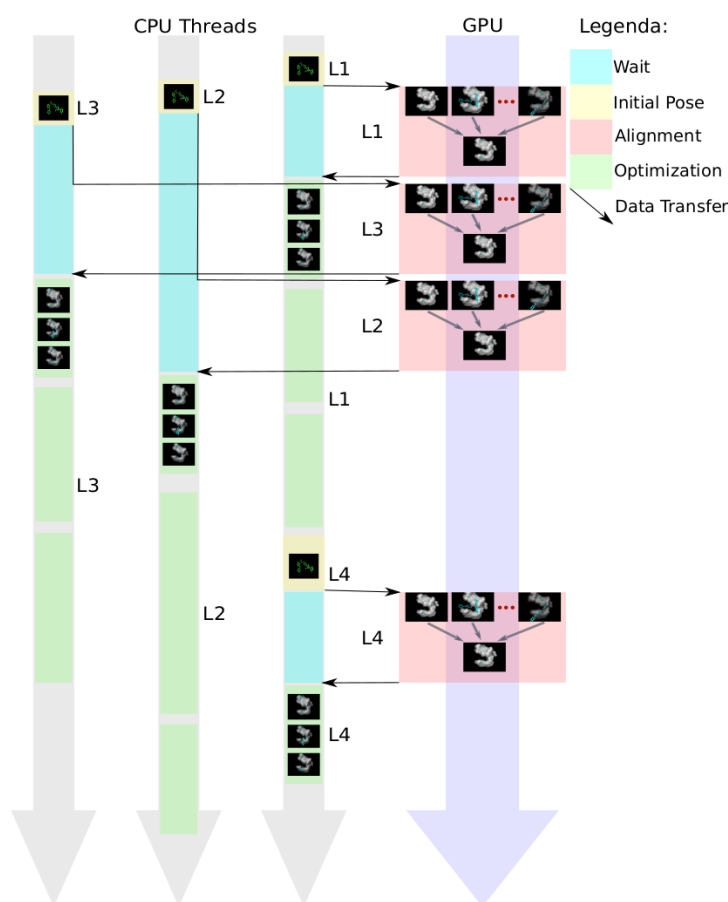
Fig. 3: Overview of the hybrid OpenMP/OpenACC implementation of the docking algorithm. The alignment phase of the *ligand* is offloaded to GPU, while the optimization phase is perfomed in the CPU. Each OpenMP thread interact with a single GPU. The arrows identifies data transfer between host and device.

4.2 Hybrid OpenMP/OpenACC Implementation

The *GeoDock* application implemented in GPU, and described in the previous section, has two main limits. On one hand, it is not able to use the available CPU cores to perform the computation. On the other hand, not all the phases of the application are able to fully exploit the architectural features of the GPU. As a consequence, the application is wasting or misusing a large fraction of the node computation capabilities. Given that our target is to optimize the performance of *GeoDock* on the full node, this section investigates the possibility to split the workload among CPU and GPU.

Listing 3: Pseudo-code of the hybrid algorithm.

```
 1   load ( pocket );
 2   #pragma omp parallel
 3   for ( ligand : ligands )
 4   {
 5      #pragma omp single nowait
 6      #pragma omp task
 7      {
 8         ligand_t ligand_arr [N];
 9         #pragma omp critical
10         #pragma acc data
11         {
12            #pragma acc parallel loop
13            for ( pose_id = 0; pose_id < N, pose_id++){
14               ligand_arr [ pose_id ] = ligand ;
15            }
16            #pragma acc parallel loop gang
17            for ( pose_id = 0; pose_id < N, pose_id++){
18               generate_starting_pose ( ligand_arr [ pose_id ]);
19               #pragma acc worker
20               align_ligand ( ligand_arr [ pose_id ], pocket );
21            }
22         }
23         for ( pose_id = 0; pose_id < N, pose_id++){
24            for( rep = 0; rep <num_repetitions; rep++){
25               optimize_pose ( ligand_arr [ pose_id ], pocket );
26            }
27         }
28      }
29   }
```

From the profiling information of the GPU implementation, instead of simply partitioning the data among CPU processes and GPU processes, we modified the algorithm to bring the pose optimization phase back to the CPU. In particular, we would like to exploit the multicore architecture, enabling each CPU thread to evaluate one *ligand*, and offloading only the alignment kernel to the GPU. The basic idea, depicted in Figure 3, is to exploit the GPU for the kernel that benefits most of massive-parallel architecture, while mapping the other kernels to the CPU. Each CPU thread takes care of a different *ligand*, avoiding data movement among threads, therefore maximizing also the parallelism determined by the *ligand* library. The only data movements are between CPUs and GPUs.

In this *GeoDock* implementation, we use OpenACC for the GPU kernel programming, while we exploit OpenMP for the CPU-level parallelism. Listing 3 shows the pseudocode of the algorithm. The outermost loop that iterates over the *ligand* library is parallelized using an OpenMP parallel region (line 2), where for every different *ligand* we create a `single nowait` task (lines 5–6). *Task* is a construct that was introduced in OpenMP 3 and it is used to describe parallel jobs leaving the organization of the parallelism to the scheduler. They are particularly effective for parallelizing irregular algorithms. The `single` keyword specify that an OpenMP region (in this case the *task*) executes a single instance of the related region. It is required to enforce that each *task* is in charge of an iteration of the outermost loop. The `nowait` keyword

is used to skip the implicit barrier at the end of the single region. In this way the thread does not wait for the completion of the task, but can execute the next iteration of the loop.

The execution time spent for docking a *ligand* depends on several factors, such as the number of atoms, the number of fragments, and geometrical properties of both the target *pocket* and *ligand*. Since these factors might drastically change between *ligands* of the same library, we might consider our docking algorithm as an irregular application. Therefore, the proposed implementation leverages the tasks construct to create a task for every *ligand* to be docked. As soon as an OpenMP thread becomes free, a pending task is assigned to it, until there is a task waiting to be executed. Moreover, we use the tied task implementation to limit migration, restraining a task to be executed on the same thread that generated it. Moreover, we bind each OpenMP thread to a physical core, by using the OpenMP environment variable `OMP_PLACES=cores`. In this way, we are able to associate a *ligand* to one physical core, avoiding the extra movement of the data.

Beside the *Task* construct, we tried to exploit other OpenMP strategies to make the application parallel. In particular, we evaluated the *taskloop* keyword, that is a construct used to create one task for each iteration of a loop, and the traditional *parallel for*. The first construct is not supported by PGI 17.10, so we were not able to exploit it. The *parallel for* has empirically shown slightly worse throughput in the considered case. We also tried all the OpenMP scheduling algorithms (static, dynamic and guided) and none of them was able to improve the performance obtained with the task construct.

The GPU kernel is implemented inside an OpenMP `critical` region (line 9) to avoid race conditions. We also considered to use OpenACC features, such as asynchronous queues, however they performed worse than this implementation. We exploit the implicit barrier at the end of the parallel region to enforce thread synchronization at the end of the library of *ligands* to be docked.

The *GeoDock* algorithm implementation is similar to the one described in Section 4.1. In particular, the data replication (lines 12–15) and the alignment phase (lines 16–21) are almost the same. The only difference is in the data structure implementation, due to the limited support of C++ standard libraries from OpenACC. For this reason, we manually managed data copies before and after the critical section used to offload the alignment to the GPU. These changes are omitted in the application pseudocode. However, we encountered a key issue in the memory management of the hybrid solution. In the GPU version, we used CUDA unified memory to reduce the impact of data organization on the application developer. This feature enables addresses accessible from different types of architectures (normal CPU and CUDA GPU cores) hiding the complexity of the management from the programmer. If we use this implementation, the Unified Memory support for the Kepler architecture fails to properly allocate memory[2]. To solve this issue, we manually

---

[2] When we enable the multi-threading with OpenMP, the CUDA managed memory fails. The manager tries to allocate the memory, from different threads, in the same area and returns a runtime error.

Listing 4: Pseudo-code of the hybrid multi-GPU algorithm.

```
1   load(pocket);
2   omp_lock_t lock_array[N_GPUS];
3   #pragma omp parallel
4   for (ligand : ligands)
5   {
6     #pragma omp single nowait
7     #pragma omp task
8     {
9       ligand_t ligand_arr[N];
10      omp_set_lock(lock_array[tid%N_GPUS]);
11      #pragma acc set device_num(tid%N_GPUS)
12      #pragma acc data
13      {
14        #pragma acc parallel loop
15        for (pose_id = 0; pose_id < N, pose_id++){
16          ligand_arr[pose_id] = ligand;
17        }
18        #pragma acc parallel loop gang
19        for (pose_id = 0; pose_id < N, pose_id++){
20          generate_starting_pose(ligand_arr[pose_id]);
21          align_ligand(ligand_arr[pose_id], pocket);
22        }
23      }
24      omp_unset_lock(lock_array[tid%N_GPUS]);
25      for (pose_id = 0; pose_id < N, pose_id++){
26        for(rep = 0; rep <num_repetitions; rep++){
27          optimize_pose(ligand_arr[pose_id], pocket);
28        }
29      }
30    }
31  }
```

manage the memory allocation and transfers, by using OpenACC pragmas. For this reason, we created a data region around the offloaded kernels (line 10). The pose optimization kernel is no more decorated with pragmas (lines 23–26) because it is executed on the CPU, therefore we need to iterate over the aligned poses (line 23).

Tuning considerations. The hybrid approach requires a careful tuning to efficiently exploit the computing resources of the heterogeneous node. In fact, we can highlight two possible problems: GPU idle time and CPU thread waiting time. In the first case, the CPU threads are not able to provide enough data to fully exploit the GPU, leading to resource underutilization. It is possible to notice this effect in Figure 3 on the GPU side. After the execution of the alignment phase of the *ligand* L2, all the other CPU threads are still busy on the pose optimization phase. Therefore, The GPU is in idle state until the alignment of the *ligand* L4 is offloaded. The second problem happens when there are too many CPU threads and the GPU is overloaded. In this case, each CPU thread can have a long waiting time before accessing the GPU to offload the alignment kernel. In Figure 3 we can notice an example of this problem at the beginning of the execution, where the *ligand* L2 and the *ligand* L3 are waiting for alignment of the *ligand* L1 to end. For these two reasons, balancing the load between CPU and GPU is very important. In the experimental
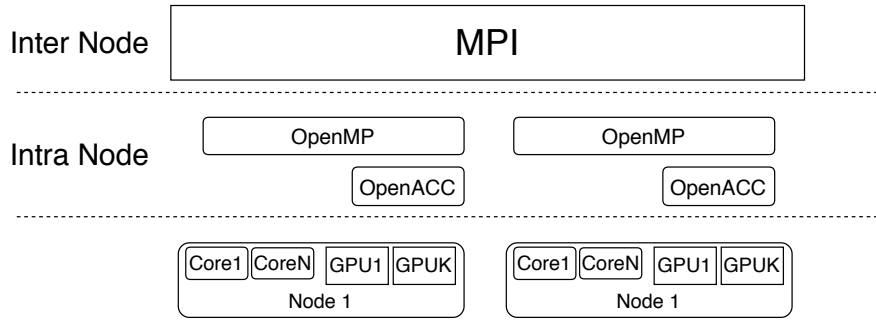
Fig. 4: Complete organization of the application.

results, we show how we tuned of the number of CPU threads to optimize the full node performance.

Multi-GPU. The considerations on the application tuning are even more important when we address multi-GPU nodes. From the implementation point of view, to distribute the workload across multiple devices, it is enough to provide different values to the `#pragma acc set device_num(...)`. We used the thread number to decide on which GPU the thread will offload the kernel.

Moreover, we substituted the original critical section with an OpenMP mutex. This gave us the possibility to exploit the parallelism in the kernel offloading having one kernel in each GPU. The algorithm is reported in Listing 4. In particular, we set the device using the related OpenACC pragma (line 11), after locking the mutex (line 10). In this way, a set of threads is associated with a single GPU. As already mentioned, tasks are associated with a thread only when they start executing, and not at their creation. This characteristic of tasks manages the load balancing.

Multi-Node. *GeoDock* has been designed to run on large HPC machines, exploiting MPI for inter-node communication. The complete application organization is reported in Figure 4. OpenMP and OpenACC are used to manage CPU and GPU parallelism within the node as described in previous sections. MPI is used to handle data parallelism across the different nodes. We employ a pure Master-Slave paradigm, where the master process dispatches groups of *ligands* to the slave processes. However, the focus of this work is the intra-node optimization, since the advantages obtained on a single node are replicated in all the involved nodes. For this reason, we will run and analyze all the experiments on a single node, neglecting the MPI overheads due to the communication with the master process.

(a) Pure CPU   (b) Pure GPU   (c) Hybrid
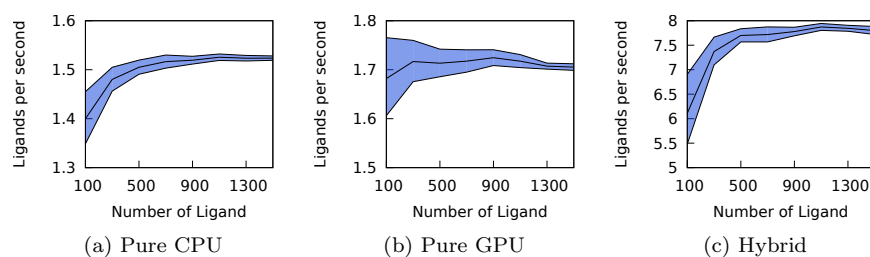
Fig. 5: Analysis of the stability of the time required to dock a ligand.

5 Experimental Result

We performed the experimental campaign using one GPU node of the GALILEO2 machine at CINECA[3]. The target node is equipped with a 2x8-core Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz CPU and two NVIDIA Tesla K80 GPU cards. The operating system was CentOS 7.0, and we compiled the program using PGI Compiler 17.10 enabling the *fastsse* flag to activate the vectorization on top of the *O3* optimization level.

The data shown in this section are the results of several runs using different sets of *ligands* and target *pocket* to keep into consideration possible performance variability. The evaluation has been done considering a large set of *ligands*. The *ligands* complexity was different in terms of number of atoms and fragments. In particular, the number of atoms per *ligand* across the whole set is on average 39.6 with a standard deviation of 6.7. The maximum number of atoms is 73, and the minimum is 28. The average number of fragments is 13.3 with a standard deviation of 4.3. The maximum number of fragments is 34 while the minimum is 6.

Given that the application performance is dependent on the *ligand* complexity, our first experiment wants to define a reasonable number of *ligands* that is large enough to absorb those differences. Figure 5 shows the convergence analysis done by increasing the size of the set of *ligands* used for the experiment. In particular, we performed several runs with a different number of *ligands*, and we measure the application throughput reporting the average and the standard deviation. We can see that all the three different versions of the code can be considered stable with few *ligands* to dock (>500). In all the next experiments we used 1500 *ligands*, to further absorb the possible performance variance. This set of *ligands* can be seen as a workload of a slave MPI process running on a single node in the context of a larger master-slave MPI application.

To easily and fairly compare the different implementations we presented, we need to define the terminology that we use in the following sections. We use the term CPU process CPU^{proc} when we refer to an MPI process executing the
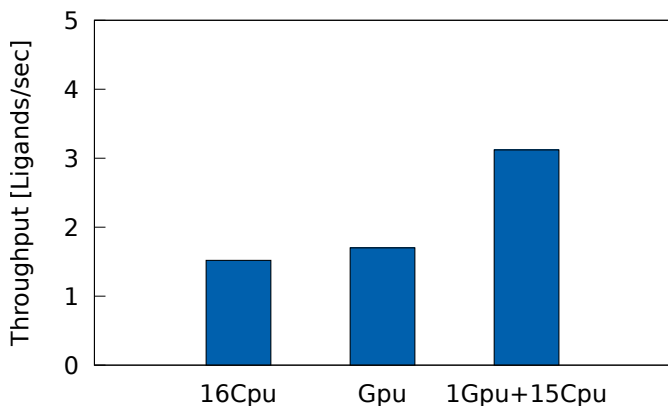
---

[3] http://www.hpc.cineca.it/hardware/galileo-0

Fig. 6: Throughput of the *GeoDock* application considering a single GPU, by varying its configuration.

baseline CPU version of the docking algorithm, described in Section 3. We use the term GPU process GPU^{proc} when we refer to an MPI process executing the OpenACC version presented in Section 4.1, which uses one CPU thread and one GPU. Finally, we use the term hybrid process $HY^{proc}_{\#ompTh,\#GPUs}$ when we refer to an MPI process executing the OpenMP/OpenACC version described in Section 4.2. In particular, $\#ompTh$ is the number of OpenMP threads, while $\#GPUs$ is the number of used GPUs.

5.1 Single GPU

This section analyzes the performance of the proposed hybrid implementation of *GeoDock*, focusing on a single GPU case, to compare with previous implementations. The first experiment aims at defining a baseline throughput, in terms of ligands per second, using the reference dataset. Figure 6 shows the throughput computed with different configurations of *GeoDock* implementations: (i) by using all the CPUs ($16 \times CPU^{proc}$); (ii) by using the GPU ($1 \times GPU^{proc}$); (iii) by using the GPU and the remaining CPU cores ($15 \times CPU^{proc} + 1 \times GPU^{proc}$). The number of CPU^{proc} and the sum of CPU^{proc} and GPU^{proc} has been kept equal to the number of cores available in the node, because having more OpenMP threads than CPU cores does not increase the performance. The results show how a single GPU^{proc} has a speedup of 1.1x with respect the original application that exploits all the CPUs of the node ($16 \times CPU^{proc}$). Therefore, if consider the configuration that uses the GPU and the CPU cores ($15 \times CPU^{proc} + 1 \times GPU^{proc}$), we are able to achieve a speed-up of 2x. We use this third configuration as baseline for comparing the proposed hybrid approach.
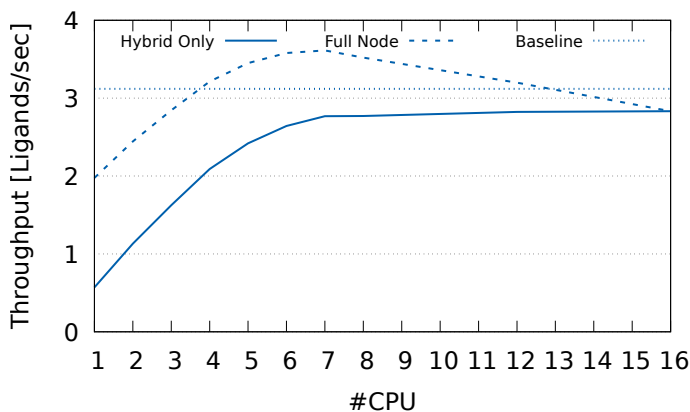
Fig. 7: Scaling analysis of the hybrid approach in terms of throughput, by changing the number of OpenMP threads. Baseline represents the $15 \times CPU^{proc} + 1 \times GPU^{proc}$ application.

The second experiment aims at analyzing the performance of the hybrid solution, by varying the number of OpenMP threads from 1 to 16. Figure 7 shows the experimental results. In particular, the x-axis represents the number of the OpenMP threads used in the evaluated configuration ($HY_{n,1}^{proc}$, with n={1,...,16}). The y-axis represents the reached throughput. The solid line represent the throughput reached only by the $HY_{x,1}^{proc}$, while the dashed line represents the throughput when it is combined with a number of CPU processes sufficient to completely fill the node, i.e. $HY_{n,1}^{proc} + (16-n) \times CPU^{proc}$.

The results can be split into two different regions. On the left side of the figure, we can see that increasing the number of OpenMP threads up to 7 the performance of $HY_{n,1}^{proc}$ almost linearly increases. This is mainly due to the increment of GPU usage. Few CPU threads are not able to fully exploit the GPU. On the other hand, starting from 7 OpenMP threads the $Hy_{n,1}^{proc}$ performance reaches a saturation point since the GPU starts to be the bottleneck. Indeed, at 7 OpenMP threads, the GPU is already fully used and with more CPU threads feeding it does not increment the throughput. Similarly, the performance of $HY_{n,1}^{proc} + (16-n) \times CPU^{proc}$ reaches the maximum throughput when the saturation for the hybrid version happens (i.e. $HY_{7,1}^{proc} + 9 \times CPU^{proc}$). After this configuration, the performance of $HY_{n,1}^{proc} + (16-n) \times CPU^{proc}$ reduces while increasing the number of the OpenMP threads for the hybrid version. If we use more CPU threads for the hybrid version, we are reducing the number of the CPU process we can exploit and thus their cumulative throughput contribution. The result for the optimal configuration reports a speedup of 1.16x with respect to the $15 \times CPU^{proc} + 1 \times GPU^{proc}$ configuration, which was set as our baseline.
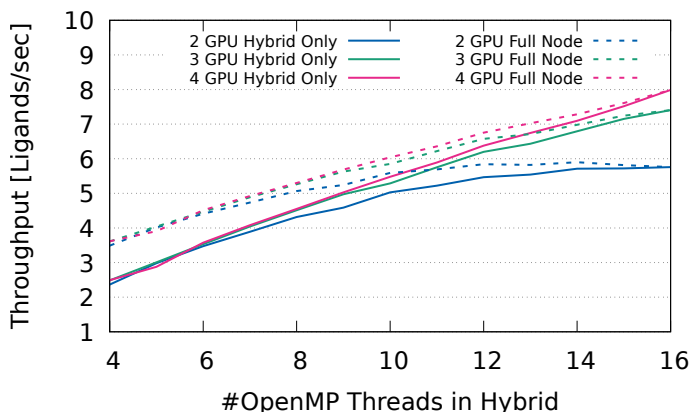
Fig. 8: Scaling analysis of the hybrid approach in terms of throughput, by changing the number of OpenMP threads and the number of GPUs.

5.2 Multi-GPUs

This experiment analyses the performance of the hybrid approach according to the number of available GPUs in the target node. The analysis is done by considering up to 4 GPUs which is the limit of our node (each K80 card includes 2 GPUs).

Figure 8 shows the application throughput of the hybrid process while varying the number of OpenMP threads (x-axis) and the number of GPUs. In particular, solid lines represent *GeoDock* configurations that uses only hybrid processes ($HY_{n,k}^{proc}$, with n={4, ..., 16} and k={2, 3, 4}). While dashed lines represent the full node behaviour, where it uses CPU^{proc} for the unused cores, i.e. the $HY_{n,k}^{proc} + (16 - n) \times CPU^{proc}$ configuration.

If we focus on *GeoDock* configurations that use only the hybrid approach with a node composed of two GPUs, experimental results shows how the application has an almost linear growth up to 8 cores (from 2.5 ligands per second to 4.5 ligands per second). Then, the throughput gain slows down and it is almost negligible if we increase the number of OpenMP threads from 12 to 16 (the throughput ends at 5.8 ligands per second). On the other hand, if we focus on *GeoDock* configurations that use 3 and 4 GPUs, we have a steady growth in the application throughput over the entire range of OpenMP threads. We might conclude that with the number of OpenMP threads we considered, the GPUs are under-utilized. In particular, the *GeoDock* configurations that use 3 GPUs are slowing down a bit only in the last part of the plot, while in the *GeoDock* configurations that use 4 GPUs the throughput grows almost linearly.

The dashed lines represent the usage of CPU^{proc} for the spare cores. The results show a similar trend. The main difference lies in the first part of the plot, where there are few OpenMP threads employed in the hybrid approach.
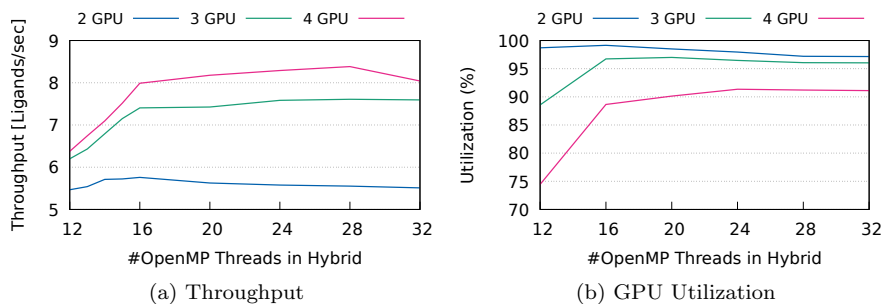
(a) Throughput

(b) GPU Utilization

Fig. 9: Analysis of the application when using more OpenMP threads than CPU availables.

The maximum throughput for the full node is the maximum point of these lines. We can notice from the picture that, according to the number of GPUs, the highest point of the functions is reached by the following configurations: $HY_{14,2}^{proc} + 2 \times CPU^{proc}$, $HY_{16,3}^{proc}$, and $HY_{16,4}^{proc}$.

To improve the GPU utilization when we have access to three and four GPUs, we empirically evaluate the benefits of using more OpenMP threads than available cores. Figure 9 depicts the application throughput (Figure 9a) and GPU utilization (Figure 9b) of the multi-GPU hybrid approach. In this experiment, we evaluated a number of OpenMP threads between 12 (less than the number of cores) and 32 (two times the number of cores). When we have access to two GPUs, the application reaches the peak throughput by using 16 OpenMP threads, which is equal to the number of available cores. This is an expected result since the application completely utilizes the GPUs without overloading the system with OpenMP threads. However, when we have access to more than two GPUs, the application reaches the peak throughput by using a number of OpenMP threads greater than the available cores. This benefit is due to the increased utilization of the GPUs. Indeed, the operating system is able to serve an available worker while a second thread waits for the completion of the kernel on the GPU. In particular, with three GPUs the best configuration is $HY_{24,4}^{proc}$, while with four GPUs the best configuration is $HY_{28,4}^{proc}$. If we increase the number of OpenMP threads above such configurations, the context switch overhead hinders the application performance.

To conclude our performance analysis, Figure 10 shows the comparison of the best configurations exploiting the full heterogeneous node. As *baseline* we used $k \times GPU^{proc} + (16 - k) \times CPU^{proc}$ (where k is the number of available GPUs) and as *proposed* we select the best configuration obtained with the hybrid approach according to the previous analyses. In all cases, by varying the number of available GPUs, the configurations including the hybrid version have a higher throughput. This is due to the best exploitation of the GPUs only for the kernels where there is a higher speedup. In particular, the performance improvement in the case of 1, 2, 3 and 4 GPUs is respectively of 15%, 25%,
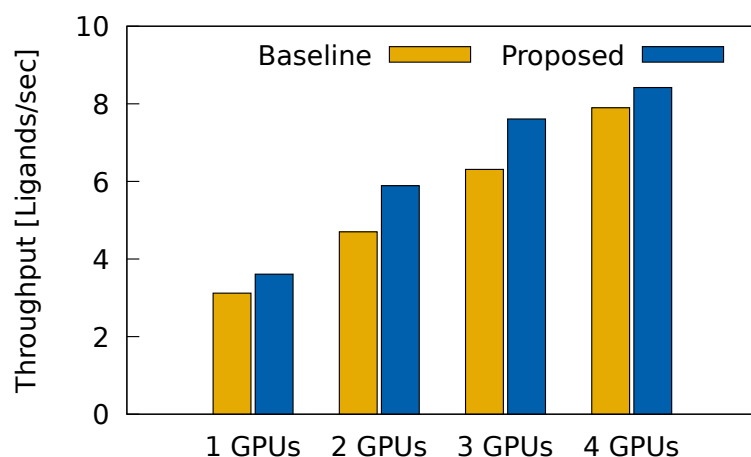
Fig. 10: Comparison of the baseline configuration of *GeoDock* with the proposed hybrid approach, by changing the number of GPUs. The baseline is the configuration with $k \times GPU^{proc} + (16 - k) \times CPU^{proc}$.

20%, and 6%. As mentioned before, in the case of 4 GPUs the performance speedup with respect to the baseline is lower because the 16 cores are not enough to fully exploit all the GPUs.

6 Conclusion

In the drug discovery process, the virtual screening of a huge library of ligands is a crucial job. The benefits of an improvement in the execution time of this task, are twofold. On one side it might reduce the cost of the process. On the other side, it enables the end-user to increase the number of the evaluated *ligands*, increasing the probability of finding a better solution.

In this paper, we focused on a geometric docking application, *GeoDock*, optimized for the CPU architecture of an HPC platform, and we ported it to GPU architectures. First, we leveraged OpenACC to port the most intensive kernels on the GPU. Then, we modified the application flow, by limiting the GPU usage only to the kernels with the largest speedup, thus splitting the workload among the heterogeneous resources. This has been done by implementing a hybrid solution combining OpenMP and OpenACC to organize the computations. We performed an experimental campaign to evaluate the scaling of *GeoDock* when using all the available resources of the node. The experimental results have shown how the usage of the hybrid OpenMP/OpenACC version of *GeoDock* guarantees a better exploitation of the resources of the node reaching a performance improvement up to 25%.

References

1. Beato, C., Beccari, A.R., Cavazzoni, C., Lorenzi, S., Costantino, G.: Use of experimental design to optimize docking performance: The case of ligendock, the docking module of ligen, a new de novo design program. Journal of Chemical Information and Modeling **53**(6), 1503–1517 (2013)
2. Dagum, L., Menon, R.: Openmp: an industry standard api for shared-memory programming. Computational Science & Engineering, IEEE **5**(1), 46–55 (1998)
3. Ewing, T.J., Makino, S., Skillman, A.G., Kuntz, I.D.: Dock 4.0: search strategies for automated molecular docking of flexible molecule databases. Journal of computer-aided molecular design **15**(5), 411–428 (2001)
4. Farber, R.: Parallel programming with OpenACC. Newnes (2016)
5. Friesner, R.A., Banks, J.L., Murphy, R.B., Halgren, T.A., Klicic, J.J., Mainz, D.T., Repasky, M.P., Knoll, E.H., Shelley, M., Perry, J.K., et al.: Glide: a new approach for rapid, accurate docking and scoring. 1. method and assessment of docking accuracy. Journal of medicinal chemistry **47**(7), 1739–1749 (2004)
6. Jain, A.N.: Surflex-dock 2.1: robust performance from ligand energetic modeling, ring flexibility, and knowledge-based search. Journal of computer-aided molecular design **21**(5), 281–306 (2007)
7. Knüpfer, A., Rössel, C., Mey, D.a., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A., Nagel, W.E., Oleynik, Y., Philippen, P., Saviankou, P., Schmidl, D., Shende, S., Tschüter, R., Wagner, M., Wesarg, B., Wolf, F.: Score-p: A joint performance measurement run-time infrastructure for periscope,scalasca, tau, and vampir. In: H. Brunst, M.S. Müller, W.E. Nagel, M.M. Resch (eds.) Tools for High Performance Computing 2011, pp. 79–91. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
8. Kramer, B., Rarey, M., Lengauer, T.: Evaluation of the flexx incremental construction algorithm for protein-ligand docking. Proteins: Structure, Function, and Bioinformatics **37**(2), 228–241 (1999)
9. Markidis, S., Gong, J., Schliephake, M., Laure, E., Hart, A., Henty, D., Heisey, K., Fischer, P.: Openacc acceleration of the nek5000 spectral element code. The International Journal of High Performance Computing Applications **29**(3), 311–319 (2015). DOI 10.1177/1094342015576846
10. Morris, G.M., Huey, R., Lindstrom, W., Sanner, M.F., Belew, R.K., Goodsell, D.S., Olson, A.J.: Autodock4 and autodocktools4: Automated docking with selective receptor flexibility. Journal of computational chemistry **30**(16), 2785–2791 (2009)
11. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with cuda. Queue **6**(2), 40–53 (2008)
12. OpenACC.org: Openacc programming and best practices guide (2015). URL https://www.openacc.org/sites/default/files/inline-files/OpenACC_Programming_Guide_0.pdf
13. Sawadsitang, S., Lin, J., See, S., Bodin, F., Matsuoka, S.: Understanding performance portability of openacc for supercomputers. In: 2015 IEEE International Parallel and Distributed Processing Symposium Workshop, pp. 699–707 (2015). DOI 10.1109/IPDPSW.2015.60
14. Stone, J.E., Gohara, D., Shi, G.: Opencl: A parallel programming standard for heterogeneous computing systems. IEEE Des. Test **12**(3), 66–73 (2010). DOI 10.1109/MCSE.2010.69. URL http://dx.doi.org/10.1109/MCSE.2010.69
15. Thomsen, R., Christensen, M.H.: Moldock: a new technique for high-accuracy molecular docking. Journal of medicinal chemistry **49**(11), 3315–3321 (2006)
16. Top500.org: Top 500 list (2018). https://www.top500.org/lists/2018/11/

17. Trott, O., Olson, A.J.: Autodock vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading. Journal of computational chemistry **31**(2), 455–461 (2010)
18. Vitali, E., Gadioli, D., Palermo, G., Beccari, A., Silvano, C.: Accelerating a geometric approach to molecular docking with openacc. In: Proceedings of the 6th International Workshop on Parallelism in Bioinformatics, PBio 2018, pp. 45–51. ACM, New York, NY, USA (2018). DOI 10.1145/3235830.3235835. URL http://doi.acm.org/10.1145/3235830.3235835
19. Wang, R., Lai, L., Wang, S.: Further development and validation of empirical scoring functions for structure-based binding affinity prediction. Journal of computer-aided molecular design **16**(1), 11–26 (2002)
20. Xu, R., Chandrasekaran, S., Chapman, B.: Exploring programming multi-gpus using openmp and openacc-based hybrid model (2013). DOI 10.1109/IPDPSW.2013.263
21. Xu, R., Tian, X., Chandrasekaran, S., Chapman, B.: Multi-gpu support on single node using directive-based programming model. Sci. Program. **2015**, 3:3–3:3 (2016). DOI 10.1155/2015/621730. URL https://doi.org/10.1155/2015/621730
22. Yan, Y., Lin, P.H., Liao, C., R. de Supinski, B., J. Quinlan, D.: Supporting multiple accelerators in high-level programming models pp. 170–180 (2015). DOI 10.1145/2712386.2712405
23. Yan, Y., Liu, J., W. Cameron, K., Umar, M.: Homp: Automated distribution of parallel loops and data in highly parallel accelerator-based systems pp. 788–798 (2017). DOI 10.1109/IPDPS.2017.99