Parallelizing Irregular Computations for Molecular Docking

Leonardo Solis-Vasquez*, Diogo Santos-Martins†, Andreas F. Tillack†,
Andreas Koch*, Jérôme Eberhardt†, Stefano Forli†
*Embedded Systems and Applications Group
Technical University of Darmstadt, Darmstadt, Germany
Email: {solis, koch}@esa.tu-darmstadt.de
†Center for Computational Structural Biology
The Scripps Research Institute, La Jolla, CA, United States
Email: {diogom, atillack, jerome, forli}@scripps.edu

***Abstract*—AUTODOCK is a molecular docking software widely used in computational drug design. Its time-consuming executions have motivated the development of AUTODOCK-GPU, an OpenCL-accelerated version that can run on GPUs and CPUs. This work discusses the development of AUTODOCK-GPU from a programming perspective, detailing how our design addresses the irregularity of AUTODOCK while pushing towards higher performance. Details on required data transformations, re-structuring of complex functionality, as well as the performance impact of different configurations are also discussed. While AUTODOCK-GPU reaches speedup factors of 341x on a Titan V GPU and 51x on a 48-core Xeon Platinum 8175M CPU, experiments show that performance gains are highly dependent on the molecular complexity under analysis. Finally, we summarize our preliminary experiences when porting AUTODOCK onto FPGAs.**

***Index Terms*—Variable execution performance, divergent control structures, OpenCL, molecular docking, AutoDock**

I. INTRODUCTION

Molecular docking is a computational method widely used in drug discovery. It aims to predict the interaction between a small molecule (ligand) and a macromolecular target (receptor) [1]. A receptor can model a protein or nucleic acid. Ligands that inhibit the harmful effects of a given receptor are considered good drug candidates, and thus, selected for subsequent wet lab experiments. Typically, libraries containing thousands of ligands are analyzed. This procedure called *virtual screening* uses molecular docking as the computational engine, and aims to identify ligands that are suitable candidates.

One of the most widely-used molecular docking tools is AUTODOCK [2], whose runtimes can reach several hours/days when utilized for virtual screening. Despite the fact that AUTODOCK is undoubtedly useful (e.g., FightAIDS@Home [3]), one of its major drawbacks over the years have been its long execution runtimes. This is mainly due to its inability to leverage its embarrassing internal parallelism, even on widespread-available platforms such as multi-core CPUs. In recent years, we have been developing an accelerated version that, when executed on a many-core GPU, is at least 50x faster than the original AUTODOCK. This version called AUTODOCK-GPU has been initially implemented in OpenCL [4], and very recently, has been successfully ported to CUDA in order to run on the Summit supercomputer with the aim to contribute against the SARS-CoV-2 virus [5].

AUTODOCK employs a Lamarckian Genetic Algorithm to predict energetically-strong poses of ligand-receptor systems. In algorithmic terms, AUTODOCK is characterized by nested loops with variable upper bounds and divergent control performing a molecular search, as well as by time-intensive score evaluations typically invoked 10^6 times within these iterations.

Our previous work on AUTODOCK-GPU [6] evaluated overall compute performance and prediction quality. In this work, however, we present our experiences when parallelizing AUTODOCK using OpenCL, by providing a development rather than a domain-oriented perspective, like we did in prior work. Specifically, in this paper, we discuss the OpenCL development and challenges of dealing with the intrinsic AUTODOCK irregularity while seeking higher performance. Additionally, we analyze the impact on runtime from relevant factors, which comprise the chosen OpenCL work-group size, the molecular complexity of different ligand inputs, as well as the employed local-search methods. Evaluations were performed both on high-end GPUs and CPUs. Finally, we extend our discussion to sharing technical details when porting onto other accelerators, specifically FPGAs. This provides additional insights on how the application irregularity can be tackled differently based on the target architecture.

II. AUTODOCK MOLECULAR DOCKING

Molecular docking is an optimization problem where different poses of the ligand, i.e., its spatial geometrical arrangements, are systematically explored in order to find those that bind strongly to a given region on the receptor. AUTODOCK encodes such poses using a set of variables, whose size depends on the structure of molecules under analysis. Each pose is quantified with a score, which is calculated by a scoring function. A single execution of AUTODOCK (Algorithm 1), i.e., a *docking job*, consists of the iterative execution of independent runs, where each run performs an hybridized Lamarckian Genetic Algorithm.

A. Encoding

Ligand poses are represented using a combination of variables that describe: *first*, overall motion as a rigid body; and *second*, internal body flexibility. As a rigid body, the ligand can experience two types of motions: translation and rotation. Translation can be encoded with variables describing displacement in x, y, and z directions. Rotation as a rigid body can be described with ϕ, θ, and α axis-angle coordinates. The internal body flexibility models the rotation allowed for specific atomic bonds, which results in rotating ligand fragments around bond axes. If the ligand is configured with N_{rot} rotatable bonds, each of these bonds can be represented with a ψ variable. The full set of degrees of freedom of the ligand pose (also called *genes*, $N_{\text{genes}} = N_{\text{rot}} + 6$) constitutes the pose encoding $\Omega = \{x, y, z, \phi, \theta, \alpha, \psi_1, \psi_2, \ldots, \psi_{N_{\text{rot}}}\}$ to be optimized based on the *strongest* associated score.

B. Lamarckian Genetic Algorithm

This is the systematic method employed for generating poses, and thus, for exploring the landscape described by the scoring function. A Lamarckian Genetic Algorithm (LGA) hybridizes the principles of biological evolution by coupling a genetic algorithm (GA) used as a global search, with a local search (LS) method that refines the poses produced by the GA.

Each pose is treated as a member of a population, i.e., an *individual*, which is represented by its *genotype*, i.e., set of *genes*. New individuals are generated through rules of genetic evolution (i.e., via crossover, mutation, and selection) from ancentors (i.e., individuals from a previous generation). The subset of poses whose scores were improved by LS iterations are *re-introduced* into the genetic population. A single LGA run terminates when a pre-defined max. number of score evaluations (default: $N^{\text{MAX}}_{\text{score-evals}}$ = 2 500 000) or generations (default: $N^{\text{MAX}}_{\text{gens}}$ = 27 000) is reached, whichever comes first.

Algorithm 1: Lamarckian Genetic Algorithm (LGA)

```
Function AutoDock
    /* Coarse-Level Parallelism */
    for each LGA-run do
        while (N_score-evals < N^MAX_score-evals) and (N_gens < N^MAX_gens) do
            /* Medium-Level Parallelism */
            GA (population)
            /* Medium-Level Parallelism */
            for individual in random-subset (population) do
                LS (get-genotype (individual))
```

C. Scoring Function

The molecular interactions are quantified with the semi-empirical free-energy force field (kcal/mol) in Equation 1. As detailed in [7], while all these terms are characterized by dimensionless coefficients (W_{vdw}, W_{hb}, W_{el}, W_{ds}, W_{rot}) as well as by look-up tables (A, B, C, D, S, V, E, q), the score is mainly determined by the interatomic distance r_{ij} (between atoms i and j) that changes during the docking process.

$$\text{SF} = \sum_{i,j} \left[W_{\text{vdw}} \left(\frac{A_{ij}}{r_{ij}^{12}} - \frac{B_{ij}}{r_{ij}^{6}} \right) + W_{\text{hb}} E(t) \left(\frac{C_{ij}}{r_{ij}^{12}} - \frac{D_{ij}}{r_{ij}^{10}} \right) + W_{\text{el}} \left(\frac{q_i q_j}{\epsilon(r_{ij}) r_{ij}} \right) + W_{\text{ds}} \left(S_i V_j + S_j V_i \right) e^{\frac{-r_{ij}^2}{2\sigma^2}} \right] + W_{\text{rot}} N_{\text{rot}} \tag{1}$$

D. Local Search method

AUTODOCK uses the Solis-Wets method [8] as LS to generate a `new-genotype1` by adding small random `delta` changes to each gene of an initial genotype. Then, if the score is not minimized by `new-genotype1`, changes are subtracted instead of being added, and another comparison using instead the score of `new-genotype2` is performed. At each iteration, the change size (`step`) is either increased or decreased depending on whether the number of consecutive successful or failed attempts is greater than four, respectively. Similar to LGA, the LS termination is runtime-defined, specifically when either the number of iterations or the change size reach their max. (default: $N^{\text{MAX}}_{\text{LS-iters}}$ = 300) or min. (default: step^{MIN} = 0.01) limits, respectively.

Algorithm 2: Solis-Wets (SW) local search

```
/* Fine-Level Parallelism */
Function SW (genotype)
    while (N_LS-iters < N^MAX_LS-iters) and (step > step^MIN) do
        delta = create-delta (step)
        // new-genotype1
        for each gene in N_genes do
            new-gene1 = gene + delta
        if SF (new-genotype1) < SF (genotype) then
            genotype = new-genotype1
            success++; fail = 0
        else
            // new-genotype2
            for each gene in N_genes do
                new-gene2 = gene - delta
            if SF (new-genotype2) < SF (genotype) then
                genotype = new-genotype2
                success++; fail = 0
            else
                success = 0; fail++
        step = update-step (success, fail)
```

E. Relevant remarks

AUTODOCK explores ligand poses by performing independent LGA runs, each up to a max. user-defined count of score evaluations ($N_{\text{score-evals}}$) and genetic generations (N_{gens}). The Solis-Wets LS has termination criteria (based on $N_{\text{LS-iters}}$ and step) that can be set by the user through program arguments, too. The fact that AUTODOCK comprises a self-adaptive search implies that the number of score evaluations on every LS call is variable, and depends on how the search evolves during execution. The heuristics in the GA (crossover, mutation, and selection), as well as the runtime adaptability of the LS, make

AUTODOCK capable of exploring different ligand poses on every docking job, but also make it a very irregular application. This irregularity is exacerbated when molecules with different number of atoms and bonds are analyzed, which is often the case in virtual screening.

AUTODOCK has been originally developed as a single-threaded application. Its computational intensity increases mainly with larger values of the aforementioned program arguments and more complex molecular structures. The program bottleneck is the scoring function [9], [10], whose accumulated evaluations across all LGA runs take up to $\sim$90% of the total execution time. Algorithms 1 and 2 have three levels of parallelism: coarse (entire LGA runs), medium (individuals), and fine (genes, scoring) that we leverage in order to speedup overall AUTODOCK executions.

III. OPENCL PARALLELIZATION

OpenCL provides a platform- and vendor-agnostic parallel framework, which allows the usage of the same source code to target accelerators with different underlying hardware architectures. This section describes our experience designing AUTODOCK-GPU focusing on the programming aspects.

A. Design considerations for host code

1) Re-structuring the original program into a parallel-friendly version: the main obstacle encountered when parallelizing AUTODOCK was the fact that it was developed with a focus almost exclusively on the docking functionality. While this is convenient for software prototyping, this generally leads to code without clear separation between control- or computation-dominated regions, the later ones being suitable for acceleration. Although it has been long well-understood that scoring computations are the main performance bottleneck, acceleration attempts based *only* on score computations did not yield significant speedups. Moreover, their scope was limited to the GA, i.e., completely excluding LS from parallelization [11], [12].

The original AUTODOCK is mainly coded as a large `switch` statement, where the program progressively reads a docking configuration file and executes certain tasks. Depending on the order of the configuration options, the execution often intertwines I/O and compute tasks. In contrast, for an efficient parallelization, we have significantly re-structured the original code so I/O (configuration readout, result write) and actual docking computations have been separated in different code regions, and their execution takes place in different program stages. The resulting code clearly exposes GA, and specially LS, as the most runtime-consuming program regions. We thus chose these two functions, comprising several score evaluations, for acceleration in AUTODOCK-GPU.

2) Transforming tree structures into arrays for parallel on-device processing: AUTODOCK constructs a tree-like structure containing ligand atoms affected by a given rotatable bond. In order to determine a given pose, the program recursively rotates every tree node. As recursion is not a suitable coding pattern for hardware accelerators, the tree is transformed into an array comprising a list of rotations. Every array element is a 32-bit `int` that basically encodes the ID of the atom to be rotated, and the type of rotation.

Similarly, as not all ligand atoms actually contribute to the score, an array with each of its elements carrying the IDs of contributor atoms i and j is constructed in order to avoid traversing the tree when computing the score. These are the so-called *intramolecular contributors* (Section III-C1).

B. OpenCL programming models

OpenCL supports the two parallel programming models described as follows. *Data parallelization* is achieved when each processor within a multi-processor system performs identical tasks on different pieces of distributed data. Computationally-intensive parts of the program are executed on device as *kernels*. Kernels are processed by multiple *work-items*, which can be thought as processing threads. Work-items are grouped into *work-groups*, which are independent from each each other, and each being executed on a device *compute unit* (CU). *Task parallelization* is instead achieved by distributing different tasks across multiple processors. In OpenCL, a task can be implemented with a kernel running a *single* work-item.

Massively data-parallel accelerators like GPUs are widespread, and thus, the data parallelism aspects of OpenCL have been of primary interest over the years [13]. However, as other hardware accelerators like FPGAs handle parallelism differently (i.e., leveraging pipelining instead of SIMD), task-parallelization is receiving increasing attention. Our work here focuses on a data-parallelization of AUTODOCK due to the higher performance achieved on GPUs and CPUs. In Section V, we describe the usage of task-parallelization in order to improve the performance on FPGAs.

C. Data parallelization on accelerator devices

1) Re-designing the scoring function: based on the group of atoms involved, AUTODOCK expresses their score as the sum of two *independent* interactions: *intermolecular* (receptor atoms - ligand atoms) and *intramolecular* (ligand atoms - ligand atoms). For speeding up calculations, AUTODOCK interpolates from tabulated values, instead of evaluating Equation 1 analytically. Basically, the code pre-calculates both types of interactions, and generates look-up tables to be accessed during docking. Thus, the receptor is instead represented by grid maps for every atom type of the ligand. Grid values are processed using trilinear interpolation. The intramolecular scores are modeled as arrays carrying interaction values that vary according to the interatomic distance.

AUTODOCK-GPU follows the same approach, but *only* for the intermolecular interactions. For the intramolecular component, we opted to instead perform the analytical calculation, since the number of ligand-ligand atomic pairs is much lower than that involved in receptor-ligand interactions. Doing so provides more accurate calculations, and leverages the massive compute power available on modern accelerators.

Algorithm 3 shows the SF code structure in AUTODOCK-GPU. The pose-calculation accesses the rotation list from

Section III-A2 and outputs the resulting atomic positions. The intermolecular component loops through all N_{atom} ligand atoms. The x, y, z coordinates of a given ligand atom are used for accessing a given receptor grid. This implies *random* memory accesses that cannot be optimized since addresses depend on unpredictable coordinate values. The intramolecular component accesses the atomic contributor-list from Section III-A2. For every pair of atoms involved in a given loop iteration, their coordinates are retrieved and their corresponding spatial distance is calculated. This also involves random memory accesses to look-up tables (Section II-C), which depend on the types of atoms being involved.

Algorithm 3: Scoring Function (SF)

```
/* Fine-Level Parallelism */
Function SF (genotype)
    for each rot-item in N_pose-rot do
        PoseCalculation
    for each lig-atom in N_atom do
        InterInteraction
    for each intra-pair in N_intra-contrib do
        IntraInteraction
```

2) Mapping operations into OpenCL elements: based on our prior work [9], the parallelization consists of mapping the main AUTODOCK operations (i.e., GA, LS, SF) onto OpenCL processing elements (i.e., kernels, work-groups, work-items) to achieve a suitable level of parallelism (i.e., coarse, medium, fine).

A docking job is composed of R independent LGA runs, where each run (Run_{ID}: 0, 1, 2, ..., R-1) processes a population of P individuals (Ind_{ID}: 0, 1, 2, ..., P-1) through a GA, followed by an LS refinement (Section II-D). These GA and LS functions involve score computations over several individuals, and thus, are implemented as the OpenCL kernels Krnl_GA and Krnl_LS (Fig. 1).

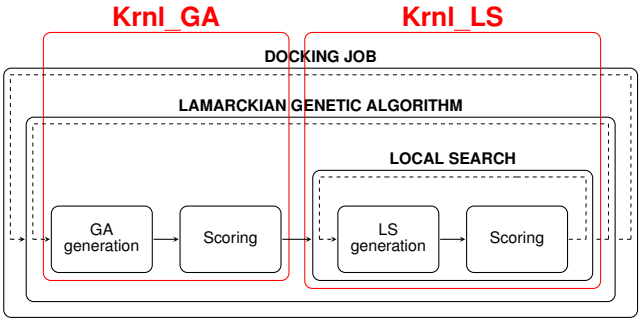

Fig. 1. Functions GA and LS are mapped onto kernels that execute nested loops controlling LGA runs and their inner processing of individuals.

In AUTODOCK, LGA runs and their inner processing of individuals (by either GA or LS) are controlled by nested loops. In AUTODOCK-GPU, such loops are merged into a single one to increase parallelism. Therefore, $R \times P$ individuals from multiple LGA runs are processed in parallel, each by an OpenCL work-group, thus achieving coarse- and medium-level parallelization (Fig. 2). The functional correctness is ensured by keeping track of the Run_{ID} associated to each Ind_{ID}, which is mapped to a work-group (WG_{ID}) as follows: $\text{WG}_{\text{ID}} = \text{Run}_{\text{ID}} \times P + \text{Ind}_{\text{ID}}$. Moreover, processing an individual involves fine-grained tasks carried out by OpenCL work-items, thus achieving fine-grained parallelization. Such {GA/LS generation, pose calculation, intermolecular- and intramolecular-interaction} tasks have different computational intensities that depend on $\{N_{\text{genes}}, N_{\text{pose-rot}}, N_{\text{atom}}, N_{\text{intra-contrib}}\}$, respectively. Furthermore, in order to ensure correct score calculations and genotype updates, OpenCL work-items have to be synchronized at certain program points. For this purpose, the OpenCL `barrier()` function is appropriately called.

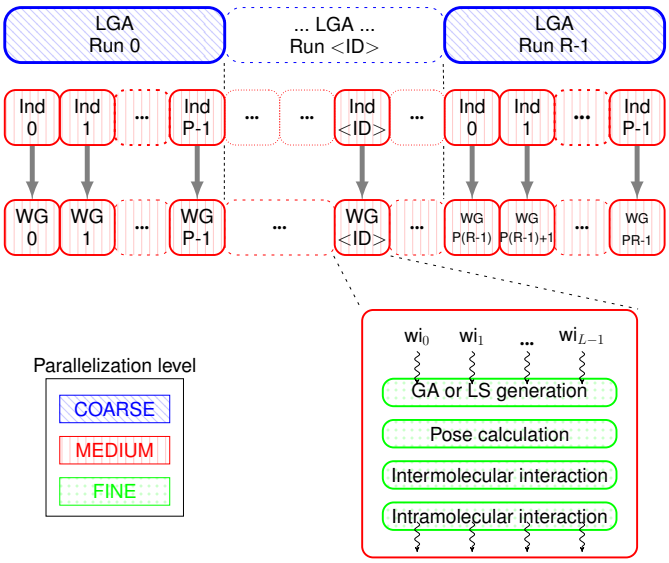

Fig. 2. A population processed by an LGA run (Run_{ID}) is decomposed into their individuals, and each individual (Ind_{ID}) is mapped onto a work-group (WG_{ID}). Fine-grained tasks are processed by work-items ($\text{wi}_0 \ldots \text{wi}_{L-1}$).

3) Integrating alternative Local Search methods: the pose refinement provided by LS can be further leveraged by utilizing more effective search methods. AUTODOCK-GPU has been developed considering the easy incorporation of alternatives to Solis-Wets. Our previous work [6] provides a mathematical background on the newly-incorporated ADADELTA method, and evaluates the enhancement of predicted poses. Here, we focus instead in its related implementation aspects.

Algorithm 4 shows the core of ADADELTA: the gradient of the scoring function. Since scores are calculated as the sum of intermolecular and intramolecular interactions (Section III-C1), the gradients are hence composed of analogous parts calculated using numerical and analytical derivatives, respectively. Algorithm 5 shows how ADADELTA performs $N^{\text{MAX}}_{\text{LS-iters}}$ iterations, and uses the gradient for generating a `new-genotype`. Similarly, ADADELTA carries out irregular computations involved within score and gradient calculations. It is important to highlight that the LS methods considered are mutually exclusive, i.e., Krnl_LS performs *either* Solis-Wets

or ADADELTA, in a given docking job.

Algorithm 4: Gradient Calculation (GC)

```
/* Fine-Level Parallelism */
Function GC (genotype)
    /* Gradients in atomic space */
    for each rot-item in N_pose-rot do
        PoseCalculation
    for each lig-atom in N_atom do
        InterGradient
    for each intra-pair in N_intra-contrib do
        IntraGradient
    /* Conversion into genetic space */
    Gtrans // Translational gradients
    Grigidrot // Rigid-body rotation gradients
    Grotbond // Rotatable-bond gradients
```

Algorithm 5: ADADELTA (AD) local search

```
/* Fine-Level Parallelism */
Function AD (genotype)
    gradient = GC (genotype)
    while (N_LS-iters < N^MAX_LS-iters) do
        new-genotype = update-rule (genotype, gradient)
        if SF (new-genotype) < SF (genotype) then
            genotype = new-genotype
        gradient = GC (genotype)
```

4) Further improvements based on similar loop structures: both SF and GC calculate poses identically, and share the same loop-structure for their intermolecular and intramolecular components. In order to leverage data locality in the ADADELTA implementation, SF and GC calculations are grouped together as much as possible. Basically, a single pose calculation is used for both scores and gradients, whereas structure-equivalent SF and GC calculations are *fused* into single intermolecular and intramolecular loops. This code re-factoring results in faster ADADELTA executions (∼18%) compared an initial design where SF and GC calculations were called *separately*.

D. OpenCL-specific aspects

In previous experiments [6], we found that in most cases higher speedups are achieved on GPUs rather than on CPUs. Since an docking program is typically employed at screening compound libraries of thousands of candidate ligands, a domain scientist would prefer to utilize (if available) the *fastest* device type in the first place. Based on this typical scenario, our application development was entirely performed targeting the AMD Vega 64 (Table I). Since optimizations applied on this GPU would not be beneficial on other devices (e.g., CPUs, or those with a different architecture), and because tool support for optimization analysis of OpenCL programs might not be available for other targets (e.g., profiling for Nvidia GPUs), we opted for a platform-agnostic development, so that AUTODOCK-GPU is *functionally* portable, and not dependent on vendor-specific optimizations nor extensions.

1) Arithmetic precision: floating-point (FP) calculations in AUTODOCK are performed in *double* precision (64 bits). According to [9], [10], [14], using a reasonably-lower precision does not deteriorate the quality of pose predictions. Hence, OpenCL `native_*` built-in functions, as well as *single* precision (32 bits) operands were employed.

2) Memory layout: the chosen layout can significantly impact performance [15], [16]. In general, struct-of-arrays (SoA) helps *adjacent* work-items to access adjacent memory (achieving coalescing on GPUs), whereas array-of-structs (AoS) helps *individual* work-items to access adjacent memory (leveraging cache hierarchies on CPUs). Although a runtime-selectable layout as used in [15] would be the optimum, we opted to use SoA because: *First*, the initial target device was a GPU. *Second*, the significant code changes required for AoS are beyond the scope of evaluating a single-source code.

3) Vectorization: SIMD architectures are exposed differently to an OpenCL than to a C developer [16]. Developing a single-source code that maximizes the SIMD efficiency on several devices is difficult due to the runtime peculiarities introduced by each vendor. For instance:

- AMD: for CPUs, the *explicit* usage of vector types (e.g., `float4`) enables the generation of SSE/AVX code. While it is often beneficial for both CPUs and GPUs, using vectorization for GPUs can *negatively* affect performance as it requires more vector registers for storage [17].
- Nvidia: there is no performance benefit from using vector types since the CUDA architecture is scalar [18].
- Intel: their runtime carries out *implicit* vectorization, which consists of packing several work-items and executing them with SIMD instructions. Vector operations already in the code are scalarized and re-vectorized [19].

Based on that, we opted to use vector types only for easier development in critical code sections, e.g., for some of the data conversion required for gradients. That calculation is coded in a simpler manner by calling built-in vector functions (e.g., `cross`, `dot`, `fast_length`), instead of writing a verbose scalar counterpart.

4) Work distribution: the global work size (NDR_{size}), i.e., the total number of work-items for each kernel, was configured as a function of: the number of LGA runs (R), population size (P), LS rate (lsrate), and work-group size (WG_{size}), namely:

$$\text{NDR}_{\text{size}}^{\mathsf{Krnl\_GA}} = \{R \times P \times \text{WG}_{\text{size}},\ 1,\ 1\} \quad (2)$$

$$\text{NDR}_{\text{size}}^{\mathsf{Krnl\_LS}} = \{R \times P \times \mathsf{lsrate} \times \text{WG}_{\text{size}},\ 1,\ 1\} \quad (3)$$

For all experiments, the first two parameters were set as R = 100 and P = 150. lsrate indicates the percentage of a population that undergoes LS, which was set to the max. possible, i.e., lsrate = 100%. For instance, setting $\text{WG}_{\text{size}}^{\text{GPU}} = 64$ and $\text{WG}_{\text{size}}^{\text{CPU}} = 16$ would result in $\text{NDR}_{\text{size}}^{\text{GPU}} = \{960000,\ 1,\ 1\}$ and $\text{NDR}_{\text{size}}^{\text{CPU}} = \{240000,\ 1,\ 1\}$. Section IV-B presents the performance impact of different WG_{size} values.

IV. EVALUATION

Here we present the impact of relevant factors on overall performance, such as the OpenCL work-group size, molecular

complexity, and employed LS method.

A. Experimental setup

1) Program configuration: besides the number of LGA runs and population size (already set as R = 100 and P = 150 in Section III-D), other relevant AUTODOCK-GPU parameters are $N^{\text{MAX}}_{\text{score-evals}}$ and $N^{\text{MAX}}_{\text{gens}}$. The latter was set to a considerably larger value ($N^{\text{MAX}}_{\text{gens}}$ = 99 999) than the default one (27 000) in order to ensure that the program is terminated only when it reaches $N^{\text{MAX}}_{\text{score-evals}}$ = 2 048 000. Other parameters were left as default [20], unless otherwise specified (e.g., lsrate = 100%).

2) Dataset: a set of 20 ligand-receptor inputs was selected. We included eleven entries from [21] (IDs: `1u4d`, `1xoz`, `1yv3`, `1owe`, `1oyt`, `1ywr`, `1t46`, `2bm2`, `1mzc`, `1r55`, `1kzk`), four from [22] (IDs: `3s8o`, `1hfs`, `1jyq`, `2d1o`), and five from [23] (IDs: `5wlo`, `5kao`, `3drf`, `4er4`, `3er5`). Our set covers a range of up to 31 rotatable bonds, considering that AUTODOCK supports $N^{\text{MAX}}_{\text{rot}}$ = 32.

3) Hardware: for the baseline test, i.e., the measurement of the execution time of the original *single-threaded* AUTODOCK4.2.6 (implementing only Solis-Wets), we used an Intel Xeon Platinum 8124M @3.0 GHz CPU core. For parallel executions, accelerators based on commercial devices were selected (Table I). An AMD Vega 64 GPU was used as the main development platform. The more powerful Titan V GPU, and selected CPUs from Amazon Web Services (AWS) were used as porting-target devices. Regarding CPUs, high-performance instances were used: `c5.18xlarge` (virtualized) and `m5.metal` (bare-metal).

TABLE I
SETUP IN TERMS OF INSTANCE TYPE, PEAK MEMORY BANDWIDTH AND SINGLE-PRECISION FP PERFORMANCE, AND OPENCL COMPUTE UNITS.

Device	Instance type	GB/s	GFLOP/s	# CU
AMD Radeon RX Vega 64	On-premise GPU	483	12 660	64
Nvidia Volta Titan V	On-premise GPU	651	14 900	80
Intel Xeon Platinum 8124M @3.0 GHz	AWS 36-core CPU `c5.18xlarge`	260	3456	36
Intel Xeon Platinum 8175M @2.5 GHz	AWS 96-core CPU `m5.metal`	260	7680	96

B. Impact of OpenCL work-group size

We analyzed the speedups achieved on all selected accelerators when using different OpenCL work-group size (WG_{size}) configurations, e.g., {16, 32, 64, 128, 256} work-items. Fig. 3 shows the cases of `1u4d`, `3s8o`, and `3er5`.

For CPUs, as found in our previous work [9], a WG_{size} of 16 work-items clearly leads to higher speedups on both `c5.18xlarge` and `m5.metal` instances.

For GPUs, in contrast to [9], where a WG_{size} of 64 work-items was the fastest configuration for the only tested lsrate = 6% on an AMD R9 290X GPU, Fig. 3 shows that higher speedups can be obtained also with either smaller (32) or larger (128) values of WG_{size}, regardless of the chosen device. According to the vendor's guidelines [17], [18], a suitable WG_{size} is an integer multiple of either an AMD *wavefront* size (64), or a Nvidia *warp* size (32). For next

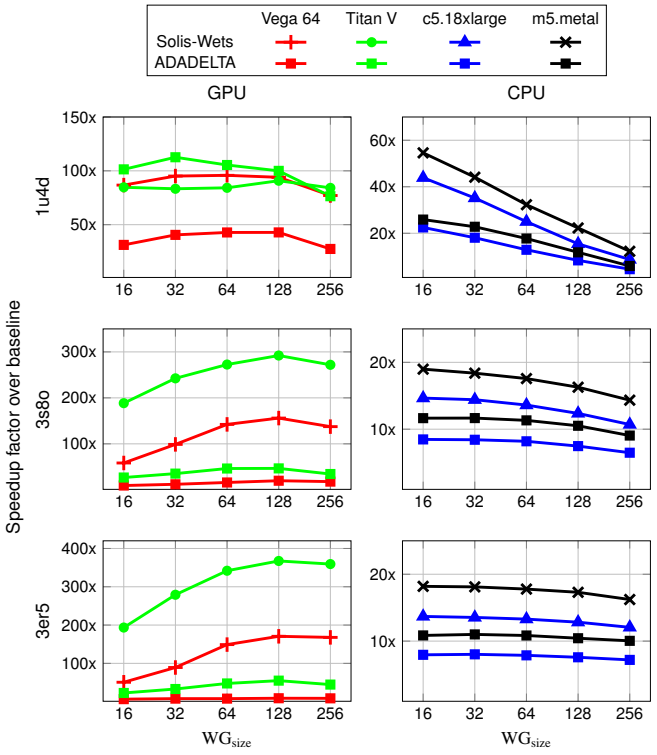

Fig. 3. Speedups of AUTODOCK-GPU vs. AUTODOCK for different work-group sizes (R = 100 LGA runs, lsrate = 100%). Vertical scales are different.

experiments, we set WG_{size} = 64, i.e., the min. integer multiple for any GPU from these two vendors. While the optimum WG_{size} value will depend on the dataset used, by using WG_{size} = 64 on GPUs, we aim to minimize the inter work-group communication overhead, which might slow down the program for larger WG_{size} values.

C. Impact of molecular complexity

Fig. 4 provides an overall comparison of the speedups achieved on all selected devices for our entire dataset. For both Solis-Wets and ADADELTA executions, the reported speedups were obtained with respect to the baseline AUTODOCK running the Solis-Wets method. Although calculating ADADELTA speedups with respect to a Solis-Wets baseline is somewhat arguable, they still show meaningful performance gains from the parallelization of a more complex search method.

In general, on any given device, it can be observed that ADADELTA speedups are significantly *lower* than their corresponding Solis-Wets ones. The reason is the more computationally-demanding genotype generation in ADADELTA, specifically due to the gradient calculation (Section III-C3) compared to the simpler random generation in Solis-Wets (Section II-D).

For a given LS method, speedup factors show a behavior affected by the input complexity, i.e., N_{rot} and N_{atom} values. Running Solis-Wets, the respective speedups using {`1u4d`, `3s8o`, `3er5`} as inputs when running on:

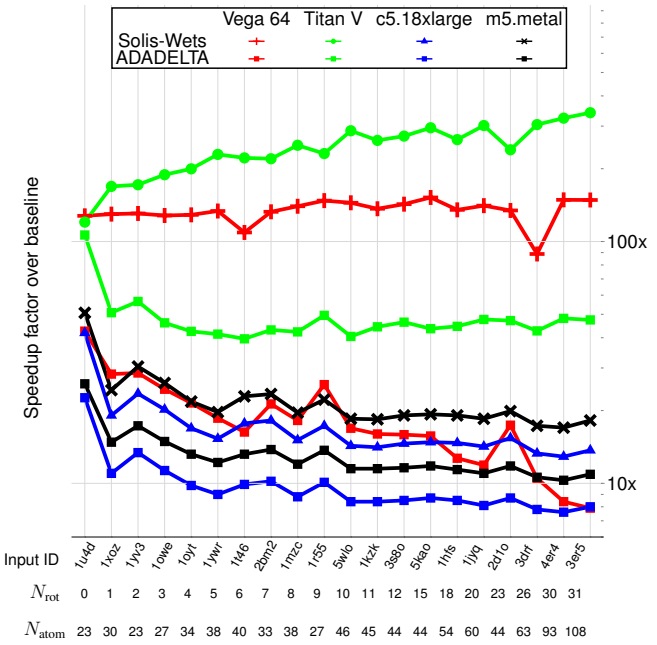

Fig. 4. Speedups of AUTODOCK-GPU vs. AUTODOCK achieved for the entire dataset (R = 100 LGA runs, lsrate = 100%).

- GPUs, tend to *increase* with larger inputs: e.g., {128x, 143x, 149x} on Vega 64, and {120x, 273x, 341x} on Titan V.
- CPUs, tend to *decrease* with larger inputs: e.g., {42x, 15x, 14x} on `c5.18xlarge`, and {51x, 19x, 18x} on `m5.metal`.

Running ADADELTA on any device results in decreasing speedups as the inputs become more complex. This can be attributed to the irregularity of the gradient calculation:

- First, the loop bounds. $N_{\text{pose-rot}}$ (dependent on N_{rot}), N_{atom} and $N_{\text{intra-contrib}}$ (dependent on N_{atom}) are larger for more-complex inputs, and in turn, result in longer and input-dependent processing times.
- Second, its limited parallelism. This procedure is performed in a gene-type manner, and thus, also involves three independent fine-grained tasks (`Gtrans`, `Grigidrot`, `Grotbond` in Algorithm 4) that can be distributed across the work-items (of a work-group) in different ways. A simple way would be to run these tasks simultaneously, each by a different work-item. Another way would be to parallelize these tasks with as many work-items as possible, running only one task at a time.

We opted to use a combination of both ways: While `Gtrans` and `Grigidrot` each present a loop with an upper bound of N_{atom}, they must also perform sequences of data-dependent operations, which are not suitable for parallelism. Thus, each of these two tasks was executed by a single work-item. The operations within `Grotbond` are also data-dependent, but are repeated for each rotatable bond. Hence, `Grotbond` is processed by N_{rot} work-items.

Finally, in most cases, GPUs achieve higher efficiencies than CPUs when using the *same* LS method. A notable exception to this behavior occurs when using ADADELTA with larger inputs {`3drf`, `4er4`, `3er5`} (right side in Fig. 4), where the achieved speedup is {11x, 8x, 8x} on the Vega 64, and {11x, 10x, 11x} on the `m5.metal`, respectively. The typically faster executions of GPUs can be attributed to the more suitable mapping of OpenCL elements onto their hardware. On CPUs, however, each work-group is executed by a single CPU core, and thus, the group's work-items are executed *serially* [15], [24]. The purpose of such serialization is to avoid the excessive synchronization penalties incurred if work-items within a work-group were executed in parallel, since work-items on CPUs are mapped to OS threads, instead of using the lighter-weight hardware threads available on GPUs.

V. PORTING ONTO OTHER ACCELERATORS

This section summarizes our experiences when porting AUTODOCK to FPGAs. The focus is on device rather than host code, whose design (Section III-A) remains mostly unchanged.

Porting the proposed *data-parallel* design onto FPGAs was a more involved process. In our previous work [9], code refactoring (e.g., replacing `struct` kernel arguments with built-in types, to avoid compiler errors) and a number of compiler updates were required to achieve a functionally-correct implementation on a Xilinx Virtex 7 FPGA. Unfortunately, this was three-orders of magnitude *slower* than the serial baseline. Moreover, porting the same design onto an AWS `f1.2xlarge` instance [25] resulted in executions hanging indefinitely. As this behavior was not manifested during emulation, incrementally enabling code helped to determine that the update of the `__local` atomic coordinates in `PoseCalculation` (Section II-C) was causing the problem.

Although the root causes of latter problems are not yet understood, even if they were resolved, we would expect very slow executions on `f1.2xlarge`. In general, performance gains in a data-parallel design are expected to result from the *simultaneous* execution of work-groups over the available CUs. Applying this approach on FPGAs implies replicating (as often as possible) those CUs to achieve higher parallelism. However, this replication is limited in practice, due to the constrained *area* of the programmable logic, as well as the typically lower memory bandwidth on FPGA-based accelerators (in tens of GB/s for DDR technology) compared to GPUs.

For a more comprehensive evaluation, our *task-parallel* design for FPGAs [14] is discussed here. Although it cannot keep up with GPUs and CPUs, it is ∼3x faster than the serial baseline when running the Solis-Wets LS on an Intel Arria 10 FPGA. Instead of replicating CUs, task parallelism on FPGAs relies on deeper pipelining and custom memory hierarchies. Our fastest configuration (also the largest in terms of required resources) contains 27 kernels (each processing a *single* work-item) connected with OpenCL blocking/non-blocking pipes. In terms of overall functionality, this is comparable to other implementations running Solis-Wets only (Section VI). The FPGA implementation does not include the more advanced

ADADELTA algorithm, as its incorporation would require significant architectural changes that would exceed the capacity of the target FPGA. We found that the fastest configuration, which includes nine replicas of the Solis-Wets LS kernel, *barely* fits on the Arria 10 FPGA. Adding ADADELTA would require additional hardware area that could only be freed-up by reducing the number of LS kernel instances. This would slow down AUTODOCK on the FPGA even further.

Finally, the functional portability on FPGAs is currently dependent on the specific language constructs actually supported by OpenCL-to-FPGA tools. For instance, it was not possible to port the task-parallel design onto `f1.2xlarge` instances, since non-blocking pipes were not supported [26]. Despite the significant improvements in OpenCL-to-FPGA tools over the years, the lower development productivity compared to GPUs/CPUs can be attributed mainly to the far longer build times relative to software. E.g., building an FPGA binary required around eight hours of tool runtime for AUTODOCK.

VI. Related work

Regarding AUTODOCK acceleration, Pechan et al. [27] published a survey reporting CUDA-based approaches that exclude [11] and include [10] the Solis-Wets LS. In addition, that survey describes a Verilog design for FPGAs that was competitive with GPUs [10], when both include the Solis-Wets LS. Moreover, Mendonça et al. [12] proposed a hybrid CPU-GPU design utilizing OpenMP and CUDA, but without the Solis-Wets LS. Among the aforementioned studies, only those by Pechan et al. [10] include a LS method (only Solis-Wets), and thus, are the only ones truly comparable to ours. However, as *their* experiments were performed on *older* devices, their reported speedups are not directly comparable.

VII. Conclusions

We described the challenges at parallelizing the irregular AUTODOCK code using OpenCL, and discussed the design considerations in both host and device code. To cope with the irregularity of AUTODOCK, large-scale code re-structuring was required. For instance, by transforming the tree-like data structures into arrays, it was possible it distribute atomic-rotations and score calculations into multiple OpenCL work-items. Our approach does not focus only on the major performance bottleneck (score function), but also on simultaneously processing multiple ligand poses with OpenCL work-groups.

The performance of AUTODOCK-GPU was evaluated for both the Solis-Wets and the ADADELTA local-search methods, as well as for different work-group sizes. Using 64 work-items on a Titan V GPU, and 16 work-items on a Xeon Platinum 8175M CPU, AUTODOCK-GPU achieves max. speedup factors of 341x and 51x, respectively. Moreover, our results show that the AUTODOCK-GPU performance does depend on the characteristics of the specific compounds analyzed.

Finally, preliminary experiments on FPGAs show that the performance portability was not achievable. This suggests that hardware-awareness is still required for efficiently parallelizing irregular codes such as AUTODOCK for different compute platforms, even when using abstractions such as OpenCL. We hope that our experience will be useful when porting other irregular applications to different target hardware architectures.

References

[1] I. Halperin et al., "Principles of docking: An overview of search algorithms and a guide to scoring functions," *J. Proteins: Structure, Function, and Bioinformatics*, 2002.

[2] U. Yadava, "Search algorithms and scoring methods in protein-ligand docking," *Int. J. Endocrinology & Metabolism*, 2018.

[3] "FightAIDS@Home." [Online]. Available: http://fightaidsathome.scripps.edu

[4] Khronos Group, "OpenCL Resources." [Online]. Available: https://www.khronos.org/opencl/resources

[5] S. LeGrand et al., "GPU-Accelerated Drug Discovery with Docking on the Summit Supercomputer: Porting, Optimization, and Application to COVID-19 Research," *arXiv*, 2020. [Online]. Available: https://arxiv.org/abs/2007.03678

[6] D. Santos-Martins et al., "Accelerating AutoDock4 with GPUs and Gradient-Based Local Search," *ChemRxiv*, 2019. [Online]. Available: https://chemrxiv.org/articles/Accelerating_AutoDock4_with_GPUs_and_Gradient-Based_Local_Search/9702389/1

[7] R. Huey et al., "A semiempirical free energy force field with charge-based desolvation," *J. Comp. Chem.*, 2007.

[8] F. J. Solis et al., "Minimization by Random Search Techniques," *J. Mathematics of Operations Research*, 1981.

[9] L. Solis-Vasquez et al., "A Performance and Energy Evaluation of OpenCL-accelerated Molecular Docking," in *5th IWOCL*. ACM, 2017.

[10] I. Pechan et al., "Molecular Docking on FPGA and GPU Platforms," in *21st Int. Conf. FPL*. IEEE, 2011.

[11] K. Sarnath et al., "Porting Autodock to CUDA," in *Congress on Evolutionary Computation*. IEEE, 2010.

[12] E. Mendonça et al., "Accelerating Docking Simulation Using Multicore and GPU Systems," in *17th ICCSA*. Springer, 2017.

[13] P. Jääskeläinen et al., "Exploiting Task Parallelism with OpenCL: A Case Study," *J. Signal Processing Systems*, 2019.

[14] L. Solis-Vasquez et al., "A Case Study in Using OpenCL on FPGAs: Creating an Open-Source Accelerator of the AutoDock Molecular Docking Software," in *5th FSP*. VDE Verlag, 2018.

[15] S. J. Pennycook et al., "An investigation of the performance portability of OpenCL," *JPDC*, 2013.

[16] S. J. Pennycook et al., "Developing Performance-Portable Molecular Dynamics Kernels in OpenCL," in *SC: High Performance Computing, Networking Storage and Analysis*. IEEE, 2012.

[17] "AMD OpenCL Programming Optimization Guide," 2015. [Online]. Available: http://developer.amd.com/wordpress/media/2013/12/AMD_OpenCL_Programming_Optimization_Guide2.pdf

[18] "Nvidia OpenCL Best Practices Guide 1.0." 2009. [Online]. Available: https://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf

[19] "Intel SDK for OpenCL: Benefiting from Implicit Vectorization," 2018. [Online]. Available: https://software.intel.com/en-us/iocl-tec-opg-benefiting-from-implicit-vectorization

[20] G. M. Morris et al., "AutoDock4 and AutoDockTools4: Automated docking with selective receptor flexibility," *J. Comp. Chem.*, 2009.

[21] M. J. Hartshorn et al., "Diverse, high-quality test set for the validation of protein-ligand docking performance," *J. Medicinal Chemistry*, 2007.

[22] Y. Li et al., "Comparative assessment of scoring functions on an updated benchmark: 2. Evaluation methods and general results," *J. Chemical Information and Modeling*, 2014.

[23] H. M. Berman et al., "The Protein Data Bank," *J. Nucleic Acids Research*, 2000.

[24] D. Kaeli et al., *Heterogeneous Computing with OpenCL 2.0*, 3rd ed. Morgan Kaufmann, 2015.

[25] "Amazon EC2 F1 Instances." [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1

[26] [Online]. Available: https://forums.xilinx.com/t5/SDAccel/SDAccel-OpenCL-examples-with-non-blocking-pipe-functions/td-p/912707

[27] I. Pechan et al., "Hardware Accelerated Molecular Docking: A Survey," in *Bioinformatics*. InTech, 2012.

A. Artifact Appendix

A.1 Abstract

This artifact appendix provides instructions on how to retrieve, compile, and evaluate the developed AUTODOCK-GPU code. This includes instructions on how to obtain the input datasets, as well as scripts to regenerate the execution runtimes, which were used for the speedup factors reported in this paper. This will allow the evaluation of our results on any of the accelerator devices described in the paper, like Vega 64 GPU, Titan V GPU, AWS `c5.18xlarge` and `m5.metal` CPU instances. Moreover, this will allow evaluation of AUTODOCK-GPU on any other GPU/CPU-based accelerators supporting OpenCL.

A.2 Artifact check-list (meta-information)

- **Algorithm:** OpenCL-based parallelization of AUTODOCK.
- **Program:** AUTODOCK-GPU (all sources can be downloaded from GitHub), version 1.2 (commit `eed190f`), size ∼50 MB.
- **Compilation:** `g++` 6 or above.
- **Binary:** Source code and scripts included to generate binaries.
- **Data set:** Molecular structures prepared for both AUTODOCK and AUTODOCK-GPU (all input files can be downloaded from Zenodo), ready to use, size ∼1.4 GB.
- **Run-time environment:** AUTODOCK-GPU requires any Linux distribution supporting OpenCL. We recommend Ubuntu 18.04 for verifying results on AWS CPU instances. OpenCL drivers provided by AMD, Nvidia, and Intel are required. No need of root access.
- **Hardware:** We recommend AWS `c5.18xlarge` and `m5.metal` instances featuring Intel Xeon Platinum CPUs, as well as AMD Vega 64 and Nvidia Titan V GPUs.
- **Execution:** Sole user. AUTODOCK-GPU benchmarks on all chosen four accelerators take ∼7 hours. AUTODOCK benchmark (baseline) takes ∼16 hours.
- **Metrics:** Execution runtimes in seconds.
- **Output:** Console indicating execution runtime of a given experiment. Additional: *docking log files* (.dlg) indicating execution runtime and resulting molecular poses.
- **Experiments:** Bash scripts (provided). Maximum allowable variation of execution runtimes: 10%.
- **How much disk space required (approximately)?:** Maximum: 20 GB.
- **How much time is needed to prepare workflow (approximately)?:** Two hours.
- **How much time is needed to complete experiments (approximately)?:** 24 hours.
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** GNU Lesser General Public License.
- **Data licenses (if publicly available)?:** Creative Commons Attribution 4.0 International.
- **Archived (provide DOI)?:** Yes. `DOI: 10.5281/zenodo.4073350`

A.3 Description

A.3.1 How to access

All material is publicly available:

- AUTODOCK-GPU source code and scripts are available on GitHub: `https://github.com/ccsb-scripps/AutoDock-GPU`
- AUTODOCK (serial baseline) source code and related utilities are available on its own website: `http://autodock.scripps.edu/downloads`
- Data sets are available on Zenodo: `https://doi.org/10.5281/zenodo.4031961`
- Artifacts (scripts, additional configuration files and output examples) are available on Zenodo: `https://doi.org/10.5281/zenodo.4073350`

A.3.2 Hardware dependencies

In order to obtain comparable results, we recommend the following hardware:

- For CPUs, AWS `c5.18xlarge` (36 physical cores) and `m5.metal` (48 physical cores) instances featuring Intel Xeon Platinum 8124M and 8175M CPUs, respectively. As suggested by AWS for HPC workloads, hyper-threading should be disabled (if possible).
- For GPUs, AMD Vega 64 and Nvidia Titan V GPUs.

However, AUTODOCK-GPU should run on any OpenCL-capable GPU/CPU-based accelerator.

A.3.3 Software dependencies

OpenCL drivers are required for both CPUs and GPUs. Installation instructions are provided by respective GPU/CPU vendors.

- For CPUs, we recommend Intel OpenCL 2.0 (Build 0) with driver Intel OpenCL 18.1.0.0920.
- For GPUs, we have tested using OpenCL 1.2 AMD driver version 1.1, and OpenCL 1.2 CUDA driver 440.82.

Our binary files are for Linux distributions. Ubuntu 18.04 has worked without any problems. Other Linux (and even Unix) distributions should work as well, but have not been extensively tested.

A.4 Installation

A.4.1 AUTODOCK-GPU

- Cloning repository:

```
$ git clone https://github.com/ccsb-scripps/AutoDock-GPU.git
```

- Making sure version 1.2 (commit `eed190fd`) is being used:

```
$ cd AutoDock-GPU/
$ git checkout eed190fd
```

- Compiling (a single binary):

```
$ make DEVICE=<TYPE> NUMWI=<NWI>
```

where: <TYPE> specifies the accelerator chosen (`GPU` or `CPU`), and <NWI> is the number of work-items in a work-group (WG_{size}). By default, binaries are placed under the `bin/` folder. For instance, CPU binaries running the WG_{size} values used in the paper can be all compiled like this:

```
$ for i in 16 32 64 128 256;do make
    ↪ DEVICE=CPU NUMWI=$i;done
```

- Relocating binaries into the main AUTODOCK-GPU folder:

```
$ pwd
/home/user/AutoDock-GPU
$ cp bin/* .
```

Note "." at the end of the `cp` command. This step is done just for compatibility with scripts provided.

A.4.2 AUTODOCK

Installation instructions are available under `http://autodock.scripps.edu/downloads`. Similarly, place the AUTODOCK binary (`autodock4`) under the main AUTODOCK-GPU folder.

A.5 Experiment workflow

As a reference for next steps, consider the following structure for the AUTODOCK-GPU folder:

```
$ pwd
/home/user/AutoDock-GPU
$ tree -L 1
    bin/
    data/
    results/
    results_autodock426/
    dpf_autodock426/
    autodock426.sh
    ia3_exp_numwi_gpus_titanv.sh
    autodock4
    autodock_cpu_16wi
    Makefile
    ...
```

1. Make sure data sets from Zenodo are present. These input files should be available under the `data/` folder.
2. Make sure the folders `results/` and `results_autodock426/` exist. These folders will store the .dlg files produced by AUTODOCK and AUTODOCK-GPU programs, respectively. If such folders are not present, simply create them using the `mkdir` command.
3. To obtain serial baseline runtimes from AUTODOCK, run:

```
$ ./autodock426.sh
```

4. To obtain runtimes for analyzing the impact of OpenCL work-group sizes (see Section IV-B of paper), run:
   For GPUs:

```
$ ./ia3_exp_numwi_gpus_vega64.sh
$ ./ia3_exp_numwi_gpus_titanv.sh
```

   For CPUs:

```
$ ./ia3_exp_numwi_cpus_c518x.sh
$ ./ia3_exp_numwi_cpus_m5.sh
```

5. To obtain runtimes for analyzing the impact of molecular complexity (see Section IV-C of paper), do the following. First, select the 64 work-items version for GPUs, and the 16 work-items version for CPUs. Second, run:
   For GPUs:

```
$ ./ia3_exp_perf_gpus_vega64.sh
$ ./ia3_exp_perf_gpus_titanv.sh
```

   For CPUs:

```
$ ./ia3_exp_perf_cpus_c518x.sh
$ ./ia3_exp_perf_cpus_m5.sh
```

For low-variance results, specially for CPUs, we suggest to perform the experiments on a system with no other compute- or memory-intensive running simultaneously.

A.6 Evaluation and expected result

Results of every execution will be stored under the following folders: `results/` for AUTODOCK-GPU, and `results_autodock426/` for AUTODOCK. These folders will contain the predicted molecular poses by each program execution (i.e., docking job), including the overall execution runtime. These execution runtimes will be used for calculating the speedup factors given in the corresponding figures in the paper (Fig 3 and Fig. 4). A simply way to display these runtimes is to run:

```
$ grep "Program run time" ./results/*
$ grep "Real=" ./results_autodock426/*
```

Output examples (.dlg) are provided under the folder `dlg_examples/` within the artifacts repository.

A.7 Experiment customization

For experiments using AUTODOCK-GPU, scripts are fully customizable and allow changing the docking configuration directly on the commands within the respective scripts. For instance, the following configuration:

- CPU target with $\mathrm{WG}_{\mathrm{size}} = 32$ (work-items)
- Ligand-receptor: `1u4d`
- # LGA runs: $R = 1000$
- Population size: $P = 50$
- $N^{\mathrm{MAX}}_{\text{score-evals}} = 20\,000$
- $N^{\mathrm{MAX}}_{\text{gens}} = 2\,700$
- LS method: ADADELTA with lsrate = 6%
- Name of the output docking log file (.dlg): "output-log"

can be run using this *custom* command:

```
./bin/autodock_cpu_32wi -lfile data/1u4d/
    ↪ rand-0.pdbqt -ffile data/1u4d/
    ↪ protein.maps.fld -nrun 1000 -psize
    ↪ 50 -nev 20000 -ngen 2700 -lsmet ad
    ↪ -lsrat 6.00 -resnam output-log
```

For experiments using AUTODOCK, the configuration of a docking job has to be done by editing directly the *docking parameter file* (.dpf, see examples under the `dpf_autodock426/` folder within the artifacts repository). Configuration options are intuitive and their IDs are similar to those of AUTODOCK-GPU.

A.8 Methodology

The artifact appendix for this paper was submitted according to the guidelines at `https://ctuning.org/ae/submission-20200102.html`