SOLIS VASQUEZ, LEONARDO

ACCELERATING MOLECULAR DOCKING BY PARALLELIZED HETEROGENEOUS COMPUTING – A CASE STUDY OF PERFORMANCE, QUALITY OF RESULTS, AND ENERGY-EFFICIENCY USING CPUS, GPUS, AND FPGAS

Printed and/or published with the support of the German Academic Exchange Service (DAAD).

ACCELERATING MOLECULAR DOCKING BY PARALLELIZED HETEROGENEOUS COMPUTING – A CASE STUDY OF PERFORMANCE, QUALITY OF RESULTS, AND ENERGY-EFFICIENCY USING CPUS, GPUS, AND FPGAS

at the Computer Science Department
of the Technische Universität Darmstadt

submitted in fulfilment of the requirements for the
degree of Doctor of Engineering
(Dr.-Ing.)

Doctoral thesis
by Solis Vasquez, Leonardo
from Lima, Peru

Reviewers
Prof. Dr.-Ing. Andreas Koch
Prof. Dr. Christian Plessl

Date of the oral exam
October 14, 2019

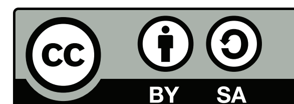
Darmstadt, 2019
D 17

Solis Vasquez, Leonardo: *Accelerating Molecular Docking by Parallelized Heterogeneous Computing – A Case Study of Performance, Quality of Results, and Energy-Efficiency using CPUs, GPUs, and FPGAs*
Darmstadt, Technische Universität Darmstadt
Date of the oral exam: October 14, 2019

Please cite this work as:
URN: urn:nbn:de:tuda-tuprints-92886
URL: https://tuprints.ulb.tu-darmstadt.de/id/eprint/9288

This document is provided by TUprints,
The Publication Service of the Technische Universität Darmstadt
https://tuprints.ulb.tu-darmstadt.de

This work is licensed under a Creative Commons "Attribution-ShareAlike 4.0 International" license.



ERKLÄRUNGEN LAUT PROMOTIONSORDNUNG

§8 Abs. 1 lit. c PromO

Ich versichere hiermit, dass die elektronische Version meiner Dissertation mit der schriftlichen Version übereinstimmt.

§8 Abs. 1 lit. d PromO

Ich versichere hiermit, dass zu einem vorherigen Zeitpunkt noch keine Promotion versucht wurde. In diesem Fall sind nähere Angaben über Zeitpunkt, Hochschule, Dissertationsthema und Ergebnis dieses Versuchs mitzuteilen.

§9 Abs. 1 PromO

Ich versichere hiermit, dass die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen verfasst wurde.

§9 Abs. 2 PromO

Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, August, 2019

Solis Vasquez, Leonardo

Dedicated to my lovely parents and sister.

ABSTRACT

Molecular Docking (MD) is a key tool in computer-aided drug design that aims to predict the binding pose between a small molecule and a macromolecular target. At its core, MD calculates the strength of possible binding poses, and searches for the energetically-stronger ones among those generated during simulation. Automatic Docking (AutoDock) is a widely-used MD code that employs a physics-based scoring function to quantify the binding strength. AutoDock also uses a Lamarckian Genetic Algorithm (LGA), and in turn, the Solis-Wets method, as a local-search algorithm, in order to find strong interactions of such molecular systems. Due to the highly-parallel nature of the LGA tasks involved, AutoDock can benefit from runtime acceleration based on parallelization.

This thesis presents an OpenCL-based parallelization of AutoDock, and a corresponding evaluation in terms of execution performance, quality-of-results, and compute-energy efficiency, achieved on different platforms based on: multi-core Central Processing Unit (CPU)s, Graphics Processing Unit (GPU)s, and Field Programmable Gate Array (FPGA)s. While a *data-parallel* approach has proven its effectiveness in accelerating AutoDock on CPUs and GPUs, it was observed that for FPGAs, such approach resulted in slower executions in the range of three-orders of magnitude when compared against the original single-threaded AutoDock. To overcome this drawback, a *task-parallel* implementation for FPGAs is discussed as well.

Besides presenting an AutoDock implementation being parallelized using OpenCL, this thesis also extends the LGA search with new *alternative* local-search methods based on gradients (of the scoring function) such as: Steepest-Descent, FIRE, and ADADELTA. Among these, it was found that ADADELTA provides significant algorithmic benefits over Solis-Wets, yielding a reduction in calculation effort down to 1/1300 of the legacy Solis-Wets method, while achieving equivalent quality-of-results. Compared to the original single-threaded AutoDock, the proposed data-parallel design achieves a speedup of up to ∼399x and improves the compute-energy efficiency by up to ∼297x when running on modern V100 GPUs. Furthermore, this thesis describes the adaptations performed on the proposed OpenCL-based implementation for supporting challenging real-world MD scenarios.

ZUSAMMENFASSUNG

Molecular Docking (MD) ist ein Schlüsselinstrument für das computergestützte Wirkstoffdesign, mit dem die Bindungspose zwischen einem kleinen Molekül und einem makromolekularen Ziel vorhergesagt werden soll. Im Kern berechnet MD die Stärke möglicher Bindungsposen und sucht nach den energetisch stärkeren unter denen, die während der Simulation erzeugt wurden. Automatic Docking (AutoDock) ist ein weit verbreiteter MD Code, bei dem die Bindungsstärke mithilfe einer physikbasierten Bewertungsfunktion quantifiziert wird. AutoDock verwendet auch einen Lamarckschen Genetischen Algorithmus (LGA) und als lokalen Suchalgorithmus die Solis-Wets-Methode, um starke Wechselwirkungen solcher molekularer Systeme zu finden. Aufgrund der hohen Parallelität der beteiligten LGA Aufgaben kann AutoDock von einer Laufzeitbeschleunigung auf der Grundlage der Parallelisierung profitieren.

Diese Dissertation präsentiert eine OpenCL-basierte Parallelisierung von AutoDock und eine entsprechende Bewertung in Bezug auf Ausführungsleistung, Ergebnisqualität und Rechen-Energieeffizienz, die auf verschiedenen Plattformen durchgeführt wird, basierend auf: Multi-Core Central Processing Unit (CPU)s, Graphics Processing Unit (GPU)s und Field Programmable Gate Array (FPGA)s. Während ein *datenparalleler* Ansatz seine Wirksamkeit bei der Beschleunigung von AutoDock auf CPUs und GPUs bewiesen hat, wurde beobachtet, dass ein solcher Ansatz bei FPGAs im Vergleich zum ursprünglichen AutoDock mit einem Thread zu langsameren Ausführungen im Bereich von drei Größenordnungen führte. Um diesen Nachteil zu überwinden, wird auch eine *aufgabenparallele* Implementierung für FPGAs diskutiert.

Neben der Darstellung einer AutoDock Implementierung, die mit OpenCL parallelisiert wird, erweitert diese Arbeit die LGA-Suche um neue *alternative* lokale Suchmethoden auf der Basis von Gradienten (der Bewertungsfunktion) wie Steepest-Descent, FIRE und ADADELTA. Unter diesen wurde festgestellt, dass ADADELTA signifikante algorithmische Vorteile gegenüber Solis-Wets bietet, was zu einer Reduzierung des Rechenaufwands auf 1/1300 der herkömmlichen Solis-Wets-Methode bei gleichwertiger Ergebnisqualität führt. Im Vergleich zum ursprünglichen single-threaded AutoDock erzielt das vorgeschlagene datenparallele Design eine Geschwindigkeitssteigerung von bis zu ∼399x und verbessert die Rechen-Energieeffizienz um bis zu ∼297x, wenn es auf modernen V100 GPUs ausgeführt wird. Darüber hinaus beschreibt diese Arbeit die Anpassungen, die an der vorgeschlagenen OpenCL-basierten Implementierung vorgenommen wurden, um herausfordernde reale MD Szenarien zu unterstützen.

PUBLICATIONS

Teamwork is the ability to work together toward a common vision. The ability to direct individual accomplishments toward organizational objectives. It is the fuel that allows common people to attain uncommon results. — Andrew Carnegie

Main ideas and figures have appeared previously in the following publications:

[1] Léa El Khoury, Diogo Santos-Martins, Sukanya Sasmal, Jérôme Eberhardt, Giulia Bianco, Francesca A. Ambrosio, **Solis-Vasquez, Leonardo**, Andreas Koch, Stefano Forli, and David Mobley. "Comparison of affinity ranking using AutoDock-GPU and MM-GBSA scores for BACE-1 inhibitors in the D3R Grand Challenge 4." In: *Journal of Computer-Aided Molecular Design* (2019). DOI: 10.1007/s10822-019-00240-w.

[2] Diogo Santos-Martins, **Solis-Vasquez, Leonardo**, Andreas Koch, and Stefano Forli. "Accelerating AutoDock4 with GPUs and Gradient-Based Local Search." In: *ChemRxiv (preprint)* (2019). DOI: 10.26434/chemrxiv.9702389.v1.

[3] Diogo Santos-Martins, Jérôme Eberhardt, Giulia Bianco, **Solis-Vasquez, Leonardo**, Francesca Alessandra Ambrosio, Andreas Koch, and Stefano Forli. "D3R Grand Challenge 4: prospective pose prediction of BACE1 ligands with AutoDock-GPU." In: *Journal of Computer-Aided Molecular Design* (2019). DOI: 10.1007/s10822-019-00241-9.

[4] **Solis-Vasquez, Leonardo** and Andreas Koch. "A Performance and Energy Evaluation of OpenCL-accelerated Molecular Docking." In: *Proceedings of the 5th International Workshop on OpenCL (IWOCL)*. Toronto, ON, Canada: ACM, 2017. DOI: 10.1145/3078155.3078167.

[5] **Solis-Vasquez, Leonardo** and Andreas Koch. "A Case Study in Using OpenCL on FPGAs: Creating an Open-Source Accelerator of the AutoDock Molecular Docking Software." In: *Proceedings of the 5th International Workshop on FPGAs for Software Programmers (FSP)*. Dublin, Ireland: VDE VERLAG, 2018. URL: https://ieeexplore.ieee.org/document/8470463.

[6] **Solis-Vasquez, Leonardo**, Diogo Santos-Martins, Andreas Koch, and Stefano Forli. "Performance Analysis of Molecular Docking in OpenCL: A Case Study of AutoDock enhanced with Gradients." In: *Submitted to the 34th International Parallel and Distributed Processing Symposium (IPDPS)*. Submitted, 2019.

[7] **Solis-Vasquez, Leonardo**, Diogo Santos-Martins, Andreas Koch, and Stefano Forli. "Evaluating the Energy Efficiency of OpenCL-accelerated AutoDock Molecular Docking." In: *Submitted to the*

28th Euromicro International Conference on Parallel, Distributed, and Network-based Processing (PDP). Submitted, 2020.

Gratitude changes everything.

— Anonymous

ACKNOWLEDGMENTS

I am deeply grateful to *Prof. Dr.-Ing. Andreas Koch* for his mentorship throughout my entire doctorate period, especially for the opportunity to be part of the *Embedded Systems and Applications Group (ESA)* at TU Darmstadt, Germany, as well as for his patience, continuous encouragement, and enriching support. I am also very thankful to my collegues at TU Darmstadt for their precious help and interesting discussions.

I would like to also thank *Prof. Dr. Christian Plessl* for the short-term preparation of the second review.

Special thanks to *Prof. Stefano Forli* and other collegues at *The Scripps Research Institute* in California, USA, the original authors of AutoDock, for their continuous support on computational chemistry matters, as well as for the interesting discussions during our collaborative research work.

Many thanks to the *German Academic Exchange Service* (DAAD) and the *Peruvian National Program for Scholarships and Educational Loans* (PRONABEC) for the financial aid. I also would like to thank AMD Inc., Xelera Technologies, and the AWS Cloud Credits for Research Program, for providing access and support of the hardware-acceleration technologies used in this work.

Many thanks to my friends for all the moments we spent together, and to those who *dared* getting along with me during this time, wherever they are now. Many of them made me *redefine* my idea of happiness . . . so now it also includes riding a bicycle and sharing some drinks, even in tough times!

Last but not least, I want to thank my lovely parents, sister, and close relatives. I am certainly not capable of finding the words to express my love and gratitude to you. However, writing these lines means that I have no more *excuses* that have been keeping me physically apart from you during this time.

CONTENTS

1 INTRODUCTION 1
- 1.1 Current trends in high-performance and scientific computing . . . 1
- 1.2 Target application and research problems . . . 3
- 1.3 Thesis contribution . . . 5
- 1.4 Thesis outline . . . 7

2 FUNDAMENTALS OF AUTODOCK MOLECULAR DOCKING 9
- 2.1 Background on molecular docking . . . 9
- 2.2 AutoDock: a software for automated docking . . . 11
  - 2.2.1 Encoding . . . 11
  - 2.2.2 Lamarckian Genetic Algorithm . . . 12
  - 2.2.3 Solis-Wets local search . . . 13
  - 2.2.4 Scoring function . . . 14
  - 2.2.5 AutoDock algorithm . . . 17
  - 2.2.6 Validation . . . 18

3 RELATED WORK 21
- 3.1 OpenCL programming and energy-efficiency aspects of heterogeneous systems . . . 21
  - 3.1.1 Performance portability of OpenCL . . . 21
  - 3.1.2 OpenCL for FPGA programming . . . 22
  - 3.1.3 Compute-energy efficiency . . . 24
- 3.2 Parallel implementations of molecular docking tools . . . 25
  - 3.2.1 FFT-based tools . . . 26
  - 3.2.2 Evolutionary-based tools . . . 29
  - 3.2.3 AutoDock . . . 31
  - 3.2.4 Pairwise potentials . . . 32
  - 3.2.5 Intrinsically-parallel tools . . . 33
  - 3.2.6 Other parallelization approaches . . . 34
- 3.3 Algorithmic improvements in molecular docking . . . 35
- 3.4 Wrap-up discussion . . . 36

4 OCLADOCK: OPENCL-ACCELERATED AUTODOCK ON CPUS AND GPUS 39
- 4.1 OpenCL implementation of AutoDock . . . 39
  - 4.1.1 Data-based parallelization . . . 39
  - 4.1.2 Code architecture . . . 40
- 4.2 Experimental evaluation . . . 42
  - 4.2.1 Setup . . . 42
  - 4.2.2 Validation . . . 44
  - 4.2.3 Execution performance . . . 45
  - 4.2.4 Compute-energy efficiency . . . 47

5 OCLADOCK-FPGA: PORTING AUTODOCK TO FPGAS USING OPENCL 53
5.1 Data-parallel approach on FPGAs 53
5.2 Task-parallel approach: reformulated strategy for FPGAs 54
5.2.1 Reference pipeline design for FPGAs 54
5.2.2 The development phases 54
5.2.3 Further optimization techniques 64
5.3 Experimental evaluation 65
5.3.1 Setup 65
5.3.2 Validation 66
5.3.3 Design configurations and resource utilization . 67
5.3.4 Execution performance 68
5.3.5 Compute-energy efficiency 70
5.3.6 Further analysis 70
6 ENHANCING OCLADOCK WITH GRADIENTS OF THE SCORING FUNCTION 73
6.1 Gradient-based optimization 73
6.1.1 Gradient calculation 73
6.1.2 Gradient conversion from atomic into genetic space 74
6.1.3 Gradient-based local-search methods 76
6.1.4 Incorporation into OCLADock 78
6.2 Experimental evaluation 80
6.2.1 Setup 80
6.2.2 Validation 80
6.2.3 Profiling analysis for optimum local-search rate 83
6.2.4 Efficiency of gradient-based methods 86
6.2.5 Portability to other accelerators 88
6.2.6 Compute-energy efficiency 93
7 USING OCLADOCK FOR COMPETITIVE DRUG DISCOVERY 97
7.1 The challenge of docking macrocyclic molecular structures 97
7.1.1 Why is this actually a challenge? 97
7.2 Handling macrocycles with OCLADock 99
7.2.1 Macrocycle-oriented scoring-function terms . . 99
7.2.2 Macrocycle-oriented development 100
7.2.3 Experimental evaluation 100
8 CONCLUDING REMARKS 103
8.1 Summary 103
8.2 Lessons learned 104
8.2.1 OpenCL for FPGAs 104
8.2.2 OpenCL for GPUs and CPUs 105
8.2.3 OpenCL beyond datacenters 106
8.3 Remaining research and engineering challenges 107
8.3.1 Extending functionality of OCLADock 107
8.3.2 Enhancing performance of OCLADock on FPGAs 108

A KEY IMPLEMENTATION DIFFERENCES COMPARED TO ORIGINAL AUTODOCK CODE 109
B COMPARING PERFORMANCE AGAINST OTHER PARALLELIZED DOCKING SOFTWARE 111
C MEMORY REQUIREMENTS 115
D FUTURE TRENDS OF OPENCL 119

BIBLIOGRAPHY 123

LIST OF FIGURES

Figure 2.1 Degrees of freedom of a theoretical ligand composed of atoms A, B, C, . . . , O. Bonds between atoms are depicted as connecting lines. Each rotatable bond (E–H and I–J) corresponds to a torsion, i. e., rotation of affected ligand atoms around the rotatable-bond axis. 12

Figure 2.2 A grid-based approach is used for speeding up the calculation of intermolecular interactions. AutoGrid pre-calculates grid maps of interaction for various ligand atom types. Afterwards, AutoDock interpolates certain grid values to calculate the total intermolecular interaction. . 17

Figure 2.3 AutoDock block diagram [233] with default values of LGA parameters. 18

Figure 4.1 Mapping AutoDock – GA and LS – functions onto OpenCL kernels. Kernel blocks enclose nested loops controlling LGA runs, as well as GA and LS inner processing. Kernels operate over several individuals simultaneously, with each individual being mapped to a single OpenCL work-group. 40

Figure 4.2 A population processed by an LGA run ($\mathrm{Run}_{\mathrm{ID}}$) can be decomposed into its individuals, and each individual ($\mathrm{Ind}_{\mathrm{ID}}$) can be mapped onto a work-group ($\mathrm{WG}_{\mathrm{ID}}$). The entire set of work-groups is distributed by the GPU runtime scheduler over the available Q compute units (CUs). A CU is a multi-threaded hardware unit capable of processing one work-group (composed of L work-items) at a time. The runs, individuals, and fine-grain tasks are colored according to their associated level of parallelism: high (blue), medium (red), and low (green). 41

Figure 4.3 The overall OCLADock workflow consists of a sequence of host (Hx) and device (Dx) functions. Program execution always starts and finishes in host functions (depicted at the left side). OpenCL kernels are executed iteratively on the device (depicted at the right side), while their termination is controlled by the host. 43

Figure 4.4 Speedups of OCLADock vs. single-threaded AutoDock achieved on GPU/CPU devices for different work-group sizes. Vertical scales are different. 46
Figure 4.5 Speedups of OCLADock vs. single-threaded AutoDock achieved on CPU (16 work-items) and GPU (64 work-items). Vertical scales are different. 48
Figure 4.6 Power measurements on the RX-290X GPU for 10 LGA runs using `3c1x`. 49
Figure 4.7 Energy-efficiency gains of OCLADock vs. single-threaded AutoDock achieved on CPU (16 work-items) and GPU (64 work-items). Vertical scales are different. 51
Figure 5.1 Pipeline processing of the LGA of AutoDock proposed in [146]. 54
Figure 5.2 First development phase: initial OpenCL design. 55
Figure 5.3 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. 59
Figure 5.4 Third development phase: local-search kernels are *replicated* three times, while an arbiter kernel is added to handle simultaneous score-calculation requests. Score calculation kernels are omitted for simplicity. 60
Figure 5.5 Fourth development phase: local-search kernels are further replicated, while the arbitration mechanism is improved. 64
Figure 5.6 Speedups of OCLADock-FPGA fastest design DC4b vs. single-threaded AutoDock. 69
Figure 5.7 Energy-efficiency gains of OCLADock-FPGA fastest design DC4b vs. single-threaded AutoDock. 71
Figure 6.1 Speedups of OCLADock vs. single-threaded AutoDock achieved on a Vega 56 GPU (R = 100 LGA runs, `lsrate` = 100 %). 87
Figure 6.2 Speedups of OCLADock vs. single-threaded AutoDock achieved on selected GPU/CPU devices for different work-group sizes (R = 100 LGA runs, `lsrate` = 100 %). Vertical scales are different. 89

Figure 6.3 Speedups of OCLADock vs. single-threaded AutoDock achieved on selected GPU/CPU devices using work-group sizes of 64/32 work-items, respectively (R = 100 LGA runs, `lsrate` = 100 %). 90
Figure 6.4 Statistics of speedup factors: OCLADock vs. single-threaded AutoDock achieved on all selected devices (R = 100 LGA runs, `lsrate` = 100 %). 92
Figure 6.5 Power measurements of OCLADock for Solis-Wets and ADADELTA executions on the Vega 56 GPU (R = 100 LGA runs, `lsrate` = 100 %). . . 94
Figure 6.6 Statistics of energy-efficiency gains: OCLADock over single-threaded AutoDock achieved on devices where measuring power was feasible (R = 100 LGA runs, `lsrate` = 100 %). 95
Figure 7.1 Three-dimensional representation of a macrocycle example: 1nm6 ($C_{27}H_{33}ClN_6O_2$). Atoms are carbon (gray), hydrogen (not shown), chlorine (green), nitrogen (blue), and oxygen (red). The number of atoms in the ring is $N_{\text{atom}}^{\text{ring}} = 19$, and that of active rotatable bonds is $N_{\text{rot}}^{\text{active}} = 12$. Figure was obtained from [19]. 98
Figure 7.2 Left: identification of the ring bond to be broken: A – B. Right: introduction of the so-called *invisible* atoms A_{inv} and B_{inv}, used for the ring-closure procedure during docking. Both sides show the assignment of non-standard atomic types (*CG*, *Go*) to atoms in the broken bond. . 101

LIST OF TABLES

Table 3.1 Parallelized MD tools. Fields specified with a "–" mean the program/feature was not existent/not implemented. 27
Table 4.1 OCLADock configuration for experiments. . . 44
Table 4.2 Functional validation of OCLADock vs. single-threaded AutoDock, both running the Solis-Wets local-search method. All results were obtained using 100 LGA runs, and RMSD tolerance = 2 Å. Best values within each criterion are colored. 45

Table 4.3 Execution time (s) and speedups for 100 LGA runs on CPU (16 work-items) and GPU (64 work-items). 47
Table 4.4 Measured power values (approximated) on the CPU. 49
Table 4.5 Energy consumption (kJ) results and energy-efficiency gains for 100 LGA runs on CPU (16 work-items) and GPU (64 work-items). 50
Table 5.1 Functional validation of OCLADock-FPGA vs. single-threaded AutoDock, both using the Solis-Wets local-search method. RMSD values are omitted for simplicity. All results were obtained using 100 LGA runs. Best values within each criterion are colored. 67
Table 5.2 Development phases and design configurations. 68
Table 5.3 FPGA resource utilization and maximum frequency. 68
Table 5.4 Execution time (s) for 100 LGA runs. 69
Table 5.5 Compute-energy consumption (kJ) for 100 LGA runs. 70
Table 6.1 Hardware and software setup in terms of instance type, peak memory bandwidth (GB/s), peak single-precision FP performance (GFLOP/s), number of OpenCL compute units (# CU), preferred OpenCL work-group size (WG_{size}), and tool versions. 81
Table 6.2 Functional validation of OCLADock vs. single-threaded AutoDock for Solis-Wets (SW) and gradient methods: Steepest Descent (SD), FIRE, and ADADELTA (AD). Serial SW results were obtained on a E5-2666 CPU core, while the OpenCL ones targeted a Vega 56 GPU, using 100 LGA runs for all cases. The best values within each case are colored. Non-colored cells indicate ligand-receptor cases where equally-good results were found, or that it was not possible to determine the best method. 82
Table 6.3 Comparison of OCLADock local-search (LS) kernels using profiling metrics on the Vega 56 GPU (100 LGA runs). 84
Table 6.4 Resource utilization and its equivalent number of wavefronts in `Krnl_LS` for the experiment in Table 6.3. VGPR values, which limit the overall GPU occupancy, are highlighted. . . . 86

Table 7.1 Experiments on a set of 20 ligands performed for the GC4 blind prediction competition. The best values within each case are colored. 101
Table A.1 Bit-field description of a 32-bit rotation-list item. 110
Table B.1 Configuration of benchmarked MD codes. . . 112
Table B.2 Average results of MD codes benchmarking. Accelerator devices were previously utilized in Chapter 6. 113
Table C.1 Upper limits of MD parameters in OCLADock (defined in repository [187] /common/defines.h). 116
Table C.2 Constant data structures and their members in OCLADock. 117
Table C.3 Additional constant data structures and their members for gradient calculation in OCLADock. 118

LIST OF ALGORITHMS

1 Lamarckian Genetic Algorithm (LGA) 13
2 Solis-Wets local search 14
3 Scoring Function (SF) 16
4 Attempt to synchronize accesses to a given location in external memory using fences. Correct data transactions between kernels (e. g., `Rx_Val` receiving the value initially stored in `Tx_Val`) occur only in emulation. For that reason, this alternative design was discarded. 56
5 Code structure used in the `InterScore` kernel implementation. Similar structures are used in `PoseCalc` and `IntraScore`. The outermost while-loop is controlled by the `active` signal. The main-computation loop lists only simplified operations. 58
6 In the second development phase, the random number generator `RNG` function invoked within `LGA` was replaced with a `LFSR-GA` kernel. 60
7 In the third development phase, `Arbiter` kernel reads *ready* signals and genotypes from producer kernels: `GA` and three `LS`. Accumulation and dispatch to `PoseCalc` of received genotype data is omitted for simplicity. 62
8 In the fourth development phase, genotypes generated in either `GA` or any of the nine `LS` kernel are sent directly to `PoseCalc`, instead of being accumulated in `Arbiter`, as in the third development phase. 63
9 Gradient Calculation (GC) 74

10 Incorporating ADADELTA local search into OCLADock. The other gradient-based methods, Steepest Descent and FIRE, are incorporated similarly, and thus are not shown here. 79

ACRONYMS

AutoDock Automatic Docking

CPU Central Processing Unit

D3R Drug Design Data Resource

DSP Digital Signal Processing

FPGA Field Programmable Gate Array

GPU Graphics Processing Unit

HDL Hardware Description Language

HLS High-Level Synthesis

HPC High-Performance Computing

LGA Lamarckian Genetic Algorithm

MD Molecular Docking

OpenCL Open Computing Language

OCLADock OpenCL-Accelerated Molecular Docking

RTL Register-Transfer Level

TSRI The Scripps Research Institute

1

INTRODUCTION

In this chapter, current trends in High-Performance Computing (HPC) applied to scientific computing are summarized. This paves the way for introducing aspects of Molecular Docking (MD), and one of the most cited software tools in this domain: AutoDock. Analyzing this scenario allows identifying relevant research problems that are addressed in this thesis. Finally, the contributions and structure of this work are presented.

1.1 CURRENT TRENDS IN HIGH-PERFORMANCE AND SCIENTIFIC COMPUTING

As stated by Golub and Ortega [59], scientific computing is the science – i. e., the collection of tools, techniques, and theories – that employs computer systems for solving mathematical models of science and engineering problems. Such problems arise from several disciplines like mathematics (e. g., algebra, analysis, topology, statistics, etc), engineering (e. g., electrical, mechanical, civil, chemical, etc), and natural sciences (e. g., biology, chemistry, physics, astronomy, earth sciences, etc). Scientific computing involves interdisciplinary activities, which according to Eijkhout [43], mainly focus on three aspects:

Examples on classification of scientific disciplines can be found in [52, 54, 119].

- the mathematical modeling of real-world phenomena,
- the numerical analysis of modeling algorithms, and
- the efficient computation of numerical algorithms.

Due to the increasing computational demands in the aforementioned aspects, efficient computer systems are becoming extremely required in current research. As reported in the 52nd edition of the *Top500* list [231] (corresponding to November 2018), *Summit* [184] – the fastest computer system nowadays – achieves a theoretical peak performance of 200 PFLOPs, and peak of 143.5 PFLOPs according to the High Performance LINPACK benchmark [232]. The fact that the top five systems in the *Top500* list – located in the United States (*Summit*, *Sierra*), China (*Sunway TaihuLight*, *Tianhe-2A*), and Switzerland (*Piz Daint*) – are utilized for scientific studies, as well as for collaborations

towards *Exascale Computing* such as *CORAL* [45, 101], clearly shows the need of HPC systems in scientific computing.

The historical developments in terms of the computer architectures utilized in *Top500* systems indicate that the upcoming trend are accelerators, i. e., systems comprising co-processors attached to the main CPU [43]. Particularly, a total of 138 systems in the *Top500* list use accelerators, mainly dominated by Tesla P100 [133] and V100 [134] GPU technologies, which are present in 60 and 41 out of such top 138 systems, respectively [230]. While GPU and *the-now-discontinued* Xeon Phi [80] are the only accelerator technologies used in latest *Top500* systems, the installation of large-scale FPGA systems (in e. g., Florida [56], Texas [189], Paderborn [142]) indicates that these reconfigurable accelerators are becoming increasingly attractive for scientists as well.

Traditionally, achieving higher performance has been the main goal in HPC. However, energy efficiency is becoming increasingly important. For current large-scale systems, the importance of energy efficiency is reflected on the *Green500* list [229]. Similarly as in the *Top500* list, the usage of GPUs in *Green500* is becoming significant, since eight out of the first ten systems use Tesla GPUs. Moreover, as stated by Plessl [149], the energy-efficiency issue opens another dimension for competition in the HPC world. Such competitive scenario has motivated several porting efforts of scientific computing applications to HPC [69, 122, 128, 208], as well as large investments in accelerator technologies by cloud and data center companies like Amazon, Microsoft, Baidu, IBM, and Huawei.

As of November 2018, the most efficient energy system according to Green500 is Shoubu system B [170], achieving a power efficiency of 17.6 GFLOPs / Watts.

In the HPC scenario, computer architects and programmers are moving towards the paradigm known as *heterogeneous computing*, where the best capabilities from different co-processors can be combined for further performance gains. However, the *broadly* different architectures and programming models required for co-processors bring significant challenges to achieving efficient computations [114]. From the standpoint of a scientific computing researcher, it is *likely* that architectural matters in computer systems would not be as much explored as programming models. This is due to the fact that – for achieving efficient executions of scientific software – designing and configuring computer architectures requires highly-specialized knowledge, rather than steering an application *more directly* with programming models.

The benefits of leveraging HPC are promising. However, it is timely to recall the popular adage: there is no such a thing as a free lunch.

Among the existing programming models used in parallel computing (e. g., OpenMP, Pthreads, CUDA), the Open Computing Language (OpenCL) [63] (initially released in 2009), thought of as a language based on C/C++ with extensions for parallel programming, is likely *the only one* providing actual code portability to various types of devices (e. g., CPUs, Digital Signal Processors, GPUs) from several vendors, including those with completely different architectures (i. e., FPGAs). In recent years, SYCL [65] (initially released in 2014), a C++

Overviews on the OpenCL, SYCL, CUDA, and OpenMP ecosystems can be found in [21, 64, 66, 135], respectively.

single-source abstraction layer for OpenCL, is gaining an increasing traction from both academia [4, 151, 172] and industry [29, 85, 95]. While this trend seems to be growing – making SYCL a (royalty-free) future competitor of (proprietary lock-in) CUDA – SYCL is not yet as *mature* and *outspread* as OpenCL. Due to the fact that OpenCL lays the foundations of SYCL, and despite the larger popularity of CUDA compared to both OpenCL and SYCL, the usage of OpenCL for current research and development is relevant, especially in scientific communities where open standards are preferred.

1.2 TARGET APPLICATION AND RESEARCH PROBLEMS

A science domain that comprises computationally-expensive applications is computational chemistry. The availability of software packages from such domain in several HPC academic centers [25, 36, 47, 118, 125, 143], as well as the interest of cloud providers to push research in this direction [6, 15, 169], are examples of the significant attention the research in this domain is receiving. Within this domain, computational drug discovery, which combines pharmaceutical chemistry and computational biology [106], has become a critical field fighting against diseases like acquired immunodeficiency syndrome (AIDS) [67] and cancer [161].

Molecular Docking (MD) is a widely-used method in computational drug discovery. Basically, it aims to predict the interaction between small molecules and macromolecular targets, with both molecule types referred to as ligand and receptor, respectively. This technique seeks ligands that effectively inhibit the harmful activity of certain receptor proteins [70]. The interaction in such molecular systems is estimated with scoring functions, which quantify with great detail the ligand-receptor binding, and take into account (among other factors) hundreds of atoms. For exploring the poses of molecules resulting from such interactions, an MD program can easily invoke a scoring function more than 10^6 times. On a large scale, MD is employed in virtual screening to rapidly identify drug candidates from a database typically composed of hundred thousands to million ligands. This, in turn, results in 10^{10} to 10^{16} score evaluations [96].

As listed by the Swiss Institute of Bioinformatics [185], and reported by Pagadala, Syed, and Tuszynski [140], there are more than 60 MD tools – available either as open-source or commercially – that have been developed during the last two decades. One of the most popular MD tools throughout these years [49, 179, 216, 220] has been AutoDock, originally developed by Goodsell and Olson [61] at The Scripps Research Institute (TSRI). The inherent parallelism from its genetic algorithm engine has made it suitable for different acceleration approaches that range from grid (e. g., *World Community Grid* [207]), distributed (e. g.,

combining MPI and OpenMP [124]) to heterogeneous (e. g., using CUDA [88, 144], Verilog [146]) computing.

Besides the aforementioned relevance of AutoDock in computational drug discovery, its intrinsic paralellism and algoritmic complexity make AutoDock a suitable *real-world* case study for OpenCL-based parallelization targeting (co-)processors typically employed in scientific environments (e. g., CPUs, GPUs, FPGAs). Therefore, the research problems addressed by this thesis are:

1. To the best of our knowledge, there is no open-source OpenCL implementation of AutoDock. Implementations with other programming models [88, 144, 146] are not publicly available. While there are some studies claiming successful OpenCL implementations, it was found that in such cases, authors focused only on certain parts of AutoDock (i. e., the genetic algorithm, considered the global search engine), usually excluding the so-called local search. From the computational-chemistry standpoint, the local search is a method that can considerably enhance MD results at the price of increasing the algorithmic complexity (by introducing execution paths controlled at runtime), and execution time (with larger time-intensive loops).

2. In recent years, the adoption of High-Level Synthesis (HLS) for FPGA design has increased significantly, as it enables programmers without a deep knowledge of the underlying architecture and Hardware Description Language (HDL)s to harness the efficiency of FPGAs. Although the higher level of abstraction at the specification phase, achieving high performance is still challenging due to the lack of direct control of low-level characteristics such as resource usage, placement and timing constraints [201, 204]. Studies in different aspects – such as programming practices [164], tool usage [163], and acceleration of scientific code [93, 162, 204, 218] – suggest that OpenCL is promising for FPGAs. Despite that, the OpenCL efficiency on FPGAs for MD was not yet explored.

3. Concurrently to efforts to speed-up MD processing times, several studies have been aiming to improve the MD *quality* with more efficient local-search methods [216]. Despite the fact that certain local-search methods (e. g., based on gradients of the scoring function) have shown significant enhancements over traditional ones [3, 53, 188], their efficient parallelization and incorporation into well-established codes like AutoDock is still lacking. Such incorporation is particularly challenging, as AutoDock involves a heuristic process, which in order to be verified, requires several executions (with different inputs) of considerably long durations.

4. Most of the research and development involving OpenCL focuses on parallel programming towards faster executions [87, 167]. As already mentioned, energy efficiency is becoming as important as performance in HPC, and thus, scientific applications are not exempted from these two concerns. Narrowing it down to MD, it was found that only one software (BUDE [109]) has been analyzed before in terms of power consumption and energy efficiency.

1.3 THESIS CONTRIBUTION

Based on the aforementioned research problems, the contributions of this thesis are the following:

- An OpenCL implementation of AutoDock, including the default Solis-Wets local-search method, using a *data-parallel* approach for CPUs and GPUs, which are the most popular accelerators available in scientific computing environments.

- An OpenCL implementation of AutoDock using a *task-parallel* approach for FPGAs. This involves a detailed exploration of 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. An analysis of the OpenCL portability between CPUs, GPUs, and FPGAs is provided as well.

- Incorporation of *gradient-based* methods for local search (i. e., Steepest-Descent, FIRE, and ADADELTA) into the OpenCL implementation of AutoDock. These newly-incorporated methods are evaluated in terms of runtime performance and energy efficiency on recent on-premise and cloud accelerators. Moreover, such methods are compared against the default Solis-Wets method in terms of MD quality.

- Extending the implementation in order to tackle a more challenging MD scenario, i. e., for docking *macrocyclic* molecules (a problem difficult to model), by leveraging OpenCL faster executions, and thus exploring more complex MD simulations that would be excessively time-consuming or even not successful to handle otherwise, i. e., with the original sequential AutoDock and default Solis-Wets method.

- To the best of our knowledge, this thesis provides the *first* open-source OpenCL implementation of AutoDock following two parallelization approaches: a data-parallel one for CPUs/GPUs, and a task-parallel one for FPGAs. The code developed in this thesis is called OpenCL-Accelerated Molecular Docking (OCLADock), and

has been released as open-source under the following GitLab repositories:

- *OCLADock - OpenCL Accelerated Molecular Docking*.
  https://git.esa.informatik.tu-darmstadt.de/docking/ocladock
- *OCLADock-FPGA: OpenCL Accelerated Molecular Docking on FPGAs*.
  https://git.esa.informatik.tu-darmstadt.de/docking/ocladock-fpga

Additionally, our CPU/GPU code will be officially incorporated into the *mainline* AutoDock suite, and maintained by TSRI – the original AutoDock developers – under the following GitHub repository:

- *AutoDock for GPUs using OpenCL*.
  https://github.com/ccsb-scripps/AutoDock-GPU

The listed contributions have already been peer-reviewed, and their corresponding publications were accepted at international conferences/journals in the HPC and computational chemistry domains:

- *A Performance and Energy Evaluation of OpenCL-accelerated Molecular Docking*.
  Solis-Vasquez, L. et al. [233]

- *A Case Study in Using OpenCL on FPGAs: Creating an Open-Source Accelerator of the AutoDock Molecular Docking Software*.
  Solis-Vasquez, L. et al. [234]

- *Comparison of affinity ranking using AutoDock-GPU and MM-GBSA scores for BACE-1 inhibitors in the D3R Grand Challenge 4*.
  El Khoury, L. et al. [44]

- *D3R Grand Challenge 4: prospective pose prediction of BACE1 ligands with AutoDock-GPU*.
  Santos-Martins, D. et al. [166]

The following publications, currently under review, have been submitted to conferences/journals in the HPC and computational chemistry domains:

- *Evaluating the Performance and Portability of Molecular Docking in OpenCL: A Case Study of AutoDock enhanced with Gradients*.
  Solis-Vasquez, L. et al. [235]

- *Evaluating the Energy Efficiency of OpenCL-accelerated AutoDock Molecular Docking*.
  Solis-Vasquez, L. et al. [236]

- *Accelerating AutoDock4 with GPUs and Gradient-Based Local Search*.
  Santos-Martins, D. et al. [165]

1.4 THESIS OUTLINE

This thesis is structured as follows:

Chapter 2 introduces the fundamentals of MD, the functionality of AutoDock, and its main components such as a Lamarckian Genetic Algorithm (LGA) using the Solis-Wets local-search method, and a scoring function.

Chapter 3 reviews the literature comprising studies using OpenCL, compute-energy analysis for heterogeneous systems, as well as porting efforts of AutoDock and other MD tools to different computing systems.

Chapter 4 describes OCLADock, a data-based parallelization of AutoDock using OpenCL for targeting multi-core CPUs and many-core GPUs, reporting performance and energy efficiency gains.

Chapter 5 describes a task-based parallelization of AutoDock using OpenCL for targeting FPGAs, reporting – besides performance and energy efficiency gains – key programming challenges and differences with respect to the data-parallel approach.

Chapter 6 describes the incorporation into the data-parallel OCLADock of local-search methods based on *gradients* of the scoring function, as an alternative to the legacy Solis-Wets method. It provides an analysis of performance, quality-of-results, and energy efficiency gains.

Chapter 7 extends the applicability of OCLADock by providing a description of the program adaptations made in order to scale to the complex requirement of docking *macrocyclic* molecules.

Chapter 8 reports lessons learned from using OpenCL for porting OCLADock between CPUs, GPUs, and FPGAs, as well as an outlook of results, remaining engineering challenges, and future work.

2

FUNDAMENTALS OF AUTODOCK MOLECULAR DOCKING

This chapter provides the fundamentals of MD and specifics of AutoDock. The most critical components of MD programs such as the search algorithm and scoring function are described, as well as the AutoDock functionality and its validation procedure.

2.1 BACKGROUND ON MOLECULAR DOCKING

MD simulations aim to predict the predominant binding poses of a (small) ligand molecule and a (macromolecular) receptor target, both of known three-dimensional structure. An MD software is used to identify ligands that effectively inhibit the harmful activity of certain receptor proteins [70].

MD performs a search procedure for minimum value of the scoring function.

This prediction aims for the *best* pose, i. e., it involves solving an optimization problem that suffers from a combinatorial explosion due to the many degrees of freedom of such molecular systems – i. e., all possible translations, orientations and torsions – experienced by molecules during interaction. The interaction is estimated with scoring functions, which quantify the ligand-receptor binding in great detail. For exploring the scoring landscape, an MD program can easily invoke a scoring function more than 10^6 times. As already described in Chapter 1, MD is employed in virtual screening to rapidly identify drug candidates from a database typically composed of hundred thousands to million ligands, which in turn results in 10^{10} to 10^{16} score evaluations [96].

The most critical components of an MD program are the search algorithm and scoring function [49, 202, 216]. There are many different design approaches for these two MD components, and hence, many different MD software exist (more than 60 [140, 185]). For a deeper understanding of the MD functionality, a classification along with a brief description of the different approaches of search algorithms and scoring functions is provided as follows.

Search algorithms can be classified into three categories [202, 216]:

1. Shape matching: performs several alignments between the ligand and receptor, while seeking to maximize their geometrical overlap. E. g., DOCK [192], EUDOC [141].

2. Systematic: faces the combinatorial explosion by progressively changing all ligand's degrees of freedom. Algorithms can be further divided into exhaustive search, fragmentation, and conformational ensemble. E. g., eHiTS [228], GLIDE [58].

3. Stochastic: performs random changes in the ligand, generates ensembles of conformations, and thus populates a *wide* range of the scoring landscape. Favorable changes are accepted. Algorithms can be based on *Monte Carlo*, tabu search, genetic algorithms, swarm optimization. E. g., GOLD [55], AutoDock.

Scoring functions can be classified into four categories. An extensive study on this classification by Liu and Wang [105] is summarized as follows:

1. Force field: are based on physics, and consist of a sum of non-covalent energy terms including *van der Waals*, hydrogen bonding, electrostatics, and desolvation. Such functions often need empirical scaling parameters to fit their results to experimental data obtained using *x-ray crystallography*.

X-ray crystallography *is a technique used for structure determination of biological macromolecules. Structure-based drug design is one of the many areas in which x-ray crystallography has provided clarification [174].*

2. Empirical: consist of a sum of *rewarding scores* and *penalties*. Additional penalties might be considered in case of covalent docking. Since multiple terms with different implications are combined, such functions rely on multivariate linear regression to derive the weight factors for each contributing term.

3. Knowledge-based: consist of a sum of pairwise (i. e., between a pair of atoms) statistical potentials between a ligand and receptor. The potentials employed are extracted through statistical analysis of structures, rather than from attempts to reproduce known binding energies.

4. Descriptor-based: if the properties of the ligand and receptor, as well as their interaction patterns, can be coded with certain descriptors, then machine-learning techniques employed in modern quantitative structure-activity relationship (QSAR) analysis can be applied to derive statistical models to compute scores.

Regarding how such scoring-function categories compare with each other, there are relevant details that can be briefly mentioned:

- The boundary between force-field and empirical is not strict, because both decompose the ligand-receptor binding energy into individual terms. Their major difference is, that force-field functions possess a complete theoretical framework, while empirical functions often adopt a flexible and intuitive form composed from scratch. Moreover, empirical function terms are simpler than those of force fields, and hence, are much faster to compute.

- Knowledge-based potentials are derived through statistical analysis without the need of experimental binding data, and hence, their main benefit is the computational and conceptual simplicity compared to force-field and empirical functions, respectively.
- Force-field and empirical functions, as well as knowledge-based potentials have linear forms, while a descriptor-based function – depending on the machine-learning technique used – does not necessarily have such form. For selecting a descriptor-based model (i. e., a combination of descriptors), such scoring functions rely on machine learning, whose selection rationale is often vague and has no interpretable physical meaning.

2.2 AUTODOCK: A SOFTWARE FOR AUTOMATED DOCKING

AutoDock is a widely-used and open-source MD software [49, 179, 216, 220]. It was developed and is currently maintained by TSRI [35]. Over the years, it has been continuously enhanced with more efficient search methods and scoring functions [77]. Different search methods are currently available in AutoDock: *simulated annealing*, *distributed simulated annealing* [61], and Lamarckian Genetic Algorithm (LGA) [116]. In this thesis, the focus is on the latest available AutoDock: version 4.2.6. More specifically, on its LGA search engine, since it has proven to achieve better MD results among all prior algorithms [116].

2.2.1 *Encoding*

MD is an optimization problem where different poses of the ligand, i. e., spatial geometrical arrangements, are explored in order to find the one that binds *strongly* to a certain binding region on the receptor. The most direct way of exploring the landscape of molecular poses would be to generate different three-dimensional positions for all N_{atom} (e. g., hundred) atoms present in the ligand. However, due to the combinatorial explosion that this approach suffers from, a more efficient representation or encoding for the ligand poses is required.

Such encoding can be devised by considering the ligand as a *flexible* body, whose poses can be represented using a combination of variables that describe: *first*, overall motion as a rigid body; and *second*, internal body flexibility (Figure 2.1).

Similar to other software reported in [140], AutoDock *treats the ligand as flexible, whereas the receptor is considered rigid.*

1. As a rigid body, the ligand can experience two types of motions: translation and rotation. Translation can be encoded with variables describing displacement in x, y, and z directions. Rigid-body rotation (orientation) can be described in axis-angle coordinates, i. e., with ϕ, θ, and α variables.

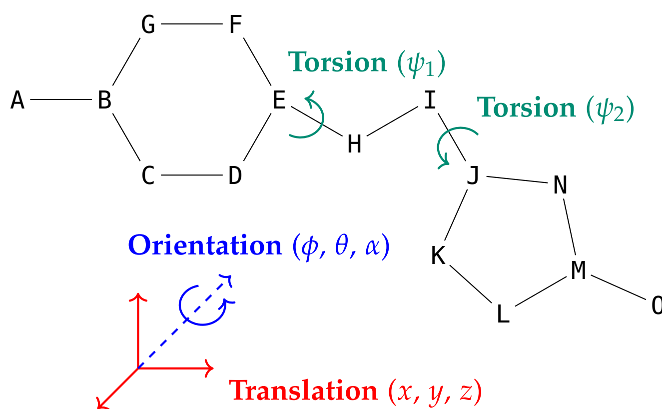


Figure 2.1: Degrees of freedom of a theoretical ligand composed of atoms A, B, C, ..., O. Bonds between atoms are depicted as connecting lines. Each rotatable bond (E–H and I–J) corresponds to a torsion, i. e., rotation of affected ligand atoms around the rotatable-bond axis.

2. The internal body flexibility models the rotation allowed for specific atomic (rotatable) bonds, which results in rotating ligand fragments around such bond axes. If a ligand possesses N_{rot} rotatable bonds, each of these can be represented with a torsional variable ψ.

The full set of $N_{\text{rot}} + 6$ variables describes the degrees of freedom of the ligand, and collectively, they constitute the encoded *solution* Ω to be optimized during the MD process:

$$\Omega = x,\ y,\ z,\ \phi,\ \theta,\ \alpha,\ \psi_1,\ \ldots,\ \psi_{N_{\text{rot}}} \tag{2.1}$$

Each solution corresponds to a different ligand pose, whose quality – from the biophysical standpoint – is evaluated by a scoring function:

The lower *values of the scoring function, the* stronger *ligand-receptor interactions.*

$$\text{SF}\left(x,\ y,\ z,\ \phi,\ \theta,\ \alpha,\ \psi_1,\ \ldots,\ \psi_{N_{\text{rot}}}\right) \tag{2.2}$$

Moreover, a term typically used in MD is the so-called *conformation*. A conformation refers to a change in the molecular structure that happens only when some angles between bonds are altered. Putting this in terms of degrees of freedom (translation, orientation, rotatable bonds): a *conformation* is defined by the rotatable bonds, while a *pose* is defined by the entire set of degrees of freedom.

2.2.2 *Lamarckian Genetic Algorithm*

The Lamarckian Genetic Algorithm (LGA) is the optimization method used to generate ligand poses for exploring the landscape described by the scoring function SF. An LGA (Algorithm 1) hybridizes the principles of biological evolution by coupling a genetic algorithm (GA) used as a global search method, with a local search (LS) method for refining

the poses initially identified by the GA. The *Lamarckian* denomination implies that poses whose scores were improved by LS iterations are *re-introduced* into the genetic population. Each member of a population is an *individual*, which is represented by its *genotype*, i. e., set of *genes*. New individuals are generated by genetic evolution (i. e., via crossover, mutation, and selection rules) from ancentors (i. e., individuals from a previous generation). A population subset is subjected to an additional LS optimization, which in AutoDock is based on the Solis-Wets method [175]. The LGA execution stops when a pre-defined maximum number of score evaluations (default: $N^{\text{MAX}}_{\text{score-evals}}$ = 2 500 000) or generations (default: $N^{\text{MAX}}_{\text{gens}}$ = 27 000) is reached, whichever comes first.

Algorithm 1: Lamarckian Genetic Algorithm (LGA)

```
Program AutoDock
    /* High-Level Parallelism */
    for each LGA-run do
        while (N_score-evals < N^MAX_score-evals) and (N_gens < N^MAX_gens) do
            /* Medium-Level Parallelism */
            GA (population)

            /* Medium-Level Parallelism */
            for individual in random-subset (population) do
                LS (get-genotype (individual))
```

The usefulness of biological evolution in MD optimizations relies on the mapping between these concepts, summarized as follows:

1. Each gene corresponds to a specific ligand motion variable (x, y, z, ϕ, θ, α, ψ_1, $\ldots$, $\psi_{N_{\text{rot}}}$).

2. A genotype – i. e., full set of N_{genes} variables – represents a pose (the encoded solution Ω).

Relation between number of genes and rotatable bonds: $N_{genes} = N_{rot} + 6$.

2.2.3 *Solis-Wets local search*

The LGA comprises the Solis-Wets local search (LS) method (Algorithm 2) that subjects a population subset of randomly-chosen individuals (`lsrate`, default: 6 %) to an adaptive-iterative process that aims to improve (i. e., minimize) their scores. In this method, new genotypes (`new-genotype1` or `new-genotype2`) are generated either by adding or subtracting small random changes (`delta`) to each gene of an initial genotype. New genotypes are stored if their scores (calculated by SF) are lower than those of a previous genotype.

At each iteration, the change in `delta` (`step`) is either increased or decreased, depending on whether the number of consecutive successful or unsuccessful search attempts is greater than four, respectively. Simi-

lar to LGA, the Solis-Wets LS termination is runtime-defined. Specifically, it finishes when either the number of LS iterations or change in `delta` reach their maximum (default: $N_{\text{LS-iters}}^{\text{MAX}}$ = 300) or minimum (default: step^{MIN} = 0.01) limits, respectively.

Algorithm 2: Solis-Wets local search

```
/* Low-Level Parallelism */
Function Solis-Wets (genotype)
  while (N_LS-iters < N_LS-iters^MAX) and (step > step^MIN) do
    // delta
    delta = bias + CONSTANT * random() * step

    // new-genotype1
    for each gene in N_genes do
      newgene1 = gene + delta
    if SF (new-genotype1) < SF (genotype) then
      genotype = new-genotype1
      success++
      fail = 0
    else
      // new-genotype2
      for each gene in N_genes do
        newgene2 = gene - delta
      if SF (new-genotype2) < SF (genotype) then
        genotype = new-genotype1
        success++
        fail = 0
      else
        success = 0
        fail++

    // step
    if success >= 4 then
      step *= EXPANSION-FACTOR
      success = 0
    else if fail >= 4 then
      step *= CONTRACTION-FACTOR
      fail = 0
```

2.2.4 *Scoring function*

The chemical interactions between a ligand and receptor are quantified with a semi-empirical free-energy force field (kcal/mol):

$$\text{SF} = \sum_{i,j} \left[W_{\text{vdw}} \left(\frac{A_{ij}}{r_{ij}^{12}} - \frac{B_{ij}}{r_{ij}^{6}} \right) + W_{\text{hb}}\, E(t) \left(\frac{C_{ij}}{r_{ij}^{12}} - \frac{D_{ij}}{r_{ij}^{10}} \right) + W_{\text{el}} \left(\frac{q_i q_j}{\epsilon(r_{ij}) r_{ij}} \right) + W_{\text{ds}} \left(S_i V_j + S_j V_i \right) e^{\frac{-r_{ij}^2}{2\sigma^2}} \right] + W_{\text{rot}}\, N_{\text{rot}} \tag{2.3}$$

The scoring function, denoted as SF in Equation 2.3, is composed of five terms. The first four terms comprise the summation performed over all pairs of ligand and receptor atoms. These terms are: *van der Waals* (dispersion/repulsion), hydrogen bonding, electrostatics, and desolvation. The fifth term predicts the (unfavorable) loss of ligand entropy binding due to the N_{rot} rotatable bonds. All terms are characterized by dimensionless coefficients, constant look-up tables, and other parameters. More importantly, the score is mainly determined by the interatomic distance r_{ij} between atoms i and j, which is calculated at runtime. Additional details of the aforementioned terms are provided as follows:

Note that in this work, the term energy *refers to either the binding energy (kcal/mol), i. e., the scoring function value associated with a ligand pose; or the compute energy (Joules) consumed during the MD program execution. Here, we mean binding energy.*

- The dimensionless weighting constants W_{vdw}, W_{hb}, W_{el}, W_{ds}, and W_{rot} are empirically determined using linear regression on a set of ligand-receptor inputs with known binding constants.
- The following coefficients depend on the atom types:
  - A_{ij} (kcal/mol Å^{12}) and B_{ij} (kcal/mol Å^{6}) correspond to the *Lennard-Jones* (12-6) potential [103] between neutral atoms i and j.
  - C_{ij} (kcal/mol Å^{12}) and D_{ij} (kcal/mol Å^{10}) correspond to the *hydrogen bonding* (12-10) potential [78] between hydrogen-bond acceptor and donor atoms i and j.
  - S and V are respectively the *solvation parameter* [176], and the *atom volume* [191] that shelters it from a solvent.
- The dimensionless directional weight $E()$ of the angle t provides directionality from ideal hydrogen bonding geometry.
- Atomic charges q_i and q_j of atoms i and j, respectively.
- The dielectric function $\epsilon()$ of the interatomic distance r_{ij} (between atoms i and j).
- An independent constant: $\sigma = 3.5$ Å.

Both Equation 2.2 and Equation 2.3 represent the *same* scoring function. Particularly, Equation 2.2 is a function of the genetic degrees of freedom of the ligand (x, y, z, ϕ, θ, α, ψ_1, $\ldots$, $\psi_{N_{\text{rot}}}$), whereas Equation 2.3 depends mainly on the interatomic distances (r_{ij}), determined in turn by the three-dimensional coordinates of atoms i and j. In other words, Equation 2.2 and Equation 2.3 are expressed in two different spaces, *genetic* and *atomic*, respectively. This observation highlights the relevance of using biological evolution in AutoDock since it maps the MD-problem from the atomic space (three-dimensional coordinates of all N_{atom} ligand atoms) into the genetic space (genotype Ω), hence considerably reducing the number of variables to be optimized

from $N_{\text{atom}} \times 3$ (tens or hundreds) down to N_{genes} (= $N_{\text{rot}} + 6$, with $N_{\text{rot}}^{\text{MAX}} = 32$).

Furthermore, the overall molecular interaction can be expressed as the sum of *two* independent interactions based on the ligand and receptor group of atoms. This expression for the SF (Algorithm 3) performs a pose calculation first, which inputs a genotype and outputs a set of three-dimensional coordinates for all ligand atoms, iterating over all $N_{\text{pose-rot}}$ rotation items. Thereafter, the interatomic distances are processed by the (aforementioned) two independent interactions, which are described as follows:

1. *Intermolecular* interactions could be computed analytically using Equation 2.3. However, since the number of ligand-receptor atom pairs is typically large (i. e., thousands), the analytical calculation is thus replaced by a trilinear interpolation of pre-calculated grids (Figure 2.2) that model the contribution of the receptor for each ligand atom-type [116]. This is achieved by using the *AutoGrid* program (part of AutoDockTools [117]), which calculates interaction energy maps with a default resolution (i. e., grid spacing) of 0.375 Å, and hence speeds up intermolecular interaction estimates compared to pairwise methods. This component iterates over all N_{atom} ligand atoms.

2. *Intramolecular* interactions within the ligand can be calculated using Equation 2.3 as well, but similarly to the intermolecular component, these interactions (i. e., ligand-ligand) are pre-calculated for all $N_{\text{intra-contrib}}$ intramolecular contributor pairs and stored in one-dimensional look-up tables.

Algorithm 3: Scoring Function (SF)

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

Within the receptor, there exist as well *intramolecular* interactions (i. e., receptor-receptor), which are constant since the receptor 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.

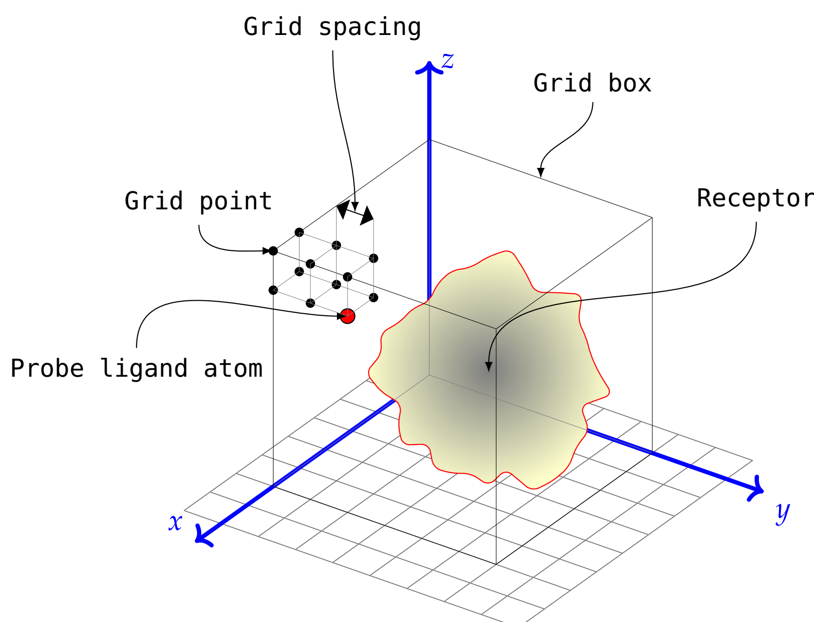


Figure 2.2: A grid-based approach is used for speeding up the calculation of intermolecular interactions. AutoGrid pre-calculates grid maps of interaction for various ligand atom types. Afterwards, AutoDock interpolates certain grid values to calculate the total intermolecular interaction.

2.2.5 *AutoDock algorithm*

The operation of AutoDock is depicted in Figure 2.3. Its overall MD process starts reading and parsing input files, which include:

- The three-dimensional structures of both ligand and receptor molecules, described in the standard *.pdbqt* file format.
- The runtime parameters – e. g., number of LGA runs, population size, maximum number of: score evaluations ($N^{MAX}_{\text{score-evals}}$) and generations ($N^{MAX}_{\text{gens}}$), grid maps, etc – listed in the AutoDock *.dpf* docking parameters file.

The actual AutoDock computations take place within every LGA run, for which a sequence of four main steps (`Step 1`, . . . , `Step 4`) is executed during global (genetic algorithm) and local search (Solis-Wets).

- `Step 1` generates new individuals, represented by their genotypes, under different rules depending on the LGA phase – either global or local – being executed. During global search, genotypes are subjected to genetic operations (e. g., crossover, mutation, and selection). During local search, new genotypes are generated according to the Solis-Wets method, i. e., by adding or subtracting small delta variations to their current values.

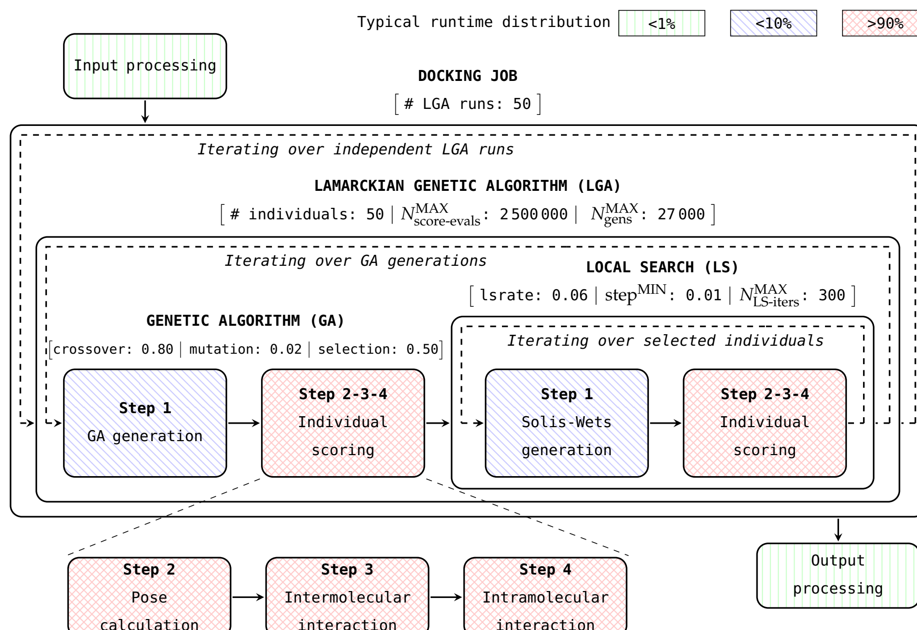


Figure 2.3: AutoDock block diagram [233] with default values of LGA parameters.

- `Step 2`, `Step 3`, and `Step 4` calculate the ligand pose (from a genotype), the intermolecular, and intramolecular interaction, respectively. This sequence of steps is repeated for every genotype, and is limited by either the maximum number of score evaluations ($N^{MAX}_{score\text{-}evals}$) or generations ($N^{MAX}_{gens}$) within every LGA run. All invokations to these steps account *together* more than 90 % of AutoDock execution time.

Finally, once all LGA runs have been executed, their *best* resulting ligand poses and corresponding scores are written to an output *.dlg* docking log file.

2.2.6 *Validation*

The validation criteria presented here is also used to analyze the quality-of-results in later chapters.

Since an LGA involves heuristics, the AutoDock outputs should be further analyzed in order to assess the functional correctness of the MD simulation. From the many different protocols [76] possible for validating such correctness, the experiments performed in this work are the so-called *re-docking* studies. In that approach, *already studied* ligand-receptor inputs are docked again. This allows a comparison between well-known reference solutions, and the ones obtained by implementations under test. The typical validation procedure [117] is based on the following criteria:

- Lowest binding score (LBS): refers to the best score – or lowest binding energy (in kcal/mol) – found among all executed LGA runs.

- Root mean square deviation (RMSD): estimates the geometrical deviation (in Å) of the resulting ligand (of an LGA run) with respect to a known-good reference pose. An RMSD equal to zero would be a perfect match between the resulting solution and its corresponding ground-truth x-ray crystallographic structure. For validation, LGA runs whose resulting ligand poses have RMSD within just 2 Å from each other are grouped into *clusters*.
- Size of best cluster (SBC): the best cluster is the one containing the LBS pose. Larger clusters are *better* because they indicate the ability of an MD program to find more *geometrically-similar* poses.

3

RELATED WORK

This chapter covers revelant observations and achievements from studies using OpenCL (especially for FPGAs), and compute-energy analysis for heterogeneous systems. Moreover, it reviews recent studies on execution- and algorithmic-performance enhancements of accelerated MD.

3.1 OPENCL PROGRAMMING AND ENERGY-EFFICIENCY ASPECTS OF HETEROGENEOUS SYSTEMS

3.1.1 *Performance portability of OpenCL*

In general, while portability can have different meanings [217]; from an application-efficiency perspective, performance portability is understood as the program capability of achieving good performance accross multiple architectures [171]. Recent studies on this topic quantify performance portability through the *Pennycook* metric [75, 217], analyze the performance achieved with different configuration knobs [147, 148, 168, 222], and evaluate such portability of parallel programming models based on empirical measurements [108, 177]. Due to the fact that computer architectures diversify in order to provide systems with higher performances, performance portability is a becoming a more desirable feature [223].

Among several parallel programming models, the Open Computing Language – OpenCL [64] – is one of the *most functionally* portable [108]. OpenCL provides a standard for writing parallel programs that execute across heterogeneous platforms consisting of a general-purpose processor (*host*) coupled with specialized accelerators (*devices*) such as many-core GPUs, dense multi-core CPUs, digital signal processors, and recently, hardware-reconfigurable FPGAs.

In order to understand how performance portable OpenCL is, applications from different domains have been used as case studies in heterogeneous computing. **Pennycook et al.** reported the development of wavefront [148] and molecular dynamics [147] kernels, showing the impact of work distribution, memory-access patterns, and *single instruction, multiple data* (SIMD) utilization. **Zhang, Sinclair, and Chien** [222] identified tuning metrics (e.g., memory layout,

prefetching/caching) critical to performance of three benchmarks: matrix multiply, sparse vector multiply, and fast Fourier transform. **Martineau, McIntosh-Smith, and Gaudin** [108] assessed the performance portability of emerging parallel programming models against the mature CUDA and OpenCL, porting a mini-application that solves a heat conduction equation. Besides the aforementioned *relatively small* applications, **McIntosh-Smith et al.** [110] analyzed the performance portability of an OpenCL implementation of the BUDE docking engine [57].

More details on the parallelization of BUDE [57] are provided in Section 3.1.3 and Section 3.2.2.

From previous studies, there is a clear sign: while OpenCL is indeed functionally portable, its performance portability is not guaranteed [108, 147, 148]. In fact, in order to maximize the performance when porting an *abstract* OpenCL code to different accelerators, it is critical to consider the *underlying hardware* of the target. Examples of the required *awareness* include, e. g., using target-specific memory accesses for exploiting: local memories on GPUs, cache hierarchies on CPUs, fully-customized pipelines on FPGAs. These design considerations can be expressed with *coding techniques*, e. g., array-of-structs (AoS) vs. struct-of-arrays (SoA) memory layout [148], or data vs. task parallelization [93, 218, 227]. Besides that, target-dependent optimizations result in typically longer development cycles, which could be further extended when working with domain-specific applications tackling real-world problems, like those in MD.

3.1.2 *OpenCL for FPGA programming*

Traditionally, FPGA programming has been an exclusive domain of hardware developers, which implement high-performance designs reasoning at the Register-Transfer Level (RTL). Working at this level requires developers to specify a number of hardware blocks performing concurrent low-level operations synchronized with a clock signal. These blocks communicate with each other through wires. Depending on the logical values transferred, these wires can control the internal state of such hardware blocks. Since the invention of FPGAs in 1985 [215], developers have been following this RTL-based approach for describing hardware. However, due to the concurrent nature and complexity of hardware blocks used in modern designs, it is in practice *very difficult* to reason in terms of transfers between hardware registers.

The development flow for FPGAs comprises several steps [72]. Among these, *synthesis* and *place & route* are automated by computer-aided design (CAD) tools. However, other steps like *design specification* (based on RTL descriptions), *system integration* and *verification* must be carefully handled by developers themselves. From the standpoint of a hardware developer, while the burden of the overall development

can be alleviated *somehow* by using Intellectual Property (IP) cores (e. g. [82, 137, 212]), it is still a time-consuming process.

In fact, as indicated in the industrial survey in [79], the productivity of a hardware developer (∼dozens lines of code per day) is *lower* compared to that of a software programmer (between 10 – 100 lines of code per day). Moreover, as pointed out by **Kapre and Bayliss** [89], since the number of software programmers is larger than that of hardware developers (in a ratio of 10x), it is then *more expensive* to develop an application on FPGAs than its equivalent software counterpart.

In order to increase productivity, the design-specification approach for FPGAs known as High-Level Synthesis (HLS) has gained increasing interest in recent years. Compared to traditional RTL-based languages or HDLs (e. g., VHDL, Verilog, System Verilog), HLS languages (e. g., Handel-C [112], LegUp [194], Vivado HLS [213], OpenCL [64, 83, 210]) require significantly *smaller* and *easier-to-understand* codes to describe a given functionality. In this way, HLS aims to increase productivity, and in turn, enables programmers without a deep knowledge of the underlying FPGA architecture and HDLs to harness the performance and power efficiency of such devices.

Studies using OpenCL for accelerating applications on FPGAs are described as follows. Since there are several studies meeting this criteria, our selection was made on the basis of key design patterns or programming techniques that are relevant for the work of this thesis.

As discussed in Chapter 5, a useful design approach is task-based parallelism, which is supported so far only in OpenCL and SYCL [66] (among high-level descriptions).

Zohouri et al. [227] evaluated the performance and power requirements of six Rodinia benchmarks targeting a Stratix-V FPGA, against a Tesla K20c GPU and a Xeon CPU. The effectiveness of specific optimizations (e. g., sliding windows) on FPGAs is reflected in a 3.4x power-efficiency superiority over the above GPU, and better runtime and power efficiency over the above CPU. The authors highlight that the OpenCL implementation of FPGA-specific strategies is *completely different* from common OpenCL strategies on GPUs.

Later in [226], **Zohouri, Podobas, and Matsuoka** combined *spatial and temporal blocking* for accelerating two- and three-dimensional stencil computations using OpenCL. Respective designs achieved compute performances of 760 and 375 GFLOP/s on an Arria 10 FPGA running at frequencies of 256 MHz. These results rival those achieved on a GTX 980Ti GPU (performance- and energy-wise), and on a Tesla P100 GPU (energy-wise).

Kenter and Plessl [94] developed a Finite-Difference Time-Domain (FDTD) solver for photonic microcavity simulations. Their OpenCL implementation consists of multiple *read*, *compute* and *write* kernels communicated through pipes. The read/write kernels transfer input/output from/to the external DDR3 memory, whereas the compute kernel performs the actual FDTD calculation. The latter kernel was pipelined into 36 stages, which passed data *only* through on-chip memory regions. On a Virtex 7 FPGA, this design runs at 140 MHz

DDR3 stands for Double Data Rate Type 3. *It is a type of synchronous dynamic RAM [41].*

and reaches processing rates higher than ∼1600 Mcells/s. This clearly outperforms an OpenMP implementation running eight threads on a Xeon E5620 CPU, which reaches a maximum of ∼500 Mcells/s for more than 2^{20} grid points.

In a further work [93], **Kenter, Förstner, and Plessl** parametrized their FDTD implementation, making their OpenCL design achieve portability and flexibility across FPGA platforms from *different vendors* by using pre-processor macros. Authors state that while some pre-processor macros are tedious to maintain, these drawbacks may be mitigated by future OpenCL-to-FPGA tool releases.

Yang et al. [218] examined design patterns consisting 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 hand-shaking 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.

The research by **Sanaullah and Herbordt** [163] presents a non-conventional usage of intermediate HDL files generated with the Intel FPGA OpenCL tool. Basically, the HDL code generated out of an initial compilation step – of a user's OpenCL code – is extracted and utilized in a custom development flow capable of more accurate resource/latency estimates. Using this strategy on benchmarks from the Rodinia suite and molecular-dynamic codes results in designs achieving speedups around {37x, 4.8x, 3.5x} over {OpenCL designs on FPGAs, GPUs, and Verilog designs on FPGAs}, respectively.

3.1.3 *Compute-energy efficiency*

While the surveys by **Mittal and Vetter** [114, 115] and **Bridges, Imam, and Mintz** [24] *comprehensively* discuss several methodologies for power measurement and energy analysis for GPUs and CPUs, our intention here is to list relevant studies where the performance and energy efficiencies of hardware-accelerated scientific applications were evaluated. Examples include sparse and dense linear algebra operations [14], partial differential equations solvers [204], biomolecular and cellular simulations [180]. Some authors analyzed the energy consumption of benchmarks (e. g., LINPACK [232]) comparing different programming models [60], while others compared the energy consumed on Intel and ARM systems running high-energy physics calculations [2]. Additionally, the previously-mentioned study by **McIntosh-Smith et al.** [109] reported the power costs and carbon emissions of an OpenCL implementation of the BUDE docking engine [57].

The power and energy efficiency of OpenCL designs running on FPGAs has been recently studied too. In [204], **Weller et al.** imple-

mented partial differential equations, and explored a set of generic and specific optimizations techniques using OpenCL. Comparison of energy efficiencies (in MB/J) showed that a GeForce GTX 980 GPU is ∼2x more efficient than a Stratix V FPGA when running the above differential solvers. Moreover, the previously discussed studies from **Zohouri et al.** [226, 227] performed energy-efficiency analysis using power draws estimated via FPGA-compilation tools. Finally, **Davis et al.** [38] presented a tool that allows OpenCL programmers to query *live* kernel-level power consumption using calls invoked from within the OpenCL host code. The support provided by such tool was limited to *embedded* platforms equipped with Cyclone V FPGAs.

3.2 PARALLEL IMPLEMENTATIONS OF MOLECULAR DOCKING TOOLS

This section aims to provide a significant description of research efforts that parallelize different MD tools using heterogeneous systems. To ellaborate this section, we have reviewed the *most* relevant surveys.

The survey by **Pechan and Fehér** [145] – published in 2012 – provides an overview of MD-acceleration that focuses on strategies targeting heterogeneous systems based on either GPUs and FPGAs. However, since during the course of last years other MD tools have been developed, we observed that [145] is becoming limited. This limitation is not merely with regards to the number of newer MD tools that should be included in such a survey, but more importantly, because recent programming approaches and parallelization strategies are innovative too.

Besides heterogeneous systems, a more recent survey by **Dong et al.** [42] – published in 2018 – covers other acceleration approaches targeting clusters, supercomputers, and even cloud computing systems. We find that [42] is somehow limited too, especially in its section dedicated to heterogeneous systems, which covers few GPU-based approaches.

In general, due to the variety in parallelization possibilites of different MD algorithms, we believe it would be *very* difficult to come up with an *exhaustive* survey on MD-acceleration. In fact, the attempt of this section is not to provide an ample review, but one that is sufficiently detailed to understand the state of the art. This section is based on the survey in [145], and complemented with key information from [42], as well as our own literature review. The work presented in this thesis, i. e., OCLADock, is *contextualized* within this section, and its details are extensively discussed in the following chapters.

Table 3.1 organizes the studies discussed in our review of MD-acceleration into five categories:

1. FFT-based tools

2. Evolutionary-based tools
3. AutoDock
4. Pairwise potentials
5. Intrinsically-parallel tools

Furthermore, since the scope of this thesis is heterogeneous systems within a single compute node, alternative/complementary approaches are not deeply discussed here, but included in an overall discussion in Section 3.2.6.

3.2.1 *FFT-based tools*

In ZDOCK [26], the attractive and repulsive interactions are grouped into a single term representing the so-called shape complementarity.

ZDOCK is a tool for docking rigid molecules that uses a Fast Fourier Transform (FFT) algorithm [27] to optimize a force-field scoring function [26] composed of desolvation, electrostatics, and (grid-based and pairwise) *shape complementarity* terms. While the original ZDOCK implementation [26] was written in C and parallelized using Message Passing Interface (MPI [107, 190]), the efforts discussed as follows target heterogeneous platforms.

In [197], instead of executing the original FFT-based search on floating point numbers, a three-dimensional correlation in spatial domain is implemented. This enables a long pipeline and low-precision arithmetic, which are suitable for FPGAs. This was an initial work that focused *only* on the search method that looks for the best shape-matches through correlation of voxel data, which at this point carried only two bits that: distinguish molecule interiors from exteriors, and mark the surface of molecules. The improved version in [198] uses tuple data-types for voxels in order to carry additional information, e. g., force-field interactions. In both versions, the core of the correlation architecture is a three-dimensional *systolic array* of cells. In such array, a voxel of the ligand is stored within each cell, while the receptor is stored in external memory. Speedups achieved on a Virtex-II FPGA were in a range between [100x, 1 000x] compared to a 3.0 GHz Xeon CPU.

Voxel, i. e., a volumetric pixel, represents a value on a regular grid in three-dimensional space.

PIPER [100] is a tool for rigid-molecule docking that, similar to ZDOCK, adds a pairwise desolvation term into the scoring function for improving its FFT-based search. However, a fundamental innovation

1 FFT: Fast Fourier Transform.
2 GA: Genetic Algorithm.
3 SW: Solis Wets.
4 DE: Differential Evolution.
5 ACO: Ant Colony Optimization.
6 NMS: Nelder and Mead algorithm.
7 ILS: Iterated Local Search, the *global* search method in Vina [195].
8 BFGS: Broyden-Fletcher-Goldfarb-Shanno.
9 HPL: Heterogeneous Programming Library.

Table 3.1: Parallelized MD tools. Fields specified with a "–" mean the program/feature was not existent/not implemented.

Original MD tool	Parallelized version	Release Year	Scoring function	Global & Local search method	Target accelerator	Description language
ZDOCK [26]	Van Court et al. [197, 198]	2004, 2006	–	3D correlation & –	FPGA	VHDL
PIPER [100]	Sukhwani et al. [182]	2009	Force-field	FFT[1] & –	GPU	CUDA
PIPER [100]	Sukhwani et al. [181, 183]	2008, 2010	Force-field	3D correlation/-	FPGA	VHDL
Katchalski-Katzir et al. [91]	Feng et al. [48]	2010	Force-field	FFT & –	GPU	CUDA
Hex [156]	Ritchie et al. [157]	2010	Force-field	FFT & –	GPU	CUDA
MolDock [193]	Simonsen et al. [173]	2013	Force-field	DE[2] & –	GPU/CPU	CUDA/OpenMP
PLANTS [97]	Korb et al. [98]	2011	Empirical	ACO[3] & NMS[4]	GPU	OpenGL & Nvidia Cg
BUDE [57]	McIntosh-Smith et al. [110]	2014	Force-field	GA & –	GPU/CPU	OpenCL
–	Altuntaş et al. [9]	2016	Force-field?	GA & –	GPU	HPL[5]
AutoDock [116]	Kannan et al. [88]	2010	Force-field	GA & –	GPU	CUDA
	Pechan et al. [146]	2010	Force-field	GA[6] & SW[7]	FPGA	Verilog
	Pechan et al. [144]	2011	Force-field	GA & SW	GPU	CUDA
	Mendonça et al. [111]	2017	Force-field	GA & –	GPU + CPU	CUDA & OpenMP
	OCLADock [165, 233, 234]	2017, 2018, 2019	Force-field	GA & SW/ADADELTA	GPU/CPU/FPGA	OpenCL
–	Roh et al. [158]	2009	Force-field	–	GPU	CUDA
–	Guerrero et al. [68]	2011	Force-field	–	GPU	CUDA
–	Saadi et al. [159, 160]	2017, 2019	Force-field	–	GPU/CPU	CUDA/OpenMP
AutoDock Vina	Trott et al. [195]	2009	Empirical	ILS[8] & BFGS[9]	CPU	C++
AutoDockFR	Ravindranath et al. [154]	2015	Force-field	GA & SW	CPU	Python & C++

in PIPER is the approximation of the pairwise-interaction matrix by an *eigenvector* decomposition. Low eigenvalues are often discarded, and thus, the computational complexity is reduced.

The PIPER implementation for FPGAs by Sukhwani and Herbordt [181] extends their systolic-array architecture previously used for ZDOCK [197, 198] (described above) in order to support the docking of *two* large molecules (i. e., protein-protein). Contrary to their previous design where the ligand is stored within the array cells, now both molecules are stored in external memories. Moreover, since PIPER must calculate multiple correlations, the basic cell structure has been *augmented* with: a new weighted scorer module that sums the partial correlation results, and new FIFOs to propagate the scorer-module output to the input of the next one. Experiments were perfomed on a Virtex-II FPGA [181], and then on a Stratix-II EP2S180 FPGA [183].

FIFO stands for the first-in, first-out *method used for manupulating a data buffer.*

Using the Nvidia CUFTT library [126], Sukhwani and Herbordt developed a PIPER version for GPUs [182], in which FFT computations were performed directly instead of using correlation as for the FPGA counterparts. On a Tesla C1060 GPU, a similar execution performance was achieved with two alternative parallelization strategies, which consisted in assigning either a complete or a portion of a two-dimensional FFT-plane to a thread block. Overall, for small sizes of the ligand grid (i. e., 4, 8), speedups on FPGA (∼36x) are higher than those on GPU (< 33x). However, for larger ligand-grid sizes (i. e., 16, 32), while speedups decrease for any device, speedups achieved on the GPU are higher (∼16x) than those on FPGAs (∼3x).

The work of **Feng, Tian, and Chang** [48] uses GPUs to accelerate rigid-molecule dockings. The scoring function is composed of two terms describing the molecular shape and electric fields, while the search is based on an FFT algorithm. Similar to the PIPER acceleration previously described, FFT-based operations are implemented using the CUFFT library. However, to the best of our efforts, it was not possible to clearly identify from their manuscript [48], what computations are carried out by CUDA threads. Experiments on a GeForce 9800GT GPU reach up to 4x of speedup compared to sequential executions on an AthlonX2 3600+ CPU.

The survey by Pechan and Fehér [145] indicates, however, that the GPU-based implementations of Feng, Tian, and Chang [48] and PIPER [182] are similar.

Hex [156] is a tool for protein-protein interactions based on FFT. In [157], Ritchie and Venkatraman describe their CUDA version of Hex. The difference with respect to most FFT-based MD tools is that, Hex uses grids expressed in *spherical polar* rather than *cartesian* coordinates. According to [157], FFT-based approaches that use a cartesian representation can compute *only* translational correlations, and must be repeated over multiple rotations to cover a six-dimensional search space. The CUFFT library is used for implementing the one- and three-dimensional FFT on a GeForce GTX 285 GPU, whose executions were ∼45x faster than the original Hex on a single CPU core.

3.2.2 *Evolutionary-based tools*

The **MolDock** software [193] is very similar to AutoDock. Its scoring function consists of the summation of pairwise force-field energy terms expressed as intermolecular (*van der Waals*, hydrogen bonds, electrostatics potentials) and intramolecular (*van der Waals*, hydrogen bonds, torsional, clash penalties) components. Its search method is based on a variant of an evolutionary algorithm (EA), called differential evolution (DE). Similar to genetic algorithms (GAs), DE is an optimization technique also inspired by the *Darwinian* evolution theory. However, DE uses a different approach to select and modify candidate individuals, i. e., DE creates new individuals from a *weighted difference* of parent individuals. The DE variant used in MolDock [193] performs no local search.

The GPU-based parallelization of MolDock developed by Simonsen et al. [173] is similar to that of AutoDock by Pechan and Fehér [144]. The difference is that, in [173] threads within the same block perform a task (e. g., interpolation) for the same ligand atom of different individuals, while in [144] threads within the same block perform such task for the different atoms of the same individual. The CPU-based parallelization also provided in [173] is much simpler as MolDock is parallelized on the genetic population level, distributing multiple (DE) runs over CPU cores. Experiments consisting in docking 133 ligands of different sizes on an Intel Core 2 quad core (Q9450) CPU @ 2.6 GHz and a Nvidia GeForce 8800GT GPU resulted on average speedups of 3.9x and 27.4x over a single CPU core, respectively.

PLANTS [97] uses an Ant Colony Optimization (ACO) as a search method. ACO is a swarm intelligence technique that mimics the behavior of real ants, e. g., when they (as a colony) find the shortest path between the nest and food source. Real ants lay down *pheromones* directing each other to resources while exploring the surrounding environment. Analogously, *simulated ants* in PLANTS [97] generate solutions by selecting one value for each degree of freedom taking into account *artificial* pheromone values. Pheromone levels are decreased for solutions of weak (unfavorable) molecular interactions. In order to improve the quality-of-solutions in PLANTS [97], authors introduce a local search based on the Nelder and Mead (NMS) algorithm [120], which is applied to *all* ants.

Pheromones are substances secreted to the outside by an individual and received by a second individual of the same species, in which they release a specific behavior or developmental process [90].

In the parallelization proposed in [98], multiple ant colonies are processed in parallel. The conformation generation and scoring function evaluation are executed on the GPU, while the overall optimization algorithm (ACO + NMS) runs on the CPU. According to authors, the adopted programming methodology based on OpenGL and Nvidia Cg was less flexible compared to the general-purpose CUDA or OpenCL, due to the restricted programming model offered by languages for graphic computations. Experiments on a Nvidia GeForce 8800 GTX

GPU and a single CPU core Pentium 4 @ 3.0 GHz reached speedup factors of up to 50x compared to an optimized CPU-based implementation.

BUDE is a MD engine developed by Gibbs, Clarke, and Sessions [57] at University of Bristol. Its scoring function is of force-field type composed of steric, electrostatic, and desolvation terms. Its search method is based on a GA, which similarly to that in AutoDock, creates successive generations of candidate solutions from best candidates of previous generations.

The preliminary work in [109] provides an initial OpenCL implementation of BUDE, where a single pose (each represented by an individual of the genetic population) is processed by a single OpenCL work-item. Later enhancements described in [110] comprise an increase of four poses processed per work-item, as well as optimization techniques like memory-access coalescing (i. e., using a structs-of-array memory layout), and code refactoring for reducing the negative performance impact of thread divergence (i. e., converting conditional branches into equivalent combinations of predicated selection and multiplication). Faster executions (∼60x) and larger energy savings (∼3x) were achieved on a system based on two Nvidia C2050 GPUs compared to an Intel Core2Duo SU9400 @ 1.4 GHz CPU, both systems running the same OpenCL-accelerated BUDE code. This code is used as a *baseline* for benchmarking the performance impact of optimizations introduced in [110], whose executions on an AMD FirePro S10000 GPU resulted in speedups of 5x compared to those of above OpenCL baseline.

The implementation by Altuntaş, Bozkus, and Fraguela [9] is assumed to employ a scoring function of force-field type. However, details of its scoring-function terms were not provided.

Altuntaş, Bozkus, and Fraguela [9] presented *their own* parallel MD algorithm based on a GA. It is implemented with the Heterogeneous Programming Library (HPL) [23], which is an open-source C++ framework [74] that provides an easy and portable way to exploit heterogeneous computing systems on top of OpenCL. The most computationally-expensive part of this algorithm is made of a consumer-producer chain of subroutines that perform population generation, score evaluation, and genetic operations (tournament selection, mating, and mutation). The total number of threads is equal to the population size. Within each subroutine, each individual (carrying a pose information) is processed by a single thread. The search method is only global and it is provided by the GA itself (i. e., no local search is present here). The experiments consisted of 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 2.1 GHz Xeon CPU core.

3.2.3 *AutoDock*

Here, we discuss the most relevat details of studies addressing the acceleration of AutoDock.

Kannan and Ganji developed a parallel implementation [88] of AutoDock that *excludes* the Solis-Wets local search (LS) method from the LGA (Table 3.1 indicates that this implementation provides only GA as global search method). The reason for not parallelizing the LS was to avoid the low GPU utilization that results when optimizing a only a subset instead of the full population of individuals, as in the GA. Their parallelization strategy consisted in assigning an individual to a CUDA thread block, whose threads execute their inner tasks cooperatively. Speedups of 47x were achieved when running the overall program on a Tesla C1060 GPU, compared to the original AutoDock executed on a 2.4 GHz single Athlon CPU.

In [88], slightly different results compared to AutoDock *were obtained due to the single-precision arithmetic used on GPUs.*

To this thesis work, the *most influential* previous studies have been those carried out by **Pechan, Fehér, and Bérces** in [144, 146], which *include* the LS for achieving a fully-operational program equivalent to AutoDock.

For GPUs, the work in [144] proposes an strategy where the independent LGA runs are executed in parallel, while each individual is processed by a CUDA thread block (similarly as in [88]). Also, GA and LS are assigned each to a different CUDA kernel. Particularly, instead of launching a CUDA thread block for each new individual (of every independent LGA run) as in the GA kernel, a CUDA thread block is launched for each *randomly-selected* individual in LS. Performance tests on a GeForce GTX 260 GPU compared to a 3.2 GHz Xeon CPU core (running the original AutoDock) resulted in a maximum speedup of 65x.

For FPGAs, a totally different strategy compared to that for GPUs is used to leverage the underlying fully-programmable architecture [146]. Basically, LGA runs are executed one at a time. Each of these runs is implemented as a three-stage pipeline that can process up to three individuals simultaneously. Although the *apparent* lower processing rate, this design was efficient due to its fine-grained pipelines. Executions on a Virtex-4 FPGA were 23x faster than a 3.2 GHz Xeon CPU core (running the original AutoDock). In terms of execution performance, the GPU outperforms the FPGA, if the number of LGA runs is larger than 20.

Similar to the PIPER versions for GPU [182] and FPGA [183], these two platforms are advantageous at different parameter ranges (e. g., number of LGA runs) for the parallel AutoDock *codes in [144, 146].*

Recently, **Mendonça et al.** proposed a hybrid parallelization of AutoDock. In [111], time-consuming computations are offloaded onto both a multi-core CPU and a GPU. Authors claimed that their hybrid design achieves higher speedups (80x), and thus, outperforming implementations using these accelerators separately, i. e., either a 2.2 GHz quad-core Xeon CPU (8x) or a Tesla C2050 GPU (35x). However, the code snippets in their manuscript indicate the usage of an erroneous

local-search algorithm. As discussed in Section 2.2.3, the local search in AutoDock implements the Solis-Wets algorithm, whereas the implementation in [111] performs genetic operations such as crossover, mutation, and selection. The latter might not affect *enormously* the quality-of-results, but *definitely* affects performance. This is because the Solis-Wets method is sequential and contains data-dependent operations, which hinder parallelization. On the contrary, the genetic operations mentioned above are intrinsically parallel. Despite the interesting approach, the latter observations make the results reported in [111] questionable.

3.2.4 *Pairwise potentials*

The following studies focus *specifically* on accelerating certain pairwise potentials, i. e., score terms that could be integrated into a more complete scoring function. However, these studies do not parallelize any *production* or *fully-functional* MD code.

The work by **Roh et al.** [158] accelerates *only* the calculation of the pairwise potentials in a scoring function composed of two terms: *van der Waals* and electrostatic interactions. What is particular in this work is the usage of a separate GPU for each of the aforementioned scoring terms. As discussed in [145], this strategy would be *impractical* for a real application due to large number of data transfers required between both GPUs and a host CPU.

In [68], the tiled design results in faster executions than the basic one. The speedup ratios between these two designs is not clearly explained, neither depicted in the plots provided.

Guerrero et al. described in [68] their strategy for parallelizing pairwise electrostatic interactions between a receptor and a ligand. Similar to the work in [158], this calculation was accelerated in isolation and not integrated into any MD code. The CUDA kernel implementation is straightforward: each thread computes the electrostatic interaction between its corresponding receptor atom and *all* ligand atoms. Additionally, authors implemented a tiled design in which ligand atoms are grouped into thread blocks. Experiments were performed on a Tesla C1060 GPU and a single core of a Xeon E5530 CPU. Their results indicate that speedups – achieving factors within the range of [10x, 260x] – increase with larger number of receptor and ligand atoms.

Recently, the studies by **Saadi et al.** [159, 160] have aimed to parallelize scoring-function computations used for blind docking. The blind-docking procedure consists in scanning the whole surface of a target protein, instead of just specific binding sites. It enables the discovery of *new* binding *pockets*, and thus, helps enhancing the MD quality at the expense of increasing exponentially the computational complexity.

In [159], authors parallelized their *own* sequential code based on the desolvation term in AutoDock (Section 2.2.4). This code is composed of three nested loops that iterate the desolvation calculation over a given number of spots on the protein surface (outermost), ligands atoms,

and receptors atoms (innermost). Authors provide a simple along with an optimized CUDA implementation. In the *simple* one, a thread block executes the computations associated to a single spot on the protein surface. Each thread (within a block) corresponds to a ligand atom, and computes its energy contribution with the entire set of protein atoms. In the *optimized* design, a small number of protein atoms are grouped into tiles. In this way, a thread (representing a ligand atom) calculates the energy contribution with a tile, instead of with just a single protein atom. Experiments on a Kepler K40m GPU resulted in speedups of 62x over a 4-core E3-1220 CPU, and 223x over a single CPU core. All experiments were performed using a compound with 100 spots on the protein surface.

The previous study was extended in [160], where simultaneous kernels are launched on a GTX 1080Ti GPU. Compared to a sequential version running on a single core of a 24-core 2.2 GHz Xeon E5-2650 CPU, their newer CUDA version reaches speedups of up to ∼213x. Moreover, an OpenMP version is also provided, reaching speedups of ∼15x on the aforementioned 24-core CPU.

3.2.5 *Intrinsically-parallel tools*

For these tools, the inherent parallelism of their algorithms was considered during code development. The target platforms are multi-core CPUs due to their ubiquity in HPC environments.

AutoDock Vina, or simply Vina [195], is a MD tool alternative to AutoDock, and developed as well at TSRI. From an algorithmic perspective, Vina is significantly different from AutoDock. The Vina scoring function is composed of steric, hydrophobic, hydrogen bonding, and rotatable-bond related terms. This function is empirical in constrast to the *possibly* too-strict score models based on force-fields used in AutoDock. Its global search consists of several Iterated Local Search (ILS) steps. Each ILS step comprises a local optimization consisting in the Broyden-Fletcher-Goldfarb-Shanno (BFGS) method, i. e., a quasi-Newton method based on second-order derivatives of the scoring function. The number of ILS steps in a run is determined adaptively, and several runs can be executed in parallel, each on a CPU core. Multi-threading in Vina is implemented using the `Boost::Thread` library [22], and thus, provides a portable C++ code for multi-core CPU architectures. Experiments in [195] reported that, besides its higher MD quality-of-results, Vina was on average ∼65x faster when running on an 8-core CPU machine compared to the original single-threaded AutoDock.

AutoDockFR [154] was developed in recent years at TSRI. Particularly, it models the *receptor flexibility* (as a set of side chains) and accounts it for ligand binding prediction. AutoDockFR addresses the growth of the search space with a customized scoring function and a

new GA. AutoDockFR and AutoDock can be compared in two aspects: scoring function and search method. Besides the AutoDock scoring terms discussed in Section 2.2.4, the scoring function in AutoDockFR includes additional terms that describe the interactions between flexible receptor atoms against {ligand, flexible receptor, and rigid receptor} atoms. Similar to AutoDock, AutoDockFR uses grid maps to speedup the calculation of score contributions of rigid receptor atoms. Regarding the search method, AutoDockFR uses a slightly different GA compared to that of AutoDock. The GA in AutoDockFR applies a local *minimization* step (i. e., a local search) to *every* individual in the population right after the mutation operation. Moreover, instead of clustering best individuals resulting from independent LGA runs after the entire MD has finished (as in AutoDock), AutoDockFR clusters individuals at every cycle within each GA run.

AutoDockFR is implemented in Python and by default, runs all GA runs in parallel, each executed on a single CPU core. While in terms of MD quality-of-results, AutoDockFR outperforms Vina on receptors with up to 12 flexible side-chains, it was reported as being ∼230x slower than Vina. More recent code updates [203], consisting in a C++ port of the scoring function and minimization step, have reported runtime improvements of ∼280x speedup compared to the originally published Python implementation [154].

3.2.6 *Other parallelization approaches*

Studies in this Section 3.2.6 do not target heterogeneous systems, and thus, are not included in Table 3.1.

Here, we comment briefly on approaches targeting a variety of systems that range between clusters, grid, and cloud computing. Since, it is not possible to introduce all existing approaches, readers are referred to the survey in [42] for additional details.

Norgan et al. [124] proposed the distribution of AutoDock jobs and their LGA runs, over a *cluster* with ∼eight thousand CPU cores. This is an *hybrid* solution that uses MPI and OpenMP at two different levels to accelerate virtual screening processes. MPI is used to parallelize the main function by distributing docking jobs accross cluster nodes, while OpenMP enables multi-threading within each job, i. e., at the level of LGA runs. The results show that their design scales *almost* linearly up to 8 192 cores, reaching speedup values of 8 192x over a single CPU code.

A similar hybrid approach is used by **Lang Yu et al.** that implemented VinaSC [102], which is a port of Vina onto a large-scale heterogeneous cluster. Compared to Vina that supports only single-node CPU platforms, VinaSC has ported Vina onto a multi-node system, each node comprising a host CPU and a MIC co-processor. MPI was used to implement the internode scheduler, whereas Pthread was used for managing the intranode computations. The experimental system had six nodes, and the baseline execution distributed Vina instances

only on host CPUs (for all nodes). Compared to such baseline, VinaSC achieved 2.3x speedup when the number of docking jobs is 3 600.

Another important usage of AutoDock and Vina is in the project known as **FightAIDS@Home** [50]. Launched in 2000, FightAIDS@Home is an *distributed-computing* approach that allows volunteer users to contribute with idle resources within their computing systems to accelerate research into new drug therapies for the *human immunodeficiency virus* (HIV), which causes the *acquired immune deficiency syndrome* (AIDS). In 2005, FightAIDS@Home joined the **World Community Grid** [207], a public internet-based distributed-computing infrastructure devoted to active projects that benefit humanity, including efforts combating Zika, cancer, tuberculosis, etc [206].

According to [50], FightAIDS@Home is the first biomedical distributed-computing project ever launched.

More recently, **De Paris et al.** have utilized AutoDock as the MD engine in cloud-based systems in Amazon [39]. In such study, their purpose is to reduce the dimensionality of fully-flexible receptor models, and the overall docking execution time using HPC resources in the cloud. The access to such system was made through a cloud-based web environment called *wFReDow*, while at the background a new automated workflow dispatches docking jobs to cluster nodes. The job distribution was implemented using MPI and OpenMP, similarly to previously-discussed studies in [102, 124]. For the experiments in [39], authors used from one up to eight Amazon EC2 instances, increasing the core count from one up to 64. The maximum speedup achieved was ∼60x when using 64 CPU cores for running 3 100 MD tasks.

3.3 ALGORITHMIC IMPROVEMENTS IN MOLECULAR DOCKING

Besides increasing the accuracy of description models (e. g., scoring terms), employing more effective MD algorithms *might* result in significantly higher quality-of-results. In this section, we describe relevant studies evaluating the resulting quality of alternative local-search (LS) methods introduced into MD codes.

Here, we provide a qualitative discussion of algorithmic improvements. A quantitative evaluation of our own additions to AutoDock *is performed in Chapter 6.*

Tavares, Mesmoudi, and Talbi [188] incorporated a limited-memory version of BFGS (L-BFGS) [123] into the LGA of the original (and single-threaded) AutoDock code. Basically, their work performs an empirical analysis of the *upgraded* evolutionary algorithm, i. e., an LGA running an LS based on the L-BFGS method, instead of the legacy Solis-Wets. The quality of resulting molecular poses was assessed in terms of the lowest binding score (LBS), and spatial deviation (RMSD) (Section 2.2.6). Authors concluded that the legacy Solis-Wets method *might not* be most suitable LS component in evolutionary algorithms, since their experiments using the alternative L-BFGS resulted in poses with superior quality than those of Solis-Wets, and of a non-hybridized GA (i. e., without LS).

Fuhrmann et al. [53] presented a new LS method suitable for MD. Similar to [188], the work in [53] proposed a gradient-based score

Singularity is a point at which a given mathematical object is not defined. Non-differentiable functions are those whose derivative cannot be computed at certain values, i.e., singularity points [18].

minimization. Their proposal considers the singularities arising during the gradient-based optimization on poses represented as a set of translation, orientation, and torsions. Authors proposed the usage of *exponential mapping* for defining the molecular orientation, which in turn, eases the calculation of the *orientational* gradient. To avoid singularities, the LS is modified, so the orientational variables are changed while preserving the molecular orientation. This work uses the L-BFGS method and compares it against Solis-Wets. Contrary to the aforementioned work in [188] and our own additions (Chapter 6) that fully integrate LS methods into an MD tool (AutoDock, in this case), experiments in [53] evaluate LS methods *in isolation*. Despite that, the usage of gradients allowed reaching significantly lower (better) scores in fewer steps than Solis-Wets.

Afanasiev et al. [3] compared four algorithms by using them as LS methods integrated into an MD code running a *Monte Carlo* search. The selected algorithms were: L-BFGS, conjugate gradient, truncated Newton's method, and Powell's method. Contrary to the first three methods, the Powell's objective function does not need to be differentiable, and thus, no derivatives are taken. The Powell's method minimizes the objective function by using a bi-directional search. Besides the improvements in the search itself, authors used a more complex MD calculation. Examples of such additional complexity include a more rigorous description of the desolvation effects, and the full-flexibility allowed for ligands. Experiments showed that Powell's method remarkably outperforms the others in terms of LBS.

3.4 WRAP-UP DISCUSSION

This section provides an overall review and contextualizes our own work, OCLADock.

Previous studies on performance portability indicate that when using OpenCL, device-specific optimizations are required for achieving performance-portable implementations accross GPUs and CPUs. Besides these devices, it is also possible to target FPGAs using OpenCL as its high-level description can be *synthesized* into hardware blocks.

Although OpenCL promises higher code-productivity compared to HDLs, achieving high performance is still challenging due to the lack of *direct control* of low-level FPGA characteristics. In fact, there is still a gap in understanding which OpenCL constructs map well on FPGAs. This gap is reflected in cases where the performance obtained with OpenCL is lower in comparison to that achieved with HDLs.

Besides achieving performance gains out of a parallelized application, understanding its compute-energy savings is critical for efficiently deploying such application on HPC systems. This becomes even more important, when the application is used at scale, such as the massive drug discovery use-case MD is used for.

There are several studies about hardware-acceleration of MD tools. These differ by the type of search and scoring functions they implement. Among the studies on AutoDock discussed in this chapter, only those by Pechan and Fehér [144, 146] are truly comparable to OCLADock, as both implementations include the Solis-Wets local-search method. Excluding the local optimization from the overall evolutionary algorithm removes many data-dependencies, and thus, eases the parallelization. However, doing so can impact *negatively* the quality-of-results. Previous studies show that gradient-based methods outperform Solis-Wets in terms of MD quality. OCLADock includes the original Solis-Wets method, and goes beyond the work of Pechan and Fehér, as it includes alternative local-search methods based on gradients (Chapter 6).

Appendix B provides a benchmark between OCLADock *and other MD tools, including that of Pechan and Fehér [144].*

4

OCLADOCK: OPENCL-ACCELERATED AUTODOCK ON CPUS AND GPUS

This chapter details OCLADock, an OpenCL-based data-parallel implementation of AutoDock, and provides an evaluation of its performance and compute-energy efficiency. Contents of this chapter have been previously published in [233].

4.1 OPENCL IMPLEMENTATION OF AUTODOCK

4.1.1 *Data-based parallelization*

The data-parallel source code is available in the OCLADock *repository at [187].*

As previously shown in Figure 2.3, a docking job is composed of several LGA runs (default: 50), where each LGA run processes a large population of individuals (default: 50) through a genetic algorithm (GA), followed by a local-search (LS) refinement. By using their particular rules, the GA and LS functions generate new individuals and score them. Similarly to prior work [144], the proposed OpenCL parallelization consists of mapping main AutoDock functions to OpenCL processing elements (i. e., kernels, work groups, work items) according to a suitable level of parallelism. Since the GA and LS functions involve computations over large genetic populations, these are mapped to the `Krnl_GA` and `Krnl_LS` OpenCL kernels, respectively (Figure 4.1).

Data parallelism of AutoDock can be exploited at three different processing levels: *high*, *medium*, and *low*. First, based on the fact that a docking job consists of several independent LGA runs, the high-level parallelization consists of trivially performing these independent runs in parallel. Second, individuals from a single genetic generation (within an LGA run) are independent from each other, and thus, correspond to medium-level parallelism. Finally, the low-level parallelism correspond to fine-grained tasks that pertain to a single individual, such as calculating the ligand pose and evaluating the scoring function.

Figure 4.2 depicts the proposed data-based parallelization, which starts combining the high-level and medium-level strategies first. A docking job is composed of R LGA runs (Run_{ID}: 0, 1, 2, ..., R - 1), with each LGA run processing a population of P individuals (Ind_{ID}: 0, 1, 2, ..., P - 1). The execution of such runs, and the processing of their individuals, are controlled by nested loops in the serial im-

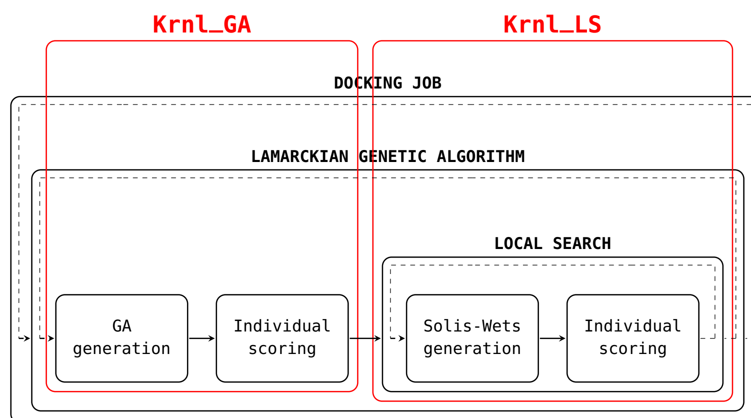


Figure 4.1: Mapping AutoDock – GA and LS – functions onto OpenCL kernels. Kernel blocks enclose nested loops controlling LGA runs, as well as GA and LS inner processing. Kernels operate over several individuals simultaneously, with each individual being mapped to a single OpenCL work-group.

plementation, which can be merged into a single loop for optimal parallelism. By doing so, individuals from different LGA runs can be processed simultaneously, each as an OpenCL work-group identified with a $\mathrm{WG_{ID}}$ obtained as follows:

$$\mathrm{WG_{ID}} = \mathrm{Run_{ID}} \cdot P + \mathrm{Ind_{ID}} \tag{4.1}$$

The entire set of $P \cdot R$ work-groups is distributed by the GPU runtime scheduler over the available Q compute units (CUs). Each CU executes a work-group associated with a single individual, achieving high-level and medium-level parallelization simultaneously.

Finally, processing an individual involves fine-grain tasks (genotype generation, calculation of ligand pose, intermolecular and intramolecular interactions) that can be assigned to OpenCL work-items, hence achieving low-level parallelization.

The intramolecular component of the scoring function (Section 2.2.4) could also be referred to as the pairwise interaction.

4.1.2 *Code architecture*

Figure 4.3 depicts the overall workflow of OCLADock. This consists of a sequence of functions executed either on the host (`Hx`) or on the device (`Dx`). After the application inputs are parsed in `H1`, the populations of *all* LGA runs are initialized with random values in `H2`. Then, the OpenCL setup takes place in `H3`, which comprises the identification and selection of platform and device, as well as the definition of other OpenCL objects such as *context* and *command queues*. This process includes the creation of a *program* object containing all machine-specific instructions to be executed on the device. Since the host is responsible for launching and keeping track of kernel executions, it needs to know how to interact with the kernels. Therefore, in `H4`, the OpenCL kernels

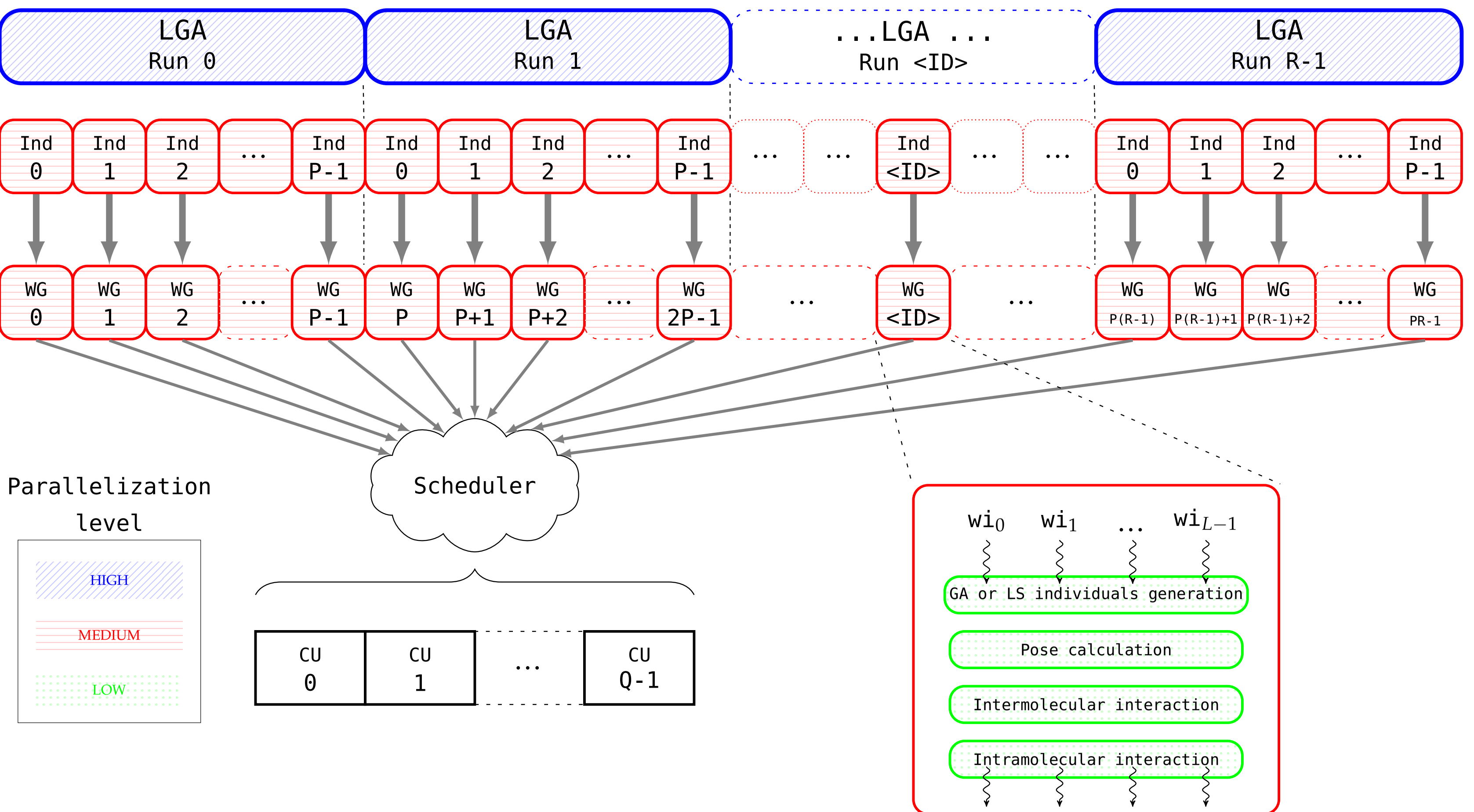


Figure 4.2: A population processed by an LGA run (Run_{ID}) can be decomposed into its individuals, and each individual (Ind_{ID}) can be mapped onto a work-group (WG_{ID}). The entire set of work-groups is distributed by the GPU runtime scheduler over the available Q compute units (CUs). A CU is a multi-threaded hardware unit capable of processing one work-group (composed of L work-items) at a time. The runs, individuals, and fine-grain tasks are colored according to their associated level of parallelism: high (blue), medium (red), and low (green).

(`Krnl_INIT`, `Krnl_EVAL`, `Krnl_GA`, `Krnl_LS`) are specified in terms of arguments (variables holding e. g., initial population values, number of genes (N_{genes}) and ligand atoms (N_{atom}), etc), global size (total number of work-items to be processed by the device), as well as local size (number of work-items to be processed within each work-group).

`Krnl_INIT` and `Krnl_EVAL` are auxiliary kernels that together consumed less than ∼5 % of execution time.

The first kernel executed is `Krnl_INIT`, which calculates the initial score of individuals from all LGA runs. The second kernel, `Krnl_EVAL`, counts the number of scoring function calls (stored in device memory) performed by previously-executed kernels. After `Krnl_EVAL` is executed, control is handed back to the host, which checks whether the LGA termination criteria are met, i. e., if the number of either score evaluations ($N^{\text{MAX}}_{\text{score-evals}}$) or generations ($N^{\text{MAX}}_{\text{gens}}$) reached their maximum values (Section 2.2.2). The core of the application is the iterative execution of kernels, with each cycle starting with `Krnl_GA`, then going through `Krnl_LS`, and finishing with `Krnl_EVAL`. While the inter-kernel synchronization is controlled entirely on the host via *in-order* command queues, the transfer of solution data and their scores between kernels occurs by device-side global-memory accesses. Finally, when the LGA termination criteria are met, the *final* solutions found by the device (and residing in its global memory) are copied back to the host, where results are written to output *.dlg* files compatible with AutoDockTools [117].

The `Krnl_LS` kernel executes either the Solis-Wets method as in the original AutoDock, *or any of the gradient-based methods newly added in Chapter 6.*

4.2 EXPERIMENTAL EVALUATION

4.2.1 *Setup*

A set of 20 ligand-receptor inputs used during tests were obtained from the Protein Data Bank (PDB) [17]. These were preprocessed before docking following the standard protocol using AutoDockTools [76]. 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 MD configuration, relevant parameters were set according to the GA and Solis-Wets LS values specified in Table 4.1. From a performance point of view, the most relevant parameters are the maximum number of score evaluations ($N^{\text{MAX}}_{\text{score-evals}}$) and the maximum number of generations ($N^{\text{MAX}}_{\text{gens}}$), because these control how long the MD program runs. The choice of $N^{\text{MAX}}_{\text{score-evals}} = 2\,500\,000$ has followed the guidelines provided in [73]. These guidelines suggest – for ligands with up to ten rotatable bonds (i. e., $N_{\text{rot}} \leq 10$), as in this experiment – to run the program until either $N^{\text{MAX}}_{\text{score-evals}}$ reaches a value between [250 000, 25 000 000] or $N^{\text{MAX}}_{\text{gens}} = 27\,000$, whichever comes first.

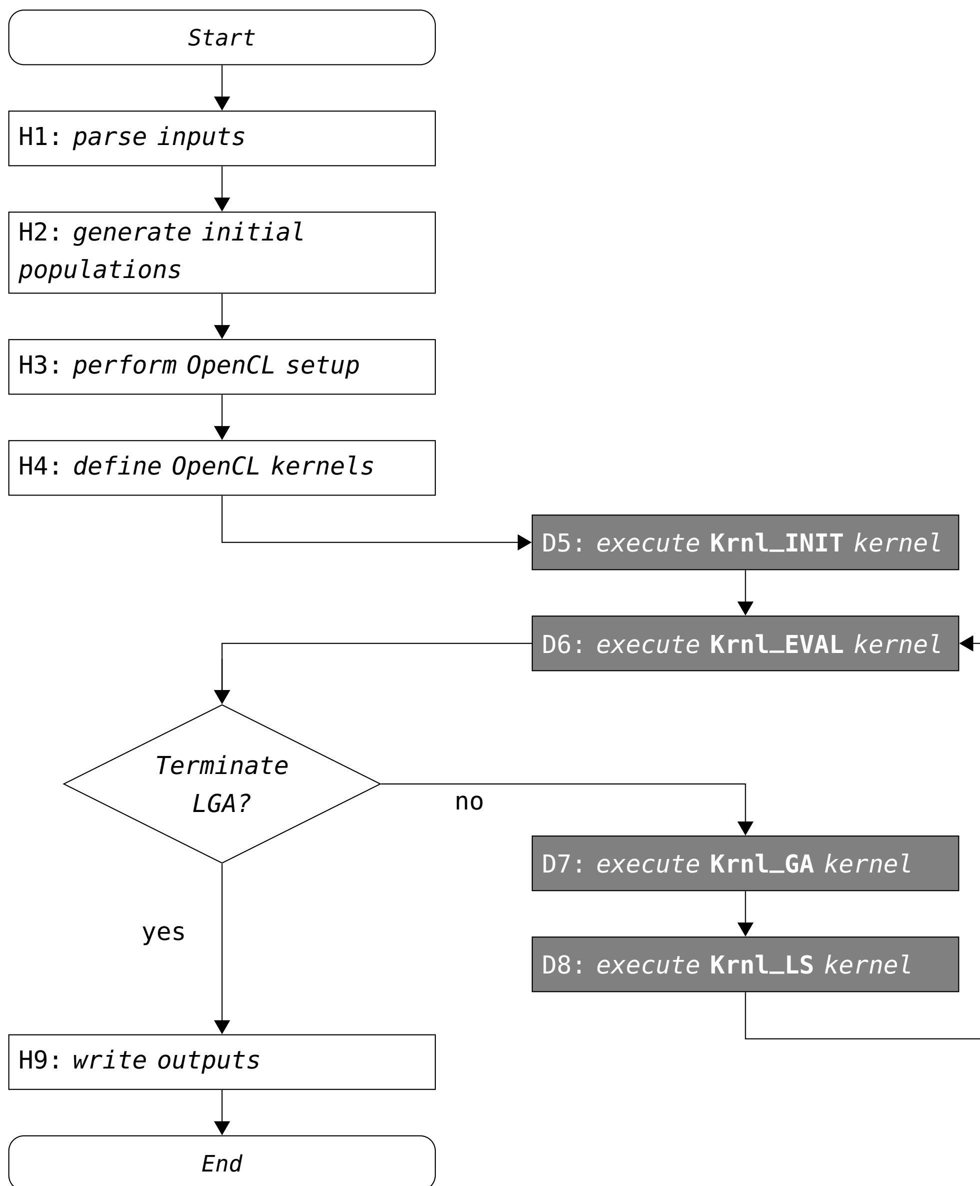


Figure 4.3: The overall OCLADock workflow consists of a sequence of host (Hx) and device (Dx) functions. Program execution always starts and finishes in host functions (depicted at the left side). OpenCL kernels are executed iteratively on the device (depicted at the right side), while their termination is controlled by the host.

Table 4.1: OCLADock configuration for experiments.

	Parameter description	Notation	Value	
GA	Population size	P	150	
	Maximum # score evaluations	$N^{\text{MAX}}_{\text{score-evals}}$	2 500 000	
	Maximum # generations	$N^{\text{MAX}}_{\text{gens}}$	27 000	
LS	Local-search rate	`lsrate`	6	%
	Maximum # local-search iterations	$N^{\text{MAX}}_{\text{LS-iters}}$	300	
	Lower bound of initial variance	step^{MIN}	0.01	

The Solis-Wets variance, referred to as `step` in Section 2.2.3, specifies the size of the solution space to sample, i. e., the amount by which orientation and torsion angles change on every Solis-Wets cycle. The variance value can be initially specified by a user, typically $\text{step}^{\text{INITIAL}} = 1$. However, it changes during local optimization depending on the search success.

The target system used in this evaluation provides two processing elements:

- An Intel i5-6600K CPU clocked at 3.5 GHz.
- An AMD R9-290X GPU with 2 816 multiprocessors and 44 active compute units.

The CPU was used to collect the baseline characteristics of the original single-threaded implementation, as well as a target to execute OCLADock on multiple CPU cores.

4.2.2 *Validation*

These experiments aim to verify the correct operation of OCLADock accounting for the three key metrics already introduced in Section 2.2.6: lowest binding score (LBS), root mean square deviation (RMSD), and size of best cluster (SBC). Table 4.2 shows these metrics obtained for 100 LGA runs considering six compounds of different sizes in terms of number of atoms (N_{atom}) and rotatable bonds (N_{rot}): `3ptb` is the smallest ($N_{\text{atom}} = 13$, $N_{\text{rot}} = 2$), while `3c1x` is the largest one ($N_{\text{atom}} = 46$, $N_{\text{rot}} = 8$).

Details on key implementation differences compared to AutoDock *are provided in Appendix A.*

The most noticeable differences between the serial and OpenCL implementations are found in their corresponding RMSDs (`4hmg`, `3c1x`) and SBCs (`3ptb`, `3c1x`, `3ce3`). As reported in our previous publication [233], such discrepancies were initially (May 2017) attributed to the different selection schemes employed during GA: *proportional selection* (AutoDock) and *binary tournament* (OCLADock). The choice of using *binary tournament* instead of *proportional selection* is strongly mo-

Table 4.2: Functional validation of OCLADock vs. single-threaded AutoDock, both running the Solis-Wets local-search method. All results were obtained using 100 LGA runs, and RMSD tolerance = 2 Å. Best values within each criterion are colored.

Ligand-receptor input			Lowest binding score (LBS) (kcal/mol)		RMSD (Å) of LBS		Size of best cluster (SBC)	
ID	N_{rot}	N_{atom}	Serial baseline	OpenCL GPU	Serial baseline	OpenCL GPU	Serial baseline	OpenCL GPU
3ptb	2	13	−5.55	−5.55	0.42	0.41	100	72
1stp	5	18	−8.37	−8.32	0.42	0.38	100	100
3bgs	5	24	−6.68	−6.59	0.75	0.78	95	90
4hmg	10	27	−3.68	−3.95	0.97	0.81	34	51
3ce3	5	37	−11.59	−11.08	0.93	0.77	94	71
3c1x	8	46	−13.61	−13.29	0.80	1.18	90	63

tivated by the shorter execution times for tournament-based selection schemes.

Further code analysis performed in collaboration with TSRI (March 2018) helped us discovering that, the OCLADock version used up to this point in time has been implementing the scoring function (Section 2.2.4) without *smoothing* the *van der Waals* and hydrogen bonding potentials according to certain threshold interatomic distances [35]. This smoothing feature is enabled by default in AutoDock, and using it can *favorably* affect RMSDs and SBCs, i. e., producing poses with smaller RMSDs and larger clusters. LBS of best resulting poses are *virtually not* affected since poses with *weak* (unfavorable) scores are discarded during the genetic evolution.

The inclusion of the smoothing feature on OCLADock was completed later (September 2018), and the quality of the corresponding *correct* results is extensively discussed in Chapter 6. Nevertheless, we observed that execution runtimes of OCLADock were not affected by adding the smoothing. Therefore, its execution speedups and energy gains reported as follows in this chapter – as well as in [233] – are still meaningful.

4.2.3 *Execution performance*

The performance results are grouped into CPU and GPU categories. For each of them, work-group sizes of {16, 32, 64} work-items were tested. The first experiment aimed to determine the impact of the work-group size on the execution time for each computing platform. The tendency in most of cases, as depicted in Figure 4.4, is that better results are achieved with 16 and 64 work-items for CPU and GPU, respectively.

All execution times reported in this thesis correspond to full program executions for both serial and parallel versions.

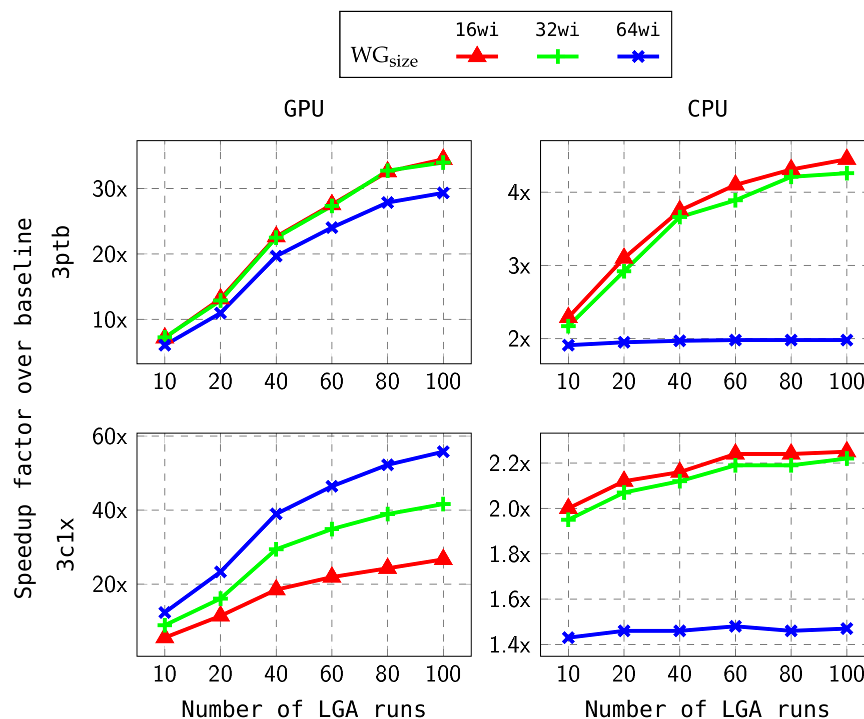


Figure 4.4: Speedups of OCLADock vs. single-threaded AutoDock achieved on GPU/CPU devices for different work-group sizes. Vertical scales are different.

The case of 3ptb on the GPU is an exception, where a configuration of {16, 32} work-items compared to that of 64 work-items led to higher speedups. This can be explained by the limited degree of parallelism provided by this very small molecule.

The speedup behavior for different compounds was further analyzed by using work-group sizes of 16 and 64 work-items for CPU and GPU, respectively. Figure 4.5 shows that the achieved speedup varies between compounds. According to Algorithm 3, the algorithmic complexity is also dependent on the molecule size because:

1. Additional operations are performed for calculating the ligand pose for compounds with more rotatable bonds.

2. Larger grid maps are read for compounds having more receptor atoms.

3. More intramolecular interactions are computed for compounds having more ligand atoms.

To illustrate this, consider that small compounds (3ptb, 1stp) achieved higher speedups than bigger ones (3ce3, 3c1x) on the CPU. On the other hand, bigger compounds are executed faster on the GPU, since they provide more data parallelism than can be leveraged by the larger number of compute units: 44 on the R9-290X GPU vs. 4 cores on the

Table 4.3: Execution time (s) and speedups for 100 LGA runs on CPU (16 work-items) and GPU (64 work-items).

Input ID	Execution time (s) Serial baseline	OpenCL CPU	OpenCL GPU	Speedup OpenCL CPU	Speedup OpenCL GPU
`3ptb`	586.3	131.8	20.0	4.5x	29.3x
`1stp`	836.5	241.1	27.1	3.2x	30.9x
`3bgs`	1102.9	312.2	28.3	3.5x	39.0x
`4hmg`	1416.2	403.1	32.9	3.6x	43.1x
`3ce3`	1867.7	617.0	36.2	3.0x	51.7x
`3c1x`	2841.8	1265.7	51.0	2.3x	55.8x

CPU. On the complete set of 20 compounds, the geometric mean of the speedup is ~3.3x and ~40.4x for the CPU and GPU, respectively.

In order to evaluate the optimality of the achieved speedups, the utilization of computing resources was investigated by profiling the execution of 100 LGA 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), cache hit-rates of ~84 % for both `Krnl_GA` and `Krnl_LS` kernels were observed, as well as ~57 % and ~20 % of GPU time that the memory unit was active per each `Krnl_GA` and `Krnl_LS` execution, respectively.

The higher memory-access rate that characterizes the `Krnl_GA` is due to the required creation of new individuals for the next generation. Specifically, this kernel must access data of the entire current population stored in the external memory at the beginning and end of the GA, as well as of grid maps and intramolecular weights during the score calculation of all individual members of the population.

On the other hand, `Krnl_LS` itself consists in an iterative process consuming ~95 % of total GPU execution time. However, its memory-access rate is much lower than that of `Krnl_GA` since it does not need to retrieve *all* individuals, 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 4.3, and consistently show higher speedups achieved by the GPU.

4.2.4 *Compute-energy efficiency*

The compute energy required by the different OCLADock variants (CPU and GPU ones) was obtained by sampling the drawn power

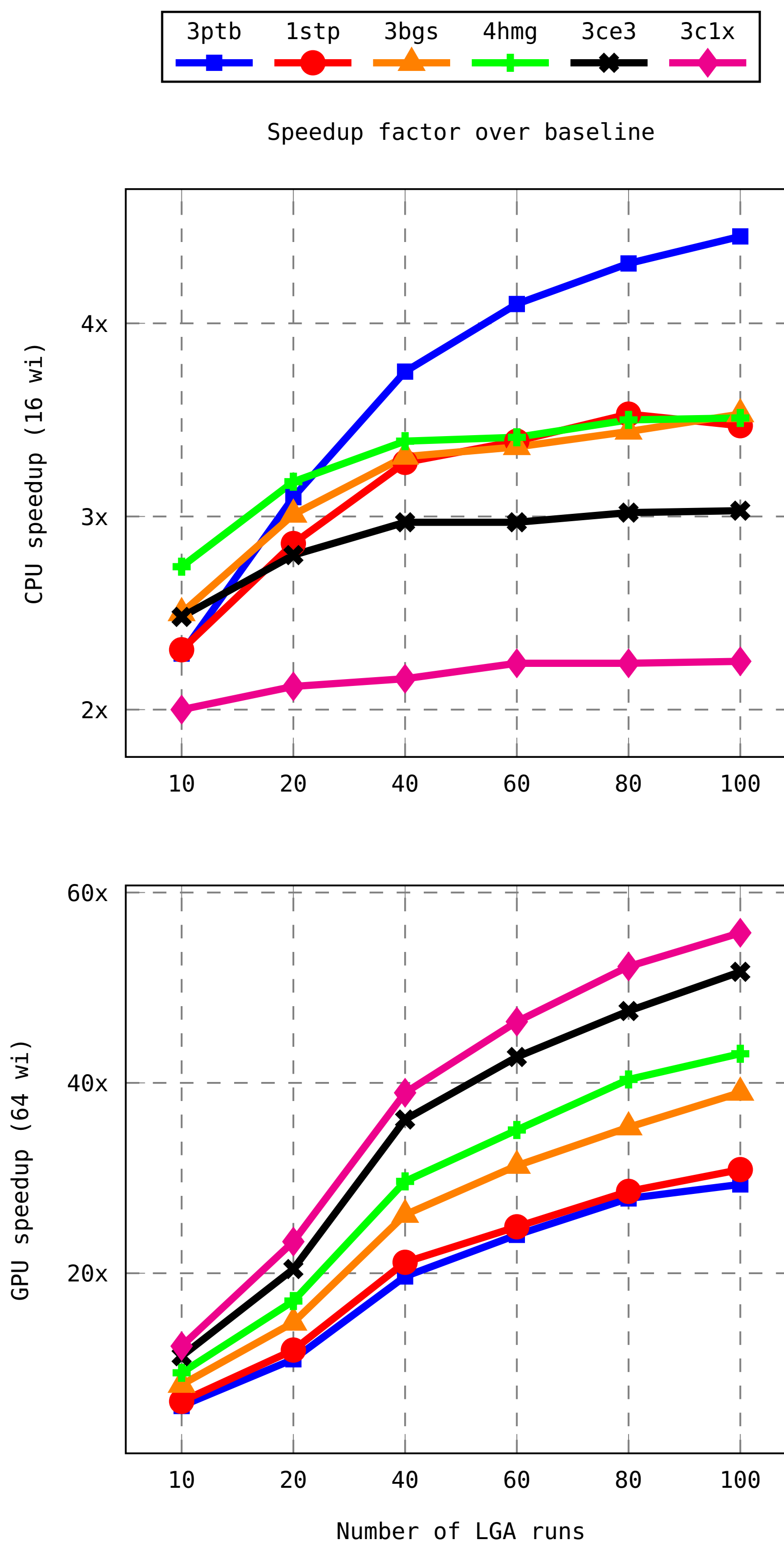


Figure 4.5: Speedups of OCLADock vs. single-threaded AutoDock achieved on CPU (16 work-items) and GPU (64 work-items). Vertical scales are different.

Table 4.4: Measured power values (approximated) on the CPU.

Execution type	Power (W)
Idle	4
Baseline (single CPU core)	19
OpenCL (four CPU cores)	48

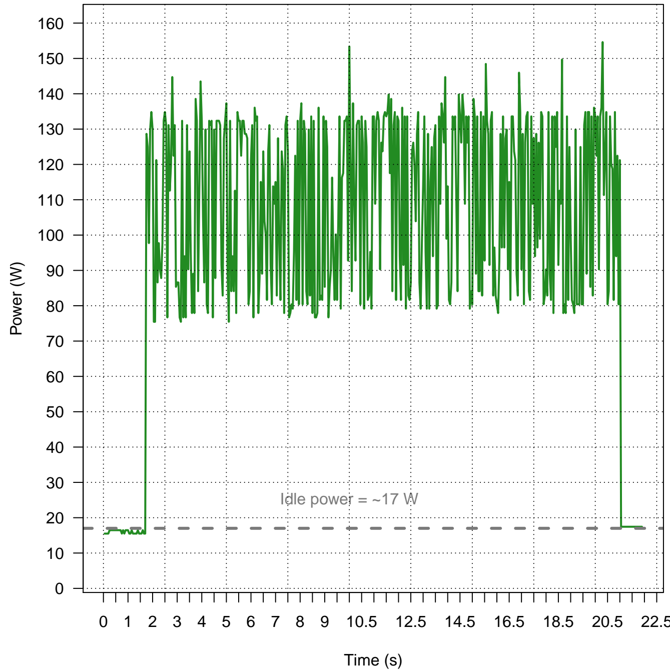


Figure 4.6: Power measurements on the RX-290X GPU for 10 LGA runs using `3c1x`.

in $T_{\text{sampling}} = 50$ ms intervals, using power performance counters on both CPU and GPU to avoid the inaccuracies typically associated with external measurements (e. g., shunt-based). The power samples are then integrated over time to derive the energy.

Using this methodology, it was discovered 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 a docking job (Table 4.4). On the GPU, however, the power draws varied between [∼75 W, ∼155 W] over the execution time (Figure 4.6). This different behavior is attributed to the algorithm switching between the `Krnl_GA` and `Krnl_LS` kernels, with the latter having a much lower degree of parallelism that the first one. On the GPU, this leads 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.

Table 4.5: Energy consumption (kJ) results and energy-efficiency gains for 100 LGA runs on CPU (16 work-items) and GPU (64 work-items).

Input ID	Energy consumption (kJ) Serial baseline	Energy consumption (kJ) OpenCL CPU	Energy consumption (kJ) OpenCL GPU	Efficiency gain OpenCL CPU	Efficiency gain OpenCL GPU
`3ptb`	11.8	5.9	2.4	1.9x	4.9x
`1stp`	16.7	11.7	3.7	1.4x	4.5x
`3bgs`	21.6	15.2	4.2	1.4x	5.2x
`4hmg`	28.1	19.4	4.8	1.4x	5.8x
`3ce3`	36.3	30.4	5.8	1.2x	6.2x
`3c1x`	54.9	61.2	8.7	0.9x	6.3x

The energy consumption results are grouped into CPU and GPU categories. Table 4.5 shows energy consumption for serial and parallel execution in the case of selected compounds. Despite that the GPU required a higher amount of *power* than the CPU during certain time periods, the GPU achieved greater *energy* savings than the most parallel version on the CPU.

Figure 4.7 shows the gains in energy efficiency for different LGA runs. In both CPU and GPU cases, the efficiency gain (Figure 4.7) behaves similarly as the speedup presented previously (Figure 4.5), depending as well on the molecular input 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: 2841.8 s, OpenCL CPU: 1265.7 s) and their power measurements (baseline: ∼19 W, OpenCL CPU: ∼48 W) for 100 LGA runs (Table 4.3, Table 4.4). 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 efficiency for this experiment. Considering the 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.

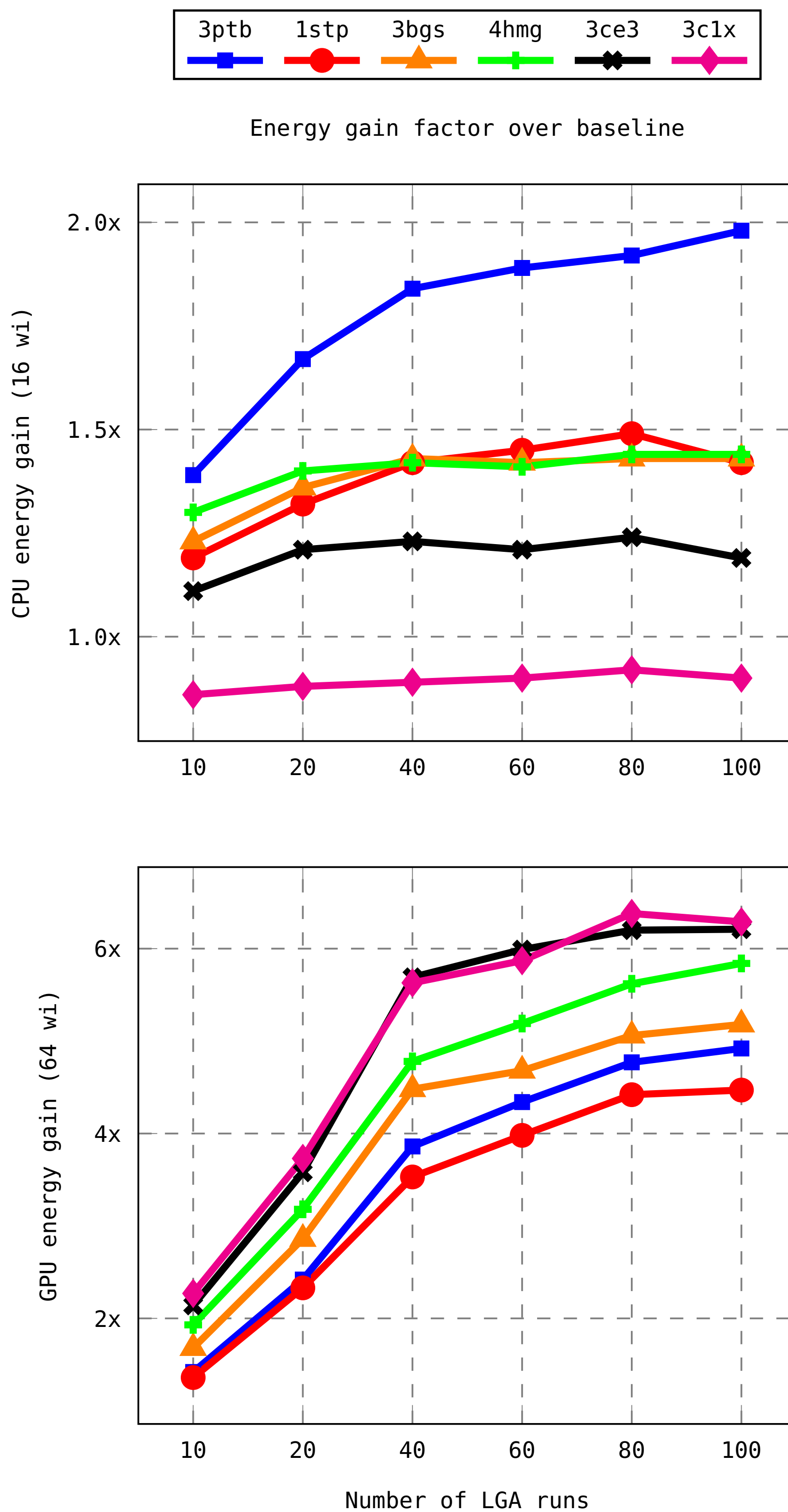


Figure 4.7: Energy-efficiency gains of OCLADock vs. single-threaded AutoDock achieved on CPU (16 work-items) and GPU (64 work-items). Vertical scales are different.

5

OCLADOCK-FPGA: PORTING AUTODOCK TO FPGAS USING OPENCL

This chapter details OCLADock-FPGA, an OpenCL-based *task* parallelization of AutoDock, and provides an evaluation of its performance and compute-energy efficiency. Contents of this chapter have been previously published in [234].

5.1 DATA-PARALLEL APPROACH ON FPGAS

In Chapter 4, a data-parallel design of AutoDock was evaluated on CPUs and GPUs. The initial intention was to complete the evaluation by also including FPGAs as a compute platform for OpenCL-based acceleration. However, SDAccel v2015.4 [210] – the OpenCL-to-FPGA compiler employed – had significant robustness problems. Code that ran perfectly on the CPU or the GPU often resulted in:

- The SDAccel tool crashing during compilation.
- Truly excessive tool runtimes (longer than a week), and memory requirements (more than 128 GB) during hardware generation.
- Crashes and execution errors on two different FPGA platforms: Alpha Data 7v3 card [37] (Virtex-7), and Micron AC-505 module [113] (Kintex-7).

After much rewriting of the OpenCL code to overcome tool bugs, along with an update to SDAccel v2016.1, a data-parallel version that actually executed correctly on the Alpha Data 7v3 card was finally achieved. However, it suffered from a severe slowdown over the sequential baseline (i. e., the original AutoDock running on a single i5-6600K CPU core) in the range of three orders of magnitude. These difficulties were attributed to the unsuitability of such data-parallel design for FPGAs, as well as the somewhat unstable nature of that compiler version.

As a more promising alternative,we investigated the use of a task-parallel implementation scheme. The task-parallel approach is commonly believed to be more suitable to the underlying FPGA hardware. Also, we switched from Xilinx to Intel (Altera) FPGAs for further work, as the Intel OpenCL-to-FPGA compiler is more mature than its Xilinx

counterpart. Furthermore, a number of FPGA-specific optimization techniques were exploited. The patterns required to optimize the FPGA design made us consider several device-specific architectural and micro-architectural choices, expressed using the high-level abstractions of OpenCL. This evolved towards a parallel AutoDock implementation with improved runtime and energy-efficiency, achieving actual speedups with respect to the serial baseline. The techniques presented here may also be beneficial when accelerating other applications.

5.2 TASK-PARALLEL APPROACH: REFORMULATED STRATEGY FOR FPGAS

5.2.1 *Reference pipeline design for FPGAs*

As already discussed in Chapter 3, in 2010, Pechan, Fehér, and Bérces [146] used Verilog to implement an architecture that consists of a three-stage pipeline composed of four modules, each – in turn – consisting of parallel and fine-grained pipelines (Figure 5.1). Specifically, `Step 1` controls the generation of individuals by the rules of either the genetic algorithm (GA) or local search (LS). `Step 2` calculates the ligand pose. The third stage is composed of `Step 3` and `Step 4` that calculate the intramolecular and intermolecular interactions, respectively. Performance gains of ∼23x using a Virtex-4 FPGA compared to a 3.2 GHz Xeon CPU core were reported.

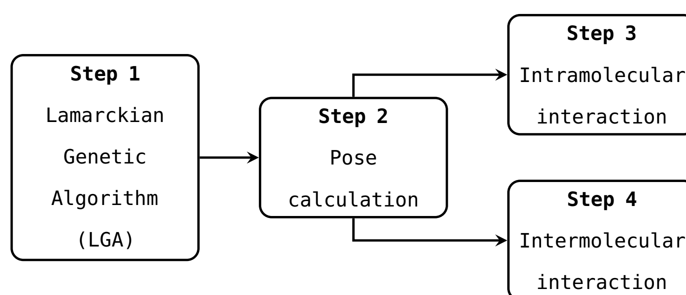


Figure 5.1: Pipeline processing of the LGA of AutoDock proposed in [146].

5.2.2 *The development phases*

The task-parallel source code is available in the OCLADock-FPGA *repository at [186].*

Since pipeline processing is well exploitable on FPGAs, the design proposed by Pechan, Fehér, and Bérces [146] was adopted as the starting architecture (Figure 5.1) for our OpenCL implementation. This architecture executes the entire docking job sequentially, i. e., by starting a new LGA run only after the previous run has finished, while pipelining the GA calculations *within* each run. From a programming perspective, such an architecture is realizable following a task-parallel approach, in which each task is coded as a single work-item kernel. The actual

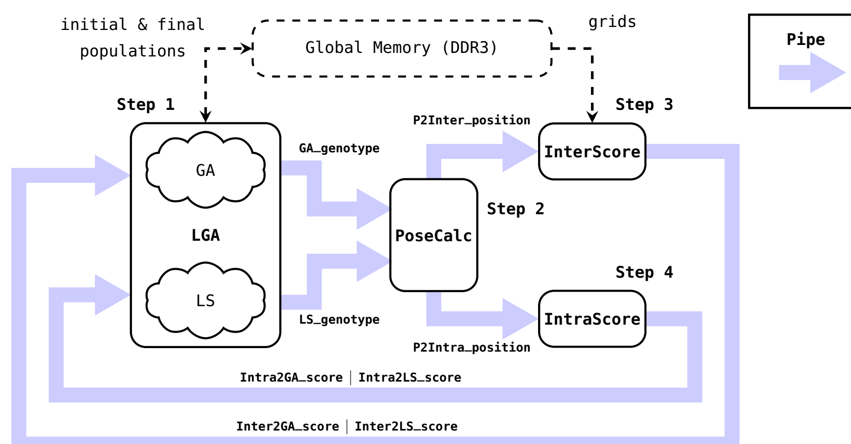


Figure 5.2: First development phase: initial OpenCL design.

OpenCL implementation took place in the following four development phases. For each, we describe the design and optimization steps applied incrementally over the previous ones.

5.2.2.1 *First development phase*

Figure 5.2 represents the initial OpenCL design. `Step 1` of the genetic-algorithm (`GA`) and local-search (`LS`) functions (Figure 2.3) have been merged into a single `LGA` kernel. This is because these two kernels execute in sequence (i. e., `GA` followed by `LS`), and exchange data of individuals and their scores on every genetic cycle. Merging them into a single `LGA` kernel avoids exchanging data through off-chip memory, and allows it through on-chip memory instead.

The code in OCLADock-FPGA *repository [186] uses the term* energy *to quantify the binding interaction. Here, we use* score *instead to avoid confusions with the* compute energy.

The `LGA` kernel controls the overall functionality of the system, which is composed also of the `PoseCalc`, `InterScore`, and `IntraScore` kernels that in turn correspond to `Step 2`, `Step 3`, `Step 4` (Figure 5.2), respectively. The communication between all kernels is achieved through OpenCL pipes that serve as FIFO-based mechanisms passing data such as:

OpenCL pipes are often referred to as channels, *according to the Intel-specific terminology.*

- Genotypes:
  - From `LGA` to `PoseCalc`, through `GA_genotype` and `LS_genotype`.
- Ligand poses:
  - From `PoseCalc` to `InterScore`, through `P2Inter_position`.
  - From `PoseCalc` to `IntraScore`, through `P2Intra_position`.
- Computed scores (or MD binding energies):
  - From `InterScore` to `LGA`, through `Inter2GA_score` and `Inter2LS_score`.

– From `IntraScore` to `LGA`, through `Intra2GA_score` and `Intra2LS_score`.

This design forms a closed loop consisting of kernels and channels that prevents the compiler from optimizing any channel depth [83]. As an attempt to *avoid* this closed-loop scenario, and in turn, to help the compiler perform more aggressive optimizations, the following experiment was attempted.

Instead of sending the feedback scores to the `LGA` kernel through channels, these scores were initially passed through global memory. The idea was to instantiate an (additional but not shown in Figure 5.2) `Store` kernel that receives such scores from `InterScore` and `IntraScore`, and then writes them onto off-chip memory. Thereafter, these scores could be read by `LGA`.

In the OpenCL memory model, the global *region is the abstraction of the physically off-chip memory (Figure 5.2). Here, we also refer to it as* external.

For this idea to work, OpenCL fences (`mem_fence`) on global memory accesses were used in `Store` and `LGA` (Algorithm 4). Although this worked correctly in emulation, it did *not* work on the FPGA. This was due to races in off-chip memory accesses, which happen when a *certain* off-chip location is accessed simultaneously by different kernels, with at least one kernel performing a write. According to the Intel OpenCL-to-FPGA tool documentation [83], the consistency of global memory accesses is ensured *only* within a *single* kernel, and such races cannot be prevented by using fences within *separate* kernels. Therefore, in order to achieve a correct functionality on the actual FPGA, the idea of using global memory for avoiding the closed-loop of kernels was *discarded*. In other words, the `Store` kernel was removed, while the feedback channels were included back in the initially-proposed design.

Algorithm 4: Attempt to synchronize accesses to a given location in external memory using fences. Correct data transactions between kernels (e. g., `Rx_Val` receiving the value initially stored in `Tx_Val`) occur only in emulation. For that reason, this alternative design was discarded.

```
/* Producer kernel */
void kernel Store (global const <type> *ext, ...)
    ...
    ext [Addr] = Tx_Val;
    mem_fence (CLK_GLOBAL_MEM_FENCE);
    ...

/* Consumer kernel */
void kernel LGA (global const <type> *ext, ...)
    ...
    mem_fence (CLK_GLOBAL_MEM_FENCE);
    Rx_Val = ext [Addr];
    ...
```

Another consideration regarding global memory was to minimize the number of accesses. Particularly, the `GA` logic updates population

data throughout an entire LGA run, so storing populations only in off-chip memory would result in significant lower performance. This was addressed by reading from and writing to global memory only at the start and end of each LGA run, while keeping intermediate populations on-chip, using `__local` OpenCL two-dimensional arrays. These arrays adopt a `2D-data` $[P^{\text{MAX}}][N^{\text{MAX}}_{\text{genes}}]$ form, which holds a maximum population size (P) of 150, and genotype size (N_{genes}) of 38.

From Chapter 2: $N_{genes} = N_{rot} + 6$. For AutoDock: $N^{MAX}_{rot} = 32$.

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 large map representing an *entire* receptor molecule would be required. Second, due to the irregular reads performed by `InterScore`, any caching strategy consisting in switching the scope of *partial* grid data from OpenCL `__global` into OpenCL `__constant` memory space, resulted in significant misses.

See Table C.2 for grid sizes.

A further discussion on caching other data is provided in Section 5.2.3.

5.2.2.2 *Second development phase*

Since `PoseCalc`, `InterScore`, and `IntraScore` turned out to be the major bottleneck, their microarchitecture was optimized 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* (Algorithm 5).

These kernels always execute together as a chain of blocks, being invoked a number of times controlled by `LGA`, i. e., restricted by the maximum number of either score 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, the goal was to minimize the initiation-interval (II, ideally II = 1) of each loop.

Different code-refactoring techniques such as *shift registers*, *local-memory banking*, *unrolling of inner loops* resulted in significant reduction of data dependencies. In the case of `PoseCalc`, it was not possible to remove the data dependency created for keeping 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 `InterScore` and `IntraScore` kernels involve long latencies such as random accesses to off-chip grids, and single-precision floating-point calculations required for Equation 2.3, respectively. Despite these, all their loops (i. e., outer while-loops and inner computation-loops) were fully pipelined.

These code-refactoring techniques are detailed in the Intel OpenCL-to-FPGA tool documentation [83].

The next optimizations performed on the `LGA` kernel aimed to potentially increase the execution concurrency in subsequent design phases (i. e., from Section 5.2.2.3 on), leading to the architecture depicted in Figure 5.3. First, the local-search logic was moved out of `LGA` and implemented as a separate `LS` kernel, where each `LS`-execution is

Algorithm 5: Code structure used in the `InterScore` kernel implementation. Similar structures are used in `PoseCalc` and `IntraScore`. The outermost while-loop is controlled by the `active` signal. The main-computation loop lists only simplified operations.

```
/* Kernel calculating intermolecular interactions */
void kernel InterScore (global const <type> *grids, ... )
    // Active signal
    char active = 0x01;

    // Other declarations go here
    // But are not listed for simplicity

    // Setting active = 0x00 terminates kernel
    while active do
        /* Reading from input channels */
        active = read_channel_intel (P2Inter_active);
        mem_fence (CLK_CHANNEL_MEM_FENCE);

        // Not merged with following loop
        // Doing so allows trying wider data-types
        // for data transferred through channels
        for each lig-atom in N_atom do
            coordinates [lig-atom] = read_channel_intel (P2Inter_position);

        /* Main-computation loop */
        for each lig-atom in N_atom do
            xyz = coordinates (lig-atom);

            // Reading and processing grid maps:
            // van der Waals, hydrogen bonding,
            // electrostatics, and desolvation
            read_vdw_hb = grids [xyz + offset_vdw_hb];
            read_el = grids [xyz + offset_el];
            read_ds = grids [xyz + offset_ds];

            partial_vdw_hb = interpolation (read_vdw_hb);
            partial_el = interpolation (read_el);
            partial_ds = interpolation (read_ds);

            // Accumulating intermolecular interaction
            interS += partial_vdw_hb + partial_el + partial_ds;

        /* Writing to output channels */
        if Running GA then
            write_channel_intel (Inter2GA_score, interS);
        else
            // Running LS
            write_channel_intel (Inter2LS_score, interS);
```

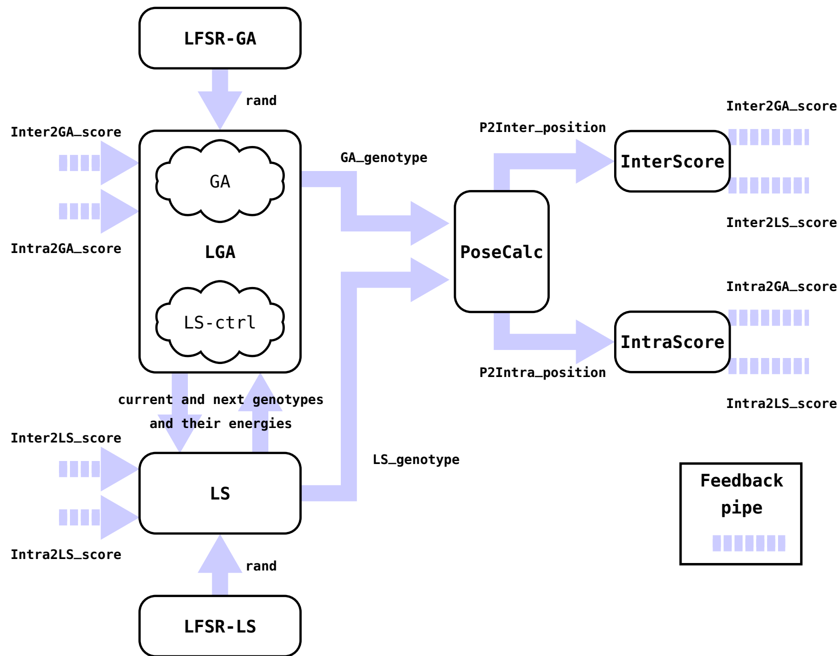


Figure 5.3: Second development phase: local-search logic is implemented as a separate kernel. From now on, feedback channels are shown as dashed connections, while global-memory accesses are omitted for simplicity.

triggered from within the LS-ctrl control loop. The LS kernel reads genotypes from LGA, performs the score minimization, and returns to LGA new genotypes and their respective scores. Similarly to LGA, LS sends genotypes via the LS_genotype channel to be evaluated by the {PoseCalc, InterScore, IntraScore} chain.

Furthermore, the pseudo-random number generators – initially implemented as linear congruential generators (LCG [46]) – invoked within the genetic-algorithm and local-search logic, were converted into and replaced with separate LFSR-GA and LFSR-LS kernels, each featuring a 32-bit linear feedback shift register (LFSR [5]). Corresponding implementations in Algorithm 6 show that, while both alternative implementations return a float-type number to the LGA kernel, the LFSR-GA implementation does it through a channel.

5.2.2.3 *Third development phase*

Due to the iterative nature of the local search, the initial focus was on its microarchitectural optimization. Although code refactoring guided by compiler suggestions [83] helped to pipeline the majority of its inner LS-loops, pipelining its outermost loop was not possible due to dependencies created by genotype-data carried through inner loops, and channel invocations for score calculation. In order to compensate for this, the LS kernel was *replicated*, together with its LFSR-LS and channel interconnects, three times (Figure 5.4).

Algorithm 6: In the second development phase, the random number generator RNG function invoked within LGA was replaced with a LFSR-GA kernel.

```
/* LCG implemented as a function */
float Function RNG (uint* rng)
    *rng = CONST_1 * (*rng) + CONST_2;
    return convert_float (*rng / MAX_UINT) * 0.999999f;

/* LFSR implemented as a separate kernel */
void kernel LFSR-GA (uint* seed, uchar N_genes)
    uint lfsr = seed;
    bool stop = false;

    while !stop do
        bool active = true;
        active = read_channel_nb_intel (LGA2RNG_prng, &stop);

        for each gene in N_genes do
            float rand;
            uchar lsb;
            lsb = lfsr & 0x01u;
            lfsr >>= 1;
            lfsr ^= (-lsb) & 0xA3000000u;
            rand = (0.999999f / MAX_UINT) * lfsr;

            bool success = false;
            if !stop then
                success = write_channel_nb_intel (RNG2LGA_prng, rand);
```

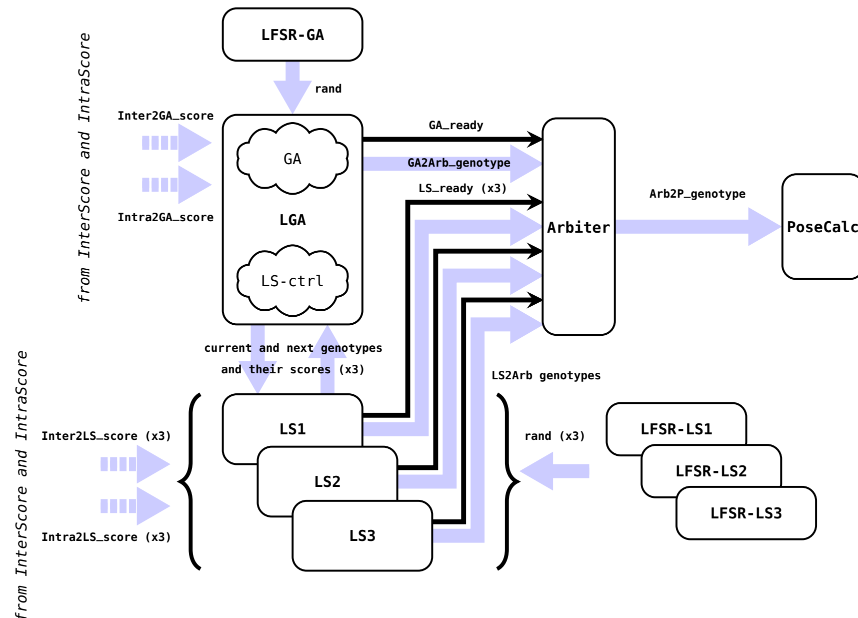


Figure 5.4: Third development phase: local-search kernels are *replicated* three times, while an arbiter kernel is added to handle simultaneous score-calculation requests. Score calculation kernels are omitted for simplicity.

This architectural change has a particular consequence described as follows. As already described in Section 5.2.2.2, the initially single LS kernel invokes the execution of the {PoseCalc, InterScore, IntraScore} block chain very much like LGA during the genetic algorithm. In this scenario, LGA and LS act as producers, whereas PoseCalc acts as a consumer of genotypes. Since the genetic-algorithm and local-search functions are mutually exclusive during LGA execution, the arbitration in PoseCalc 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 (Figure 5.4), multiple score 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 corresponding genotypes from each producer (Algorithm 7). The ready signals identify the actual producers whereas the genotypes corresponding to valid ready signals are accumulated using OpenCL __local arrays and dispatched in order towards PoseCalc.

Furthermore, each replicated LS kernel in Figure 5.4 has its own channels replicated as well. By using the corresponding channel (e. g., channel <k> for LS <k>, with k = {1, 2, 3}), calculated scores (in InterScore and IntraScore) are ensured to return back to the correct LS kernel.

5.2.2.4 *Fourth development phase*

The LS kernel was replicated more often, as long as the resulting circuit fit on the target FPGA (Figure 5.5). The replication factor was based on the upper bound of the LS-control loop, whose default value is determined by the number of individuals that undergo local-search during a single LGA evolution, i. e., nine individuals that represent a random subset (lsrate = 6 %) of the population size (P = 150, as set in Section 5.2.2.1). As more LS instances imply fewer loop-rounds, the LS replication factor was increased from three (Section 5.2.2.3), up to five and nine in this phase.

Subsequently, Arbiter was optimized similarly as in [218], 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 PoseCalc that selects incoming genotypes directly from a specific producer (Algorithm 8), instead of being accumulated and reordered through Arbiter as performed in Section 5.2.2.3.

Algorithm 7: In the third development phase, `Arbiter` kernel reads *ready* signals and genotypes from producer kernels: `GA` and three `LS`. Accumulation and dispatch to `PoseCalc` of received genotype data is omitted for simplicity.

```
void kernel Arbiter (uint N_genes)
    bool active = true;
    __local float GA_genotype [LENGTH];
    __local float LS<j>_genotype [LENGTH]; // Definitions for j = 1, 2, 3

    while active do
        bool Off_valid, GA_valid = false;
        bool LS<j>_valid = false; // Definitions for j = 1, 2, 3

        bool Off_active, GA_active;
        bool LS<j>_active; // Definitions for j = 1, 2, 3

        // Keep polling ready signals if no genotype was received
        while
        (Off_valid == false) &&
        (GA_valid == false) &&
        (LS1_valid == false) &&
        (LS2_valid == false) &&
        (LS3_valid == false) do
            Off_active = read_channel_nb_intel (Off, &Off_valid);
            GA_active = read_channel_nb_intel (GA_ready, &GA_valid);
            // Statements for j = 1, 2, 3
            LS<j>_active = read_channel_nb_intel (LS<j>_ready, &LS<j>_valid);

        // Initializing counter of received genes
        uchar bound_tmp = 0;

        // Checking if Arbiter kernel should be turned off
        active = Off_valid ? Off_active : true;

        if active == true then
            for each gene "i" in N_genes do
                if GA_valid == true then
                    GA_genotype [i] =
                      read_channel_intel (GA2Arb_genotype);

                // Statements for j = 1, 2, 3
                if LS<j>_valid == true then
                    LS<j>_genotype [i] =
                      read_channel_intel (LS<j>2Arb_genotype);

            if GA_valid == true then
                bound_tmp++;

            // Statements for j = 1, 2, 3
            if LS<j>_valid == true then
                bound_tmp++;

        // Accumulating genotypes ready to be dispatched
        ...

        // Dispatching genotypes to PoseCalc
        ...
```

Algorithm 8: In the fourth development phase, genotypes generated in either `GA` or any of the nine `LS` kernel are sent directly to `PoseCalc`, instead of being accumulated in `Arbiter`, as in the third development phase.

```
void kernel PoseCalc (...)
    char active = 0x01;

    while active do
        /* Reading from input channels */
        char actmode = read_channel_intel (Arbiter2P_actmode);
        mem_fence (CLK_CHANNEL_MEM_FENCE);

        // Updating active and mode signals
        active = actmode;
        char mode = actmode;

        // Multiplexing from input channels with incoming genotypes
        for each gene in N_genes do
            // Variable carrying incoming genotype
            // from the kernel indicated by mode
            float tmp;

            switch mode do
                case 'G': tmp = read_channel_intel (GA_genotype); break;
                case 1: tmp = read_channel_intel (LS1_genotype); break;
                case 2: tmp = read_channel_intel (LS2_genotype); break;
                case 3: tmp = read_channel_intel (LS3_genotype); break;
                case 4: tmp = read_channel_intel (LS4_genotype); break;
                case 5: tmp = read_channel_intel (LS5_genotype); break;
                case 6: tmp = read_channel_intel (LS6_genotype); break;
                case 7: tmp = read_channel_intel (LS7_genotype); break;
                case 8: tmp = read_channel_intel (LS8_genotype); break;
                case 9: tmp = read_channel_intel (LS9_genotype); break;

        /* Main-computation loop */
        for each rot-item in N_pose-rot do
            // Rotating based on orientation and torsional genes.
            // Calculating atomic coordinates upon rotation
            ...

        /* Writing to output channels */
        // Sending atomic coordinates to InterScore and IntraScore
        ...
```

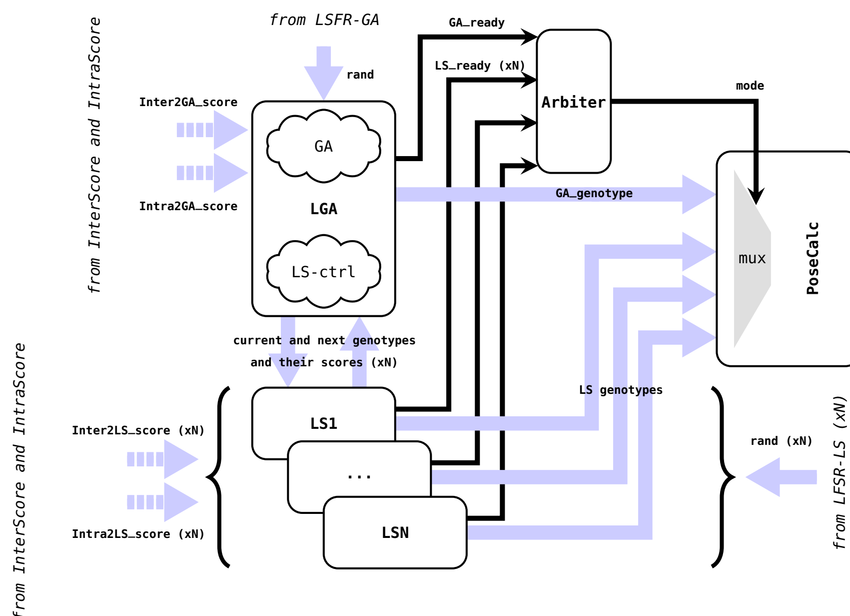


Figure 5.5: Fourth development phase: local-search kernels are further replicated, while the arbitration mechanism is improved.

5.2.3 *Further optimization techniques*

In addition to the FPGA-specific optimizations techniques [83] so far employed, the following three were considered in greater detail:

1. All kernel constants were pre-calculated in the host and passed into kernels afterwards, e. g., scaled crossover, mutation and selection rates (`LGA`), reference orientations (`PoseCalc`), offsets for indexing grid maps depending on constant grid sizes (`InterScore`), as well as scoring function weights (`IntraScore`). This was implemented in the second phase and maintained during subsequent development phases.

Caching attempts of grid data were already discussed in Section 5.2.2.1.

2. 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 a kernel cannot fit `__constant` arguments in the cache, related accesses suffer from larger performance penalties than those of `__global const` due to misses. This is because off-chip accesses are implemented with extra circuitry for tolerating longer latencies. The look-up tables used in `PoseCalc`, `InterScore`, and `IntraScore` occupy a total of 12 kB in on-chip constant memory. On the other hand, larger data such as rotation list, grid maps, and the list of intramolecular contributors (corresponding to `PoseCalc`, `InterScore`, and `IntraScore`, respectively) were declared with the `__global const` qualifier, as these altogether require ~270 kB.

This was implemented in the third phase and maintained during the fourth one.

3. For most FPGA designs, fixed-point arithmetic leads to faster designs compared to their floating-point counterparts. However, for `InterScore` and `IntraScore`, floating-point resulted in an overall faster design, which can be attributed to the hardened floating-point Digital Signal Processing (DSP) units in the Arria 10 FPGA [81]. Surprisingly for `PoseCalc`, which initially suffered from a latency of II = 36 (Section 5.2.2.2), a fixed-point representation reduced it down to II = 10. This can be explained by the fact that the 30 addition and subtraction operations of the problematic datapath expressed in fixed-point were implemented using Adaptive Logic Modules (ALMs) instead of DSPs, thus avoiding the DSP latency of four clock cycles each [81]. Similar to the previous technique, this one was also implemented in the third phase and maintained during the fourth one.

It is important to mention that in Section 5.3.3, we measure the *combined* impact of the last two optimizations just described – i. e., caching (using selectively `__constant` and `__global const` memory) and arithmetic precision (choosing conveniently fixed and floating point), and not the separate effect of each one of them.

From a productivity standpoint, this was motivated by the higher priority of achieving faster executions of the *overall* design in short development cycles, than in deeply understanding the separate effect of each technique. In fact, in initial development phases, different techniques (e. g., caching using `__local` memory, shift registers, etc) did *not* produce visible performance gains when evaluated independently. In some cases, *slowdowns* were observed when applying optimizations *suggested* by the FPGA vendor. Although independently evaluating each optimization would be ideal for any design, it became *non-practical* in later development phases due to the long building times required for each FPGA bitstream (∼eight hours).

5.3 EXPERIMENTAL EVALUATION

5.3.1 *Setup*

Similarly to the data-parallel version (Chapter 4), re-docking experiments are performed here. Regarding the dataset, five ligand-receptor inputs obtained from the Protein Data Bank (PDB) [17] were tested (PDB IDs: `3ptb`, `1stp`, `4hmg`, `3ce3`, `3c1x`). All MD parameters were set to the default values as suggested in AutoDockTools [76].

At the time of development, the Intel FPGA SDK for OpenCL v16.0 was latest version suported by the corresponding board support package.

The hardware platforms used in our experiments were:

- An Intel i5-6600K CPU clocked at 3.5 GHz.

- A Gidel Proc10A card equipped with an Arria 10 GX 1150 FPGA and 16 GB RAM.

The CPU was used for collecting baseline characteristics of the original single-threaded implementation. On the other hand, FPGA binaries were built using the Intel FPGA SDK for OpenCL v16.0 compiler.

5.3.2 *Validation*

Even though this study focuses on the methodological aspects, it is important to show that the OpenCL version operates correctly, especially with the changed arithmetic, i. e., the mix of fixed and floating point.

Since previous acceleration studies of AutoDock [144, 146, 233] demonstrate that a reduced precision does not diminish the MD quality with respect to the original AutoDock (in double-precision floating-point), a 16.16 fixed-point format was utilized for the `LS`, `PoseCalc`, `InterScore`, and `IntraScore` kernels. This format allows simply re-purposing the OpenCL `int` and `long` primitive types, and was sufficient to represent genotypes and rotations generated in `LS` and `PoseCalc`. For the `InterScore` and `IntraScore` kernels, this format might not be sufficiently precise in cases where scores might reach out-of-bound values, i. e., when the *van der Waals* ($\frac{A}{r_{ij}^{12}} - \frac{B}{r_{ij}^{6}}$) and hydrogen bonding ($\frac{C}{r_{ij}^{12}} - \frac{D}{r_{ij}^{10}}$) terms grow rapidly as the interatomic distances (r_{ij}) become very short. However, the erroneous molecular poses derived from these incorrect out-of-bound values are so bad that they will be discarded by the genetic algorithm anyway. While such precision issues were not observed in practice, and because the floating-point implementations for the `InterScore` and `IntraScore` kernels were actually faster than fixed-point counterparts (Section 5.2.3), further experiments were performed using floating-point representation in such kernels (Table 5.2).

Fixed-point format was beneficial only when used for `PoseCalc`. This format reduced II in `PoseCalc` (Section 5.2.3), and the overall execution time (Section 5.3.4). On the contrary, floating point was preferred for `InterScore` and `IntraScore` kernels.

Therefore, the resulting designs from each development phase were compared against the serial baseline according to three key metrics already introduced in Section 2.2.6: lowest binding score (LBS), root mean square deviation (RMSD), and size of best cluster (SBC). As similar results were obtained with small designs, only the MD validation of the largest one (nine `LS` kernels, Section 5.2.2.4) is presented in Table 5.1.

The explanation for the SBC discrepancies in Table 5.1 – between OCLADock-FPGA and AutoDock – are the same as those already provided in Section 4.2.2 for the initial OCLADock design for GPUs. In our previous FPGA-related publication [234], we stated that the above discrepancies were due to the different selection scheme used (*binary tournament*) with respect to that used in AutoDock (*proportional selection*). As later discovered (March 2018), it was in fact the score

Table 5.1: Functional validation of OCLADock-FPGA vs. single-threaded AutoDock, both using the Solis-Wets local-search method. RMSD values are omitted for simplicity. All results were obtained using 100 LGA runs. Best values within each criterion are colored.

Ligand-receptor input			Lowest binding score (LBS) (kcal/mol)		Size of best cluster (SBC)	
ID	N_{rot}	N_{atom}	Serial baseline	OpenCL FPGA	Serial baseline	OpenCL FPGA
`3ptb`	2	13	−5.55	−5.53	100	66
`1stp`	5	18	−8.37	−7.76	100	69
`4hmg`	10	27	−3.68	−4.11	34	25
`3ce3`	5	37	−11.59	−10.88	94	48
`3c1x`	8	46	−13.61	−12.61	90	22

implementation lacking of *smoothing* the real cause of SBC (and RMSD) discrepancies. The smoothing feature was added into the scoring function implementation and pushed onto the project repository [186] later (August 2018). Corresponding code changes fixed the above issues, and have virtually no impact on execution time on FPGA tests.

5.3.3 *Design configurations and resource utilization*

Table 5.2 lists the four development phases and their respective design configurations (*DC1*, *DC2*, *DC3*, and *DC4* {*a*, *b*, *c*, *d*}) that summarize the 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 `PoseCalc` (*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-GA` (used in selection, crossover, mutation, selection for local-search), one `Arbiter`, one `PoseCalc`, one `InterScore`, and one `IntraScore`. Table 5.3 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 `LGA` removes the hardware required to carry genotype data in `GA` and `LS`, both initially managed within `LGA`. On the other hand, there is an expected overall increase in resource usage when going from design *DC3* through *DC4a* towards *DC4b*, which directly corresponds to the increase (from three up to five instances) of the `LS` replication. More-

Table 5.2: Development phases and design configurations.

Development phase	Design configuration	# `LS` replicas	Arithmetic representation		
			`LS`	`PoseCalc`	`InterScore` `IntraScore`
First	DC1	1	float	float	float
Second	DC2	1	float	float	
Third	DC3	3	fixed	fixed	
Fourth	DC4a	5	fixed	fixed	float
	DC4b	9	fixed	fixed	
	DC4c	9	float	fixed	
	DC4d	9	float	float	

Table 5.3: FPGA resource utilization and maximum frequency.

Design configuration	ALMs 427 200	%	RAMs 2713	%	DSPs 1518	%	Frequency (MHz)
DC1	129 301	30	1075	40	388	26	215.2
DC2	128 018	30	999	37	262	17	174.4
DC3	158 586	37	1799	66	548	36	187.5
DC4a	177 509	42	1826	67	586	39	172.6
DC4b	222 372	52	1880	69	662	44	187.5
DC4c	220 427	52	1898	70	659	43	185.7
DC4d	219 359	51	1944	72	383	25	185.7

over, 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 (∼186 MHz). Smaller designs (*DC2* and *DC4a*, both at ∼173 MHz) do not always result in higher frequencies compared to larger ones (*DC4* {*b*, *c*, *d*} at ∼186 MHz). Furthermore, higher frequencies do not necessarily imply faster circuits, e. g., *DC1*, capable of running at ∼215 MHz, is at least ∼4.4x slower than the serial baseline (Table 5.4).

5.3.4 *Execution performance*

Table 5.4 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 overall *decrease* in execution performance (except for the small `3ptb` input), which can be attributed to the lower frequency

Table 5.4: Execution time (s) for 100 LGA runs.

Design configuration	Ligand-receptor input				
	`3ptb`	`1stp`	`4hmg`	`3ce3`	`3c1x`
Serial CPU	586	836	1416	1867	2841
DC1	2903	5784	6636	8519	12 573
DC2	2550	6678	8121	9247	14 502
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

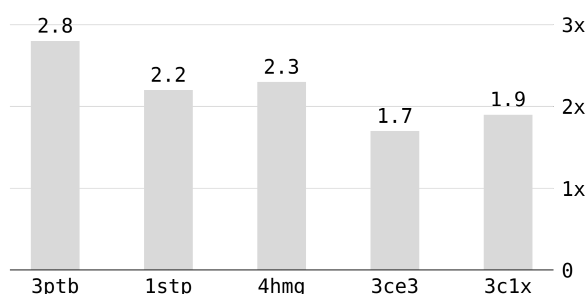


Figure 5.6: Speedups of OCLADock-FPGA fastest design DC4b vs. single-threaded AutoDock.

achieved (Table 5.3), and the increased computation required by larger ligand inputs, i. e., those with at least five torsions ($N_{\mathrm{rot}} \geq 5$) such as `1stp`, `4hmg`, `3ce3`, and `3c1x`.

Although *DC2* seemed to be going in the *wrong optimization direction*, it introduced the architectural modifications (Section 5.2.2.2) that led to the performance improvements in later designs *DC3* and *DC4*. This is reflected in the significant runtime reductions when going from *DC2* to *DC3*, e. g., a maximum difference of ∼12 000 s for `3c1x`. These improvements are the result of the `LS` replication, the careful allocation of constant look-up tables, and selective usage of fixed-point arithmetic precision (Section 5.2.3).

Moreover, in Table 5.4, when going from *DC3* through *DC4a* towards *DC4b*, there is progressive speedup of the execution runtime due to 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 `PoseCalc` calculations are expressed in floating-point as in *DC4d* (e. g.,

Table 5.5: Compute-energy consumption (kJ) for 100 LGA runs.

Design configuration	Ligand-receptor input				
	`3ptb`	`1stp`	`4hmg`	`3ce3`	`3c1x`
Serial CPU	11.8	16.7	28.1	36.3	54.9
DC4b	6.3	11.5	18.7	32.3	44.6

a maximum difference of 321 s for `1stp`). These results are due to the II improvement of the outermost-loop in `PoseCalc` achieved with fixed-point (*DC4b*, II = 10) vs. floating-point (*DC4d*, II = 36). The case of `InterScore` and `IntraScore` is the opposite, because expressing their calculations in fixed-point resulted in ∼20 % of performance decrease (Section 5.2.3), due to larger hardware area that led to lower frequencies (< 170 MHz) for a design comparable to *DC3* in Table 5.3.

For the two score-calculating 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 speedups obtained with the fastest design (*DC4b*) over the sequential baseline were 2.8x (for `3ptb`) and 1.7x (for `3ce3`). This correspondance can be explained by the computation effort imposed by molecular inputs, whose complexity directly increases in cases where more atoms and rotatable bonds are present in the ligand.

5.3.5 *Compute-energy efficiency*

Table 5.5 reports the compute-energy consumption of the two target devices used: the single CPU core, and the FPGA. For the CPU case, the drawn power was sampled in T_{sampling} = 50 ms intervals using power performance counters. The power samples were then integrated over time to derive the energy. For the FPGA case, estimated power values from fully placed-and-routed OpenCL projects were obtained using *quartus_pow*, similarly as in [227]. The power estimate of ∼30 W was multiplied by the respective runtime (Table 5.4) to obtain the energy. Clearly, bigger/smaller energy-efficiency gains (1.8x for `3ptb`, 1.1x for `3ce3`) correspond to bigger/smaller speedup factors (2.8x for `3ptb`, 1.7x for `3ce3`) as already shown in Table 5.4.

5.3.6 *Further analysis*

5.3.6.1 *Comparison against state-of-the-art accelerated AutoDock*

Compared against the RTL-based implementation by Pechan, Fehér, and Bérces [146] (achieving ∼23.3x of average speedup on a Xilinx

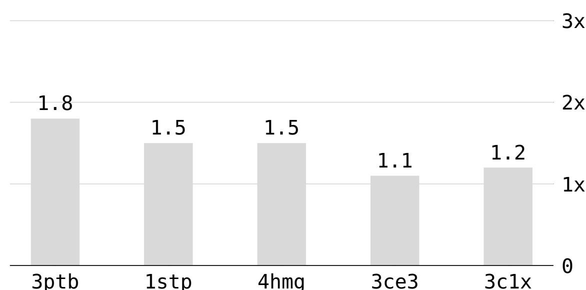


Figure 5.7: Energy-efficiency gains of OCLADock-FPGA fastest design DC4b vs. single-threaded AutoDock.

Virtex 4), much *lower* arithmetically-averaged speedups (∼2.2x on an Intel Arria 10) were achieved in this chapter.

Despite all efforts of optimizing the FPGA design discussed in this chapter, GPUs do an even better job in MD acceleration. In Chapter 4, improvements over the serial baseline reached gain factors (for `3c1x`) of up to ∼55.7x (speedup) and ∼6.3x (energy efficiency). The fact that, in this Chapter 5, the optimal pipelining (II = 1) was achieved for the `InterScore` and `IntraScore` kernels belonging to the bottleneck chain, suggests that the control mechanisms used in channel-based communications, as well as within the `Arbiter` kernel, are most likely not yet optimal in the currently proposed FPGA design.

5.3.6.2 *Tool support and productivity*

In general, for each development phase, the design specification and its corresponding emulation-based verification were as easy on the FPGA as with GPUs. Before generating FPGA binaries, optimization reports provided by the tool were extensively utilized 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 5.2.2.1) was completed in ∼four weeks.

However, during each development phase, the corresponding validation on actual hardware and subsequent optimization cycles were much more involved. At this stage, the hardware profiler was used 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, this was not a *guarantee* that the mapped designs would execute as expected on the actual FPGA. Among the known emulator limitations [83], the most critical issue was the data-consistency errors created by concurrent global-memory accesses from different kernels (Algorithm 4).

Moreover, it was essential to consider the possible reorderings of operations potentially performed by the OpenCL compiler. This is the

case for the LGA kernel, where several read and write channel calls happen at *different* variable scopes throughout the entire MD execution. At first glance, the order of channel calls can be enforced by using *channel fences* [83]. 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, guard variables had to be explicitly introduced, and manually set/reset to enforce a valid execution order.

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

5.3.6.3 *Programming recipes and challenges for achieving higher performance*

During the entire development, the aim was to achieve the highest possible performance estimation (II = 1) for each single work-item kernel. Although this was not possible in all parts of the proposed design, the difficulty was mitigated by spliting large sections (LGA in the first development cycle) into smaller instances (as multiple LS and LFSR-LS kernels) in order to harness the parallelism such OpenCL kernels. Moreover, appropriate data allocation on either on-chip or off-chip memory, as well as arithmetic representation, were considered as key tools for achieving higher performance. The benefits of these general recipes are reflected in the most drastic performance improvements, i. e., moving from *DC2*, through *DC3* towards *DC4b* (Section 5.3.3).

From the software-programming perspective, a big challenge faced during development was the required *awareness* of the underlying hardware. Even when developing at the OpenCL level of abstraction, this translated into the need for:

- Explicit synchronization between multiple read and write channels within single work-item kernels for ensuring correctness.
- Re-arranging the local memory layout for increasing II.
- Code refactoring using hardware constructs such as shift-registers, for better pipelining and unrolling loops.

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

6

ENHANCING OCLADOCK WITH GRADIENTS OF THE SCORING FUNCTION

While the previous chapters focused on increasing the performance of AutoDock by its parallelization on multi-core CPUs, GPUs, and FPGAs, this chapter focuses on algorithmic improvements to also achieve a higher quality-of-results. Thus, this chapter details the incorporation of local-search methods based on gradients of the scoring function into OCLADock, which are an alternative to the legacy Solis-Wets method. Moreover, it provides an evaluation of its performance and compute-energy efficiency. Contents of this chapter have been submitted to [165, 235, 236].

6.1 GRADIENT-BASED OPTIMIZATION

As already described in Section 3.3, previous studies [53, 188] suggest that gradient-based methods can significantly outperform Solis-Wets, when used as local search in AutoDock. The key idea of this optimization is to obtain the derivatives of the scoring function terms, and use them to efficiently direct the LGA towards stronger molecular poses.

6.1.1 *Gradient calculation*

The gradient calculation, denoted as GC, is the process by which the gradient g is derived from the scoring function SF with respect to each variable, i. e., gene, of the genotype Ω:

$$g = \nabla \text{SF}\left(\Omega\right) = \frac{\partial \text{SF}}{\partial x}, \frac{\partial \text{SF}}{\partial y}, \frac{\partial \text{SF}}{\partial z}, \frac{\partial \text{SF}}{\partial \phi}, \frac{\partial \text{SF}}{\partial \theta}, \frac{\partial \text{SF}}{\partial \alpha}, \frac{\partial \text{SF}}{\partial \psi_1}, \ldots, \frac{\partial \text{SF}}{\partial \psi_{N_{\text{rot}}}} \quad (6.1)$$

Since SF is expressed as the sum of intermolecular and intramolecular interactions (Section 2.2.4), the gradients are hence composed of analogous parts calculated using numerical and analytical derivatives, respectively. Besides the fact that the pose calculation is identical in both SF and GC, Algorithm 9 shows that the calculation of both intermolecular and intramolecular gradient parts follow the same loop structure as their corresponding SF counterparts (Algorithm 3). For incorporating the gradients into the LGA, the subsequent calls in Algorithm 9 perform their required conversion from the atomic space

(originally using the interatomic distance r_{ij} as in Equation 2.3) into the genetic space (adopting the form in Equation 6.1).

Algorithm 9: Gradient Calculation (GC)

```
/* Low-Level Parallelism */
Function GC (genotype)
    /* Gradients in atomic space */
    for each rot-item in N_pose-rot do
        PoseCalculation
    for each lig-atom in N_atom do
        InterGradient
    for each intra-pair in N_intra-contrib do
        IntraGradient
    /* Conversion from atomic into genetic space */
    Gtrans // Translational gradients

    Grigidrot // Rigid-body rotation gradients

    Grotbond // Rotatable-bond gradients
```

6.1.2 *Gradient conversion from atomic into genetic space*

Since the analytical form of the scoring function SF (Equation 2.3) is expressed in the atomic space, the gradient calculation GC is a two-step process. In the first step, the atomic partial derivatives a_i of the scoring function SF were calculated with respect to the motion of single ligand atom i in x, y and z directions:

$$a_i = \nabla \text{SF}\ (x_i,\ y_i,\ z_i) = \frac{\partial \text{SF}}{\partial x_i}, \frac{\partial \text{SF}}{\partial y_i}, \frac{\partial \text{SF}}{\partial z_i} \tag{6.2}$$

In the second step, the gradient expressed in the atomic space (i. e., in terms of a_i) is converted into the genetic space. This conversion is specific for each of the three motion types: translation, rigid-body rotation, and rotatable bonds.

The partial derivatives of SF with respect to translation genes x, y, and z (`Gtrans` in Algorithm 9) are calculated as the sum of the atomic partial derivatives of all N_{atom} ligand atoms:

$$\frac{\partial \text{SF}}{\partial x}, \frac{\partial \text{SF}}{\partial y}, \frac{\partial \text{SF}}{\partial z} = \sum_{i}^{N_{\text{atom}}} a_i = \sum_{i}^{N_{\text{atom}}} \frac{\partial \text{SF}}{\partial x_i}, \frac{\partial \text{SF}}{\partial y_i}, \frac{\partial \text{SF}}{\partial z_i} \tag{6.3}$$

The partial derivatives of SF with respect to rigid-body rotation genes ϕ, θ, and α (`Grigidrot` in Algorithm 9) are calculated from the torque vector τ:

$$\tau = \sum_{i}^{N_{\text{atom}}} r_i \times a_i \tag{6.4}$$

where:

r_i vector from the rotation center of the ligand to atom i

$\times$ cross product

a_i atomic partial derivatives of atom i

According to this definition, τ represents the derivative of SF with respect to rotation over the axis containing τ. The magnitude of the derivative is the length of τ, in units of score per radian.

The next step is to convert τ into the partial derivatives of SF with respect to ϕ, θ and α. These genes define rotation of the ligand in the axis-angle representation, where ϕ and θ define the axis of rotation in spherical coordinates, and α defines the amount of rotation. Considering an initial genotype with $\{\phi_0, \theta_0, \alpha_0\}$ and a resulting genotype with $\{\phi_1, \theta_1, \alpha_1\}$ upon rotation of the ligand about the τ axis by 0.001 radians from the initial pose, the partial derivatives are then:

$$\frac{\partial \text{SF}}{\partial \phi} = \frac{||\tau||}{0.001} (\phi_1 - \phi_0)\, h_1(\theta_0)\, h_2(\alpha_0) \tag{6.5}$$

$$\frac{\partial \text{SF}}{\partial \theta} = \frac{||\tau||}{0.001} (\theta_1 - \theta_0)\, h_2(\alpha_0) \tag{6.6}$$

$$\frac{\partial \text{SF}}{\partial \alpha} = \frac{||\tau||}{0.001} (\alpha_1 - \alpha_0) \tag{6.7}$$

where:

$||\tau||$ module of a vector τ

$h()$ empirically-discovered approximation functions

To define the $h()$ functions, the numerical partial derivatives of SF were calculated with respect to ϕ, θ and α, by changing the value of each of these genes by a small amount in the range of $[10^{-8}, 10^{-3}]$, and dividing the difference in the resulting score by the change in the gene.

Finally, the partial derivatives of SF with respect to rotatable bond genes ψ_j (`Grotbond` in Algorithm 9) are the projection of torque vectors τ_j on a unit vector u_j defining the axis of rotation of the rotatable bond of interest:

$$\frac{\partial \text{SF}}{\partial \psi_j} = \tau_j \cdot u_j \tag{6.8}$$

where:

j index of the rotatable bond

$\cdot$ dot product

Torque vectors are calculated as in Equation 6.4, but exclusively for atoms affected by the rotatable bond of interest. This is because each rotatable bond ψ_j rotates a different set of ligand atoms, and hence, is associated with a different torque τ_j.

6.1.3 *Gradient-based local-search methods*

Multiple optimization methods based on gradients exist in the literature [16, 123]. In this work, we have experimented with three of them: Steepest Descent [40], FIRE [20], and ADADELTA [221]. While the first is a generic one, the last two were chosen due to their suitability for minimizing objective functions describing molecular interactions.

6.1.3.1 *Steepest Descent*

The Steepest Descent *method was first published by Debye [40] in 1909.*

The basic idea is to generate a new solution by taking steps from a previous one. For Steepest Descent, these steps are directly proportional – with a factor λ – to the negative of the gradient g of the previous iteration t. Hence, the update of a solution holding a genotype Ω is described as:

$$\Omega_{t+1} = \Omega_t - \lambda\, g_t \tag{6.9}$$

Choosing an appropriate value of λ is non-trivial, especially because the gradient magnitude can be very large due to the repulsive term $(A_{\text{ij}}/r_{\text{ij}}^{12})$ in SF (Equation 2.3). A very small λ would be preferred in case of unfavorable poses, whereas a larger λ would be beneficial in more acceptable poses. To mitigate the negative effects of unsuitable λ values, and thus, to help finding better solutions, *limits* for the change in genes were defined according to their types:

- Maximum translation change: 2.0 Å.
- Maximum orientation or torsional change: 0.5 radians.

In the first iteration (t = 0), λ is set to the maximum possible value such that these limits are not violated. If the score decreases (i. e., improves), λ is increased by 20 %. Otherwise, λ is decreased by 50 % and Ω_t is reverted to Ω_{t-1}. At every iteration, λ is tested to guarantee that the maximum change in any gene does not exceed the above limits.

6.1.3.2 *FIRE*

According to [99], FIRE stands for Fast Inertial Relaxation Engine*.*

The main idea of FIRE is analogous to that of a *blind skier* searching for the fastest way to the bottom of a mountain [20], whose landscape is described by SF. At each iteration t, the skier should introduce acceleration in a direction that is *steeper* than the current motion direction, if the skier is moving downhill.

Such a downhill movement corresponds to a positive value of a term known as *kinetic power*, so that the skier stops as soon as this value becomes negative. Specifically, the kinetic power P_t is defined by two terms: kinetic force and velocity. The kinetic force F_t is opposite to

the gradient g_t, i.e., $F_t = -g_t$. The velocity v_t describes the direction and speed at which the skier moves.

$$P_t = F_t \cdot v_t \tag{6.10}$$

$$v_t = (1 - \alpha_F)\, v_t + (\alpha_F)\, \frac{F_t}{||F_t||}\, ||v_t|| \tag{6.11}$$

where:

$\cdot$	dot product
α_F	FIRE hyperparameter

Remember that α represents a rotational gene, while α_F is a FIRE hyperparameter.

The genotype Ω is updated using the velocity v_t, and a factor dt whose value is either decreased or increased depending whether the skier moves uphill or downhill, respectively:

$$\Omega_{t+1} = \Omega_t + (dt)\, v_t \tag{6.12}$$

The idea of using an adaptable factor dt is similar to that of using variable λ values in Steepest Descent. However, the FIRE optimization does not use gradients directly, but a more sophisticated search vector (i.e., velocity v_t) derived from gradients.

6.1.3.3 *ADADELTA*

The basic idea of ADADELTA [221] is to alleviate the task of choosing a learning rate of the variables to be optimized. Applied to MD, ADADELTA consists in introducing a new *dynamic* learning-rate (i.e., an update vector for genotypes) that is computed per-dimension (i.e., per-gene) using *first-order* derivative information. In other words, the genotype Ω is updated using an update vector $\Delta\Omega$ at each iteration t:

$$\Omega_{t+1} = \Omega_t + \Delta\Omega_t \tag{6.13}$$

The value of the update vector $\Delta\Omega_t$ depends not only on the gradient g_t, but also on the *history* of past gradients and past update vectors:

$$\Delta\Omega_t = -\frac{\sqrt{E[\Delta\Omega^2]_{t-1} + \epsilon}}{\sqrt{E[g^2]_t + \epsilon}}\, g_t \tag{6.14}$$

where:

$E[\Delta\Omega^2]$	running average of squared updates
$E[g^2]$	running average of squared gradients

The vectors $E[\Delta\Omega^2]$ and $E[g^2]$ have the same number of elements (N_{genes}) as the genotype Ω.

The mathematical relationship between last two terms is as follows:

$$E[\Delta\Omega^2]_t = \rho\, E[\Delta\Omega^2]_{t-1} + (1-\rho)\, \Delta\Omega_t^2 \tag{6.15}$$

$$E[g^2]_t = \rho\, E[g^2]_{t-1} + (1-\rho)\, g_t^2 \tag{6.16}$$

where:

ϵ ADADELTA hyperparameter 1

ρ ADADELTA hyperparameter 2

In particular, the constant ϵ prevents the denominator in Equation 6.14 from becoming zero, if $E[g^2]$ is zero. Furthermore, ϵ is required to produce non-zero updates in the first iteration ($t = 0$), because the running average of squared updates of the preceeding iteration $E[\Delta\Omega^2]_{-1}$ is assumed to be zero. Based on our own tuning experiments [165], ϵ and ρ were set to 0.01 and 0.8, respectively.

Although the higher complexity with respect to previous methods, ADADELTA can help finding good solutions by using a dynamic learning-rate for each gene. As this learning rate depends on averaged terms from previous iterations, ADADELTA might provide a more consistent method for updating genotypes.

6.1.3.4 *Comparison*

The aforementioned gradient-based methods leverage *first-order* derivatives, in contrast to higher-order methods e. g., BFGS [123]. The choice of first-order methods was motivated by their reduced complexity, and because some of them (e. g., FIRE) can be competitive with BFGS for optimizing functions describing molecular interactions [20].

Moreover, each of the chosen first-order methods has a number of hyperparameters that influence the calculation of the update vector:

- Steepest Descent has four: λ, and the three maximum limits introduced here for the changes in motion types (translation, orientation, torsion).
- FIRE has five: α_F, two for the increasing and decreasing dt rates, and other two indicated in [20].
- ADADELTA has only two: ϵ and ρ.

From the programming perspective, the more hyperparameters a method has, the harder it is to find suitable values for the concrete use-case. In fact, this is one of the motivations why in ADADELTA the number of hyperparameters is only two. Moreover, according to [221], the ADADELTA optimization is *insensitive* to hyperparameters.

6.1.4 *Incorporation into OCLADock*

Incorporating the above gradient-based local-search methods into OCLADock implies running an LGA with *extended* functionality. Such extended version still needs to run both `Krnl_GA` and `Krnl_LS` kernels following the program workflow already described in Section 4.1.2. The main difference is that now, it is possible to select the optimizer to be run in `Krnl_LS`. All local-search methods implemented in this work

are mutually exclusive, i. e., `Krnl_LS` performs only one among the overall four choices available at this point: the legacy Solis-Wets, and the three gradient-based methods newly incorporated here: Steepest Descent, FIRE, ADADELTA.

Algorithm 10 shows how OCLADock configures `Krnl_LS` to specifically run the ADADELTA implementation (Section 6.1.3.3). Due to the similar code structure and same parallelization level between SF and GC, as well as between Solis-Wets (Section 2.2.3) and any chosen gradient-based method, the OpenCL parallelization still follows the same strategy used in Figure 4.2.

Algorithm 10: Incorporating ADADELTA local search into OCLADock. The other gradient-based methods, Steepest Descent and FIRE, are incorporated similarly, and thus are not shown here.

```
Program OCLADock
    for each LGA-run do
        /* Configuring overall program */
        LS-config = ADADELTA
        ...

        /* Writing initial populations to OpenCL device */
        ...

        /* Launching kernels on OpenCL device */
        while (N_score-evals < N^MAX_score-evals) and (N_gens < N^MAX_gens) do
            Krnl_GA
            Krnl_LS (LS-config)
            ...

        /* Reading resulting populations from OpenCL device */
        ...

    /* Low-Level Parallelism */
    Function ADADELTA (genotype)
        gradient = GC (genotype)

        while (N_LS-iters < N^MAX_LS-iters) do
            new-genotype = update-rule (genotype, gradient)

            if SF (new-genotype) < SF (genotype) then
                genotype = new-genotype

            gradient = GC (genotype)
```

Moreover, as described in Section 6.1.1, SF and GC calculate poses identically, and share the same loop-structure for their intermolecular and intramolecular components. In order to leverage data locality in the gradient-based implementations, SF and GC calculations are grouped together as much as possible. Basically, a single pose calculation is used for both scores and gradients, whereas structure-

equivalent SF and GC calculations are fused into single intermolecular and intramolecular loops. Gradient conversion (Section 6.1.2) was left unmodified. This code refactoring results in faster gradient-based executions (∼18 %) compared to an initial design where SF and GC calculations were invoked separately.

6.2 EXPERIMENTAL EVALUATION

6.2.1 *Setup*

This prepared set covers a large range of rotatable bonds supported by AutoDock: $N_{rot}^{MAX} = 32$.

Twenty ligand-receptor inputs from well-established sets for assessing MD methodologies were selected, including:

- Eleven from Astex [71]: `1u4d`, `1xoz`, `1yv3`, `1owe`, `1oyt`, `1ywr`, `1t46`, `2bm2`, `1mzc`, `1r55`, `1kzk`.
- Four from CASF-2013 [104]: `3s8o`, `1hfs`, `1jyq`, `2d1o`.
- Five from PDB [17]: `5wlo`, `5kao`, `3drf`, `4er4`, `3er5`.

In addition, as suggested for high-performance workloads [13], hyperthreading was disabled for all CPU instances used.

The following describes the hardware configuration. For the baseline test, i. e., the measurement of execution time and power draws of the original single-threaded AutoDock version 4.2.6 (implementing only the Solis-Wets local search), a Xeon E5-2666 clocked at 2.6 GHz CPU core was used. For parallel executions, on-premise and cloud accelerators based on commercial GPUs, CPUs, and FPGAs were evaluated (Table 6.1). At the time of writing, the best choice in terms of performance offered by Amazon Web Services (AWS) CPU instances would be the `c5.18xlarge` rather than the `c4.8xlarge` [11]. However, the `c4.8xlarge` was also taken into account because it was the *only* platform among the higher-end CPU instances that actually supported real-time power sampling (Section 6.2.6).

6.2.2 *Validation*

For consistency, all experiments follow again a re-docking approach. The configuration of the most important LGA parameters include: a population size of $P = 150$ individuals, $N_{\text{score-evals}}^{\text{MAX}} = 2\,048\,000$ score evaluations, as well as $N_{\text{gens}}^{\text{MAX}} = 99\,999$ generations. The latter parameter was set to a considerably larger value (99 999) than the default one (27 000) in order to ensure that the program is terminated *only* when it reaches the specified $N_{\text{score-evals}}^{\text{MAX}}$.

Table 6.2 compares the original single-threaded AutoDock (running only Solis-Wets) against the proposed OpenCL implementations with four different local-search methods (Solis-Wets, Steepest Descent, FIRE, ADADELTA) in terms of the three evaluation criteria (Section 2.2.6) for the entire dataset. Contrary to previous chapters, at this point in development, the *smoothing* feature of the scoring function *is* already

Table 6.1: Hardware and software setup in terms of instance type, peak memory bandwidth (GB/s), peak single-precision FP performance (GFLOP/s), number of OpenCL compute units (# CU), preferred OpenCL work-group size (WG_{size}), and tool versions.

Device Name	Instance Type	GB/s GFLOP/s	# CU	Preferred WG_{size}
AMD Radeon RX Vega 56 GPU	On-premise	410 10 566	56	64
Nvidia Tesla V100 GPU	AWS `p3.2xlarge`	900 15 700	80	32
Intel Xeon E5-2666 v3 @ 2.6 GHz 18-core CPU	AWS `c4.8xlarge`	136 1500	18	128
Intel Xeon Platinum 8124M @ 3.0 GHz 36-core CPU	AWS `c5.18xlarge`	260 3456	36	128
Gidel Proc10A Arria 10 GX 1150 FPGA	On-premise	20 1366	–	–
Host	Compiler: g++ 5.4.0 Flags: -O3			
Device	AMDAPPSDK 3.0, CodeXL [28] CUDA 9.0 Intel SDK-2017 Intel FPGA RTE v16.0 OpenCL flags: none			

implemented in OCLADock. Therefore, the numbers reported in Table 6.2 resulted from having an equivalent scoring function in both AutoDock and OCLADock.

Regarding lowest binding score (LBS), both AutoDock and OCLADock running Solis-Wets provide very similar values. OCLADock running Steepest Descent found solutions having lower (better) energies than when running Solis-Wets instead. All following examples report corresponding energy values in kcal/mol:

- `5kao`: -9.54 (Solis-Wets) vs. -11.08 (Steepest Descent)
- `1jyq`: -7.09 (Solis-Wets) vs. -12.77 (Steepest Descent)
- `3er5`: -6.49 (Solis-Wets) vs. -11.37 (Steepest Descent)

OCLADock FIRE produced solutions with significantly higher (worse) energies than OCLADock Steepest Descent for large inputs (e. g., `1jyq`, `3drf`, and `3er5`), while the binding energies calculated by OCLADock ADADELTA are the best in all cases.

Root mean square deviations (RMSD) reported are those of the resulting molecular pose that achieved the LBS threshold. For all small

Table 6.2: Functional validation of OCLADock vs. single-threaded AutoDock for Solis-Wets (SW) and gradient methods: Steepest Descent (SD), FIRE, and ADADELTA (AD). Serial SW results were obtained on a E5-2666 CPU core, while the OpenCL ones targeted a Vega 56 GPU, using 100 LGA runs for all cases. The best values within each case are colored. Non-colored cells indicate ligand-receptor cases where equally-good results were found, or that it was not possible to determine the best method.

Ligand-receptor input				Lowest binding score (LBS, in kcal/mol)					RMSD (Å) of LBS					Size of best cluster (SBC)				
ID	Source	N_{rot}	N_{atom}	Serial SW	OpenCL SW	OpenCL SD	OpenCL FIRE	OpenCL AD	Serial SW	OpenCL SW	OpenCL SD	OpenCL FIRE	OpenCL AD	Serial SW	OpenCL SW	OpenCL SD	OpenCL FIRE	OpenCL AD
1u4d	Astex	0	23	−7.26	−7.26	−7.27	−7.27	−7.27	1.38	1.36	1.35	1.36	1.36	86	98	23	23	99
1xoz	Astex	1	30	−10.49	−10.49	−10.49	−10.49	−10.49	0.29	0.26	0.27	0.26	0.27	99	97	97	51	100
1yv3	Astex	2	23	−11.67	−11.67	−11.68	−11.67	−11.68	0.50	0.50	0.50	0.50	0.50	96	98	50	14	91
1owe	Astex	3	27	−9.87	−9.89	−9.95	−9.74	−9.95	1.50	1.68	1.55	1.49	1.55	96	98	50	14	91
1oyt	Astex	4	34	−11.74	−11.76	−11.77	−11.74	−11.81	0.32	0.39	0.31	0.35	0.38	72	45	65	10	100
1ywr	Astex	5	38	−11.38	−11.32	−11.45	−11.17	−11.47	0.64	0.70	0.67	0.67	0.67	30	31	18	7	83
1t46	Astex	6	40	−14.59	−14.98	−15.13	−14.18	−15.13	0.39	0.23	0.58	0.52	0.56	34	53	48	2	71
2bm2	Astex	7	33	−9.35	−9.73	−9.59	−8.87	−10.58	1.30	1.52	0.94	1.03	5.31	5	7	10	4	36
1mzc	Astex	8	38	−9.60	−10.13	−10.19	−8.46	−10.37	1.04	1.23	1.23	2.79	1.21	4	6	6	8	44
1r55	Astex	9	27	−6.58	−6.97	−7.19	−6.96	−7.25	1.42	1.23	1.20	1.30	1.27	17	33	59	21	61
5wlo	PDB	10	46	−13.70	−14.13	−14.29	−13.20	−14.34	0.71	0.91	0.85	0.68	0.84	28	35	43	19	54
1kzk	Astex	11	45	−8.26	−10.61	−12.43	−9.00	−12.89	5.00	1.31	0.81	4.30	0.71	1	4	4	1	26
3s8o	CASF-2013	12	44	−6.41	−7.61	−7.75	−9.08	−11.52	5.07	2.83	1.44	2.06	2.51	1	1	1	1	6
5kao	PDB	15	44	−7.99	−9.54	−11.08	−8.98	−11.22	3.23	2.39	2.49	3.38	1.75	1	3	4	2	12
1hfs	CASF-2013	18	54	−10.87	−13.35	−14.72	−11.85	−16.06	6.08	2.69	4.71	10.95	3.80	1	2	1	1	5
1jyq	CASF-2013	20	60	−5.44	−7.09	−12.77	−6.77	−15.61	4.78	4.76	2.35	4.63	2.53	1	1	1	1	4
2d1o	CASF-2013	23	44	−7.44	−8.50	−8.52	−7.66	−11.99	3.78	7.19	3.32	7.49	3.20	2	1	1	1	5
3drf	PDB	26	63	−2.11	−3.28	−5.30	−2.01	−5.77	7.14	6.78	5.25	6.78	7.45	1	1	1	1	3
4er4	PDB	30	93	−2.13	−5.38	−9.75	−5.99	−14.09	5.40	5.37	5.09	5.45	3.50	1	1	1	1	1
3er5	PDB	31	108	0.93	−6.49	−11.37	−3.31	−14.99	3.46	3.39	4.47	6.49	3.88	1	1	1	1	1

inputs, all methods provide similar RMSDs, while for larger molecules (e. g., 1kzk, 3s8o, . . . , 3er5) ADADELTA clearly outperforms the rest. Additionally, an LGA run is successful if the RSMD of its returned ligand pose is within 2.0 Å from the reference x-ray pose. In Table 6.2, this success criterion is met by all LS methods for the first eleven inputs (1u4d, 1xoz, . . . , 5wlo). Although this is not true for the other larger inputs (i. e., RMSD > 2.0 Å), it can be noted that in five cases (1kzk, 5kao, 1jyq, 2d1o, and 4er4) out of nine largest (1kzk, 3s8o, . . . , 3er5), ADADELTA found the smallest RMSD among all methods.

Regarding the size of the best cluster (SBC), results from both AutoDock and OCLADock running Solis-Wets are very similar. However, in this SBC regard, OCLADock using ADADELTA pulls ahead having a clear superiority over all other methods for the first 18 inputs. FIRE results in considerably smaller clusters (worse) even for small- and medium-size inputs.

From a quality-of-results perspective[1], these results indicate that in most cases ADADELTA is a better choice for most inputs, followed by good and moderate results from Steepest Descent and Solis-Wets respectively, and FIRE being the *least* efficient among all.

6.2.3 *Profiling analysis for optimum local-search rate*

An important aspect of AutoDock is the selected local-search rate (lsrate). As already described in Section 2.2.2, during an LGA run, only a subset of the genetic population is subjected to LS optimization. By default, this subset is 6 % (lsrate = 6 %), e. g., only nine out of a population of P = 150 individuals will undergo LS. While for AutoDock this lsrate value was defined as the minimum possible to obtain sufficiently good solutions without incurring excessive performance penalties, the initial hypothesis was that increasing lsrate – to improve MD quality – will not result in performance degradation in our parallelization, since OCLADock processes many individuals *simultaneously*. Therefore, our analysis started by comparing lsrate = 6 % (the default) against lsrate = 100 % (the most computationally demanding) in order to select the most suitable lsrate in terms of performance.

Here, the profiling capabilities of CodeXL [28] – e. g., application timeline trace *and* performance counters *– were employed on the Vega 56 GPU.*

Table 6.3 reports profiling metrics of Krnl_LS (WG_{size} = 64wi, further discussed in Section 6.2.5.1) – *% Total time*, *# Calls*, *Avg. time*, and *% Occupancy* – when using three ligand-receptor inputs, which are representative of low (1u4d), medium (3s8o), and high (3er5) complexity.

1 The quality should be primarily evaluated based on the score. The RMSD is a valid metric for search performance only for inputs for which the scoring function is correct. Colleagues at TSRI estimate that the global minimum of the AutoDock version 4.2.6 scoring function corresponds to a correct pose (RMSD $\leq$ 2 Å) for about 75 % of inputs. Furthermore, the accuracy in the predicted binding energy is a computational chemistry score-modeling problem, which is not the focus of this work.

Table 6.3: Comparison of OCLADock local-search (LS) kernels using profiling metrics on the Vega 56 GPU (100 LGA runs).

Krnl_LS metrics	Input ID	Solis-Wets lsrate 6 %	Solis-Wets lsrate 100 %	Steepest Descent lsrate 6 %	Steepest Descent lsrate 100 %	FIRE lsrate 6 %	FIRE lsrate 100 %	ADADELTA lsrate 6 %	ADADELTA lsrate 100 %
Fraction of	1u4d	88	98	93	97	98	99	98	99
Total exec.	3s8o	92	99	98	99	99	99	99	99
time (%)	3er5	90	98	98	99	99	99	99	99
# Calls	1u4d	1738	128	6889	1501	725	46	719	46
to	3s8o	1560	120	2030	198	722	46	719	46
LS	3er5	2033	152	10 201	2573	722	46	719	46
Avg. time (ms)	1u4d	4	41	8	23	47	695	35	486
per LS kernel	3s8o	25	225	185	1610	588	9244	352	5526
enqueue	3er5	77	767	627	1343	3664	57 782	2924	5526
Device	1u4d	20	20	10	10	10	10	10	10
utilization	3s8o	20	20	10	10	10	10	10	10
Occupancy (%)	3er5	20	20	10	10	10	10	10	10

According to the % *Total time* metric, `Krnl_LS` comprises at least 88 % and up to 99 % of the overall OCLADock execution time.

The # *Calls* metric refers to the number of times `Krnl_LS` is enqueued for execution. In all cases, for each LS method, there is a seemingly paradoxical *reduction* in # *Calls* when increasing `lsrate` from 6 % to 100 %. This is because when running with `lsrate` = 100 %, more individuals are processed by `Krnl_LS`. This, in turn, results in *more* score calculations being performed in each `Krnl_LS` execution, thus requiring *fewer* enqueues to reach $N^{MAX}_{\text{score-evals}} = 2\,048\,000$ for each LGA run.

The *Avg. time* metric indicates the mean elapsed time (ms) of a single `Krnl_LS` execution. Running with `lsrate` = 100 % increases *Avg. time* in all cases because the number of individuals to evaluate increases. For instance, when considering `Krnl_LS` running FIRE on `3er5`, it can be noted that *Avg. time* increases by a factor of ∼21x (= $\frac{57782}{3664}$). This is because 100 % · 150 · 100 OpenCL work groups (calculated as $P \cdot R$ in Section 4.1.1) have to be distributed among 56 compute units (CUs of the Vega 56 GPU), instead of 6 % · 150 · 100 work-groups created when running `lsrate` = 6 %.

Additionally, the kernel % *Occupancy* measures how efficiently GPU resources are utilized during `Krnl_LS` execution. This is measured as the number of *in-flight* GPU *wavefronts* (N_{wave}), and is determined at runtime by four factors [28]:

- Size of work-group (WG_{size})
- Local-memory usage (kB) per work-group (LDS)
- Scalar GPR usage per work-item (SGPR)
- Vector GPR usage per work-item (VGPR)

LDS: local-data share.

GPR: general-purpose register.

As already described, WG_{size} was set equal to `64wi`, which narrows down the occupancy analysis to the other three factors. Table 6.4 shows that all factors affecting N_{wave} are independent from either the input (`1u4d`, `3s8o`, `3er5`) or the `lsrate` (6 % or 100 %) chosen. Factors like LDS and SGPR increase and decrease respectively when selecting ADADELTA (LDS: 10 496 kB, SGPR: 84) instead of Solis-Wets (4 096 kB, SGPR: 88). Based on the N_{wave} columns, the overall limiting factor is the VGPR, whose usage was 121 (Solis-Wets), 153 (Steepest Descent), and 152 (FIRE, ADADELTA) out of a device limit of 256. Considering that each Vega 56 CU has four vector units, then N_{wave} is estimated as $\frac{256}{\text{VGPR}} \cdot 4$, and hence, only eight (Solis-Wets) and four (Steepest Descent, FIRE, ADADELTA) – out of a device limit of 40 wavefronts – were active on the Vega 56 GPU. Based on [28], this results in % *Occupancy* values (Table 6.3) of 20 % for Solis-Wets, and 10 % for {Steepest Descent, FIRE, ADADELTA}.

Besides increasing the chances to improve the MD quality by optimizing more solutions through LS (executed in parallel) [165], the

Table 6.4: Resource utilization and its equivalent number of wavefronts in `Krnl_LS` for the experiment in Table 6.3. VGPR values, which limit the overall GPU occupancy, are highlighted.

Resource type	Resource utilization					Equivalent N_{wave}				
	SW	SD	FIRE	AD	Limit	SW	SD	FIRE	AD	Limit
LDS (kB)	4096	10 240	11 008	10 496	65 536	16	6	5	6	40
SGPR	88	89	92	84	104	32	32	32	32	40
VGPR	121	153	152	152	256	8	4	4	4	40

main advantage of using `lsrate` = 100 % instead of lower rates is the overall shorter executions (whose duration can be derived by multiplying # *Calls* and *Avg. time* from Table 6.3). For that reason, all of the next experiments are performed using `lsrate` = 100 %.

6.2.4 *Efficiency of gradient-based methods*

The algorithmic and execution efficiencies of the selected gradient-based methods are evaluated using that of Solis-Wets as reference. The purpose is to determine which gradient-based method is the best alternative to the legacy Solis-Wets method.

6.2.4.1 *Algorithmic efficiency*

Experiments in Section 6.2.2 showed that the ADADELTA local search was the *best* choice from a quality-of-results perspective. Here, we briefly expand the comparison between ADADELTA and the legacy Solis-Wets method used in OCLADock.

The methodology used to estimate the required number of scoring-function calls is beyond the scope of this thesis, but it is discussed in detail in our previous work [165].

The total execution time increases with the number of scoring function (SF) calls. The use of gradients *reduces* the number of SF calls required to find good solutions. An estimation of the number of calls required by Solis-Wets and ADADELTA to yield correct docking solutions revealed that for ligands with about eight rotatable bonds (N_{rot} = 8), ADADELTA requires ∼50x fewer calls than Solis-Wets. For ligands with about N_{rot} = 20, ADADELTA requires ∼1300x fewer calls. Overall, the use of gradients reduces the number of SF calls, especially for ligands with many rotatable bonds ($N_{rot} > 8$), resulting in faster and more efficient dockings.

6.2.4.2 *Execution efficiency*

Although calculating speedups (and compute energies in Section 6.2.6) using different local-search methods in the baseline (implementing only Solis-Wets) and OpenCL versions (in the case of gradients) is not completely fair, it is still essential to report them, as they indicate the performance penalties incurred by using the more complex gradient methods.

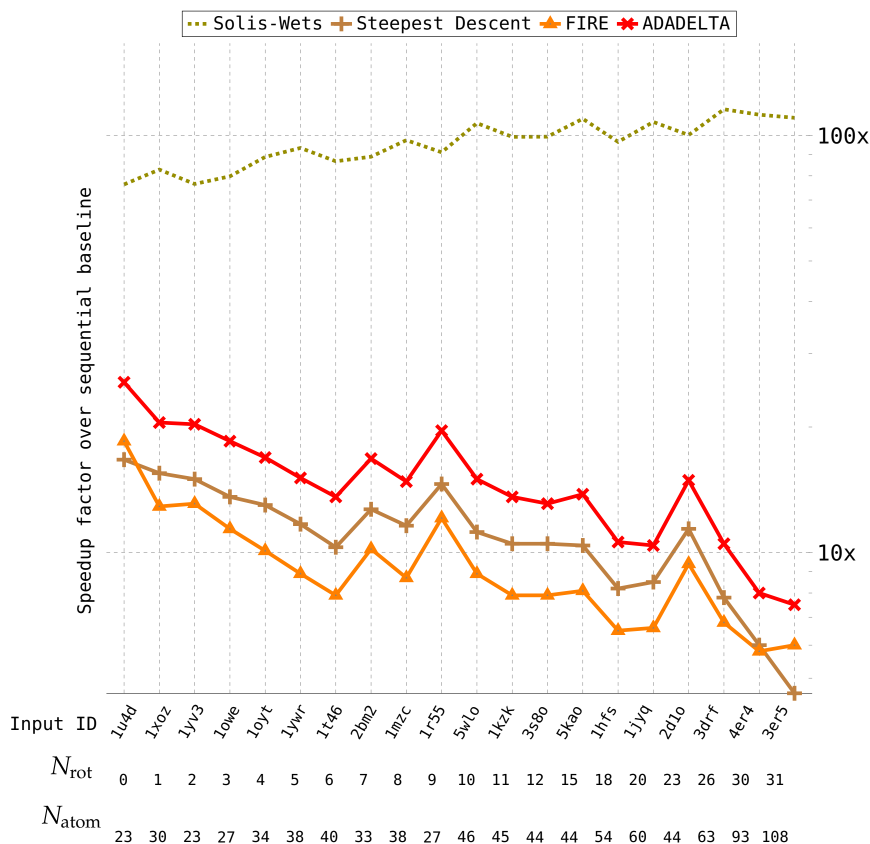


Figure 6.1: Speedups of OCLADock vs. single-threaded AutoDock achieved on a Vega 56 GPU (R = 100 LGA runs, `lsrate` = 100 %).

Figure 6.1 shows speedups achieved on a Vega 56 GPU. The highest speedup factors are obtained for all inputs using Solis-Wets, reaching a minimum of 76x (`1u4d` and `1yv3`) and up to 115x (`3drf`). Also for Solis-Wets, the speedup increases with larger N_{rot} and N_{atom} values, and as such, higher speedups are obtained with larger inputs such as `3drf`, `4er4`, and `3er5`.

Conversely, speedup factors for any gradient method *decrease* with larger N_{rot} and N_{atom} values. Speedups of Steepest Descent, FIRE and ADADELTA are significantly lower than those of Solis-Wets in all cases. The reason for this is the more complex gradient calculation itself (Section 6.1.1), and the gradient conversion from atomic into genetic space (Section 6.1.2), which together are more computationally demanding than the simple random delta generation of Solis-Wets (Section 2.2.3). The *fastest* gradient-based method was ADADELTA, achieving a minimum speedup of ∼7x against the baseline, but with much *higher* quality-of-results than other methods evaluated, as shown in Table 6.2.

6.2.5 *Portability to other accelerators*

From now on, further experiments focus on the legacy Solis-Wets, and ADADELTA, the best gradient-based method according to the algorithmic and execution efficiency criteria. Since the number of work-items in an OpenCL work-group (WG_{size}) can have a significant impact on the performance of the proposed *data-parallel* design, it is important to first determine its most suitable value.

6.2.5.1 *Finding the optimum size of OpenCL work-groups*

Since the number of work-items (`wi`) in a work-group (WG_{size}) can have a significant impact on performance, its most suitable value should be determined before deployment. For that purpose, we analyzed the speedups achieved on all selected accelerators when using different WG_{size} configurations, e. g., `16wi`, `32wi`, `64wi`, `128wi`, `256wi`. Figure 6.2 shows only the cases of `1u4d`, `3s8o`, and `3er5`, however it represents the general trend observed with *most* of the inputs in our dataset. For CPUs, as found in our previous work [233], a configuration of WG_{size} = `16wi` clearly leads to higher speedups on both Xeon Platinum 8124M and Xeon E5-2666 instances, despite the preferred WG_{size} = `128wi` (Table 6.1).

For GPUs, in contrast to [233], where WG_{size} = `64wi` was the fastest configuration for the only tested `lsrate` = 6 % on an AMD R9 290X GPU, Figure 6.2 shows that higher speedups can be obtained – besides `64wi` (most cases) – with either smaller (`32wi`) or larger (`128wi`) sizes, regardless of the chosen device. According to vendors guidelines [1, 132], a suitable WG_{size} is an integer multiple of either an AMD *wavefront* size (`64wi`), or a Nvidia *warp* size (`32wi`). By setting WG_{size} = `64wi` – i. e., the minimum multiple integer for any GPU from these two vendors – we aim to minimize the inter work-group communication overhead, which seem to be slowing down the program for larger WG_{size}s (e. g., `256wi`). More importantly, using WG_{size} = `64wi` increases the chances of achieving higher speedups in cases outside our dataset.

6.2.5.2 *Performance analysis*

For the Vega 56 GPU, such speedup metrics correspond to values reported in Figure 6.1.

Figure 6.3 provides an overall comparison of the speedups achieved on all selected GPU and CPU devices for each input in our dataset. For both Solis-Wets and ADADELTA executions, the reported speedups were obtained with respect to the original AutoDock running the Solis-Wets method as local search. Although calculating ADADELTA speedups with respect to a Solis-Wets baseline is somewhat arguable, they still offer meaningful performance gains out of the parallelization of a *more complex* LS method. Similarly as in Figure 6.1, it can be observed that

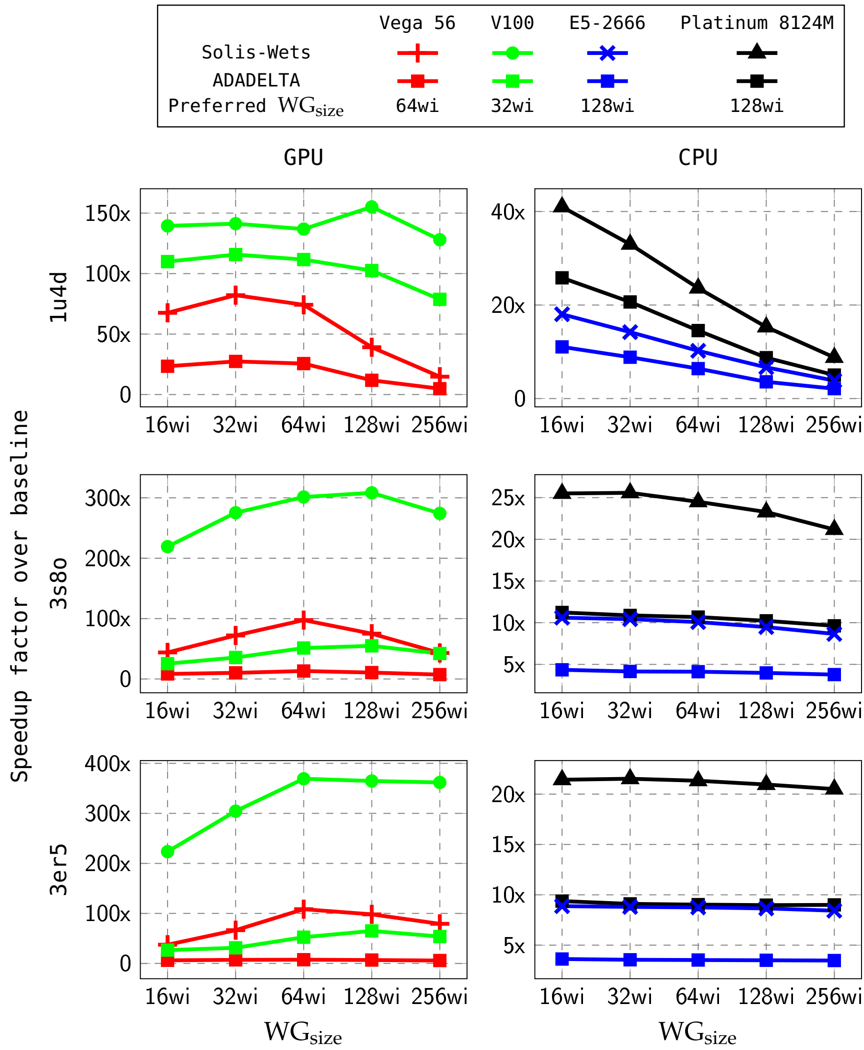


Figure 6.2: Speedups of OCLADock vs. single-threaded AutoDock achieved on selected GPU/CPU devices for different work-group sizes (R = 100 LGA runs, `lsrate` = 100 %). Vertical scales are different.

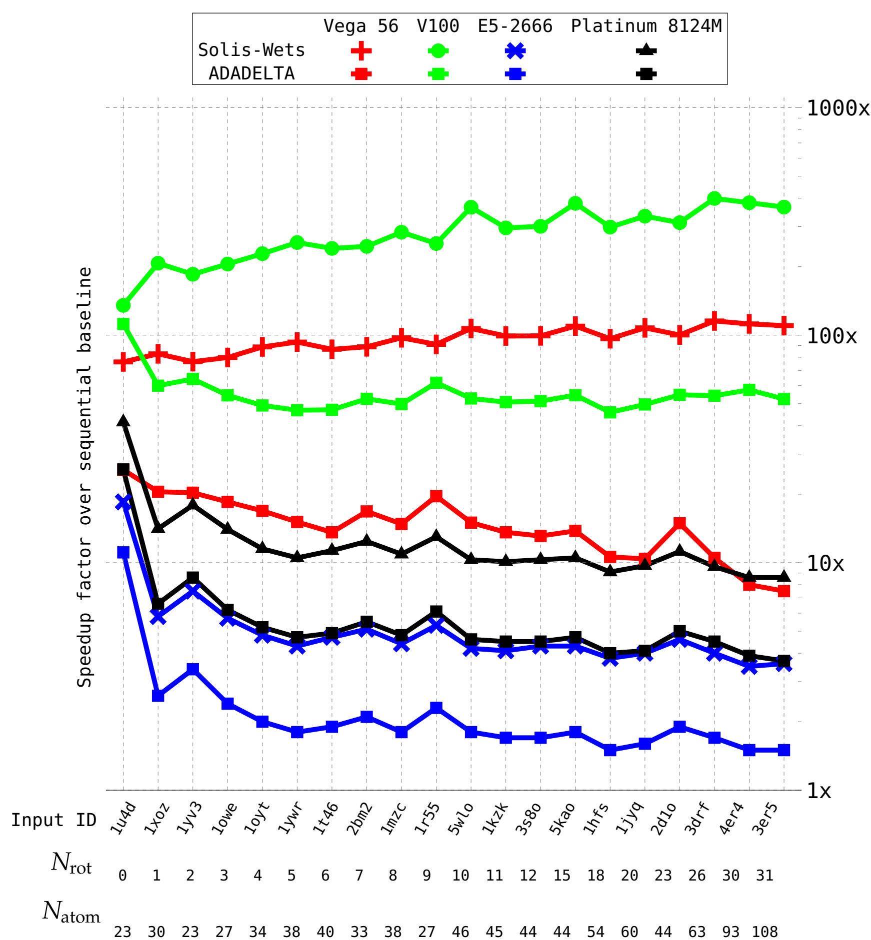


Figure 6.3: Speedups of OCLADock vs. single-threaded AutoDock achieved on selected GPU/CPU devices using work-group sizes of 64/32 work-items, respectively (R = 100 LGA runs, `lsrate` = 100 %).

on any selected device, ADADELTA speedups are significantly *lower* than their corresponding Solis-Wets ones.

The original AutoDock *code is capable* neither *of multi-threading*, nor *the ADADELTA gradient-based search. Thus, we had to use the original* single-threaded *Solis-Wets-based* AutoDock *code as baseline.*

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

- GPUs, tend to *increase* with larger inputs: e.g., {76x, 99x, 110x} on Vega 56, and {135x, 300x, 365x} on V100.
- CPUs, tend to *decrease* with larger inputs: e.g., {41x, 10x, 8x} on Xeon Platinum 8124M, and {19x, 4.3x, 3.6x} on Xeon E5-2666.

Running ADADELTA on any device results in decreasing speedups processing more complex inputs, which is attributed to the following two reasons:

1. The upper-bounds of loops in the GC (Section 6.1.1) – i.e., $N_{\text{pose-rot}}$ (dependent on N_{rot}), N_{atom}, and $N_{\text{intra-contrib}}$ (dependent

on N_{atom}) – are larger for more complex inputs, and in turn, result in longer procesing times.

2. The limited parallelism of the gradient conversion (Section 6.1.2). This procedure is performed in a gene-type basis, and hence, presents three completely independent fine-grained tasks (`Gtrans`, `Grigidrot`, `Grotbond`) that can be distributed between work-items (of a work-group) in different ways. A simple way would be to execute these tasks simultaneously, each by a different work-item. Another way would be to parallelize these tasks with as many work-items as possible, executing only one task at a time. We opted to use a combination of both ways detailed as follows. While `Gtrans` and `Grigidrot` present each a loop with an upper bound of N_{atom}, they must also perform sequences of data-dependent operations, which are not suitable for parallelism. Thus, each of these two tasks was executed by a single work-item. The operations within `Grotbond` are also data-dependent, but are repeated for each rotatable bond. Hence, `Grotbond` is processed by N_{rot} work-items.

6.2.5.3 *Porting onto other GPUs and CPUs*

Porting the data-parallel OCLADock *design from a Vega 56 GPU to other GPUs and CPUs simply requires a re-compilation step.*

Figure 6.4 provides compact statistics for the speedups achieved using the entire dataset – *geometric mean*, *maximum*, *minimum*, and *standard deviation* – and together with Figure 6.3, is used for our analysis on portability across devices.

For GPUs, it is corroborated that the more powerful V100 achieves shorter executions than the Vega 56. For instance, the ratio between V100 and Vega 56 *geometric-mean* speedups is 2.8x (= $\frac{274}{95}$, for Solis-Wets), and 3.9x (= $\frac{55}{14}$, for ADADELTA). Performance improvements can be further analyzed using the speedup profiles in Figure 6.3, where at first glance, V100 profiles appear to be a scaled version of those of the Vega 56. For Solis-Wets, profiles are almost uniformly scaled, with speedup ratios within the range of [2x, 4x] for all inputs. For ADADELTA, profiles show that the portability scaling factors are kept uniform (also within [2x, 4x]) for all inputs smaller than `1hfs`, from which the performance advantage of the V100 over the Vega 56 becomes higher (e.g., 4.3x) than that of Solis-Wets, and reaches $\sim$7x with `4er4` and `3er5`. The performance advantage of the V100 over the Vega 56 comes at a relatively *higher* economic cost, currently by a factor of 20x at street prices (mid 2019). Similar speedups could be achieved much cheaper by running four Vega 56 cards in parallel. This is easily possible in the drug discovery use-case, due to the embarassingly parallel outer loop of the problem.

For CPUs, the design exposes performance improvements for both local-search methods and all inputs. The ratios between *geometric-mean* speedups between the Xeon Platinum 8124M and Xeon E5-2666 are

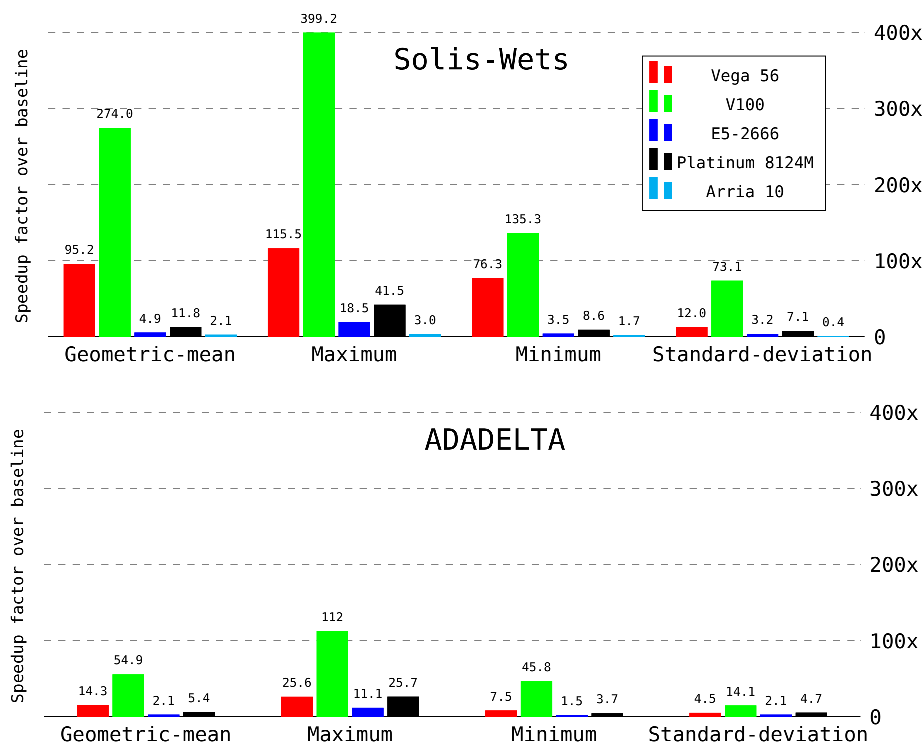


Figure 6.4: Statistics of speedup factors: OCLADock vs. single-threaded AutoDock achieved on all selected devices (R = 100 LGA runs, `lsrate` = 100 %).

2.4x (= $\frac{11.8}{4.9}$, for Solis-Wets) and 2.6x (= $\frac{5.4}{2.1}$, for ADADELTA), being both values superior than the 2.0x expected improvement resulting from the increase of physical CPU cores (= $\frac{36}{18}$, Table 6.1). Moreover, from Figure 6.3, it can be clearly noted that profiles are almost perfectly scaled among devices. This was confirmed by finding that the speedup ratios between both CPUs (calculated for all inputs) are within the [2.2x, 2.6x] range, for both Solis-Wets and ADADELTA. The aforementioned CPU performance improvement of at least 2.2x corresponds to a price-increase factor of ~1.9x, obtained considering the hourly charges of $3.49/h (`c5.18xlarge`) and $1.82/h (`c4.8xlarge`) [10].

Indicated AWS charges comprise only compute capacity for the Frankfurt region.

Finally, in most cases, GPUs achieve higher efficiencies than CPUs when utilizing the same local-search method. For instance, using ADADELTA, *minimum* speedups of {7.5x, 3.7x} on the {Vega 56, Xeon Platinum 8124M} were achieved. A notable (and the only!) exception to this behavior happens when using ADADELTA with `1u4d` (left side in Figure 6.3, and *maximum* bars in Figure 6.4), where the achieved speedup is 25.6x on the Vega 56, and 25.7x on the Xeon Platinum 8124M, respectively. The faster executions of GPUs are attributed to the more suitable mapping of OpenCL elements onto their hardware. On CPUs, however, each work-group is executed by a single CPU core, and thus, *its* work-items are executed serially [87, 148]. The purpose of such serialization is to avoid the excessive synchronization

penalties incurred if work-items within a work-group were executed in parallel, since work-items on CPUs are mapped to operating system (OS) threads instead of the lighter-weight hardware threads used on GPUs.

6.2.5.4 *Porting onto FPGAs*

For a more comprehensive evaluation, the OpenCL *task-parallel* implementation for FPGAs discussed in Chapter 5 was included also in the overall comparison in Figure 6.4. In that regard, the fastest task-parallel configuration – also the largest in terms of required FPGA resources – contains 27 kernels (each processing a *single* OpenCL work-item) connected with OpenCL pipes. In terms of functionality, this is comparable to other implementations running *only* Solis-Wets. This FPGA implementation does not include any gradient method, as its incorporation would require significant architectural changes, i. e., instantiating a kernel for each gradient method, and connecting those additional kernels into the main design via OpenCL pipes.

It was found that the fastest configuration, which includes nine replicas of the local-search (LS) kernel running Solis-Wets, barely fits on the Intel Arria 10 FPGA. Adding the gradient methods would require extra hardware area that could only be freed-up by reducing the number of local-search kernel instances. This would slow down MD on the FPGA even further. For that reason, gradients were not implemented on the design for FPGAs. The only change was the addition of smoothing [117] to the score calculations to improve the quality-of-results. As depicted in Figure 6.4, the FPGA running Solis-Wets achieves speedups with geometric mean, maximum, minimum, and standard deviation of $\sim$\{2.1, 3.0, 1.7, 0.4\}x, respectively.

6.2.6 *Compute-energy efficiency*

The energy consumed during an MD simulation is calculated by numerically integrating the power over time. This is because power values might fluctuate greatly between samples. Here, power profiles are analyzed first since through them it is possible to understand the energy consumption behavior of OCLADock on the selected accelerators.

Power was measured by using software utilities rather than external meters, as this approach allows to consistently sample power consumption on *most* devices, including the remote ones on AWS. Details of the experimental setup for power sampling are provided as follows:

- For the Vega 56, a proprietary tool from AMD was used. For the V100 and Xeon E5-2666, the publicly-available Nvidia System Management Interface [131] and Turbostat [196] were used, respectively.

- On above devices, power was sampled over the entire MD simulation at $T_{\text{sampling}} = 50$ ms. While this sampling rate might be too coarse for shorter `Krnl_LS` executions, it was the highest possible sampling frequency on the Vega 56, and therefore, this rate was employed for other devices to obtain comparable measurements.

- On the Xeon Platinum 8124M, it was not possible to sample power due to the lack of control of *P-states*, i. e., desired performance in CPU frequency. This issue of AWS `c5` instances is documented in [12].

- For the Arria 10, fully placed-and-routed designs were power analyzed using *quartus_pow* as in Section 5.3.5. The reported power draws of ∼30 W were very stable across entire executions and are typical for this kind of PCI Express-attached FPGA board.

Figure 6.5 depicts the power consumption over time on the Vega 56 using the `3s8o` input. For understanding the impact of `Krnl_LS` executions on power profiles, we use the insights provided in Table 6.3. For all inputs, profiles of both Solis-Wets and ADADELTA executions are characterized by transitions between low and high power draws, ranging between 100 W and 220 W. These frequent power swings correspond to the switching between host-side and kernel (a sequence of `Krnl_GA` and `Krnl_LS`) executions.

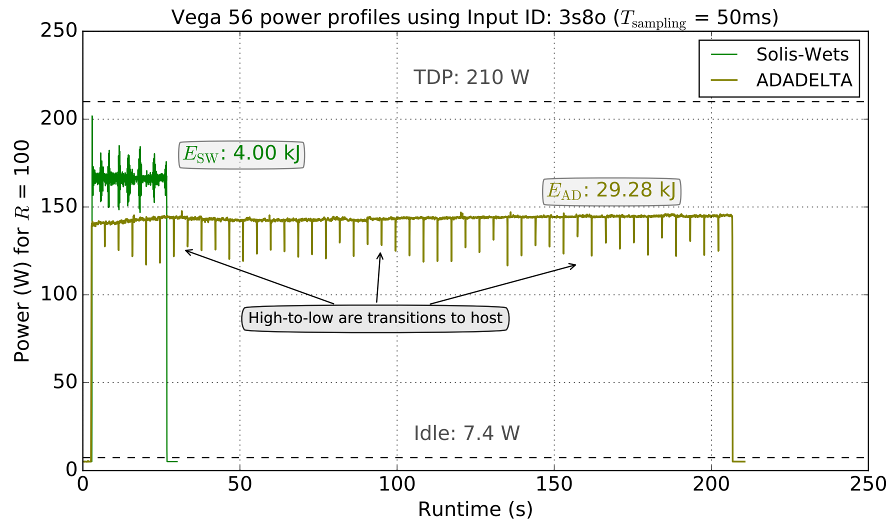


Figure 6.5: Power measurements of OCLADock for Solis-Wets and ADADELTA executions on the Vega 56 GPU (R = 100 LGA runs, `lsrate` = 100 %).

Since Solis-Wets has many more `Krnl_LS` *# Calls*, the frequency of power transitions is also higher in Solis-Wets compared to ADADELTA. Table 6.3 corroborates this, e. g., using `3s8o` as input results in 120 (Solis-Wets) and 46 (ADADELTA) `Krnl_LS` enqueues. In Figure 6.5, it is even possible to count the 46 high-to-low power transitions for the

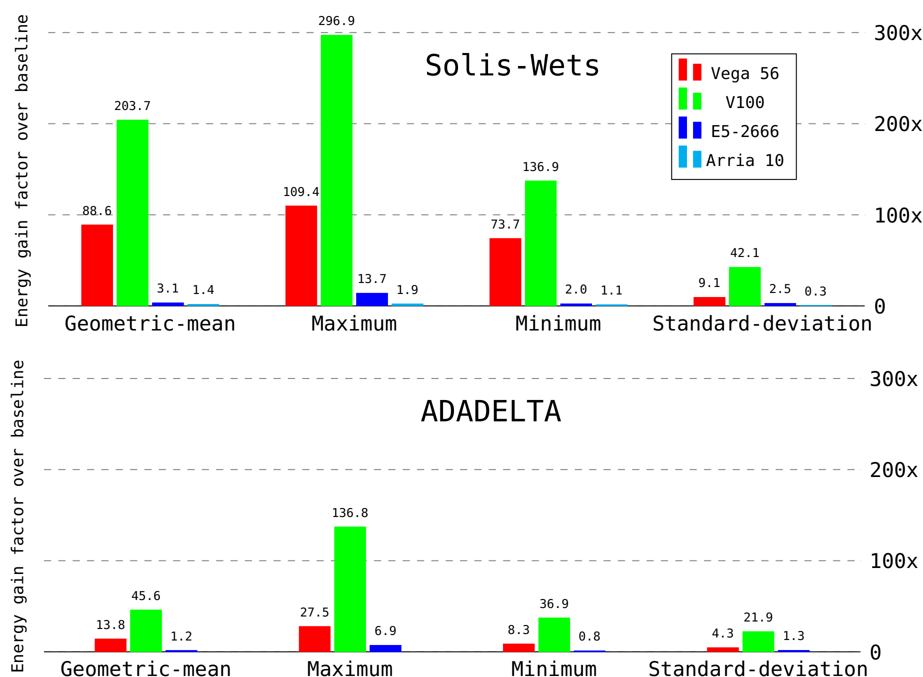


Figure 6.6: Statistics of energy-efficiency gains: OCLADock over single-threaded AutoDock achieved on devices where measuring power was feasible (R = 100 LGA runs, `lsrate` = 100 %).

ADADELTA execution. This is complemented by the fact that power draws are mostly located around ~170 W for Solis-Wets, and ~140 W for ADADELTA, typically during `Krnl_LS` executions.

Even with ADADELTA performing more complex computations for gradients, and thus taking longer to complete, its kernel occupancy drops to 10 % (from 20 % by Solis-Wets in Table 6.3) due to the required serialization and OpenCL `atomic` operations for achieving correct partial derivatives for translational and rigid-body rotational genes (Section 6.1.2). From internal discussions with the vendor, the lower occupancy of `Krnl_LS` implies that some Vega 56 block units are not utilized, and hence, are automatically turned off by the GPU. This would explain the lower power draw of ADADELTA vs. Solis-Wets.

Figure 6.6 shows compact statistics of the energy-efficiency gains computed with respect to the baseline (only Solis-Wets) for the entire dataset. As already mentioned, at the time of writing, power sampling was not supported on the Xeon Platinum 8124M, and thus, not shown.

Despite the fact that power draws of up to ~300 W were observed on the V100 (i. e., higher than for any of the other devices), due to its much shorter runtimes, it yields higher energy gain factors over the baseline: ~270x (Solis-Wets) and ~137x (ADADELTA). Also, since the MD quality of both Steepest Descent and FIRE was inferior to that of ADADELTA (Section 6.2.2), their corresponding efficiencies are not reported. Nevertheless, in this aspect, all gradient methods are

comparable, achieving e. g., for 3er5 on the Vega 56: ∼11.9x (Steepest Descent), ∼11.7x (FIRE), and ∼11x (ADADELTA). Furthermore, while the estimated power draw on the FPGA is the lowest (∼30 W), it yields the lowest energy efficiencies (maximum ∼1.9x) due to its much longer execution times. As already described in Section 6.2.5, the FPGA version only implements Solis-Wets.

Although gain factors in terms of speedup (Figure 6.4) and energy (Figure 6.6) show a significant advantage of Solis-Wets over ADADELTA, the longer execution-times and higher energy-consumptions of ADADELTA result in better-quality dockings in *many* cases, as listed in Table 6.2. This is a Solis-Wets vs. ADADELTA trade-off, where for inputs with few rotatable bonds ($N_{rot} < 8$), Solis-Wets could lead to sufficiently good results, and thus, spending more computing resources running ADADELTA is not worth it. However, for larger inputs ($N_{rot} > 8$), ADADELTA is likely to find *better* solutions, even in cases where Solis-Wets is simply not able to at all, e. g., for inputs in Table 6.2 where $N_{rot} > 11$.

7

USING OCLADOCK FOR COMPETITIVE DRUG DISCOVERY

This chapter details own modifications applied to OCLADock for dealing with *macrocyclic* molecules. These additional capabilities were used to participate in the *Grand Challenge* [34] molecular prediction competition. Details on how OCLADock was used for docking macrocyclic molecules have been previously published in [44, 166].

7.1 THE CHALLENGE OF DOCKING MACROCYCLIC MOLECULAR STRUCTURES

The *Grand Challenge* (GC) is a competition organized by the Drug Design Data Resource (D3R) project, which aims to advance the technology of computer-aided drug discovery through the interchange of protein-ligand datasets and workflows [34]. Since 2015, D3R has been organizing yearly editions of GC, which are community-wide *blinded* prediction competitions.

Blinded prediction refers to the exploration over an unknown, typically large, surface of protein-ligand interaction.

In its 4th edition (GC4), the challenge is to predict the binding pose, affinity ranking, and free energy of ligands against two different protein targets: Beta secretase 1 (*BACE*), and Cathepsin S (*CatS*). As part of our collaboration with The Scripps Research Institute (TSRI), an extended version of OCLADock was employed to predict the interactions of different ligands against *BACE*. This enzyme is essential for the generation of β-amyloid peptide in neural tissue [199], a component of amyloid plaques believed to be critical in the development of the Alzheimer's disease [152].

Amyloid plaques are clumps of protein fragments called β-amyloid, which are toxic to neurons as once they bind to each other, plaques destruct neuron connections, i. e., sinapses [205].

For this competition, D3R provided a dataset comprising small molecule inhibitors along with previously undisclosed crystallographic structures. Specifically, for each prediction criterion, there were provided a number of *BACE* inhibitors: affinity (154), pose (20), and free energy (34).

7.1.1 *Why is this actually a challenge?*

An especially large number of macrocyclic ligands are part (∼85 %) of the entire dataset provided at GC4. According to Yudin [219], macrocyclic molecules – from now on simply referred to as *macrocycles*

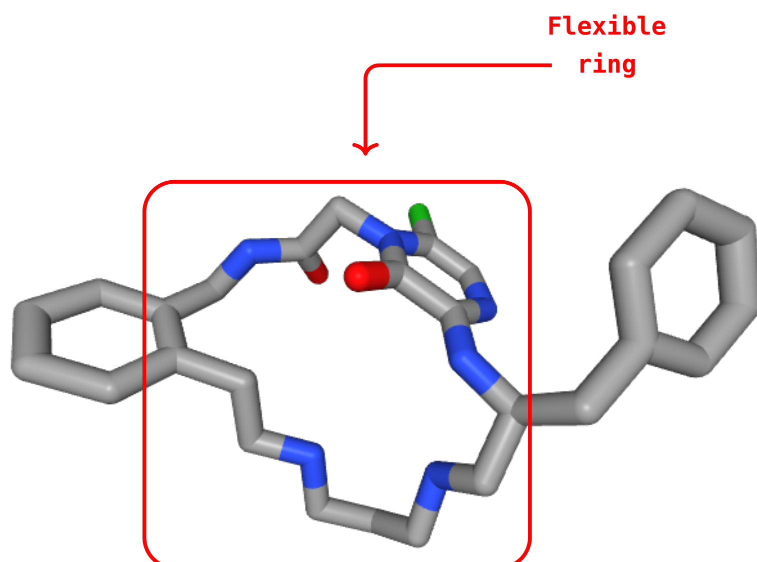


Figure 7.1: Three-dimensional representation of a macrocycle example: `1nm6` ($C_{27}H_{33}ClN_6O_2$). Atoms are carbon (gray), hydrogen (not shown), chlorine (green), nitrogen (blue), and oxygen (red). The number of atoms in the ring is $N_{atom}^{ring} = 19$, and that of active rotatable bonds is $N_{rot}^{active} = 12$. Figure was obtained from [19].

– are large and contain within their structure flexible rings, i.e., a sequence of rotatable bonds forming a closed cycle (Figure 7.1).

As mentioned in Section 2.2.1, a conformation refers to a change in the molecular structure that happens only when some angles between bonds are altered.

Based on internal communications with TSRI, these molecules can adopt very different conformations, but the torsion tree-like representation typically used in MD software is unable to describe such conformations because the angles of rotatable bonds depend on each other in order to maintain a cyclic structure. Despite the promising advances in macrocycle docking, performing adequate conformational search for macrocycles is still a challenge [8].

As reported by Allen, Dokholyan, and Bowers [7], several techniques (based on e.g., *Monte Carlo*, molecular dynamics, no energy-function, genetic algorithm, etc) have been employed for conformational search. These techniques differ primarily in the scoring function used to treat the macrocyclic ligand, as well as the flexibility allowed in the ligand-protein pocket. Moreover, most MD software require the *pre-generation* of macrocycle conformations prior to the simulation due to their limited capacity to sample such conformations.

Originally, AutoDock was not able to manage the bond flexibility due to the cyclic ligands. Basically, the rotation of an intra-cyclic bond would result in a distorsion of the ligand structure, and hence, cyclic portions were treated as rigid. In order to overcome this limitation, Forli and Botta [51] developed a protocol that converts the cyclic ligand into its corresponding *acyclic* form by breaking a ring bond, and then docking it as fully flexible. During the MD simulation, AutoDock is able

restore the original cyclic structure by introducing a *new* potential term into the scoring function.

The additional potential term in the AutoDock scoring function models the broken bond, and allows AutoDock to bring the associated atoms back together during docking. Closing the distance between the atoms associated with the broken bond consists of assigning them a suitable score contribution, i. e., a very large positive energy (unfavorable) to conformations where the atoms in question are far apart. With this technique, only chemically relevant structures have low energies (favorable), and thus, their genotypes are likely to result as predominant solutions out of the LGA. We extended OCLADock to also perform these computations.

7.2 HANDLING MACROCYCLES WITH OCLADOCK

7.2.1 *Macrocycle-oriented scoring-function terms*

The basic idea for defining such additional potential terms is that of re-using already existing terms, and parametrizing them with specific coefficients corresponding to the so-called *glue* or G atom types, i. e., those associated with the *modeled* broken bond. For that purpose, the scoring function (SF) defined in Equation 2.3 is adapted as proposed in [51].

In Equation 2.3, the contributing *van der Waals* and hydrogen bonding terms are partly determined by the coefficient pairs (A_{ij}, B_{ij}) and (C_{ij}, D_{ij}), respectively. In this context, if a given pair of ligand atoms has a *van der Waals* interaction, then their hydrogen bonding term is zero, and hence, both of their corresponding C and D coefficients are zero as well. For handling macrocycles, an additional G coefficient is introduced for setting the score contribution equal to zero for all ligand atoms, except for those associated with the broken bond, i. e., the atomic pair that is intended to be brought very close together (ideally to an interatomic distance r equal to zero). This additional term SF_G – characterized by its G coefficient – has a linear dependency with respect to the interatomic distance r, and hence its gradient g_G is constant:

Subindexes i *and* j *refer to atoms.*

$$\mathrm{SF}_G = G \cdot r \tag{7.1}$$

$$g_G = G \tag{7.2}$$

such that

$$G = \begin{cases} 50 & \text{for the two atoms associated with the broken bond} \\ 0 & \text{otherwise} \end{cases}$$

7.2.2 *Macrocycle-oriented development*

Besides the scoring function, other aspects have to be considered for supporting macrocycles' docking in OCLADock. These aspects are the ligand characteristics depending on atomic types (e. g., *van der Waals* radii, solvation volume, etc), as well as the incorporation of this technique into the multi-level parallelization described in Chapter 4.

The efforts towards a successful macrocycles' support were spent in introducing the SF_G term into the OpenCL structure of the scoring function SF, as well as in adapting the treatment of macrocyclic atoms into the overall OpenCL design, both in host and device sides. Such development was carried in the following three steps:

1. Addition of *G* atom types into the ligand *.pdbqt* input specification. These are two new non-standard atom types denoted as *CG* and *G0*, which are assigned to the atoms associated with the broken bond (Figure 7.2):
   - *CG* is a copy of the standard carbon atom type *C*. Atoms of type *CG* adopt the same values for equivalent parameters of *C* atoms: e. g., *van der Waals* radii, solvation volume, etc.
   - *G0* is an invisible type, i. e., all its atomic parameters have a value equal to zero.
2. Modification of intramolecular scoring-function parameters for specific atom pairs. The macrocyclic ligand contains two *CG* atoms forming an intramolecular pair for which scoring-function terms must be calculated. Due to the fact that atoms of both *CG* and *C* types possess atomic parameters with the same value, then the *van der Waals* interaction of the *CG* – *CG* pair would be the same as for any other *C* – *C* pair (in the macrocycle). However, in order to bring the *CG* – *CG* atomic pair back together, their *van der Waals* coefficients (A, B) must be zero. This modification strictly affects the *CG* – *CG* pair, but not e. g., a *CG* – *C* pair.
3. Finally, a new functional form for the *CG* – *G0* must be added. Specifically for this pair of atoms, their score term f_G should be linearly dependent on their corresponding interatomic distance, as specified in Equation 7.1.

7.2.3 *Experimental evaluation*

For the GC4 challenge, OCLADock was executed for a large number of LGA runs using the *best* local-search (LS) methods from Chapter 6: Solis-Wets and ADADELTA. The configuration of our experiments, organized in four batches, is listed in Table 7.1. In these, the ADADELTA method performed significantly better than Solis-Wets for the same

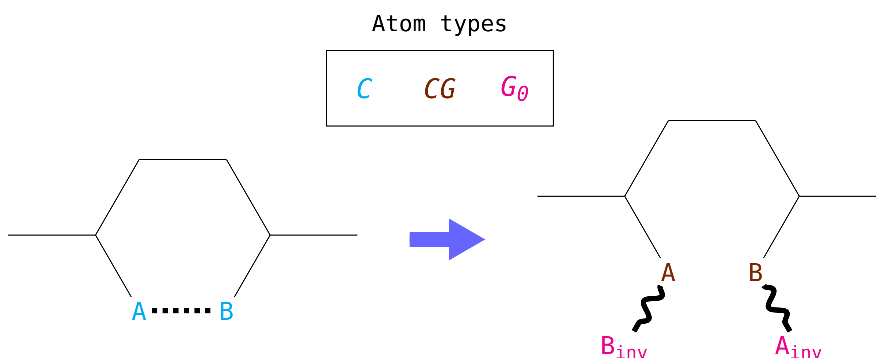


Figure 7.2: Left: identification of the ring bond to be broken: A – B. Right: introduction of the so-called *invisible* atoms A_{inv} and B_{inv}, used for the ring-closure procedure during docking. Both sides show the assignment of non-standard atomic types (*CG*, *G0*) to atoms in the broken bond.

Table 7.1: Experiments on a set of 20 ligands performed for the GC4 blind prediction competition. The best values within each case are colored.

Experiment ID	Local Search method	# LGA runs (R)	$N^{MAX}_{score\text{-}evals}$ (millions)	Median RMSD (Å)
SW1	Solis-Wets	100	10	10.1
AD1	ADADELTA	100	10	1.2
SW2	Solis-Wets	200	32	9.1
AD2	ADADELTA	200	32	1.0

number of both LGA runs (R) and score evaluations ($N^{MAX}_{score\text{-}evals}$). The median RMSD calculated for 20 ligands shows the clear superiority of ADADELTA. Similarly as in previous chapters, an RMSD cutoff of 2.0 Å was used to classify a prediction as correct.

GC4 results using OCLADock *correspond to the submission made by TSRI labelled as* Stefano Forli.

Compared to other *GC4* competitors, the MD predictions obtained with OCLADock submitted by TSRI were among the top 25 % in terms of pose prediction [32, 33], and affinity ranking [30, 31]. All competitors disclosed their numeric predictions along with the corresponding protocol. These protocols include useful information on how simulations were undertaken, e. g., methods and parameters used for ligand preparation and pose prediction, the employed MD engine, etc. As this competition seeks for the *best* results and methodologies from the MD perspective, execution runtimes were neither requested nor disclosed, and thus, a performance-wise comparison between OCLADock and other software used in the competition is not possible. As a reference, it took ∼4 hours to complete the AD1 experiment (20 ligands) running OCLADock on a GTX1080 GPU. According to TSRI, OCLADock running ADADELTA achieves average speedups of ∼36x on a GTX1080 GPU.

On the other hand, using instead AutoDock to reach the same level of MD quality for the AD1 experiment would have required ∼144 hours.

Considering all evaluation criteria, the competition results show that there is no *single winner* in this competition, but instead an overall advancement in the shared knowledge for the scientific community in this field.

8

CONCLUDING REMARKS

8.1 SUMMARY

This thesis describes and evaluates OCLADock: an OpenCL-based parallelization of the AutoDock MD software. An efficient parallelization of AutoDock on GPUs and CPUs was achieved following a data-based parallelization at two levels: first, each individual of a genetic population was processed by an OpenCL work-group; second, fine-grained tasks for each individual were processed by OpenCL work-items. However, a key result of this study is the lack of performance portability of OpenCL when porting onto FPGAs. The data-parallel approach that allows initial speedups of more than 50x on GPUs leads to a slowdown of three orders of magnitude when used on FPGAs.

In order to improve the performance on FPGAs, a set of design and optimization steps based on a task-parallel pipeline architecture was followed. The architecture is composed of single work-item kernels communicated through OpenCL pipes. This study explored different architectural choices, such as kernel replication and channel arbitration, to improve the efficiency of the encountered *multiple-producers to single-consumer* datapaths in AutoDock.

Introducing gradient-based local search into the aforementioned data-based OpenCL implementation of AutoDock has resulted in stronger molecular poses than when using the legacy Solis-Wets method. The experiments show that ADADELTA outperforms Solis-Wets in terms of MD quality, as well as Steepest-Descent and FIRE in both quality and speed, still yielding comparable compute-energy savings as the latter two gradient-based methods.

Further experiments on the *best* local-search methods found, i. e., Solis-Wets and ADADELTA, show that obtaining stronger poses requires significantly more computational effort, which in turn, is translated into slower executions for ADADELTA compared to Solis-Wets. Despite that, the data-based parallelization has yielded up to ∼399x (Solis-Wets) and ∼112x (ADADELTA) speedups with respect to the original single-threaded AutoDock (running Solis-Wets) on modern V100 GPUs.

Regarding the achieved compute-energy savings, the V100 GPU was the *most* efficient device among the platforms chosen for eval-

uation, achieving maximum gains of ∼297x (Solis-Wets) and ∼137x (ADADELTA). With the energy consumption of all gradient methods being comparable, the higher quality of ADADELTA dockings makes it the best choice, even compared to the fastest Solis-Wets method, especially from ligand inputs that contain more than eight rotatable bonds. On an Arria 10 FPGA, it was found that even for the *simpler* Solis-Wets method, both maximum speedups (∼3.0x) and energy efficiency gains (∼1.9x) are the lowest among the selected devices.

Finally, the functionality of OCLADock was extended to support macrocycles, i. e., molecules that contain ring structures that are difficult to dock effectively due to the complex rotation dependency of their ring members. By incorporating the methodology proposed in [51] into OCLADock, successful dockings were achieved (i. e., the resulting spatial deviation or RMSD was below than 2.0 Å). Our results were ranked among the top 25 % in the drug discovery GC4 competition [34].

8.2 LESSONS LEARNED

The research problems proposed in Section 1.2, were addressed throughout this thesis. The following set of questions-and-answers clarify the lessons learned during this work.

8.2.1 *OpenCL for FPGAs*

Q1: Based on the AutoDock parallelization developed in this work, how promising is OpenCL for FPGAs?

The overall OpenCL-based parallelization of AutoDock started focusing on FPGAs as accelerator targets. Back then, the first OpenCL-to-FPGA development tool employed was SDAccel 2015.4 [210]. Due to the relatively inmature state of that tool, several problems were encountered during system setup (e. g., FPGA board and respective driver bring-up) that were resolved after several weeks. After some upgrades introduced in SDAccel 2016.1, a data-parallel OCLADock running correctly on the FPGA was achieved. As already reported in Chapter 4, this version was however three orders of magnitude slower than AutoDock running on a single CPU code.

With regards to code productivity, most of the application development and functional verification were performed on tools used for Intel CPUs [84] and AMD GPUs [28]. Although SDAccel provides users with CPU/GPU-like functionality verification in the form of *SW-/ HW-emulation* modes, both of these required excessive run times (more than 1 day) that were impractical for verifying a typical docking job consisting of 100 LGA runs, each run comprising 2.5 million score evaluations.

In many cases, as no significant progress in *SW-emulation* was observed (e. g., after ∼10 hours), SDAccel was utilized for emulating dockings with reduced settings (e. g.: 10 LGA runs, 40 000 score evaluations), or even for just invoking the underlying Vivado suite [200] to build actual FPGA hardware bitstreams. From a practical standpoint, skipping *HW-emulation* and building FPGA binaries directly resulted in a more productive development, since the overall build process took ∼8 hours, and showed the true design-behavior on hardware.

Over the years 2017 - 2019, it was found that SDAccel has been enhanced significantly in several aspects, and thus, it has overcome issues encountered before (e. g., bug fixes, lack of documentation, etc). Although many enhancements given by vendor-specific directives (e. g., loop pipelining, RAM partitioning, etc) look beneficial for kernel acceleration based on OpenCL/C/C++, these could not be fully exploited, even in our more efficient task-parallel OpenCL design (Chapter 5). The reason for this was the lack of support of *non-blocking* OpenCL pipes in SDAccel 2018.2, as reported in [209].

Regarding OpenCL-to-FPGA tools, in general, that of Intel provided an smoother development experience compared to that of Xilinx. With Intel tools, we could mostly focus on design exploration and optimization instead of dealing with bugs or lack of support of some OpenCL constructs, as experienced with Xilinx SDAccel. However, one of the major issues observed when executing OCLADock on more recent FPGA platforms and tools (Stratix, v18.1) is the *decreasing* performance with respect to older ones (Gidel Proc10A, v16.0). Although the latest introduction of compiler directives promising higher speedups (e. g., multiple calling sites for pipes in a single kernel, efficient kernel replication, etc), the compiler technology seems yet to be unable to *efficiently* map onto FPGA logic, those constructs whose behavior is determined at runtime, e. g., loops with variable loop bounds and pipes.

8.2.2 *OpenCL for GPUs and CPUs*

Q2: How was the experience of using OpenCL on GPUs and CPUs?

Porting OCLADock to GPUs and CPUs was usually smooth, allowing functional verification in a reasonable amount of time (e. g., four hours). While in most development phases, OpenCL allowed functional and performance portability, there were cases where features developed and succesfully tested on GPUs did not work right away on CPUs after re-compilation.

Particularly, as described in Chapter 6, gradients-based local-search methods were incorporated into the LGA flow. For reducing the duration of testing cycles for time-consuming LGA runs, all the corresponding tuning was performed on GPUs. After successfully testing

OCLADock on four different mid-end and high-end GPU cards from Nvidia and AMD, it was surprising to find that, when the program was configured to execute any gradient method, it suffered from *segmentation faults* on all CPU platforms tested, even when using different OpenCL drivers such as POCL [86] and Intel [84]. This issue was caused by program variables carrying information of:

- Interatomic distances required to calculate grid-map indexes for the intermolecular interaction. These *must* be declared as `unsigned int` instead of `int` (which worked flawlessly on GPUs), otherwise out-of-bound accesses on grid maps are attempted.

- Gradient variables must be *explicitly* initialized for CPUs instead of leaving this task to the OpenCL runtime (as originally on GPUs), otherwise wrong values of variables might be derived from gradients, and utilized as a termination condition (e. g., negative values) of while loops, leading to hangs.

Despite the fact that solving the aforementioned issues is not complicated, it took ∼3 days to locate the problematic code regions. Surprisingly, OpenCL-oriented utilities usually employed on GPUs such as CodeXL [28], or even the architecture-agnostic Oclgrind [136], were *not* able to even detect such problem. Conversely, Valgrind [121] – configured to run *Memcheck* – helped to effectively detect the memory-related errors that lead to crashes and unpredictable behavior.

Beyond these issues, most of the OpenCL development on GPUs and CPUs had no complications due to tool unstability. For high-end GPU cards, we found that the support of OpenCL 1.2 is fairly mature, and therefore no major complications were encountered.

8.2.3 *OpenCL beyond datacenters*

Q3: How efficient is OpenCL in other domains besides datacenters?

Although the focus of this thesis is the acceleration of AutoDock on high-end devices using OpenCL, part of a further case study (not reported in this thesis) – on heterogeneous systems for *autonomous driving* [177, 178] – involved a code deployment on *embedded* devices, as well as an analysis of the programming productivity of OpenCL compared to other models such as OpenMP and CUDA. This additional experience gave complementary and still relevant OpenCL insights that are contextualized as follows.

For GPUs and CPUs, the OpenCL support in datacenters is currently more extensive than in the embedded domain. Particularly, all high-end platforms used in this thesis had fully-featured proprietary drivers (OpenCL 2.0 for AMD and Intel, and OpenCL 1.2 for Nvidia), whereas for Nvidia embedded platforms used in [177] (Jetson TX2 [130] and

AGX Xavier [129]), the usage of OpenCL 1.2 code was only possible with POCL [86], specifically through its experimental – and limited in terms of functionality – LLVM Nvidia PTX backend [150] for code generation. From the study, it was found that, while in general terms the performance that can be achieved with OpenCL, OpenMP, and CUDA was comparable, the programming productivity by using OpenCL was *behind* of that by using OpenMP or CUDA.

For embedded FPGAs, the OpenCL support is still *inmature*, and many steps behind of that for datacenters. For instance, a Xilinx ZCU102 platform [214] had proprietary support limited to OpenCL 1.2, while enabling some OpenCL 2.0 features such as pipes, though. Analogously to SDAccel 2018.2 used for high-end devices, SDSoC 2018.2 [211] used for the ZCU102 had severe limitations in emulation. To avoid tool crashes from these limitations, experiments had to be run with a minimalistic configuration, i. e., considerably reducing both the number of time-consuming iterations, as well as the size of input data.

8.3 REMAINING RESEARCH AND ENGINEERING CHALLENGES

8.3.1 *Extending functionality of OCLADock*

As described in this thesis, the fast executions and high algorithmic quality of OCLADock are respectively due to the OpenCL-based parallelization, and the incorporated ADADELTA gradient-based local-search method. While at this point of development, it could be claimed that the execution performance of OCLADock is *satisfying*, this might not hold true for *more* complex scenarios, e. g., those requiring:

- Higher-order gradient-based methods, e. g., BFGS [123], which *might* achieve even higher MD quality than ADADELTA (Chapter 6).
- LGA variants, e. g., that in AutoDockFR [154] implementing an *adaptive* termination criterion, rather than the fixed one in AutoDock and OCLADock, being both based on maximum number of score evaluations or generations.

In such cases, while the LGA parallelization described in this thesis would remain mostly valid, performance penalties due to more-complex methods have to be taken into account. Examples of such penalties include larger data-transfer between OpenCL host and device, more-frequent accesses to device memory from OpenCL kernels, etc.

8.3.2 *Enhancing performance of OCLADock on FPGAs*

While the execution performance of OCLADock on GPUs and CPUs seems to be *good* for typical requirements in drug discovery, the potentially larger savings in compute-energy consumptions (kJ) of FPGAs, still make them an interesting alternative among general-purpose accelerators.

As mentioned in Section 8.2.1, OpenCL support for FPGAs is under continuous development. With respect to the task-based parallelization for FPGAs described in Chapter 5, there are two main remaining challenges:

- Further research on *non-blocking* OpenCL pipes and their mapping onto FPGA fabric needs to be carried out. Most studies focus on *blocking* pipes, typically invoked from within code regions executed a constant number of times known at compilation time. The latter scenarios are favorable for OpenCL-to-FPGA compilers as they allow further static optimizations. However, these are often not applicable for use in AutoDock, which contains many *irregular* execution patterns derived from local-search executions.

- Support for alternative gradient-based methods needs to be provided. As previously mentioned, obtaining gradients involves a complex procedure, and hence, it would require a significant extra amount of FPGA hardware resources. In this work, it was not possible to fit such design on the chosen FPGA. However, doing so might be feasible targeting larger next-generation devices.

Addressing the above points will reduce the gap between FPGAs and {GPUs, CPUs} when using OCLADock, and thus, could make FPGAs suitable for realistic MD usages, such as the blind docking competition described in Chapter 7.

A

KEY IMPLEMENTATION DIFFERENCES COMPARED TO ORIGINAL AUTODOCK CODE

The OCLADock implementation involves many modifications to the original AutoDock functionality in order to better exploit parallel processing, and the execution performance, without negatively affecting the MD quality. These modifications are:

1. **Arithmetic precision**. Scoring and search calculations in AutoDock are performed using double-precision floating point (64 bits). As previous studies [144, 146, 233] suggest that performing MD computations with reasonably lower precision does not lead to deterioration in terms of best score, spatial deviation, and clustering size, then those calculations in OCLADock were implemented in single-precision floating point (32 bits).

2. **Arrangement of data structures**. Structures were re-arranged for better parallel processing of rotation and pairwise interaction. Ligand flexibility can be described by two rotation types. First, a general one that considers the ligand as a rigid body, and a second type due to rotatable bonds for which a tree-like structure is constructed. AutoDock serially traverses the nodes of such flexibility tree in a recursive manner. Although doing so is feasible on OpenCL devices capable of enqueuing kernels independently from the host (a feature known as *device-side enqueuing*), this would not be portable to devices with more limited language support, i. e., prior to OpenCL 2.0 [62].

   To tackle this in OCLADock, the *recursion*-based approach was translated into an *iterative*-based one, which was achieved by transforming the flexibility tree into an array-like rotation list. This list is composed of *integer*-type items (32 bits) with fields detailed in Table A.1. Similarly, for the pairwise interaction, instead of having a GPU (likely inefficiently) traversing the tree, the host defines another array-like list, containing *intramolecular-contributing* atomic pairs.

3. **Selection scheme**. Regarding the criterion to choose which individuals will reproduce in the genetic algorithm, the original *proportional selection* was replaced with *binary tournament* (default

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

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

rate: 60 %). In proportional selection, individuals with better-than-average scores receive proportionally more offspring [116]. One of its major defficiencies is that if the initial population contains one or two energetically-stronger individuals, then these would dominate the rest, and consequently, would prevent the population from exploring other potentially better solutions by escaping from a local optimum [155, 224].

On the other hand, in tournament selection, sets of individuals are randomly selected from the entire population. The highest-scoring individual in the set is the tournament winner, and therefore selected for crossover. This scheme also suffers from diversity loss, which happens with large set sizes. But the implementation in OCLADock minimizes this possibility as the *minimal* tournament set size is chosen (i. e., two, hence the binary denomination). Moreover, the major advantage of tournament selection is the low computational effort, especially if implemented in parallel [155], which according the previous studies [144, 146] results in faster executions than those of proportional selection.

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

B

COMPARING PERFORMANCE AGAINST OTHER PARALLELIZED DOCKING SOFTWARE

The goal of finding the most *efficient* MD tool on given molecular targets has motivated several studies [70, 216], and has been exemplified as well in Chapter 7, where the spatial deviation or RMSD was used as the *most* relevant metric for comparison (lower is better). However, from the HPC perspective, it would be also interesting to know which parallelized MD tool is the most efficient in terms of processing time. While processing times have been used as a comparison metric for benchmarking some single-threaded [92] and parallelized [225] MD codes, there are some caveats when using processing times for MD benchmarking, even when chemical inputs requiring similar computational effort are taken into account.

As discussed in Chapter 2, while most MD programs execute a *scoring function* and a *search method*, they are mostly differentiated by their particular score and search implementations. Such executions are typically controlled by user-defined parameters that, in turn, influence the processing times. The fact that there is *no* direct compute-wise or complexity-wise equivalence between – score and search – algorithms running at the core of different MD programs, makes it very difficult to compare such programs when using just processing times as a benchmark metric. For instance, when comparing the following two:

- Vina: empirical scoring, with an Iterated Local Search (ILS) search [195].
- AutoDock: physics-based scoring, with an LGA search [116].

Typically for Vina, its ILS runs are also referred to as Monte Carlo runs.

Vina involves much simpler score calculations than AutoDock, while its search utilizes *second-order* gradients. Although default parameter values are suggested (by the respective developers) for the score and search of both programs, these control very different algorithms, and thus, cannot be used for setting a *comparable* benchmark in terms of processing times.

Despite these difficulties, here we provide a comparison of some parallelized MD tools in terms of processing times and compute energy associated. For this *modest* benchmark, two programs were selected to be compared against OCLADock:

Table B.1: Configuration of benchmarked MD codes.

MD code	Scoring function	Search method	Termination criteria # runs (R)/$N^{\text{MAX}}_{\text{score-evals}}$	Target accelerator
OCLADock	Force field	LGA/ Solis-Wets	100 (LGA)/ 2 500 000	GPU/CPU (OpenCL)
Pechan and Fehér [144]	Force field	LGA/ Solis-Wets	100 (LGA)/ 2 500 000	GPU (CUDA 9.0)
Vina [195]	Empirical	ILS/ BFGS	100 (*Monte Carlo*)/ -	CPU (C++)

- Pechan and Fehér [144]: a CUDA implementation of AutoDock. As stated in Chapter 3, it is the only related work that is truly comparable to ours.
- Vina [195]: a very popular multi-threaded program for CPUs. Similarly as AutoDock, it was developed by TSRI.

Due to the implementation differences in the benchmarked MD codes, the results provided here should be taken only as singular samples rather than conclusive.

While the values chosen for the user-defined configuration parameters in Table B.1 might be arguable, especially when comparing the termination criteria of Vina vs. *other* codes, they are still *realistic*. Furthermore, although both LGA and *Monte Carlo* runs represent the outermost loops in all programs, setting the maximum number of runs $R = 100$ in all selected MD tools does not truly ensure a completely fair experimental setup due to the aforementioned implementations differences.

Table B.2 presents the geometrically averaged results of a re-docking experiment using five inputs (`1u4d`, `1owe`, `5wlo`, `4er4`, `3er5`) with low, medium, and high complexity in terms of number of atoms and torsions (Table 6.2). Results are organized according to the accelerator devices utilized.

While Pechan and Fehér [144] and Vina [195] target *exclusively* and respectively GPUs and CPUs, OCLADock can be used for both platforms. The executions of both OCLADock and Pechan and Fehér [144] use the Solis-Wets local search. Moreover, for the work of Pechan and Fehér [144] developed in 2012, the now-deprecated APIs (e. g., `cudaThreadSynchronize()`) were replaced by their updated versions (e. g., `cudaDeviceSynchronize()`). For Vina, the program was compiled for release mode (i. e., using the `-O3` flag).

For the GPU case, in average, OCLADock results in *faster* dockings and *lower* compute-energy consumptions than that of Pechan and Fehér [144]. The extremely large (unfavorable) RMSDs (e. g., 38.8 Å) obtained using [144] are due to the fact that this program utilizes the same ligand input as reference for the RMSD calculation. However, for re-docking experiments this is *not correct* because the ligand input is typically a structure with randomized conformations, and

Table B.2: Average results of MD codes benchmarking. Accelerator devices were previously utilized in Chapter 6.

Device name	MD code	RMSD (Å)	Processing time (s)	Compute Energy (kJ)
V100 GPU	OCLADock	2.3	12.0	2.1
	Pechan and Fehér [144]	38.8	24.4	2.5
E5-2666 18-core CPU	OCLADock	2.1	612.9	131.3
	Vina [195]	2.2	127.0	27.3

thus, it *must not* be used as a reference structure. On the other hand, OCLADock correctly calculates the RMSD using as a reference pose the one obtained via x-ray crystallography (i. e., the experimental pose) instead.

For the CPU case, while OCLADock slightly outperforms Vina in terms of averaged RMSD (lower is better), Vina is ∼4.8x faster, and consumes ∼4.8x less compute-energy than OCLADock. The above superiority of Vina can be attributed to the following. As described in Section 3.2.5, Vina implements an empirical scoring function that is computationally less-intensive than the physics-based one in OCLADock. Moreover, Vina adapts the number of iterative steps depending on the search success, and uses second-order gradients such as BFGS. Thus, Vina *might* require fewer global iterations during search than OCLADock (here running Solis-Wets) to achieve a given quality-of-results.

As indicated in Chapter 3, BFGS *stands for Broyden-Fletcher-Goldfarb-Shanno, the creators of such minimization method.*

Nevertheless, the comparison between OCLADock and Vina presented here is simply a single highlight because the termination criteria selected (R = 100 LGA runs and $N^{\mathrm{MAX}}_{\text{score-evals}} = 2\,500\,000$ for OCLADock, and R = 100 *Monte Carlo* runs for Vina) are not equivalent, and thus, could change drastically if termination values for any MD code were chosen otherwise, e. g., R = 500 *Monte Carlo* runs for Vina. Moreover, both OCLADock and Vina exploit fully (i. e., reach 100 % of utilization) all 18 cores of the Xeon E5-2666 CPU during actual MD computation.

C

MEMORY REQUIREMENTS

In order to guarantee that the relatively limited memory capacity of a GPU card – compared to most multi-core CPU servers – does not negatively impact the practical usage of OCLADock, an analysis of the memory size required to hold the processing data was performed. Table C.1 defines the upper limits of MD parameters in OCLADock, i. e., the maximum number of elements for the following data: ligand atomic types, ligand atoms, rotatable bonds, pairwise contributors, rotations, population size, LGA runs, and grid points. Although by imposing these limits the capabilities of OCLADock *might* be constrained, they prevent data allocation beyond the typical memory capacity of most consumer GPU cards (some few GBs).

The first concern is the memory occupied by *constant* data, which is composed of relatively large look-up tables used in different MD calculations. Table C.2 lists all constant arrays utilized when OCLADock is configured to run Solis-Wets local-search method. These are conveniently grouped into the {A, B, C, D, E} structs, and passed into GPU memory as OpenCL buffer objects. Depending on the assigned OpenCL memory-space qualifier, a struct can be placed either in the GPU on-board memory (`__global const`), or in the GPU on-chip memory (`__constant`). Ideally, one would place everything on-chip for faster access. However, due to the on-chip capacity limits (in the range of few MBs), this is not always possible, and consequently, on-board memory must be used as well. The ADADELTA gradient-based local-search method requires additional space in memory, which is attributed to the {F, G} structs listed in Table C.3.

Same requirements apply to Steepest-Descent and FIRE gradient-based methods.

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

E. g., for the Vega 56: 32 kB (local memory) × 44 (# CUs) = ∼ 1.4 MB.

Table C.1: Upper limits of MD parameters in OCLADock (defined in repository [187] `/common/defines.h`).

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

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

$$P_{\text{maxsize}} = R \cdot P \cdot L_{\text{genotype}} \cdot S_{\text{float}} \tag{C.1}$$

$$E_{\text{maxsize}} = R \cdot P \cdot S_{\text{float}} \tag{C.2}$$

where R and P are respectively the number of LGA runs and the population size (both specified by the user), L_{genotype} is the constant genotype length (= 64) in global memory, and S_{float} is the size of a `float` or single precision floating-point datatype (4 Bytes).

For R and P in above equations, upper limits have been defined as well. This means that OCLADock accepts for R and P any integer value so that $R \leq 1\,000$ and $P \leq 2\,048$. If the user inputs either an invalid or an out-of-range value, then OCLADock outputs a warning message, and then proceeds with execution using default values of R = 1 and P = 150. Then, as corner cases:

$$P_{\text{maxsize}} = 1\,000 \cdot 2\,048 \cdot 64 \cdot 4 = 524.28 \text{ MB} \tag{C.3}$$

$$E_{\text{maxsize}} = 1\,000 \cdot 2\,048 \cdot 4 = 8.19 \text{ MB} \tag{C.4}$$

which together account for 532.48 MB. Considering both – current and next – populations and their scores, their total memory footprint together is 1064.96 MB.

Table C.2: Constant data structures and their members in OCLADock.

Struct label	Constant array struct member	Element datatype	Size definition	Size calculation	Size (Bytes)
A	atom_charges	float	MAX_NUM_OF_ATOMS	4×256	1024
	atom_types	char	MAX_NUM_OF_ATOMS	1×256	256
B	intraE_contributors	char	3 × MAX_INTRAE_CONTRIBUTORS	$1 \times 3 \times 256 \times 256$	196 608
C	reqm	float	ATYPE_NUM	4×22	88
	reqm_hbond	float	ATYPE_NUM	4×22	88
	atom1_types_reqm	unsigned int	ATYPE_NUM	4×22	88
	atom2_types_reqm	unsigned int	ATYPE_NUM	4×22	88
	VWpars_AC	float	(MAX_NUM_OF_ATYPES× MAX_NUM_OF_ATYPES)	$4 \times 14 \times 14$	784
	VWpars_BD	float	(MAX_NUM_OF_ATYPES× MAX_NUM_OF_ATYPES)	$4 \times 14 \times 14$	784
	dspars_S	float	MAX_NUM_OF_ATYPES	4×14	56
	dspars_V	float	MAX_NUM_OF_ATYPES	4×14	56
D	rotlist	int	MAX_NUM_OF_ROTATIONS	$4 \times 256 \times 32$	32 768
E	ref_coords_x	float	MAX_NUM_OF_ATOMS	4×256	1024
	ref_coords_y	float	MAX_NUM_OF_ATOMS	4×256	1024
	ref_coords_z	float	MAX_NUM_OF_ATOMS	4×256	1024
	rotbonds_moving_vectors	float	3 × MAX_NUM_OF_ROTBONDS	$4 \times 3 \times 32$	384
	rotbonds_unit_vectors	float	3 × MAX_NUM_OF_ROTBONDS	$4 \times 3 \times 32$	384
	ref_orientation_quats	float	4 × MAX_NUM_OF_RUNS	$4 \times 4 \times 1000$	16 000
				Subtotal size (Bytes)	252 528
GRIDS	fgrids	float	MAX_NUM_OF_ATYPES× MAX_NUM_GRIDPOINTS^3	$4 \times 16 \times 256^3$	1 073 741 824
				Total size (Bytes)	1 073 994 352

Table C.3: Additional constant data structures and their members for gradient calculation in OCLADock.

Struct label	Constant array struct member	Element datatype	Size definition	Size (Bytes)
F	rotbonds_atoms	int	MAX_NUM_OF_ATOMS × MAX_NUM_OF_ROTBONDS	32 768
G	rotbonds	int	2 × MAX_NUM_OF_ROTBONDS	256
	num_rotating_atoms_per_rotbond	int	MAX_NUM_OF_ROTBONDS	128
	angle	float	1000	4000
	dependence_on_theta	float	1000	4000
	dependence_on_rotangle	float	1000	4000
			Total size (Bytes)	45 152

Summing up both maximum possible sizes of constant and variable data, the rounded-up memory space required by OCLADock is less than 2.2 GB, which is lower than the amount typically available even on mid-range consumer GPU cards, as exemplified in Table 6.1. Thus, there is no need for compute-specialized GPUs with larger memories, which are significantly more expensive, such as the DGX-2 hardware [127].

D

FUTURE TRENDS OF OPENCL

Since its first release in 2008, OpenCL has been improving continuously. Despite its well-known benefits – i. e., royalty-freedom, portability, performance, etc – some say that its adoption does not seem promising mainly due to Nvidia's CUDA dominance, particularly, in HPC computing [138]. In other words, besides

- the barely-competitive GPUs from competitors,
- the incomplete open-source OpenCL drivers, and
- the fact that closed drivers depend heavily on specific Linux kernel versions,

one of the main reasons discouraging developers increasingly adopting OpenCL is the very rich CUDA development ecosystem [126, 135] comprising:

- Optimized libraries (e. g., *cuBLAS*: dense linear algebra, *cuSPARSE*: sparse linear algebra, *Thrust*: scan, sort, reduce, transform, etc).
- Powerful directives (from OpenACC [139] that specifies code regions to be offloaded to accelerators).
- Widespread programming language and API support (Microsoft Direct X11, Python for CUDA, CUDA-x86, CUDA Fortran, OpenCL).
- Informative tools (Nvidia Visual Profiler, TAU Performance System).

Despite these advantages, the proprietary nature of CUDA implies risks due to *vendor lock-in*. From the developer/customer perspective, the vendor lock-in risks of proprietary software described in [153] can be contextualized for CUDA as follows:

Vendor lock-in is the situation in which customers are dependent on a single manufaturer for some product, and cannot move to another vendor without substantial costs [153].

- Porting CUDA codes to other parallel frameworks, and onto non-Nvidia GPUs, requires a significant effort. This is translated into substantial expenses and inconveniences.
- The dominance of CUDA might cause a lack of bargaining ability for price reduction and service enhancement.

However, the recent announcement of Intel's intentions to contribute support for SYCL into Clang/LLVM [85] opens a more concrete royalty-free alternative to the vendor lock-in issue of Nvidia. While SYCL could be a promising contender for CUDA, it is still a very early step towards an open ecosystem, with similar maturity to the one currently provided by Nvidia.

FULL LIST OF OWN PUBLICATIONS

[1] Léa El Khoury, Diogo Santos-Martins, Sukanya Sasmal, Jérôme Eberhardt, Giulia Bianco, Francesca A. Ambrosio, **Solis-Vasquez, Leonardo**, Andreas Koch, Stefano Forli, and David Mobley. "Comparison of affinity ranking using AutoDock-GPU and MM-GBSA scores for BACE-1 inhibitors in the D3R Grand Challenge 4." In: *Journal of Computer-Aided Molecular Design* (2019). DOI: 10.1007/s10822-019-00240-w.

[2] Diogo Santos-Martins, **Solis-Vasquez, Leonardo**, Andreas Koch, and Stefano Forli. "Accelerating AutoDock4 with GPUs and Gradient-Based Local Search." In: *ChemRxiv (preprint)* (2019). DOI: 10.26434/chemrxiv.9702389.v1.

[3] Diogo Santos-Martins, Jérôme Eberhardt, Giulia Bianco, **Solis-Vasquez, Leonardo**, Francesca Alessandra Ambrosio, Andreas Koch, and Stefano Forli. "D3R Grand Challenge 4: prospective pose prediction of BACE1 ligands with AutoDock-GPU." In: *Journal of Computer-Aided Molecular Design* (2019). DOI: 10.1007/s10822-019-00241-9.

[4] Lukas Sommer, Florian Stock, **Solis-Vasquez, Leonardo**, and Andreas Koch. *EPHoS: Evaluation of Programming Models for Heterogeneous Systems*. Berlin, Germany: German Association of the Automotive Industry (VDA: Verband der Automobilindustrie), 2019. URL: https://www.vda.de/de/services/Publikationen/fat-schriftenreihe-317.html.

[5] Lukas Sommer, Florian Stock, **Solis-Vasquez, Leonardo**, and Andreas Koch. "Work-in-Progress: DAPHNE - An Automotive Benchmark Suite for Parallel Programming Models on Embedded Heterogeneous Platforms." In: *Proceedings of the International Conference on Embedded Software (EMSOFT)*. New York, NY, USA: ACM, 2019. DOI: 10.1145/3349568.3351547.

[6] **Solis-Vasquez, Leonardo** and Andreas Koch. "A Performance and Energy Evaluation of OpenCL-accelerated Molecular Docking." In: *Proceedings of the 5th International Workshop on OpenCL (IWOCL)*. Toronto, ON, Canada: ACM, 2017. DOI: 10.1145/3078155.3078167.

[7] **Solis-Vasquez, Leonardo** and Andreas Koch. "A Case Study in Using OpenCL on FPGAs: Creating an Open-Source Accelerator of the AutoDock Molecular Docking Software." In: *Proceedings of the 5th International Workshop on FPGAs for Soft-*

ware Programmers (FSP). Dublin, Ireland: VDE VERLAG, 2018. URL: https://ieeexplore.ieee.org/document/8470463.

[8] **Solis-Vasquez, Leonardo**, Diogo Santos-Martins, Andreas Koch, and Stefano Forli. "Performance Analysis of Molecular Docking in OpenCL: A Case Study of AutoDock enhanced with Gradients." In: *Submitted to the 34th International Parallel and Distributed Processing Symposium (IPDPS)*. Submitted, 2019.

[9] **Solis-Vasquez, Leonardo**, Diogo Santos-Martins, Andreas Koch, and Stefano Forli. "Evaluating the Energy Efficiency of OpenCL-accelerated AutoDock Molecular Docking." In: *Submitted to the 28th Euromicro International Conference on Parallel, Distributed, and Network-based Processing (PDP)*. Submitted, 2020.

BIBLIOGRAPHY

[1] AMD. *OpenCL Programming Optimization Guide*. Last accessed: June 30, 2019. URL: http://developer.amd.com/wordpress/media/2013/12/AMD_OpenCL_Programming_Optimization_Guide2.pdf.

[2] David Abdurachmanov, Peter Elmer, Giulio Eulisse, Robert Knight, Tapio Niemi, Jukka K. Nurminen, Filip Nyback, Gonçalo Pestana, Zhonghong Ou, and Kashif Khan. "Techniques and tools for measuring energy efficiency of scientific software applications." In: *J. Physics: Conf. Series* 608 (2015). DOI: 10.1088/1742-6596/608/1/012032.

[3] Alexander Afanasiev, Igor Oferkin, Mikhail Posypkin, Anton Rubtsov, Alexey Sulimov, and Vladimir Sulimov. "A Comparative Study of Different Optimization Algorithms for Molecular Docking." In: *Proceedings of the 3rd International Workshop on Science Gateways for Life Sciences (IWSG)*. London, United Kingdom: CEUR Workshop Proceedings, 2011. URL: http://ceur-ws.org/Vol-819/.

[4] Ayesha Afzal, Christian Schmitt, Samer Alhaddad, Yevgen Grynko, Jurgen Teich, Jens Forstner, and Frank Hannig. "Solving Maxwell's Equations with Modern C++ and SYCL: A Case Study." In: *Proceedings of the 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. Milan, Italy: IEEE, 2018. DOI: 10.1109/ASAP.2018.8445127.

[5] University of Alberta. *Linear Feedback Shift Register (High-Level Digital ASIC Design Using CAD (course notes))*. Last accessed: July 30, 2019. URL: http://www.ece.ualberta.ca/~elliott/ee552/studentAppNotes/1999f/Drivers_Ed/lfsr.html.

[6] *Alibaba Hires Chief Quantum Scientist In Continued Cloud Push*. Last accessed: April 30, 2019. URL: https://www.alizila.com/alibaba-hires-chief-quantum-scientist-in-contineud-cloud-push.

[7] Scott E. Allen, Nikolay V. Dokholyan, and Albert A. Bowers. "Dynamic Docking of Conformationally Constrained Macrocycles: Methods and Applications." In: *ACS Chemical Biology* 11.1 (2016), pp. 10–24. DOI: 10.1021/acschembio.5b00663.

[8] Hiba Alogheli, Gustav Olanders, Wesley Schaal, Peter Brandt, and Anders Karlén. "Docking of Macrocycles: Comparing Rigid and Flexible Docking in Glide." In: *Journal of Chemical Informa-*

tion and Modeling 57.2 (2017), pp. 190–202. DOI: 10.1021/acs.jcim.6b00443.

[9] Serkan Altuntaş, Zeki Bozkus, and Basilio B. Fraguela. "GPU Accelerated Molecular Docking Simulation with Genetic Algorithms." In: *Applications of Evolutionary Computation: 19th European Conference, EvoApplications*. Springer, 2016, pp. 134–146. DOI: 10.1007/978-3-319-31153-1_10.

[10] *Amazon EC2 Pricing*. Last accessed: July 30, 2019. URL: https://aws.amazon.com/ec2/pricing/on-demand.

[11] *Amazon Elastic Compute Cloud, Compute Optimized Instances*. Last accessed: March 30, 2019. URL: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/compute-optimized-instances.html.

[12] *Amazon Elastic Compute Cloud, Processor State Control for Your EC2 Instance*. Last accessed: March 30, 2019. URL: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/processor_state_control.html.

[13] *Amazon Elastic Compute Cloud, User Guide for Linux Instances*. Last accessed: January 30, 2019. URL: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-optimize-cpu.html.

[14] Hartwig Anzt, Stanimire Tomov, and Jack Dongarra. "On the performance and energy efficiency of sparse linear algebra on GPUs." In: *The International Journal of High Performance Computing Applications* 31.15 (2017), pp. 375 –390. DOI: 10.1177/1094342016672081.

[15] Microsoft Azure. *Predicting ocean chemistry using Microsoft Azure*. Last accessed: April 30, 2019. URL: https://customers.microsoft.com/pt-br/story/taylorshellfishfarms.

[16] Ashok D. Belegundu and Tirupathi R. Chandrupatla. *Optimization Concepts and Applications in Engineering: Second Edition*. 3rd ed. Cambridge University Press, 2019. URL: https://www.cambridge.org/de/academic/subjects/engineering/control-systems-and-optimization/optimization-concepts-and-applications-engineering-3rd-edition?format=HB&isbn=9781108424882.

[17] Helen M. Berman, John Westbrook, Zukang Feng, Gary Gilliland, Talapady N. Bhat, Helge Weissig, Ilya N. Shindyalov, and Philip E. Bourne. "The Protein Data Bank." In: *Journal of Nucleic Acids Research* 28.1 (2000), pp. 235–242. DOI: 10.1093/nar/28.1.235.

[18] Geoffrey C. Berresford and Andrew M. Rocket. *Applied Calculus*. 7th ed. CENGAGE Learning, 2016. URL: https://www.cengage.com/c/applied-calculus-7e-berresford/.

[19] National Center for Biotechnology Information. *PubChem Database. CID=4369278*. Last accessed: September 30, 2019. URL: https://pubchem.ncbi.nlm.nih.gov/compound/4369278.

[20] Erik Bitzek, Pekka Koskinen, Franz Gähler, Michael Moseler, and Peter Gumbsch. "Structural relaxation made simple." In: *Journal of Physical Review Letters* 97.17 (2006), p. 170201. DOI: 10.1103/PhysRevLett.97.170201.

[21] OpenMP ARB (Architecture Review Boards). *OpenMP Resources*. Last accessed: April 30, 2019. URL: https://www.openmp.org/resources.

[22] *Boost.Thread – Overview*. Last accessed: July 30, 2019. URL: https://www.boost.org/doc/libs/1_71_0/doc/html/thread.html.

[23] Zeki Bozkus and Basilio B. Fraguela. "A Portable High-Productivity Approach to Program Heterogeneous Systems." In: *Proceedings of the*. Shanghai, China: IEEE, 2012. DOI: 10.1109/IPDPSW.2012.15.

[24] Robert A. Bridges, Neena Imam, and Tiffany M. Mintz. "Understanding GPU Power: A Survey of Profiling, Modeling, and Simulation Methods." In: *Journal of ACM Computing Surveys* 49.3 (2016), 41:1–41:27. DOI: 10.1145/2962131.

[25] Leibniz Supercomputing Centre. *Scientific Application Packages*. Last accessed: April 30, 2019. URL: https://doku.lrz.de/display/PUBLIC/Scientific+Application+Packages.

[26] Rong Chen, Li Li, and Zhiping Weng. "ZDOCK: an initial-stage protein-docking algorithm." In: *Journal of Proteins: Structure, Function, and Bioinformatics* 52.1 (2003), pp. 80 –87. DOI: 10.1002/prot.10389.

[27] Rong Chen and Zhiping Weng. "Docking unbound proteins using shape complementarity, desolvation, and electrostatics." In: *Journal of Proteins: Structure, Function, and Bioinformatics* 47.3 (2002), pp. 281 –294. DOI: 10.1002/prot.10092.

[28] *CodeXL - A comprehensive tool suite that enables developers to harness the benefits of CPUs, GPUs and APUs.* Last accessed: January 30, 2019. URL: https://github.com/GPUOpen-Tools/CodeXL.

[29] Codeplay. *Codeplay Announces World's First Fully-Conformant SYCL 1.2.1 Solution.* Last accessed: April 30, 2019. 2018. URL: https://www.codeplay.com/portal/08-23-18-codeplay-announces-world-s-first-fully-conformant-sycl-1-2-1-solution.

[30] Drug Design Data Resource (D3R) Community. *Grand Challenge 4: Affinity Predictions - BACE (Stage 1A)*. Last accessed: April 20, 2019. URL: https://drugdesigndata.org/php/d3r/gc4/combined/scoring/index.php?component=1146&method=structure.

[31] Drug Design Data Resource (D3R) Community. *Grand Challenge 4: Affinity Predictions - BACE (Stage 2)*. Last accessed: April 20, 2019. URL: https://drugdesigndata.org/php/d3r/gc4/combined/scoring/index.php?component=1479&method=structure.

[32] Drug Design Data Resource (D3R) Community. *Grand Challenge 4: Pose Prediction Method - BACE (Stage 1A)*. Last accessed: April 20, 2019. URL: https://drugdesigndata.org/php/d3r/gc4/combined/pose/index.php?component=1146&results=rmsd&chart=pose&partial=0&ligand=Mean.

[33] Drug Design Data Resource (D3R) Community. *Grand Challenge 4: Pose Prediction Method - BACE (Stage 1B)*. Last accessed: April 20, 2019. URL: https://drugdesigndata.org/php/d3r/gc4/combined/pose/index.php?component=1470&results=rmsd&chart=pose&partial=0&ligand=Mean.

[34] Drug Design Data Resource (D3R) Community. *Grand Challenge 4*. Last accessed: April 20, 2019. URL: https://drugdesigndata.org/about/grand-challenge-4.

[35] Center for Computational Structural Biology. *AutoDock4: Computational Docking of Ligands to Biomolecular Targets*. Last accessed: April 30, 2019. URL: https://ccsb.scripps.edu/autodock.

[36] Louisiana State University: High Performance Computing. *Alphabetical List of Software*. Last accessed: April 30, 2019. URL: http://www.hpc.lsu.edu/docs/guides/index.php#Chemistry.

[37] Alpha Data. *ADM-PCIE-7V3 – High Performance Computing*. Last accessed: May 30, 2019. URL: https://www.alpha-data.com/dcp/products.php?product=adm-pcie-7v3.

[38] James J. Davis, Joshua M. Levine, Edward A. Stott, Eddie Hung, Peter Y. K. Cheung, and George A. Constantinides. "KOCL: Power Self- Awareness for Arbitrary FPGA-SoC-Accelerated OpenCL Applications." In: *Journal of IEEE Design Test*. IEEE, 2017. DOI: 10.1109/MDAT.2017.2750909.

[39] Renata De Paris, Fábio A. Frantz, Osmar Norberto de Souza, and Duncan D. A. Ruiz. "wFReDoW: A Cloud-Based Web Environment to Handle Molecular Docking Simulations of a Fully Flexible Receptor Model." In: *Journal of BioMed Research International* 2013 (2013). DOI: 10.1155/2013/469363.

[40] P. Debye. "Näherungsformeln für die Zylinderfunktionen für große Werte des Arguments und unbeschränkt veränderliche Werte des Index." In: *Journal of Mathematische Annalen* 67.4 (1909), pp. 535 –558. DOI: 10.1007/BF01450097.

[41] The Tech Terms Computer Dictionary. *DDR3*. Last accessed: November 30, 2019. URL: https://techterms.com/definition/ddr3.

[42] Dong Dong, Zhijian Xu, Wu Zhong, and Shaoliang Peng. "Parallelization of Molecular Docking: A Review." In: *Journal of Current Topics in Medicinal Chemistry* 28.12 (2018), pp. 1015 – 1028. DOI: 10.2174/1568026618666180821145215.

[43] Victor Eijkhout. *Introduction to High-Performance Scientific Computing*. 2nd ed. 2016. URL: http://pages.tacc.utexas.edu/~eijkhout/istc/istc.html.

[44] Léa El Khoury, Diogo Santos-Martins, Sukanya Sasmal, Jérôme Eberhardt, Giulia Bianco, Francesca A. Ambrosio, **Solis-Vasquez, Leonardo**, Andreas Koch, Stefano Forli, and David Mobley. "Comparison of affinity ranking using AutoDock-GPU and MM-GBSA scores for BACE-1 inhibitors in the D3R Grand Challenge 4." In: *Journal of Computer-Aided Molecular Design* (2019). DOI: 10.1007/s10822-019-00240-w.

[45] U.S. Department of Energy. *Fact Sheet: Collaboration of Oak Ridge, Argonne, and Livermore (CORAL)*. Last accessed: April 30, 2019. URL: https://www.energy.gov/downloads/fact-sheet-collaboration-oak-ridge-argonne-and-livermore-coral.

[46] Karl Entacher. *A Collection of Selected Pseudorandom Number Generators with Linear Structures*. Last accessed: July 30, 2019. 1997. URL: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.3686.

[47] Max Planck Computing & Data Facility. *HPC Application Packages*. Last accessed: April 30, 2019. URL: https://www.mpcdf.mpg.de/services/computing/software/hpc_application_packages.html.

[48] Zhi-wei Feng, Xu-hong Tian, and Shan Chang. "A Parallel Molecular Docking Approach Based on Graphic Processing Unit." In: *Proceedings of the 4th International Conference on Bioinformatics and Biomedical Engineering*. Chengdu, China: IEEE, 2010. DOI: 10.1109/ICBBE.2010.5514919.

[49] Leonardo G. Ferreira, Ricardo N. Dos Santos, Glaucius Oliva, and Adriano D. Andricopulo. "Molecular Docking and Structure-Based Drug Design Strategies." In: *Journal of Molecules* 20.7 (2015), pp. 13384–13421. DOI: 10.3390/molecules200713384.

[50] *FightAIDS@Home*. Last accessed: July 30, 2019. URL: http://fightaidsathome.scripps.edu.

[51] Stefano Forli and Maurizio Botta. “Lennard-Jones Potential and Dummy Atom Settings to Overcome the AUTODOCK Limitation in Treating Flexible Ring Systems.” In: *Journal of Chemical Information and Modeling* 47.4 (2007), pp. 1481–1492. DOI: 10.1021/ci700036j.

[52] German Research Foundation (DFG: Deutsche Forschungsgemeinschaft). *DFG Classification of Scientific Disciplines, Research Areas, Review Boards and Subject Areas (2016-2019)*. Last accessed: April 30, 2019. URL: https://www.dfg.de/download/pdf/dfg_im_profil/gremien/fachkollegien/amtsperiode_2016_2019/fachsystematik_2016-2019_en_grafik.pdf.

[53] Jan Fuhrmann, Alexander Rurainski, Hans-Peter Lenhof, and Dirk Neumann. “A new method for the gradient-based optimization of molecular complexes.” In: *Journal of Computational Chemistry* 30.9 (2009), pp. 1371–1378. DOI: 10.1002/jcc.21159.

[54] Qatar National Research Fund. *The fields of science and technology classification*. Last accessed: April 30, 2019. URL: https://www.qnrf.org/en-us/FOS.

[55] *GOLD*. Last accessed: May 30, 2019. URL: https://www.ccdc.cam.ac.uk/solutions/csd-discovery/components/gold.

[56] Alan D. George, Martin C. Herbordt, Herman Lam, Abhijeet G. Lawande, Jiayi Sheng, and Chen Yang. “Novo-G#: Large-scale reconfigurable computing with direct and programmable interconnects.” In: *Proceedings of the High Performance Extreme Computing Conference (HPEC)*. Waltham, MA, USA: IEEE, 2016. DOI: 10.1109/HPEC.2016.7761639.

[57] Nick Gibbs, Anthony R. Clarke, and Richard B. Sessions. “Ab initio protein structure prediction using physicochemical potentials and a simplified off-lattice model.” In: *Proteins: Structure, Function, and Bioinformatics* 43.2 (2001), pp. 186–202. DOI: 10.1002/1097-0134(20010501)43:2<186::AID-PROT1030>3.0.CO;2-L.

[58] *Glide – A complete solution for ligand-receptor docking*. Last accessed: May 30, 2019. URL: https://www.schrodinger.com/glide.

[59] Gene H. Golub and James M. Ortega. *Scientific Computing and Differential Equations: An Introduction to Numerical Methods*. Academic Press, Inc., 1992. URL: https://dl.acm.org/citation.cfm?id=574155.

[60] Robson Gonçalves, Alessandro Girardi, and Claudio Schepke. “Performance and Energy Consumption Analysis of Coprocessors Using Different Programming Models.” In: *Euromicro Int. Conf. on Parallel, Distributed and Network-based Processing*

(PDP). Cambridge, UK: IEEE, 2018. DOI: 10.1109/PDP2018.2018.00086.

[61] David S. Goodsell and Arthur J. Olson. "Automated docking of substrates to proteins by simulated annealing." In: *Proteins: Structure, Function, and Bioinformatics* 8.3 (1990), pp. 195–202. DOI: 10.1002/prot.340080302.

[62] Khronos Group. *OpenCL 2.0 Reference Pages*. Last accessed: May 30, 2019. URL: https://www.khronos.org/registry/OpenCL/sdk/2.0/docs/man/xhtml.

[63] Khronos Group. *OpenCL Overview: The open standard for parallel programming of heterogeneous systems.* Last accessed: April 30, 2019. URL: https://www.khronos.org/opencl.

[64] Khronos Group. *OpenCL Resources*. Last accessed: April 30, 2019. URL: https://www.khronos.org/opencl/resources.

[65] Khronos Group. *SYCL Overview: C++ Single-source Heterogeneous Programming for OpenCL.* Last accessed: April 30, 2019. URL: https://www.khronos.org/sycl.

[66] Khronos Group. *SYCL Resources*. Last accessed: April 30, 2019. URL: https://www.khronos.org/sycl/resources.

[67] Wan-Gang Gu, Xuan Zhang, and Jun-Fa Yuan. "Anti-HIV Drug Development Through Computational Methods." In: *The Journal of American Association of Pharmaceutical Scientists* 16.4 (2014), pp. 674–680. DOI: 10.1208/s12248-014-9604-9.

[68] Ginés D. Guerrero, Horacio Pérez-Sánchez, Wolfgang Wenzel, José M. Cecilia, and José M. García. "Effective Parallelization of Non-bonded Interactions Kernel for Virtual Screening on GPUs." In: *Proceedings of the 5th International Conference on Practical Applications of Computational Biology & Bioinformatics (PACBB)*. Salamanca, Spain: Springer, 2011. DOI: 10.1007/978-3-642-19914-1_9.

[69] Christoph Hagleitner. *Application Porting & Optimization on GPU-accelerated POWER Architectures - Best practices for porting scientific applications*. Last accessed: April 30, 2019. URL: http://juser.fz-juelich.de/record/840162/files/6-CHagleitner-Best_Practices.pdf.

[70] Inbal Halperin, Buyong Ma, Haim Wolfson, and Ruth Nussinov. "Principles of docking: An overview of search algorithms and a guide to scoring functions." In: *Journal of Proteins: Structure, Function, and Bioinformatics* 47.4 (2002), pp. 409–443. DOI: 10.1002/prot.10115.

[71] Michael J. Hartshorn, Marcel L. Verdonk, Gianni Chessari, Suzanne C. Brewerton, Wijnand T.M. Mooij, Paul N. Mortenson, and Christopher W. Murray. "Diverse, high-quality test set for the validation of protein-ligand docking performance." In: *Journal of Medicinal Chemistry* 50.4 (2007), pp. 726–741. DOI: 10.1021/jm061277y.

[72] Scott Hauck and André DeHon, eds. *Reconfigurable Computing – The Theory and Practice of FPGA-based Computing*. Morgan Kaufmann, 2008. URL: https://dl.acm.org/citation.cfm?id=1564780.

[73] Csaba Hetényi and David van der Spoel. "Efficient docking of peptides to proteins without prior knowledge of the binding site." In: *Journal of Protein Science* 11.7 (2002), pp. 1729–1737. DOI: 10.1110/ps.0202302.

[74] *Heterogeneous Programming Library. Facilitates the use of accelerators on top of OpenCL*. Last accessed: July 30, 2019. URL: https://github.com/fraguela/hpl.

[75] Abigail Hsu, David N. Asanza, Joseph A. Schoonover, Zach Jibben, Neil N. Carlson, and Robert Robey. "Performance Portability Challenges for Fortran Applications." In: *Proceedings of the International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. Dallas, TX, USA: IEEE, 2018. DOI: 10.1109/P3HPC.2018.00008.

[76] Ruth Huey, Garret M. Morris, and Stefano Forli. *Using AutoDock 4 and AutoDock Vina with AutoDockTools: A Tutorial*. Last accessed: March 30, 2019. URL: https://autodock.scripps.edu/faqs-help/tutorial/using-autodock-4-with-autodocktools/2012_ADTtut.pdf.

[77] Ruth Huey, Garrett M. Morris, Arthur J. Olson, and David S. Goodsell. "A semiempirical free energy force field with charge-based desolvation." In: *Journal of Computational Chemistry* 28.6 (2007), pp. 1145–1152. DOI: 10.1002/jcc.20634.

[78] *Hydrogen Bonding*. Last accessed: July 30, 2019. URL: https://chem.libretexts.org/Bookshelves/Physical_and_Theoretical_Chemistry_Textbook_Maps/Supplemental_Modules_(Physical_and_Theoretical_Chemistry)/Physical_Properties_of_Matter/Atomic_and_Molecular_Properties/Intermolecular_Forces/Specific_Interactions/Hydrogen_Bonding.

[79] Open SystemC Initiative. *The Next IC Design Methodology Transition Is Long Overdue*. Last accessed: July 30, 2019. 2010. URL: https://www.accellera.org/images/resources/articles/icdesigntrans/ic_design_transition_feb2010.pdf.

[80] *Intel Dumps Knights Hill, Future of Xeon Phi Product Line Uncertain*. Last accessed: April 30, 2019. URL: https://www.top500.org/news/intel-dumps-knights-hill-future-of-xeon-phi-product-line-uncertain.

[81] Intel. *Intel Arria 10 Device Overview*. Last accessed: January 30, 2019. URL: https://www.altera.com/en_US/pdfs/literature/hb/arria-10/a10_overview.pdf.

[82] Intel. *Intel FPGA IP Portfolio*. Last accessed: July 30, 2019. URL: https://www.intel.com/content/www/us/en/products/programmable/intellectual-property.html.

[83] Intel. *Intel FPGA SDK for OpenCL*. Last accessed: January 30, 2019. URL: https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html.

[84] Intel. *Intel SDK for OpenCL Applications*. Last accessed: April 30, 2019. URL: https://software.intel.com/en-us/intel-opencl.

[85] Intel. *SYCL compiler: zero-cost abstraction and type safety for heterogeneous computing*. Last accessed: April 30, 2019. URL: https://llvm.org/devmtg/2019-04/talks.html#Talk_14.

[86] Pekka Jääskeläinen, Carlos Sánchez de La Lama, Erik Schnetter, Kalle Raiskila, Jarmo Takala, and Heikki Berg. "pocl: A Performance-Portable OpenCL Implementation." In: *International Journal of Parallel Programming* 43.5 (2015), pp. 752–785. DOI: 10.1007/s10766-014-0320-y.

[87] David Kaeli, Perhaad Mistry, Dana Schaa, and Dong Ping Zhang. *Heterogeneous Computing with OpenCL 2.0*. 3rd ed. Morgan Kaufmann, 2015. URL: https://dl.acm.org/citation.cfm?id=2815521.

[88] Sarnath Kannan and Raghavendra Ganji. "Porting Autodock to CUDA." In: *Proceedings of the IEEE Congress on Evolutionary Computation*. Barcelona, Spain: IEEE, 2010. DOI: 10.1109/CEC.2010.5586277.

[89] Nachiket Kapre and Samuel Bayliss. "Survey of domain-specific languages for FPGA computing." In: *Proceedings of 26th International Conference on Field Programmable Logic and Applications (FPL)*. Lausanne, Switzerland: IEEE, 2016. DOI: 10.1109/FPL.2016.7577380.

[90] P. Karlson and M. Lüscher. "'Pheromones': a New Term for a Class of Biologically Active Substances." In: *International Journal of Science* 183.4653 (1959), pp. 55 –56. DOI: 10.1038/183055a0.

[91] E. Katchalski-Katzir, I. Shariv, M. Eisenstein, A. A. Friesem, C. Aflalo, and I. A. Vakser. "Molecular surface recognition: determination of geometric fit between proteins and their ligands by correlation techniques." In: *Proceedings of the National Academy of Sciences* 89.6 (1992), pp. 2195 –2199. DOI: 10.1073/pnas.89.6.2195.

[92] Esther Kellenberger, Jordi Rodrigo, Pascal Muller, and Didier Rognan. "Comparative evaluation of eight docking tools for docking and virtual screening accuracy." In: *Proteins: Structure, Function, and Bioinformatics* 57.2 (2004), pp. 225–242. DOI: 10.1002/prot.20149.

[93] Tobias Kenter, Jens Förstner, and Christian Plessl. "Flexible FPGA design for FDTD using OpenCL." In: *Proceedings of the 27th International Conference on Field Programmable Logic and Applications (FPL)*. Ghent, Belgium: IEEE, 2017. DOI: 10.23919/FPL.2017.8056844.

[94] Tobias Kenter and Christian Plessl. "Microdisk Cavity FDTD Simulation on FPGA using OpenCL." In: *Proceedings of the Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. Salt Lake City, UT, USA, 2016. URL: https://h2rc.cse.sc.edu/2016/papers/paper_26.pdf.

[95] Ronan Keryell and Lin-Ya Yu. "Early Experiments Using SYCL Single-source Modern C++ on Xilinx FPGA: Extended Abstract of Technical Presentation." In: *Proceedings of the 6th International Workshop on OpenCL (IWOCL)*. Oxford, United Kingdom: ACM, 2018. DOI: 10.1145/3204919.3204937.

[96] Virginia A. Kincaid et al. "Virtual Screening for UDP-Galactopyranose Mutase Ligands Identifies a New Class of Antimycobacterial Agents." In: *Journal of ACS Chemical Biology* 10.10 (2015), pp. 2209–2218. DOI: 10.1021/acschembio.5b00370.

[97] Oliver Korb, Thomas Stützle, and Thomas E. Exner. "An ant colony optimization approach to flexible protein–ligand docking." In: *Journal of Swarm Intelligence* 1.2 (2007), pp. 115 –134. DOI: 10.1007/s11721-007-0006-9.

[98] Oliver Korb, Thomas Stützle, and Thomas E. Exner. "Accelerating Molecular Docking Calculations Using Graphics Processing Units." In: *Journal of Chemical Information and Modeling* 51.4 (2011), pp. 865 –876. DOI: 10.1021/ci100459b.

[99] Pekka Koskinen, Erik Bitzek, Franz Gähler, Michael Moseler, and Peter Gumbsch. *FIRE: Fast Inertial Relaxation Engine for Optimization on All Scales*. Last accessed: July 30, 2019. 2006. URL: http://users.jyu.fi/~pekkosk/resources/pdf/FIRE.pdf.

[100] Dima Kozakov, Ryan Brenke, Stephen R. Comeau, and Sandor Vajda. "PIPER: An FFT-based protein docking program with pairwise potentials." In: *Journal of Proteins: Structure, Function, and Bioinformatics* 65.2 (2006), pp. 392 –406. DOI: 10.1002/prot.21117.

[101] Lawrence Livermore National Laboratory. *What is CORAL?* Last accessed: April 30, 2019. URL: https://asc.llnl.gov/coral-info.

[102] Lang Yu, Zhongzhi Luan, Xiangzheng Sun, Zhe Wang, and Hailong Yang. "VinaSC: Scalable Autodock Vina with fine-grained scheduling on heterogeneous platform." In: *Proceedings of the International Conference on Bioinformatics and Biomedicine (BIBM)*. Shenzhen, China: IEEE, 2016. DOI: 10.1109/BIBM.2016.7822624.

[103] *Lennard-Jones Potential*. Last accessed: July 30, 2019. URL: https://chem.libretexts.org/Bookshelves/Physical_and_Theoretical_Chemistry_Textbook_Maps/Supplemental_Modules_(Physical_and_Theoretical_Chemistry)/Physical_Properties_of_Matter/Atomic_and_Molecular_Properties/Intermolecular_Forces/Specific_Interactions/Lennard-Jones_Potential.

[104] Yan Li, Li Han, Zhihai Liu, and Renxiao Wang. "Comparative assessment of scoring functions on an updated benchmark: 2. Evaluation methods and general results." In: *Journal of Chemical Information and Modeling* 54.6 (2014), pp. 1717–1736. DOI: 10.1021/ci500081m.

[105] Jie Liu and Renxiao Wang. "Classification of Current Scoring Functions." In: *Journal of Chemical Information and Modeling* 55.3 (2015), pp. 475–482. DOI: 10.1021/ci500731a.

[106] Tingting Liu, Dong Lu, Hao Zhang, Mingyue Zheng, Huaiyu Yang, Yechun Xu, Cheng Luo, Weiliang Zhu, Kunqian Yu, and Hualiang Jiang. "Applying high-performance computing in drug discovery and molecular simulation." In: *Journal of National Science Review* 3.1 (2016), pp. 49–63. DOI: 10.1093/nsr/nww003.

[107] *MPI Forum*. Last accessed: July 30, 2019. URL: https://www.mpi-forum.org.

[108] Matthew Martineau, Simon McIntosh-Smith, and Wayne Gaudin. "Assessing the performance portability of modern parallel programming models using TeaLeaf." In: *Journal of Concurrency and Computation: Practice and Experience* 29.15 (2017). DOI: 10.1002/cpe.4117.

[109] Simon McIntosh-Smith, Terry Wilson, Amaurys Ávila Ibarra, Jonathan Crisp, and Richard B. Sessions. "Benchmarking energy efficiency, power costs and carbon emissions on heterogeneous systems." In: *Computer Journal* 55.2 (2012), pp. 192 –205. DOI: 10.1093/comjnl/bxr091.

[110] Simon McIntosh-Smith, James Price, Richard B. Sessions, and Amaurys A. Ibarra. "High performance in silico virtual drug screening on many-core processors." In: *The International Journal of High Performance Computing Applications* 29.2 (2014), pp. 119 –134. DOI: 10.1177/1094342014528252.

[111] Everton Mendonça, Marcos Barreto, Vinícius Guimarães, Nelci Santos, Samuel Pita, and Murilo Boratto. "Accelerating Docking Simulation Using Multicore and GPU Systems." In: *Proceedings of the 17th International Computational Science and Its Applications (ICCSA)*. Trieste, Italy: Springer, 2017. DOI: 10.1007/978-3-319-62392-4_32.

[112] Mentor. *Handel-C Synthesis Methodology – Handel-C to FPGA for Algorithm Design*. Last accessed: July 30, 2019. URL: https://www.mentor.com/products/fpga/handel-c.

[113] Micron. *AC-505 Overview*. Last accessed: May 30, 2019. URL: https://www.micron.com/products/advanced-solutions/advanced-computing-solutions/ac-series-hpc-modules/ac-505.

[114] Sparsh Mittal and Jeffrey S. Vetter. "A Survey of CPU-GPU Heterogeneous Computing Techniques." In: *Journal of ACM Computing Surveys* 47.4 (2015), 69:1–69:35. DOI: 10.1145/2788396.

[115] Sparsh Mittal and Jeffrey S. Vetter. "A Survey of Methods for Analyzing and Improving GPU Energy Efficiency." In: *Journal of ACM Computing Surveys* 47.2 (2015), 19:1–19:23. DOI: 10.1145/2636342.

[116] Garrett M. Morris, David S. Goodsell, Robert S. Halliday, Ruth Huey, William E. Hart, Richard K. Belew, and Arthur J. Olson. "Automated docking using a Lamarckian genetic algorithm and an empirical binding free energy function." In: *Journal of Computational Chemistry* 19.14 (1998), pp. 1639–1662. DOI: 10.1002/(SICI)1096-987X(19981115)19:14<1639::AID-JCC10>3.0.CO;2-B.

[117] Garrett M. Morris, Ruth Huey, William Lindstrom, Michel F. Sanner, Richard K. Belew, David S. Goodsell, and Arthur J. Olson. "AutoDock4 and AutoDockTools4: Automated docking with selective receptor flexibility." In: *Journal of Computational Chemistry* 30.16 (2009), pp. 2785–2791. DOI: 10.1002/jcc.21256.

[118] BioWulf: High Performance Computing at the NIH. *Scientific Applications on NIH HPC Systems*. Last accessed: April 30, 2019. URL: https://hpc.nih.gov/apps.

[119] National Science Foundation (NSF). *Science and Engineering Degrees: 1966–2012*. Last accessed: April 30, 2019. URL: https://www.nsf.gov/statistics/2015/nsf15326/pdf/nsf15326.pdf.

[120] J. A. Nelder and R. Mead. "A Simplex Method for Function Minimization." In: *The Computer Journal* 7.4 (1965), pp. 308 –313. DOI: 10.1093/comjnl/7.4.308.

[121] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation." In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. San Diego, CA, USA: ACM, 2007. DOI: 10.1145/1273442.1250746.

[122] Marco A. S. Netto, Rodrigo N. Calheiros, Eduardo R. Rodrigues, Renato L. F. Cunha, and Rajkumar Buyya. "HPC Cloud for Scientific and Business Applications: Taxonomy, Vision, and Research Challenges." In: *Journal of ACM Computing Surveys* 51.1 (2018), 8:1–8:29. DOI: 10.1145/3150224.

[123] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. 2nd ed. Springer, 2006. URL: https://link.springer.com/book/10.1007/978-0-387-40065-5.

[124] Andrew P. Norgan, Paul K. Coffman, Jean-Pierre A. Kocher, David J. Katzmann, and Carlos P. Sosa. "Multilevel Parallelization of AutoDock 4.2." In: *Journal of Cheminformatics* 3.1 (2011), p. 12. DOI: 10.1186/1758-2946-3-12.

[125] University of North Texas: High Performance Computing. *Scientific Software Guide*. Last accessed: April 30, 2019. URL: https://hpc.unt.edu/software?field_research_area_value=chem.

[126] Nvidia. *CUDA Libraries and Ecosystem Overview*. Last accessed: april 30, 2019. URL: http://developer.download.nvidia.com/GTC/PDF/1061_Woolley.pdf.

[127] Nvidia. *DGX-2. The world's most powerful AI system for the most complex AI challenges.* Last accessed: July 30, 2019. URL: https://www.nvidia.com/en-us/data-center/dgx-2.

[128] Nvidia. *GPU-Accelerated Applications*. Last accessed: April 30, 2019. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/gpu-applications-catalog.pdf.

[129] Nvidia. *Jetson AGX Xavier Developer Kit*. Last accessed: April 30, 2019. URL: https://developer.nvidia.com/embedded/buy/jetson-agx-xavier-devkit.

[130] Nvidia. *Jetson TX2 Module*. Last accessed: April 30, 2019. URL: https://developer.nvidia.com/embedded/buy/jetson-tx2.

[131] Nvidia. *Nvidia System Management Interface*. Last accessed: March 30, 2019. URL: https://developer.nvidia.com/nvidia-system-management-interface.

[132] Nvidia. *OpenCL Best Practices Guide 1.0*. Last accessed: June 30, 2019. URL: https://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf.

[133] Nvidia. *Tesla P100*. Last accessed: April 30, 2019. URL: https://www.nvidia.com/en-us/data-center/tesla-p100.

[134] Nvidia. *Tesla V100*. Last accessed: April 30, 2019. URL: https://www.nvidia.com/en-us/data-center/tesla-v100.

[135] Nvidia. *Tools & Ecosystem*. Last accessed: april 30, 2019. URL: https://developer.nvidia.com/tools-ecosystem.

[136] *Oclgrind - An OpenCL device simulator and debugger.* Last accessed: April 30, 2019. URL: https://github.com/jrprice/Oclgrind.

[137] *Open Cores – The reference community for Free and Open Source gateware IP cores*. Last accessed: July 30, 2019. URL: https://opencores.org.

[138] *Open-Source OpenCL Adoption Is Sadly An Issue In 2017*. Last accessed: April 30, 2019. URL: https://www.phoronix.com/scan.php?page=news_item&px=XDC2017-OpenCL-GPGPU.

[139] OpenACC-standard.org. *OpenACC: More Science Less Programming*. Last accessed: July 30, 2019. URL: https://www.openacc.org.

[140] Nataraj S. Pagadala, Khajamohiddin Syed, and Jack Tuszynski. "Software for molecular docking: a review." In: *Journal of Biophysical Reviews* 9.2 (2017), pp. 91–102. DOI: 10.1007/s12551-016-0247-1.

[141] Yuan-Ping Pang, Emanuele Perola, Kun Xu, and Franklyn G. Prendergast. "EUDOC: a computer program for identification of drug interaction sites in macromolecules and drug leads from chemical databases." In: *Journal of Computational Chemistry* 22.15 (2001), pp. 1750–1771. DOI: 10.1002/jcc.1129.

[142] Universität Paderborn: Paderborn Center for Parallel Computing (PC2). *HPC Services: FPGA Research Clusters*. Last accessed: April 30, 2019. URL: https://pc2.uni-paderborn.de/hpc-services/available-systems/fpga-research-clusters.

[143] Universität Paderborn: Paderborn Center for Parallel Computing (PC2). *Software*. Last accessed: April 30, 2019. URL: https://wikis.uni-paderborn.de/pc2doc/Software#Software_Availability.

[144] Imre Pechan and Bela Fehér. "Molecular Docking on FPGA and GPU Platforms." In: *Proceedings of the 21st International Conference on Field Programmable Logic and Applications (FPL)*. Chania, Greece: IEEE, 2011. DOI: 10.1109/FPL.2011.93.

[145] Imre Pechan and Béla Fehér. "Hardware Accelerated Molecular Docking: A Survey." In: *Bioinformatics*. London, United Kingdom: InTech, 2012. DOI: 10.5772/48125.

[146] Imre Pechan, Béla Fehér, and Attila Bérces. "FPGA-based acceleration of the AutoDock molecular docking software." In: *Proceedings of the 6th Conference on Ph.D. Research in Microelectronics Electronics*. Berlin, Germany: IEEE, 2010. URL: https://ieeexplore.ieee.org/document/5587139.

[147] S. J. Pennycook and S. A. Jarvis. "Developing Performance-Portable Molecular Dynamics Kernels in OpenCL." In: *Proceedings of the SC Companion: High Performance Computing, Networking Storage and Analysis*. Salt Lake City, UT, USA: IEEE, 2012. DOI: 10.1109/SC.Companion.2012.58.

[148] S.J. Pennycook, S.D. Hammond, S.A. Wright, J.A. Herdman, I. Miller, and S.A. Jarvis. "An investigation of the performance portability of OpenCL." In: *Journal of Parallel and Distributed Computing* 73.11 (2013), pp. 1439 –1450. DOI: 10.1016/j.jpdc.2012.07.005.

[149] Christian Plessl. "Keynote 2: FPGA-accelerated high-performance computing - Close to breakthrough or pipedream?" In: *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. Cancun, Mexico: IEEE, 2017. DOI: 10.1109/RECONFIG.2017.8279813.

[150] *Portable Computing Language | NVIDIA GPU support via CUDA backend*. Last accessed: April 30, 2019. URL: http://portablecl.org/cuda-backend.html.

[151] Ralph Potter, Paul Keir, Russell J. Bradford, and Alastair Murray. "Kernel Composition in SYCL." In: *Proceedings of the 3rd International Workshop on OpenCL (IWOCL)*. Palo Alto, CA, USA: ACM, 2015. DOI: 10.1145/2791321.2791332.

[152] Federica Prati, Giovanni Bottegoni, Maria Laura Bolognesi, and Andrea Cavalli. "BACE-1 Inhibitors: From Recent Single-Target Molecules to Multitarget Compounds for Alzheimer's Disease." In: *Journal of Medicinal Chemistry* 61.3 (2018), pp. 619–637. DOI: 10.1021/acs.jmedchem.7b00393.

[153] The Linux Information Project. *Vendor Lock-in Definition*. Last accessed: July 30, 2019. URL: http://www.linfo.org/vendor_lockin.html.

[154] Pradeep A. Ravindranath, Stefano Forli, David S. Goodsell, Arthur J. Olson, and Michel F. Sanner. "AutoDockFR: Advances in Protein-Ligand Docking with Explicitly Specified Binding Site Flexibility." In: *Journal of PLOS Computational Biology* 11.12 (2015), pp. 1 –28. DOI: 10.1371/journal.pcbi.1004586.

[155] Noraini M. Razali and John Geraghty. "Genetic Algorithm Performance with Different Selection Strategies in Solving TSP." In: *World Congress on Engineering*. London, United Kingdom: IAENG, 2011. URL: http://www.iaeng.org/publication/WCE2011/WCE2011_pp1134-1139.pdf.

[156] David W. Ritchie, Dima Kozakov, and Sandor Vajda. "Accelerating and focusing protein–protein docking correlations using multi-dimensional rotational FFT generating functions." In: *Journal of Bioinformatics* 24.17 (2008), pp. 1865 –1873. DOI: 10.1093/bioinformatics/btn334.

[157] David W. Ritchie and Vishwesh Venkatraman. "Ultra-fast FFT protein docking on graphics processors." In: *Journal of Bioinformatics* 26.19 (2010), pp. 2398 –2405. DOI: 10.1093/bioinformatics/btq444.

[158] Youngtae Roh, Jun Lee, Sungjun Park, and Jee-In Kim. "A molecular docking system using CUDA." In: *Proceedings of the International Conference on Hybrid Information Technology*. Daejeon, Korea: ACM, 2009. DOI: 10.1145/1644993.1644999.

[159] Hocine Saadi, Nadia Nouali Taboudjemat, Abdellatif Rahmoun, Baldomero Imbernón, Horacio Pérez-Sánchez, and José M. Cecilia. "Parallel Desolvation Energy Term Calculation for Blind Docking on GPU Architectures." In: *Proceedings of the 46th International Conference on Parallel Processing Workshops (ICPPW)*. Bristol, United Kingdom: IEEE, 2017. DOI: 10.1109/ICPPW.2017.16.

[160] Hocine Saadi, Nadia Nouali Taboudjemat, Abdellatif Rahmoun, Baldomero Imbernón, Horacio Pérez-Sánchez, and José M. Cecilia. "Efficient GPU-based parallelization of solvation calculation for the blind docking problem." In: *The Journal of Supercomputing* (2019). DOI: 10.1007/s11227-019-02834-5.

[161] F. Anthony San Lucas, Jerry Fowler, Kyle Chang, Scott Kopetz, Eduardo Vilar, and Paul Scheet. "Cancer In Silico Drug Discovery: A Systems Biology Tool for Identifying Candidate Drugs to Target Specific Molecular Tumor Subtypes." In: *Journal of Molecular Cancer Therapeutics* 13.12 (2014), pp. 3230–3240. DOI: 10.1158/1535-7163.MCT-14-0260.

[162] Ahmed Sanaullah and Martin C. Herbordt. "FPGA HPC Using OpenCL: Case Study in 3D FFT." In: *Proceedings of the 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART)*. Toronto, ON, Canada: ACM, 2018. DOI: 10.1145/3241793.3241800.

[163] Ahmed Sanaullah and Martin C. Herbordt. "Unlocking Performance-Programmability by Penetrating the Intel FPGA OpenCL Toolflow." In: *Proceedings of the 2018 High Performance Extreme Computing Conference (HPEC)*. Waltham, MA, USA: IEEE, 2018. DOI: 10.1109/HPEC.2018.8547646.

[164] Ahmed Sanaullah, Rushi Patel, and Martin C. Herbordt. "An Empirically Guided Optimization Framework for FPGA OpenCL." In: *Proceedings of the 2018 International Conference on Field Programmable Technology (ICFPT)*. Naha, Okinawa, Japan: IEEE, 2018. DOI: 10.1109/FPT.2018.00018.

[165] Diogo Santos-Martins, **Solis-Vasquez, Leonardo**, Andreas Koch, and Stefano Forli. "Accelerating AutoDock4 with GPUs and Gradient-Based Local Search." In: *ChemRxiv (preprint)* (2019). DOI: 10.26434/chemrxiv.9702389.v1.

[166] Diogo Santos-Martins, Jérôme Eberhardt, Giulia Bianco, **Solis-Vasquez, Leonardo**, Francesca Alessandra Ambrosio, Andreas Koch, and Stefano Forli. "D3R Grand Challenge 4: prospective pose prediction of BACE1 ligands with AutoDock-GPU." In: *Journal of Computer-Aided Molecular Design* (2019). DOI: 10.1007/s10822-019-00241-9.

[167] Matthew Scarpino. *OpenCL in Action*. Manning Publications, 2012. URL: https://www.manning.com/books/opencl-in-action.

[168] Ada Sedova, John D. Eblen, Reuben Budiardja, Arnold Tharrington, and Jeremy C. Smith. "High-Performance Molecular Dynamics Simulation for Biological and Materials Sciences: Challenges of Performance Portability." In: *Proceedings of the International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. Dallas, TX, USA: IEEE, 2018. DOI: 10.1109/P3HPC.2018.00004.

[169] Amazon Web Services. *Pharma & Biotech in the Cloud*. Last accessed: April 30, 2019. URL: https://aws.amazon.com/health/biotech-pharma.

[170] *Shoubu system B*. Last accessed: April 30, 2019. URL: https://www.top500.org/system/179165.

[171] Balint Siklosi, Istvan Z. Reguly, and Gihan R. Mudalige. "Heterogeneous CPU-GPU Execution of Stencil Applications." In: *Proceedings of the International Workshop on Performance, Porta-*

bility and Productivity in HPC (P3HPC). Dallas, TX, USA: IEEE, 2018. DOI: 10.1109/P3HPC.2018.00010.

[172] Hércules Cardoso Da Silva, Flávia Pisani, and Edson Borin. "A Comparative Study of SYCL, OpenCL, and OpenMP." In: *Proceedings of the 2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*. Los Angeles, CA, USA: IEEE, 2016. DOI: 10.1109/SBAC-PADW.2016.19.

[173] Martin Simonsen, Mikael H. Christensen, René Thomsen, and Christian N. S. Pedersen. "GPU-Accelerated High-Accuracy Molecular Docking Using Guided Differential Evolution." In: *Massively Parallel Evolutionary Computation on GPGPUs*. Springer, 2013, pp. 349 –367. DOI: 10.1007/978-3-642-37959-8_16.

[174] M. S. Smyth and J. H. J. Martin. "x Ray crystallography." In: *Journal of Molecular Pathology* 53.1 (2000), pp. 8 –14. DOI: 10.1136/mp.53.1.8.

[175] Francisco J. Solis and Roger J. B. Wets. "Minimization by Random Search Techniques." In: *Journal of Mathematics of Operations Research* 6.1 (1981), pp. 19–30. DOI: 10.1287/moor.6.1.19.

[176] *Solutions, Solvation, and Dissociation*. Last accessed: July 30, 2019. URL: https://chem.libretexts.org/Bookshelves/General_Chemistry/Book%3A_General_Chemistry_Supplement_(Eames)/Chemical_Reactions_and_Interactions/Solutions%2C_Solvation%2C_and_Dissociation.

[177] Lukas Sommer, Florian Stock, **Solis-Vasquez, Leonardo**, and Andreas Koch. *EPHoS: Evaluation of Programming Models for Heterogeneous Systems*. Berlin, Germany: German Association of the Automotive Industry (VDA: Verband der Automobilindustrie), 2019. URL: https://www.vda.de/de/services/Publikationen/fat-schriftenreihe-317.html.

[178] Lukas Sommer, Florian Stock, **Solis-Vasquez, Leonardo**, and Andreas Koch. "Work-in-Progress: DAPHNE - An Automotive Benchmark Suite for Parallel Programming Models on Embedded Heterogeneous Platforms." In: *Proceedings of the International Conference on Embedded Software (EMSOFT)*. New York, NY, USA: ACM, 2019. DOI: 10.1145/3349568.3351547.

[179] Sérgio F. Sousa, Pedro A. Fernandes, and Maria J. Ramos. "Protein-ligand docking: Current status and future challenges." In: *Journal of Proteins: Structure, Function, and Bioinformatics* 65.1 (2006), pp. 15–26. DOI: 10.1002/prot.21082.

[180] John E. Stone, Michael J. Hallock, James C. Phillips, Joseph R. Peterson, Zaida Luthey-Schulten, and Klaus Schulten. "Evaluation of Emerging Energy-Efficient Heterogeneous Computing

Platforms for Biomolecular and Cellular Simulation Workloads." In: *Int. Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Chicago, IL, USA: IEEE, 2016. DOI: 10.1109/IPDPSW.2016.130.

[181] Bharat Sukhwani and Martin C. Herbordt. "Acceleration of a production rigid molecule docking code." In: *Proceedings of the International Conference on Field Programmable Logic and Applications*. Heidelberg, Germany: IEEE, 2008. DOI: 10.1109/FPL.2008.4629955.

[182] Bharat Sukhwani and Martin C. Herbordt. "GPU Acceleration of a Production Molecular Docking Code." In: *Proceedings of the 2nd Workshop on General Purpose Processing on Graphics Processing Units*. Washington, D.C., USA: ACM, 2009. DOI: 10.1145/1513895.1513898.

[183] Bharat Sukhwani and Martin C. Herbordt. "FPGA acceleration of rigid-molecule docking codes." In: *Journal of IET Computers Digital Techniques* 4.3 (2010), pp. 184 –195. DOI: 10.1049/iet-cdt.2009.0013.

[184] *Summit - Oak Ridge National Laboratory's 200 petaflop supercomputer*. Last accessed: April 30, 2019. URL: https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit.

[185] Swiss Institute of Bioinformatics. *Directory of computer-aided Drug Design tools*. Last accessed: March 30, 2019. URL: https://www.click2drug.org.

[186] Embedded Systems TU Darmstadt and Applications (ESA) Group. *OCLADock-FPGA: OpenCL Accelerated Molecular Docking on FPGAs*. Last accessed: May 30, 2019. URL: https://git.esa.informatik.tu-darmstadt.de/docking/ocladock-fpga.

[187] Embedded Systems TU Darmstadt and Applications (ESA) Group. *OCLADock: OpenCL Accelerated Molecular Docking*. Last accessed: May 30, 2019. URL: https://git.esa.informatik.tu-darmstadt.de/docking/ocladock.

[188] Jorge Tavares, Salma Mesmoudi, and El-Ghazali Talbi. "On the Efficiency of Local Search Methods for the Molecular Docking Problem." In: *Proceedings of the European Conference on Evolutionary Computation, Machine Learning and Data Mining in Bioinformatics (EvoBIO)*. Tübingen, Germany: Springer, 2009. DOI: 10.1007/978-3-642-01184-9_10.

[189] The University of Texas Austin: Texas Advanced Computing Center. *Project Catapult: A Reconfigurable Architecture For Large Scale Machine Learning*. Last accessed: April 30, 2019. URL: https://www.tacc.utexas.edu/systems/catapult.

[190] CORPORATE The MPI Forum. "MPI: A Message Passing Interface." In: *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*. Portland, Oregon, USA: ACM, 1993. DOI: 10.1145/169627.169855.

[191] *The Modern View of Atomic Structure*. Last accessed: July 30, 2019. URL: https://chem.libretexts.org/Bookshelves/General_Chemistry/Map%3A_Chemistry_-_The_Central_Science_(Brown_et_al.)/02._Atoms%2C_Molecules%2C_and_Ions/2.3%3A_The_Modern_View_of_Atomic_Structure.

[192] *The Official UCSF DOCK Web-site*. Last accessed: May 30, 2019. URL: http://dock.compbio.ucsf.edu.

[193] René Thomsen and Mikael H. Christensen. "MolDock: A New Technique for High-Accuracy Molecular Docking." In: *Journal of Medicinal Chemistry* 49.11 (2006), pp. 3315 –3321. DOI: 10.1021/jm051197e.

[194] University of Toronto. *LegUp High-Level Synthesis*. Last accessed: July 30, 2019. URL: http://legup.eecg.utoronto.ca.

[195] Oleg Trott and Arthur J. Olson. "AutoDock Vina: Improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading." In: *Journal of Computational Chemistry* 31.2 (2010), pp. 455–461. DOI: 10.1002/jcc.21334.

[196] *Turbostat - System Manager's Manual*. Last accessed: March 30, 2019. URL: https://www.linux.org/docs/man8/turbostat.html.

[197] Tom Van Court, Yongfeng Gu, and Martin C. Herbordt. "FPGA acceleration of rigid molecule interactions." In: *Proceedings of the 12th Annual Symposium on Field-Programmable Custom Computing Machines*. Napa, CA, USA: IEEE, 2004. DOI: 10.1109/FCCM.2004.33.

[198] Tom Van Court, Yongfeng Gu, Vikas Mundada, and Martin Herbordt. "Rigid Molecule Docking: FPGA Reconfiguration for Alternative Force Laws." In: *Journal on Advances in Signal Processing (EURASIP)* 2006.1 (2006), p. 097950. DOI: 10.1155/ASP/2006/97950.

[199] Robert Vassar, Dora M. Kovacs, Riqiang Yan, and Philip C. Wong. "The β-Secretase Enzyme BACE in Health and Alzheimer's Disease: Regulation, Cell Biology, Function, and Therapeutic Potential." In: *Journal of Neuroscience* 29.41 (2009), pp. 12787–12794. DOI: 10.1523/JNEUROSCI.3657-09.2009.

[200] *Vivado Design Suite - HLx Editions*. Last accessed: March 30, 2019. URL: https://www.xilinx.com/products/design-tools/vivado.html.

[201] Gongyu Wang, Herman Lam, Alan George, and Glen Edwards. “Performance and productivity evaluation of hybrid-threading HLS versus HDLs.” In: *Proceedings of the High Performance Extreme Computing Conference (HPEC)*. Waltham, MA, USA: IEEE, 2015. DOI: 10.1109/HPEC.2015.7322439.

[202] Zhe Wang, Huiyong Sun, Xiaojun Yao, Dan Li, Lei Xu, Youyong Li, Sheng Tian, and Tingjun Hou. “Comprehensive evaluation of ten docking programs on a diverse set of protein–ligand complexes: the prediction accuracy of sampling power and scoring power.” In: *Journal of Physical Chemistry Chemical Physics* 18.18 (2016), pp. 12964–12975. DOI: 10.1039/C6CP01555G.

[203] *Welcome to the AutoDockFR Home Page*. Last accessed: July 30, 2019. URL: http://adfr.scripps.edu/AutoDockFR/adfr.html.

[204] Dennis Weller, Fabian Oboril, Dimitar Lukarski, Juergen Becker, and Mehdi Tahoori. “Energy Efficient Scientific Computing on FPGAs Using OpenCL.” In: *Proceedings of the 25th International Symposium on Field-Programmable Gate Arrays*. Monterey, California, USA: ACM, 2017. DOI: 10.1145/3020078.3021730.

[205] *What are Amyloid Plaques? - Definition & Significance.* URL: https://study.com/academy/lesson/what-are-amyloid-plaques-definition-lesson.html.

[206] *World Community Grid: Active Research*. Last accessed: March 30, 2019. URL: https://www.worldcommunitygrid.org/research/viewAllProjects.do.

[207] *World Community Grid*. Last accessed: March 30, 2019. URL: https://www.worldcommunitygrid.org.

[208] Qiang Wu, Yajun Ha, Akash Kumar, Shaobo Luo, Ang Li, and Shihab Mohamed. “A heterogeneous platform with GPU and FPGA for power efficient high performance computing.” In: *Proceedings of the International Symposium on Integrated Circuits (ISIC)*. Singapore, Singapore: IEEE, 2014. DOI: 10.1109/ISICIR.2014.7029447.

[209] *Xilinx SDAccel Forum*. Last accessed: May 30, 2019. URL: https://forums.xilinx.com/t5/SDAccel/SDAccel-OpenCL-examples-with-non-blocking-pipe-functions/td-p/912707.

[210] Xilinx. *SDAccel: Enabling Hardware-Accelerated Software*. Last accessed: March 30, 2019. URL: https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.

[211] Xilinx. *SDSoC Programmers Guide*. Last accessed: April 30, 2019. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1278-sdsoc-programmers-guide.pdf.

[212] Xilinx. *Vivado Design Hub - Designing with IP*. Last accessed: July 30, 2019. URL: https://www.xilinx.com/support/documentation-navigation/design-hubs/dh0003-vivado-designing-with-ip-hub.html.

[213] Xilinx. *Vivado High-Level Synthesis – Accelerates IP Creation by Enabling C, C++ and System C Specifications*. Last accessed: July 30, 2019. URL: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[214] Xilinx. *ZCU102 Evaluation Board: User Guide*. Last accessed: April 30, 2019. URL: https://www.xilinx.com/support/documentation/boards_and_kits/zcu102/ug1182-zcu102-eval-bd.pdf.

[215] Xilinx. *Xcell: The Quarterly Journal for Programmable Logic Users*. Last accessed: July 30, 2019. 1999. URL: https://www.xilinx.com/publications/archives/xcell/Xcell32.pdf.

[216] Umesh Yadava. "Search algorithms and scoring methods in protein-ligand docking." In: *International Journal of Endocrinology & Metabolism* 6.6 (2018). URL: https://medcraveonline.com/article?id=16934&fbclid=IwAR3fsVeMH21u2Y0a_LV8g1KZAEfDskn3A_p0QMbspeoPEzapIrqlZO5yjNU.

[217] Charlene Yang et al. "An Empirical Roofline Methodology for Quantitatively Assessing Performance Portability." In: *Proceedings of the International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. Dallas, TX, USA: IEEE, 2018. DOI: 10.1109/P3HPC.2018.00005.

[218] Chen Yang, Jiayi Sheng, Rushi Patel, Ahmed Sanaullah, Vipin Sachdeva, and Martin C. Herbordt. "OpenCL for HPC with FPGAs: Case study in molecular electrostatics." In: *Proceedings of the 2017 High Performance Extreme Computing Conference (HPEC)*. Waltham, MA, USA: IEEE, 2017. DOI: 10.1109/HPEC.2017.8091078.

[219] Andrei K. Yudin. "Macrocycles: lessons from the distant past, recent developments, and future directions." In: *Journal of Chemical Science* 6.1 (2015), pp. 30–49. DOI: 10.1039/C4SC03089C.

[220] Elizabeth Yuriev and Paul A. Ramsland. "Latest developments in molecular docking: 2010-2011 in review." In: *Journal of Molecular Recognition* 26.5 (2013), pp. 215–239. DOI: 10.1002/jmr.2266.

[221] Matthew D. Zeiler. "ADADELTA: An Adaptive Learning Rate Method." In: *arXiv* abs/1212.5701 (2012). URL: https://arxiv.org/abs/1212.5701.

[222] Yao Zhang, Mark Sinclair, and Andrew A. Chien. "Improving Performance Portability in OpenCL Programs." In: *Proceedings of the International Supercomputing Conference (ISC)*. Leipzig, Germany: Springer, 2013. DOI: 10.1007/978-3-642-38750-0_11.

[223] Tuowen Zhao, Samuel Williams, Mary Hall, and Hans Johansen. "Delivering Performance-Portable Stencil Computations on CPUs and GPUs Using Bricks." In: *Proceedings of the International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. Dallas, TX, USA: IEEE, 2018. DOI: 10.1109/P3HPC.2018.00009.

[224] Jinghui Zhong, Xiaomin Hu, Jun Zhang, and Min Gu. "Comparison of Performance between Different Selection Strategies on Simple Genetic Algorithms." In: *Proceedings of the International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce*. Vienna, Austria: IEEE, 2005. DOI: 10.1109/CIMCA.2005.1631619.

[225] Zhiyong Zhou, Anthony K. Felts, Richard A. Friesner, and Ronald M. Levy. "Comparative performance of several flexible docking programs and scoring functions: enrichment studies for a diverse set of pharmaceutically relevant targets." In: *Journal of Chemical Information and Modeling* 47.4 (2007), pp. 1599–608. DOI: 10.1021/ci7000346.

[226] Hamid R. Zohouri, Artur Podobas, and Satoshi Matsuoka. "Combined Spatial and Temporal Blocking for High-Performance Stencil Computation on FPGAs Using OpenCL." In: *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Monterey, CA, USA: ACM, 2018. DOI: 10.1145/3174243.3174248.

[227] Hamid. R. Zohouri, Naoya Maruyama, Aaron Smith, Motohiko Matsuda, and Satoshi Matsuoka. "Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Salt Lake City, UT, USA: IEEE, 2016. DOI: 10.1109/SC.2016.34.

[228] Zsolt Zsoldos, Darryl Reid, Aniko Simon, Sayyed B. Sadjad, and A. Peter Johnson. "eHiTS: A new fast, exhaustive flexible ligand docking system." In: *Journal of Molecular Graphics and Modelling* 26.1 (2007), pp. 198–212. DOI: 10.1016/j.jmgm.2006.06.002.

[229] Top500 project. *The Green500 list - November 2018*. Last accessed: April 30, 2019. URL: https://www.top500.org/green500/lists/2018/11.

[230] Top500 project. *The Top500 list - Highlights*. Last accessed: April 30, 2019. URL: https://www.top500.org/lists/2018/11/highs.

[231] Top500 project. *The Top500 list - November 2018*. Last accessed: April 30, 2019. URL: https://www.top500.org/lists/2018/11.

[232] Top500 project. *The Top500 list - The LINPACK Benchmark*. Last accessed: April 30, 2019. URL: https://www.top500.org/project/linpack.

[233] **Solis-Vasquez, Leonardo** and Andreas Koch. "A Performance and Energy Evaluation of OpenCL-accelerated Molecular Docking." In: *Proceedings of the 5th International Workshop on OpenCL (IWOCL)*. Toronto, ON, Canada: ACM, 2017. DOI: 10.1145/3078155.3078167.

[234] **Solis-Vasquez, Leonardo** and Andreas Koch. "A Case Study in Using OpenCL on FPGAs: Creating an Open-Source Accelerator of the AutoDock Molecular Docking Software." In: *Proceedings of the 5th International Workshop on FPGAs for Software Programmers (FSP)*. Dublin, Ireland: VDE VERLAG, 2018. URL: https://ieeexplore.ieee.org/document/8470463.

[235] **Solis-Vasquez, Leonardo**, Diogo Santos-Martins, Andreas Koch, and Stefano Forli. "Performance Analysis of Molecular Docking in OpenCL: A Case Study of AutoDock enhanced with Gradients." In: *Submitted to the 34th International Parallel and Distributed Processing Symposium (IPDPS)*. Submitted, 2019.

[236] **Solis-Vasquez, Leonardo**, Diogo Santos-Martins, Andreas Koch, and Stefano Forli. "Evaluating the Energy Efficiency of OpenCL-accelerated AutoDock Molecular Docking." In: *Submitted to the 28th Euromicro International Conference on Parallel, Distributed, and Network-based Processing (PDP)*. Submitted, 2020.