Accelerating AUTODOCK4 with GPUs and Gradient-Based Local Search

Diogo Santos-Martins[1,a], Leonardo Solis-Vasquez[1,b], Andreas Koch[b,✉], and Stefano Forli[a,✉]

[1]These authors contributed equally to this work.
[a]Department of Integrative Structural and Computational Biology, The Scripps Research Institute, La Jolla, California, USA
[b]Embedded Systems and Applications Group, Technische Universität Darmstadt, Germany

AUTODOCK4 is a widely used program for docking small molecules to macromolecular targets. It describes ligand-receptor interactions using a physics-inspired scoring function that has been proven useful in a variety of drug discovery projects. However, compared to more modern and recent software, AUTODOCK4 has longer execution times, limiting its applicability to large scale dockings. To address this problem, we describe an OpenCL implementation of AUTODOCK4, called AUTODOCK-GPU, that leverages the highly parallel architecture of GPU hardware to improve the docking throughput up to 170-fold. Moreover, we introduce the gradient-based local search method ADADELTA, which is more efficient than the original Solis-Wets method, especially for conformationally complex ligands. We estimate a 1300x reduction in the number of scoring function calls for ligands with 20 rotatable bonds, and even higher reductions likely for more complex ligands. The improvements reported here, both in terms of docking throughput and search efficiency, expand the domain of applicability of the AUTODOCK4 scoring function considerably.

Molecular Docking | AutoDock | High Performance Computing | GPU

Correspondence: ***koch@esa.tu-darmstadt.de forli@scripps.edu***

INTRODUCTION

The identification of new molecules with a given biological activity is a non-trivial task, and a key step in the drug discovery process. Structure-based virtual screening in which small organic molecules are docked to biologically and therapeutically relevant targets is a very popular computational technique for the identification of new chemical scaffolds with biological activity.

Over the years, this technique has provided numerous positive results competitive with experimental screenings (1–5). The two main factors contributing to its success is the use of energy estimate methods (or scoring functions) that are sufficiently accurate to distinguish (to a certain extent) between binders and non-binders, while being sufficiently fast to be applicable on a very large scale. Typical virtual screenings now can easily involve virtual libraries of compounds with hundred thousand to several million molecules against relatively rigid target structures. Depending on the search algorithms, this translates in 10^{10} to 10^{16} calls to the scoring function. The increasing availability of targets from structural genomics initiatives (6), as well as the use of target conformational ensembles (7) to address receptor conformational variability, can easily add three or more orders of magnitude.

Molecular docking is particularly suitable for parallel computing, given the embarrassingly parallel nature of the tasks involved. Scoring function and search calls can be easily distributed across multiple computing units, either being CPU cores on the same machine (8), or many CPU nodes across multiple machines in high-performance computing (HPC) environments (9), or distributed computing resources (10, 11). From that perspective, GPU computing represents a hybrid scenario, in which a large number of computing units (i.e., GPU cores) can be accessed on a single machine. The availability of GPU cards with thousands cores represents a cheap but powerful computational resource that has already been exploited to accelerate a number of molecular modeling computations (12–14).

AUTODOCK is a widely used molecular docking program. From a computational standpoint, it is characterized by its single-threaded behavior, which means it only utilizes a single CPU core during execution. By exploiting the underlying algorithms' embarrassingly parallel nature, its efficient adaption to accelerators can result in significant performance improvements. Expressing such parallelism in a suitable manner can be achieved through parallel programming frameworks (15–17). One of these, the Open Computing Language – OpenCL – provides an open standard that is portable across hybrid platforms such as CPUs and GPUs. From an algorithmic standpoint, and based on the potential gains from an OpenCL implementation, more complex search methods can be explored without incurring the big performance penalties that would affect the single-threaded AUTODOCK version.

In this work, we describe our OpenCL implementation of AUTODOCK for GPUs. This program is based on an improved version of the AUTODOCK search algorithm, which introduces scoring-function gradients for translation, rotation, and torsion variables. For global search, it still employs a Lamarckian Genetic Algorithm (LGA) similar to the one used in AUTODOCK (since version 4.0) (18). For the local search, we newly incorporate the ADADELTA (19) gradient-based method, while also including the original random numerical optimizer Solis-Wets (20). Through a number of experiments, we evaluated the computational and algorithmic enhancements provided by our OpenCL implementation over the original AUTODOCK.

METHODOLOGY

Brief introduction to AUTODOCK docking. In AUTODOCK, molecular interactions are modeled by a scoring function f that quantifies the free energy of a given binding pose (21):

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

The AUTODOCK scoring function (Equation 1) is a semi-empirical free-energy force field (kcal/mol) composed of four pairwise energetic terms (dispersion/repulsion, hydrogen bonding, electrostatics, and desolvation), and an additional term that predicts the (unfavorable) loss of ligand entropy upon binding. The dimensionless weighting constants W_{vdw}, W_{hb}, W_{el}, W_{ds}, and W_{rot} were empirically determined using linear regression on a set of ligand-receptor complexes with known binding constants. The following constants depend on the atom types: A_{ij} (kcal/mol Å^{12}) and B_{ij} (kcal/mol Å^{6}) correspond to the Lennard-Jones (12-6) potential between atoms i and j; C_{ij} (kcal/mol Å^{12}) and D_{ij} (kcal/mol Å^{10}) correspond to the hydrogen bonding (12-10) potential between hydrogen-bond acceptor and donor atoms i and j; S and V are respectively the solvation parameter and the atom volume that shelters it from the solvent, while σ = 3.5 Å is an independent constant. The $E(t)$ function represents the dimensionless directional weight of the angle t that provides directionality from ideal hydrogen bonding geometry. Additionally, q_i and q_j are atomic charges, while $\epsilon(r_{ij})$ is a dielectric function of r_{ij}, the interatomic distance between atoms i and j. The last term, proportional to the number of torsions N_{rot}, measures the unfavorable loss of ligand entropy due to the restriction of conformational degrees of freedom in the bound state.

The overall molecular interaction can be expressed as the sum of two independent interactions based on ligand and receptor group of atoms. Intermolecular interactions (i.e., ligand-receptor) are precalculated and cached using the grid-based method of the AUTOGRID program, which calculates interaction energy maps for all ligand atoms, with a default resolution of 0.375 Å. These maps are used to lookup interaction energies using trilinear interpolation, speeding up intermolecular interaction estimates compared to pairwise methods. Similarly, in the AUTODOCK4 implementation, intramolecular interactions (i.e., ligand-ligand) are precalculated for all atom pairs and stored in one-dimensional look-up tables.

A Lamarckian Genetic Algorithm (LGA) is used to generate ligand poses exploring the energy landscape described by the scoring function (22). The LGA combines a global search – based on a genetic algorithm (GA) – with local search (LS) iterations to refine poses identified by the GA. Poses that are improved by the LS method are re-introduced into the GA population (hence the Lamarckian denomination). All calculated poses form a population, where each member, i.e., an individual, is represented by a vector of genes (i.e., genotype). Each gene represents a specific ligand motion, controlling the conformation and orientation of the ligand. New individuals are generated by the GA from ancestors chosen with a proportional selection scheme. A population subset (default: 6%) is subjected to the LS optimization, which in AUTODOCK4 is based on the Solis-Wets method (20). The LGA execution stops when a pre-defined number of score evaluations or GA generations is reached, whichever comes first.

Furthermore, a docking job consists of several independent LGA runs (default: 50). Each LGA run optimizes the sum of intermolecular (ligand-receptor) and intramolecular (ligand-ligand) interactions described by the scoring function f, and returns the best pose (lowest score) it could find. After all LGA runs are complete, the returned poses are clustered using the root mean square deviation (RMSD) as distance metric. The size of the cluster containing the pose with the best score can be used as a proxy to identify convergence of the LGA runs towards a particular solution (pose) within the search space. As a general rule, the search effort is considered sufficient if the first cluster contains about 20% (or more) of poses returned by LGA runs.

OpenCL implementation of AUTODOCK. As both GA and LS optimizations require many scoring function calls for a single docking job (default: in the order of million evaluations), these optimizations become the bottleneck accounting together for more than 90% of the entire application runtime (23). Therefore, offloading these optimization methods onto an accelerator, like a GPU, can profit significantly from the speedups enabled by more specialized hardware.

The improvements of the AUTODOCK search algorithm introduced in this paper were built on top of our previous work (23), released as the open-source software project *OpenCL Accelerated Molecular Docking*, hereafter simply referred to as AUTODOCK-GPU (24). In this section, we describe various software-related aspects of our implementation, starting with OpenCL basics, going through our parallelization approach, as well as providing insights of the main design considerations made in this work.

OpenCL background. Programming frameworks are essential to leverage the performance capabilities of highly-parallel computing platforms (15–17). In that regard, OpenCL provides an open, and royalty-free standard for writing parallel programs that execute across heterogeneous platforms consisting of CPUs, GPUs, and other hardware accelerators, providing software developers with a portable abstraction for accelerating computationally-intensive tasks (25).

A brief introduction to the OpenCL programing model is presented as follows. An OpenCL *platform* is a vendor-specific implementation of the standard, and defines the logical representation of the hardware capable of executing an OpenCL application. Basically, an OpenCL platform is composed of two main elements, i.e., a *host* processor, and a number of computing *devices*, both being elements connected through a bus. The host is responsible for the general platform management, while the device is in charge of executing clearly-defined functions in parallel, which are called *kernels*. These kernels are described using OpenCL C, which is a C-like language with extensions for parallel programming such as multithreading and synchronization mechanisms. A kernel can be described as a function of indices (Figure 1). Each index-based kernel element is called a *work-item* (`wi`), which can be thought of as a processing thread. Moreover, the total number of indices is partitioned into *work-groups* (`WG`). Communication between individual work-items within a work-group is possible and can be achieved through global or locally-shared memory, while their synchronization can be enforced through barrier operations.

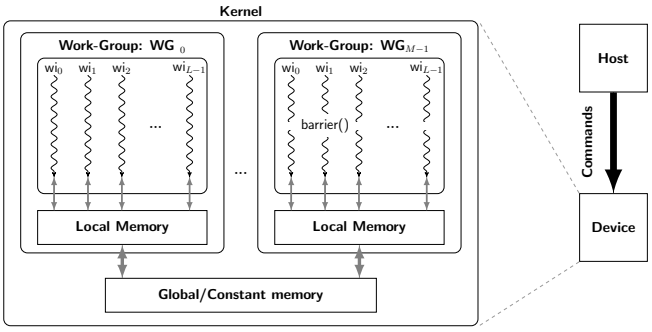

Fig. 1. An OpenCL platform is composed of a host and one or more devices. The host controls the device(s) via commands. A device is in charge of executing kernels, each of which can be subdivided into M work-groups (`WG`s). Each work-group can be further subdivided into L work-items (`wi`s). Within a work-group, communication and synchronization between work-items is possible using local memory and barrier operations, respectively.

Parallelization opportunities and OpenCL strategy. As reported in a previous study (26), data parallelism of AUTODOCK is exploited at three different processing levels: *high*, *medium*, and *low*. First, based on the fact that a docking job consists of several independent LGA runs (default: 50), the high-level parallelization consists of trivially performing these runs in parallel. Second, individuals from a single genetic generation within an LGA run are processed independently, achieving medium-level parallelization. Finally, the low-level strategy consists of the parallel execution of fine-grained tasks that pertain to a single individual, such as rotating the ligand or evaluating the scoring function.

Our implementation (Figure 2) combines the high- and medium-level strategies first. A docking job is composed of R LGA runs (Run_{ID}: 0, 1, 2, ..., R - 1), each run processing a population of P individuals (Ind_{ID}: 0, 1, 2, ..., P - 1). The execution of such runs, and the processing of their individuals, are controlled by nested loops in the serial implementation, but can be merged into a single loop for optimal parallelism. By doing so, individuals from different docking runs can be processed simultaneously, each as an OpenCL work-group identified with a WG_{ID} obtained as follows:

$$\text{WG}_{\text{ID}} = \text{Run}_{\text{ID}} * P + \text{Ind}_{\text{ID}} \quad \textbf{(2)}$$

The entire set of $P * R$ work-groups is distributed by the GPU runtime scheduler over the available Q compute units (CU). Each CU executes a `WG` associated with a single individual, achieving high- and medium-level parallelization simultaneously. Finally, processing an individual involves fine-grain tasks (genotype generation, ligand rotation, intermolecular and pairwise interactions) that can be assigned to OpenCL work-items, thus achieving low-level parallelization.

Code architecture. The overall workflow of AUTODOCK-GPU is depicted in Figure 3. This consists of a sequence of functions executed either on the host (`H`) or on the device (`D`). After the application inputs are parsed in `H1`, the populations of *all* docking runs are initialized with random values in `H2`. Then, the OpenCL setup takes place in `H3`, which comprises the identification and selection of platform and device, as well as the definition of other OpenCL objects such as *context* and *command queues*. This process includes the creation of a *program* object containing all machine-specific instructions to be executed on the device. Since the host is responsible for launching and keeping track of kernel executions, it needs to know how to interact with the kernels. Therefore, in `H4`, OpenCL kernels (`INIT`, `EVAL`, `GA`, `LS`) are specified in terms of arguments (variables holding e.g., initial population values, number of genes and ligand atoms, etc.), global size (total number of work-items to be processed by the device), as well as local size (number of work-items to be processed in each work-group).

The first kernel executed is `INIT`, which calculates the initial score of individuals from all docking runs. The second kernel, `EVAL`, counts the number of scoring function calls (whose resulting scores reside in device memory) performed by previously-executed kernels. After `EVAL` executed, control is handed back to the host, which checks whether the LGA termination criteria are met, i.e., if the number of either score evaluations or generations reached their maximum values (default: $2.5 * 10^6$ and $27 * 10^3$, respectively). The core of the application is the iterative execution of kernels, with each cycle starting with `GA`, then going through `LS`, and finishing with `EVAL`. In particular, the `LS` kernel executes either the Solis-Wets method, as in the original AUTODOCK, or the ADADELTA gradient-based method newly added in this work. While the inter-kernel synchronization is controlled entirely on the host via *in-order* command queues, the transfer of solution data and their scores between kernels occurs by device-side global-memory accesses. Finally, when the LGA termination criteria are met, the final solutions found by the device (and residing in its global memory) are copied back to the host, where results are written to output *.dlg* files compatible with AUTODOCKTOOLS (18).

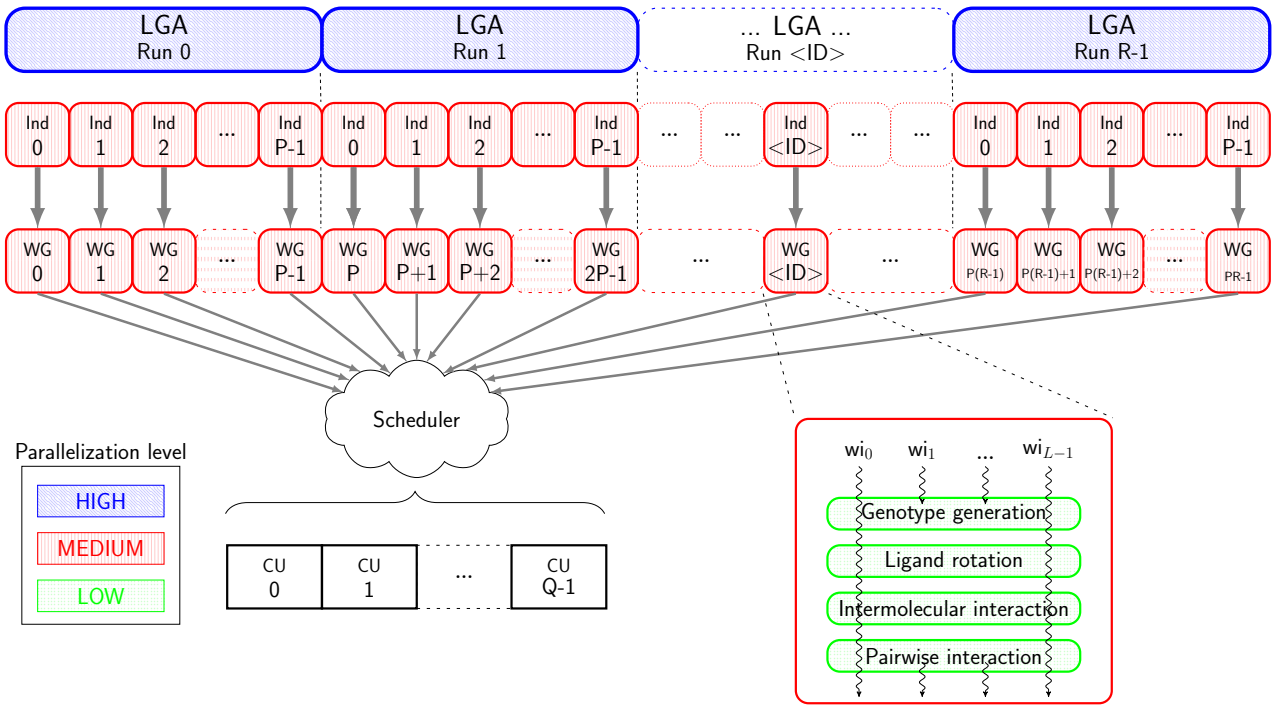

Fig. 2. A population processed by a LGA run (Run_{ID}) can be decomposed into their individuals, and each individual (Ind_{ID}) can be mapped onto a work-group (WG_{ID}). The entire set of work-groups is distributed by the GPU runtime scheduler over the available Q compute units (CUs). A CU is a multi-threaded hardware unit capable of processing one work-group (composed of L work-items) at a time. The runs, individuals, and fine-grain tasks are colored according to their associated level of parallelism: high (blue), medium (red), and low (green).

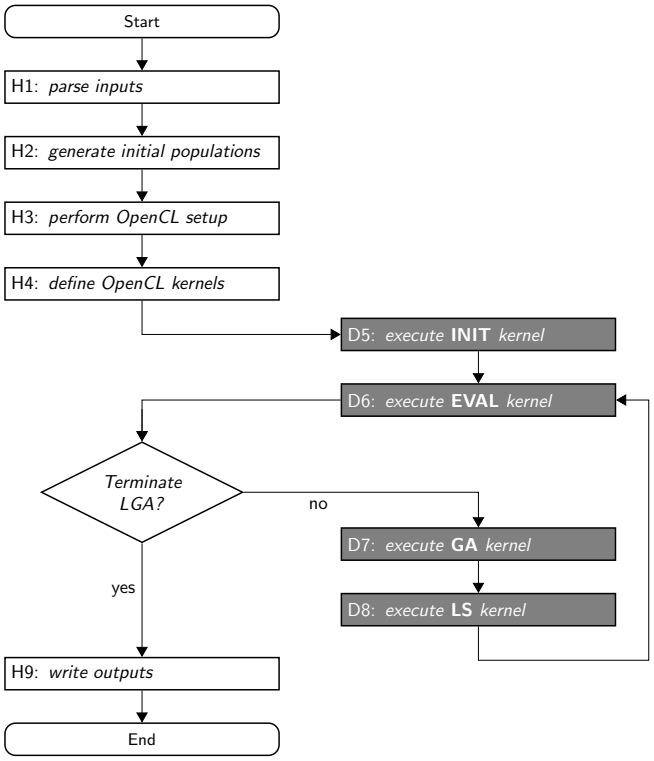

Fig. 3. The overall AUTODOCK-GPU workflow consists of a sequence of host (H) and device (D) functions. Program execution always starts and finishes in host functions (depicted at the left side). Time-consuming functions, i.e., kernels, are executed iteratively on the device (depicted at the right side), while their termination is controlled by the host.

Memory requirements. In order to guarantee that the limited memory capacity (compared to most multi-core CPU servers) of a GPU card does not negatively impact the practical usage of AUTODOCK-GPU, an analysis of the memory size required to hold the processing data is performed. As shown in Table 5, we defined upper limits on AUTODOCK-GPU's docking parameters, i.e., the maximum number of elements for the following data: ligand atomic types, ligand atoms, rotatable bonds, pairwise contributors, rotations, population size, docking runs, and grid points. Although by doing this we *might* constrain the capabilities of AUTODOCK-GPU, these limits prevent data allocation beyond the typical memory capacity (few GBs) of most consumer GPU cards.

The first concern is the memory occupied by constant data, which is composed of relatively large look-up tables used in different docking calculations. Table 6 lists all constant arrays utilized when AUTODOCK-GPU is configured to run Solis-Wets. These are conveniently grouped into structs (A, B, C, D, E), and passed into GPU memory as OpenCL buffer objects. Depending on the assigned OpenCL memory-space qualifier, a struct can be placed either in the GPU on-board memory (`__global const`), or in the GPU on-chip memory (`__constant`). Ideally, one would place everything on-chip for faster access. However, due to the on-chip capacity limits (in the range of few MB), this is not always possible and consequently, on-board memory must be used as well. The ADADELTA gradient-based search method requires additional space in memory, which is attributed to the structs (F, G) listed in Table 7. Moreover, grid maps can occupy a large memory region as their size depends exponentially on

the number of grid points (cubic growth). A maximum limit of 256 grid points would allow users to analyze reasonably large binding regions while keeping the memory space below 1.1 GB. Larger values would require excessive space that cannot be allocated on typical GPU-card memories: e.g., 512 grid points would require more than 8 GB. Then, with the current configuration, the maximum memory space required to store constant arrays is 252 kB (A, ..., E) + 45 kB (F, ..., G) + 1,073 MB (GRIDS) = 1,074 MB, which is possible to be stored on-chip.

Another concern is the storage for variable data, i.e., the information being updated during the entire docking procedure. This data consists of both current and next populations, as well as the scores of their component individuals. As all docking runs are processed in parallel, the maximum memory size required to store current populations (P_{maxsize}), and individual scores (E_{maxsize}), both expressed in bytes, can be calculated as follows:

$$P_{\text{maxsize}} = R * P * L_{\text{genotype}} * S_{\text{float}} \quad \textbf{(3)}$$

$$E_{\text{maxsize}} = R * P * S_{\text{float}} \quad \textbf{(4)}$$

where R and P are respectively the number of docking runs and the population size (both specified by the user), L_{genotype} is the constant genotype length (= 64) in global memory, and S_{float} is a size of a `float` or single precision floating-point datatype (4 bytes). For R and P in Equations 3 and 4, we have defined upper limits, too. This means that AUTODOCK-GPU accepts for R and P any integer value $\leq$ 1,000 and $\leq$ 2,048, respectively. If the user inputs either an invalid or an out-of-range value, then AUTODOCK-GPU outputs a warning message, and then proceeds with execution using default values of R = 1 and P = 150. As corner cases, we would have P_{maxsize} = 1,000 $*$ 2,048 $*$ 64 $*$ 4 = 524.28 MB, and E_{maxsize} = 1,000 $*$ 2,048 $*$ 4 = 8.19 MB, which together account for 532.48 MB. Considering both – current and next – populations and their scores, their total memory footprint together is 1064.96 MB.

Summing up both maximum possible sizes of constant and variable data, the rounded-up memory space required by AUTODOCK-GPU is less than 2.2 GB, which is lower than the amount typically available on consumer GPU cards, as exemplified in Table 3. Thus, there is no need for compute-specialized GPUs with larger memories, which are significantly more expensive.

***Main differences compared to* AUTODOCK.** The AUTODOCK-GPU implementation involves modifications to the original AUTODOCK functionality in order to better exploit parallel processing, and the execution performance, without negatively affecting the docking quality. These modifications include the following:

Arithmetic precision. Scoring and search calculations in AUTODOCK are performed using double-precision floating point (64 bits). As previous studies (23, 26, 27) suggest that performing docking computations with reasonably lower precision does not lead to deterioration in terms of best score and clustering size, we opted to implement those calculations in single-precision floating point (32 bits).

Arrangement of data structures. Data structures were rearranged for better parallel processing of rotation and pairwise interaction. Ligand flexibility can be described by two rotation types. First, a general one that considers the ligand as a rigid body, and a second type due to rotatable bonds for which a tree-like structure is constructed. AUTODOCK serially traverses the nodes of such flexibility tree in a recursive manner. Although doing so is feasible on OpenCL devices capable of enqueuing kernels independently from the host (a feature known as *device-side enqueuing*), this would not be portable to devices with more limited language support, i.e., prior to OpenCL 2.0 (28). To tackle this, the recursion is translated into an iterative approach, which is achieved by transforming the flexibility tree into an array-like rotation list. This list is composed of *integer*-type items (32 bits) with fields detailed in Table 1. Similarly, for the pairwise interaction, instead of having a GPU (likely inefficiently) traversing the tree, the host defines another array-like list, containing *intramolecular-contributing* atomic pairs.

Selection scheme. Regarding the criterion to choose which individuals will reproduce, the original *proportional selection* was replaced with *binary tournament* (default rate: 60%). In proportional selection, individuals with better-than-average scores receive proportionally more offspring (22). One of its major defficiencies is that if the initial population contains one or two energetically-stronger individuals, then these would dominate the rest, and consequently, would prevent the population from exploring other potentially better solutions by escaping from a local optimum (29, 30). On the other hand, in tournament selection, sets of individuals are randomly selected from the entire population. The highest-scoring individual in the set is the tournament winner, and therefore selected for crossover. This scheme also suffers from diversity loss, which happens with large set sizes. Our implementation minimizes this possibility as the minimal tournament set size is chosen (i.e., two, hence the binary denomination). Moreover, the major advantage of tournament selection is the low computational effort, especially if implemented in parallel (30), which according the previous studies (26, 27) results in faster executions than those of proportional selection.

Specification of program arguments. AUTODOCK arguments are specified using a docking parameters file (*.dpf*) containing parameters to control various aspects of a docking job. Here, we replaced the *.dpf* file with command-line program arguments, making the program more suitable for scripting, which is useful for highly iterative tasks such as virtual screening.

Local search methods. In addition to Solis-Wets, the local search method in the original AUTODOCK4, this work additionally newly implements the ADADELTA optimizer

Table 1. Bit-field description of a 32-bit rotation-list item.

Bits	Description
7 - 0	ID of atom to be rotated ($ATOM_{ID}$)
15 - 8	ID of rotatable bond ($ROTBOND_{ID}$) around which an atom with $ATOM_{ID}$ is to be rotated
16	1: if first rotation of atom with $ATOM_{ID}$, 0: otherwise
17	1: if general rotation, then $ROTBOND_{ID}$ is ignored, 0: otherwise
18	1: if dummy rotation, then no rotation, 0: otherwise
31 - 19	Unused

based on *gradients* of the scoring function.

Gradient of the scoring function. The process of docking a ligand is implemented as an optimization problem where different types of ligand motion correspond to variables to be optimized. The first three variables control translation of the ligand in x, y, and z directions; the next three variables – ϕ, θ, and α – correspond to rotation of the ligand as a rigid body; and the remaining ψ variables are associated with N_{rot} rotatable bonds. These variables are also referred to as *genes*. Collectively, they constitute a *genotype*, denoted by Omega:

$$\Omega = x, y, z, \phi, \theta, \alpha, \psi_1, ..., \psi_{N_{\text{rot}}} \quad \textbf{(5)}$$

Each genotype corresponds to a different pose of the ligand. The quality of a pose, from the biophysical standpoint, is evaluated by a scoring function f:

$$f(x, y, z, \phi, \theta, \alpha, \psi_1, ..., \psi_{N_{\text{rot}}}) \quad \textbf{(6)}$$

Here, f is the same scoring function as in Equation 1, but expressed in terms of the optimization variables (genes) instead of interatomic distances and atomic parameters. Docking a ligand consists of finding the genotype Ω that corresponds to the lowest possible value of the scoring function f. Gradient-based optimizers use the gradient g of the scoring function f, with respect to the genotype Ω:

$$g = \nabla f(\Omega) = \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}, \frac{\partial f}{\partial \phi}, \frac{\partial f}{\partial \theta}, \frac{\partial f}{\partial \alpha}, \frac{\partial f}{\partial \psi_1}, ..., \frac{\partial f}{\partial \psi_{N_{\text{rot}}}}, \quad \textbf{(7)}$$

ADADELTA local search. ADADELTA (19) is a gradient-based optimization algorithm, by which the genotype Ω is updated at each iteration t:

$$\Omega_{t+1} = \Omega_t + \Delta\Omega_t \quad \textbf{(8)}$$

where $\Delta\Omega_t$ is the update vector, which depends not only on the gradient g, but also on the history of past gradients and past update vectors:

$$\Delta\Omega_t = -\frac{\sqrt{E[\Delta\Omega^2]_{t-1} + \epsilon}}{\sqrt{E[g^2]_t + \epsilon}} g_t \quad \textbf{(9)}$$

where $E[\Delta\Omega^2]$ is a running average of squared updates, and $E[g^2]$ is a running average of squared gradients:

$$E[\Delta\Omega^2]_t = \rho E[\Delta\Omega^2]_{t-1} + (1-\rho)\Delta\Omega_t^2 \quad \textbf{(10)}$$

$$E[g^2]_t = \rho E[g^2]_{t-1} + (1-\rho) g_t^2 \quad \textbf{(11)}$$

where ρ and ϵ are ADADELTA's hyperparameters.

In particular, the constant ϵ prevents the denominator in Equation 9 from becoming zero if $E[g^2]$ is zero. Furthermore, ϵ is required to produce non-zero updates in the first iteration (t = 0), because the running average of squared updates of the preceeding iteration $E[\Delta\Omega^2]_{-1}$ is assumed to be zero. The running averages $E[g^2]$ and $E[\Delta\Omega^2]$ are vectors of length equal to the number of *genetic* variables. The update rule is applied to each variable separately, implementing a different learning rate for each dimension. The hyperparameter ρ controls the amount of *memory* of preceeding iterations for $E[g^2]$ and $E[\Delta\Omega^2]$. Smaller ρ make the running averages more sensitive to the current iteration.

ADADELTA is one of several first-order local optimization algorithms. Ideally, we would have implemented a Quasi-Newton method, such as BFGS (31), but we opted for a first-order method because it is simpler to be integrated in our highly-parallel OpenCL architecture. Examples of other popular first-order methods are Adam (32), and FIRE (33). All these algorithms have hyperparameters that influence the calculation of the update vector. We chose ADADELTA because it has only two hyperparameters, ρ and ϵ, making it simpler to optimize the algorithm for docking, while Adam and FIRE have four and five hyperparameters, respectively. After evaluating ADADELTA using various combinations of ρ and ϵ we set ρ = 0.8 and ϵ = 0.01, because these values provided the best search performance (see Figure 6 and associated discussion).

Dataset. A dataset of 140 protein-ligand complexes was used to compare our OpenCL implementation to AUTODOCK4. Our dataset is mainly sourced from two well-established sets for assessing docking methodologies for drug discovery purposes: the Astex diversity set (34) and CASF-2013 (35). We included all 85 entries from the Astex diversity set, in which most ligands have up to 10 rotatable bonds. With regard to the CASF-2013 set, we included 35 complexes, enriching our set with ligands in the range of 11 - 20 rotatable bonds. Furthermore, we included 20 additional complexes selected from the Protein Data Bank (36) covering a large range of rotatable bonds (up to 32) and displaying both wide and narrow binding pockets.

Identification of global minima. We identified global minima for the complexes in our dataset by analyzing the distribution of scores produced by LGA runs (Figure 4). Increasing the number of evaluations shifts the distribution of scores towards more negative values (i.e., better scores). For most complexes, we observed convergence of the distributions towards a lower bound: once that lower bound is reached, increasing the number of evaluations does not change the lower bound, but increases the number of LGA runs that actually found the lower bound score. The upper panel of Figure 4 shows an example of convergence towards a lower bound. Based on this observation, we devised a criterion to systematically identify global optima, which is explained as follows: for a protein-ligand complex, we consider the scores returned

by LGA runs with up to 8,192,000 evaluations. A total of 4 docking replicates are performed, where each replicate starts with a different conformation and orientation of the input ligand. From each replicate, we retrieve the 5 top scores (out of 100 LGA runs), creating a subset of 20 scores. This procedure is repeated for 4 docking replicates with 100 LGA runs each, executed up to 4,096,000 evaluations. The best score from each replicate is added to the subset of scores, increasing its size from 20 to 24. If the difference between the best and worst scores within the subset (of 24 scores) is less or equal to 0.1 kcal/mol, the best score is considered to be the global minimum. While there is no theoretical guarantee that an even lower minimum does not exist, we did not observe a single case in which the distribution of scores initially converged to an intermediate value, and then escaped to an even lower score after further evaluations. This suggests that the distribution of scores is not easily trapped in local minima. Moreover, varying local-search rate and local-search method results in convergence to *identical* global minima: the largest difference in global minima found from different dockings was just 0.23 kcal/mol.

Technical details. All calculations reported in this work employ a maximum of 300 local-search iterations. The maximum number of LGA generations is set to 99,999 to guarantee that docking does not stop before the maximum number of score evaluations is reached.

Ligand and receptor input files were prepared with OpenBabel-2.4.1 (37), after removing all water molecules and metal atoms for which AUTODOCK4 misses atom types.

Regarding software support, the latest AUTODOCK4.2.6 was used as a single-threaded CPU baseline. For GPUs, OpenCL drivers provided within the CUDA 10.0 and AMDAPPSDK 3.0 packages were used. For evaluating the portability of OpenCL to multicore CPU machines, we employed the Intel SDK-2017 driver.

Measuring the evaluation rate. Docking a ligand involves a large number of score evaluations, often in the order of millions. The time spent per evaluation is one of the factors determining docking runtime. To calculate the evaluation rate, we run dockings using a varying number of evaluations, and fit a linear model defining the docking runtime y as a function of the number of evaluations x:

$$y = ax + b \tag{12}$$

where b is the intercept with the y-axis and a is the slope.

The slope a is the rate at which score evaluations are performed, which we chose to express in microseconds (μs) per evaluation. Dockings were performed from 32,000 to 2,048,000 evaluations, in exponential increments of 2-fold, providing 7 datapoints to fit the linear model from which the evaluation rate is determined. Four replicates were performed for each docking, and the minimum runtime was considered. Identical results would be obtained if the average runtime were considered instead of the minimum (evaluation rates increased on average by only 1%). Evaluation rates for which the coefficient of determination R^2 is less than 0.99 were excluded. In the *Results and Discussion* section, we report the evaluation rate for all 140 complexes in our dataset, running AUTODOCK-GPU on different accelerators, and using either Solis-Wets or ADADELTA as local-search methods.

RESULTS

Validating the scoring function implementation. Docked poses from AUTODOCK-GPU *.dlg* output files were re-scored using AUTODOCK4.2.6 (keyword *epdb*). A total of 1015 ligand poses from 10 complexes were considered in this analysis. We observe that for docked poses in the favorable range (no clashes), the differences are in the order of 0.01 kcal/mol for grid-interpolated energies, and 0.02 kcal/mol for ligand-ligand (pairwise) interactions. In the case of unfavorable energies, such as when clashes occur, larger differences were observed. These differences are irrelevant for drug-design applications, because the interest is in the most favorable poses of each ligand, whereas high-energy (unfavorable) poses are discarded.

Score differences between AUTODOCK4 and AUTODOCK-GPU are mainly attributed to the following two reasons. First, scoring and search computations in AUTODOCK-GPU are performed with single-precision arithmetic, which might cause a loss of precision in comparison to double-precision floats used in AUTODOCK4. Second, for quantifying ligand-ligand interactions, AUTODOCK4 interpolates values from look-up tables, while AUTODOCK-GPU evaluates the analytical form of the scoring function f (Equation 1). This implementation discrepancy can cause larger score differences – in the order of 1 or more kcal/mol – because some of the scoring terms have steep energy profiles for short interatomic distances. Despite the fact that score differences between AUTODOCK4 and AUTODOCK-GPU are insignificant for well optimized ligand poses without steric clashes, users should be aware of significant discrepancies in high-energy poses arising primarily from the interpolation of ligand-ligand terms in AUTODOCK4. In this regard, AUTODOCK-GPU is a more accurate implementation of the scoring function.

Algorithmic performance. In the field of global optimization, computational effort is often expressed as the number of fitness (or score) evaluations (38). More efficient algorithms spend fewer evaluations to achieve a certain level of performance.

Despite that the number of score evaluations may not correlate with runtime, using it for assessing algorithms, such as Solis-Wets and ADADELTA, is advantageous. The reason is that, in contrast to runtime, the number of score evaluations allows the assessment of algorithmic performance independently from code optimization, choice of compiler, and hardware specifications.

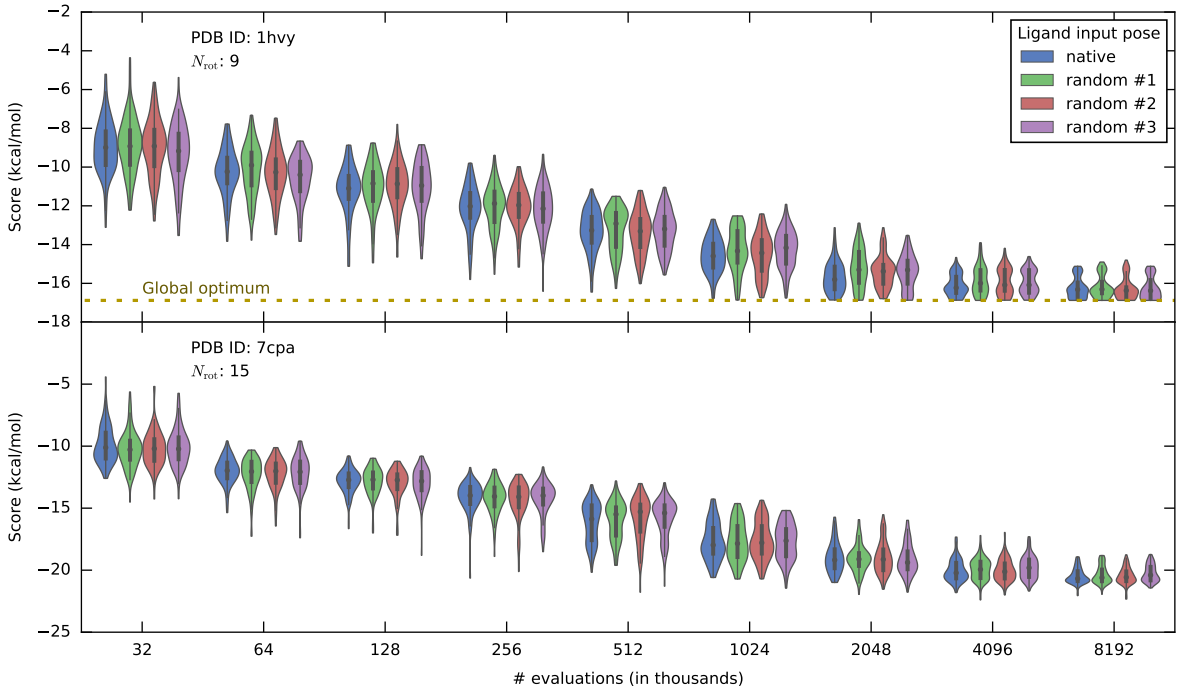

Fig. 4. Distribution of scores returned by LGA runs with increasing number of evaluations. Each violin plot represents scores from 100 LGA runs, and the color is associated with the input conformation and orientation of the ligand. The global minimum was identified for the protein-ligand complex represented in the upper panel (PDB ID: 1hvy), but not for the complex in the bottom panel (PDB ID: 7cpa) because the distribution of scores did not converge towards a lower bound. The local-search method used is ADADELTA, and the local-search rate is 100%.

Success rate of LGA runs. Our assessment of search performance starts by determining if individual LGA runs are successful or not. We use two criteria to define success, one based on the docking score, and another based on the root mean square deviation (RMSD) from the native pose determined by X-ray crystallography.

According to the score criterion, an LGA run is successful if the returned pose has a score within 1.0 kcal/mol from the best possible score for a given protein-ligand complex, i.e., the global minimum. The protocol for finding global minima in our dataset is described in the *Methodology* section. We found global minima for 105 out of 140 complexes, while for the rest that was not possible due to the presence of a large number of rotatable bonds, particularly for ligands containing more than 19 of these.

According to the RMSD criterion, an LGA run is successful if the returned pose is within 2 Å from the native pose. We note that the job of search algorithms is to find the global minimum of the scoring function, and there is no guarantee that the global minimum corresponds to a pose that is within 2 Å RMSD from the native pose. This happens because scoring functions are not perfect, and sometimes give better scores to incorrect poses. Since our goal is to evaluate search performance, we need to guarantee that the optimization target of the search algorithm (i.e., the global minimum of the scoring function) corresponds to a solution that is deemed correct by the success criterion. Therefore, we only use the RMSD criterion when the global minimum of the scoring function is below 2 Å RMSD of the actual physical pose (78 out 140 complexes in our dataset).

Since the LGA algorithm is stochastic, we employ a statistical approach involving a relatively large number of LGA runs. Specifically, we perform four replicates of 100 LGA runs, each replicate starting with the ligand in a different orientation and conformation: one of the four inputs is the native pose, whereas the other three are a randomized pose each (AUTODOCK VINA (8) was used for randomization). Using this data, we calculate the fraction of successful LGA runs.

The fraction of successful LGA runs increases with the number of evaluations, and its corresponding success-rate curve typically follows a sigmoidal profile (Figure 5). An insufficient number of evaluations results in a low (possibly zero) probability of finding either the native pose or the global minimum. After a certain number of evaluations, the success probability asymptotically approaches 100%. Increasing the number of evaluations beyond the higher inflexion point provides only relatively small increases ($\sim$10%) in the probability of success.

Halfway number of evaluations (E_{50}). From success probability curves (Figure 5), we estimate the number of evaluations required to achieve a 50% probability of success. We refer to this number as *halfway number of evaluations*, and abbreviate it as E_{50}.

The values of E_{50} depend on both the protein-ligand complexity, as well as the algorithmic efficiency of search methods. If the global minimum (or native pose) of a given protein-ligand complex is easy to find, high-success proba-

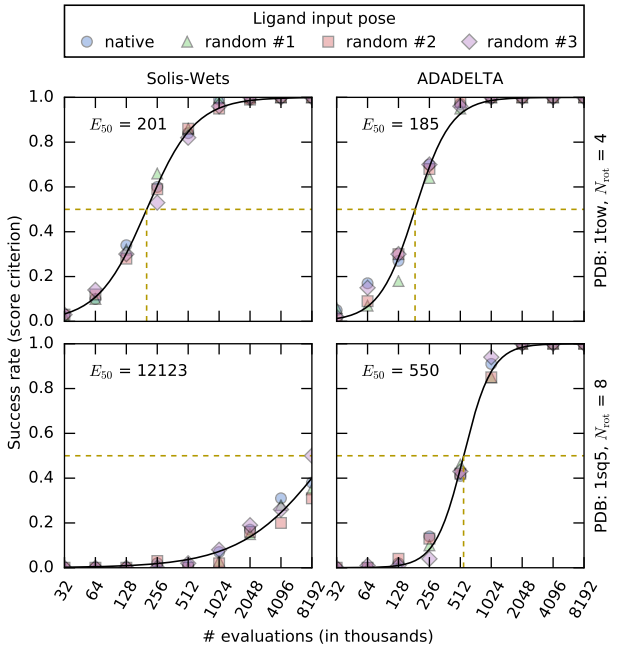

Fig. 5. Fraction of successful LGA runs as a function of the number of score evaluations, using Solis-Wets and ADADELTA as local-search methods. The upper plots correspond to an easy search problem with only four rotatable bonds (PDB ID: 1tow), while the bottom plots correspond to a moderately difficult ligand with eight rotatable bonds (PDB ID: 1sq5). According to the score criterion, an LGA run is successful if it reports a pose within 1.0 kcal/mol from the global optimum. The black line is a fitted sigmoid curve (Equation 13) that estimates E_{50}, which is the number of evaluations at which 50% of LGA runs are successful. The dashed orange lines are a visual representation of E_{50} values.

bilites are achieved with a relatively small number of evaluations. Analogously, lower E_{50} numbers indicate more efficient search algorithms, because the halfway performance is achieved with less computational effort, i.e., fewer score evaluations. This is illustrated in Figure 5, where the top and bottom panels correspond to an easy and (moderately) difficult case, respectively. For example, considering the success-rate curves for the moderately difficult complex (PDB ID: 1sq5) in Figure 5, it can be inferred that ADADELTA outperforms Solis-Wets because it requires ∼20x fewer evaluations to achieve 50% of success rate. In fact, for this complex, using Solis-Wets – with 8 million evaluations per LGA run – results in success rates between 30% and 50%, while ADADELTA approaches 100% of success with only 2 million evaluations.

In order to calculate E_{50} values, we fit the following sigmoid function to the success-rate curves depicted in Figure 5:

$$\text{Fraction of successful LGA runs} = \frac{1}{1+e^{-\beta(x-E_{50})}} \quad \textbf{(13)}$$

where x is the number of score evaluations, while β and E_{50} are variable parameters optimized during sigmoid fitting.

Since E_{50} values describe search performance by a single number, we use it to assess search performance on a large number of protein-ligand complexes, by plotting E_{50} values as a function of the number of ligand rotatable bonds N_{rot}. Search difficulty increases with N_{rot}, because each rotatable bond adds one dimension to the optimization problem. In the following sections, we use E_{50} values to guide the optimization of ADADELTA hyperparameters, and to demonstrate the superiority of ADADELTA over Solis-Wets for large N_{rot}.

Tuning ADADELTA hyperparameters. As previously mentioned, ADADELTA has two hyperparameters: ρ and ϵ. In an initial stage of this work, we manually tuned these hyperparameters to yield low scores in as few evaluations as possible, using a small number of protein-ligand complexes, and visualizing the progression of scores over a few hundred iterations. Eventually, we settled on $\rho = 0.8$, and $\epsilon = 0.01$, but noted afterwards that such ρ was smaller than the smallest ρ (= 0.9) tested in the original ADADELTA publication (19).

To validate our initial choice of hyperparameters, we used a subset of 19 protein-ligand complexes from our dataset (for which global minima were identified), and tested six combinations of ρ (0.8, 0.9, 0.95), and ϵ (0.01, 0.0001) values. Larger values for ϵ result in larger initial steps, because in the first ADADELTA iteration ($t = 0$) the running average of squared updates of the preceeding iteration $E[\Delta\Omega^2]_{-1}$ is assumed to be zero, so the magnitude of the update is scaled by $\sqrt{\epsilon}$ (Equation 9). Smaller values for ρ reduce the impact of *preceeding* iterations, making the optimization more sensitive to the gradient of the *current* iteration (Equation 11).

Search performance was measured by E_{50} values, calculated according to the score criterion (LGA is successful if reported pose is within 1.0 kcal/mol from the global optimum). By plotting E_{50} as a function of N_{rot} (Figure 6), we observed that our initial hyperparameter setup ($\rho = 0.8$, and $\epsilon = 0.01$) yields the lowest E_{50} values among the tested hyperparameter combinations. Decreasing ϵ from 0.01 to 0.0001 results in larger E_{50} (i.e., lower performance), while increasing ρ results in very small performance gains. Fitting an exponential equation of the form $E_{50} = 2^{a*N_{\text{rot}}+b}$ (green dashed lines in Figure 6) helps in quantifying performance: the lower the regression coefficients (a and b), the smaller E_{50}, and thus the higher performance. This regression analysis confirms the visual interpretation of E_{50} plots, as $\rho = 0.8$, and $\epsilon = 0.01$ result in the lowest a and b, and thus the smallest number of iterations to reach a 50% probability of computing the optimum solution.

Algorithmic performance of ADADELTA and Solis-Wets. The algorithmic performances of ADADELTA and Solis-Wets were compared by calculating their E_{50} values for all protein-ligand complexes in our dataset. Moreover, for both LS methods, E_{50} was calculated as a function of N_{rot}, as well as for three different rates of performing local search on a genetic population (LS rate): 6%, 25%, and 100% (Figure 7).

In general, both LS methods require more score evaluations to achieve E_{50} if the ligand has a larger number of rotatable bonds (Figure 7). Interestingly, when using Solis-Wets, E_{50} values increase more steeply with N_{rot} than when using ADADELTA. This occurs for E_{50} values calculated based on

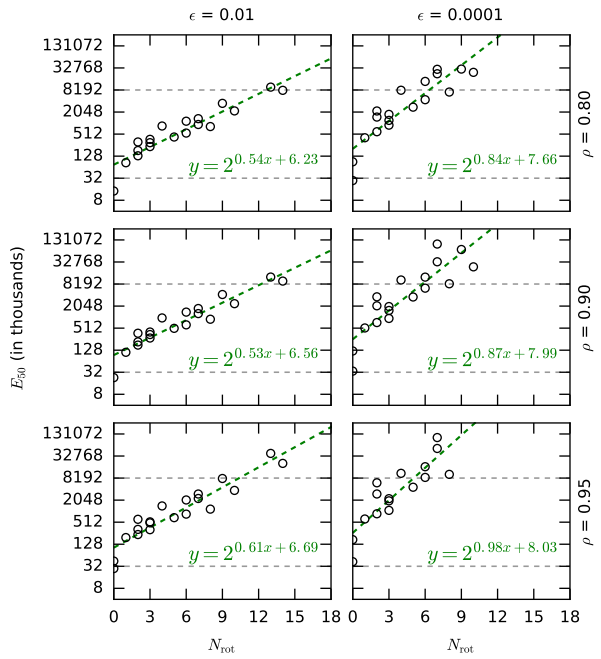

Fig. 6. Dependency of E_{50} on the number of rotatable bonds N_{rot} for six combinations of ADADELTA hyperparameters ρ and ϵ. The score criterion was used to calculate E_{50} values. The green dashed lines represents the fitted equation (also in green). A total of 19 protein-ligand complexes were used, but it was not always possible to fit E_{50} values, so there are less than 19 data-points in some plots. Local-search rate is 100%, and the maximum number of local-search iterations is 300.

both the score and RMSD as criteria for LGA success. When docking ligands with many rotatable bonds, ADADELTA reaches a 50% success rate with fewer evaluations than Solis-Wets. Notably, using Solis-Wets, we could not fit sigmoid curves (Equation 13) for any ligand containing 12 or more rotatable bonds because the LGA success rates were too low. On the other hand, using ADADELTA resulted in E_{50} below 8 million for some complexes with more than 15 rotatable bonds.

Analysis of regression lines associated with the equation $E_{50} = 2^{a*N_{\text{rot}}+b}$ reveals that increasing the LS rate results in lower coefficients a and b, and hence lower average E_{50} values. We argue that the a coefficient is more important than b because it dictates how E_{50} scales with the number of rotatable bonds. The influence of LS rate is more pronounced for ADADELTA than for Solis-Wets, with noticeable improvements in E_{50} values after increasing LS rate from 6% to 25%. Moreover, there is virtually no difference in performance when using either 25% or 100% LS rate, but using 100% is recommended because the corresponding a coefficients are slightly lower.

Let us compare the algorithmic performance of Solis-Wets and ADADELTA by using their E_{50} values, calculated according to the score criterion, and 100% LS rate. The regression model predicts an E_{50} value of $\sim 2^{0.99N_{\text{rot}}+5.50}$ for Solis-Wets, while for ADADELTA an E_{50} value of $\sim 2^{0.40N_{\text{rot}}+6.94}$ is predicted. For a ligand with zero rotatable bonds, the Solis-Wets E_{50} would be ∼45,000 evaluations, whereas the ADADELTA E_{50} would be ∼123,000. Therefore, ADADELTA needs ∼3x more evaluations than Solis-Wets. If the number of rotatable bonds is increased to 12, Solis-Wets would need ∼170,000,000 evaluations to reach a 50% success rate, while ADADELTA requires only 3,400,000 (a 50x reduction). Assuming these equations are still valid for 20 rotatable bonds, this factor would increase to ∼1300x. In other words, ADADELTA would need ∼30 *million* evaluations, while Solis-Wets is predicted to require ∼40 *billion* instead.

The agreement of E_{50} values calculated using both the score and RMSD criteria gives us more confidence in our analysis. Before using the RMSD criterion, we hypothesized that the role of ADADELTA was merely in removing clashes from poses that were already very close to the X-ray pose, and that Solis-Wets was just as good at producing poses within 2 Å RMSD from the X-ray, even though with less favorable energies. The RMSD criterion for LGA success invalidates this hypothesis, and shows that ADADELTA helps the LGA to find the native pose.

To better determine the number of rotatable bonds beyond which ADADELTA outperforms Solis-Wets, we plotted Solis-Wets E_{50} values against those of ADADELTA (Figure 8), both obtained using an LS rate of 100%. Ligands with three or fewer rotatable bonds (light blue circles) are typically below the equality line, which means that Solis-Wets has lower E_{50} values, and hence a higher algorithmic performance. On the other hand, and according to the score criterion, ligands with four or more rotatable bonds have lower E_{50} values using ADADELTA. Interestingly, according to the RMSD criterion, some ligands with N_{rot} between four and nine have lower E_{50} with Solis-Wets. Nevertheless, the majority of ligands with four or more rotatable bonds have lower E_{50} values with ADADELTA, according to either the score or RMSD criteria. This analysis considers only complexes for which E_{50} values could be calculated for both Solis-Wets and ADADELTA, so the maximum number of rotatable bonds is 11 (Solis-Wets has very low success rates for ligands with 12 or more rotatable bonds, and E_{50} values could not be estimated). Based on these results, we conclude that Solis-Wets has better algorithmic performance for ligands with fewer than three rotatable bonds, while ADADELTA has better algorithmic performance for ligands with four or more rotatable bonds.

Evaluation rate performance. The runtime of docking depends mostly on the number of evaluations, often reaching hundreds of millions. For example, performing 100 LGA runs with 2.5 million evaluations each results in 250 million evaluations being executed in a single docking. For this reason, the evaluation rate is a useful indicator of docking performance. In this section, we evaluate some speedups attained by GPU accelerators based on the *evaluation rate* – the time spent per score evaluation – expressed in microseconds (μs) per evaluation.

In Figure 9, we depict the evaluation rate attained when using our full dataset, and executing AUTODOCK-GPU on differ-

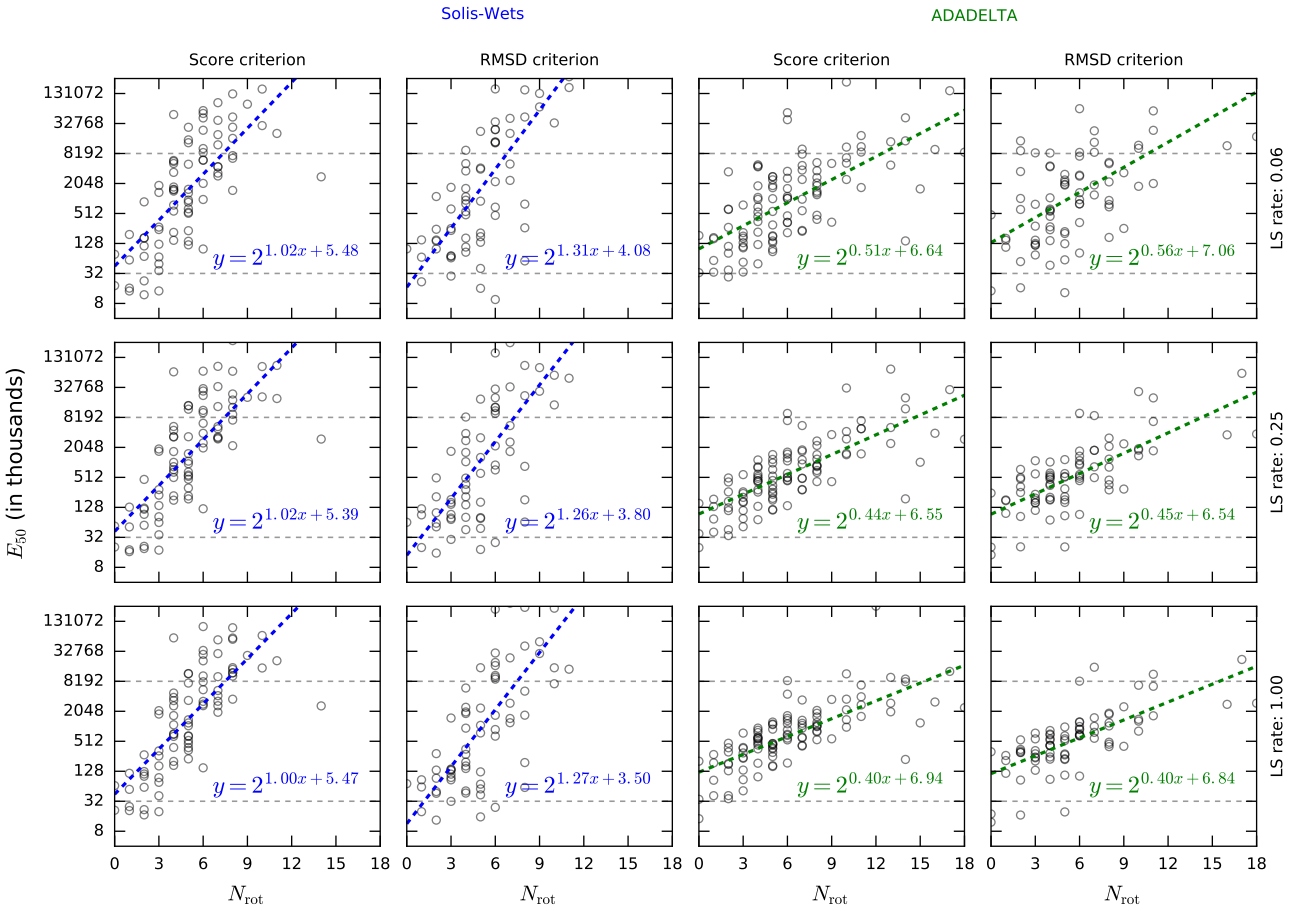

Fig. 7. Dependency of E_{50} on the number of ligand rotatable bonds N_{rot} for Solis-Wets and ADADELTA local-search methods. The LS rate is 6% in the top row, 25% in the middle row, and 100% in the bottom row.

ent GPUs (listed in Table 3), as well as AUTODOCK4 on a single CPU core. Evaluation rate speedups for each complex in our dataset were calculated by dividing the evaluation rate of AUTODOCK4 by that of AUTODOCK-GPU. To establish a fair baseline, AUTODOCK4 was run on a recent compute instance (c5.18xlarge) – provided by Amazon Web Services (AWS) – that features an Intel Xeon Platinum 8124M CPU @ 3.00GHz (Table 4).

AUTODOCK-GPU achieves the fastest evaluation rate on a GTX 1080 Ti GPU using Solis-Wets, spending between 0.007 - 0.28 μs per evaluation, depending on the protein-ligand complex. In contrast, AUTODOCK4 yields the lowest evaluation rate, ranging between 1.25 - 54.7 μs per evaluation.

In all platforms, more ligand atoms lead to slower calculations. For Solis-Wets, the time per evaluation increases in an almost linear fashion with the number of ligand atoms (N_{atom}). For ADADELTA, the time per evaluation increases in a more quadratic fashion. In consequence, ADADELTA becomes slower in comparison to Solis-Wets for large N_{atom} values. To provide an estimate, ADADELTA is about 4x as slow as Solis-Wets for ligands with ∼20 atoms, and about 16x slower for ligands with ∼100 atoms.

Table 2. Evaluation rate ($\mu s/evaluation$) for different GPU platforms and varying number of ligand atoms. Evaluation rates reported here are interpolated by polynomial fitting as shown in Figure 9.

	Evaluation rate (μs / evaluation)					
	Solis-Wets			ADADELTA		
# Ligand atoms	20	40	80	20	40	80
M2000	0.22	0.49	1.21	0.39	2.76	16.11
GTX 980	0.05	0.13	0.37	0.21	1.03	5.62
Vega 56	0.04	0.10	0.31	0.16	0.77	4.08
GTX 1080 Ti	0.03	0.06	0.17	0.06	0.44	2.40
AUTODOCK4	3.66	10.64	34.58			

Overall, the evaluation rate speedup achieved by each GPU depends on the hardware specifications (summarized in Table 3), with more powerful GPUs attaining larger speedups. Furthermore, evaluation rate speedups depend not only on the accelerator platform, but also on the choice of LS method and the number of ligand atoms (Figure 10). For dockings using Solis-Wets, evaluation rate speedups increase with the number of ligand atoms, while the opposite behavior occurs for ADADELTA. In the case of very large ligands (∼100 atoms), we observed evaluation rate speedups of ADADELTA being lower than 10x on a Vega 56 GPU.

Table 3. Comparison of selected hardware characteristics of the evaluated GPU-based accelerator cards.

GPU-based Accelerator Card	Peak Memory Bandwidth (GB/s)	Peak Compute Performance (GFLOPS/s)		On-board Memory (GB)	Release Year	Geometric Average Evaluation Rate Speedup	
		FP32	FP64			Solis-Wets	ADADELTA
M2000	105.8	1786	55.82	4	2016	19.6	5.4
GTX 980	224.4	4981	155.6	4	2014	75.4	14.9
Vega 56	409.6	10566	660.4	8	2017	94.4	17.3
GTX 1080 Ti	484.4	11340	354.4	11	2017	157.8	36.0

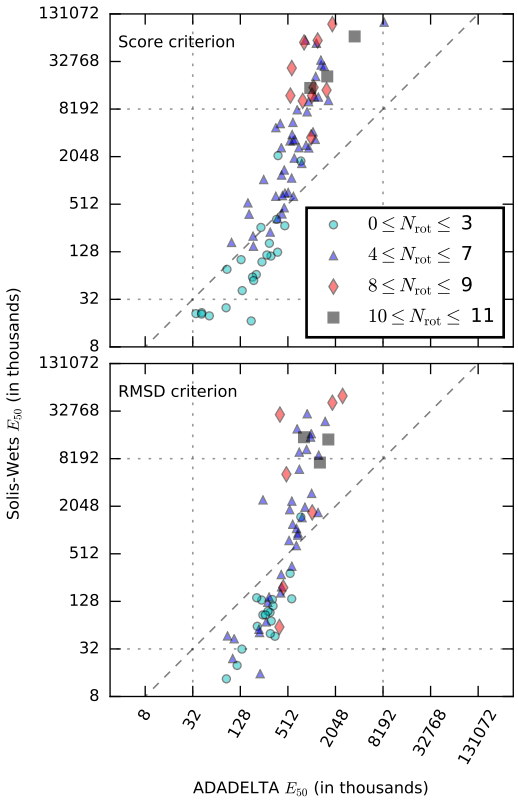

Fig. 8. Direct comparison of E_{50} values between ADADELTA and Solis-Wets using 100% LS rate.

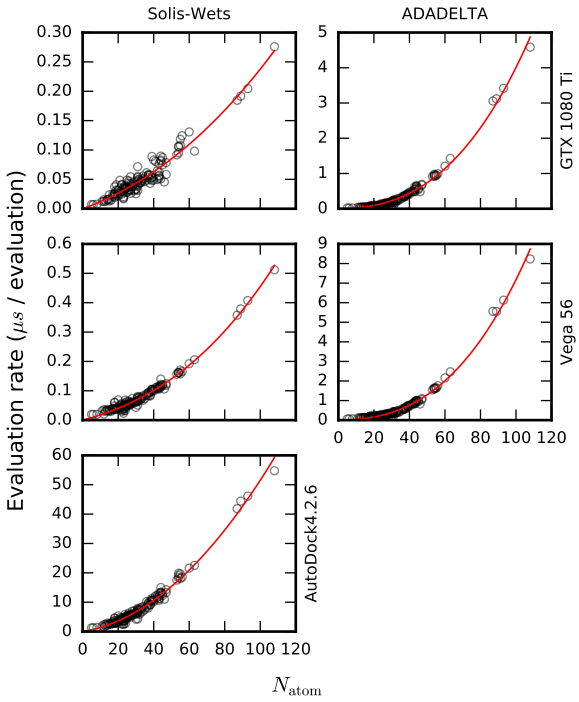

Fig. 9. Evaluation rate of AUTODOCK-GPU and AUTODOCK4. Note that each subplot has a different scale for the y-axis. The red lines are third-degree polynomial fits. These fits were used to interpolate the evaluation rate values reported in Table 2.

Balancing algorithmic and evaluation rate performance. In our previous analysis, we estimated *algorithmic* improvements of ADADELTA over Solis-Wets to be in the order of 50x (for ligands with 12 or more rotatable bonds), as well as faster executions of Solis-Wets compared to ADADELTA in the order of 16x (for ligands containing ∼100 atoms). Thus, for ligands with 12 or more rotatable bonds, ADADELTA is the obvious choice, because the reduction in the number of score evaluations outweighs the faster evaluation rate of Solis-Wets.

However, the choice between these LS methods is less obvious for ligands with fewer than 12 rotatable bonds. To clarify this, we calculated the time required to perform E_{50} evaluations with each method. This time-to-E_{50} is the product of E_{50} by the corresponding evaluation rate (μs / evaluation) for each protein-ligand complex. By plotting the resulting time-to-E_{50} values of Solis-Wets against those of ADADELTA (Figure 11), we compare both LS methods in terms of time efficiency for reaching a 50% success rate on a single LGA run. While ADADELTA is successful in fewer evaluations than Solis-Wets for ligands with four or more rotatable bonds, the faster evaluation rate of Solis-Wets compensates for the larger number of evaluations. As a result, ADADELTA is more time-efficient for ligands with eight or more rotatable bonds, while Solis-Wets is more time-efficient for ligands with seven or fewer rotatable bonds. These numbers were calculated using evaluation rates obtained with the Vega 56 GPU, which has an intermediate performance within the range of GPU cards we tested. While results may be slightly different on other GPU cards, this analysis provides an estimate of the time-efficiency of each LS method.

Additionally, we note that in the current version of AUTODOCK-GPU, the number of evaluations is predetermined at the start of docking by a (user) command-line option. Thus, the time-to-completion of docking is not strictly related to the time-to-E_{50} described above. However, in the future, self-tunning stopping criteria may

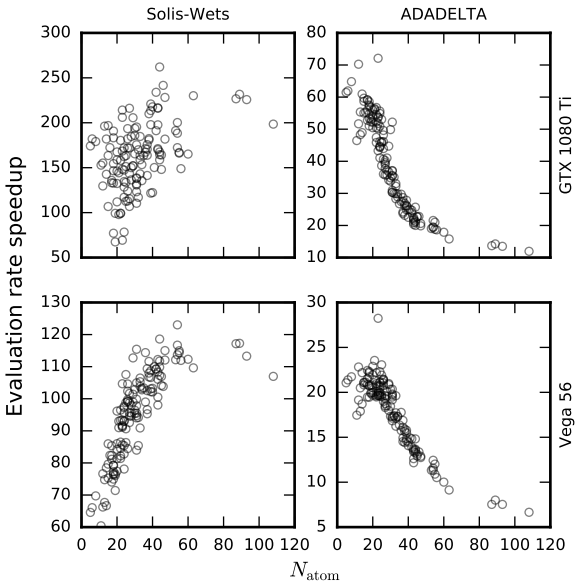

Fig. 10. Evaluation rate speedup of AUTODOCK-GPU (on GTX 1080 Ti and Vega 56 GPUs) over AUTODOCK4 (on a single CPU core of an AWS c5.18x instance). Note that each subplot has a different scale for the y-axis.

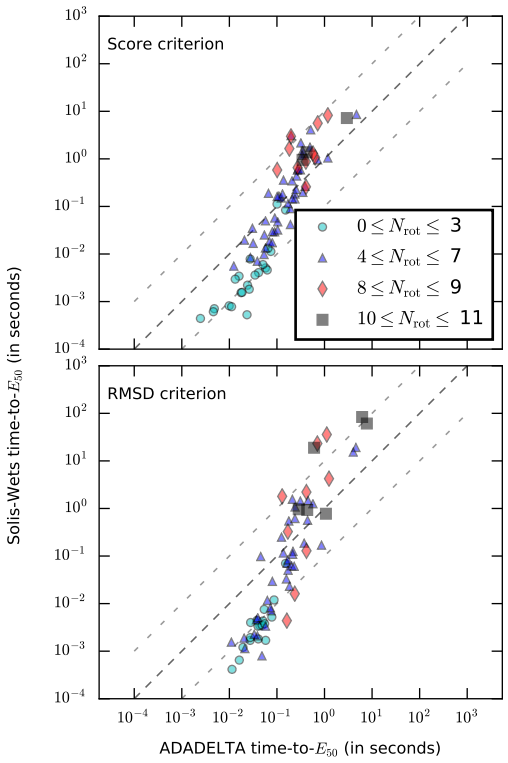

Fig. 11. Direct comparison of time-to-E_{50} values between ADADELTA and Solis-Wets using 100% LS rate. For a given protein-ligand complex, time-to-E_{50} is the product of E_{50} by the corresponding evaluation rate (μs / evaluation). Here, evaluation rates were collected on the Vega 56 GPU platform.

be developed to automatically identify convergence (e.g., AUTODOCKFR (39)), and hence, reduce the overall computation time by preventing unnecessary evaluations from being performed past the point of convergence. In that scenario, the time-to-E_{50} could be a good estimator for the time-to-completion of docking.

Runtime performance. Here, we report the achieved runtime speedups of AUTODOCK-GPU (on various GPUs) with respect to AUTODOCK4 (on a single CPU core). This analysis is based on the *time-to-completion*, understood as the total *wall clock* time spent by the docking program. The corresponding runtime speedups are therefore slightly different from evaluation rate speedups reported in a previous section. While score evaluations account for the generally most time-consuming part of docking, there are other tasks, such as loading input files or clustering LGA final poses, that are not accounted for by the evaluation rate. For this reason, reporting runtime speedups is complementary to evaluation rate speedups.

For this analysis, we selected five different ligand-receptor complexes with different number of atoms (N_{atom}) and rotatable bonds (N_{rot}). Although this set is small, it is yet representative for the purpose of this section as it provides *enough* variety of low (PDB ID: 5tim), medium (PDB ID: 2bm2, 5wlo), and high (PDB ID: 4er4, 3er5) molecular size and complexity.

The results are presented in Figure 12. Overall, Solis-Wets leads to higher runtime speedups than ADADELTA, in agreement with the previously discussed evaluation rate speedups. The better the GPU card, the higher the runtime speedups, with a nearly 10x difference observed between the M2000 and the GTX 1080 Ti GPUs. For Solis-Wets, ligands with more atoms yield higher speedups, while the opposite trend is observed for ADADELTA. This is in agreement with the severe evaluation rate slowdown observed for ADADELTA with increased number of atoms. However, while a Solis-Wets docking will finish *faster* for a given number of evaluations, ADADELTA may return significantly *better* poses, both in terms of score and RMSD, due to its higher algorithmic performance for ligands with many rotatable bonds, especially eight or more.

Leveraging OpenCL portability. A key feature of OpenCL is its portability. This allows running applications – *ideally with none or only minor recoding effort* – on different devices. In this study, this feature was leveraged by using multicore CPUs as execution platform in addition to GPUs. Our implementation ensures that the porting effort from GPUs to CPUs is reduced to just a re-compilation step, i.e., re-building the application targeting a different device architecture, but requiring no code changes.

Performance scaling on multicore CPUs. Performance scaling refers to the performance enhancement due to an increase of computing resources. This is typically translated into shorter runtimes, and hence, into lower costs for such resources. In order to assess the resource investment in terms of

Table 4. Hardware characteristics of evaluated CPU-based platforms. Prices comprise the basic cost of compute services only (40) (i.e., underlying servers in the EU zone), and do not include charges for additional services (e.g., storage, database, migration, etc). In addition, as suggested for high-performance workloads (41), hyperthreading was disabled, allowing only a single thread per CPU core.

AWS CPU-based platform	Number of cores	Peak Memory Bandwidth (GB/s)	Peak Compute Performance FP32 (GFLOPS/s)	Total RAM Memory (GB)	On-demand EC2 price EU zone ($/hour)	Release Year
`c5.4xlarge`	8	42	768	30	0.776	2017
`c5.9xlarge`	16	85	1536	70	1.746	2017
`c5.18xlarge`	36	128	3456	140	3.492	2017

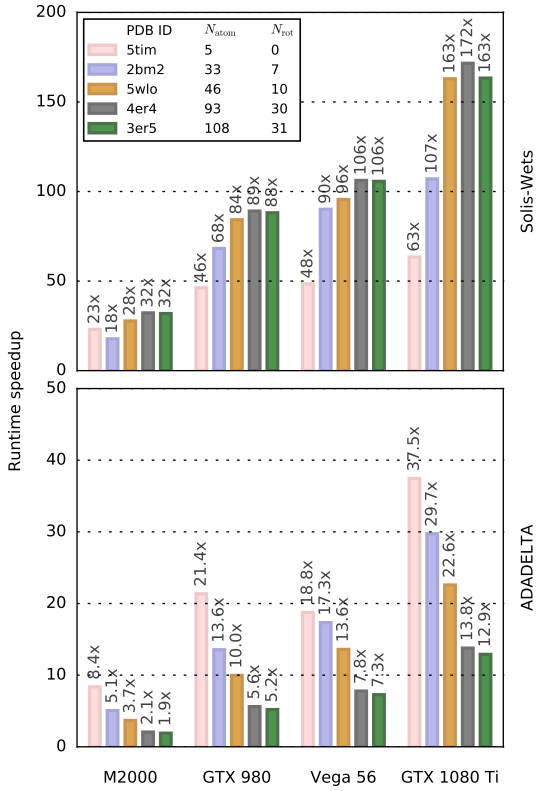

Fig. 12. Runtime speedups of AUTODOCK-GPU (on various GPUs) with respect to AUTODOCK4 on a single CPU core. The number of LGA runs is 100, and the number of evaluations is 2,048,000.

the returned performance gains, we discuss the performance scaling behavior of AUTODOCK-GPU on multicore CPUs.

To do that, we selected AWS CPU instances equipped with different numbers of cores – e.g., `c5.4xlarge` (8 cores), `c5.9xlarge` (16 cores), and `c5.18xlarge` (36 cores) – the latter being the same CPU platform used in previous experiments (only a single CPU core was used for AUTODOCK4). This analysis was performed on the same five complexes used for reporting GPU runtime speedups (Figure 12).

In contrast to GPUs, where the overall performance when running Solis-Wets *increases* along with N_{atom} (Figure 12), multicore CPUs yield *lower* runtime performance for larger N_{atom} values (Figure 13). For instance, on an 8-core instance, Solis-Wets achieves ∼12x of runtime speedup with the smallest complex (PDB ID: 5tim), while only reaching ∼2x with the largest one (PDB ID: 3er5). We attribute the different speedup dependencies on N_{atom} and N_{rot} values between multicore CPU and GPU to the considerably larger number of fine-grained computing cores on GPUs (e.g., 3584 CUDA cores on a GTX 1080 Ti GPU) that more efficiently leverage the larger fine-grained parallelism – provided by more atoms and rotatable bonds – than by using fewer (here: eight) multicore CPU cores (e.g., a maximum of 36 cores on the AWS `c5.18xlarge` instance).

From the numbers in Figure 13, it can be noted that speedups increase with a factor of ∼2x when e.g., migrating from an instance with eight cores (`c5.4xlarge`) to another with 16 cores (`c5.9xlarge`). Slightly higher runtime speedup gains are observed when migrating to the largest `c5.18xlarge` instance, which can be simply explained by the upgrade provided by having more than double the number of cores (= 36). A similar speedup scaling behavior was observed when running ADADELTA. Additionally, Figure 13 depicts the resource-utilization efficiency, calculated as the ratio between the runtime speedup and the number of CPU cores present in the employed AWS CPU instances. For each LS method, all CPU instances achieve similar resource-utilization efficiencies, which are higher when no rotatable bonds are present (`5tim`), achieving ∼1.4x and ∼0.75x when using Solis-Wets and ADADELTA, respectively. For the other complexes (having more than seven rotatable bonds), such efficiencies become lower but more stable, i.e., ∼0.3x and ∼0.2x when using Solis-Wets and ADADELTA, respectively.

While the larger AWS CPU instance (`c5.18xlarge`) provided the highest runtime speedup among all multicore CPUs employed, it is relevant to contextualize its performance when running AUTODOCK-GPU by comparing it against high-end GPUs offered by AWS as well. For that purpose, we considered one of the latest AWS GPU-based machines – i.e., a `p3.2xlarge` instance – equipped with a single Volta (V100) GPU card, and costing 3.823 $/hour in the EU zone (40). Figure 14 shows the much higher runtime speedups achieved on the `p3.2xlarge` instance (a maximum of ∼367x on `4er4`) compared to those of `c5.18xlarge` (also shown in Figure 13). Furthermore, Figure 14 compares the economic efficiency yielded on both instances. This latter metric, calculated as the ratio between the runtime speedups and their prices, shows that – despite the slightly higher price-per-hour of `p3.2xlarge` (3.823 $/hour) with respect to that of `c5.18xlarge` (3.456 $/hour) – the returned gains in terms of the runtime

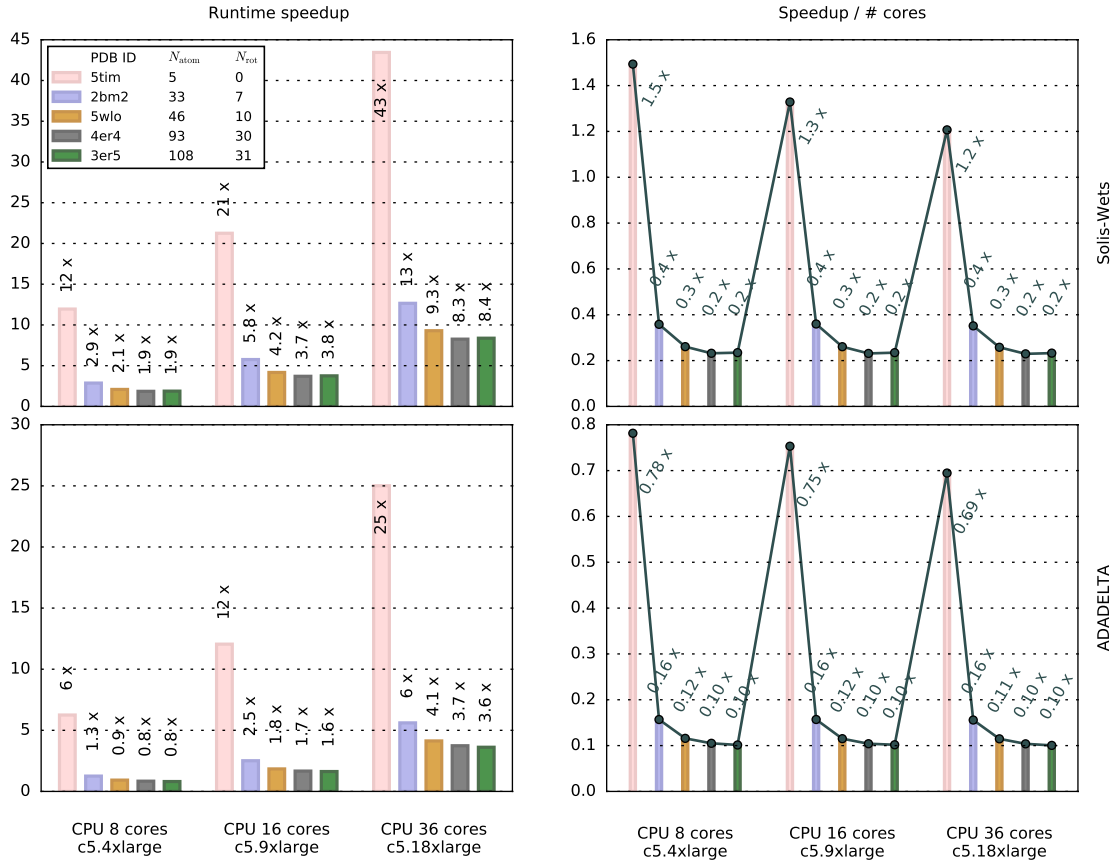

Fig. 13. Performance scalability (left) and resource-utilization efficiency (right), both in terms of the number of CPU cores. The number of LGA runs is 100, and the number of evaluations is 2,048,000. For each ligand-receptor complex, same program commands were executed on all three CPU instances.

speedup are *always* higher for GPUs than for CPUs, when using both LS methods. Therefore, it is worth investing on larger machines (having more computing resources) for executing AUTODOCK-GPU.

Porting AutoDock to other hardware accelerators. The *functional* portability of OpenCL can be exploited further. For instance, porting AUTODOCK-GPU onto emerging OpenCL-capable accelerators (which might lack driver support from the vendor) would be feasible by using *open-source* implementations such as POCL (42). Moreover, in recent years, OpenCL has gained increasing attention from manufacturers of highly reconfigurable accelerators such as Field Programmable Gate Arrays (FPGAs). Development tools and compilers for OpenCL on FPGAs are becoming more competitive to those of GPUs and CPUs, and are closing the gap between all these technologies. Although *performance* portability is still a known issue of OpenCL, our previous study of AUTODOCK-GPU on FPGAs (43) shows that this can be mitigated (up to a certain extent) with more specialized programming styles and patterns.

CONCLUSIONS

This work describes AUTODOCK-GPU, an OpenCL-accelerated version of AUTODOCK4 running on GPUs, enhanced with the ADADELTA gradient-based method used for local search. We observed improvements of both algorithmic performance as well as in the evaluation rate.

Algorithmic performance refers to the number of score evaluations required to achieve a pre-defined level of solution quality. This level is based on the score defined by the AUTODOCK4 scoring function, together with the structural quality of docked poses (measured by the RMSD from the native pose). Based on our experiments, the algorithmic improvement of ADADELTA over Solis-Wets increases with the number of rotatable bonds. For a ligand with three or four rotatable bonds, the algorithmic performance of both local-search methods is similar. For a ligand with 12 rotatable bonds, ADADELTA finds correct solutions while performing only 1/50th of evaluations than Solis-Wets. For a ligand with 20 rotatable bonds, we estimate this reduction to increase to 1/1300. Algorithmic improvements of this magnitude can be the difference between a successful and a failed docking.

The evaluation rate is the time spent (in μs) per score evaluation. On GPUs, the time spent in a single ADADELTA evaluation is 4x - 16x longer than in one Solis-Wets evaluation. This is because ADADELTA requires the calculation of gradients, which is *computationally* more expensive.Therefore, for ligands with a small number of rotatable bonds, the algorithmic advantage of ADADELTA does not outweigh its increased computational cost. Taking into account both the time spent per evaluation, as well as the number of evalua-

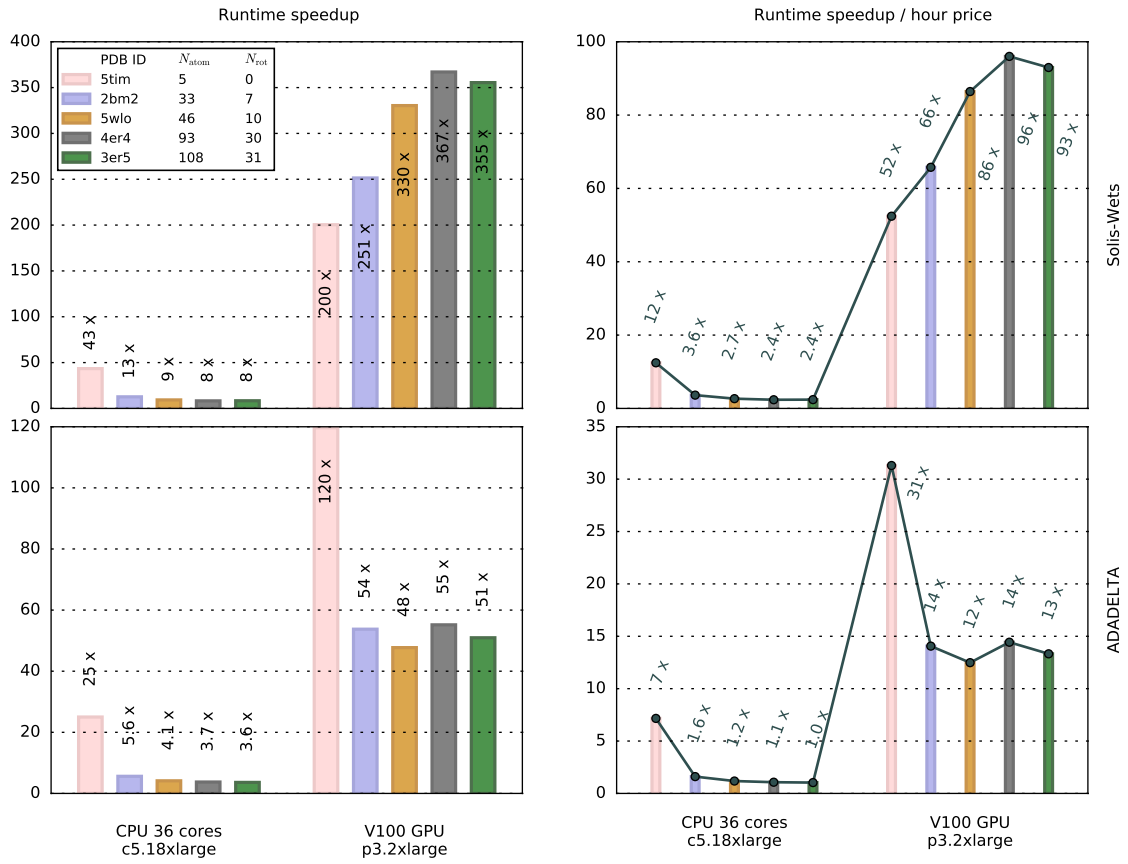

Fig. 14. Comparison of runtime speedup (left) and economic efficiency (right) between CPU and GPU instances on AWS: `c5.18xlarge` and `p3.2xlarge`, respectively. The number of LGA runs is 100, and the number of evaluations is 2,048,000. For each ligand-receptor complex, same program commands were executed on both instances.

tions required to produce correct solutions, we estimate that Solis-Wets is better for ligands with seven or fewer rotatable bonds, while ADADELTA is better for ligands with eight or more rotatable bonds.

Regarding runtime improvements, AUTODOCK-GPU achieves speedups ranging between ∼2x and ∼170x on GPUs, depending on the hardware specifications of the accelerator platform, and the local-search method. The number of ligand atoms also affects runtime speedups, with larger ligands yielding higher speedups for Solis-Wets, but lower speedups for ADADELTA. Interestingly, if the accelerator platform is a multicore CPU, more ligand atoms always cause a lower speedup, independently of the local-search method used. This is probably due to less effective parallelization of fine-grained tasks on CPUs, with their reduced number (compared to GPUs) of available processing elements (cores). We found that such tasks can be parallelized more effectively on GPUs, with their many (in the order of thousands) fine-grained processing elements.

The accelerator platforms tested in this study were commercial GPUs from different vendors (Nvidia & AMD) that ranged from low- to high-end devices. Using GPUs with such different computing capabilities allows potential users to estimate the performance of AUTODOCK-GPU on systems similar to those available at their sites. Furthermore, we leveraged the portability of OpenCL by using AUTODOCK-GPU on general-purpose multicore CPUs and showing that it is still able to provide scalable performance gains. This could be beneficial in research settings where high-end GPUs are not available.

Finally, the improvements reported here are very large in terms of improved computational efficiency. Although other docking programs exist that implement gradient-based local-search methods (e.g., AUTODOCK VINA), this work provides a much faster computation of the AUTODOCK4 scoring function, adding a new highly efficient tool to the toolbox of medicinal chemists.

ACKNOWLEDGMENTS. This work was supported by the National Institutes of Health GMO69832 (to SF), the AWS Cloud Credits for Research program, and by the ALEPRONA funding program #57186883 from the German Academic Exchange Service (DAAD) and the Peruvian National Program for Scholarships and Educational Loans (PRONABEC).

We thank JC Ducom and the HPC facility of The Scripps Research Institute, as well as AMD Inc. for technical support. We acknowledge the use of GNU Parallel (44), Matplotlib (45) and Seaborn (46).

AVAILABILITY. AUTODOCK-GPU is open source under a GPL licence. Its source code as well as its documentation

is available at:

https://autodock.scripps.edu
https://github.com/ccsb-scripps/AutoDock-GPU

REFERENCES

1. Brian K. Shoichet. Virtual screening of chemical libraries. *Nature*, 432:862–865, 2004.
2. John J. Irwin and Brian K. Shoichet. Zinc - a free database of commercially available compounds for virtual screening. *J. Chemical Information and Modeling*, 45(1):177–182, 2005.
3. Brian K. Shoichet. Screening in a spirit haunted world. *J. Drug Discovery Today*, 11(13): 607–615, 2006.
4. Brian K. Shoichet and Brian K. Kobilka. Structure-based drug screening for g-protein-coupled receptors. *J. Trends in Pharmacological Sciences*, 33(5):268 – 272, 2012.
5. Virginia A. Kincaid, Nir London, Kittikhun Wangkanont, Darryl A. Wesener, Sarah A. Marcus, Annie Héroux, Lyudmila Nedyalkova, Adel M. Talaat, Katrina T. Forest, Brian K. Shoichet, and Laura L. Kiessling. Virtual screening for udp-galactopyranose mutase ligands identifies a new class of antimycobacterial agents. *J. ACS Chemical Biology*, 10(10):2209–2218, 2015.
6. Marc-André Elsliger, Ashley M. Deacon, Adam Godzik, Scott A. Lesley, John Wooley, Kurt Wüthrich, and Ian A. Wilson. The jcsg high-throughput structural biology pipeline. *J. Acta Crystallographica Section F*, 66(10):1137–1142, 2010.
7. Jung-Hsin Lin, Alexander L. Perryman, Julie R. Schames, and J. Andrew McCammon. The relaxed complex method: Accommodating receptor flexibility for drug design with an improved scoring scheme. *J. Biopolymers*, 68(1):47–62, 2003.
8. Oleg Trott and Arthur J. Olson. Autodock vina: Improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading. *J. Computational Chemistry*, 31(2):455–461, 2010.
9. Stefano Forli, Ruth Huey, Michael E. Pique, Michel F. Sanner, David S. Goodsell, and Arthur J. Olson. Computational protein–ligand docking and virtual drug screening with the autodock suite. *Nature Protocols*, 11:905–919, 2016.
10. David P. Anderson. Boinc: A system for public-resource computing and storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, Washington, DC, USA, 2004. IEEE Computer Society.
11. World Community Grid. https://www.worldcommunitygrid.org. accessed January 30, 2019.
12. Daniel J. Mermelstein, Charles Lin, Gard Nelson, Rachael Kretsch, J. Andrew McCammon, and Ross C. Walker. Fast and flexible gpu accelerated binding free energy calculations within the amber molecular dynamics package. *J. Computational Chemistry*, 39(19):1354–1358, 2018.
13. John E. Stone, David J. Hardy, Jan Saam, Kirby L. Vandivort, and Klaus Schulten. Chapter 1 - gpu-accelerated computation and interactive display of molecular orbitals. In *GPU Computing Gems Emerald Edition*, Applications of GPU Computing Series, pages 5–18. Morgan Kaufmann, 2011.
14. John E. Stone, Antti-Pekka Hynninen, James C. Phillips, and Klaus Schulten. Early experiences porting the namd and vmd molecular simulation and analysis software to gpu-accelerated openpower platforms. In *High Performance Computing*, Cham, 2016. Springer.
15. Sparsh Mittal and Jeffrey S. Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Computing Surveys*, 47(4):69:1–69:35, 2015.
16. Evgenij Belikov, Pantazis Deligiannis, Prabhat Totoo, Malak Aljabri, and Hans-Wolfgang Loidl. A survey of high-level parallel programming models. Technical Report HW-MACS-TR-0103, 2013.
17. J. Diaz, C. Muñoz-Caro, and A. Niño. A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on Parallel and Distributed Systems*, 23 (8):1369–1386, 2012.
18. Garrett M. Morris, Ruth Huey, William Lindstrom, Michel F. Sanner, Richard K. Belew, David S. Goodsell, and Arthur J. Olson. Autodock4 and autodocktools4: Automated docking with selective receptor flexibility. *J. Computational Chemistry*, 30(16):2785–2791, 2009.
19. Matthew D. Zeiler. ADADELTA: an adaptive learning rate method. *arXiv*, abs/1212.5701, 2012.
20. Francisco J. Solis and Roger J. B. Wets. Minimization by random search techniques. *J. Mathematics of Operations Research*, 6(1):19–30, 1981.
21. Ruth Huey, Garrett M. Morris, Arthur J. Olson, and David S. Goodsell. A semiempirical free energy force field with charge-based desolvation. *J. Computational Chemistry*, 28(6): 1145–1152, 2007.
22. Garrett M. Morris, David S. Goodsell, Robert S. Halliday, Ruth Huey, William E. Hart, Richard K. Belew, and Arthur J. Olson. Automated docking using a lamarckian genetic algorithm and an empirical binding free energy function. *J. Computational Chemistry*, 19 (14):1639–1662, 1998.
23. Leonardo Solis-Vasquez and Andreas Koch. A performance and energy evaluation of opencl-accelerated molecular docking. In *Proceedings of the 5th International Workshop on OpenCL*, New York, NY, USA, 2017. ACM.
24. OCLADock - OpenCL Accelerated Molecular Docking. https://git.esa.informatik.tu-darmstadt.de/docking/ocladock. accessed October 10, 2018.
25. The OpenCL Specification. https://www.khronos.org/opencl. accessed October 10, 2018.
26. Imre Pechan and Bela Feher. Molecular docking on fpga and gpu platforms. In *Proceedings of the 21st International Conference on Field Programmable Logic and Applications*. IEEE, 2011.
27. Imre Pechan, Béla Fehér, and Attila Bérces. Fpga-based acceleration of the autodock molecular docking software. In *Proceedings of the 6th Conference on Ph.D. Research in Microelectronics Electronics*. IEEE, 2010.
28. OpenCL 2.0 Reference Pages. https://www.khronos.org/registry/OpenCL/sdk/2.0/docs/man/xhtml. accessed October 10, 2018.
29. Jinghui Zhong, Xiaomin Hu, Jun Zhang, and Min Gu. Comparison of performance between different selection strategies on simple genetic algorithms. In *Proceedings of the International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce*. IEEE, 2005.
30. Noraini Mohd Razali and John Geraghty. Genetic algorithm performance with different selection strategies in solving tsp. In *World Congress on Engineering*. IAENG, 2011.
31. Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, 2 edition, 2006.
32. Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv*, abs/1412.6980, 2014.
33. Erik Bitzek, Pekka Koskinen, Franz Gähler, Michael Moseler, and Peter Gumbsch. Structural relaxation made simple. *J. Physical Review Letters*, 97(17):170201, 2006.
34. Michael J. Hartshorn, Marcel L Verdonk, Gianni Chessari, Suzanne C Brewerton, Wijnand TM Mooij, Paul N Mortenson, and Christopher W Murray. Diverse, high-quality test set for the validation of protein- ligand docking performance. *J. Medicinal Chemistry*, 50(4): 726–741, 2007.
35. Yan Li, Li Han, Zhihai Liu, and Renxiao Wang. Comparative assessment of scoring functions on an updated benchmark: 2. evaluation methods and general results. *J. Chemical Information and Modeling*, 54(6):1717–1736, 2014.
36. Helen M. Berman, John Westbrook, Zukang Feng, Gary Gilliland, T. N. Bhat, Helge Weissig, Ilya N. Shindyalov, and Philip E. Bourne. The protein data bank. *J. Nucleic Acids Research*, 28(1):235–242, 2000.
37. Noel M. O'Boyle, Michael Banck, Craig A. James, Chris Morley, Tim Vandermeersch, and Geoffrey R. Hutchison. Open babel: An open chemical toolbox. *J. Cheminformatics*, 3(1): 33, 2011.
38. Nikolaus Hansen, Anne Auger, Raymond Ros, Steffen Finck, and Petr Pošík. Comparing results of 31 algorithms from the black-box optimization benchmarking bbob-2009. In *Proceedings of the 12th Annual Conference Companion on Genetic and Evolutionary Computation*, New York, NY, USA, 2010. ACM.
39. Pradeep A. Ravindranath, Stefano Forli, David S. Goodsell, Arthur J. Olson, and Michel F. Sanner. Autodockfr: Advances in protein-ligand docking with explicitly specified binding site flexibility. *J. PLOS Computational Biology*, 11(12):1–28, 2015.
40. Amazon EC2 Pricing. https://aws.amazon.com/ec2/pricing/on-demand/, . accessed January 30, 2019.
41. Amazon Elastic Compute Cloud, User Guide for Linux Instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-optimize-cpu.html, . accessed January 30, 2019.
42. Pekka Jääskeläinen, Carlos Sánchez de La Lama, Erik Schnetter, Kalle Raiskila, Jarmo Takala, and Heikki Berg. pocl: A performance-portable opencl implementation. *Int. J. of Parallel Programming*, 43(5):752–785, 2015.
43. Leonardo Solis-Vasquez and Andreas Koch. A case study in using opencl on fpgas: Creating an open-source accelerator of the autodock molecular docking software. In *Proceedings of the 5th International Workshop on FPGAs for Software Programmers*, Berlin, Germany, 2018. VDE Verlag GmbH.
44. O. Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36 (1):42–47, 2011.
45. J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi: 10.1109/MCSE.2007.55.
46. Seaborn, 10.5281/zenodo.592845.

APPENDIX

Table 5. Upper limits on AUTODOCK-GPU docking parameters (defined in /common/defines.h).

Upper limit identifier	Description (as maximum number of)	Value
ATYPE_NUM	Ligand atomic types for smoothing	22
MAX_NUM_OF_ATOMS	Ligand atoms	256
MAX_NUM_OF_ATYPES	Ligand atomic types for scoring function	14
MAX_NUM_OF_ROTBONDS	Rotatable bonds	32
MAX_INTRAE_CONTRIBUTORS	Intramolecular (pairwise) energy contributors	MAX_NUM_OF_ATOMS * MAX_NUM_OF_ATOMS
MAX_NUM_OF_ROTATIONS	Rotations to be performed	MAX_NUM_OF_ATOMS * MAX_NUM_OF_ROTBONDS
MAX_POPSIZE	Individuals in a population	2048
MAX_NUM_OF_RUNS	Docking runs	1000
MAX_NUM_GRIDPOINTS	Grid points per dimension	256

Table 6. Constant data structures and their members in AUTODOCK-GPU.

Struct label	Constant array struct member	Element datatype	Size definition	Size calculation	Size (Bytes)
A	atom_charges	float	MAX_NUM_OF_ATOMS	$4*256$	1024
	atom_types	char	MAX_NUM_OF_ATOMS	$1*256$	256
B	intraE_contributors	char	3*MAX_INTRAE_CONTRIBUTORS	$1*3*256*256$	196608
C	reqm	float	ATYPE_NUM	$4*22$	88
	reqm_hbond	float	ATYPE_NUM	$4*22$	88
	atom1_types_reqm	unsigned int	ATYPE_NUM	$4*22$	88
	atom2_types_reqm	unsigned int	ATYPE_NUM	$4*22$	88
	VWpars_AC	float	(MAX_NUM_OF_ATYPES * MAX_NUM_OF_ATYPES)	$4*14*14$	784
	VWpars_BD	float	(MAX_NUM_OF_ATYPES * MAX_NUM_OF_ATYPES)	$4*14*14$	784
	dspars_S	float	MAX_NUM_OF_ATYPES	$4*14$	56
	dspars_V	float	MAX_NUM_OF_ATYPES	$4*14$	56
D	rotlist	int	MAX_NUM_OF_ROTATIONS	$4*256*32$	32768
E	ref_coords_x	float	MAX_NUM_OF_ATOMS	$4*256$	1024
	ref_coords_y	float	MAX_NUM_OF_ATOMS	$4*256$	1024
	ref_coords_z	float	MAX_NUM_OF_ATOMS	$4*256$	1024
	rotbonds_moving_vectors	float	3*MAX_NUM_OF_ROTBONDS	$4*3*32$	384
	rotbonds_unit_vectors	float	3*MAX_NUM_OF_ROTBONDS	$4*3*32$	384
	ref_orientation_quats	float	4*MAX_NUM_OF_RUNS	$4*4*1000$	16000
				Subtotal size (Bytes)	252528
GRIDS	fgrids	float	MAX_NUM_OF_ATYPES * MAX_NUM_GRIDPOINTS^3	$4*16*256^3$	1073741824
				Total size (Bytes)	1073994352

Table 7. Additional constant data structures and their members for gradient calculation in AUTODOCK-GPU.

Struct label	Constant array struct member	Element datatype	Size definition	Size calculation	Size (Bytes)
F	rotbonds_atoms	int	MAX_NUM_OF_ATOMS* MAX_NUM_OF_ROTBONDS	$4*256*32$	32768
G	rotbonds	int	2*MAX_NUM_OF_ROTBONDS	$4*2*32$	256
	num_rotating_atoms_per_rotbond	int	MAX_NUM_OF_ROTBONDS	$4*32$	128
	angle	float	1000	$4*1000$	4000
	dependence_on_theta	float	1000	$4*1000$	4000
	dependence_on_rotangle	float	1000	$4*1000$	4000
				Total size (Bytes)	45152