A Performance and Energy Evaluation of OpenCL-accelerated Molecular Docking

Leonardo Solis-Vasquez
Technische Universität Darmstadt
Darmstadt, Germany
solis@esa.tu-darmstadt.de

Andreas Koch
Technische Universität Darmstadt
Darmstadt, Germany
koch@esa.tu-darmstadt.de

ABSTRACT

Molecular Docking is a methodology used extensively in modern drug design. It aims to predict the binding position of two molecules by calculating the energy of their possible binding poses. One of the most cited docking tools is AutoDock. At its core, it solves an optimization problem by generating a large solution space of possible poses, and searches among them for the one having the lowest energy. These complex algorithms thus benefit from parallelization based run-time acceleration. This work presents an OpenCL implementation of AutoDock, and a corresponding performance evaluation on two different platforms based on multi-core CPU and GPU accelerators. It shows that OpenCL allows highly efficient docking simulations, achieving speedups of ~4x and ~56x over the original serial AutoDock version, as well as energy efficiency gains of ~2x and ~6x. respectively. To the best of our knowledge, this work is the first one also considering the energy efficiency of molecular docking programs.

CCS CONCEPTS

•**Applied computing** →**Molecular structural biology ;**
•**Computing methodologies** →*Massively parallel algorithms ;*
•**Hardware** →*Enterprise level and data centers power issues;*

KEYWORDS

Molecular docking, AutoDock, OpenCL, parallelization, GPU, energy efficiency

ACM Reference format:
Leonardo Solis-Vasquez and Andreas Koch. 2017. A Performance and Energy Evaluation of OpenCL-accelerated Molecular Docking. In *Proceedings of The OpenCL Conference, Toronto, Canada, May 2017 (IWOCL'17),* 11 pages.
DOI: 10.475/123_4

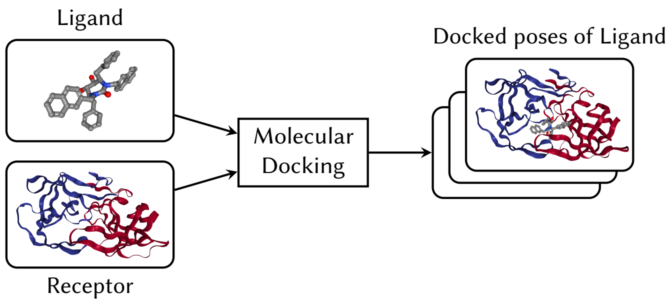

Figure 1: Molecular docking searches for multiple possible docking poses for a ligand and a receptor.

1 INTRODUCTION

Molecular docking (MD) computations are used extensively in structure-based drug design. They aim to predict the predominant binding pose(s) of two molecules: a ligand and a receptor, both of known three-dimensional structure. MD is used to identify ligands that react as good inhibitors or drug candidates, in the interaction with a given target receptor. MD is based on simulation, and models the interaction of these two molecules in great detail, allowing flexibility for ligand modeling and detailed molecular mechanics for energy calculation. In a general, a docking simulation starts with a ligand placed randomly outside the target. During simulation the ligand experiences translations, orientations and conformations, until a suitable binding site is found (Figure 1).

The molecular interaction is quantified by a relatively complex equation that is heavily utilized through the entire docking simulation, typically requiring 2,500,000 energy calculations. Several docking runs are required for a virtual drug candidate screening, during which ligands of a large compound database are docked one by one to the target. This complex process thus benefits from parallelization based run-time acceleration.

There are many docking software tools available either as proprietary or open-source. AutoDock, released as open source under GNUGPL and developed by the Scripps Research Institute, is the most cited docking software according to the ISI Web of Science database [13]. From an availability perspective, it is still preferred over proprietary tools. Another open-source docking tool called AutoDock Vina provides a more recent alternative to AutoDock. While Vina already exploits multicore parallelism and can be up to two orders of magnitude faster than its predecessor, AutoDock is still widely used as it provides a wider range options controlling the simulation. This includes user-calculated maps, more extensible physics-based force fields, and more control of the simulation details (explicit water molecules, mutable atoms, etc.). Also, the energy calculation functions used by AutoDock and Vina are different (molecular mechanics vs. knowledge-based), with Vina being less precise (standard error [in Kcal mol^{-1}]: 2.52 for AutoDock vs. 2.85 for Vina). Therefore, any of these tools may be preferable for a specific problem [15].

As OpenCL supports a wide range of applications through a portable abstraction, we consider an OpenCL implementation of

AutoDock relevant as this can facilitate the portability of this application to different computing systems (e.g. multi-core CPU, GPU, FPGA) with minimal code modifications.

In this article, we present an OpenCL implementation of AutoDock, and a corresponding performance and energy evaluation on two different platforms based on CPUs and GPUs. The remainder of this article is organized as follows: Section 2 summarizes the main MD concepts. Section 3 provides an overview of current state of the art concerning hardware acceleration of AutoDock. Section 4 provides details on the employed parallelization strategy and the performed code optimizations. Section 5 reports the experimental evaluation consisting of a preliminary validation of functional correctness, followed by the measurement of execution times and energy. Section 6 concludes and looks out to future work.

2 MOLECULAR DOCKING

Molecular Docking is an optimization problem that suffers from combinatorial explosion due to the many degrees of freedom of molecules, i.e. all possible positions, orientations and conformations. A number of heuristics have been applied to systematically search this solution space. One of these, Genetic Algorithms (GA), are inspired by biological evolution processes. In the their use for MD, the state variables of a ligand are defined by a set of values describing its translation, rotation and conformation with respect to the receptor. In GA, each state variable corresponds to a gene and the ligand'state corresponds to a genotype. Any legal binding pose between two molecules corresponds to an individual (also referred to as an entity), which in turn is represented by its genotype. All calculated poses conform a population. New populations are generated by mating individuals through a crossover operator. The offspring may experience gene mutation and be selected for the next generation. In order to evaluate and search for better binding poses (stronger individuals), AutoDock employs a scoring function and a search method.

A **scoring function** [8] models chemical interactions in order to quantify the free energy of a given arrangement of molecules. It uses a semi-empirical free-energy force field to evaluate molecule conformations during simulation. The force field V (Kcal mol^{-1}) is composed of four pair-wise energetic terms such as dispersion/repulsion, hydrogen bonding, electrostatics, and desolvation:

$$V = W_{vdw} \sum_{i,j} \left(\frac{A_{ij}}{r_{ij}^{12}} - \frac{B_{ij}}{r_{ij}^{6}}\right) + W_{hbond} \sum_{i,j} E(t)\left(\frac{C_{ij}}{r_{ij}^{12}} - \frac{D_{ij}}{r_{ij}^{10}}\right) + W_{elec} \sum_{i,j} \left(\frac{q_i q_j}{\epsilon(r_{ij}) r_{ij}}\right) + W_{sol} \sum_{i,j} (S_i V_j + S_j V_i) e^{\frac{-r_{ij}^2}{2\sigma^2}} \quad (1)$$

The dimensionless weighting constants W_{vdw}, W_{hbond}, W_{elec}, and W_{sol} are empirically determined using linear regression from a set of receptor-ligand complexes with known binding constants. The following constants depend on the atom types: A_{ij} (Kcal mol^{-1} Å^{12}) and B_{ij} (Kcal mol^{-1} Å^{6}) correspond to the Lennard-Jones (12-6) potential between neutral atoms i and j; C_{ij} (Kcal mol^{-1} Å^{12}) and D_{ij} (Kcal mol^{-1} Å^{10}) correspond to the hydrogen bonding (12-10) potential between hydrogen atom and hydrogen-bond acceptor 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. E(t) is a directional weight function (dimensionless) 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} (Coulomb's law) , the interatomic distance between atoms i and j. The whole interaction comprises the summation performed over all pairs of ligand and receptor atoms.

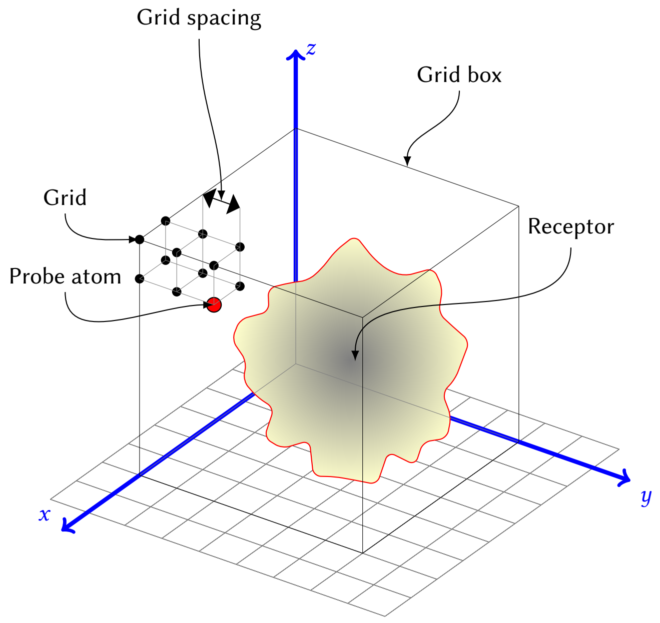

Figure 2: AutoDock Grid Map used for calculating the intermolecular energy.

Based on the interaction between molecules, the force field can be represented as a contribution of three energy components: intramolecular energy of ligand and receptor, and an intermolecular energy between both molecules. The intramolecular energy of the ligand can be calculated directly using equation Eq. 1. The intramolecular energy of the receptor is constant since this is treated as a rigid molecule. This allows minimizing calculations because a given molecule can contribute to the force field by itself only if the difference between energies of its bound and unbound states is different than zero. Similarly, the intermolecular energy could be computed also using equation Eq. 1. However, in order to evaluate it rapidly (even for a typically large number of ligand-receptor atom pairs), Eq. 1 is replaced by a trilinear interpolation based on pre-calculated grids (Figure 2) that model the energy contribution of the receptor for each ligand atom-type [8].

A **search method** seeks the global minimum of the scoring function, i.e., the predicted binding pose. AutoDock provides different search methods that can be classified as *global*: Simulated Annealing (SA), Distributed SA [3], Genetic Algorithm (GA); *local*: Solis & Wets; and *hybrid*: Lamarckian GA (LGA) [8]. As depicted in Figure 3, the LGA is basically a global-local optimization algorithm that results from the combination of GA and Solis & Wets. The GA itself is a global method as it generates new entities, and selects the stronger ones from the whole population that survive through generations. LGA is also a local method as it subjects a population subset of user-defined size to an adaptive-iterative process that improves (lower is better) the energy of the randomly chosen entities. This process takes the genotype of an entity and generates a new genotype by adding small deviations to it. The entity's energy corresponding to the new genotype is compared to the original one. If the energy is not improved, deviations are subtracted instead of being added, and another comparison is performed. On each iteration, the deviation variance is adapted depending on the number of successful or unsuccessful search attempts. In both, global and local search methods, the selection of entities is based on their energies, as shown in Listing 1. The LGA was proven to the be more efficient and reliable than the other methods [8], which is why we picked it for the proposed OpenCL implementation.

```
lamarckian_genetic_algorithm {
  while stop condition is false {
    // global
    genetic_generation(population);

    // local
    for every entity in population subset
      genotype = get_genotype(entity);
      local_search(genotype);
  }
}

local_search (genotype) {
  while stop condition is false {
    deviation = create_deviation(devpar);
    newgenotype1 = genotype + deviation;

    if (energy(newgenotype1) < energy(genotype))
      success++; fail = 0;
    else
      newgenotype2 = genotype - deviation;

      if (energy(newgenotype2) < energy(genotype))
        success++; fail = 0;
      else
        success = 0; fail++;

    devpar = change_deviation_variance(success,fail);
  }
}
```

Listing 1: Pseudo-code of local search in the LGA.

3 RELATED WORK

There are several reported MD implementations on GPUs, FPGAs, and multi-core CPUs. Here we report the most revelant ones.

Altuntaş et al. [1] present their own MD algorithm, implemented with the Heterogeneous Programming Library (HPL). The most computationally expensive part of this algorithm is made of a consumer-producer chain of subroutines: `populate()`, `score()`, `tournament()`, `mate()`, and `mutate()`, as shown in Listing 2. The total number of threads is equal to the population size. Within each subroutine, the parallelization is exploited by many active threads, whose number depends on the number of entities processed by the subroutine. The search method is only global and it is provided by the GA itself. In the experimental part, it performed speedup tests for a different number of docking runs (25, 50, 100) using only three chemical compounds which differ slightly from each other in the number of torsions (7, 5, 8), and the number of atoms (25, 19, 28). It achieves a speedup of around ~14x using a Tesla C2050/C2070 GPU with respect to a single 2.13 GHz Xeon CPU core.

```
// N: size of population
population = populate(ligand);   // N   active threads
for every generation {
    score(population, receptor); // N   active threads
    tournament(population);      // N/4 active threads
    mate(population);            // N/8 active threads
    mutate(population);          // N   active threads
}
```

Listing 2: Pseudo-code of genetic algorithm in [1].

Pechan et al. [10] present a CUDA implementation of AutoDock composed of two main kernels that perform `genetic_generation()` and `local_search()`. Both kernels utilize the same energy function but differ on how the degrees of freedom are generated. Basically, CUDA thread blocks and threads were used for exploiting parallelism at different levels: high (HLP), medium (MLP), and low (LLP). More specifically, entities were assigned to thread blocks, while an active thread within a block performs many separate tasks for each gene during energy calculation (Listing 3). In order to speedup the GA, a different selection scheme, i.e. binary tournament, is used instead of the original proportional selection. Moreover, due to the fact that local search is applied only to a selected subset (typically 6%) of the population, not all GPU multiprocessors are active during this phase. In the experimental part, speedup tests were performed for a different number of runs (20, 40, 60, 80, 100) using a set of 60 molecules. Performance gains of ~30x and ~65x using a GeForce GTX 260 GPU compared to a 3.2 GHz Intel Xeon CPU are reported.

```
for every docking run {                     // HLP
    while stop condition is false
        genetic_generation();
        local_search();
}

genetic_generation {
    for every new entity                    // MLP
       for every degree of freedom          // LLP
          generate_dof();
       for every required rotation          // LLP
          rotate_atom();
       for every ligand atom                // LLP
          intermolecular_energy();
       for certain atom pairs               // LLP
          intramolecular_energy();
}

local_search {
    for every selected new entity           // MLP
       while local search stop condition is false
          for every degree of freedom       // LLP
             generate_dof();
          for every required rotation       // LLP
             rotate_atom();
          for every ligand atom             // LLP
             intermolecular_energy();
          for certain atom pairs            // LLP
             intramolecular_energy();
}
```

Listing 3: Pseudo-code of genetic algorithm in [10].

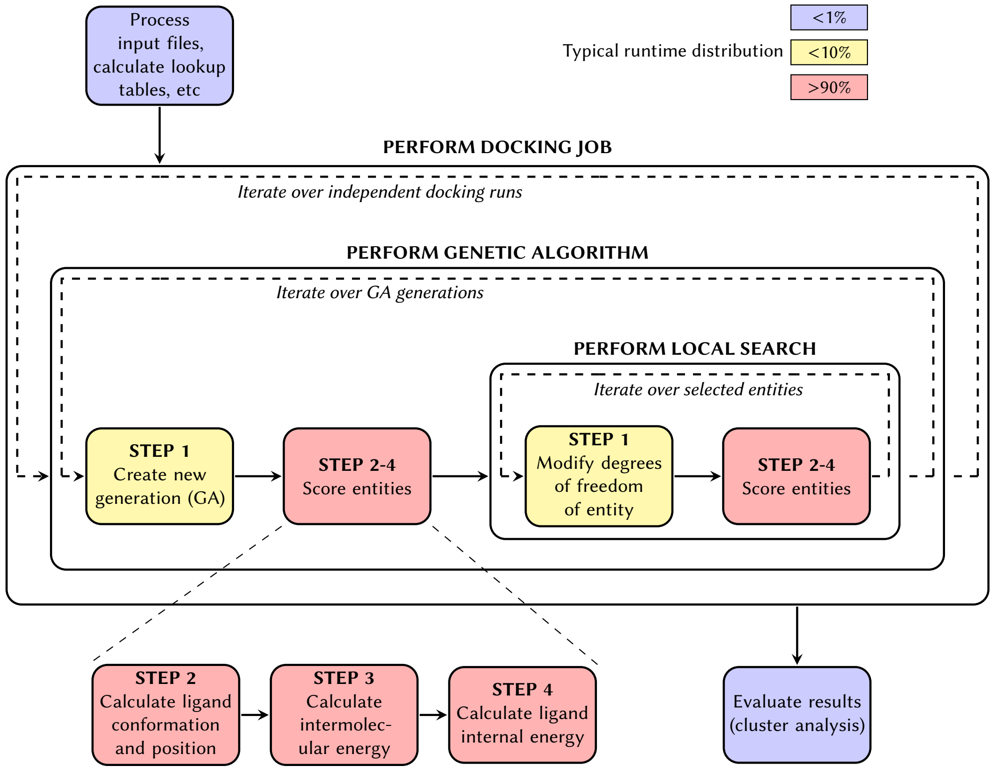
Process input files, calculate lookup tables, etc
Typical runtime distribution
<1%
<10%
>90%
PERFORM DOCKING JOB
Iterate over independent docking runs
PERFORM GENETIC ALGORITHM
Iterate over GA generations
PERFORM LOCAL SEARCH
Iterate over selected entities
STEP 1 Create new generation (GA)
STEP 2-4 Score entities
STEP 1 Modify degrees of freedom of entity
STEP 2-4 Score entities
STEP 2 Calculate ligand conformation and position
STEP 3 Calculate intermolecular energy
STEP 4 Calculate ligand internal energy
Evaluate results (cluster analysis)

Figure 3: AutoDock Block Diagram.

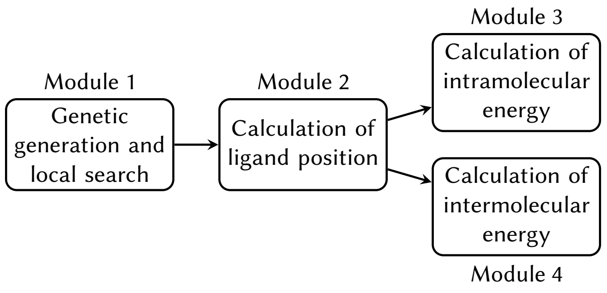
Module 1
Genetic generation and local search
Module 2
Calculation of ligand position
Module 3
Calculation of intramolecular energy
Calculation of intermolecular energy
Module 4

Figure 4: Pipeline processing of genetic algorithm in [12].

Pechan et al. [12] came up with an FPGA implementation. The hardware architecture consists of a three-stage pipeline composed of four modules (Figure 4). Specifically, the first module includes a complex state machine for controlling the genetic generation and local search. The second module calculates the position of the ligand atoms using the input gene values. Moreover, the third stage is composed of modules three and four, that calculate the internal-ligand and the intramolecular energy, respectively. All modules consist of parallel and fine-grained pipelines. The fourth module is the bottleneck of the design since it requires reading grid maps from off-chip memory. Their experiments were performed under the same docking settings as in [10] using a Xilinx Virtex-4 FPGA mounted on a SGI RASC RC100 module. It achieves ~23x speedups over a 3.2 GHz Intel Xeon CPU.

A hybrid parallelization of AutoDock targeting clusters was proposed by Norgan et al. [9]. This approach uses MPI (Message Passing Interface) and OpenMP (Open Multi-Processing) at two different levels to accelerate virtual screening. MPI is used to parallelize the `main()` function by distributing docking jobs across a system, while OpenMP enables multithreading of the LGA. The results showed that the design scales almost linearly up to 8192 CPU cores, reaching speedup values of ~8192x over a single CPU. The evaluation system was composed of a IBM BlueGene/P system and a 32-core IBM POWER7 server.

By reviewing the literature, we observed that there is no OpenCL implementation of AutoDock. The closest attempt is the work in [1] that is based on HPL, which in turns acts as a front-end of OpenCL. It proposed a non-standard docking method and was validated under a very reduced set of compounds (only three). In contrast to other CUDA implementations as [7] and [14], the parallel version of AutoDock in [10] includes the local search, and a validation using a reasonably large (60) set of compounds. Despite these important features, the lack of portability is its main drawback. Finally, another important factor not addressed in all existing parallel implementations of AutoDock, is the energy consumption by both serial and accelerated versions.

4 PARALLELIZATION

4.1 Parallelization opportunities

The hotspots of AutoDock were already identified in different studies. As already reported in in [11], the AutoDock parallelization can be exploited at three different levels: high, medium and low, as shown in Listing 3. In these approaches, the key concept is the entity, i.e., an individual member of a population of possible binding poses. In programming terms, this can be simply represented as a variable holding a collection of position, angle, and rotation values of the ligand.

The high-level parallelization is the most trivial one. A single docking job, i.e., a simulation corresponding to a given ligand-receptor pair, executes up to 100 docking runs. These runs are completely independent and can be executed simultaneously. For every docking run, new entities are created according to the GA or local search rules. The medium-level parallelism can be expressed by processing these entities concurrently. Additionally, entities are generated by a sequence of three steps: calculation of ligand conformation and position, inter- and intra-molecular energy calculation. This sequence is common for both the GA and the local search. However, the degrees of freedom or genes are generated differently. The GA is applied on every entity, while the local search is applied only on randomly selected ones. The low-level parallelism is present within the three steps, allowing the parallel execution of the following: generation of genes of a new single entity and rotation of different atoms, the intermolecular energy contributions of different ligand atoms, and the intramolecular energy contributions of different ligand atom-pairs.

4.2 OpenCL implementation

Our design is based on the work in [12]. The main difference is the programming language used and the associated optimizations for improving performance. While the reference implementation is based on CUDA, ours is an OpenCL version, that due to the language portability, allows switching easily between CPU and GPU as kernel-execution device.

The main idea behind the OpenCL data-parallel approach is to assign AutoDock main functions and data to OpenCL processing elements (kernels, work groups, work items) according to the associated level of parallelism. For instance, the genetic algorithm and local search are functions composed of main computation steps: generating or modifying new entities, and scoring them (Figure 5). Therefore, they are processed by different kernels, **K_GENETIC_GENERATIONAL** and **K_LOCAL_SEARCH** respectively, that execute sequentially within a given docking run. The execution of docking runs and the processing of entities are controlled by nested loops in the original AutoDock. In the parallel version, these loops were merged into a unique one, so that several entities of different runs can be processed simultaneously by work-groups within each kernel, thus achieving high and medium level parallelism. On the other hand, entities are processed by fine-grained tasks than can be assigned to work items, thus achieving low-level parallelism (Listing 3). The kernels communicate entities and corresponding energy values through the globally shared memory. Moreover, both kernels utilize the same energy calculation differing only in the rules for generating entities.

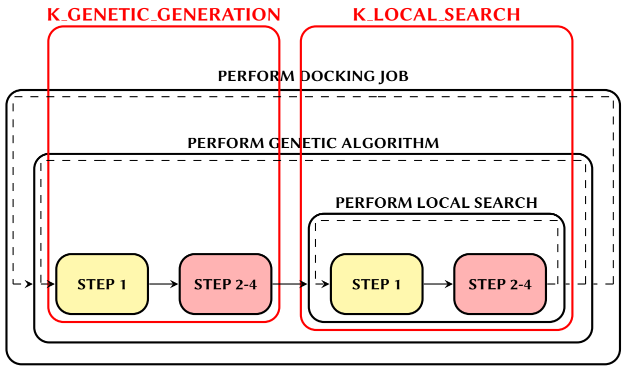

Figure 5: Kernels that perform genetic generation and local search.

4.3 OpenCL code optimizations

The host code is in charge of the system initialization and general management. The program starts by reading input files that can be grouped based on content type. First, files containing docking parameters such as the number of docking runs, termination criteria based on maximum number of energy evaluations and generations, number of local search iterations, etc. Second, we read files with information on molecules such as initial spatial coordinates, electric charges, atom types and grid values (Listing 4).

```
// 2cpp.pdbqt
REMARK  0 active torsions:
REMARK  status: ('A' for Active; 'I' for Inactive)
ROOT
HETATM    1  C1  CAM A 422      46.508  44.528  14.647  1.00 16.04     0.024 C
HETATM    2  C2  CAM A 422      44.913  44.451  14.586  1.00 17.04     0.136 C
HETATM    3  O   CAM A 422      44.197  44.497  15.571  1.00 17.69    -0.297 OA
HETATM    4  C3  CAM A 422      44.562  44.319  13.094  1.00 16.37     0.084 C
HETATM    5  C4  CAM A 422      45.987  44.434  12.432  1.00 16.57     0.001 C
HETATM    6  C5  CAM A 422      46.538  45.892  12.569  1.00 16.28     0.007 C
HETATM    7  C6  CAM A 422      46.856  45.977  14.100  1.00 15.77     0.018 C
HETATM    8  C7  CAM A 422      46.875  43.585  13.416  1.00 15.69    -0.029 C
HETATM    9  C8  CAM A 422      46.482  42.074  13.565  1.00 15.46     0.016 C
HETATM   10  C9  CAM A 422      48.396  43.529  13.036  1.00 16.20     0.016 C
HETATM   11  C10 CAM A 422      47.083  44.219  16.031  1.00 15.15     0.024 C
ENDROOT
TORSDOF 0

// 2cpp.gpf
npts 13 13 13                         # num.grid points in xyz
gridfld 2cpp_protein.maps.fld         # grid_data_file
spacing 0.375                         # spacing(A)
receptor_types A C H HD N NA OA SA    # receptor atom types
ligand_types C OA                     # ligand atom types
receptor 2cpp_protein.pdbqt           # macromolecule
gridcenter 46.436 44.132 14.279       # xyz-coordinates or auto
smooth 0.5                            # store minimum energy w/in rad(A)
map 2cpp_protein.C.map                # atom-specific affinity map
map 2cpp_protein.OA.map               # atom-specific affinity map
elecmap 2cpp_protein.e.map            # electrostatic potential map
dsolvmap 2cpp_protein.d.map           # desolvation potential map
dielectric -0.1465                    # <0, AD4 distance-dep.diel;>0, constant
```

Listing 4: Autodock input files of the 2cpp complex.

Upon initial docking setup, data for the actual docking is pre-processed before it is sent to the device memory. This is basically a re-arrangement of data into arrays that aim to maximize the data locality for improving memory access from the device. For instance, the coefficients of equation Eq. 1 can be grouped into arrays containing A-C and B-D values, as second coefficients (C, D) in each coefficient pair can be subsequently accessed after the first ones (A, B), depending on the presence of hydrogen bonding. Then, the OpenCL framework is set, i.e., the platform and device are identified, the device memory is allocated, and kernels along with their arguments and global/local sizes are defined. Afterwards, the re-arranged data is copied to device memory, and the kernels **K_GENETIC_GENERATION** and **K_LOCAL_SEARCH** execute until the termination criteria are met, i.e., when the execution reaches either a maximum number of energy evaluations, or a maximum number of generations. Results are sent back to the host for cluster analysis, i.e. entities having similar conformations are clustered into groups listed in a energy-based descending order.

In order to increase execution performance compared to the reference design [10], some optimizations were performed on both host and device sides:

Size configuration of processing elements. The number of work groups per kernel is equal to the number of entities to be processed in all runs. As already described, all entities are processed in the genetic generation, while only a subset of the population is selected to undergo local search. Additionally, different size configurations of work groups lead to different performance results. Our criterion was based on the utilized device architecture. For instance, AMD GPUs, which execute wavefronts of 64 fragments, produce better results in configuration using 64 work-items. Similarly, a 16 work-item configuration is faster for the CPU case (see Section 5). This differs from the reference design, where CUDA blocks were fixed to 32 threads for all experiments.

Usage of native functions. OpenCL provides native versions for math functions. Specifically, these are used in the score calculations expressed by Eq. 1. As it is shown in Section 5, the quality of results is not diminished despite the lower accuracy, and the docking program can run faster speeding-up almost by ~2x with respect to using their full-precision counterparts. This is different from the reference design, which used full-precision arithmetic. However, it has already been demonstrated in [12] that using lower accuracy arithmetic in an FPGA-based accelerator does not decrease the quality of the actual docking solutions, so we can exploit this speedup potential in our OpenCL solution.

Optimization of grid calculation. The grid calculation applies a trilinear interpolation to grid values retrieved from the global memory at addresses corresponding to the ligand position. By using a mathematically equivalent formulation (eliminating redundant terms and better grouping of sub-expressions), we can reduce the number of multiplications needed in the calculation of the grid address from 24 in the reference design to just 5 in our implementation.

Minimization of host-device communication using memory mapping. The execution of the kernels **K_GENETIC_GENERATION** and **K_LOCAL_SEARCH** is monitored by the host within the main-computation loop according to the termination criteria. The number of evaluations corresponding to each run is stored by the device in its global memory. As the host monitors the docking progress on each cycle of a Lamarckian generation, it must read the evaluation counters from the device. Instead of explicit copies of status data from the device as in the reference design, we used instead clEnqueueMapBuffer to minimize the device-to-host copy latency which depends directly on the number of docking runs.

Table 1: Typical values of GA parameters

Population size	150
Max. number of energy evaluations	2 500 000
Max. number of generations simulated	27 000
Mutation rate	0.02
Crossover rate	0.80

Table 2: Typical values of local search parameters

Max. number of local search iterations	300
Local search rate	0.06
Lower bound of initial variance	0.01
Number of successes in a row before a change in the variance	4
Number of failures in a row before a change in the variance	4

5 EXPERIMENTAL EVALUATION

5.1 Test Description

From the many different protocols possible for validating the functional correctness of docking simulation, our experiment picks so-called *redocking studies*. In that approach, *already known* complexes are docked again. This allows a comparison between the well-known reference solutions and the results obtained by our MD implementation. A set of 20 ligand-receptor complexes used during tests were obtained from the Protein Data Bank (PDB) [2]. These were preprocessed before docking following the standard protocol using AutoDockTools [5]. AutoDockTools assists in preparing ligand and receptor files, annotating them with features required for AutoDock. For the ligand, the protocol consists of adding hydrogen atoms, removing water molecules, merging non-polar atoms, and choosing torsions. For the receptor, hydrogen atoms are added, and the grid box is defined manually.

Regarding the overall docking configuration, all parameters were set according to the default GA and local search values specified in Table 1 and Table 2, respectively. From a performance point of view, the most important parameters are the maximum number of energy evaluations and the maximum number of generations, because these control how long the docking program runs. For ligands with up to 10 torsions (as in our case), the work in [4] suggests to run the program until it reaches 250,000 - 25,000,000 energy evaluations, or 27,000 generations, whichever comes first. The local search variance specifies the local search space size to sample, i.e. the amount by which angle and torsion values change on every cycle of local search. The variance value can be initially specified by a user (typically 1), however it changes during local optimization depending on the success of the search.

The target system used in the experimental evaluation provides two processing elements: First, an Intel i5-6600K CPU clocked at 3.5GHz. Second, an AMD R9-290X GPU with 2816 multiprocessors and 44 active compute units. We use the CPU both to collect the baseline characteristics of the original serial implementation, as well as a target to execute the OpenCL-parallelized version.

Table 3: Functional correctness: Energy (Kcal mol^{-1})

Ligand-receptor complex	Serial Baseline	OpenCL CPU	OpenCL GPU
3ptb	−5.55	−5.55	−5.55
1stp	−8.37	−8.23	−8.32
4hmg	−3.68	−3.85	−3.95
3c1x	−13.61	−13.43	−13.29
3ce3	−11.59	−11.09	−11.08
3bgs	−6.68	−6.54	−6.59

Table 4: Functional correctness: Spatial Deviation RMSD (Å)

Ligand-receptor complex	Serial Baseline	OpenCL CPU	OpenCL GPU
3ptb	0.42	0.41	0.41
1stp	0.42	0.35	0.38
4hmg	0.97	0.87	0.81
3c1x	0.80	1.49	1.18
3ce3	0.93	0.87	0.77
3bgs	0.75	0.67	0.78

Table 5: Functional correctness: Size of best cluster

Ligand-receptor complex	Serial Baseline	OpenCL CPU	OpenCL GPU
3ptb	100	50	72
1stp	100	100	100
4hmg	34	48	51
3c1x	90	62	63
3ce3	94	70	71
3bgs	95	86	90

5.2 Validating functional correctness

These experiments aim to verify the correct operation of our MD implementation, accounting for three key aspects: energy, spatial deviation, and cluster size. Table 3 shows the energy values of the best poses obtained after 100 docking runs in all cases (lower is better). The spatial deviation is calculated as the Root Mean Square Deviation (RMSD), that measures the deviation of the resulting ligand conformation with respect to that of the initial one. Basically, it gives an estimation of the geometrical deviation considering for each ligand atom, its initial and final position coordinates. A commonly accepted criterion for considering a docking as successful, is a RMSD value smaller than 2 Å as shown in Table 4. Moreover, final ligand conformations, each corresponding to a docking run, are grouped into clusters according to a given RMSD tolerance (typically 2 Å). The sizes of the best clusters shown in Table 5, indicate how successful the re-docking was to find similar ligand conformations across different independent runs (higher is better). Similarly, a commonly accepted criterion for this is that the best cluster size should be at least the 25% of the total number of runs (25 over 100 runs in our case).

The most noticeable differences are shown in Table 4 and 5, where RMSD values (`4hmg`, `3c1x`) and best-cluster sizes (`3ptb`, `3c1x`, `3ce3`) differ considerably for both implementations. We attribute these variations to the selection scheme employed during genetic generation. AutoDock (serial) uses proportional selection, while our OpenCL implementation uses binary tournament. In **proportional selection**, entities are selected with a probability that is proportional to their energy values. Those entities with better energies have a higher probability of being chosen for the next generation of population. In **binary tournament**, two entities are selected randomly from the whole population, and these two compete against each other. The entity with the better energy is included in the next generation. The underlying advantage of proportional selection is that it preserves diversity in populations. However, if the initial population already contains one or two stronger entities (having lower energy), then these entities dominate, and prevent the whole population from exploring other potential solutions. A tournament scheme, however, is less susceptible to diversity loss, and provides better performance compared to proportional selection [16]. As a consequence, the more-diverse populations computed by from our OpenCL version, lead in most cases to less dense clusters, and to entities having different positions, i.e. having different (yet still valid) RMSD values. Additionally, the inherent randomness of each docking run, and the ligand size corresponding to each experiment, create a diversity of scenarios that contribute, depending on how the optimization randomly evolves, to results that may be both somewhat better (smaller RMSD, larger best cluster) or somewhat worse (bigger RMSD, smaller best cluster) compared to the results from the original AutoDock.

5.3 Performance results

Results reported in this section represent complete program executions for both serial and parallel versions. As already shown in Figure 3, the steps for reading input files and analyzing results requires less than 1% of the total execution time, and therefore do not impose a significant overhead. However, we observed that kernel building (online compilation of the OpenCL source code) can take a significant fraction of the entire execution time depending on the target device. For instance, considering the `3c1x` complex for a 32 work-items implementation executing 10 docking runs on the CPU, the kernel building time is ~13 s, representing ~9% of a corresponding total execution time of 146 s. In addition, kernel build times increased as we reduced the number of work items in our program on the CPU. For the case of `3c1x`, these were 14 s and 2 s for 16 and 64 work items, respectively. The building time duration for CPU was roughly the same for all compounds in our test set, depending only by the work-group size selected. On the other hand, for the GPU case, the building time was in all cases ~1 s, regardless of the work-group size.

Our performance results are grouped into CPU and GPU categories. For each of them, work-group sizes of 16, 32, and 64 work-items were tested, considering six compounds of different sizes: `3ptb` is the smallest (13 atoms, 2 rotatable bonds, 4 atom types), while `3c1x` is the largest one (46 atoms, 8 rotatable bonds, 6 atom types). Our first experiment was to determine the impact of the work-group size on the execution time for each computing platform.

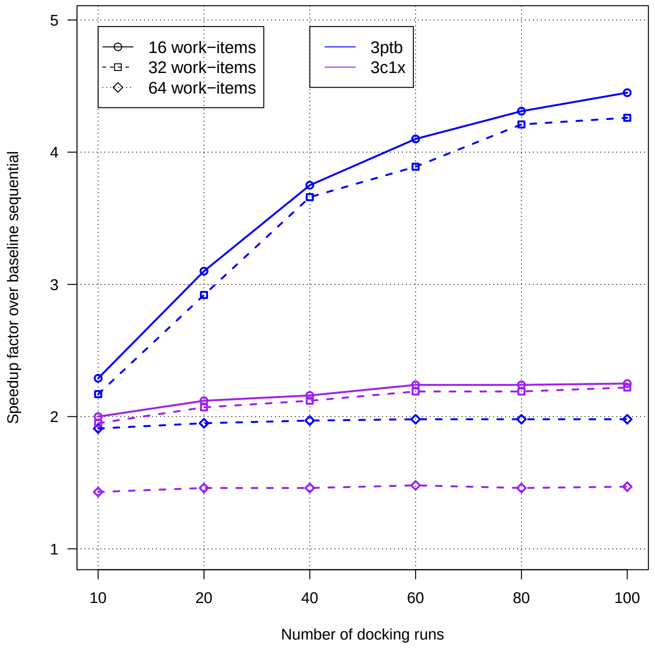

Figure 6: Achieved speedup on the CPU with different work-group sizes.

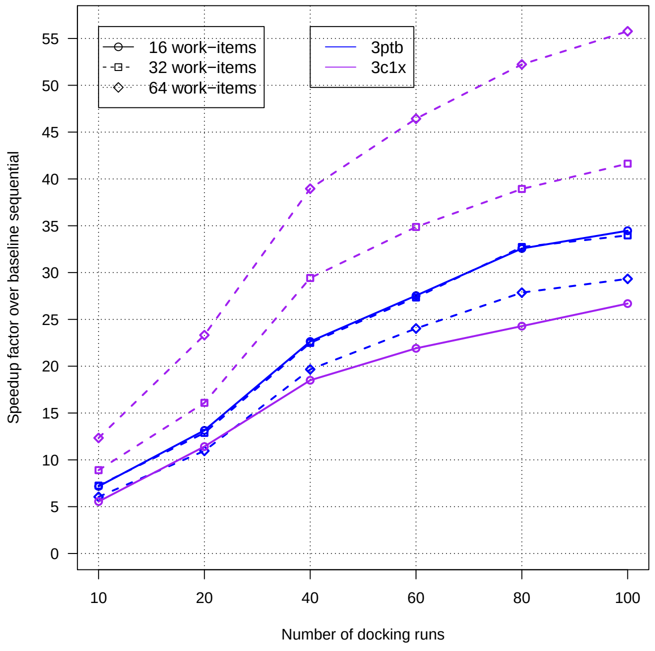

Figure 7: Achieved speedup on the GPU with different work-group sizes.

The tendency in most of cases, as depicted in Figures 6 and 7, is that better results are achieved with 16 and 64 work-items for CPU and GPU, respectively. The case of `3ptb` on the GPU is an exception, where a configuration of 16/32 compared to 64 work-items led to better speedups. This can be explained by the limited degree of parallelism provided by this very small molecule.

Then, using 16/64 work-group sizes for CPU/GPU, we analyzed the speedup behavior for different compounds. Figures 8 and 9 show that the achieved speedup varies between compounds. In particular, the algorithmic complexity is also dependent on the molecule size: first, more energy calculations are needed for compounds having more ligand atoms; second, a larger grid map is read for compounds having more receptor atoms; and finally, additional operations are performed for calculating the ligand position for compounds with more torsions. To illustrate this, consider that small compounds (`3ptb`, `1stp`) achieved better speedups than bigger ones (`3ce3`, `3c1x`) on the CPU. On the other hand, bigger compounds are executed faster on GPUs, since they provide more data parallelism than can be leveraged by the larger number of compute units: 44 on the GPU vs. 4 on the CPU. On our complete set of 20 compounds, the geometric mean of the speedup is ~3.3x and ~40.4x for CPU and GPU, respectively.

In order to evaluate the optimality of the achieved speedup, we investigated the utilization of computing resources by profiling the executing of 100 docking runs on the `3c1x` compound. For the CPU case (16 work-items), the average CPU utilization was ~97%, and the average DRAM throughput was just ~0.42%. For the GPU case (64 work-items), we observed cache hit-rates of ~84% for both kernels, as well as ~57% and ~20% of GPU time that the memory unit was active per each **K_GENETIC_GENERATION** and **K_LOCAL_SEARCH** execution, respectively. The higher memory-access rate that characterizes the genetic generation is due to the required creation of new entities for the next population. Specifically, this kernel must access data of the entire current population stored in the external memory at the beginning/end of the genetic generation, as well as of grid maps and intramolecular weights for calculating the energy of entities. On the other hand, **K_LOCAL_SEARCH** consists itself an iterative process consuming ~95% of total GPU execution time. However, its memory access rate is much lower than **K_GENETIC_GENERATION** since it does not need to retrieve all entities, but only a fixed subset consisting of 6% of the population. These findings show that the performance of this application is limited by the speed of the compute units. The best performance results are summarized in Table 6 and consistently show a higher speedup achieved by the GPU.

Table 6: Execution Time (s) and Speedup for 16/64 work-items on CPU/GPU

Ligand-receptor complex	Serial Baseline	OpenCL CPU	OpenCL GPU	Speedup CPU	Speedup GPU
3ptb	586.27	131.77	19.99	4.45	29.33
1stp	836.47	241.06	27.08	3.17	30.89
4hmg	1416.22	403.12	32.89	3.51	43.06
3c1x	2841.84	1265.72	50.96	2.25	55.77
3ce3	1867.69	617.00	36.15	3.03	51.67
3bgs	1102.88	312.20	28.29	3.53	38.98

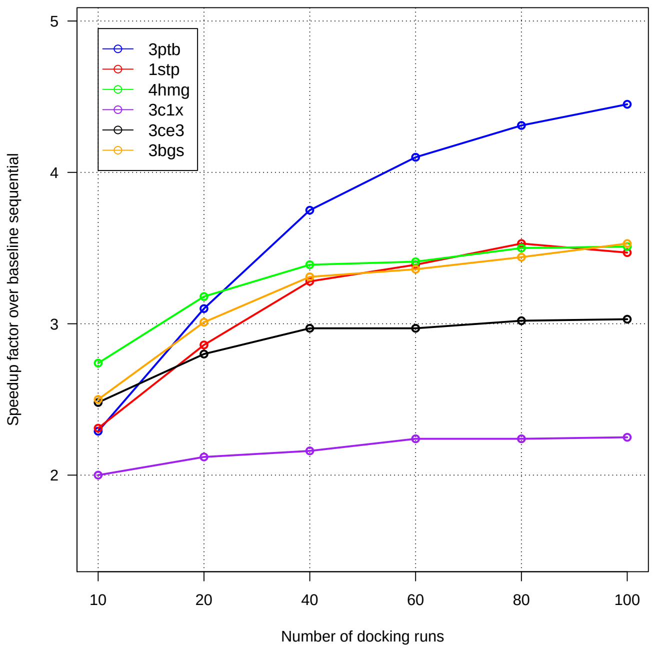

Figure 8: Achieved speedup on the CPU (16 work-items).

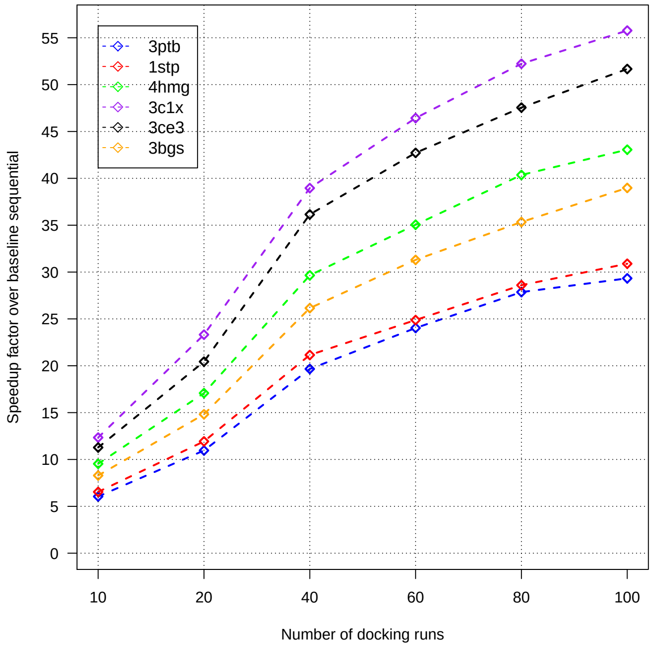

Figure 9: Achieved speedup on the GPU (64 work-items).

5.4 Energy consumption results

We compute the energy required for the different variants by sampling the drawn power in 50 ms intervals, using power performance counters both on the CPU and the GPU to avoid the inaccuracies typically associated with external measurements (e.g., shunt-based).

Table 7: Measured power values (W) on the CPU

Execution type	Power
Idle	~4
Baseline: single CPU core	~19
OpenCL: four CPU cores	~48

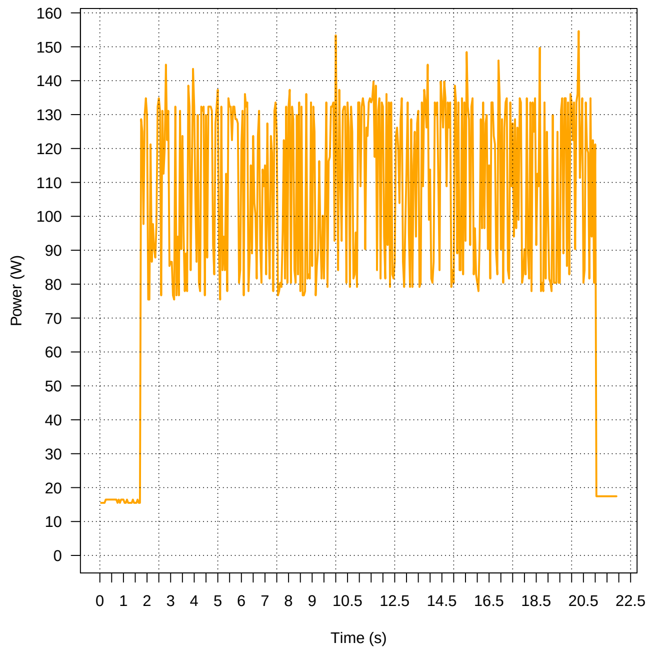

Figure 10: Power measurements on the GPU for 10 docking runs using 3c1x (idle power: ~17 W).

The power samples are then integrated over time to derive the energy.

Using this methodology, we discover that the power drawn by the CPU for the different scenarios (idle, baseline sequential, OpenCL-parallelized) stays mostly constant over the entire execution time of an MD run (Table 7). On the GPU, however, we observe power draws between 75 W ... 155 W over an execution. We attribute this different behavior to the algorithm switching between the **K_GENETIC_GENERATION** and **K_LOCAL_SEARCH** kernels, with the latter having a much lower degree of parallelism that the first. On the GPU, this will lead to some compute elements becoming idle (drawing less power). On the CPU, however, even this reduced degree of parallelism suffices to keep all cores and their internal ALUs/FPUs/LSUs busy, thus explaining the almost constant power draw.

Our energy consumption results are grouped into CPU and GPU categories. For each of them, work-group sizes of 16, 32, and 64 work-items were tested. Table 8 shows energy consumption for serial and parallel execution in the case of selected compounds for CPU and GPU. Despite the fact, that the GPU required higher amount of power than CPUs during certain time periods, it achieved greater energy savings than the most parallel version on the CPU.

Table 8: Energy consumption (KJ) results and Energy efficieny gain for 16/64 work-items on CPU/GPU

Ligand-receptor complex	Serial Baseline	OpenCL CPU	OpenCL GPU	Efficiency gain CPU	Efficiency gain GPU
3ptb	11.80	5.95	2.39	1.98	4.92
1stp	16.69	11.72	3.74	1.42	4.47
4hmg	28.07	19.43	4.81	1.44	5.84
3c1x	54.85	61.15	8.72	0.89	6.29
3ce3	36.27	30.39	5.84	1.19	6.21
3bgs	21.56	15.13	4.16	1.43	5.18

Figures 11 and 12 show the gain in energy efficiency for different docking runs. In both CPU and GPU cases, the efficiency gain behaves similarly as the speedup presented previously, depending as well on the compound's complexity and the work-group size. In particular, in the CPU accelerator, small compounds led to larger energy savings (~2x) compared to bigger ones (`4hmg`, `3ce3`), while the parallel execution using `3c1x` saved no energy with respect to the baseline case.

This seeming anomaly for `3c1x` can be explained by considering the execution time (baseline: 2841.84 s, OpenCL CPU: 1265.72 s) and their power measurements (baseline: ~19 W, OpenCL CPU: ~48 W) for 100 docking runs (Tables 6 and 7). OpenCL on the CPU reduced the execution time by a factor of ~2.2x, but this was accompanied by a power draw ~2.5x higher than for the serial baseline, thus leading to a deterioration of energy of efficiency for this experiment.

Considering our complete set of 20 compounds, the geometric mean of the energy savings compare to the sequential baseline is ~1.4x for the CPU, and ~5.4x for the GPU.

5.5 OpenCL on FPGAs

Initially, we had intended to complete this study by also including FPGAs as a compute platform for OpenCL-based acceleration. However, the OpenCL-to-FPGA compiler that we could employ had significant robustness problems. Code that ran perfectly on the CPU or the GPU often led to the tool crashing in compilation, truly excessive run-times (> week) and memory requirements (> 128 GB) during hardware generation, or crashes and execution errors on two different FPGA hardware platforms.

After much rewriting of the OpenCL code (and a number of compiler updates), we finally arrived at an implementation that actually executed correctly on the FPGA targets. However, it suffered from a severe slowdown over the sequential baseline, in the range of three orders of magnitude. This slowdown could not be alleviated even when employing a completely different parallelization strategy, using vendor-specific pragmas to formulate a task-parallel pipeline, instead of the data-parallel approach we used for CPU and GPU.

As we attribute these difficulties to the still somewhat unstable nature of the specific OpenCL-FPGA compiler available to us, we will examine the use of *other* OpenCL-to-FPGA compilers to repeat these experiments (see next section).

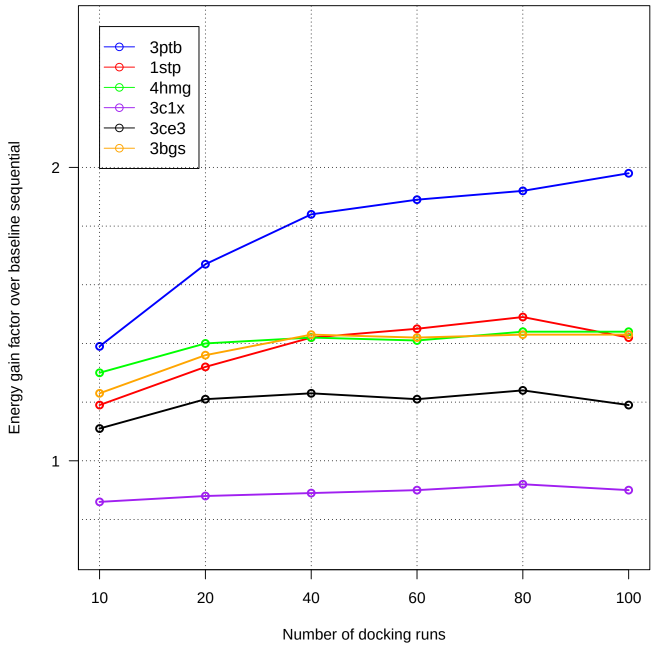

Figure 11: Achieved energy efficiency gains on the CPU (16 work-items).

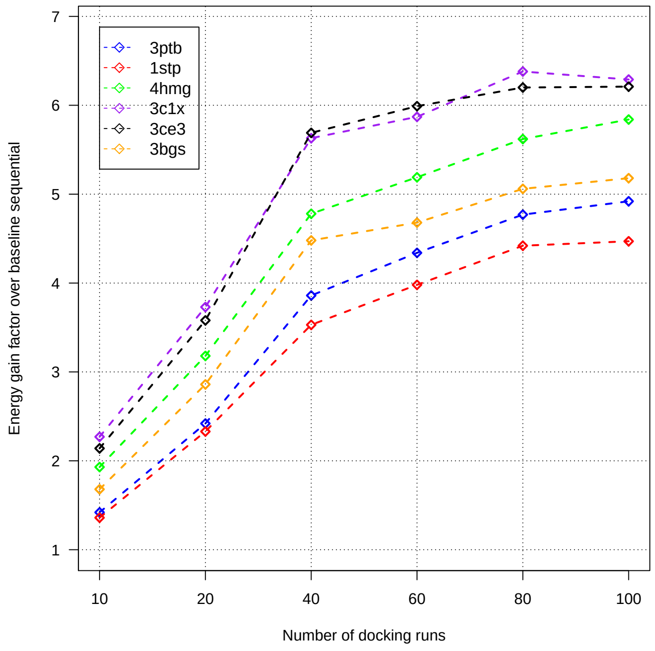

Figure 12: Achieved energy efficiency gains on the GPU (64 work-items).

6 CONCLUSIONS AND FUTURE WORK

Evaluating our docking code with only minor modifications on different platforms was possible due to the inherent portability of OpenCL. Our test consisted of different number of docking runs, each for different work-group sizes, using a set of 20 compounds of different sizes, to examine the behavior of our parallelized code under different conditions. For CPU and GPU, respectively, the speedups reached maximum values of ~4x and ~56x, with geometric means of ~3.3x and ~40.4x. Additionally, energy consumption savings reached values of ~2x and ~6.2x, with geometric mean of ~1.4x and ~5.4x, on the CPU and GPU, respectively. This work shows that the performance and energy consumption of a molecular docking program can be significantly enhanced by parallelizing using OpenCL.

In future work, we will both examine the newer Vina MD algorithm, as well as make another attempt to employ FPGAs as compute targets (employing a newer version of the OpenCL compiler as well as also considering completely different compilers, such as [6]).

ACKNOWLEDGMENTS

The authors would like to thank the German Academic Exchange Service (DAAD) and the Peruvian National Program for Scholarships and Educational Loans (PRONABEC) for financial aid, as well as AMD Inc. for technical support.

REFERENCES

[1] Serkan Altuntas, Zeki Bozkus, and Basilio B. Fraguela. 2016. *GPU Accelerated Molecular Docking Simulation with Genetic Algorithms.* Springer International Publishing, Cham, 134–146. DOI: http://dx.doi.org/10.1007/978-3-319-31153-1_10

[2] H.M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T.N. Bhat, H. Weissig, I.N. Shindyalov, and P.E. Bourne. 2000. The Protein Data Bank. *Nucleic Acids Research* 28, 1 (January 2000), 235–242. https://www.rcsb.org.

[3] David S. Goodsell and Arthur J. Olson. 1990. Automated docking of substrates to proteins by simulated annealing. *Proteins: Structure, Function, and Bioinformatics* 8, 3 (1990), 195–202. DOI: http://dx.doi.org/10.1002/prot.340080302

[4] Csaba Hetenyi and David van der Spoel. 2002. Efficient docking of peptides to proteins without prior knowledge of the binding site. *Protein Science* 11, 7 (2002), 1729–1737. DOI: http://dx.doi.org/10.1110/ps.0202302

[5] Ruth Huey, Garret M. Morris, and Stefano Forli. 2012. *Using AutoDock 4 and AutoDock Vina with AutoDockTools: A Tutorial.* https://autodock.scripps.edu/faqs-help/tutorial/using-autodock-4-with-autodocktools/2012_ADTtut.pdf.

[6] Jens Huthmann, Björn Liebig, Julian Oppermann, and Andreas Koch. 2013. Hardware/Software Co-Compilation with the Nymble System. In *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2013 8th International Workshop on.* IEEE, 1–8.

[7] S. Kannan and R. Ganji. 2010. Porting Autodock to CUDA. In *IEEE Congress on Evolutionary Computation.* 1–8. DOI: http://dx.doi.org/10.1109/CEC.2010.5586277

[8] Garrett M. Morris, David S. Goodsell, Robert S. Halliday, Ruth Huey, William E. Hart, Richard K. Belew, and Arthur J. Olson. 1998. Automated docking using a Lamarckian genetic algorithm and an empirical binding free energy function. *Journal of Computational Chemistry* 19, 14 (1998), 1639–1662. DOI: http://dx.doi.org/10.1002/(SICI)1096-987X(19981115)19:14⟨1639::AID-JCC10⟩3.0.CO;2-B

[9] Andrew P. Norgan, Paul K. Coffman, Jean-Pierre A. Kocher, David J. Katzmann, and Carlos P. Sosa. 2011. Multilevel Parallelization of AutoDock 4.2. *Journal of Cheminformatics* 3, 1 (2011), 12. DOI: http://dx.doi.org/10.1186/1758-2946-3-12

[10] I. Pechan and B. Feher. 2011. Molecular Docking on FPGA and GPU Platforms. In *2011 21st International Conference on Field Programmable Logic and Applications.* 474–477. DOI: http://dx.doi.org/10.1109/FPL.2011.93

[11] Imre Pechan and Bela Feher. 2012. Hardware Accelerated Molecular Docking: A Survey, Bioinformatics. (2012). DOI: http://dx.doi.org/10.5772/48125

[12] I. Pechan, B. Feher, and A. Berces. 2010. FPGA-based acceleration of the AutoDock molecular docking software. In *6th Conference on Ph.D. Research in Microelectronics Electronics.* 1–4.

[13] Sergio Filipe Sousa, Pedro Alexandrino Fernandes, and Maria Joao Ramos. 2006. Protein-ligand docking: Current status and future challenges. *Proteins: Structure, Function, and Bioinformatics* 65, 1 (2006), 15–26. DOI: http://dx.doi.org/10.1002/prot.21082

[14] Rene Thomsen and Mikael H. Christensen. 2006. MolDock: A New Technique for High-Accuracy Molecular Docking. *Journal of Medicinal Chemistry* 49, 11 (2006), 3315–3321. DOI: http://dx.doi.org/10.1021/jm051197e arXiv:http://dx.doi.org/10.1021/jm051197e PMID: 16722650.

[15] Oleg Trott and Arthur J. Olson. 2010. AutoDock Vina: Improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading. *Journal of Computational Chemistry* 31, 2 (2010), 455–461. DOI: http://dx.doi.org/10.1002/jcc.21334

[16] Jinghui Zhong, Xiaomin Hu, Jun Zhang, and Min Gu. 2005. Comparison of Performance between Different Selection Strategies on Simple Genetic Algorithms. In *International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA-IAWTIC'06)*, Vol. 2. 1115–1121. DOI: http://dx.doi.org/10.1109/CIMCA.2005.1631619