Benchmarking the Performance of Irregular Computations in AutoDock-GPU Molecular Docking

Leonardo Solis-Vasquez\textsuperscript{a,d,1,*}, Andreas F. Tillack\textsuperscript{b,1}, Diogo Santos-Martins\textsuperscript{b}, Andreas Koch\textsuperscript{a}, Scott LeGrand\textsuperscript{c}, Stefano Forli\textsuperscript{b}

\textsuperscript{a}Embedded Systems and Applications Group, Technical University of Darmstadt, Darmstadt, Germany
\textsuperscript{b}Department of Integrative Structural and Computational Biology, The Scripps Research Institute, La Jolla, CA, United States
\textsuperscript{c}NVIDIA Corporation, Santa Clara, CA, United States
\textsuperscript{d}Hochschulstr. 10, D-64289, Darmstadt, Germany

Abstract

Irregular applications can be found in different scientific fields. In computer-aided drug design, molecular docking simulations play an important role in finding promising drug candidates. AutoDock is a software application widely used for predicting molecular interactions at close distances. It is characterized by irregular computations and long execution runtimes. In recent years, a hardware-accelerated version of AutoDock, called AutoDock-GPU, has been under active development. This work benchmarks the recent code and algorithmic enhancements incorporated into AutoDock-GPU. Particularly, we analyze the impact on execution runtime of techniques based on early termination. These enable AutoDock-GPU to explore the molecular space as necessary, while safely avoiding redundant computations. Our results indicate that it is possible to achieve average runtime reductions of 50% by using these techniques. Furthermore, a comprehensive literature review is also provided, where our work is compared to relevant approaches leveraging hardware acceleration for molecular docking.

Keywords: Variable execution performance, molecular docking, early termination, OpenCL, CUDA, AutoDock

2010 MSC: 00-01, 99-00

1. Introduction

Computational chemistry is a science domain that increasingly leverages the resources of high-performance computing (HPC) systems. Both academic computing centers \cite{1,2,3,4,5,6} and cloud providers \cite{7,8} deploy the required specialized software at-scale. Computer-aided drug design, which in turn is based on computational chemistry methods, has become an important field, as it contributes to fighting against diseases such as AIDS \cite{9}, cancer \cite{10}, and COVID-19 \cite{11}.

Molecular docking simulations are among the key methods used in computer-aided drug design for predicting molecular interactions at close distances. Specifically, they aim to predict the binding poses between a small molecule and a macromolecular target, each referred to as ligand and receptor, respectively \cite{12}. These simulations can significantly shorten the time-consuming task of identifying potential drug candidates. Subsequent wet lab experiments can then be performed in an informed fashion using an already-narrowed list of promising ligands, hence reducing the overall need for costly and slow lab experiments in drug discovery.

According to recent reports \cite{13,14}, more than 60 software tools for molecular docking have been developed in the last two decades. The tool discussed in this work, AutoDock, is one of the most widely-used open-source applications for simulating ligand-receptor docking (Fig. 1). As an example of its applicability, AutoDock is being used as a docking engine in Fight-AIDS@Home as well as in OpenPandemics: COVID-19, which are world-wide community grid projects to combat AIDS \cite{15} and COVID-19 \cite{16}, respectively.

In contrast to many more traditional scientific computing codes, AutoDock is challenging from an algorithmic perspective, as it exhibits irregular behaviors in the form of nested loops with variable upper bounds and highly divergent control flows. These are used to explore multiple ligand-receptor interactions, which are quantified by score evaluations that are typically invoked 10\textsuperscript{6} times in a single simulation run. However, AutoDock has traditionally been implemented as a single-threaded application. Thus, in its original form, it was unsuitable to exploit the embarrassing parallelism inherent in the actual docking problem using widespread computing platforms such as multi-core CPUs or GPUs. This drawback is aggravated when larger and more complex molecular structures need to be analyzed.

We have been actively developing an enhanced version of AutoDock, called AutoDock-GPU, which has been parallelized and can significantly shorten time-consuming docking simulations by employing hardware-based acceleration. AutoDock-GPU has been successfully employed in challenging prediction competitions \cite{19,20}, as well as deployed on the Summit supercomputer with the aim to contribute against the SARS-CoV-2 virus \cite{21}. Currently, the AutoDock-GPU
project maintains implementations in both OpenCL and CUDA in its public open-source code repository [22].

Our prior work in [23] is based on AutoDock-GPU v1.2 and discusses how the OpenCL implementation deals with the irregularity of the docking search problem. This earlier study analyzes the impact of the molecular complexity on runtime and the quality of results achievable using different search methods.

This current paper is based on AutoDock-GPU v1.3, an open-source project with significant contributions from multiple developers at various institutions. The following is a summary of the major milestones. The original OpenCL version (predecessor of AutoDock-GPU) was implemented by L. Solis-Vasquez, A. Koch. Gradient-based optimization was implemented in v1.1 by L. Solis-Vasquez, D. Santos-Martins. Early termination was introduced in v1.2 by A. F. Tillack. The CUDA version was ported from the OpenCL code and deterministic gradients [24] were added by S. LeGrand. The resulting code was then successively added to v1.3 by Jeff Larkin (NVIDIA) and A. F. Tillack.

Therefore, extending the prior results in [23], this current paper benchmarks the overall performance as well as the runtime impact of recent algorithmic improvements added to both the OpenCL and CUDA implementations. The algorithmic improvements in AutoDock-GPU v1.3 are based on early-termination methods, so that unproductive computations can be safely avoided. Thus, the new contributions of this paper are the following:

1. Discussion of code optimizations in v1.3, which relate to robustness, feature parity, and exploitation of hardware-specific features. In addition to evaluating such optimizations, we compare the performance of AutoDock-GPU v1.3 against that of v1.2.

2. Evaluation of algorithmic optimizations in v1.3, which feature the autostop and heuristics options to terminate AutoDock-GPU executions early.

3. A comprehensive literature review of parallelized or hardware-accelerated molecular docking, where we compare and contrast the different approaches with our own solution.

In contrast to our previous work, which also examined performance on multi-core CPUs, this work focuses on modern GPUs. Particularly, our main experiments were performed on recent NVIDIA A100 GPUs.

This manuscript contents are organized as follows. First, Section 2 provides an overview of AutoDock-GPU’s functionality. Section 3 discusses the performance and algorithmic enhancements in v1.3. The experimental setup is described in Section 4 while the corresponding results are analyzed in Section 5. A review of the current state of the art is presented in Section 6. This paper concludes in Section 7, where it summarizes the outcomes and provides some directions for future work.

2. Functionality Overview

An extensive discussion on AutoDock-GPU’s functionality is provided in our previous studies [23][25]. This section is a self-contained summary that emphasizes the factors that contribute to the irregular executions of the program.

AutoDock-GPU, as other software applications for molecular docking, systematically explores several poses of a ligand, i.e., its spatial geometrical arrangements, and aims to find the pose that binds strongly to a given region on the receptor surface. As shown in Fig. 2, AutoDock-GPU encodes such pose using the degrees of freedom (translational, orientational, torsional) experienced by the ligand during simulation. Hence, for a ligand with \( N_{\text{rot}} \) rotatable bonds, each of its poses is encoded as \( \{x, y, z, \phi, \theta, \alpha, \psi_1, \psi_2, \ldots, \psi_{N_{\text{rot}}/3}\} \), where each set element is later referred to as a gene.

The pose strength is quantified with a score, which is computed via a scoring function (SF). AutoDock-GPU uses as a scoring function a semi-empirical physics-based free-energy force field (kcal/mol), which models atomic interactions such as Van der Waals, hydrogen bonding, electrostatics, desolvation, as well as the overall entropy [26]. The score depends on the interatomic distances, which vary when a new pose is generated. The execution time of the score evaluation increases when number of ligand atoms (\( N_{\text{atom}} \)) is larger. As will be detailed shortly, scores are evaluated in the order of million times per optimization run (Section 2.1), while the mathematical derivatives of the score are used to drive the optimization more efficiently (Section 2.2).
comes first (Algorithm 1: line 3) generations (default: 250). Several methods have been incorporated to a torsion, namely the rotation of affected ligand atoms around the rotatable-bond axis.

2.1. Lamarckian Genetic Algorithm

The docking engine in AutoDock-GPU is a Lamarckian Genetic Algorithm (LGA), which performs a systematic optimization of molecular poses. By employing an LGA, AutoDock-GPU maps these pose representations into biological evolution elements, and optimizes the latter through genetic operations.

Particularly, AutoDock-GPU treats each pose as an individual of a genetic population. Each individual is represented by its genotype, which in turn is composed of a set of genes. New individuals are generated through genetic operations from their genetic ancestors. The LGA in AutoDock-GPU couples a genetic algorithm (GA) and a local search (LS). The GA performs crossover, mutation, and selection operations. The poses produced by the GA are refined by LS, which is a local minimization procedure. More details on LS methods are provided in Section 2.2. AutoDock-GPU performs independent LGA-runs (Algorithm 1 line 2), whose number by default is $R = 100$. A single LGA run terminates when a pre-defined maximum number of score evaluations (default: $N_{\text{MAX}}^{\text{score-evals}} = 2 \times 500'000$) or generations (default: $N_{\text{MAX}}^{\text{gens}} = 27'000$) is reached, whichever comes first (Algorithm 1 line 3).

### Algorithm 1: Lamarckian Genetic Algorithm (LGA)

```plaintext
1 Function AutoDock-GPU
2 /* Coarse-Level Parallelism */
3 while ($N_{\text{score-evals}} < N_{\text{MAX}}^{\text{score-evals}}$) and ($N_{\text{gens}} < N_{\text{MAX}}^{\text{gens}}$) do
4   /* Medium-Level Parallelism */
5     GA (population)
6     /* Medium-Level Parallelism */
7     for individual in random-subset (population) do
8       LS (get-genotype (individual))
```

2.2. Local Search

A local-search component refines the poses generated by the GA. Several alternative methods have been incorporated as LS and evaluated in AutoDock-GPU. Among these, and depending on the molecular complexity, two different methods produce the best scores and poses: Solis-Wets and ADADELTA. Basically, both methods generate new genotypes using an initial one as a starting point, while aiming to minimize the score with every attempt. However, these two methods differ in the way they generate genotypes.

Solis-Wets [27] generates new genotypes by adding or subtracting small random delta changes to each gene of an initial genotype. At each iteration, the change size is either increased or decreased depending on whether the number of consecutive successful (i.e., score is minimized) or failed attempts is greater than four, respectively. Solis-Wets has a runtime-defined termination (Algorithm 2 line 2), i.e., either when the number of LS iterations reaches the maximum (default: $N_{\text{LS-iters}}^{\text{MAX}} = 300$), or the change size reaches its minimum (default: step $N_{\text{step}}^{\text{MIN}} = 0.01$).

### Algorithm 2: Solis-Wets (SW) local search

```plaintext
/* Fine-Level Parallelism */
1 Function SW (genotype)
2 while ($N_{\text{LS-iters}} < N_{\text{MAX}}^{\text{LS-iters}}$) and (step $> step^{\text{MIN}}$) do
3   delta = create-delta (step)
4   // new-genotype1
5   for each gene in $N_{\text{gens}}$ do
6     new-gen1 = gene + delta
7     if SF (new-gen1) < SF (genotype) then
8       genotype = new-genotype1
9       success = success +; fail = 0
10      else
11        // new-genotype2
12        for each gene in $N_{\text{gens}}$, do
13          new-gen2 = gene - delta
14          if SF (new-gen2) < SF (genotype) then
15            genotype = new-genotype2
16            success = success +; fail = 0
17          else
18            success = success +; fail = fail +
19          step = update-step (success, fail)
```

Instead of random deltas, ADADELTA [28] generates new genotypes by using gradients calculated from the score of an initial genotype. The higher computational complexity in ADADELTA compared to Solis-Wets is due to the gradient calculation (GC) involving analytic and numerical derivatives (Algorithm 3 lines 2, 7), as well as due to the update rule using information of past gradients (Algorithm 3 line 4). An extended mathematical background and impact on pose prediction of ADADELTA is provided in our previous work [25]. This method is also characterized by a divergent execution that depends on whether the score was minimized (Algorithm 3 line 5).

Performing the local search takes more than 90% of the overall execution time. In the original single-threaded AutoDock program, only 6% of the population was sub-
3. Performance Enhancements

This section describes the overall parallelization strategy and highlights the differences between the OpenCL and CUDA variants. Moreover, it discusses the recent new features incorporated in the tool after the last publication.

3.1. Parallelization

The OpenCL implementation is based on the mapping of AutoDock-GPU computations onto OpenCL elements (Table 1). This mapping allows us to parallelize the computation in the structure visualized in Fig. 3.

<table>
<thead>
<tr>
<th>Computations</th>
<th>OpenCL elements</th>
<th>Parallelization level</th>
</tr>
</thead>
<tbody>
<tr>
<td>GA/LS generation</td>
<td>Kernel</td>
<td>Coarse</td>
</tr>
<tr>
<td>Individual</td>
<td>Work-Group</td>
<td>Medium</td>
</tr>
<tr>
<td>Scoring/Gradient</td>
<td>Work-Item</td>
<td>Fine</td>
</tr>
</tbody>
</table>

An AutoDock-GPU execution performs $R$ independent LGA runs, where runs are represented with indexes $\text{Run}_0 = \{0, 1, 2, \ldots, R-1\}$. In every LGA run, a population of $P$ individuals, with indexes $\text{Ind}_\text{ID} = \{0, 1, 2, \ldots, P-1\}$, are processed through GA and LS. Particularly, AutoDock-GPU processes simultaneously individuals from different LGA runs. Thus, $R \times P$ individuals are mapped each to an OpenCL work-group. The relation between their indexes is ruled as follows: $\text{WG}_\text{ID} = \text{Run}_0 \times P + \text{Ind}_\text{ID}$. Either GA or LS generate new individuals through their respective genetic or local methods applied on genotypes. Furthermore, GA and Solis-Wets LS involve score evaluations, while ADADELTALSA additionally computes gradients. The generation, scoring, and gradient calculations are fine-grained tasks carried out by OpenCL work-items.

The CUDA variant was developed using the OpenCL code as a starting point. This port was motivated by the interest of using AutoDock-GPU for COVID-19 research on the Summit supercomputer [21]. The computing nodes of Summit are composed of POWER9 CPUs and NVIDIA GPUs, where OpenCL is not supported. Analogously to the OpenCL case, AutoDock-GPU computations are mapped to CUDA processing elements at different granularities. Since both APIs as well as their underlying work-distribution mechanisms strongly resemble each other, the above index mapping (initially conceived for the OpenCL code) is also valid for the CUDA variant. Therefore, the initial approach of code transitioning was to replace the OpenCL processing elements (work-groups, work-items) with their respective CUDA counterparts (thread-blocks, threads).

Prior to AutoDock-GPU v1.3, a number of hardware-related optimizations were applied on top of the CUDA baseline. One of these was the enhancement of parallel reductions by explicit warp-level programming. This is based on CUDA primitives that allow a more efficient data exchange between warp threads. Using the `__syncthreads()` intrinsic, it is possible to move a value from one thread to other active threads within a warp, without accessing `__shared__` memory, but employing registers instead [29]. In order to ensure the correct execution of all parallel reductions, the size of CUDA blocks was required to be an integer multiple of 32 threads. Note that this requirement does not apply to the OpenCL variant, since OpenCL lacks low-level programming capabilities for expressing such warp- or wavefront-level optimizations.

3.2. Recent improvements

The development of AutoDock-GPU from v1.2 to v1.3 has significantly improved the robustness, feature parity between the OpenCL and CUDA variants, and the use of hardware-specific optimization. Regarding robustness, to avoid code divergence of the OpenCL and CUDA versions, and issues such as passing different parameters to the OpenCL and CUDA vari-
ant of a given kernel, the host code of both variants has been carefully unified.

As a good practice for code maintenance, improvements found in one variant of the tool are ported to the other one for feature parity (if appropriate). In particular, the OpenCL code in AutoDock-GPU v1.2 included an extra set of Solis-Wets hyper-parameters, which were introduced as additional variables (dependent of \( N_{\text{atom}} \) and \( N_{\text{rot}} \)) to control the genotype deviation at every Solis-Wets iteration (Algorithm 2). This feature was ported to the CUDA variant during the development of AutoDock-GPU v1.3.

With regard to hardware-specific optimizations, a number of changes have been incorporated into the CUDA variant. The first one is the dynamic allocation of \_shared\_ memory. This contrasts with the static allocation used in the OpenCL \_local\_ memory counterpart, where the allocation size is known at compile time. The second change is the addition of the \_launch_bounds\_ qualifier to the kernel implementations. According to [29], a kernel using fewer registers may, in turn, increase the number of threads and thread-blocks residing on a CUDA streaming multiprocessor (for more details see Section 5). The compiler uses heuristics to minimize the register usage, and a developer can provide hints for the heuristics using the above qualifier. Since the optimal values for the parameters of this qualifier differ across architectures, Listing 1 shows how the CUDA\_ARCH macro is used to specify them in a portable manner. The required parameters are two, namely, the maximum number of threads per block (NTHREADS\_BLOCK), and the desired minimum number of blocks per streaming multiprocessor (NBLOCKS\_A and NBLOCKS\_B).

Complementarily, the new heuristics option is based on an adaptive termination criterion that also prevents AutoDock-GPU from running unreasonably long executions. For this purpose, heuristics utilizes instead an alternative value of \( N_{\text{score-evals}} \) (Algorithm 1, line 3). Such alternative value depends on two terms. The first one is \( \text{heur}_{\text{evals}} \) (Equation 1), which depends on the number of rotatable bonds (\( N_{\text{rot}} \)) as well as the set of constants (\( a \) and \( b \)) that vary according to the selected local-search method. The second term is \( \frac{\text{HEURIS}}{\text{MAX}} \), which is the maximum number of score evaluations under heuristics (default: 50'000'000). Equation 2 shows how the alternative value of \( N_{\text{score-evals}} \) is calculated. Furthermore, the (capped) number of evaluations suggested by the heuristics option can be finished sooner when autostop (if also enabled) detects early convergence.

\[
\text{heur}_{\text{evals}} = \text{ceil}(1000 \times 2^{a \times N_{\text{rot}} + b})
\]  

\[
\frac{\text{HEURIS}}{\text{MAX}} \times \text{heur}_{\text{evals}} \times \frac{N_{\text{MAX}} - \text{HEURIS}}{N_{\text{score-evals}}} = \text{heur}_{\text{evals}} + \frac{N_{\text{score-evals}}}{\text{heur}_{\text{evals}}}
\]  

4. Methodology

In all our experiments, we used AutoDock-GPU v1.3, unless otherwise indicated. The program execution and runtime measurements were fully automated using bash and Python scripts. The dataset used has been publicly released and is properly documented. Details of the archive repositories hosting the sources and data are given in Section 8. Finally, the performance evaluation was carried out on compute systems featuring recent GPUs in both consumer and professional versions.

From the many different protocols possible for validating docking [30], our experiments consist of re-docking. In this approach, already-studied ligand-receptor inputs are docked again, so that resulting ligand poses can be compared to well-known reference solutions.

4.1. Program configuration

AutoDock-GPU executions perform 100 LGA runs over a population of 150 individuals. The maximum number of score evaluations per LGA run was set to 2'500'000. The maximum number of generations (per LGA run) was set to 99'999, which is larger than the default value of 27'000. The purpose of this choice is to ensure the program termination happens only when the number of score evaluations reaches the aforementioned upper bound. In all cases, the entire population is subjected to local search (lsrate = 100\%). Other parameters were left as default [31]. Table 2 lists program parameters and their configurations.

For evaluating the efficiency of the early-termination options, the corresponding defaults are used. Namely, when using autostop, the program was configured to automatically stop after reaching a deviation of \( \text{stopstd} = 0.15 \text{ kcal/mol} \) compared to the best score achieved five generations before (asfreq = 5, unless specified otherwise). Moreover, the default number of score evaluations under heuristics is 50'000'000.
4.2. Dataset

Similarly as in our previous experiments in [23], a set of 20 ligand-receptor inputs was selected from well-established sets for assessing molecular docking methodologies. Our dataset is composed of eleven entries from Astex [32] (IDs: 1u4d, 1xo2, 1yv3, 1oe1, 1oyt, 1yr, 1t46, 2bm2, 1mzc, 1r55, 1kzk), four from CASF-2013 [33] (IDs: 3a8o, 1hfs, 1jyq, 2d1o), and five from the Protein Data Bank (PDB) [34] (IDs: 5elo, 5kao, 3drf, 4er4, 3er5). Table 3 indicates the number of rotatable bonds and atoms for each input case. This dataset covers up to 31 rotatable bonds, which is a large range considering that the optimal size for a WG would be of 64 work-items.

4.3. Evaluation platforms

Table 4 lists the main technical specifications of the GPU cards used in our evaluation. Such devices feature recent architectures, as well as provide a varied range of compute capabilities that theoretically achieve from ∼9.1 TFLOPS and 448 GB/s on the RTX 2070 SUPER, up to ∼19.5 TFLOPS and 1'555 GB/s on the A100.

For a fair comparison, we disregard the different host platforms holding the various GPUs. Specifically, we include only the GPU-side kernel configuration and execution, plus all required host-GPU data movements in our measurements. Such time components are collectively reported as docking runtime. Host-side operations, such as file I/O and results processing, were not included and are considered as idle time (from the GPU perspective).

5. Results and Discussion

We begin the evaluation by determining suitable configuration choices. Then, we compare the runtimes achieved by using our current and prior baseline work. Finally, we show the impact of the new autostop and heuristics options.

5.1. Runtime-based performance

At this point, it is important to note that an OpenCL compute unit (CU) is a hardware block that processes a single OpenCL work-group (WG) at a time. Basically, the more CUs are available, the more WGs can be processed in parallel. A CUDA streaming multiprocessor (SM) corresponds to an OpenCL CU [35], thus analogously, the more SMs are available, the more thread blocks (TB) can be processed simultaneously. Table 4 indicates that for all chosen GPU cards, the ratio between the number of cores and OpenCL CUs is 64, suggesting that the optimal size for a WG would be of 64 work-items.

Fig. 4 shows the docking runtimes using three input cases: 1u4d, 2bm2, and 3er5. In terms of workload amount, 1u4d and 3er5 are the corner cases. From an algorithmic perspective, 2bm2 represents a threshold case, because for inputs with \(N_{\text{rot}} > 7\), ADADELTA starts becoming more effective than Solis-Wets at predicting molecular poses [25]. Considering these three input cases as well as both Solis-Wets and ADADELTA methods, it can be observed that OpenCL runtimes (Fig. 4 left) tend to be lower when using \(W_{G} \text{size}\) of either 64 or 128 work-items. Although there are some few exceptions, this is a general tendency observed using our dataset, and goes in line with the aforementioned ratio of number of cores and CUs. In the case of CUDA runtimes (Fig. 4 right), minimum values are achieved mostly for \(T_{B} \text{size}\) of 32 threads. An exception to this in the CUDA version happens when docking 3er5 using Solis-Wets. In this case, lower runtimes are achieved by using blocks of 64 threads on all GPU cards. Based on these results, there is no single \(W_{G} \text{size}\) or \(T_{B} \text{size}\) configuration that works best for all cases. Hence, a future optimization would be to enable AutoDock-GPU to automatically choose sizes that are likely to result in faster executions (see Section 7).

Nevertheless, similar to our previous work [23], we think 64 work-items or threads is a reasonable choice for \(W_{G} \text{size}\) or
Thus, we employed this configuration to compare the performance between all GPU cards. For that purpose, we consider the geometric mean of runtime values corresponding to the entire dataset (Fig. 5). Despite that the OpenCL and CUDA runtimes seem similar at first glance, slight differences can be found. For instance, when running Solis-Wets on the RTX2070, OpenCL runtimes (12.8 s) are in average a bit lower than those of CUDA (13.2 s). Conversely, for ADADELTA on the RTX2070, CUDA runtimes (25.1 s) are lower than the respective OpenCL average (27.1 s). Considering only raw compute capabilities (Table 4, FP32 performance), ADADELTA sees a significant speedup of more than 3.5× for the OpenCL codes from the v1.2 to the v1.3 code on the A100 GPU. Interestingly, the v1.3 CUDA code is executed a bit slower on the A100 (7.0 s) than either the v1.3 or v1.2 OpenCL codes.

Similar to Solis-Wets, the CUDA implementation of the v1.3 ADADELTA algorithm remains a bit slower than the OpenCL code. Since both OpenCL and CUDA versions in AutoDock-GPU v1.3 produce virtually identical computations, we believe such performance advantage of OpenCL over CUDA might be caused by several factors. One of these is the implemented on-device memory allocation, which for the OpenCL version is performed statically (in contrast to the dynamic allocation in the CUDA version), and thus, possibly enabling the compiler to perform more aggressive optimizations. We will investigate this, and update the code correspondingly in future releases.

From previous experiments, it is clear that faster executions (i.e., lower runtimes) are achieved on the A100 GPU. Using this device, we compare the performance of AutoDock-GPU v1.2 (used in our previous work [23]) and AutoDock-GPU v1.3 (our work here). Since [23] reported only total execution runtimes, we will also examine AutoDock-GPU v1.3 in the same way. It is important to note that this is different from the other measurements in this paper, which report only the docking runtimes, i.e., the GPU-side and data movement times.

Fig. 6 indicates that for Solis-Wets, the average total runtime (GPU and host) of the OpenCL code has been reduced from 6.5 s (v1.2) down to 5.2 s (v1.3) on the A100 GPU. Interestingly, the v1.3 CUDA code is executed a bit slower on the A100 (7.0 s) than either the v1.3 or v1.2 OpenCL codes. ADADELTA sees a significant speedup of more than 3.5× for the OpenCL codes from the v1.2 to the v1.3 code on the A100. Similar to Solis-Wets, the CUDA implementation of the v1.3 ADADELTA algorithm remains a bit slower than the OpenCL code. Since both OpenCL and CUDA versions in AutoDock-GPU v1.3 perform virtually identical computations, we believe such performance advantage of OpenCL over CUDA might be caused by several factors. One of these is the implemented on-device memory allocation, which for the OpenCL version is performed statically (in contrast to the dynamic allocation in the CUDA version), and thus, possibly enabling the compiler to perform more aggressive optimizations. We will investigate this, and update the code correspondingly in future releases.

5.2. Autostop and heuristics

For these experiments, we continue using WGsize/TBsize of 64 work-items/threads on the A100. For testing the autostop option, executions were configured with different as-
Table 5: Docking runtimes (s), scores (kcal/mol), and RMSDs (Å) achieved using the OpenCL version, with and without the **autostop** option, on the A100 GPU. The best values within each case are colored.

<table>
<thead>
<tr>
<th>LS</th>
<th>ID</th>
<th>Metric</th>
<th>No autostop</th>
<th><strong>autostop</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Solis-Wets</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1u4d</td>
<td>RunTime</td>
<td>-7.27 kcal/mol</td>
<td>1.36 s</td>
<td>0.48 s</td>
</tr>
<tr>
<td></td>
<td>RMSD</td>
<td>1.36 Å</td>
<td>-7.26 kcal/mol</td>
<td>-10.54 kcal/mol</td>
</tr>
<tr>
<td>2bm2</td>
<td>RunTime</td>
<td>-10.99 kcal/mol</td>
<td>2.92 s</td>
<td>2.90 s</td>
</tr>
<tr>
<td></td>
<td>RMSD</td>
<td>2.01 Å</td>
<td>-10.54 kcal/mol</td>
<td>5.30 Å</td>
</tr>
<tr>
<td>3er5</td>
<td>RunTime</td>
<td>-8.92 kcal/mol</td>
<td>21.61 s</td>
<td>21.58 s</td>
</tr>
<tr>
<td></td>
<td>RMSD</td>
<td>4.92 Å</td>
<td>-9.56 kcal/mol</td>
<td>3.78 Å</td>
</tr>
<tr>
<td><strong>ADADELTA</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1u4d</td>
<td>RunTime</td>
<td>-7.27 kcal/mol</td>
<td>1.69 s</td>
<td>0.52 s</td>
</tr>
<tr>
<td></td>
<td>RMSD</td>
<td>1.36 Å</td>
<td>-7.27 kcal/mol</td>
<td>1.36 Å</td>
</tr>
<tr>
<td>2bm2</td>
<td>RunTime</td>
<td>-10.59 kcal/mol</td>
<td>4.89 s</td>
<td>2.23 s</td>
</tr>
<tr>
<td></td>
<td>RMSD</td>
<td>5.31 Å</td>
<td>-10.59 kcal/mol</td>
<td>5.30 Å</td>
</tr>
<tr>
<td>3er5</td>
<td>RunTime</td>
<td>-14.74 kcal/mol</td>
<td>52.15 s</td>
<td>52.25 s</td>
</tr>
<tr>
<td></td>
<td>RMSD</td>
<td>4.57 Å</td>
<td>-13.93 kcal/mol</td>
<td>5.04 Å</td>
</tr>
</tbody>
</table>

The runtime was reduced from 1.69 s down to 0.52 s, with no penalties in the score or in RMSD. In the 2bm2 case, for Solis-Wets, there is an score improvement (from -10.09 kcal/mol to -10.54 kcal/mol) along with a significant RMSD degradation (from 2.01 Å to 5.32 Å). The minor benefits in runtime may indicate that the docking search was trapped in a local minimum during this execution. For ADADELTA processing the 2bm2 input, by using **autostop**, we required a shorter runtime (2.23 s instead of 4.89 s) to achieve the same score value (-10.59 kcal/mol) and a slightly better RMSD (5.30 Å instead of 5.31 Å). In case of large molecules, **autostop** may provide few (e.g., 3er5 for Solis-Wets), or even no advantages (e.g., 3er5 for ADADELTA). As specified in Section 3.2 the stop criterion in **autostop** is based on the score improvement rather than the runtime of its non-**autostop** counterpart. Therefore, for cases involving a challenging docking search (e.g., 3er5, Nrot = 31), it is possible that AutoDock-GPU improves the score slowly as it progresses over generations, while having the time overhead due to the additional score checking required for the **autostop** functionality.

In addition, Fig. 8 shows the impact on runtime of using the **heuristics** options as well as that of the combination of **autostop + heuristics**. Despite not being as effective as **autostop**, the **heuristics** option still provides performance improvements over the aforementioned baseline. Furthermore, the combination of both options leads to average runtime reductions of 53% (OpenCL) and 55% (CUDA) for Solis-Wets, and 73% (OpenCL) and 76% (CUDA) for ADADELTA.

5.3. Performance comparison between GPUs and CPUs

Up until this point, the impact of the **autostop** and **heuristics** options has been evaluated only on the A100 GPU. Here, to extend our evaluation, we report the achieved performance on a CPU-based platform and compare it against that on the A100 GPU. For these experiments, we have chosen an AWS c5.24xlarge instance [36] based on an Intel Xeon Platinum 8275
ADADELTA, resulted in a significantly smaller (i.e., better) RMSD on the A100 than on the c5.24xlarge (1.8 Å vs. 5.3 Å). For this particular case, we believe the cause was not the employed platform, but instead the heuristic nature of AutoDockGPU. Basically, in every program execution, the search starts from a random point in the molecular space, and thus every execution explores a different path through that space. It could be the case that the above execution on the c5.24xlarge was trapped in a local minimum, causing that any score improvement (driving the search) led to no corresponding RMSD improvement.

6. Related Work

This section discusses relevant studies following a general-to-specific manner. Thus, we start with a survey of parallelized molecular docking programs. Then, we compare the latest and forked developments of AutoDock-GPU.

6.1. Parallelization of molecular docking

Several efforts on performance optimization of molecular docking leverage hardware-based acceleration. Table 7 lists relevant studies from nearly the last two decades. The brief survey presented here aims to provide a reasonable understanding of the state of the art, and it is based on the more extensive discussions in [37, 38, 39], as well as our own recent literature review. Our scope is on single compute nodes, and hence, approaches targeting systems that range between clusters, grid, and cloud computing are not included. Studies listed in Table 7 can be grouped into the following categories: FFT/correlation, nature inspired, intrinsically parallel, and pairwise potentials.

The first category in our list includes programs based on either Fast Fourier Transform (FFT) or correlation. The ZDOCK program employs FFT to optimize force-field scoring functions. Van Court et al. [40, 41] proposed an FPGA-based approach...
where a correlation is implemented instead of the original FFT-based search. The core of the correlation architecture is a three-dimensional systolic array, which enables a long pipeline of computations as well as low-precision arithmetic. Such benefits are suitable for FPGAs, in contrast to the floating-point operations needed in the original ZDOCK FFT. With regard to PIPER, Sukhwani et al. extended the systolic-array architecture used for ZDOCK on FPGAs (described above) in order to support large molecules, i.e., receptor-receptor docking. The same authors developed a GPU version of PIPER [44], in which the FFT computations were performed directly rather than through correlation as for the FPGA counterparts. Furthermore, Ritchie et al. [45] accelerated the FFT-based interactions in Hex using the CUDA CUFFT library to implement one- and three-dimensional FFT computations.

Nature-inspired programs use search methods based on evolutionary or swarm-intelligence algorithms. MolDock employs a scoring function that is very similar to that of AutoDock. However, its search is based on Differential Evolution (DE), which uses weighted difference of parent individuals for the genetic selection. Simonsen et al. paralleled MolDock with a CUDA-based multi-level approach similar to that of AutoDock-GPU, while their OpenMP version simply distributes the multiple DE runs over CPU cores. On the other hand, PLANTS combines a global and a local search method, namely Ant Colony Optimization (ACO) and the Nelder and Mead algorithm (NMS), respectively. The parallelization of PLANTS proposed by Korb et al. offloads the generation phase and score calculation to a GPU, while the overall ACO+NMS algorithm runs on a CPU. The code was developed in OpenGL and NVIDIA Cg, both of which are intended just for graphics computations and are far less flexible compared to the general-purpose programming OpenCL or CUDA frameworks. Regarding the Bude program, McIntosh-Smith et al. provided an OpenCL implementation in which each work-item processes four molecular poses. To achieve higher performance, the authors optimized the use of memory-access coalescing, and reduced the negative impact of thread divergence.

As already discussed in Section 2.1, the core of AutoDock is the LGA, and thus, it falls into the nature-inspired category described above. Here, we describe relevant studies addressing LGA acceleration. Kannan et al. developed a CUDA version that excludes the Solis-Wets method from the LGA. The purpose of this exclusion was to avoid the low GPU utilization caused by the local search processing only a subset of the population (Section 2.1). Pechan et al. provided versions for GPUs and FPGAs, written in CUDA and Verilog, respectively. Both efforts by Pechan et al. served as an inspiration for the predecessor program of AutoDock-GPU, developed by Solis-Vasquez et al. In these latter studies, OpenCL was the main development language for both GPUs and FPGAs. While code portability was achieved with virtually no problems, performance portability proved to be more challenging, in the end requiring substantial platform-specific tuning of the code base. Furthermore, Mendonça et al. proposed a hybrid parallelization utilizing OpenMP and CUDA, which also excluded the Solis-Wets method.

Intrinsically-parallel programs were designed considering their inherent parallelism right from the beginning. Examples are AutoDock Vina [55] and AutoDockFR [56], both belonging to the AutoDock suite and leveraging the multiple cores available on a CPU. Regarding Vina, its scoring function is empirical rather than the potentially too-strict models based on force fields used in AutoDock. Multi-threading in Vina is achieved using the C++ Boost::Thread library. On the other hand, AutoDockFR models the flexibility of the receptor molecule. Such flexibility results in the growth of search space, which AutoDockFR deals with by employing a slightly different GA than AutoDock. AutoDockFR is implemented in Python, and distributes each of its GA runs on a single CPU core. For higher speedups, the scoring function of AutoDockFR has been ported to C++.

The pairwise-potentials category lists studies that do not focus on complete front-to-back programs, but instead just on certain score terms based on pairwise interactions, which could be integrated into a more complete scoring function. Roh et al. accelerated a scoring function composed of two terms: dispersion and electrostatics. The authors used a separate GPU for each of these terms, which would be impractical for real applications. Guerrero et al. focused on the electrostatics interactions between a receptor and a ligand. In their CUDA implementation, each thread computes the interaction between its corresponding receptor atom and all ligand atoms. Moreover, the recent studies of Saadi et al. [59, 60] accelerated the desolvation term in the scoring function of AutoDock. In their work, the authors aimed for blind docking [61] and provided both CUDA and OpenMP implementations to target GPUs and CPUs, respectively.

6.2. AutoDock-GPU

Legrand et al. introduced the CUDA variant of AutoDock-GPU and benchmarked it against an earlier version of the OpenCL code prior to AutoDock-GPU v1.2 [21]. This work was intended for virtual screening in COVID-19 research, and thus, for performing many docking runs. For this purpose, the program required the capability of streamlining consecutive docking jobs involving, e.g., multiple ligands with a single receptor. Initially, AutoDock-GPU was only capable of running a single docking job per program execution. However, the code modifications by Legrand et al. enabled the user-specification of multiple docking jobs, and their serial launch from a single program execution. Additional enhancements leverage task pipelining, i.e., overlapping the execution of the following tasks using OpenMP threading: docking launch of a current ligand-receptor system (on the GPU), the file read of next ligand coordinates (on the CPU host), and the file write of the prior resulting ligand poses (on the CPU host). Most improvements by Legrand et al. are aimed for virtual screening, while our work here focuses on accelerating single executions of AutoDock-GPU.

3 Blind docking refers to the exploration over an unknown, typically large, surface of the receptor-ligand interaction.
Motivated by the transition of computing facilities towards exascale systems, Thavappiragasam et al. developed a miniapp called miniAutoDock-GPU [61]. This has been directly derived from the CUDA variant of AutoDock-GPU, and its purpose is to evaluate the performance and portability on different computer architectures. Both AutoDock-GPU and miniAutoDock-GPU execute LGA runs. However, while AutoDock-GPU processes user-specified ligand-receptor inputs, the miniapp uses pre-loaded ones. This design choice avoids the inclusion of I/O when measuring the execution time for the miniapp, which focuses only on computation time. miniAutoDock-GPU has been implemented in CUDA and Kokkos. Its evaluation was carried on a V100 GPU, where the CUDA variant outperforms the Kokkos one by a factor of 1.8x for large- and medium-size ligands. In addition, a port to HIP was reported to be in progress, in which porting low-level and architecture-specific CUDA optimizations (e.g., warp-level reduction, each warp with 32 threads) to a different architecture using HIP (e.g., wavefront-level reduction, each wavefront with 64 threads) pose significant challenges. Thavappiragasam et al. ported only the Solis-Wets local search, and not ADADELTA. Our own efforts continue to target OpenCL and CUDA, but always consider the full program, and not just a stripped-down miniapp.

Our recently published study in [25] focuses mainly on evaluating the benefits of AutoDock-GPU v1.2 from an application domain perspective. Specifically, Santos-Martins et al. introduced the $E_{50}$ metric to quantify the number of score evaluations required to achieve a 50% of success, where success means finding the global optimum of either the score or RMSD. Resulting $E_{50}$ values on a large set of 140 ligand-receptor inputs indicate that according to the score criterion, ADADELTA executions require only ~1/23 of the evaluations than those required when using Solis-Wets for inputs with $N_{	ext{rot}} = 20$.

Furthermore, motivated by the increasing importance of energy efficiency in HPC systems, in [62] we measured the electrical power draws (W) on V100 GPUs due to AutoDock-GPU v1.0 executions. Energy efficiencies achieved on a V100 GPU were improved by ~67x and ~37x compared to those on a E5-2666 18-core CPU, when running equivalent Solis-Wets and ADADELTA computations, respectively.

Besides multi-core CPUs and GPUs, docking acceleration has been explored on FPGAs as well. In general, the fact that fewer studies target FPGAs is attributed to the larger development effort compared to GPUs. Traditional development for FPGAs requires reasoning in terms of low-level transfers between hardware registers (RTL) and synchronous logic design. In recent years, this entry barrier for programmers has been lowered by development tools from FPGA vendors (e.g., Xilinx Vitis [63]) and cross-industry standards (e.g., oneAPI [64]), in which the application can be written in OpenCL or SYCL, rather than the traditional VHDL or Verilog RTL hardware description languages. In this context, our previous work [23] summarized our last attempts to improve the performance of an OpenCL implementation of AutoDock, specifically tailored for FPGAs. While our FPGA implementations are faster than executing software on a CPU, they are far slower than using GPUs. Thus, FPGAs will realistically not be deployed to solve large docking problems. However, as described in Section 7 there still exist optimization opportunities which could potentially speed-up FPGA-based docking accelerators further.

### 7. Conclusions and Future Work

In this paper, we described the code and algorithmic improvements introduced in AutoDock-GPU v1.3, and evaluated them against our previous work based on AutoDock-GPU v1.2. Besides showing that v1.3 maintains (and sometimes exceeds) the average performance with respect to v1.2, we showed significant benefits by utilizing the new autostop and heuristics options introduced in v1.3. Concretely, when both options
are combined. AutoDock-GPU achieves average runtime reductions of 53% (Solis-Wets) and 73% (ADADELTA) on a NVIDIA A100 GPU.

Our literature review indicates that a variety of hardware devices are being used for accelerating different docking scenarios. Of these studies, the majority target GPUs and utilize CUDA as the main programming model. Recent ports of docking programs to OpenCL and HIP suggest a growing interest in alternatives to the proprietary NVIDIA CUDA. Additionally, hybrid approaches combining OpenMP and OpenCL/CUDA for heterogeneous CPU+GPU systems, are becoming more common.

As future work, besides further optimizations, we plan to exploit recent platforms and tools. Regarding GPUs, we will equip AutoDock-GPU with the capability of automatically choosing an appropriate work-group or thread-block size for higher performance, instead of the current manual selection. Moreover, we will perform further tests on new generation GPUs such as AMD’s MI100 device. On the FPGA side, we will explore variants of the AutoDock code that attempt to reduce the irregularity of execution, which in turn might allow more efficient FPGA execution. For the actual implementation, recently improved FPGA design tools such as Xilinx Vitis [63] and Intel oneAPI [64] could be leveraged.

8. Appendices

Source code, input data, and auxiliary material used for our experiments is open source and available in the links indicated below.

- **AutoDock-GPU**: [https://github.com/ccsb-scripps/AutoDock-GPU](https://github.com/ccsb-scripps/AutoDock-GPU)
- **Input data**: [https://gitlab.com/L30nardoSV/ad-gpu-miniset_20.git](https://gitlab.com/L30nardoSV/ad-gpu-miniset_20.git)
- **Scripts to reproduce experiments**: [https://github.com/L30nardoSV/reproduce-parcosi-moleculardocking](https://github.com/L30nardoSV/reproduce-parcosi-moleculardocking)

9. Acknowledgements

This work was supported by the National Institutes of Health GM069832 (to S. Forli).

We want to thank Jeff Larkin and Aaron Scheinberg for their code contributions during the CUDA porting as well as ORNL and NVIDIA for their impetus and support of the porting effort.

Calculations on the V100 GPU for this research were conducted on the Lichtenberg high performance computer of TU Darmstadt.

References

[1] Leibniz Supercomputing Centre, Scientific Application Packages. URL [https://doku.lrz.de/display/PUBLIC/ScientificApplicationPackages](https://doku.lrz.de/display/PUBLIC/ScientificApplicationPackages)


[6] Universität Paderborn: Paderborn Center for Parallel Computing (PC2), Software URL [https://wikis.uni-paderborn.de/pc2/doc/Software#Software_Availability](https://wikis.uni-paderborn.de/pc2/doc/Software#Software_Availability)


[14] FightAIDS@Home URL [https://www.worldcommunitygrid.org/research/faah/overview.do](https://www.worldcommunitygrid.org/research/faah/overview.do)


