Accelerating a Geometric Approach to Molecular Docking with OpenACC

Emanuele Vitali*, Davide Gadioli*, Gianluca Palermo*, Andrea Beccari+, Cristina Silvano*

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

+Dompé Farmaceutici SpA, L'Aquila, Via Campo di Pile, 67100, Italy

ABSTRACT

In a drug discovery process, the Molecular Docking task aims at estimating the three-dimensional pose of a molecule when it interacts with the target protein. This task is usually used to perform a screening on a large library of molecules to find the most promising candidates. The output of this task is used to estimate the actual strength of atomic interactions. In this document we focus on an application that performs molecular docking using geometrical features of the molecule and of the protein, to quickly screen the target chemical library.

Due to the size of the chemical library and to the complexity of the task, the application is a typical batch job that runs in an HPC platform, optimized for CPU processing. Given the amount of parallelism of this application, we evaluate the possibility to run such application on a GPU node, leveraging the OpenACC directive language.

Preliminary results show that we are able to achieve a significant speedup on the kernel that was the bottleneck on the CPU (up to 16x), while we achieve a modest speedup on the overall execution (5x).

CCS CONCEPTS

• **Computing methodologies → Massively parallel and high-performance simulations**; • **Software and its engineering →** *Software performance*; • **Applied computing →** Computational biology;

KEYWORDS

Molecular Docking, GPU, OpenACC, HPC

ACM Reference Format:

Emanuele Vitali*, Davide Gadioli*, Gianluca Palermo*, Andrea Beccari+, Cristina Silvano*. 2018. Accelerating a Geometric Approach to Molecular Docking with OpenACC. In *PBio 2018: 6th International Workshop on Parallelism in Bioinformatics (PBio 2018), September 23, 2018, Barcelona, Spain.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3235830.3235835

1 INTRODUCTION

Drug discovery is a complex process aimed to find new drugs. This process comprehends different phases ranging from computer simulations to test *in vivo*. Molecular Docking is a phase of this process performed *in silico*, working on two different inputs. On the main hand, we have the target binding site of a protein, named *pocket* . On the other hand, we have a molecule, named *ligand*, that interacts with the *pocket* . The main goal of this phase is to estimate the three-dimensional displacement of the *ligand* 's atoms, after the interaction with the target *pocket* , i.e. the pose of the *ligand* after the docking in the binding site of the *pocket* .

Molecular Docking is usually employed in two different tasks of the drug discovery process. It might be used to perform an accurate simulation for estimating the correct *ligand* pose to forward on later stages of the discovery process. However, it might also be used to perform a virtual screening of a given chemical library of molecules. In this scenario, we have as input a single *pocket* and a very large number of *ligands* to evaluate. The output of this task is the set of the most promising *ligands* to consider in the more accurate simulation. In this paper, we focus on the latter task.

Given that it is possible to process each *ligand* of the chemical library in an independent way, this problem is data parallel, therefore it is embarrassing parallel. The complexity of this task is due to the high number of *ligands* to evaluate and to the computational effort required to evaluate a single *ligand* -*pocket* pair. In fact, a subset of the *ligand* chemical bonds generates two disjoint set of atoms, i.e. *fragments*, that can be rotated independently from each other, changing the shape of the *ligand* . Therefore, on top of the six degrees of freedoms to move a rigid body in a three-dimensional space, the docking process shall consider the internal degrees of freedom of each *ligand* . Moreover, the final pose of the *ligand* depends on chemical and geometric properties of both, the *pocket* and the *ligand* .

In the context of the *LiGen* [2] work-flow, the application that performs the molecular docking, *LiGenDock* [1], employs a two-phases approach. At first, it considers geometric features for docking a *ligand* , to filter out the incompatible ones. Then, it simulates the chemical and physical interaction between the *pocket* and the remaining *ligands* to further reduce the number of promising *ligands*.

Due to the complexity and parallelism of the screening task, *LiGenDock* is a typical example of batch application optimized for homogeneous HPC platforms. In the last decade, energy consumption has become an important issue also in the HPC domain For this reason, a switch from homogeneous systems to heterogeneous systems has begun. By using different hardware accelerators, such as GPUs or Xeon-phi, heterogeneous systems usually have better energy efficiency and are able to provide more FLOPs. Indeed, most of the top positions in the Green500 list (as of November 2017, [6]) are occupied by heterogeneous machines. The improvement in efficiency often implies an increment on programming complexity, since application developers must use different paradigms to leverage features of these co-processors. In particular, GPUs have many computational cores that expose a much higher level of parallelism than CPUs. However, with respect to CPUs cores that have complex features, such as out-of-order and speculative execution, GPUs cores have a simpler architecture. Therefore, complex code and control flow operations lead to a significant degradation of the performance of a GPU application.

LiGenDock is able to leverage the parallelism on the *ligand* level using a classic MPI master/slave approach. However, in this paper we investigate the possibility to offload the geometrical docking kernel on GPU, to leverage its internal parallelism for decreasing the time to solution, leading to two main benefits for the end-user. On one hand, it decreases the monetary cost of the drug discovery process. On the other hand, it enables an increment of the number of *ligands* analyzed by *LiGen*, increasing the probability to find a good candidate.

To summarize, the main contributions of this work are the following:

- We create a kernel for accelerating a geometric docking application, using OpenACC directives.
- We analyze the obtained performance.
- We discuss the language and algorithm limits, comparing it with the CPU baseline.

The article is structured as follows: Section 2 describes related works, Section 3 describes the algorithm and the methodology that we used to create the accelerated kernel. In section 4 we present and describe the experimental results. Finally, section 5 concludes the paper.

2 RELATED WORK

Molecular Docking is a well-known problem in literature which is addressed using stochastic and deterministic approaches. Stochastic approaches leverage genetic algorithms [17], [18] or Monte Carlo simulations [7] to derive the *ligand* pose. These approaches are interesting, however, the final output may change between the subsequent execution of the application. Since expensive laboratory tests depend on the outcome of the screening task, pharmaceutical companies usually value repeatable results.

Deterministic algorithms for molecular docking leverage chemical and geometrical properties to drive the docking process, dealing with the *ligand* flexibility. Examples in this category are Dock 4.0 [4], FLEXX [10], Surflex-Dock 2.1 [8] and LiGenDock [1].

To harness GPU capabilities, application developers may choose between two main approaches. In the first approach, they use specific computing languages, such as CUDA [11] or OpenCL [16], for writing device code and for managing data transfers. Those languages provide to application developers the finest control of the computation. However, even if the language is based on C/C++, they require to rewrite the algorithm according to the memory model and the parallelization scheme of the chosen language. Moreover, they introduce a maintainability problem since the device code is not usually suitable for running on the host device, that leads to code duplication.

A second approach is to decorate the original source code with compiler directives to highlight the region of code to offload and to describe data transfers between the host and device memory. The compiler generates automatically the device code and the required glue code for data transfer. The benefit of this approach is that the application is written in a single language, which may run on the device and on the host as well. However, since the device code is automatically generated, it may suffer from performance penalties. Moreover, application developers are still in charge of exposing enough parallelism and of minimizing control flow operations, to have a performance improvement.

In this work we use the directive language OpenACC [5] to exploit GPU capabilities. Moreover, the original CPU code was already designed to expose parallelism, to leverage the CPU vector units.

Algorithm 1: Pseudo-code of the original algorithm that performs the geometrical docking for the CPU.

```
Input: Target Pocket and the initial pose of the ligand
Output: The geometric score of the evaluated poses
1  repeat
2      $Generate\_Starting\_Pose(Pose\_id)$;
3      for $angle\_x$ in range(0:360) do
4          $Rotate(angle\_x, Pose\_id)$;
5          for $angle\_y$ in range(0:360) do
6              $Rotate(angle\_y, Pose\_id)$;
7              $Evaluate\_Score(Pose\_id)$
8          end
9      end
10     for $fragment$ in $ligand\_fragments$ do
11         for $angle$ in range(0:360) do
12             $Rotate(fragment, angle, pose\_id)$
                 $Bump\ Check(fragment, pose\_id)$
                 $Score(fragment, pose\_id)$
13         end
14     end
15 until $Pose\_id < N$;
```

3 METHODOLOGY

This section describes the approach that we followed to accelerate the geometrical docking kernel of the docking application on GPUs. First, we analyze the application to identify opportunities to offload computation to the GPU. Then, we describe how we seized those opportunities to improve the application performances.

3.1 Application Description

LiGenDock application uses a mixed approach for docking a *ligand* in the target *pocket* . It starts considering geometric features, then it simulates the actual physical and chemical interaction for the most promising *ligand* poses. In this paper, we focus only on the geometrical docking phase, used to filter out incompatible *ligands* .

Algorithm 1 shows the pseudo code of the geometrical docking. Due to the high number of degree of freedom, it is unfeasible to perform an exhaustive exploration of the possible pose of the *ligand*. For this reason, the application implements a greedy optimization heuristic with multiple restarts.

The outer loop generates *N* different initial poses for the target *ligand* , maximizing the probability to avoid local minimum. Each

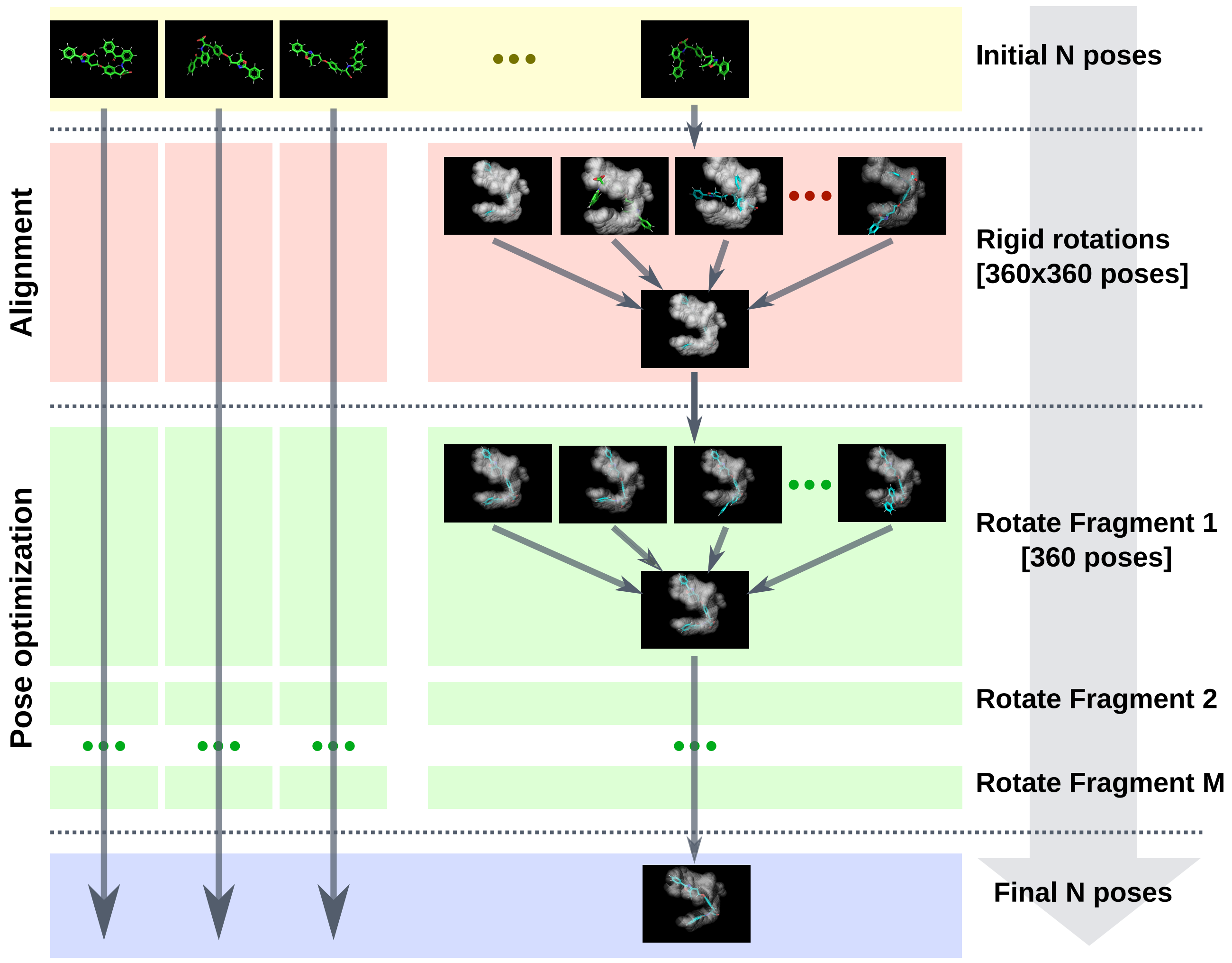

Figure 1: Algorithm work-flow that estimates the *ligand* final poses, highlighting the independent computations. In this example, we use the 1fm9 *pocket* with the related co-crystallized *ligand* .

iteration of the outer loop aims at docking the $i-th$ initial pose of the *ligand* .

Within the body of the outer loop, the docking algorithm is divided into two sections. The first one (lines 3-9) performs rigid rotations of the *ligand* , to find the best alignment with the target *pocket* , according to the scoring function. We will refer to this section of the algorithm as Rigid Rotation. In the last section of the algorithm (lines 10-14), we optimize the shape of the *ligand* by evaluating each *fragment* in an independent fashion (line 10). In particular, we rotate each *fragment* to find the angle that maximizes the scoring function without overlapping with the other atoms of the ligand (lines 11-13). We will refer to this section of the algorithm as Optimize Pose. We need to evaluate each *fragment* sequentially, since a *fragment* may include another *fragment*. Therefore, if we parallelize the pose optimization over the *fragments*, we might change the *ligand* structure in an unpredictable way, invalidating the outcome of the application.

Finally, Figure 1 depicts the geometric docking work-flow, highlighting data dependencies. In particular, the initial poses are independent, since every initial pose represents the actual starting point of the docking algorithm. For every starting pose, we perform rigid rotations to select the most suitable alignment of the initial pose of the *ligand* for the target *pocket* . After the Rigid Rotations, we proceed with the Pose Optimization phase, evaluating each fragment of the *ligand* sequentially. As output, we retrieve N poses, one for each starting pose.

3.2 Profiling

To identify bottlenecks of the application on CPU, we profiled the application using Score-P, a well-known profiling tool [9]. Figure 2 reports the result of this analysis for the most significant functions. In particular, for each function we report the percentage of time spent in that function, comprehending children, and the number of times that it is called in the algorithm. From the results, we noticed that the main bottleneck of the application is the scoring function.

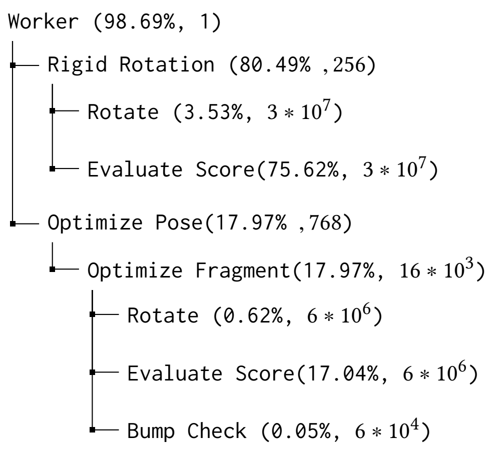
Worker (98.69%, 1)
Rigid Rotation (80.49% ,256)
Rotate (3.53%, $3 * 10^7$)
Evaluate Score(75.62%, $3 * 10^7$)
Optimize Pose(17.97% ,768)
Optimize Fragment(17.97%, $16 * 10^3$)
Rotate (0.62%, $6 * 10^6$)
Evaluate Score(17.04%, $6 * 10^6$)
Bump Check (0.05%, $6 * 10^4$)

Figure 2: The application profiling result. For each significant function, we show two information: the percentage of time spent in that function (comprehend sub-functions) and the number of calls.

Even if the function itself is rather simple, we need to call it every time we modify the *ligand* structure, to drive the pose optimization process. In particular, the scoring function evaluates "how good" is the position of every atom of the *ligand* with respect to the *pocket* . The actual score of the *ligand* is the average score of the atoms. Due to the code optimization, this function leverages the CPU vector units to process the score of each atom, leading to an execution time of less than 100*ns*. However, due to the high number of calls from the algorithm (10^7), this function becomes the bottleneck of the application. Moreover, the functions that actually rotates the *ligand* atoms or that test whether a pose is valid, have a negligible impact on the overall execution time, since they are also able to exploit the vector units of the CPU.

From the profiling analysis, the application complexity is not restricted to a single complex function, but it is due to the high number of alternative poses to evaluate for finding the best one. Moreover, since the algorithm is greedy, we need to perform multiple restarts to lower the probability of finding a local minimum. Therefore it seems to fit the parallel nature of the GPU paradigm. On the other hand, we don't have a single kernel to offload to the GPU, but we need to address the whole algorithm, or the data transfer cost would be higher than the benefit.

From the implementation point of view, we decided to use the OpenACC directive language to offload application code to the kernel. Moreover, OpenACC provides to application developers the possibility to explicitly control data transfers, minimizing the related overhead.

3.3 Implementation

From the CPU profiling of the application, we implemented a first version of the algorithm that aims at minimizing data transfer, while maintaining the application structure. We decided to introduce parallelism on the number of poses (they can all be managed in parallel and are completely independent). 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*).

Algorithm 2: Pseudo-code of the final algorithm offloaded to the GPU, where the Rigid Rotations are parallelizable.

```
Input: Target Pocket, initial pose of a ligand
Output: A set of scores, one for each pose
repeat
    Generate_Starting_Pose(Pose_id);
until Pose_id < N;
for angle_x in range(0:360) do
    for angle_y in range(0:360) do
        repeat
            Rotate_and_Score(angle_x, angle_y, Pose_id);
        until Pose_id < N;
    end
end
Reductions
repeat
    for fragment in ligand_fragments do
        for angle in range(0:360) do
            Rotate(fragment, angle, pose_id)
              Checkbump(fragment, pose_id)
              Score(fragment, pose_id)
        end
    end
until Pose_id < N;
ResultRetrievalfromGPU
```

From the implementation point of view, the following changes are required to generate the binary of the offloaded kernel, i.e. the parallel region in OpenACC jargon. All the data structures interacting with the offloaded kernel have to be compliant with the OpenACC guidelines [15] for handling data. Moreover, it is mandatory to mark each function called inside the parallel region with the OpenACC "routine" directive.

Since our plan is to parallelize the computation over the initial poses, we require a private data structure to represent the initial pose of the ligand. However, when we tried to run the application it failed: it resulted in illegal accesses to the GPU memory when trying to copy the private data structure. Since it is a class containing arrays, whose copy constructor has been redefined according to the OpenACC manual [14], it is not clear the source of the problem. Therefore, we decided to bypass the issue by replicating the initial pose and by using a manual management of the data. Even if a fair code refactoring was required, we still tried to maintain the original structure of the application.

With this modification, we fixed the illegal access issue, but the GPU application was slower than the CPU one. We analyzed the problem and noticed that this was not due to data movement since everything was resident on the GPU. We found out that we were not really exploiting the parallelism of the GPU because parallelizing the computation of all the poses was not giving enough work to the GPU. The use of the same data structure for all the Rigid Rotations of one pose was limiting 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, as shown in Algorithm 2. To achieve the desired result, we had to modify the rotation and scoring functions, unifying them to avoid to store all the temporary *ligand* poses. With this modification, 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 best score, storing only the best pose of the *ligand* for the following step.

This implementation improved the computation efficiency, moving the bottleneck from the Rigid Rotation section to the Optimize Pose function. As previously stated, since we have to process the *fragments* sequentially, it is not possible to expose more parallelism with respect to the first approach.

4 EXPERIMENTAL RESULTS

We performed the measurement on a target machine with an Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz CPU and a Nvidia Tesla K40m GPU. The operating system was CentOS 7.0, and we compiled the program using PGI 17.10. We compiled the baseline using GCC 5.4, with the avx flag to enable vectorization on top of the O3 optimization level.

We analyzed the performance of the GPU kernel using nvprof [13], in terms of execution time, occupancy and multiprocessor activity. The GPU occupancy is the number of used warps, in percentage. The multiprocessor activity is the percentage of time when the streaming multiprocessors have one or more warps issuable, i.e. not in a stalled state.

The input dataset for the experiments uses 23 different *ligand* and *pocket* pairs, taken from PDB database [3]. In particular, we used the following *pockets*: *1b9v, 1br6, 1c1b, 1ctr, 1cvu, 1cx2, 1d3h, 1ezq, 1fcx, 1fl3, 1fm6, 1fm9, 1fq5, 1gwx, 1hp0, 1hvy, 1lpz, 1mq6, 1oyt, 1pso, 1s19, 1uml* and 1ydt. For each *pocket* we docked the relative co-crystallized *ligand* . We used those molecules to have a correct estimation of the execution time of the application.

4.1 Performance evaluation on the GPU

To optimize the application performance, we tried different mapping of the computation on the GPU. 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. We investigated how these levels of parallelism are mapped on Nvidia GPUs by the PGI compiler. The only related information was found in the PGI development forum, where one of the developers mentioned that "worker is a group of vectors which conceptionally maps to a CUDA warp. Our actual implementation maps a vector to threadidx%x and worker to threadidx%y." Which means that 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. We set all the functions that change the position of the atoms at vector level. The intermediate loops are set at worker

columnwidthcolumnwidth.

Figure 3: Total execution time of the accelerated kernel, for all the 23 pockets, divided into the different functions.

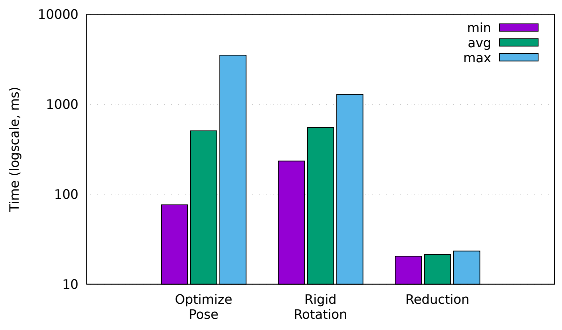

Figure 4: execution time of a single call of each function.

level. From the CUDA specification, it is known that the block maximum size is 1024 [12], that can be divided into three dimensions. However, only 2 dimensions are addressable with OpenACC. Using this information, we performed a Design Space Exploration to tune the block size, taking into account Nvidia recommended best practices. From experimental result, the best size configuration for each function is:

- Rigid Rotations: 8 workers and 128 vector length.
- Optimize Pose: 64 workers with 1 as vector length.

In particular, Figure 3 reports the total execution time of the GPU kernels. The total time is the sum of the execution time of a function across all the different datasets. We can notice that on GPU the bottleneck shifted from the Rigid Rotations to the Optimize Pose function.

Focusing on the execution time of single functions, we can notice from Figure 4 that the Optimize Pose has the greatest variance. This result is expected since this function depends on the number of fragments of each *ligand* , and on how likely they overlap with each other. We can also notice that the execution time for the Reductions is constant.

We also tried to let the compiler to select the configuration. In this case, the automatically selected configuration led to a decrease in the performance. For example, the compiler selected to organize Rigid Rotations in blocks of 128 vectors, with no workers, and in 360 gangs, with all the intermediate loop serialized. This configuration achieved a low occupancy (24%) and 4.5 times the execution time (67 seconds). The best solution we found in terms of execution time for the Optimize Pose is to avoid vector parallelism, due to control flow issues in the inner loop.

Even if the selected configuration is the best for the execution time, none of these kernels was able to obtain a full utilization of the GPU. It is possible to see the result of this experiment in Figure 5a: We were able to reach an almost full utilization only in the Reduction.The Rigid Rotations kernel was able to reach a 50% utilization. The Pose Optimization has a low utilization, due to the inherent control flow, i.e. the sequential optimization of the

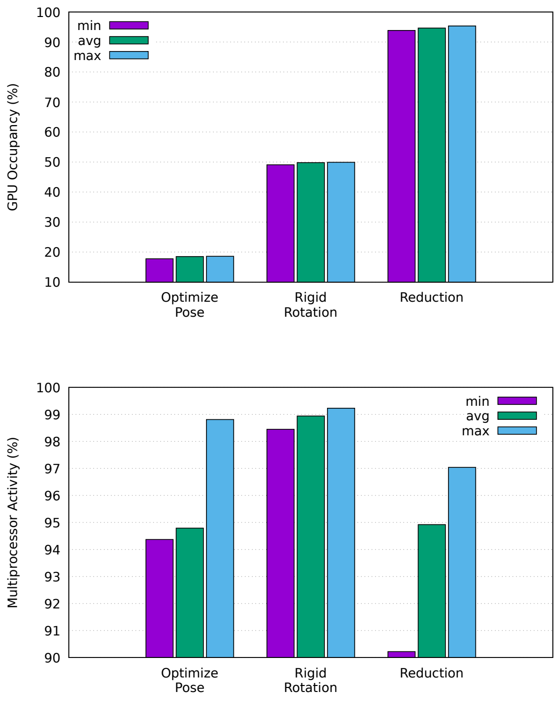

Figure 5: GPU utilization of the accelerated kernel, divided into the different functions, and multiprocessor activity in the kernel.

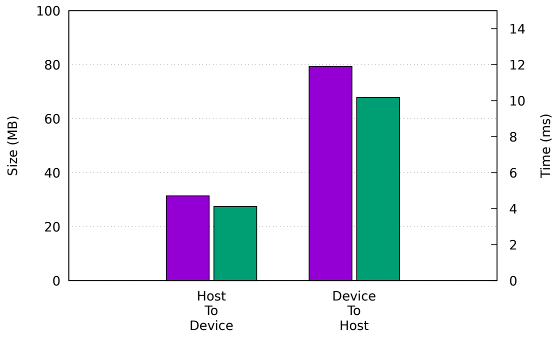

Figure 6: Data Transfer between CPU and GPU

fragments.However, as reported in Figure 5b, we can notice that all the involved processors, in all the considered functions, are heavily loaded: the lowest result is indeed 90%.

Finally, we analyze the cost of data transfer. From Figure 6 we can notice that it can be considered negligible: the total amount of data transferred, considering all the 23 different dataset execution, is less than 100MB as can be seen from the left y axis. The total elapsed time in data transfers is around 15ms, across all the executions, and it can be seen in right y axis.

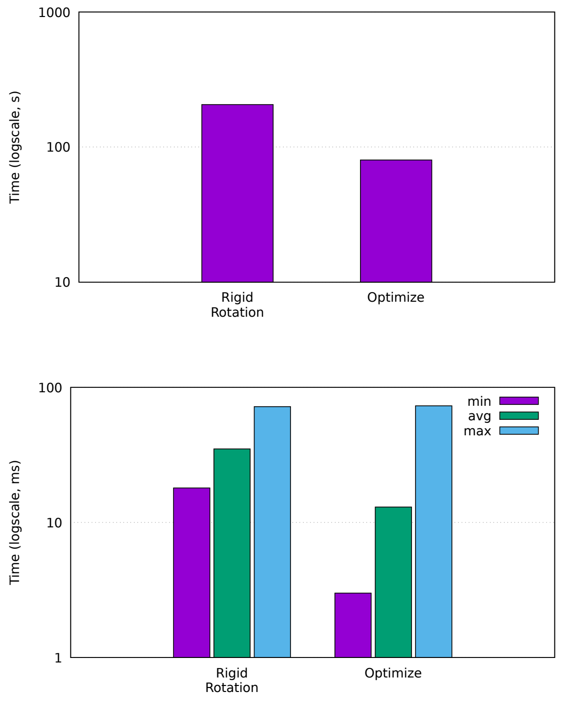

Figure 7: Execution time of the original kernel on the CPU.

4.2 Performance comparison with baseline

Figure 7 shows the execution time of the original kernel on the CPU. As previously mentioned, the most expensive function is Rigid Rotation that takes 206 seconds. We can notice that for this kernel the GPU version has a speedup of 16x (from 206s to 12s). On the other side, the Optimize Pose has only a 2x speedup (from 80s to 34s), even if in the GPU version we are performing all the initial poses in parallel. This behaviour is expected since we are able to exploit more parallelism in the Rigid Rotation function, while the sequential nature of Optimize Pose hinders the GPU performance.

If we observe the single function execution times for the CPU in Figure 7b, we can notice that even on CPU, the Optimize Pose has the largest variance.

4.3 Performance evaluation on the CPU

One of the reasons for choosing OpenACC over a CUDA implementation was to have a single source code for different architectures. Given the changes in the application that we made to optimize the performance on the GPU, this experiment aims at evaluating the performance of the new application on the CPU. From the execution time perspective, we noticed almost a 3x slowdown. To investigate the reasons behind this behaviour, we used linux perf to analyze the performance counters. The results of the experiment are reported in Table 1.

Even if the IPC is slightly lower, the cache misses are 3 orders of magnitude higher. Moreover, the number of instructions is more than doubled. As expected, the replicated initial poses of the *ligand*

Metric	Original	OpenACC Version
Execution Time	298s	831s
Number of Intructions	2,849,354,869,375	6,309,979,483,835
Cache Misses	380,672	783,114,693
IPC	3	2.4

Table 1: Comparison of the original application and the execution of the OpenACC application on the CPU: OpenACC version shows worse performance overall due mostly to data management, as the huge increase in cache misses shows.

and the inserted code to perform the reductions deteriorates the performance on the CPU. If the GPU programming paradigm requires independent data, to leverage the architecture parallelism, CPU architectures benefit from data locality. Moreover, on GPU the best practice is to perform the same operation on different data, while on CPU it is better to perform different operations on the same data.

5 CONCLUSIONS AND FUTURE WORK

In the drug discovery process, the virtual screening of a large chemical library is a crucial task. The benefits of an improvement in the time spent on evaluating the interaction from a *ligand* and the target *pocket* , are twofold. On one side it reduces the monetary 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 focus on a geometrical approach for molecular docking, optimized for the CPU architecture of an HPC platform. In particular, we leverage the OpenACC directive language, to offload the most intensive kernels on the GPU. We performed an experimental campaign to evaluate the performance of the application in terms of execution time, occupancy and multiprocessor activity.

We believe that it is possible to further improve the obtained results with a different approach. For future work, we plan to reorganize the application structure to exploit asynchronous queues to offload only the sections with heavy parallelism, while using the CPU for the other sections.

ACKNOWLEDGMENTS

This work has been partially funded by the EU H2020-FET-HPC program under the project ANTAREX - AutoTuning and Adaptivity appRoach for Energy efficient eXascale HPC systems (grant number 671623)

REFERENCES

[1] Claudia Beato, Andrea R Beccari, Carlo Cavazzoni, Simone Lorenzi, and Gabriele Costantino. 2013. 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 (2013), 1503–1517.

[2] Andrea R Beccari, Carlo Cavazzoni, Claudia Beato, and Gabriele Costantino. 2013. LiGen: a high performance workflow for chemistry driven de novo design.

[3] Helen M Berman, John Westbrook, Zukang Feng, Gary Gilliland, Talapady N Bhat, Helge Weissig, Ilya N Shindyalov, and Philip E Bourne. 2006. The protein data bank, 1999–. In *International Tables for Crystallography Volume F: Crystallography of biological macromolecules*. Springer, 675–684.

[4] Todd JA Ewing, Shingo Makino, A Geoffrey Skillman, and Irwin D Kuntz. 2001. DOCK 4.0: search strategies for automated molecular docking of flexible molecule databases. *Journal of computer-aided molecular design* 15, 5 (2001), 411–428.

[5] Rob Farber. 2016. *Parallel programming with OpenACC*. Newnes.

[6] Wu-chun Feng and Kirk Cameron. 2007. The Green500 List: Encouraging Sustainable Supercomputing. *Computer* 40, 12 (Dec. 2007), 50–55. https://doi.org/10.1109/MC.2007.445

[7] Richard A Friesner, Jay L Banks, Robert B Murphy, Thomas A Halgren, Jasna J Klicic, Daniel T Mainz, Matthew P Repasky, Eric H Knoll, Mee Shelley, Jason K Perry, et al. 2004. Glide: a new approach for rapid, accurate docking and scoring. 1. Method and assessment of docking accuracy. *Journal of medicinal chemistry* 47, 7 (2004), 1739–1749.

[8] Ajay N Jain. 2007. Surflex-Dock 2.1: robust performance from ligand energetic modeling, ring flexibility, and knowledge-based search. *Journal of computer-aided molecular design* 21, 5 (2007), 281–306.

[9] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. 2012. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir. In *Tools for High Performance Computing 2011*, Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M. Resch (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 79–91.

[10] Bernd Kramer, Matthias Rarey, and Thomas Lengauer. 1999. Evaluation of the FLEXX incremental construction algorithm for protein-ligand docking. *Proteins: Structure, Function, and Bioinformatics* 37, 2 (1999), 228–241. https://doi.org/10.1002/(SICI)1097-0134(19991101)37:2<228::AID-PROT8>3.0.CO;2-8

[11] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable Parallel Programming with CUDA. *Queue* 6, 2 (March 2008), 40–53. https://doi.org/10.1145/1365490.1365500

[12] NVIDIA. 2018. CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications.

[13] NVIDIA. 2018. Profiler User Guide. https://docs.nvidia.com/cuda/profiler-users-guide/.

[14] OpenACC-Standard.org 2017. *The OpenACC Application Programming Interface.* OpenACC-Standard.org. https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.6.final.pdf.

[15] OpenACC.org. 2015. OpenACC Programming and Best Practices Guide. https://www.openacc.org/sites/default/files/inline-files/OpenACC_Programming_Guide_0.pdf.

[16] John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *IEEE Des. Test* 12, 3 (May 2010), 66–73. https://doi.org/10.1109/MCSE.2010.69

[17] René Thomsen and Mikael H Christensen. 2006. MolDock: a new technique for high-accuracy molecular docking. *Journal of medicinal chemistry* 49, 11 (2006), 3315–3321.

[18] Oleg Trott and Arthur J Olson. 2010. AutoDock Vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading. *Journal of computational chemistry* 31, 2 (2010), 455–461.