A Case Study in Using OpenCL on FPGAs: Creating an Open-Source Accelerator of the AutoDock Molecular Docking Software

Leonardo Solis-Vasquez and Andreas Koch
Technische Universität Darmstadt, Embedded Systems and Applications (ESA) Group, Darmstadt, Germany

Abstract

In recent years, OpenCL has been increasingly adopted as it enables software programmers to harness the performance and power efficiency of FPGAs. Despite simplifying the FPGA programming challenge, achieving high performance and energy efficiency with OpenCL is still a difficult task. In order to further contribute to the advance of the OpenCL usage for FPGAs, we utilize a realistic application scenario as our case study: the AutoDock molecular docking software. While OpenCL has proven its effectiveness in accelerating molecular docking on GPUs, for FPGA-based AutoDock accelerators it struggles with difficult design patterns. Besides complex multiple-producers to single-consumer datapaths, these include time-intensive loops with variable runtimes. Therefore, this work presents the design and optimization steps for implementing AutoDock in OpenCL targeting an Arria-10 FPGA, as well as a corresponding execution runtime and energy-efficiency evaluation. Applying these techniques improved the performance of the initial OpenCL implementation for FPGAs by three orders of magnitude, with the final version of the code now yielding speed-ups of up to ∼2.7x, and energy-efficiency gains of up to ∼1.8x over the original serial AutoDock version executing on a current-generation CPU.*

1 Introduction

In recent years, the adoption of High-Level Synthesis (HLS) approaches like OpenCL has increased significantly, as they enable programmers without a deep knowledge of the underlying FPGA architecture and Hardware Description Languages (HDLs) to harness the performance and power efficiency of such devices. Although the higher level of abstraction at the specification phase, achieving high-performance or energy efficiency is still challenging due to the lack of direct control of low-level characteristics such as resource usage, placement and timing constraints [1] [2]. Most recent studies in use of OpenCL for FPGAs comprise implementations of different applications [3] [4] [5], benchmarks [6] [7], and optimization analysis [2] [8]. As already stated in [9], there is still a gap in understanding which OpenCL constructs map well on FPGAs. This can be noticed in cases where the performance obtained with OpenCL is lower in comparison to what is achieved with RTL-level HDLs. Moreover, most of the applications studied were code fragments.

Consequently, in this paper, in order to further contribute to the advancement of the OpenCL usage for FPGAs, we chose a Molecular Docking (MD) application as a more realistic case study: the AutoDock software, one of the most cited MD tools according to the ISI Web of Science database [10]. AutoDock performs time-consuming calculations and search methods that require complex design patterns. Although our OpenCL version is not yet competitive in terms of performance with respect to the best HDL-based implementation of AutoDock [11], the design and optimization steps described in this paper will be useful in applying OpenCL to other more complex algorithms. Our main contributions are as follows:

- First, we present a detailed development and exploration of several design choices not extensively discussed in previous OpenCL studies on FPGAs, such as complex multiple-producers to single-consumer datapaths, as well as time-intensive loops with variable runtime. Our focus is on task parallelism by exploring a pipeline architecture and specific optimization techniques for FPGA-based accelerators.
- Second, to the best of our knowledge, we provide the first open-source OpenCL implementation of AutoDock for FPGAs in [12]. It achieves speed-ups of up to ∼2.7x with respect to the original single-threaded CPU version. Considering that our initial FPGA implementation, which was based on OpenCL code already performing very well on CPUs and GPUs ran almost 1000x *slower* than the original single-threaded software version of AutoDock, the improvements of using our suggested implementation patterns of OpenCL for FPGAs are demonstrated.

The remainder of this article is organized as follows: Section 2 introduces basic MD concepts. Section 3 presents the related work. Section 4 describes our implementation in terms of design phases and optimization techniques. Section 5 presents the experimental evaluation and discusses the runtime and energy efficiencies. Finally, Section 6 concludes the article.

*This work was supported by the German Academic Exchange Service (DAAD) and the Peruvian National Program for Scholarships and Educational Loans (PRONABEC) as the ALEPRONA funding program #57186883; as well as by Xelera Technologies by providing access to FPGA boards.

2 Background

Molecular docking simulations 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. A MD software is used to identify ligands that react as good inhibitors when interacting with a target receptor.

Predicting such predominant pose(s) involves solving an optimization problem that suffers from combinatorial explosion due to the many degrees of freedom of molecules, i.e., all possible positions, orientations and torsions. A number of heuristics have been applied to systematically search this solution space. Of these, Genetic Algorithms (GA) are inspired by biological evolution processes. In their use for MD, the state variables of a ligand are defined by a set of values describing its translation, rotation and torsion with respect to the receptor. In GA terms, each state variable corresponds to a gene and the ligand state corresponds to a genotype. Any legal binding pose corresponds to an entity, which in turn is represented by its genotype. All calculated poses form a population. New populations are generated by mating entities through a crossover operator. The offspring may experience gene mutation and be selected for the next generation. For evaluating and searching for better binding poses, i.e., *stronger entities*, AutoDock employs a scoring function and a search method.

The **scoring function** [13] models chemical interactions to quantify the free energy of a given arrangement of molecules. It uses a semi-empirical free-energy force field F (Kcal mol^{-1}) composed of four pair-wise energetic terms such as dispersion/repulsion, hydrogen bonding, electrostatics, and desolvation:

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

The dimensionless weighting constants W_{vdw}, W_{hb}, W_{el}, and W_{ds} are 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^{-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-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 directional weight (dimensionless) of the angle t that provides directionality from ideal hydrogen bonding geometry. Additionally, q_i and q_j are atomic charges, while $\varepsilon(r_{ij})$ is a dielectric function of r_{ij}, the interatomic distance between atoms i and j. The entire interaction comprises the summation performed over all pairs of ligand and receptor atoms.

Based on the molecules' interaction, the force field can be represented as the contribution of three components. First, the *intramolecular energy of the ligand* is calculated directly using (1). Second, the *intramolecular energy of the receptor* is constant since this is treated as a rigid molecule. Because a molecule can contribute to the force field by itself only if the difference between energies of its bound and unbound states is non-zero, this component is not calculated. Finally, the *intermolecular energy* could be computed also using (1). However, since the number of ligand-receptor atom pairs can be typically large (i.e., thousands), (1) is thus replaced by a fast approximation, i.e., trilinear interpolation of pre-calculated grids that model the contribution of the receptor for each ligand atom-type [13].

The **search method** minimizes of the scoring function, i.e., it seeks the lowest energy (highest score) that characterizes the strongest entity. From the various search methods provided by AutoDock, we picked the Lamarckian Genetic Algorithm (LGA) as it proved to be more efficient and reliable than other methods [13]. The LGA is a global method that generates new entities, and selects the stronger ones from the entire population that survive through generations. Additionally, the LGA is a local method as it subjects a population subset of user-defined size to an adaptive-iterative process that minimizes (improves) the energy of randomly-chosen entities (Listing 1).

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

    for entity in random-subset (population) // local
      local_search(get_genotype (entity));

    update (population);
  }
}

local_search (genotype) {
  while ls-stop-condition is false {
    delta = create_delta (step);
    newgenotype1 = add_on_every_gene (genotype, delta);

    if (energy (newgenotype1) < energy (genotype))
      genotype = newgenotype1;
      success++; fail = 0;
    else
      newgenotype2 = sub_on_every_gene (genotype, delta
     );

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

    step = update_step (success, fail);
  }
}
```

Listing 1 Pseudo-code of the LGA and local search.

This local search takes the genotype of an entity and generates a new genotype by adding small changes (delta) to each of its genes. The energy of the new genotype is compared to that of the original one. If the energy is not minimized, changes are subtracted instead of being added, and another energy comparison is performed. On each iteration, the change step is adapted depending on the number of successful or unsuccessful search attempts.

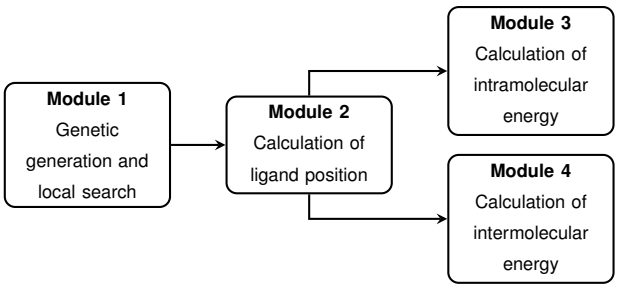

Figure 1 Pipeline processing of the LGA in [11].

3 Related Work

Relevant studies in **using OpenCL on FPGAs** are reported as follows. Yang et al. [9] examine design patterns made of a set-of-producer to a set-of-consumer datapaths in a molecular electrostatic application. Several Verilog and OpenCL versions with different arbitration and handshaking mechanisms are evaluated on an Arria-10 FPGA. Their results show that while Verilog versions achieve up to 80x of speedup factor over a single CPU core, OpenCL designs are 13x slower while using twice the resources when compared to Verilog ones. Moreover, Reza-Zohouri et al. [8] evaluate the performance and power-requirements of six Rodinia benchmarks targeting a Stratix-V FPGA against a Xeon CPU and a Tesla K20c GPU. The effectiveness of FPGA-specific optimizations, e.g., sliding windows, is reflected in a power-efficiency gain of 3.4x compared to the GPU, and better runtime and power efficiency over the CPU. The authors highlight that the OpenCL implementation of FPGA-specific strategies is completely different from common OpenCL strategies on GPUs.

Concerning **hardware acceleration of MD**, the survey in [14] provides a general overview of the applied parallelization on several MD applications. However, in this section we report only those corresponding to AutoDock. While there are several GPU implementations (in OpenCL [15], and in CUDA including [16] and excluding [17] local search), to the best of our knowledge there has been only one FPGA implementation of AutoDock published so far. Using Verilog, Pechan et al. [11] implemented an architecture that consists of a three-stage pipeline composed of four modules, each consisting of parallel and fine-grained pipelines. (Fig. 1). Specifically, Module 1 controls the genetic generation and local search. Module 2 calculates the position of the ligand atoms. The third stage is composed of Module 3 and Module 4, that calculate the intra- and intermolecular energies, respectively. Performance gains of $\sim$23x using a Virtex-4 FPGA compared to a 3.2 GHz Xeon CPU core are reported.

Additionally, Solis-Vasquez et al. [15] created an initial OpenCL implementation of AutoDock running on FPGAs as well. However, it suffered from a severe *slowdown* over the serial baseline (i.e., the original AutoDock running on a single CPU core) in the range of three orders of magnitude. In this article, we instead employ a task-parallel approach which we believe is more suitable to the underlying hardware. Furthermore, FPGA-specific optimization techniques are exploited. The patterns required to optimize our FPGA design made us consider several architectural and micro-architectural choices using the high-level abstractions of OpenCL. This evolved towards a parallel AutoDock implementation with improved runtime and energy-efficiency with respect to the serial baseline. The techniques we present here may also be beneficial when accelerating other applications.

4 Design and Optimization Methodology

The functionality of the serial baseline is represented in Fig. 2. First, the structural descriptions of both molecules as well as the docking parameters (e.g., number of docking runs, population size, number of energy evaluations and generations, etc) are read from input files. Second, for every docking run, a sequence of four main steps is executed during the global- and local-search. Step 1 generates new entities, represented by their genotypes, under different rules depending on the LGA phase being executed. During global search, genotypes are subjected to crossover, mutation, and selection operations. During local search, new genotypes are generated by adding small variations to their current values. The most computationally-intensive tasks ($>$ 90% of runtime) are represented as Steps 2, 3, 4 and calculate the ligand conformation, the inter- and intramolecular energy, respectively. This sequence is repeated for every docking run. Finally, once all runs have been executed, the results are clustered in order to assess the reliability of the docking simulation (Section 5.2).

4.1 OpenCL implementation

Since pipeline processing is more suitable for FPGAs, we adopted the design in [11] as our starting architecture (Fig. 1). This architecture executes the entire docking job sequentially, i.e., by starting a new docking run only after the previous one has finished, while pipelining the GA calculations within each run. From the programming perspective, such architecture is achieved by following a task-parallel approach, in which each task is coded as a single work-item kernel. The actual OpenCL implementation took place in the following four development phases, each describing the design and optimization steps applied incrementally over the previous ones. As described above, all of the source code is available at [12] for further study.

4.1.1 First development phase

Fig. 3 represents our initial OpenCL design. The Step 1 of the genetic-generation (GG) and local-search (LS) functions (Fig. 2) have been merged into a single GA kernel. The GA kernel controls the overall functionality of the system composed also of the Conform, InterE, and IntraE kernels that in turn correspond to Steps 2, 3, 4 (Fig. 2), respectively. The communication between all kernels is achieved through OpenCL pipes that serve as channels passing genotype data from GA to Conform (GG2C_genotype, LS2C_genotype), ligand-

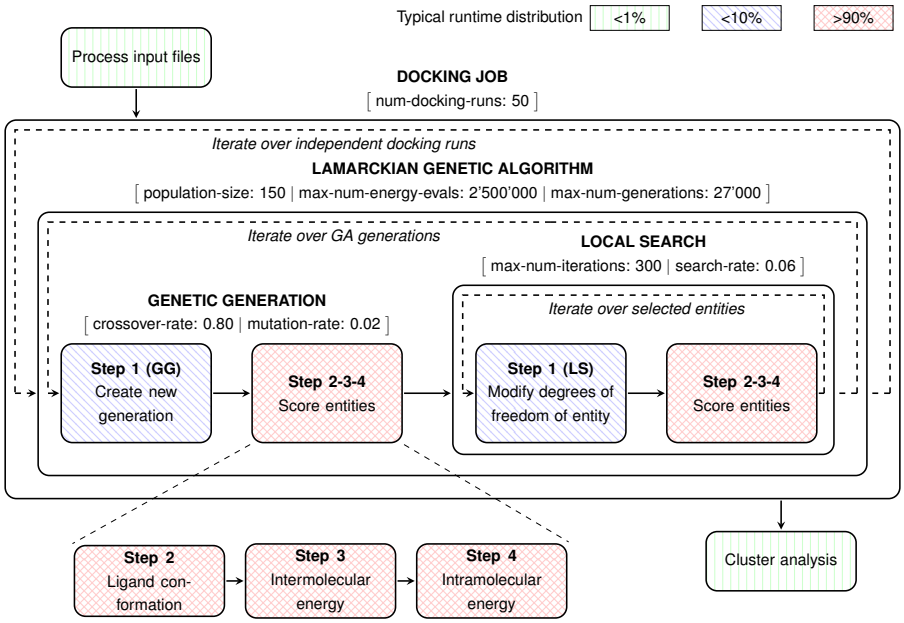

Figure 2 AutoDock Block Diagram [15] with default values of LGA parameters.

position data from Conform to InterE (C2IE_position) and to IntraE (C2IA_position), and the computed energy values from InterE (IE2GG_energy, IE2LS_energy) and IntraE (IA2GG_energy, IA2LS_energy) back to GA.

This design forms a closed loop consisting of kernels and channels that prevents the compiler from optimizing any channel depth [18]. As an attempt to avoid this scenario, we initially passed the feedback energies through global memory, by instantiating a Store kernel (not shown) in charge of receiving the computed energies from InterE and IntraE, and writing these into the off-chip memory, instead of sending them back to GA via channels. For this to work, we resorted to OpenCL fences (mem_fence(CLK_GLOBAL_MEM_FENCE)) on global memory accesses in Store and GA. Although this worked correctly in emulation, it did not work on the FPGA due to races in such global memory accesses, which were consistent only within a *single* kernel. In order to achieve a correct functionality on the FPGA, Store was removed while the feedback channels were included back in our design.

Another consideration regarding global memory was to minimize the number of accesses. Particularly, GA updates population data throughout an entire docking run, so storing populations strictly off-chip would result in significant lower performance. This was addressed by reading/writing from/to global memory only at the start/end of each docking run, while keeping intermediate populations on-chip using OpenCL 2D local arrays ([*population-size*: 150][*genotype-size*: 38]). Conversely, for read-only data such as grid maps, the number of global accesses could not be minimized due to the following two issues: first, the size of the grid data depends entirely on the docking space under analysis. As such, on-chip storage might not be sufficient for cases such as *blind* docking, where a map representing an entire receptor molecule would be required.

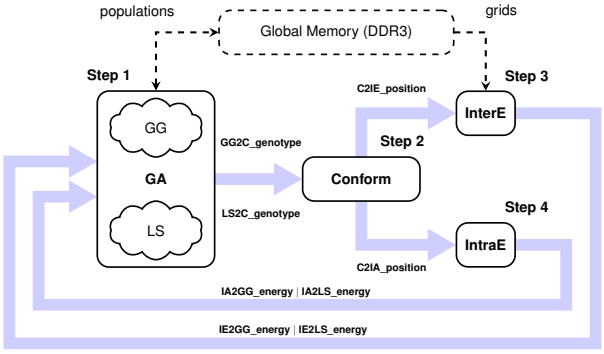

Figure 3 First development phase: initial OpenCL design.

Second, due to the irregular reads performed by InterE, any caching strategy resulted in significant misses. A further discussion of this aspect is provided in Section 4.2.

4.1.2 Second development phase

Since Conform, InterE, and IntraE turned out to be the major bottleneck, we optimized their microarchitecture separately. Each of these kernels was coded as a sequence of the following operations: *read from input-channels*, *main computation-loop*, and *write to output-channels*. These kernels always execute together as a chain of blocks, being invoked a number of times entirely controlled by GA, i.e., by the maximum number of either energy evaluations or generations. In order to support such termination criteria known only at runtime, all operations within these kernels were enclosed by a while-loop controlled by an *active* signal. Based on that, we aimed to minimize the initiation-interval (II, ideally =1) of each loop. Different techniques such as *shift registers*, *local-memory banking*, *unrolling of inner loops* resulted in significant reduction of data dependencies. In the case of Conform, we could not remove the data dependency created for keep-

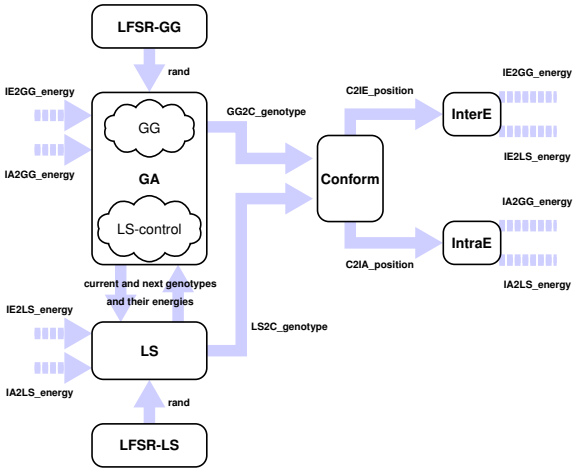

Figure 4 Second development phase: local-search logic is implemented as a separate kernel. From now on, feedback channels are shown as dashed connections, while global-memory accesses are omitted for simplicity.

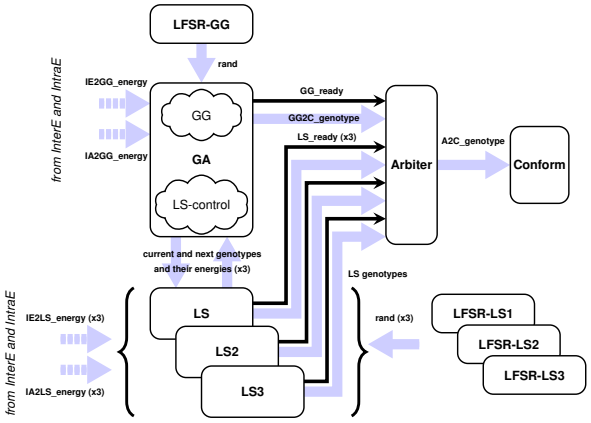

Figure 5 Third development phase: local-search kernels are replicated three times, while an arbiter is added to handle simultaneous energy-calculation requests.

ing track of the atoms to be rotated. Although this caused a high initiation interval (II=36) of the main computation-loop, the outer while-loop was fully pipelined (II=1). The InterE and IntraE kernels involve long latencies such as random accesses to off-chip grids, and single-precision floating-point calculations required for (1), respectively. Despite these, all their loops, i.e., outer while-loops and inner computation-loops, were fully pipelined.

The next optimizations performed on the GA kernel aimed to potentially increase the execution concurrency (Section 4.1.3), leading to the architecture depicted in Fig. 4. First, the local-search logic was moved out of GA and implemented as the LS kernel, where each LS-execution is triggered from within the LS-control loop. The LS kernel reads genotypes from GA, performs the energy minimization, and returns to GA new genotypes and their respective energies. Similarly to GA, LS sends genotypes via the LS2C_genotype channel to be evaluated by the Conform-InterE-IntraE chain. Moreover, the pseudo-random number generators initially implemented as inline functions in the genetic-generation and local-search logic, were converted into separated LFSR-GG and LFSR-LS kernels, each featuring a 32-bit linear feedback shift register.

4.1.3 Third development phase

Due to the iterative nature of the local search, we focused first on its microarchitectural optimization. Although code-refactoring guided by compiler suggestions [18] helped us to pipeline the majority of inner LS-loops, we could not pipeline its outmost loop due to dependencies created by genotype-data carried through inner loops, and channel invocations for energy calculation.

In order to compensate for this, we replicated LS, together with its LFSR-LS and channel interconnects, three times (Fig. 5). This architectural change has a particular consequence described as follows. As already described (Section 4.1.2), the initially single LS kernel invokes the execution of the Conform-InterE-IntraE block chain very much like GA during genetic generation. In this scenario, GA and LS act as producers, whereas Conform acts as a consumer of genotypes. Since the genetic-generation and local-search functions are mutually exclusive during GA execution, the arbitration in Conform was implemented in the previous phases simply as a pair of non-blocking channels, constantly guarding the status of both input channels, until any of them receives a complete genotype. On the other hand, when multiple LS kernels are instantiated (Fig. 5), multiple energy evaluations can be requested simultaneously resulting in a *multiple-producers to single-consumer* datapath where the aforementioned arbitration mechanism does not suffice. This was solved by inserting an Arbiter kernel that reads a *ready* signal along with its correspoding genotypes from each producer. The ready signals identify the actual producers whereas the genotypes corresponding to *valid* ready signals are accumulated using local arrays and dispatched in order towards Conform.

4.1.4 Fourth development phase

The LS kernel was further replicated as long as the resulting circuit fit on the target FPGA (Fig. 6). The replication factor was based on the upper bound of the LS-control loop, whose default value is determined by the number of entities that undergo local-search during a single GA evolution, i.e., nine entities that represent a random subset (6%) of the population size (150). As more LS instances imply fewer loop-rounds, the LS replication factor was extended from three (Section 4.1.3), up to five and nine in this phase.

Subsequently, Arbiter was optimized similarly as in [9], where only ready signals are passed through this kernel. Since many LS kernels can be simultaneously active, Arbiter queues at a given moment all producer IDs corresponding to valid ready signals into an array. These array values are sent sequentially to control the input multiplexer in Conform that selects incoming genotypes directly from a specific producer, instead of being accumulated and reordered through Arbiter as was done in Section 4.1.3. Furthermore, inferring a multiplexer in Conform instead of

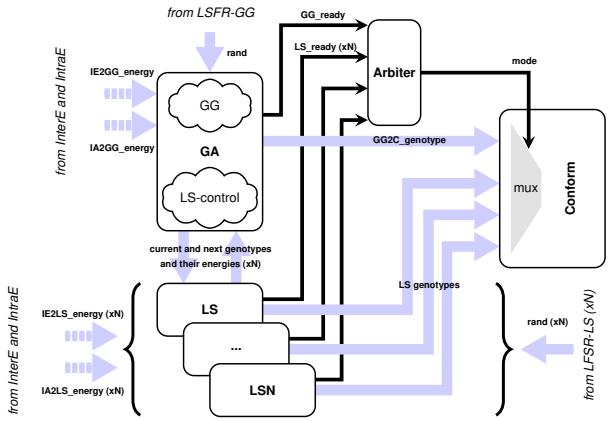

Figure 6 Fourth development phase: local-search kernels are further replicated, while the arbitration mechanism is improved.

Arbiter allows a deeper pipelining of the outermost-loop in Conform, while still achieving II=1.

4.2 Further optimization techniques

In addition to the FPGA-specific optimizations techniques [18] so far employed, we considered the following three in greater detail:

First, all kernel constants were calculated in the host and passed into kernels afterwards, e.g., scaled crossover, mutation and selection rates (GA), reference orientation-quaternions (Conform), offsets for indexing grid maps depending on constant grid sizes (InterE). This technique was applied throughout the entire OpenCL implementation.

Second, constant data was carefully allocated either into the `__constant` (on-chip cache, default size: 16 KB) or `__global const` (off-chip, maximum available: 16 GB) address space. If the size of the kernel's `__constant` arguments exceeds the cache size, accesses to them suffer from *larger* performance penalties compared to those of `__global const`, as the latter off-chip accesses are implemented with extra circuitry for tolerating longer latencies. Our look-up tables in Conform, InterE, and IntraE occupy only a total of 12 KB and can be allocated to on-chip constant memory without the risk of cache misses. On the other hand, larger data structures, such as rotation list, grid maps, and the list of intramolecular contributors (corresponding to Conform, InterE, and IntraE, respectively), were allocated to off-chip global memory with the `__global const` qualifier, as these altogether require ∼270 KB, and would lead to significant thrashing if allocated to the on-chip cache.

Finally, for most FPGA designs, fixed-point arithmetic leads to faster designs compared to their floating-point counterparts. However, for the InterE and IntraE kernels, floating-point resulted in an overall *faster* design, which can be attributed to the hardened floating-point DSP units in the Arria-10 FPGA [19]. On the other hand, for Conform, which initially suffered from a latency of II=36 (Section 4.1.2), a fixed-point representation reduced it down to II=10. This can be explained by the fact that the 30 addition/subtraction operations of the problematic datapath expressed in fixed-point were implemented using Adaptive Logic Modules (ALMs) instead of the DSPs used for the floating point version, thus avoiding the DSP latency of four clock cycles each [19].

The last two techniques were introduced in the third development phase and kept until the last one (Section 5.3).

5 Experimental Evaluation

5.1 Test description

From the many different protocols possible for the simulation, our experiment picks the so-called *redocking studies*. In this approach, *already known* complexes are docked again, allowing to test our design against well-known reference solutions. A set of five ligand-receptor complexes obtained from the Protein Data Bank (PDB) [20] were tested (PDB IDs: 3ptb, 1stp, 4hmg, 3ce3, 3c1x). All docking parameters were set to the default values as specified in AutoDockTools [21].

The hardware used for the serial baseline is composed of a 3.5 GHz i5-6600K CPU with 16 GB RAM, while the accelerator is a Gidel Proc10A card with an Arria-10 GX 1150 FPGA and 16 GB RAM. The Intel FPGA SDK for OpenCL v16.0 (latest version suported by the corresponding board support package) was used as OpenCL compiler.

5.2 Functional validation

Even though this study concentrates on the methodological aspects, we need to show that our parallelized version, especially with the changed arithmetic (mix of fixed and floating point) operates correctly.

Since previous acceleration studies of AutoDock [11] [15] [16] demonstrate that a reduced precision does not diminish the docking quality with respect to the original AutoDock (in double precision floating-point), we utilized a 16.16 fixed-point format for the LS, Conform, InterE, and IntraE kernels. This format allows simply using the OpenCL int and long primitive types, and was sufficient to represent genotypes and quaternions generated in LS and Conform. For the InterE and IntraE kernels, this format might not be sufficiently precise in cases where energies might reach out-of-bound values, i.e., when the dispersion/repulsion and hydrogen bonding terms grow rapidly as the interatomic distances become very short. However, the erroneous docking poses derived from these incorrect out-of-bound values are so bad that they will be discarded by the genetic algorithm anyway. While we did not observe such precision issues in practice, because the floating-point counterparts for the InterE and IntraE kernels actually were faster than fixed point (Section 4.2), we performed further experiments using floating-point representation in such kernels (Table 2).

Thus, the resulting designs from each development phase were compared against the serial baseline according to three key aspects: *energy*, *spatial deviation*, and *cluster size of resulting poses*. For the sake of simplicity, only the validation of our largest design (nine LS kernels, Sec-

Table 1 Comparison of energy and size of best cluster

PDB ID	# Atoms	# Torsions	Energy of best pose (Kcal mol^{-1})		Size of best cluster (100 docking runs)	
			Serial baseline	*OpenCL FPGA*	*Serial baseline*	*OpenCL FPGA*
3ptb	13	2	−5.55	−5.53	100	66
1stp	18	5	−8.37	−7.76	100	69
4hmg	27	10	−3.68	−4.11	34	25
3ce3	37	5	−11.59	−10.88	94	48
3c1x	46	8	−13.61	−12.61	90	22

Table 2 Development phases and design configurations

Develop. phase	Design config.	# LS replicas	Arithmetic representation		
			LS	*Conform*	*InterE IntraE*
First	DC1	1	float	float	float
Second	DC2	1	float	float	
Third	DC3	3	fixed	fixed	
Fourth	DC4a	5	fixed	fixed	
	DC4b	9	fixed	fixed	
	DC4c	9	float	fixed	
	DC4d	9	float	float	

Table 3 FPGA resource utilization and max. frequency

Design config.	ALMs Total: 427 200	RAMs Total: 2 713	DSPs Total: 1 518	Freq. (MHz)
DC1	129 301 (30%)	1 075 (40%)	388 (26%)	215.2
DC2	128 018 (30%)	999 (37%)	262 (17%)	174.4
DC3	158 586 (37%)	1 799 (66%)	548 (36%)	187.5
DC4a	177 509 (42%)	1 826 (67%)	586 (39%)	172.6
DC4b	222 372 (52%)	1 880 (69%)	662 (44%)	187.5
DC4c	220 427 (52%)	1 898 (70%)	659 (43%)	185.7
DC4d	219 359 (51%)	1 944 (72%)	383 (25%)	185.7

tion 4.1.4) is presented in Table 1. The energy values (lower is better) correspond to the best poses obtained after 100 docking runs. The spatial deviation is calculated using the resulting ligand conformation with respect to the initial one. As all obtained deviations meet the commonly accepted criterion for a successful docking (<2 Å), these are not reported. Moreover, final ligand conformations, each corresponding to a docking run, are clustered according to a typical spatial-deviation tolerance (2 Å). The sizes of the best clusters indicate how successful the re-docking was to find similar conformations across different independent runs (higher is better). The discrepancies in the cluster size are attributed to the different selection scheme used in this work (*binary tournament*), compared to the original one (*proportional selection*). As detailed in [15], a tournament scheme chosen for better performance leads to more diverse populations, leading in turn to less dense clusters. These smaller clusters obtained with the OpenCL version are acceptable since they meet a practical usage criterion of redocking ($\geq$ 20% of total docking runs).

5.3 Design configurations and resource utilization

Table 2 lists the four development phases and their respective design configurations (*DC1*, *DC2*, *DC3*, and *DC4* {*a*, *b*, *c*, *d*}) that summarize our most significant optimizations. Such designs differ in the number of LS kernels being replicated, i.e., *DC1* (one), *DC2* (one), *DC3* (three), *DC4a* (five), and *DC4* {*b*, *c*, *d*} (nine); as well as in the arithmetic representation for the listed kernels. Designs *DC4* {*b*, *c*, *d*}, all with nine replicated LS kernels, are employed to evaluate the impact of floating-point used in all replicas of LS (*DC4c*), as well as in Conform (*DC4d*), both compared to fixed-point (*DC4b*).

The largest designs, i.e., *DC4* {*b*, *c*, *d*}, are composed of 27 kernels each: one GA, nine LS, nine LFSR-LS, four LFSR-GG (used in selection, crossover, mutation, entity selection for local-search), one Arbiter, one Conform, one InterE, and one IntraE. Table 3 reports resource utilization in terms of ALM, RAM, and DSP blocks. The resource reduction obtained when moving from *DC1* to *DC2* can be attributed to the fact that implementing LS separately from GA removes the hardware required to carry genotype data in GG and LS, both initially managed within GA. On the other hand, there is an expected overall increase in resource usage when going from design *DC3* through *DC4a* towards *DC4b*, that directly corresponds to the increase of the LS-replication. Moreover, it is shown that a fixed-point representation of LS (*DC4b*), utilizes more DSP blocks (44%) than its floating-point counterparts such as designs *DC4c* (43%) and *DC4d* (25%).

Regarding the maximum frequency, designs *DC3* and *DC4* {*b*, *c*, *d*} reach comparable values ($\sim$186 MHz). Smaller designs (*DC2* and *DC4a*, both at $\sim$173 MHz) do not always result in higher frequencies compared to larger ones (*DC4* {*b*, *c*, *d*} at $\sim$186 MHz). Furthermore, we observed that higher frequencies do not necessarily imply faster circuits, e.g., *DC1*, capable of running at $\sim$215 MHz, is at least $\sim$4.4x slower than the serial baseline (Table 4).

5.4 Execution runtime and energy-efficiency results

Table 4 reports the full-program execution runtime for all proposed designs. On one hand, the first two designs are slower than the serial baseline. Analyzing the *DC2* performance with respect to that of *DC1*, there is an improvement only for the case 3ptb, which can be attributed to the lower frequency achieved (Table 3), and the increased computation required by larger PDB complexes with more than five torsions such as 1stp, 4hmg, 3ce3, and 3c1x. Although *DC2* seemed to be going in the wrong optimization direction, it introduced the architectural modifications (Section 4.1.2) that led to the improvements in later designs *DC3* and *DC4*. The significant runtime reductions when going from *DC2* to *DC3* (e.g., a maximum difference of $\sim$8100 s for 3c1x) are the result of the LS replication, as well as by the careful allocation of constant look-up tables (Section 4.2).

Table 4 Execution runtime and best speed-up for 100 docking runs

Execution runtime (seconds)					
Design config.	**Ligand-Receptor PDB ID**				
	3ptb	***1stp***	***4hmg***	***3ce3***	***3c1x***
Serial CPU	586	836	1416	1867	2841
DC1	2903	5784	6636	8519	12573
DC2	2550	6678	8121	9247	14502
DC3	376	739	1013	1364	1790
DC4a	315	563	788	1096	1496
DC4b	211	385	623	1077	1487
DC4c	215	388	634	1079	1491
DC4d	332	706	933	1250	1759
Best speed-up					
DC4b	2.77x	2.17x	2.27x	1.73x	1.91x

On the other hand, when going from *DC3* through *DC4a* towards *DC4b*, there is progressive speed-up of the execution runtime resulting from the increase in the number of replicated LS kernels. Comparing *DC4b* and *DC4c*, which differ only in the representation of LS as fixed- and floating-point respectively, it can be seen that both designs provide comparable runtimes, with *DC4b* being slightly superior than *DC4c* (e.g., a maximum difference of 11 s for 4hmg). On the other hand, a significant decrease in performance with respect to *DC4b* occurs when Conform calculations are expressed in floating-point as in *DC4d* (e.g., a maximum difference of 321 s for 1stp). These results are due to the II increase of Conform's outmost-loop achieved with fixed-point (*DC4b*, II=10) vs. floating-point (*DC4d*, II=36). The case of InterE and IntraE is the opposite, because expressing their calculations in fixed-point resulted in ∼20% of performance decrease (Section 4.2), due to larger hardware area that led to lower frequencies (<170 MHz) for a design comparable to *DC3* in Table 3. Moreover, for these two kernels, relaxing the order of floating-point operations and removing intermediate rounding operations enabled through compiler flags (`-fpc-relaxed` and `-fpc` respectively) provided no performance benefits.

The maximum and minimum speed-up obtained with our fastest design (*DC4b*) was 2.77x and 1.73x for the cases 3ptb and 3ce3. This can be explained by the computation effort imposed by the PDB complex, which directly increases in cases where more atoms and torsions are present in the ligand.

Table 5 reports the energy consumption of the two platforms used, i.e., the single CPU core, and the FPGA. For the CPU case, the drawn power was sampled in 50 ms intervals using power performance counters to avoid inaccuracies typically associated with external measurements (e.g., shunt-based). The power samples were then integrated over time to derive the energy. For the FPGA case, we used estimated power values from fully placed and routed OpenCL projects using *quartus_pow* similarly as in [8]. The power estimate of ∼30 W was multiplied by the respective runtime (Table 4) to obtain the energy. Clearly, bigger/smaller energy-efficiency gains (1.86x for 3ptb, 1.12x for 3ce3) correspond to bigger/smaller speed-up factors (2.77x for 3ptb, 1.73x for 3ce3) as already shown in Table 4.

Table 5 Energy consumption and best energy-efficiency gains for 100 docking runs

Energy consumption (KJ)					
Design config.	**Ligand-Receptor PDB ID**				
	3ptb	***1stp***	***4hmg***	***3ce3***	***3c1x***
Serial CPU	11.80	16.69	28.07	36.27	54.85
DC4b	6.33	11.50	18.69	32.31	44.61
Best energy improvement					
DC4b	1.86x	1.45x	1.50x	1.12x	1.23x

5.4.1 Comparison against state-of-the-art accelerated AutoDock implementations

In comparison to the RTL-designed FPGA accelerator by Pechan et al. [11], we achieved much lower arithmetically-averaged speed-ups, i.e., ∼23.3x vs. ∼2.2x. The actual execution times, though, are similar: our OpenCL version achieves an average runtime of one docking run of ∼2.1 s and ∼3.8 s for 3ptb and 1stp, respectively. These values are comparable to the average of ∼3.1 s reported in [11], but highly depend on the size and structure of the ligand.

However, despite all of our efforts, GPUs do an even better job for accelerating the docking problem: In [15], Solis-Vasquez et al. report a speed-up of ∼55.7x for 3c1x, and an improvement of energy efficiency of ∼6.3x, over the serial baseline. The fact that the optimal pipelining (II=1) was achieved for each kernel of the bottleneck chain (consisting of Conform, InterE, and IntraE), suggests that the control mechanisms used in channel-based communications, such as our Arbiter kernel, are most likely not yet optimal in our current FPGA design.

5.4.2 Tool support and productivity

In general, for each development phase, design specification and its corresponding emulation-based verification were as easy as with GPUs. Before generating FPGA binaries, we extensively utilized the optimization reports provided by the tool, in order to assess the performance impact of code modifications in terms of achieved II, and the estimated resource utilization. As a result, a fully-emulated design for the first development phase (Section 4.1.1) was completed in ∼four weeks.

However, during each development phase, the corresponding hardware validation and subsequent optimization cycles were much more involved. At this stage, we used the hardware profiler to pinpoint bottlenecks caused by channels with unbalanced communication traffic between producer and consumer kernels, as well as inefficient memory accesses. Although all designs were verified through emulation, we observed this was not a *guarantee* that the mapped designs would execute as expected on the FPGA. Among the known emulator limitations [18], we found the concurrency issue the most critical one.

To address it, it was essential to consider the possible reorderings of operations potentially performed by the OpenCL compiler. This is the case for our GA kernel,

where several read and write channel calls happen at *different* variable scopes throughout the entire docking execution. At first glance, the order of channel calls can be enforced by using *channel fences* [18]. But in practice, this mechanism worked only as long as these calls occurred at the *same* variable scope. When such calls occurred between disjoint but still interdependent code blocks, the execution order across these channel-enclosing blocks could no longer be enforced by just using fences. Instead, we had to explicitly introduce guard variables, which we have to manually set/reset to enforce a valid execution order.

In summary, we spent a total of ∼five months on this development, which was considerably delayed by the concurrency issues, and of course by the non-negligible FPGA synthesis and mapping times of ∼eight hours for each of our largest designs.

5.4.3 Programming recipes and challenges for achieving higher performance

During the entire development, we aimed to achieve the highest possible performance estimation (II=1) for each single work-item kernel. Although this was not possible in all parts of our design, the difficulty was alleviated by spliting large sections (GA in the first development cycle) into smaller instances (as multiple LS and LFSR-LS kernels) in order to harness the potential concurrency. Moreover, appropriate data allocation on either on-chip or off-chip memory, as well as arithmetic representation, were considered key tools for achieving higher performance. The benefits of these general recipes are reflected in our most drastic performance improvements, i.e., moving from *DC2*, through *DC3* towards *DC4b* (Section 5.3).

From the software-programming perspective, a big challenge faced during development was the required *awareness* of the underlying hardware. *First*, there was the need for explicit synchronization between multiple read and write channels within single work-item kernels for ensuring correctness. *Second*, the need for re-arranging the local memory layout for increasing II. *Finally*, the need for code-refactoring using hardware constructs such as shift-registers, for better pipelining and unrolling loops.

Since compiler features are already available to support such manual transformations, and the fact that a development using a mix of OpenCL and RTL specifications is possible, reinforce the impression that one still needs to be aware of the underlying FPGA characteristics, even when using OpenCL.

6 Conclusion and Future Work

In addition to the OpenCL-on-FPGA design techniques we presented above, a key result of our study is the lack of *performance portability* of OpenCL. The data-parallel approach that allows the GPU implementation [15] to reach its impressive speed-ups leads to a *slow-down* of three orders of magnitude when used on the FPGA.

However, the complex docking code makes for an interesting subject for methodology studies. The improvements we show from the initial to the final version of OpenCL on FPGA were achieved by following a set of design and optimization steps for a *task-parallel* pipeline architecture, composed of single work-item kernels which communicate through channels.

We explored several FPGA-specific techniques to tackle the intrinsic termination-criteria known only at runtime of docking nested-loops; as well as different architectural choices, such as kernel replication and channel arbitration, to improve the efficiency of the encountered *multiple-producers-to-single-consumer* datapath.

In order to further improve the performance, and come closer to that of the RTL design [11], there is evidence [9] that control logic as complex as our Arbiter kernel may carry a high performance penalty. We will investigate alternative solutions to circumvent this issue.

Acknowledgment. The authors would like to thank Andreas Engel from ESA, and Miguel Quiliano-Meza from Universidad de Navarra for their valuable inputs for this work.

7 Literature

[1] Wang, G. et al.: Performance and productivity evaluation of hybrid-threading HLS versus HDLs, High Performance Extreme Computing Conference (HPEC), IEEE, 2015

[2] Weller, D. et al.: Energy Efficient Scientific Computing on FPGAs Using OpenCL, Field-Programmable Gate Arrays, ACM/SIGDA International Symposium on, 2017

[3] Kono, F. et al.: Performance Evaluation of Tsunami Simulation Using OpenCL on GPU and FPGA, Embedded Multicore/Many-core Systems-on-Chip (MCSoC), IEEE 11th International Symposium on, 2017

[4] Wang, D. et al.: PipeCNN: An OpenCL-based open-source FPGA accelerator for convolution neural networks, Field Programmable Technology (ICFPT), International Conference on, 2017

[5] Kenter, T. et al.: Flexible FPGA design for FDTD using OpenCL, Field Programmable Logic and Applications (FPL), 27th International Conference on, 2017

[6] Ndu, G. et al.: CHO: towards a benchmark suite for OpenCL FPGA accelerators, 3rd International Workshop on OpenCL (IWOCL), 2015

[7] Gautier, Q. et al.: Spector: An OpenCL FPGA benchmark suite, Field-Programmable Technology (FPT), International Conference on, 2016

[8] Zohouri, H. R. et al.: Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs, SC16: International Conference for High Performance Computing, Networking, Storage and Analysis, 2016

[9] Yang, C. et al.: OpenCL for HPC with FPGAs: Case Study in Molecular Electrostatics, IEEE High Performance Extreme Computing Conference (HPEC), 2017

[10] Sousa, S. F. et al.: Protein-ligand docking: Current status and future challenges, Journal of Proteins: Structure, Function, and Bioinformatics, Wiley Online Library, 2006

[11] Pechan, I. et al.: FPGA-based acceleration of the AutoDock molecular docking software, 6th Conference on Ph.D. Research in Microelectronics Electronics, 2010

[12] OCLADock-FPGA, `https://git.esa.informatik.tu-darmstadt.de/docking/ocladock-fpga`

[13] Morris, G. M. et al.: Automated docking using a Lamarckian genetic algorithm and an empirical binding free energy function, Journal of Computational Chemistry, Wiley Online Library, 1998

[14] Pechan I. et al.: Hardware Accelerated Molecular Docking: A Survey, Bioinformatics, InTech, 2012

[15] Solis-Vasquez, L. et al.: A Performance and Energy Evaluation of OpenCL-accelerated Molecular Docking, 5th International Workshop on OpenCL (IWOCL), 2017

[16] Pechan, I. et al.: Molecular Docking on FPGA and GPU Platforms, 21st International Conference on Field Programmable Logic and

Applications, 2011
[17] Kannan, S. et al.: Porting Autodock to CUDA, IEEE Congress on Evolutionary Computation, 2010
[18] Intel, Intel FPGA SDK for OpenCL, 2017
[19] Intel, Intel Arria 10 Device Overview, 2017
[20] Berman, H.M. et al.: The Protein Data Bank, `https://www.rcsb.org`, 2000
[21] Huey, R. et al.: Using AutoDock 4 and AutoDock Vina with AutoDockTools: A Tutorial, 2012