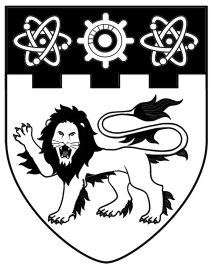

NANYANG
TECHNOLOGICAL
UNIVERSITY

OPENCL ACCELERATED MOLECULAR DOCKING WITH HISTORICAL GENETIC ALGORITHM

EKA ANTONIUS KURNIAWAN
School of Computer Engineering
2013

OpenCL Accelerated Molecular Docking with Historical Genetic Algorithm

by

Eka Antonius Kurniawan

Submitted to the Department of Computer Engineering
in Partial Fulfilment of the Requirements for the
Degree of Master of Science
in Bioinformatics

ABSTRACT

Computational tools for molecular docking are developed to help researchers to reduce discovery time and counterattack Eroom's law of rational drug design process. They can be achieved by automatically correlating target protein with ligand or drug candidate to form a stable complex. Two components in molecular docking are energy calculation and conformational search. For energy calculation, this dissertation adopts semiempirical free energy function from AutoDock 4. Whereas for conformational search, this dissertation implements new heuristic optimisation method called historical genetic algorithm. The implementation (called hppNeuroDock) is a building up of the two components using Python programming language.

This dissertation also introduces heterogeneous parallel programming by integrating Python and OpenCL. The implementation of Python-OpenCL on top of hppNeuroDock has been tested successfully. Using HIV-1 protease for the protein and Indinavir as the ligand, hppNeuroDock produces ligand conformation with RMSD 1.53 Å deviated from AutoDock 4. Which is still lower than the threshold of 2.0 Å. It also produces negative docking energy of -14.27 kcal/mol compared to -15.66 kcal/mol produced by AutoDock 4. Benchmark result shows that parallel Python improves the runtime up to 477 times faster than sequential Python.

Acknowledgment

I want to thank Prof. Kwoh for his help on matching my skill and interest to this work, supervising me for the past one year and reviewing this dissertation. I feel grateful that he also introduces me to one of his Ph.D. students, Ouyang Xuchang. I want to thank Xuchang for his help on molecular docking especially AutoDock and reviewing this dissertation.

I want to thank Prof. Hwu from University of Illinois and his team by offering Heterogeneous Parallel Programming course through Coursera from which I have learned parallel APIs (CUDA, OpenMP, MPI and especially OpenCL), GPU architecture and memory model for performance considerations.

I want to thank my family for their help and support during my study.

Contents

1 Introduction 8

1.1 Molecular Docking 9

1.2 AutoDock 4 11

1.3 Python 13

1.4 OpenCL 13

2 Theories 17

2.1 Semiempirical Free Energy Function 17

2.2 Historical Genetic Algorithm 18

2.3 Root-mean Square Deviation (RMSD) 21

2.4 Three-dimensional Linear Interpolation 21

2.5 Quaternion 23

2.6 Random Number Generator 24

3 hppNeuroDock Implementation 26

3.1 Python Implementation 26

3.2 Python-OpenCL Implementation 28

3.3 Execution Procedures 32

3.4 Benchmarking Script 35

3.5 Analysis Script 36

3.6 Source Code 36

4 Docking Result 37

4.1 Numerical 37

4.2 Visual 37

5 Benchmark 40

5.1 Hardware and Software 40

5.2 Initialisation Time 41

5.3 Runtime 43

5.4 Speedup 44

6 Conclusions and Future Works 46

Bibliography 47

List of Figures

Figure 1-1: Eroom's law shows number of drug discovered per billion USD spending over the course of sixty years is projected as a linear fall off (Scannell et al., 2012). 8

Figure 1-2: AutoDock flow includes protein-ligand preparation using AutoDockTools, free energy precalculation using AutoGrid, molecular docking using AutoDock and finally, docking result analysis using AutoDockTools (Morris et al., 2010)....11

Figure 1-3: OpenCL memory model showing the access speed from the slowest to the fastest: global memory, local memory and private memory (Khronos Group, 2012). 15

Figure 1-4: OpenCL program creation starting from compiling programs, creating and transferring kernels/images/buffers into device memory, up to executing commands from command queue (Khronos Group, 2012) 15

Figure 1-5: OpenGL and OpenCL interoperability for graphics processing and visualisation combined with general-purpose applications for the engines (Khronos Group, 2012)....16

Figure 2-1: Molecular Interaction Definitions (Huey et al., 2007) 17

Figure 2-2: Nomad minus settler minimum free energy (in kcal/mol) over 200 samples. Positive values indicate the improvement made by having settler population. 20

Figure 2-3: An atom at coordinate u, v, w in a grid box from coordinate $u0, v0, w0$ to $u1, v1, w1$. Where $p0u$ plus $p1u$ is the length from $u0$ to $u1$. It also applies to v and w directions. 22

Figure 2-4: 16-bit Linear Feedback Shift Register (LFSR) schematic....24

Figure 2-5: Random numbers generated using 16-bit LFSR....25

Figure 3-1: Python class diagram contains three main classes: `NeuroDock`, `Dock` and `GeneticAlgorithm`. 26

Figure 3-2: `DockOpenCL` class diagram inherited from `Dock` class. 28

Figure 3-3: `GeneticAlgorithmOpenCL` class diagram inherited from `GeneticAlgorithm` class. 29

Figure 3-4: hppNeuroDock execution flow providing information about nomad/settler and Python/OpenCL portions. 30

Figure 3-5: Parallel data structure showing: 1) branch rotation sequence data structure with atom IDs in row-wise, 2) population data structure with DNA in row-wise, 3) atom coordinates data structure with each atom coordinate (x, y, z) in row-wise and 4) maps data structure with maps information in row-wise. 31

Figure 3-6: Annotated Git tagging with description and source code. The source code can be browsed online or downloaded into local computer using either `zip` or `tar.gz` files...36

Figure 4-1: Indinavir ligand conformation result of hppNeuroDock (in cyan) and AutoDock 4 (in magenta) with RMSD of 1.53 Å. 38

Figure 4-2: Binding mode of Indinavir ligand (from hppNeuroDock) to HIV-1 protein in secondary structure. 39

Figure 4-3: Binding mode of Indinavir ligand (from hppNeuroDock) to HIV-1 protein in surface view. Flexible part of the protein is shown in purple, located slightly below the ligand. 39

Figure 5-1: Video card survey taken from MGLTools website showing that OpenCL supports close to 93% of the graphic card used. 40

Figure 5-2: Initialisation time comparison between sequential Python, parallel Python-OpenCL using GPU and parallel Python-OpenCL using CPU in Box Plot. 42

Figure 5-3: Combined sequential Python, parallel Python-OpenCL using GPU and parallel Python-OpenCL using CPU runtimes in 500 generations based on different population size. 44

List of Tables

Table 1-1: Comparison between CUDA and OpenCL terminologies and functions 14

Table 2-1: Comparison between nomad and settler parameters showing that nomad population has more vibrant individuals compared to settler population. 20

Table 3-1: Files needed to run hppNeuroDock grouped as Python codes, input files and scripts. 33

Table 3-2: New parameters introduced in hppNeuroDock with options, default value and description of the keywords and the options. 34

Table 5-1: Hardware used for benchmark on Apple MacBook Pro Retina (Mid 2012) laptop. 40

Table 5-2: Software used for benchmark on Apple OS X version 10.8.4 Mountain Lion. 40

Table 5-3: Intel Core i7-3615QM detail specifications (Intel, n.d.) 41

Table 5-4: NVIDIA GeForce GT 650M detail specifications (NVIDIA, n.d.) (Intel, n.d.) 41

Table 5-5: Main Memory detail specifications (Intel, n.d.) 41

Table 5-6: Sequential Python initialisation time (in second) using CPU based on different population sizes and numbers of generations. 41

Table 5-7: Parallel Python-OpenCL initialisation time (in second) using GPU based on different population sizes and numbers of generations. 42

Table 5-8: Parallel Python-OpenCL initialisation time (in second) using CPU based on different population sizes and numbers of generations. 42

Table 5-9: Sequential Python runtime (in seconds) using CPU based on different population sizes and numbers of generations. 43

Table 5-10: Parallel Python-OpenCL runtime (in seconds) using GPU based on different population sizes and numbers of generations. 43

Table 5-11: Parallel Python-OpenCL runtime (in seconds) using CPU based on different population sizes and numbers of generations. 43

Table 5-12: Sequential Python over parallel Python-OpenCL GPU speedup based on different population sizes and numbers of generations. 44

Table 5-13: Sequential Python over parallel Python-OpenCL CPU speedup based on different population sizes and numbers of generations. 44

Table 5-14: Parallel Python-OpenCL using GPU over CPU speedup based on different population sizes and numbers of generations. 45

Summary

This dissertation focuses on implementing molecular docking tool to help scientists to discover new drug efficiently. The main contributions of this dissertation are implementing AutoDock 4 semiempirical energy function in Python (which originally written in C), implementing new algorithm for conformational search called historical genetic algorithm and implementing heterogeneous parallel program by integrating Python and OpenCL.

This dissertation presents the introduction to molecular docking with additional view on the real implementation, AutoDock 4. It emphasises on two important parts: free energy calculation and conformational search. Then, it discusses the importance of Python in scientific community with OpenCL as the accelerator. This dissertation also presents theories that are required for the implementation. It includes computational biochemistry, genetic algorithm and mathematical methods on geometry and random number.

For the Python-OpenCL implementation, this dissertation presents class diagram, execution flow, parallel data structure, the inputs required and the outputs produced. It also provides design considerations, implementation limitation and link to acquire the source code. The implementation result against AutoDock 4 is done both numerically using minimum free energy and RMSD; and visually using three-dimensional molecular viewer. The result is largely acceptable by achieving negative free energy and lower than 2.0 Å RMSD.

Finally, this dissertation provides benchmarking results on initialisation time, runtime and speedup. The data is collected from different population sizes, numbers of generations and acceleration devices. The benchmark shows that parallelisation can improve runtime up to 477 times better than sequential execution.

1 Introduction

Eroom's law in the field of drug discovery is an anagram of Moore's law from silicon technology as it means the opposite (Scannell et al., 2012). It states that fewer and fewer drugs have been found per billion dollars R&D spending as shown in Figure 1-1. It also states that the discoveries decline by half per course of approximately nine years.

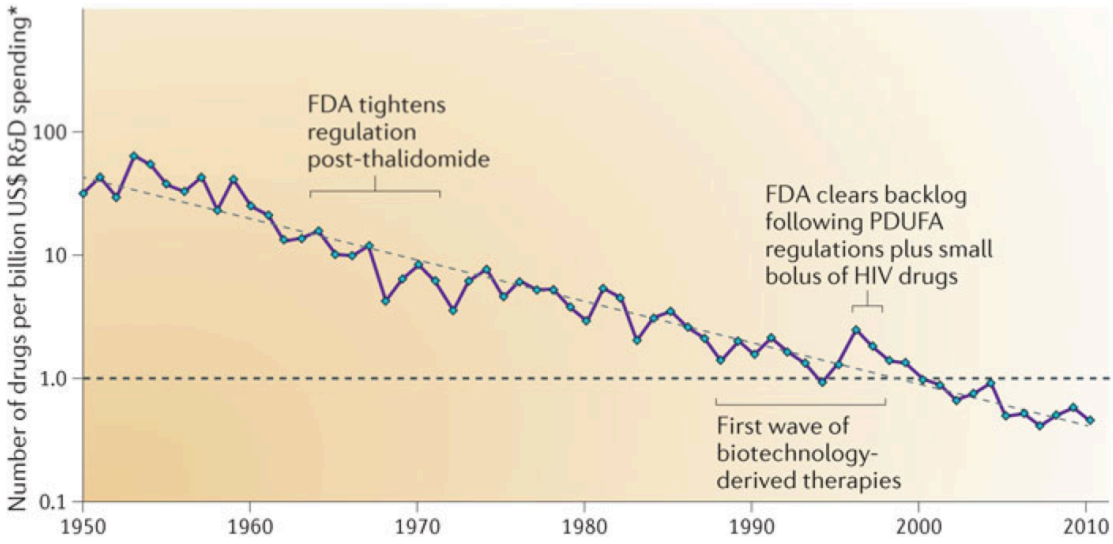

Figure 1-1: Eroom's law shows number of drug discovered per billion USD spending over the course of sixty years is projected as a linear fall off (Scannell et al., 2012).

This dissertation focuses on two portions of molecular docking workflow: semiempirical energy calculation and conformational search. The goal of this dissertation is to counterattack Eroom's law by implementing molecular docking tool using Python programming language that is familiar to natural scientists so they can work on their own representation of semiempirical energy function. Also, for computer scientists so they can easily implement new optimisation function for conformational search. This dissertation also introduces heterogeneous parallel programming using OpenCL to significantly improve the runtime while supporting multiple parallel devices.

The concepts and theories encompassing molecular docking, software engineering and mathematics are presented in detail as the foundation to the implementation. The correctness of the implementation is analysed using free energy calculation and RMSD against AutoDock 4 (http://autodock.scripps.edu) (Morris et al., 2009). Design considerations, strategies to overcome issues and limitations of the implementation are also part of this dissertation.

Main contributions of this dissertation are implementing AutoDock 4 semiempirical energy function in Python (which originally written in C), implementing new algorithm for conformational search called historical genetic algorithm and implementing heterogeneous parallel program by integrating Python and OpenCL.

1.1 Molecular Docking

Molecular modelling integrates theory in chemistry with empirical result for both physicochemical and biochemical by gathering information about examined model, analysing semiempirical result (both empirical result and theoretical estimation), comprehending the outcome in microscopic and macroscopic extents, and finally, presenting the differences (Tsai, 2002).

The workflow of molecular modelling includes acquiring and visualising molecules, energy calculation, conformational search, and analysing molecular parameters and interactions (Tsai, 2002). The well-known hosting sites for three-dimensional molecular structures are Protein Data Bank (PDB) (http://www.rcsb.org/pdb/) and Cambridge Crystallographic Data Centre (CCDC) (http://www.ccdc.cam.ac.uk). For small molecules, PubChem (http://pubchem.ncbi.nlm.nih.gov) (Bolton et al., 2008) from NCBI is more appropriate. The file format downloaded from the sites can be different. Tool like Open Babel (http://openbabel.org) (O'Boyle et al., 2011) is useful to convert the files from one format to another. To visualise three-dimensional molecule, AutoDock 4 uses Python Molecular Viewer (PMV) (Sanner, 1999) shipped under MGLTools (http://mgltools.scripps.edu). Other than PMV, PyMOL (http://www.pymol.org) from Schrödinger is also available.

For energy calculation, this dissertation uses molecular mechanics. Molecular mechanics uses empirical and semiempirical methods in contrast to quantum mechanics (Tsai, 2002). It provides efficient computation while still producing accurate outcome by using experimentation results from small molecules to predict the behaviours of bigger molecules. Despite the efficiency, the basis of molecular mechanics, empirical force field, produces only relative energy among molecules, not in the form of absolute quantities. The best binding mode between molecules is defined to be a complex with lowest estimated free energy. Conformations with negative free energy attract each other to form a firm complex. Free energy is combination of covalent and non-covalent interactions of the molecules. They include bond, angle and torsion energies; combined with van der Waals, electrostatic,

desolvation and hydrogen bond energies. Following AutoDock 4, this dissertation implements only non-covalent interaction.

Conformational search is an optimisation problem that minimise free energy in respect to the molecular geometries (Tsai, 2002) as shown below (Ouyang & Kwoh, 2012).

$$minimise\ E(s);\ s = (x, y, z, a, b, c, d, t_1, t_2, \dots, t_n) \quad \text{Eq. 1-1}$$

Where $E(s)$ is free energy function with molecular geometries as the domain (s) and free energy in kcal/mol as the range. Molecular geometries (s) consists of translational coordinate (x, y, z), rotational information in quaternion (a, b, c, d) and torsional rotations $(t_1, t_2, \dots, t_n)$.

Searching in three-dimensional space with torsional degree of freedom for all possible molecular geometries is approaching infinite. Genetic algorithm is one of heuristic methods to solve the problem (Westhead et al., 1997) (Hou et al., 1999). The algorithm is based on evolution accruing in nature (Shiffman, 2012). Each individual in a population has capability to inherit its traits to the next generation. The chance to be able to inherit depends on the fitness of the individual. Individuals who are fitter have higher probability to survive and therefore, inherit their traits to their successors as opposed to the individuals with lower score. The process is called natural selection. Organisms that reproduce sexually undergo extra step called crossover. The gametes from each parent interchange their traits resulting a new individual with different trait from its parents. The difference could occur because of mutation happening during the process of inheritance.

After acquiring lowest free energy from the population over number of generations, both energy and the conformation of the molecule are then analysed. Conformation difference is calculated using root-mean square deviation (RMSD) from each pair of atoms in the molecule (Huey et al., 2007) (Cole et al., 2005). An acceptable conformational result *in silico* is defined to be the one having RMSD lower than 2.0 Å against the one find *in vivo* or *in vitro*.

Molecular docking is a subset of molecular modelling that produces, alters, computes and estimates molecular formation together with estimating related parameters *in silico*. The molecular docking terminology used throughout this dissertation follows that given in (Tsai, 2002). Computational molecular docking is an automatic way to predict the interaction of protein-ligand complex. Protein that acting as the receptor is considerably larger molecule

compared to ligand that is commonly found in the form of drug. The affinity result between two molecules can lead scientists to the new drug discovery for the specific protein target.

1.2 AutoDock 4

AutoDock 4 is an open source automated molecular docking tool developed by The Scripps Research Institute (TSRI) (Morris et al., 2009). It uses semiempirical energy function for energy calculation and Lamarckian genetic algorithm as one of algorithms used for conformational search. The methods have been used and proven since AutoDock 3 (Morris et al., 1998). Figure 1-2 shows the entire process when using AutoDock suite.

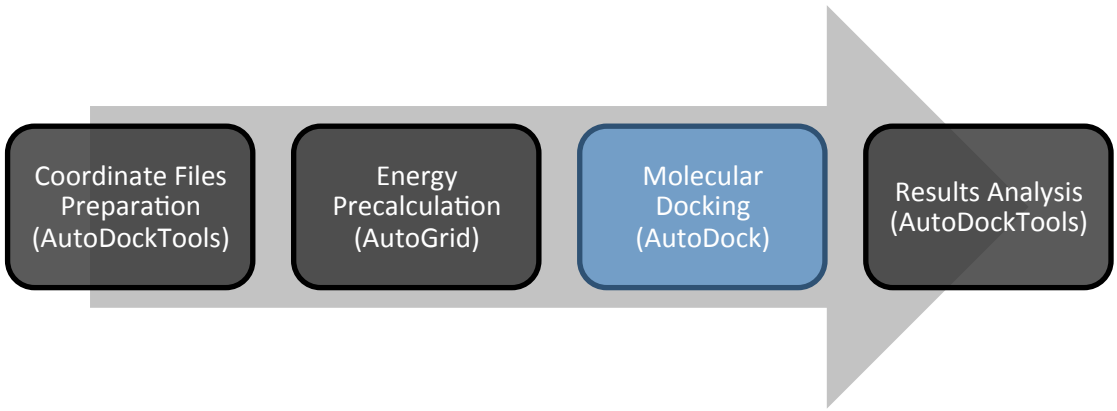

Figure 1-2: AutoDock flow includes protein-ligand preparation using AutoDockTools, free energy precalculation using AutoGrid, molecular docking using AutoDock and finally, docking result analysis using AutoDockTools (Morris et al., 2010)

AutoDockTools (ADT) (Sanner, 1999) is a GUI tool to prepare, run and analyse molecules for AutoDock. It is integrated with Python Molecular Viewer (PMV) to visualise the molecules in three-dimensional form. Both ADT and PMV are bundled together under MGLTools. When setting up molecules, ADT can help to edit PDB file like adding/removing necessary/redundant atoms, generate torsion tree for ligand, define flexible and rigid portion of protein, convert the modified PDB input files into PDBQT files, prepare both grid parameter file (`.gpf`) and docking parameter file (`.dpf`), and run AutoGrid followed by AutoDock (Huey & Morris, 2008). In molecular docking, receptor is assigned to be the rigid part with optional flexible part whereas ligand is both translationally and conformationally flexible. The support for protein flexible part is newly added in AutoDock 4 (Morris et al., 2009). When performing result analysis, ADT is able to visualise docked conformation, perform conformations clustering based on RMSD and provide energy landscape (Morris et al., 2009).

AutoGrid and AutoDock are the engines distributed under a single compressed file called AutoDock. AutoDock is the main focus of this dissertation but AutoGrid is still important. AutoGrid produces the input parameters and necessary files like map files for AutoDock (Huey & Morris, 2008). When configuring grid parameter file (`.gpf`), ADT requires user to define spacing segment between grid points. The grid segmentation of rigid protein is used by AutoDock to calculate intermolecular energy. The default is set to 0.375 Å (close to the distance of 1/4 carbon-carbon single bond) but it can be extended up to 1.0 Å. It also requires user to specify number of grid points in three-dimensional axis. The search area is calculated as the number of grid points multiplied by the space per segment. Segment space and grid points are stored inside grid parameter file that is required when executing AutoGrid.

As the result, AutoGrid produces grid data file (`.fld`) consists of grid information from user and map files (`.map`) for all atom types found in the ligand plus two extra maps for electrostatic and desolvation (Huey & Morris, 2008). Individually, map file contains pre-calculated potential energy look-up table that is essential for AutoDock.

AutoDock portion is computational intensive. It calculates semiempirical free energy (Huey et al., 2007) and performs conformational search to find the best fit of ligand to target protein (Huey & Morris, 2008). It has more parameters to set compared to AutoGrid. The parameters are stored inside docking parameter file (`.dpf`). Among them there are settings to tell which map files to use, where is the centre of ligand, which are the flexible portions of the protein, what method of genetic algorithm to use and how many iterations to run. This dissertation introduces additional parameters to enable or disable accelerator, to select different target device and to set genetic algorithm properties like community size, population size and number of generations.

Concretely, the traits in the form of DNA are implemented as molecular geometries. Genotypically, they include translational genes of rigid body (in three-dimensional axis), rotational genes of rigid body (in quaternion) and torsional genes of branches (in radian) (Morris et al., 1998). The genes of each individual can be translated into phenotypic trait as molecular coordinates. Then, the fitness of the individual can be accessed using molecular mechanics energy calculation. The lower the free energy represents the fitter the individual.

When implementing genetic algorithm, there are many modifications have been done to the original Darwinian natural selection. AutoDock 4 has two genetic algorithms to choose form,

traditional Darwinian genetic algorithm and Lamarckian genetic algorithm (Huey & Morris, 2008). Lamarckian genetic algorithm integrates local search optimisation method to the traditional Darwinian genetic algorithm (Morris et al., 1998). Instead of using either of them, this dissertation introduces new model of genetic algorithm called historical genetic algorithm or nomad-settler genetic algorithm. Decision made is to reduce complexity when implementing it on different parallel accelerators.

Each result of AutoGrid and AutoDock is saved inside grid log file (`.glg`) and docking log file (`.dlg`) respectively. Docking log file consists of all energy calculation information, atomic bounding, lowest docking and binding energies, atom coordinates of candidate poses, clustering histogram, RMSD table, entropy information, statistical mechanical analysis and total running time (Huey & Morris, 2008).

1.3 Python

Python (Python Software Foundation, 2013) is a free and open source programming language with focus on productivity and system integrity. It is a notable programming languages among scientists by combining scientific requirements through multiple packages like numerical computing (NumPy), scientific and statistical methods (SciPy), graph plotting (matplotlib), functional programming and heterogeneous computing. Python packages allow the core to be written in C/C++. The packages that are written in C/C++ get benefit from speed improvement. It is also easier to learn compared to C programming language. On top of that, Python supports object-oriented programming (OOP) as the foundation to build large application.

1.4 OpenCL

OpenCL or Open Computing Language (Khronos Group, 2013) is an open and royalty-free standard for parallel programming. It runs on multiple devices from Intel, AMD, NVIDIA and Altera FPGA. It is an emerging intersection between central processing unit (CPU) and graphics processing unit (GPU) with multiprocessor programming APIs for example OpenMP, data-parallel computing with graphics APIs and shading languages to form a heterogeneous parallel programming framework (Khronos Group, 2012).

GPU that is used to perform general-purpose application like CPU is called General-Purpose computing on Graphics Processing Units (GPGPU) or GPU computing (Owens et al., 2008). OpenCL and CUDA are the frameworks for GPU computing. The reason of choosing OpenCL is that OpenCL performance is comparable to CUDA (Feldman, 2012) while supporting multiple vendors and the interoperability with OpenGL (Open Graphics Library). Table 1-1 shows definition differences between CUDA and OpenCL (Kirk & Hwu, 2010).

Table 1-1: Comparison between CUDA and OpenCL terminologies and functions

CUDA	OpenCL
Host	Host
-	Context (collection of devices)
Device	Device
Streaming Module (SM)	Compute Unit
Streaming Processor (SP)	Processing Element (PE)
Global Memory	Global Memory
Constant Memory	Constant Memory
Shared Memory	Local Memory
Register and Local Memory	Private Memory
Host Program	Host Program
Kernel (Device Program)	Kernel (Device Program)
Grid	NDRange (index space)
Block	Work Group
Thread	Work Item

`threadIdx.x`
`threadIdx.y`
`threadIdx.z`
	`get_local_id(0)`
	`get_local_id(1)`
	`get_local_id(2)`

`blockIdx.x * blockDim.x + threadIdx.x`	`get_global_id(0)`
`blockIdx.x * blockDim.x`	`get_global_size(0)`
`blockDim.x`	`get_local_size(0)`

There are several levels of memory allocation in OpenCL as shown in Figure 1-3 (Khronos Group, 2012) (Kirk & Hwu, 2010). Before the data can be processed using accelerator device, the data need to be transferred from host memory into device global memory. Access from processing element (PE) to global memory is slower compared to local memory. For the data that are repeatedly used, storing it in local memory benefits from faster access. Furthermore, data that is shared among work items within a same workgroup can be accessed efficiently by avoiding traffic congestion to global memory. But, the data in local memory of different workgroup cannot be accessed directly.

Private memory is faster than local memory. Automatic variables are placed in private memory. For rapid access, copying the data from global memory into private memory can improve runtime. Another type of memory is constant memory. Constant memory is not writable. Despite the location, which at the same level as global memory, some devices provide faster access to constant memory compared to global memory.

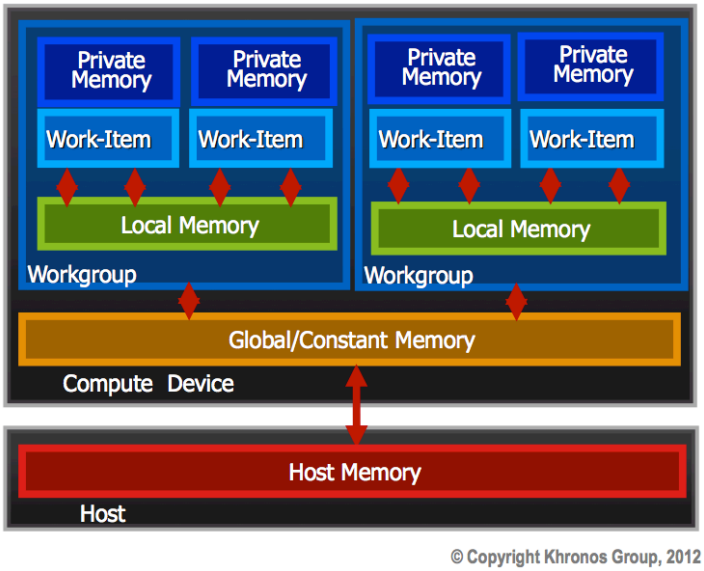

Figure 1-3: OpenCL memory model showing the access speed from the slowest to the fastest: global memory, local memory and private memory (Khronos Group, 2012).

Figure 1-4 shows OpenCL step-by-step program creation (Khronos Group, 2012). For benchmarking, initialisation time is defined as the total time of "program compilation" plus "data and argument creation". Creating data and argument include transferring kernels, images and buffers into device memory. Runtime is counted starting from "sending for execution" portion to "program termination".

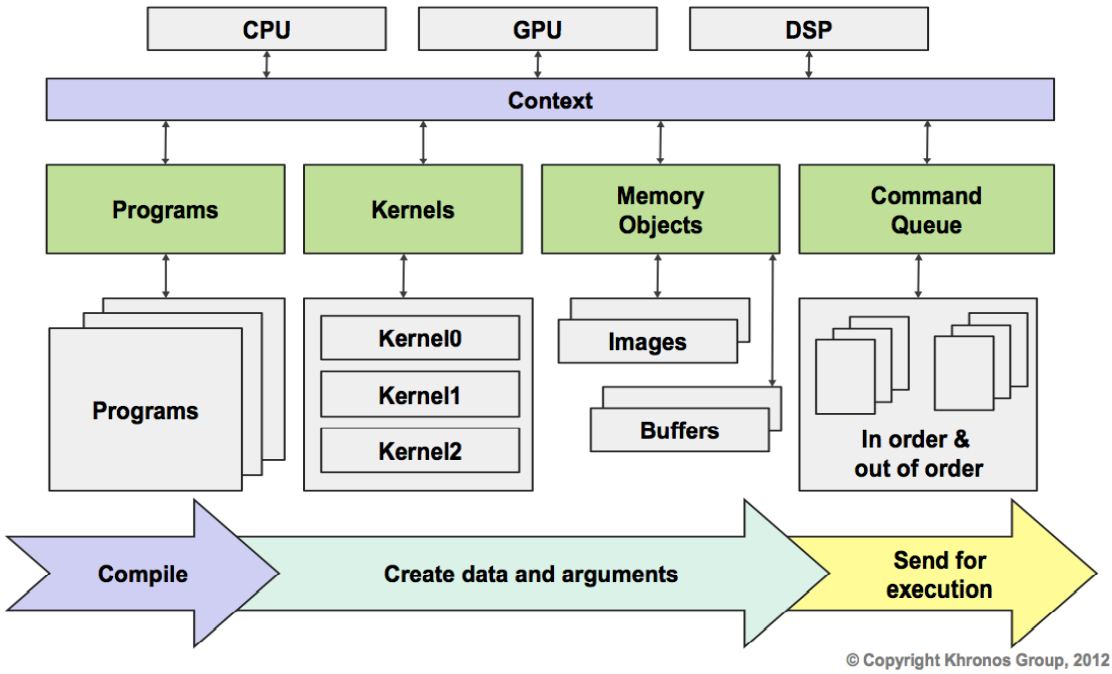

Figure 1-4: OpenCL program creation starting from compiling programs, creating and transferring kernels/images/buffers into device memory, up to executing commands from command queue (Khronos Group, 2012)

Khronos Group, the developer of both OpenCL and OpenGL standards assures efficient data exchange between the two APIs as shown in Figure 1-5 (Khronos Group, 2012).

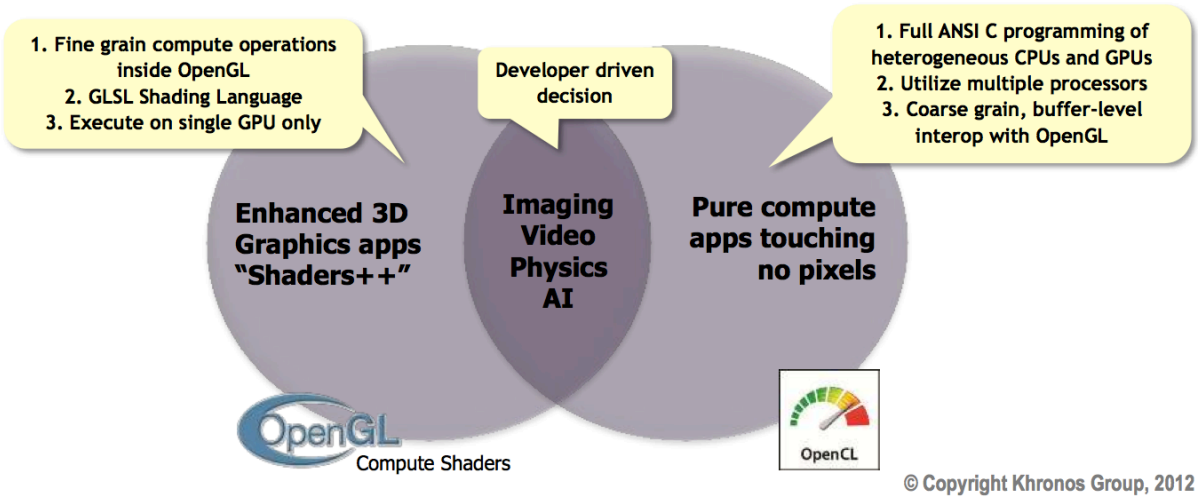

Figure 1-5: OpenGL and OpenCL interoperability for graphics processing and visualisation combined with general-purpose applications for the engines (Khronos Group, 2012).

PyOpenCL (Klöckner et al., 2012) is an open-source wrapper between Python and OpenCL developed under MIT license. It provides a complete implementation of OpenCL functions, running performance as it is developed using C++ and similar to Python, it is leak- and crash-free (Klöckner, 2013a).

2 Theories

2.1 Semiempirical Free Energy Function

Molecular free energy calculation for empirical method is also known as empirical force field (Tsai, 2002). Figure 2-1 depicts bound and unbound interactions in both intermolecular and intramolecular/internal relations of ligand, flexible part of protein and rigid part of protein (Huey et al., 2007).

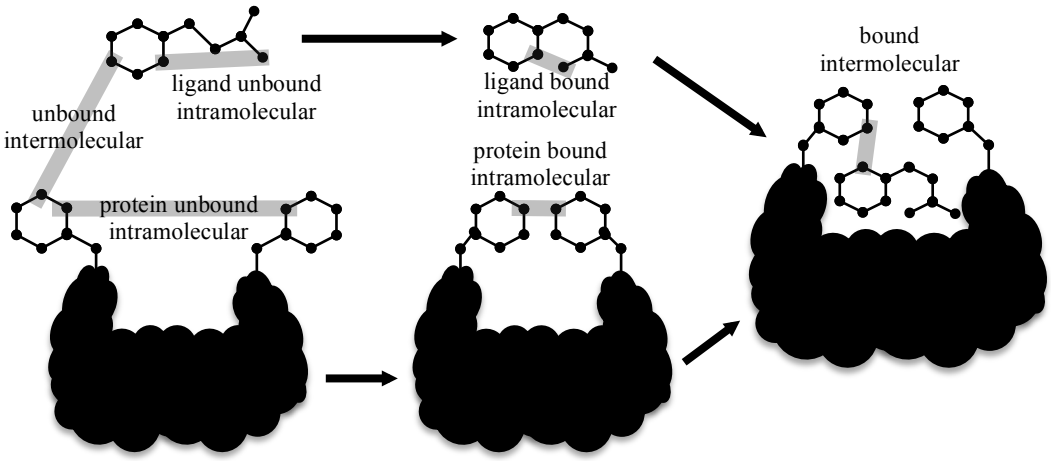

Figure 2-1: Molecular Interaction Definitions (Huey et al., 2007)

There are six pair-wise calculations (V) of AutoDock 4 semiempirical free energy force field (Huey et al., 2007) written as follow.

$$\Delta G = (V_{bound}^{L-L} - V_{unbound}^{L-L}) + (V_{bound}^{P-P} - V_{unbound}^{P-P}) + \left(V_{bound}^{P-L} - V_{unbound}^{P-L} + \Delta S_{conf}\right) \quad \text{Eq. 2-1}$$

ΔS_{conf} is added to compensate torsional entropy lost in binding mode. It is used to penalise the confirmation change brought by rotatable bond. L and P stand for ligand and protein respectively. Individual pair-wise formula is shown below.

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

All Ws are weight functions from experimental result to adjust the over all free energy. The first term is Lennard-Jones 6-12 common potential for van der Waals (dispersion/repulsion) interaction, next term is 10-12 potential, followed by Coulomb's law for charged biochemical molecules (Tsai, 2002) and desolvation potential for solvent protection.

Other than bound and unbound interactions, there are some definitions like intramolecular/internal and intermolecular energies (Morris et al., 1998). Since molecular docking treats protein as whole rigid body with optional flexible chains, intermolecular potential of the rigid body to ligand and the flexible part of receptor has been calculated by AutoGrid and stored in map files. AutoDock only needs to utilise the files as look-up table. If the atom is located inside the grid box, three-dimensional linear interpolation or trilinear interpolation is used to get the approximation free energy at the location. Electrostatics and desolvation have their own maps but van der Waals and hydrogen bond share a same map for each atom type. This first part of energy calculation is referred as intermolecular energy calculation producing intermolecular energy. The second part covers internal ligand, internal flexible chains of receptor and intermolecular interaction of ligand to the flexible chains. They need to be calculated using semiempirical free energy force field (Eq. 2-1) and individual pair-wise (Eq. 2-2) formulas. Energy coefficients and parameters are provided inside "AutoDock Linear Free Energy Model Coefficients and Energetic Parameters" file (`.dat`) (Huey et al., 2007). This second part of energy calculation is referred as internal energy calculation producing internal energy.

Other than the term for free energy, AutoDock 4 introduces two extra terms: docking energy and binding energy. Docking energy is the sum up between intermolecular energy and internal energy whereas binding energy is the result of intermolecular energy plus torsional energy (Huey & Morris, 2008). The torsional energy (ΔS_{conf}) itself is acquired from the multiplication between torsional degrees of freedom (N_{tors}) and torsions free energy coefficient (W_{conf}) (Huey et al., 2007). This dissertation uses docking energy interchangeably with general term of free energy.

2.2 Historical Genetic Algorithm

Method for conformational search introduced in this dissertation is called historical genetic algorithm. It is based on human population history evolved from nomad population that lived from place to place into recent society called settler that occupies certain area permanently.

Driving forces for the transition like fertile land and the invention of agriculture around Neolithic Revolution (National Geographic Society, 2013) are translated into the implementation as population parameters. The idea is similar to epigenetics (Weinhold, 2006) that takes other factors outside DNA sequence to interfere gene function. Nonetheless, the factor of Darwinian natural selection is still prominent. The method is also known as nomad-settler genetic algorithm.

Nomad population that is represented to have higher mutation rate has the ability to explore entire searching space efficiently but it has difficulty to get down into local minimum. After generations of selecting individuals that survive in particular areas, the nomad population is transformed into settler population with lower mutation rate in the hope of reaching local minimum. From a community that consists of multiple populations with different local minima, the one that has the lowest energy is considered as global minimum.

Individual DNA consists of three translational genes in 3D axis (x, y, z), four rotational genes in quaternion (a, b, c, d) and none or many torsional gens (t_n) (Morris et al., 1998). Each individual goes through three phases: initialisation, selection and reproduction (Shiffman, 2012). In initialisation phase, individuals in a population are generated randomly using uniform distribution. For translational genes, they are bounded within the lowest and highest grid coordinate. For rotational gens, uniformly-distributed random quaternion (UDQ) (Shoemake, 1992) is applied. And finally, torsional genes have range in between $-\pi$ to π.

In selection phase, scoring function is applied to determine individual fitness. Firstly, original molecule coordinates need to be transformed into new coordinates as candidate binding mode or pose using individual genes. Then, the pose is fit into scoring function which is semiempirical free energy function resulting the individual fitness in kcal/mol. The final free energy is normalised over the total atoms found in the ligand. Finally, the finer individuals with lower free energy have the higher chance to be selected as the parents to produce next generation individuals. The algorithm ignores out of grid poses and sets it with lowest probability. Therefore, extra search space allocated for the grid is recommended to compensate binding site that is too close to the boundary.

In reproduction phase, crossing over is performed based on the genes from two selected parents. During crossing over, genes in both translational and rotational groups can undergo two types of recombination: separate and combined. Separate recombination of

translational/rotational genes means that the new individuals can have mixed translational/rotational genes of the two parents. Whereas, combine means that the new individual can only have the whole translational/rotational genes from one of its parents. The new individual produced can also undergo mutation with two different probabilities: one for nomad population and another one for settler population. Nomad population is modelled to have higher mutation rate compared to settler population.

Table 2-1 shows parameters comparison between nomad and settler populations. Individuals in nomad population (with separate translational and rotational genes during crossing over plus higher mutation rate) are more vibrant compared to individuals in settler population (with combined translational and rotational genes during crossing over plus lower mutation rate). The number of new individuals collected is up to population size. Then, the algorithm iterates back to selection phase counted as one generation.

Table 2-1: Comparison between nomad and settler parameters showing that nomad population has more vibrant individuals compared to settler population.

Parameters	Nomad	Settler
Translational genes crossing over	Separate	Combined
Rotational genes crossing over		
Crossover probability	0.5	
Mutation probability	0.75	0.25

Figure 2-2 shows the improvement settler population makes from nomad population in kcal/mol over 200 samples. The formula used is minimum free energy from nomad population minus minimum free energy from settler population. Therefore, positive energy from the result is the one to be expected.

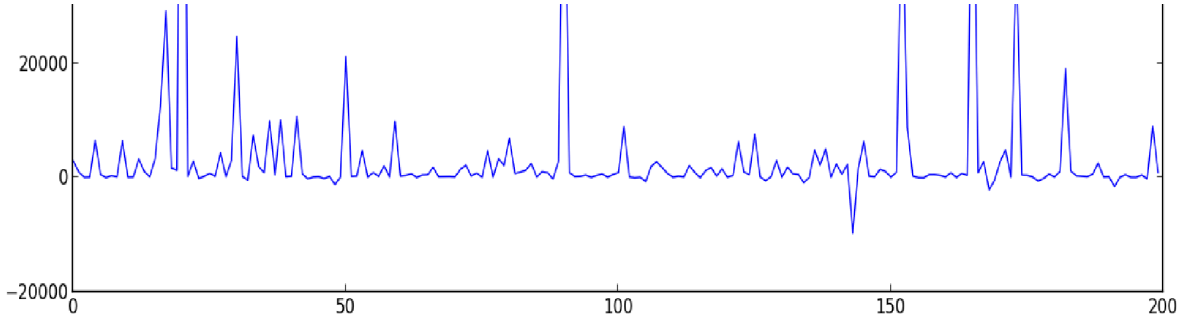

Figure 2-2: Nomad minus settler minimum free energy (in kcal/mol) over 200 samples. Positive values indicate the improvement made by having settler population.

2.3 Root-mean Square Deviation (RMSD)

One method to compare the accuracy of this new implementation (hppNeuroDock) to the one from AutoDock is by using root-mean square distance deviation (RMSD) of the pair atoms in the ligands (Kavraki, 2007). RMSD is Euclidean distance (Weisstein, n.d.) with additional normaliser $\sqrt{\frac{1}{N}}$; where N is the total atom pairs.

Following is the common formula used for RMSD in bioinformatics.

$$RMSD = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\delta_i^2} \qquad \text{Eq. 2-3}$$

To compare two ligands, v and w, following formula is used.

$$RMSD(v,w) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\|v_i - w_i\|^2} \qquad \text{Eq. 2-4}$$

2.4 Three-dimensional Linear Interpolation

For intermolecular energy calculation, all potentials have been pre-calculated by AutoGrid and stored inside map files. To get the closest estimation value for an atom inside a grid box, AutoDock uses three-dimensional linear or trilinear interpolation (Morris et al., 1998). Following are the formulas used to get approximated energy of an atom at coordinate u, v, w inside a grid box from $u0, v0, w0$ to $u1, v1, w1$ shown in Figure 2-3 (Steve, 1994). Coordinate u, v, w is the normalised value of real atom coordinate x, y, z over segment space of the grid.

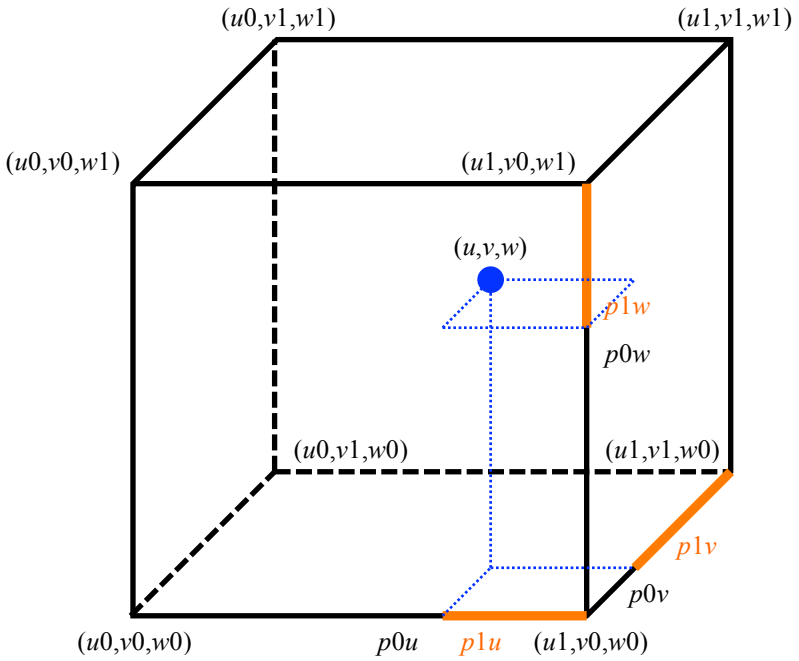

Figure 2-3: An atom at coordinate u, v, w in a grid box from coordinate $u0, v0, w0$ to $u1, v1, w1$. Where $p0u$ plus $p1u$ is the length from $u0$ to $u1$. It also applies to v and w directions.

X	Y	Z
$u = \frac{x}{spacing}$	$v = \frac{y}{spacing}$	$w = \frac{z}{spacing}$
$u0 = floor(u)$	$v0 = floor(v)$	$w0 = floor(w)$
$u1 = ceiling(u)$	$v1 = ceiling(v)$	$w1 = ceiling(w)$
$p0u = u - u0$	$p0v = v - v0$	$p0w = w - w0$
$p1u = u1 - u$	$p1v = v1 - v$	$p1w = w1 - w$

$$
\begin{aligned}
p000 &= p0u \times p0v \times p0w \\
p001 &= p0u \times p0v \times p1w \\
p010 &= p0u \times p1v \times p0w \\
p011 &= p0u \times p1v \times p1w \\
p100 &= p1u \times p0v \times p0w \\
p101 &= p1u \times p0v \times p1w \\
p110 &= p1u \times p1v \times p0w \\
p111 &= p1u \times p1v \times p1w
\end{aligned}
\qquad \text{Eq. 2-5}
$$

$p000$ to $p111$ are the scales that need to be calculated once and used by all energy maps. Following is the example of trilinear interpolation for electrostatics. Note that AutoDock array allocation for all map files is in the order of z, y, x.

$$
\begin{aligned}
e = (p000 \times e\_map[w1][v1][u1]) + (p001 \times e\_map[w1][v1][u0]) \\
+ (p010 \times e\_map[w1][v0][u1]) + (p011 \times e\_map[w1][v0][u0]) \\
+ (p100 \times e\_map[w0][v1][u1]) + (p101 \times e\_map[w0][v1][u0]) \\
+ (p110 \times e\_map[w0][v0][u1]) + (p111 \times e\_map[w0][v0][u0])
\end{aligned}
\qquad \text{Eq. 2-6}
$$

For the implementation, AutoDock has rearranged the formula to reduce number of multiplications (from 24 to 7) (Morris et al., 1998). Multiplications consume more computing power compared to additions or subtractions.

2.5 Quaternion

Quaternion is favoured against axis-angle because of its capability to perform translation and rotation efficiently in one operation (Baker, 2013). Following formula is applied to rotate rigid body of ligand (Kavraki, 2007).

$$P_{out} = q \times P_{in} \times conj(q) \qquad \text{Eq. 2-7}$$

where P_{in} is the original atom coordinate in three-dimensional space. q is the rotation in quaternion generated directly using uniformly-distributed random quaternion (UDQ) so no conversion from angle required. $conj(q)$ is quaternion conjugation of q. Finally, P_{out} represents the new coordinate of the atom.

Following formula produce uniformly-distributed random quaternion $\{a, b, c, d \in \mathbb{R}\}$ from the input of three uniform random numbers $\{x, \theta, \gamma \in \mathbb{R} \mid 0 < x, \theta, \gamma < 1\}$ (Shoemake, 1992).

$$
\begin{aligned}
r_1 &= \sqrt{1 - x} \\
r_2 &= \sqrt{x} \\
a &= \cos\theta \times r_2 \\
b &= \sin\gamma \times r_1 \\
c &= \cos\gamma \times r_1 \\
d &= \sin\theta \times r_2
\end{aligned}
\qquad \text{Eq. 2-8}
$$

2.6 Random Number Generator

There are two random number generators used in this dissertation: LFSR and RANLUX. LFSR stands for Linear Feedback Shift Register is an efficient pseudo-random numbers generator commonly used in FPGA (Alfke, 1996). The key idea is as follow. There are two components, the random number as an array of binaries and a custom logic. The custom logic takes its inputs from multiple bit locations of the number and directs one bit output to one end of the number. Initially, the number is set from seed. In one clock/iteration, the number is shifted by one bit taken the output from the custom logic as the replacement. In this process, a new random number is generated every clock cycle. In order to generate continuous random numbers, the custom logic need to be designed specifically according to total bits in the number. LFSR schematic is shown in Figure 2-4. LFSR method for random number generator is used by sequential Python implementation only. Figure 2-5 shows random numbers generated from 16-bit LFSR.

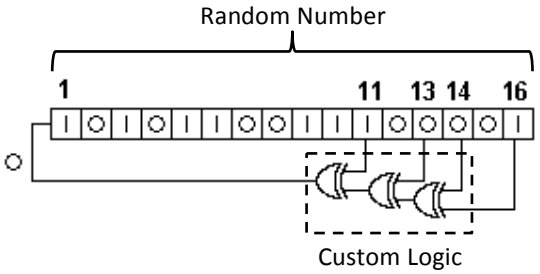

Figure 2-4: 16-bit Linear Feedback Shift Register (LFSR) schematic

LFSR method carries the behavior of counter. Depending on the bit size, the random number restarts from the seed after certain number of iterations. Following calculation is the example to produce random number for 15 days using 200 MHz clock frequency. The result shows that at least 48-bit random number is needed for such requirement.

```
total15 = (200000000 x 60 x 60 x 24 x 15) + 1
                |                      |
                |                      +--------> days
                +-------------------------------> clock frequency (Hz)
total15 =  2.5920e+14
log2(total15) = 47.881 = 48 bits
```

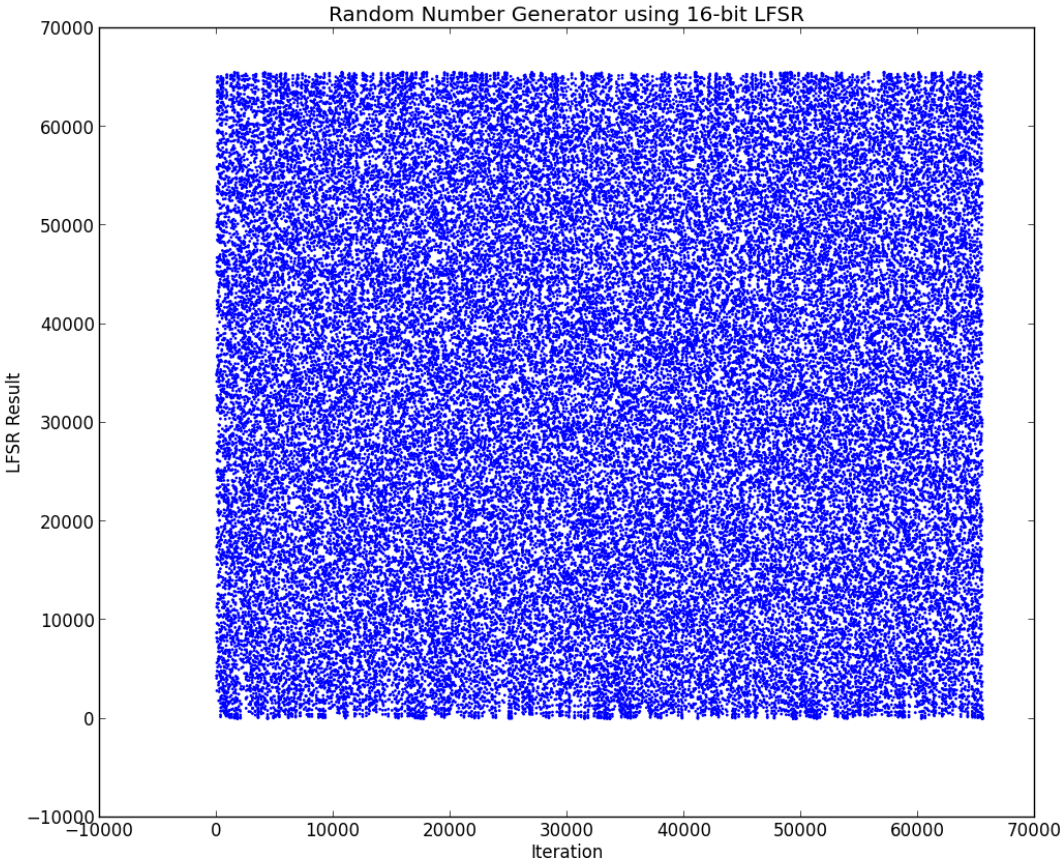

Figure 2-5: Random numbers generated using 16-bit LFSR

Parallel Python using OpenCL utilises RANLUX (LUXury RANdom numbers) for the random number generator written by Ivar Ursin Nikolaisen (Lüscher, 1994) (James, 1994). Following code shows how to use RANLUX on PyOpenCL. As the result, it fills `individuals_buf` with uniform random numbers according to the size of the buffer.

```
rng = RanluxGenerator(cl_queue)
rng.fill_uniform(individuals_buf)
```

3 hppNeuroDock Implementation

3.1 Python Implementation

Python molecular docking is implemented using object-oriented methodology, double-precision floating-point and sequential processing without any additional module. There are three major classes in the implementation: `NeuroDock`, `Dock` and `GeneticAlgorithm` classes as shown in Figure 3-1.

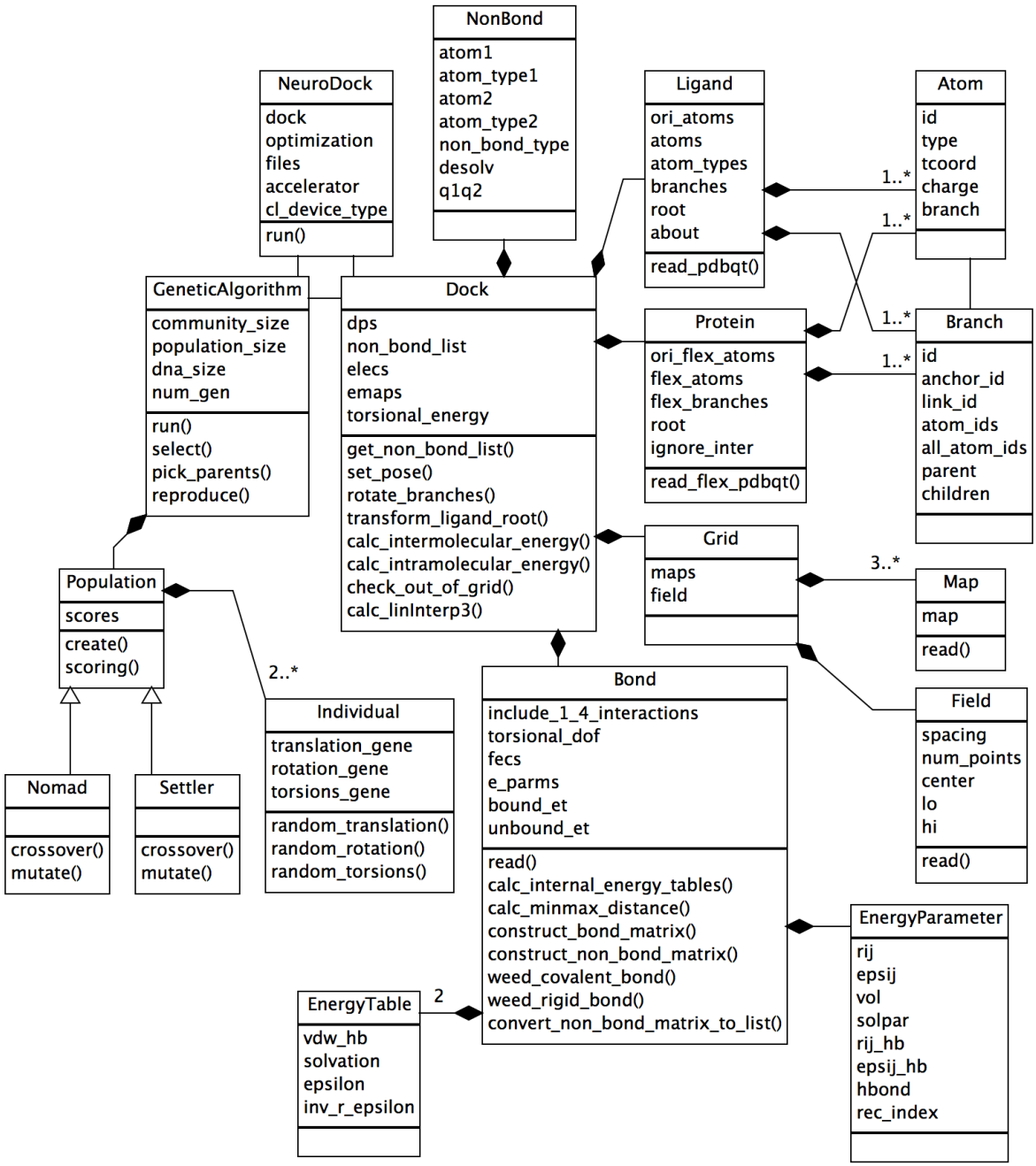

Figure 3-1: Python class diagram contains three main classes: `NeuroDock`, `Dock` and `GeneticAlgorithm`.

NeuroDock class acts like the main program. It handles user execution and processes all arguments as the input. It also parses docking parameter file (.dpf) and updates the docking parameters to target objects as it holds Dock and GeneticAlgorithm classes as its member.

Having user requirements and parameters, Dock class is instantiated as dock object that receives the control from NeuroDock class. dock object builds up objects that are related to molecular docking like the molecules (protein and ligand), the grid information (maps and field) and the atomic bound/unbound interactions for energy calculation. Original atom coordinates are maintained separately from current pose for energy calculation. Each new generated pose refers to the original atoms instead of having multiple regenerations that accumulate rounding errors. Other than accepting request to change molecular conformation, dock object also calculates both intermolecular and intramolecular/internal energies resulting total free energy. Map objects under grid object are required to calculate intermolecular energy by utilising look-up table and three-dimensional interpolation. Whereas, internal energy is calculated using semiempirical free energy function and information taken from Bond and NonBond classes.

With complete dock object that generates molecular pose and calculates energy, the control is handed over to optimisation object. As this implementation uses historical genetic algorithm for the optimisation function, the optimisation object is instantiated from GeneticAlgorithm class. optimisation object has properties like community, population size, DNA size, number of generations to run and two population (nomad and settler) objects. Each population object consists of two or more individual objects. According to Darwinian natural selection, optimisation object has selection, picking parents and reproducing methods; combined with population object that has crossover and mutation methods.

Figure 3-4 shows the entire execution flow except the rightmost column should be all in Python. Each nomad and settler populations is iterated for entire number of generations defined by user. So in total, the number of generations run is double of what stated in dpf file. At the end of settler iteration, the minimum free energy from both nomad and settler populations are reported together with total runtime.

3.2 Python-OpenCL Implementation

Python-OpenCL is implemented using PyOpenCL wrapper that requires all buffers for device to be in NumPy data type. Therefore, both NumPy and PyOpenCL modules are required for the implementation. Continuing from sequential Python implementation, floating-point data used are in double-precision unit. Note that both float and integer data types in Python are in 64-bit length whereas C's float and integer are in 32-bit length. Consequently, when receiving the data at the device program (kernel) that is written in C, the data types need to be in double and long respectively.

When implementing heterogeneous parallel program, fine-grained parallelisation and offload approach are emphasised (Ouyang & Kwoh, 2012). Fine-grained parallelisation reduces interaction among populations to avoid communication bottleneck. Instead, each population explores search space independently without exchanging individual in the populations. Offload approach divides the entire process into two: control intensive and computational intensive portions. Control unit handles population iterations that suitable for devices like CPU with advanced branch prediction. Computational unit is focused on arithmetic calculations that require GPU or FPGA with massive arrays of computing power.

As shown in Figure 3-2 and Figure 3-3, Python-OpenCL implementation only adds small portion of code under `DockOpenCL` and `GeneticAlgorithmOpenCL` classes. The two classes are inherited from `Dock` and `GeneticAlgorithm` classes respectively. `Nomad`, `Settler` and `Population` classes under `GeneticAlgorithmOpenCL` class are not the same as the classes with the same name under `GeneticAlgorithm` class. They are namespaced to each class.

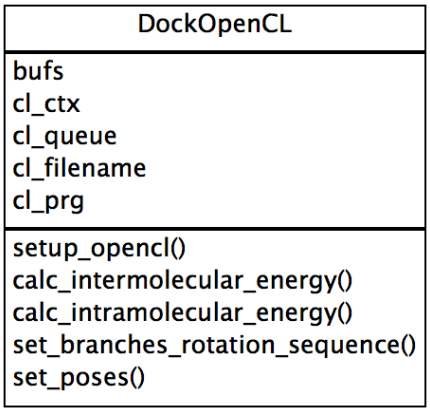

Figure 3-2: `DockOpenCL` class diagram inherited from `Dock` class.

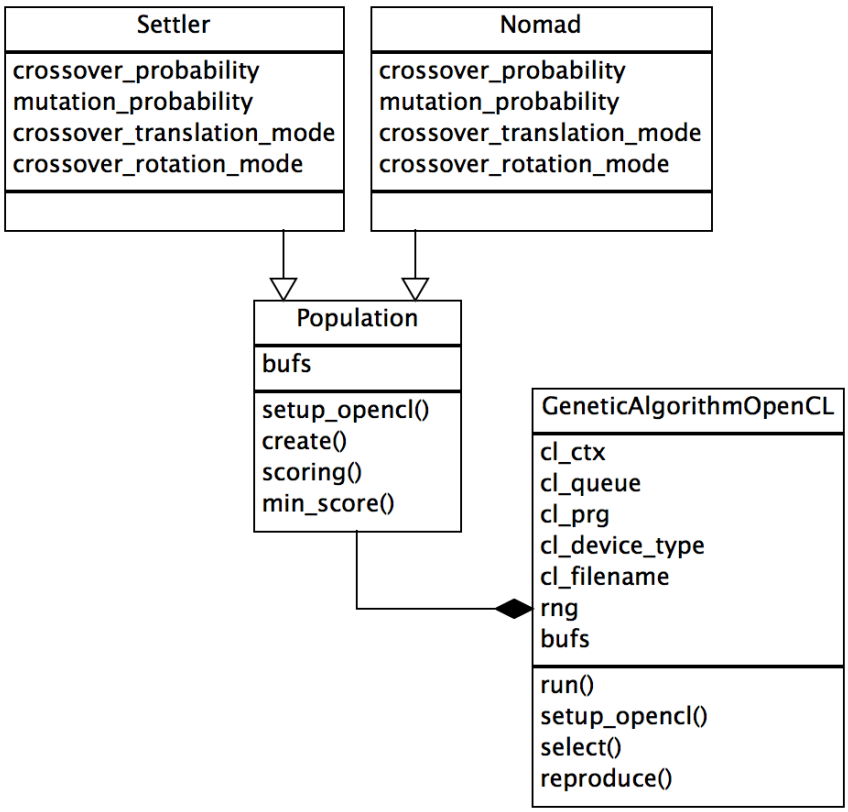

Figure 3-3: `GeneticAlgorithmOpenCL` class diagram inherited from `GeneticAlgorithm` class.

The two OpenCL classes have extra fields for OpenCL handlers (like context, queue and program) plus device buffers. `bufs` field in the class diagrams is a shorten representation of pointers to memory address in the device. So, the convention is that all variables with suffix `_np` (for NumPy) are allocated in the host but all variables with suffix `_buf` (for device buffer) are allocated in the device.

Most addition is implemented in OpenCL files located inside `OpenCL` directory and written in C, they are `Dock.cl` and `GeneticAlgorithm.cl`. The functions inside the files are computationally intensive. There are few notes on the implementation; firstly, PyOpenCL (version 2013.1) has limitation on writing data back into global memory. The process is not completely done even with `CLK_GLOBAL_MEM_FENCE` barrier. Instead, it requires the program to exit kernel in order to complete the data transfer. The limitation affects the efficiency of implementing prefix-sum operation. Instead of using kernel code, PyOpenCL provides parallel algorithms that can be executed from device side; some of them are element-wise expression evaluation (map), sums and counts (reduce), prefix sums (scan) and sorting (radix sort) (Klöckner, 2013b). Secondly, on using multiple platforms, constant data can be differently implemented from device to device. The issue encountered was with floating-point

math constant, `INFINITY`. On NVIDIA GeForce GT 650M, the value is implemented as 9223372036854775807. In contrast, Intel Core i7 returns -9223372036854775808. Therefore, once using predefined constants, it is recommended to keep using them consistently through out the code.

Figure 3-4 shows which portion is done in Python and which one in OpenCL. The execution flow is the same as sequential Python implementation. Selection and reproduction run in Python perform quick transition. They do not require intensive CPU time.

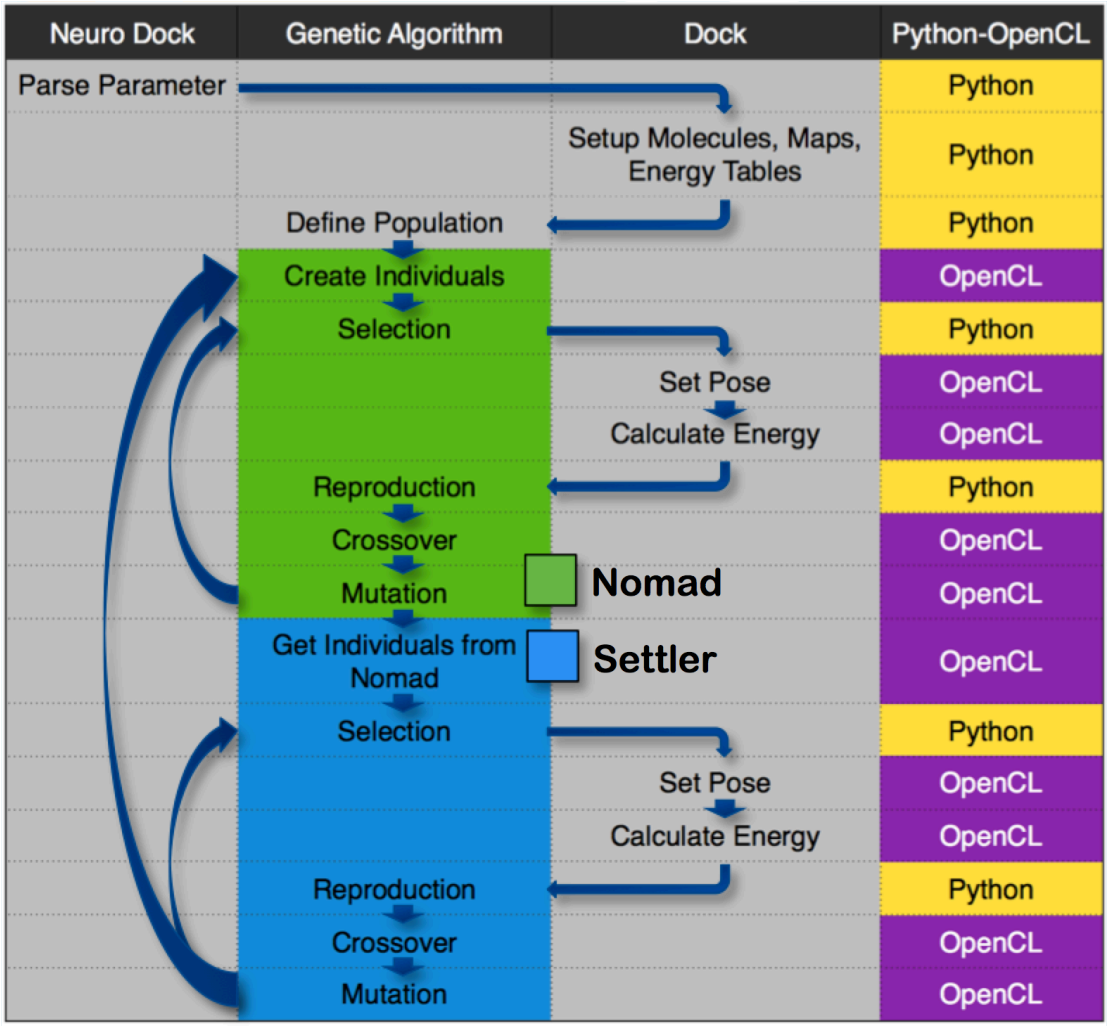

Figure 3-4: hppNeuroDock execution flow providing information about nomad/settler and Python/OpenCL portions.

Another major change is in data structure. The data is rearranged so the device can fetch as much data as possible in one DRAM burst. As Python and C have a same way of allocating array in main memory, which in row-major order, arranging required data one to another in a row manner can speed up data transfer. Figure 3-5 shows parallel data structure allocated on device main memory.

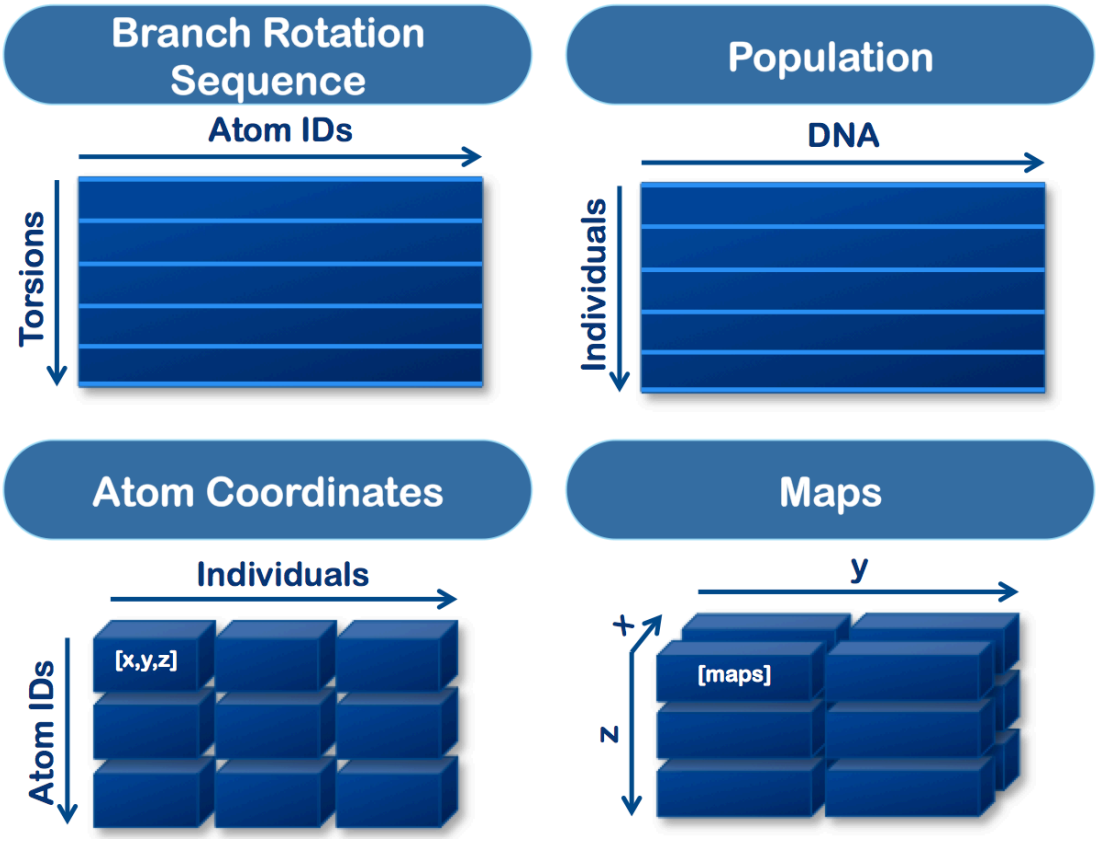

Figure 3-5: Parallel data structure showing: 1) branch rotation sequence data structure with atom IDs in row-wise, 2) population data structure with DNA in row-wise, 3) atom coordinates data structure with each atom coordinate (x, y, z) in row-wise and 4) maps data structure with maps information in row-wise.

When performing multiple level of branch rotation, the sequence has to start from the one closest to the leaf to the one near the trunk. With opposite or random order, the final molecule conformation are not the same. The order is stored inside Branch Rotation Sequence data structure. Torsions on column order define the sequence of rotation, which is from top to bottom. Whereas, atom IDs on the row order indicate atoms that can be performed in parallel without breaking the sequence. Storing the data in the array formations allows each processing element (PE) to fetch the entire atom IDs that can be process together in an efficient manner.

Population data structure is used to convert individual DNA or geometry (translation, rotation and torsions) into conformational pose using original ligand position. When performing the operation, one PE is in charge of one individual. So, by arranging the geometry in row manner, each PE can fetch the data efficiently.

Atom Coordinate data structure contains individual poses of one population for energy calculation. Since the energy calculation can be done on arbitrary atom independently, the

data structure does not require specific array formation as long as the coordinate (x, y, z) can be fetched in one burst.

Similar to the Atom Coordinate data structure, Maps data structure is also used to calculate energy. The access to the maps is not in specific order. It depends on the atom coordinate of the poses. Therefore, one way to optimise memory access is by combining map information for each atom coordinate so they can be fetched together.

3.3 Execution Procedures

Table 3-1 shows all files that are required to run hppNeuroDock. Map (`.map`) files inside `Maps` directory are automatically generated by AutoGrid. Molecule (`.pdbqt`) files inside `Inputs` directory and parameter (`.gpd`, `.dpf` and `.fld`) files in the `Parameters` directory are generated using AutoDockTools. Energy coefficient (`.dat`) file is provided by AutoDock. Other than free energy function, energy coefficient file is also a sand box to be modified by scientists to make the simulation *in silico* closer to the real experiment *in vivo* or *in vitro*.

`Python`, `Python-OpenCL_CPU` and `Python-OpenCL_GPU` directories inside `Benchmark` directory are not empty. They contain multiple `dpf` files with different parameters for benchmarking purpose. This implementation has 24 `dpf` files in each directory.

To support parallel processing feature and new genetic algorithm implementation, many keywords are newly added into docking parameter file (`.dpf`) on top of existing AutoDock 4 parameters. The new two-word method for keywords is implemented. The purpose is to increase new keywords combination possibility and for clarity, it is used to cluster the second word under the group of first word. Table 3-2 shows the list of the new keywords together with the description. Options column provides the selection for the parameters. All keywords and value assigned to them are case-sensitive.

Table 3-1: Files needed to run hppNeuroDock grouped as Python codes, input files and scripts.

Name	Type
Analysis.py	Code
Atom.py	Code
Axis3.py	Code
Constants.py	Code
Dock.py	Code
Grid.py	Code
LFSR.py	Code
Ligand.py	Code
Map.py	Code
NeuroDock.py	Code
Optimization.py	Code
Protein.py	Code
Quaternion.py	Code
OpenCL	Directory
Dock.cl	Code
GeneticAlgorithm.cl	Code
Inputs	Directory
hsg1_flex.pdbqt	Input
ind.pdbqt	Input
Parameters	Directory
AD4.1_bound.dat	Input
hsg1_rigid.maps.fld	Input
hsg1.gpf	Input
hsg1.maps.fld	Input
ind.dpf	Input
Maps	Directory
hsg1_rigid.A.map	Input
hsg1_rigid.C.map	Input
hsg1_rigid.d.map	Input
hsg1_rigid.e.map	Input
hsg1_rigid.HD.map	Input
hsg1_rigid.N.map	Input
hsg1_rigid.NA.map	Input
hsg1_rigid.OA.map	Input
hsg1.d.map	Input
hsg1.e.map	Input
Benchmark.sh	Script
Benchmark	Directory
Python	Directory
Python-OpenCL_CPU	Directory
Python-OpenCL_GPU	Directory

Table 3-2: New parameters introduced in hppNeuroDock with options, default value and description of the keywords and the options.

Keyword	Options	Default Value	Description
accelerator	sequential, opencl	opencl	Parallel processing accelerator. • Sequential: Sequential processing. • OpenCL: Heterogeneous parallel processing.
ocl_device_type	cpu, gpu, manual	cpu	OpenCL device types. • CPU: Automatically using CPU. • GPU: Automatically using GPU. • Manual: User is prompted with the list of available devices.
b_prm	-	AD4.1_bound.dat	Atomic bonding parameter file to use.
pre_energy_calc	-	-	Command to execute pre-energy calculation.
opt_type	ga	ga	Optimization types. • GA: Historical genetic algorithm.
opt_ga community_size	-	200	Number of populations in a community.
opt_ga pop_size	-	150	Number of individuals in a population.
opt_ga num_generations	-	50	Number of generations.
opt_run	-	-	Command to start optimisation process.

To run hppNeuroDock, go to `PyNeuroDock` directory and execute following command.

```
$ python NeuroDock.py
```

As the default, hppNeuroDock uses `ind.dpf` docking parameter file located inside `Parameters` directory. `-p` option is provided so that users can specify their own custom `dpf` file. Following is the example on how to use it.

```
$ python NeuroDock.py -p other_docking_parameter_file.dpf
```

The output is shown below. The runtime is the elapsed time for one community (in seconds). It includes both nomad and settler populations iterated through predefined number of generations. The first minimum score is minimum free energy (in kcal/mol) resulting from nomad population and the second one is from settler population.

```
Elapsed time community 1:  1.59 - Minimum Scores:    157.564,    149.932
Elapsed time community 2:  1.58 - Minimum Scores:    288.726,    239.274
Elapsed time community 3:  1.57 - Minimum Scores:   2066.306,   1464.533
Elapsed time community 4:  1.57 - Minimum Scores:    619.520,    595.584
Elapsed time community 5:  1.54 - Minimum Scores:    712.732,    600.406
Elapsed time community 6:  1.55 - Minimum Scores:    392.179,    281.683
Elapsed time community 7:  1.57 - Minimum Scores:    340.395,    201.145
Elapsed time community 8:  1.55 - Minimum Scores:    217.577,    210.240
Elapsed time community 9:  1.57 - Minimum Scores:    299.383,    236.211
Elapsed time community 10: 1.56 - Minimum Scores:    730.148,    571.318
```

The runtime does not include initialisation time. Initialisation time can be calculated separately using additional UNIX `time` command. Following `time` command returns overall execution time taken by "`python NeuroDock.py`" command.

```
$ time python NeuroDock.py
```

`time` command returns real, user and system times as shown below. Only real time reports "wall clock" time. Both user and system times are in CPU time. The initialisation time is taken from real time minus runtime. Using following example, the real time is 3.193 seconds and runtime is 1.39 seconds. So the initialisation time is 1.803 seconds.

```
Running... Benchmark/ind_runtime_opencl_128_100.dpf
Elapsed time community 1:  1.39 - Minimum Scores:    613.841,    245.992
Community Minimum Scores: [[613.84100212587373, 245.99190378055528]]

real   0m3.193s
user   0m1.008s
sys    0m0.245s
```

3.4 Benchmarking Script

Benchmarking script (`Benchmark.sh`) runs hppNeuroDock with different population sizes, numbers of generations and accelerator devices to compare the runtime taken among them. The script is developed using Bash shell and executed outside Python environment to query initialisation time. It gets different parameters to run from `dpf` files located inside `Benchmark` directory. To run the benchmark, go to `PyNeuroDock` directory and execute following command. The result can be found inside `Benchmark.txt` file located inside `Benchmark` directory.

```
$ ./Benchmark.sh 2>&1 | tee Benchmark/Benchmark.txt
```

Following is the example of the output.

```
Run Time Benchmark

------------------------------------------------------------------------
Running... Benchmark/ind_runtime_opencl_32_10.dpf
Elapsed time community 1:  0.03 - Minimum Scores: 126747.555,   5021.270
Community Minimum Scores: [[126747.55486051459, 5021.2697327719989]]

real   0m0.965s
user   0m0.933s
sys    0m0.159s
```

```
--------------------------------------------------------------------------------
Running... Benchmark/ind_runtime_opencl_64_10.dpf
Elapsed time community 1:  0.06 - Minimum Scores: 565042.853, 497869.751
Community Minimum Scores: [[565042.85272403888, 497869.75063370226]]

real   0m0.953s
user   0m1.124s
sys    0m0.164s
--------------------------------------------------------------------------------
```

3.5 Analysis Script

Analysis class is the place to gather functions for examining docking result. Three methods to calculate RMSD of two ligands have been developed. Each of them handles different inputs. They can be PDBQT files, ligand objects, or arrays of atom coordinates in NumPy. The result is RMSD of the two ligands in angstroms (Å). Following is the example of using RMSD function with PDBQT files as the input.

```
$ python
>>> from Analysis import Analysis
>>> print Analysis.rmsd_pdbqts("./Results/ind_autodock_-15.66.pdbqt",
"./Results/ind_pyneurodock_-14.27.pdbqt")
```

3.6 Source Code

The source code used for this dissertation is published using annotated Git Tag on following URL.

```
https://github.com/ekaakurniawan/FpgaNeuroDock/releases/tag/NTU_Dissertation
```

Figure 3-6 below shows the target link of the URL where the code can be browsed online or downloaded into local computer using either `zip` or `tar.gz` files.

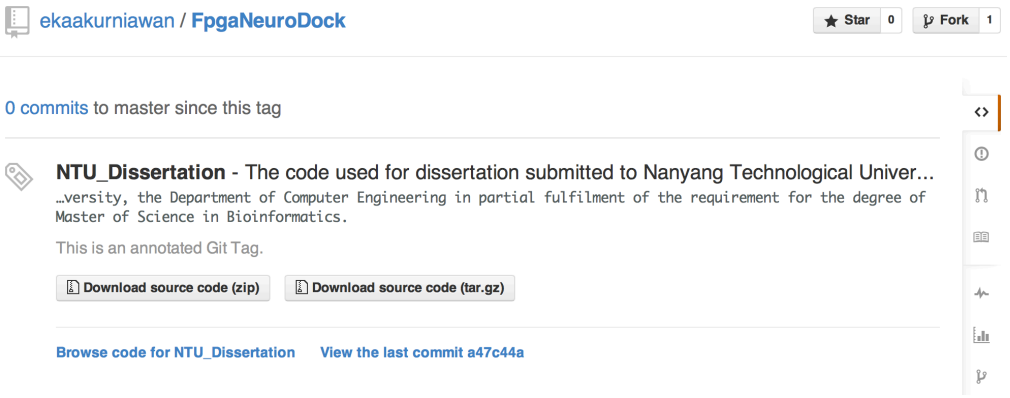

Figure 3-6: Annotated Git tagging with description and source code. The source code can be browsed online or downloaded into local computer using either `zip` or `tar.gz` files.

4 Docking Result

4.1 Numerical

Protein and ligand used for testing are HIV-1 (Goodsell, 2000) and Indinavir (National Center for Biotechnology Information, n.d.). The HIV-1 protease is divided into two parts, rigid and flexible. Out of 1844 atoms in total, only 24 of them (two chains of 12 atoms) considered to be flexible. The two chains have in total 6 branches. Whereas for the ligand, Indinavir has 49 atoms and also 6 branches. Combined together, each individual has three translational genes (x, y, z), four rotational genes (a, b, c, d) and twelve torsional genes $(t_1, t_2, \dots, t_{12})$.

AutoGrid prepares all ligand and protein information. So, for hppNeuroDock, only some parameters need to be updated on top of existing `dpf` file, they are OpenCL for accelerator to use, CPU for device type, 10000 for community size, 150 for population size and 200 for number of generations. Having the parameters ready, the test can be run according to execution procedures described in previous section.

As the result, hppNeuroDock acquires docking free energy of -14.27 kcal/mol. The result is not further apart from AutoDock 4 (version 4.2.3), which is -15.66 kcal/mol. Furthermore, RMSD between the two is at 1.53 Å, which is still lower than the threshold of 2.0 Å.

4.2 Visual

The ligand with minimum free energy is saved into PDBQT file. Using Open Babel, the PDBQT file is then converted into PDB file. PDB file is better for three-dimensional visualisation because it has information about connection between the atoms. Without atom connection information, some visualisers fail to make a proper atomic connection. PyMOL is used to visualise the three-dimensional structure of both HIV-1 and Indinavir.

Figure 4-1 visually compares both AutoDock 4 and hppNeuroDock ligands side by side. AutoDock 4 is shown in magenta and hppNeuroDock is in cyan. The rigid bodies are matched up closely as well as most of the torsions except the head on top-right corner.

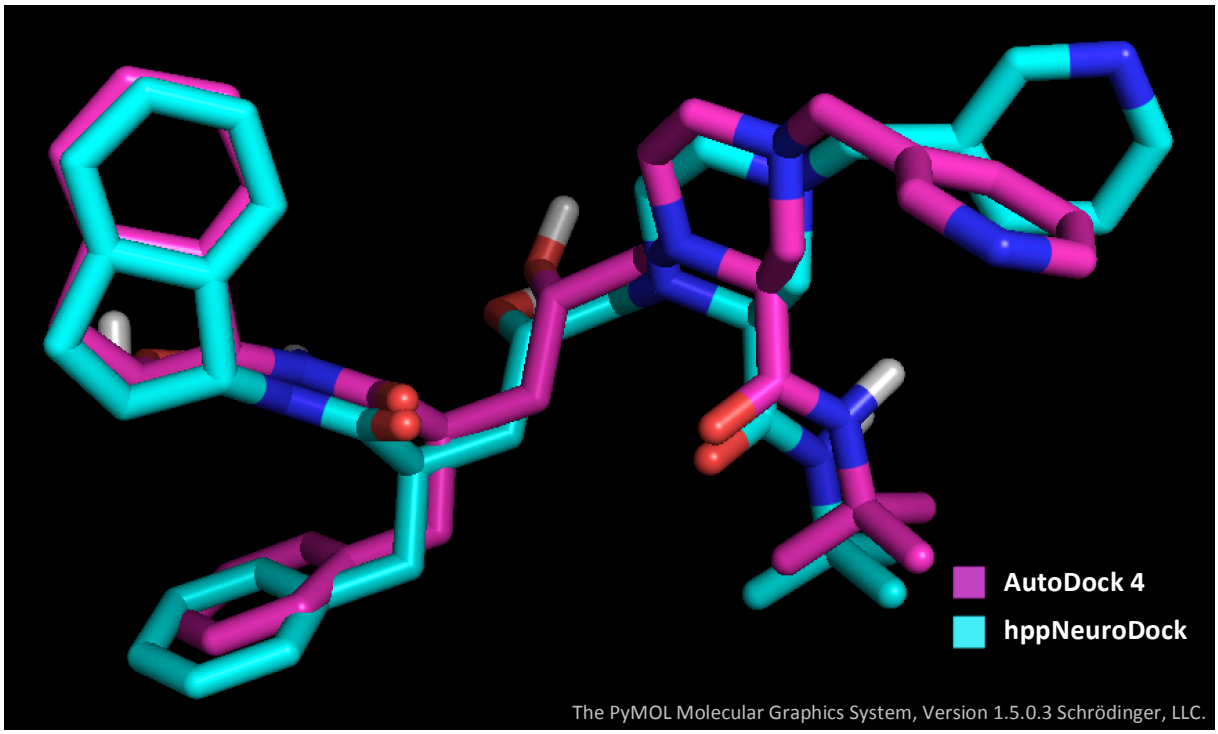

Figure 4-1: Indinavir ligand conformation result of hppNeuroDock (in cyan) and AutoDock 4 (in magenta) with RMSD of 1.53 Å.

Figure 4-2 and Figure 4-3 below visualise binding mode of HIV-1 protease and Indinavir from hppNeuroDock. The first figure is shown with the protein in secondary structure and the second figure is in surface view. The protein secondary structure and surface view are coloured in rainbow to show the flow of polypeptide chain. The second figure shows flexible part of the protein. The part has two molecule chains. One of them can be seen on the figure as coloured in purple and located slightly below the ligand. The other is located at the exact opposite of the protein.

From the two figures, the ligand is depicted to fit in the docking site. Looking from the perspective of the whole protein size, the deviation of hppNeuroDock from the AutoDock 4 result is quite minor. Of course, the simulation can only provide suggestion to scientists for further evaluation.

Figure 4-2: Binding mode of Indinavir ligand (from hppNeuroDock) to HIV-1 protein in secondary structure.

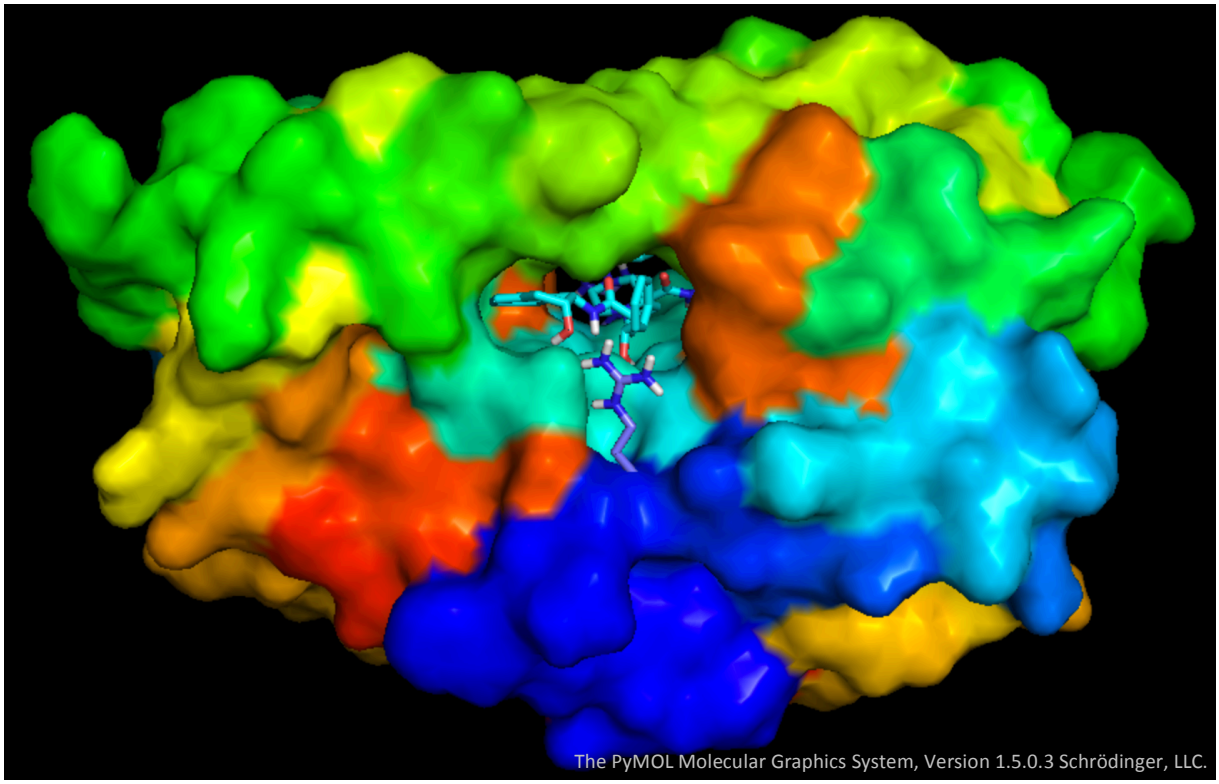

Figure 4-3: Binding mode of Indinavir ligand (from hppNeuroDock) to HIV-1 protein in surface view. Flexible part of the protein is shown in purple, located slightly below the ligand.

5 Benchmark

5.1 Hardware and Software

The main target of AutoDock implementation has been lab workstations instead of computers in clusters (Morris et al., 2010). Following survey on Figure 5-1 is taken from MGLTools website that is affiliated with AutoDock. It shows that the result for CPU-GPU hybrid is close to one fifth of the total video cards. Meaning, the support for portable computer with short runtime while providing desirable result is important. Other than that, OpenCL supports close to 93% of the graphic cards used.

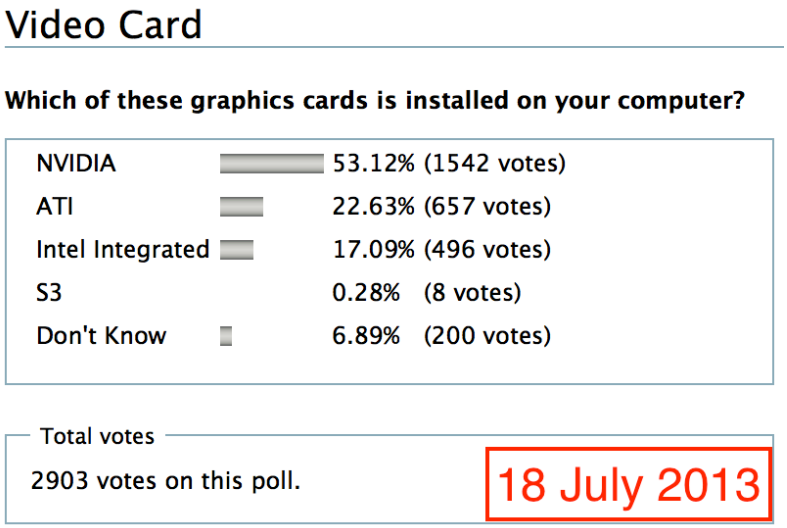

Figure 5-1: Video card survey taken from MGLTools website showing that OpenCL supports close to 93% of the graphic card used.

This benchmark is performed using Apple MacBook Pro Retina (Mid 2012) laptop with following hardware specifications (Table 5-1) and software versions (Table 5-2).

Table 5-1: Hardware used for benchmark on Apple MacBook Pro Retina (Mid 2012) laptop.

Hardware	Specifications
Processor	2.3GHz Intel Core i7 (i7-3615QM)
Graphics	NVIDIA GeForce GT 650M 1GB
Memory	8GB 1600MHz DDR3

Table 5-2: Software used for benchmark on Apple OS X version 10.8.4 Mountain Lion.

Software	Version
Python	2.7.3
NumPy	1.7.1
PyOpenCL	2013.1

Table 5-3, Table 5-4 and Table 5-5 present the hardware specifications in more detail.

Table 5-3: Intel Core i7-3615QM detail specifications (Intel, n.d.)

CPU Specifications	
Type	Core i7-3615QM
Clock Speed	2.3 GHz
Max Turbo Frequency	3.3 GHz
Cores	4
Threads	8
L2 Cache (per Core)	256 KB
L3 Cache	6 MB
Instruction Set	64-bit
GPU	Intel HD Graphics 4000
GPU Base Frequency	650 MHz
GPU Max Dynamic Frequency	1.2 GHz
GPU VRAM (total)	512 MB

Table 5-4: NVIDIA GeForce GT 650M detail specifications (NVIDIA, n.d.) (Intel, n.d.)

GPU Specifications	
Type	NVIDIA GeForce GT 650M
CUDA Cores	384
Max Graphics Clock	900 MHz
Bus	PCIe x8
VRAM (total)	1024 MB
Memory Interface Width	128-bit
Max Memory Bandwidth	80.0 GB/sec

Table 5-5: Main Memory detail specifications (Intel, n.d.)

Memory Specifications	
Space	8 GB
Interface	DDR3
Frequency	1600 MHz
Max Memory Bandwidth	25.6 GB/s

5.2 Initialisation Time

Table 5-6, Table 5-7 and Table 5-8 show initialisation time of different population sizes, numbers of generations and accelerator devices. Initialisation time can be broken down into Python setup time, docking object instantiation, and for parallel execution, it includes both OpenCL compilation and sending computational data from host to device global memory.

Table 5-6: Sequential Python initialisation time (in second) using CPU based on different population sizes and numbers of generations.

Number of Generations	Population Size					
	32	64	128	256	512	1024
10	0.935	0.893	0.885	0.89	0.889	-
50	0.874	0.884	0.882	0.883	0.887	-
100	0.884	0.883	0.888	0.893	0.889	-
500	0.898	0.878	0.88	0.889	0.895	0.895
1000	-	-	-	0.895	0.887	0.891

Table 5-7: Parallel Python-OpenCL initialisation time (in second) using GPU based on different population sizes and numbers of generations.

Number of Generations	Population Size					
	32	64	128	256	512	1024
10	1.733	1.97	2.291	1.627	1.783	-
50	1.809	1.794	1.806	1.796	1.787	-
100	1.786	1.633	1.803	1.769	1.74	-
500	2.286	1.592	1.727	1.958	1.844	1.8
1000	-	-	-	1.729	1.783	1.788

Table 5-8: Parallel Python-OpenCL initialisation time (in second) using CPU based on different population sizes and numbers of generations.

Number of Generations	Population Size					
	32	64	128	256	512	1024
10	0.812	0.744	0.744	0.744	0.749	-
50	0.733	0.734	0.744	0.744	0.746	-
100	0.738	0.747	0.733	0.745	0.738	-
500	0.766	0.733	0.743	0.777	0.856	0.767
1000	-	-	-	0.734	0.746	0.743

Figure 5-2 shows that initialisation time for Python-OpenCL GPU is considerably higher than sequential Python and Python-OpenCL CPU. Whereas, for the two implementations that run on CPU, the initialisation times are approximately the same (below one second). Overall, initialisation time does not depend on the two factors that have been considered for the data collection. They are population sizes and numbers of generations. Furthermore, in common scenario, initialisation time is still far below the runtime that can last for hours.

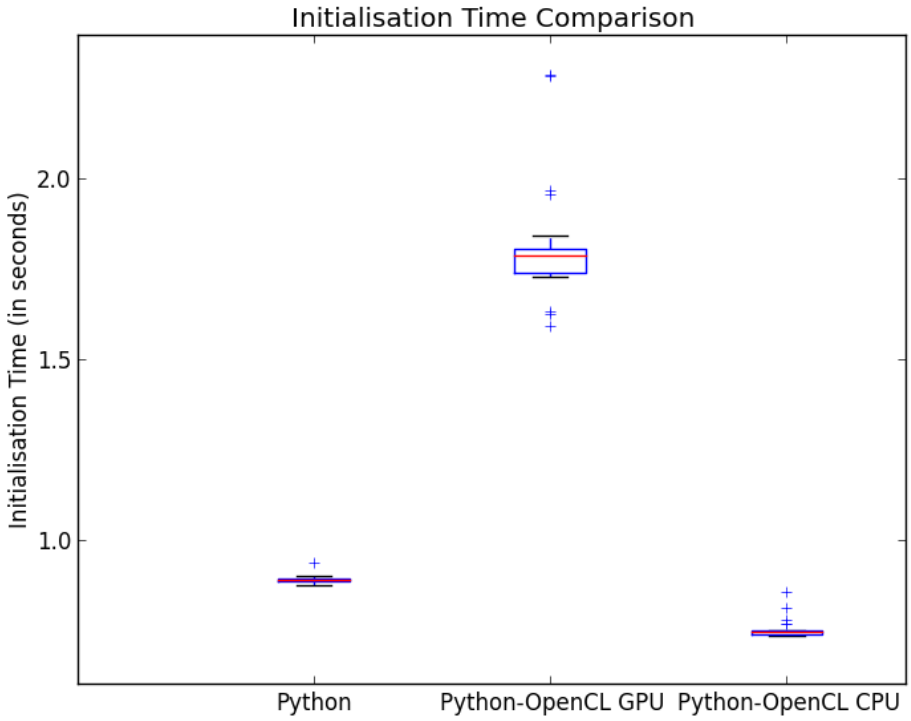

Figure 5-2: Initialisation time comparison between sequential Python, parallel Python-OpenCL using GPU and parallel Python-OpenCL using CPU in Box Plot.

5.3 Runtime

Table 5-9, Table 5-10 and Table 5-11 show runtime (in seconds) based on different population sizes, numbers of generations and accelerator devices. “Python runtime” collects the data using sequential Python implementation and run on CPU. “Python-OpenCL GPU” and “Python-OpenCL CPU” runtimes collect the data using parallel Python-OpenCL implementation and run on GPU and CPU respectively.

Table 5-9: Sequential Python runtime (in seconds) using CPU based on different population sizes and numbers of generations.

Number of Generations	Population Size					
	32	64	128	256	512	1024
10	5.16	7.43	14.48	32.74	58.12	-
50	48.9	95.78	176.14	391.87	729.37	-
100	105.53	211.02	403.57	861.12	1659.55	-
500	566.68	1156.61	2338.5	4667	9363	18593
1000	-	-	-	9217	18521	37876

Table 5-10: Parallel Python-OpenCL runtime (in seconds) using GPU based on different population sizes and numbers of generations.

Number of Generations	Population Size					
	32	64	128	256	512	1024
10	0.11	0.14	0.18	0.26	0.51	-
50	0.44	0.52	0.72	1.13	2.25	-
100	0.82	1.01	1.39	2.21	4.38	-
500	4.63	5.54	7.46	11.39	21.44	44.12
1000	-	-	-	21.82	42.84	88.53

Table 5-11: Parallel Python-OpenCL runtime (in seconds) using CPU based on different population sizes and numbers of generations.

Number of Generations	Population Size					
	32	64	128	256	512	1024
10	0.03	0.06	0.11	0.22	0.44	-
50	0.12	0.29	0.54	1.03	2.04	-
100	0.23	0.58	1.07	2.06	4.03	-
500	1.2	2.88	5.26	10.2	19.82	39.7
1000	-	-	-	20.11	39.78	79.27

Figure 5-3 below compares the runtime of different implementations over different population size in 500 generations. It shows that sequential Python runtime increases linearly as the population size increased. It also shows that Python-OpenCL GPU runtime increases linearly for population size bigger than 256. Below that, it performs slower because the GPU cores are not fully utilised. Python-OpenCL CPU runtime also increases linearly from population size of 64. At the population size 32, Python-OpenCL CPU performs significantly better than the projected linear trend.

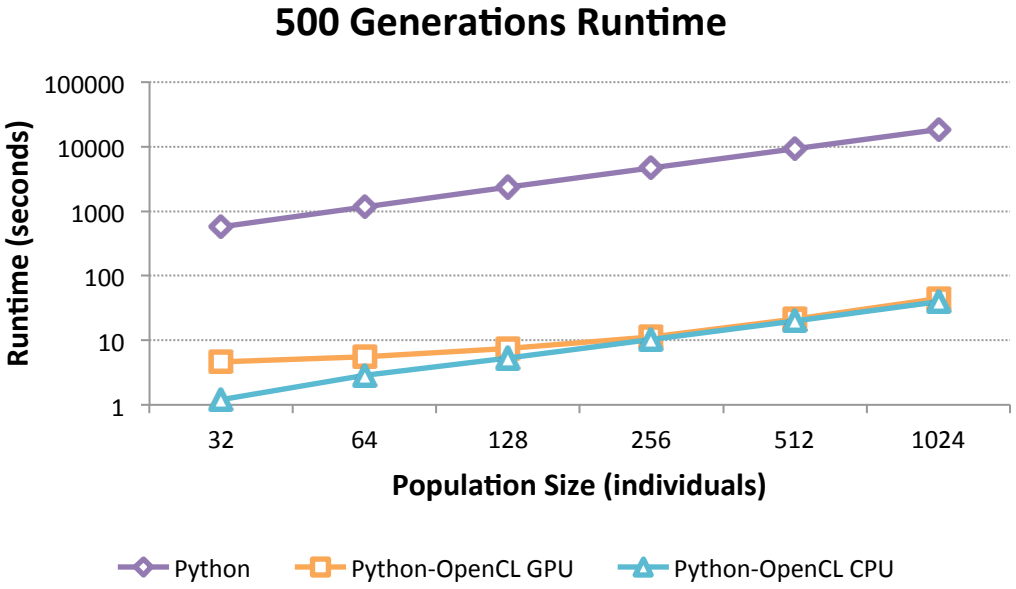

Figure 5-3: Combined sequential Python, parallel Python-OpenCL using GPU and parallel Python-OpenCL using CPU runtimes in 500 generations based on different population size.

5.4 Speedup

Table 5-12 and Table 5-13 show the speedup achieved by parallelisation. They indicate the requirement to have big population size (higher than 512 individuals) run over long generations (higher than 500 generations) to maximise the speedup. The highest speedup achieved within each table is highlighted in cyan. They are 436 times for GPU and 477 for CPU.

Table 5-12: Sequential Python over parallel Python-OpenCL GPU speedup based on different population sizes and numbers of generations.

Number of Generations	Population Size					
	32	64	128	256	512	1024
10	46.91	53.07	80.44	125.92	113.96	-
50	111.14	184.19	244.64	346.79	324.16	-
100	128.70	208.93	290.34	389.65	378.89	-
500	122.39	208.77	313.47	409.75	436.71	421.42
1000	-	-	-	422.41	432.33	427.83

Table 5-13: Sequential Python over parallel Python-OpenCL CPU speedup based on different population sizes and numbers of generations.

Number of Generations	Population Size					
	32	64	128	256	512	1024
10	172.00	123.83	131.64	148.82	132.09	-
50	407.50	330.28	326.19	380.46	357.53	-
100	458.83	363.83	377.17	418.02	411.80	-
500	472.23	401.60	444.58	457.55	472.40	468.34
1000	-	-	-	458.33	465.59	477.81

Table 5-14 compares the runtime of different accelerator types. GPU performs slower in small population size (lower than 384) because of the cores are not fully utilised. With population size higher than 256, the both devices perform quite similarly. Other than that, CPU achieves significant gain on small population size (32 individuals).

Table 5-14: Parallel Python-OpenCL using GPU over CPU speedup based on different population sizes and numbers of generations.

Number of Generations	Population Size					
	32	**64**	**128**	**256**	**512**	**1024**
10	3.67	2.33	1.64	1.18	1.16	-
50	3.67	1.79	1.33	1.10	1.10	-
100	3.57	1.74	1.30	1.07	1.09	-
500	3.86	1.92	1.42	1.12	1.08	1.11
1000	-	-	-	1.09	1.08	1.12

6 Conclusions and Future Works

Main contributions of this dissertation are implementing AutoDock 4 semiempirical energy function in Python (which originally written in C), implementing new algorithm for conformational search called historical genetic algorithm and implementing heterogeneous parallel program by integrating Python and OpenCL.

AutoDock 4 semiempirical energy function and historical genetic algorithm have been successfully tested using HIV-1 protease for the protein and Indinavir as the ligand. This implementation called hppNeuroDock produces ligand conformation with RMSD 1.53 Å deviated from AutoDock 4. Which is still lower than the threshold of 2.0 Å. It also produces negative docking energy of -14.27 kcal/mol compared to -15.66 kcal/mol produced by AutoDock 4.

Heterogeneous parallel program using OpenCL wrapper, PyOpenCL, has been successfully tested as well. Benchmarking is performed on sequential Python using CPU, parallel Python-OpenCL using GPU and parallel Python-OpenCL using CPU. Based on different population sizes and numbers of generations, benchmark result shows that parallel Python improves the runtime up to 477 times faster than sequential Python.

Despite the speedup, hppNeuroDock has difficulty to acquire free energy as low as AutoDock 4. By implementing Lamarckian genetic algorithm, the issue could be solved and in addition, runtime comparison among them becomes more appropriate. Continuing on PyOpenCL implementation using FPGA for heterogeneous parallel computing and integrating MPI onto hppNeuroDock for distributed computing can be considered for future works.

Bibliography

Scannell, J.W., Blanckley, A., Boldon, H. & Warrington, B., 2012. Diagnosing the decline in pharmaceutical R&D efficiency. *Nature Reviews Drug Discovery*, 11, pp.191-200.

O'Boyle, N.M. et al., 2011. Open Babel: An open chemical toolbox. *Journal of Cheminformatics*, 3, p.33.

Kirk, D.B. & Hwu, W.W., 2010. *Programming Massively Parallel Processors: A Hands-on Approach*. 1st ed. Burlington: Morgan Kaufmann.

Morris, G.M. et al., 1998. Automated Docking Using a Lamarckian Genetic Algorithm and and Empirical Binding Free Energy Function. *Computational Chemistry*, 19, pp.1639-62.

Morris, G.M. et al., 2009. Autodock4 and AutoDockTools4: automated docking with selective receptor flexiblity. *Computational Chemistry*, 16, pp.2785-91.

Tsai, C.S., 2002. *An Introduction to Computational Biochemistry*. New York: Wiley-Liss.

Huey, R., Morris, G.M., Olson, A.J. & Goodsell, D.S., 2007. A Semiempirical Free Energy Force Field with Charge-Based Desolvation. *Computational Chemistry*, 28, pp.1145-52.

Sanner, M.F., 1999. Python: A Programming Language for Software Integration and Development. *Journal of Molecular Graphics and Modelling*, 17, pp.57-61.

Klöckner, A. et al., 2012. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Computing*, 38, pp.157-74.

Huey, R. & Morris, G.M., 2008. *Using AutoDock 4 with AutoDockTools: A Tutorial*. [Online] The Scripps Research Institute Available at: http://autodock.scripps.edu/faqs-help/tutorial/using-autodock-4-with-autodocktools/2012_ADTtut.pdf.

Morris, G.M. et al., 2010. *User Guide AutoDock Version 4.2: Automated Docking of Flexible Ligands to Flexible Receptors*. [Online] Available at: http://autodock.scripps.edu/faqs-help/manual/autodock-4-2-user-guide/AutoDock4.2_UserGuide.pdf.

Weinhold, B., 2006. Epigenetics: the science of change. *Environmental Health Perspectives*, 114, pp.160–67.

Shoemake, K., 1992. Uniform Random Rotations. In David, K. *Graphics Gems III*. Cambridge: Academic Press. pp.124-32.

Cole, J.C. et al., 2005. Comparing protein-ligand docking programs is difficult. *Proteins*, 60, pp.325-32.

Alfke, P., 1996. *Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators*. [Online] XILINX Available at: http://www.xilinx.com/support/documentation/application_notes/xapp052.pdf.

Lüscher, M., 1994. A portable high-quality random number generator for lattice field theory simulations. *Computer Physics Communications*, 79, pp.100-10.

James, F., 1994. RANLUX: A Fortran implementation of the high-quality pseudorandom number generator of Lüscher. *Computer Physics Communications*, 79, pp.111-14.

Khronos Group, 2012. *OpenCL Overview*. [Online] Available at: http://www.khronos.org/assets/uploads/developers/library/overview/opencl_overview.pdf.

Khronos Group, 2013. *The open standard for parallel programming of heterogeneous systems*. [Online] Available at: http://www.khronos.org/opencl/ [Accessed 10 August 2013].

Python Software Foundation, 2013. *About Python*. [Online] Available at: http://www.python.org/about/ [Accessed 10 August 2013].

Westhead, D., Clark, D. & Murray, C., 1997. A comparison of heuristic search algorithms for molecular docking. *Computer-Aided Molecular Design*, 11, pp.209-28.

Hou, T.J., Wang, J.M. & Xu, X.J., 1999. A Comparison of Three Heuristic Algorithms for Molecular Docking. *Chinese Chemical Letters*, 10(7), pp.615–18.

Klöckner, A., 2013b. *Parallel Algorithms*. [Online] Available at: http://documen.tician.de/pyopencl/algorithm.html [Accessed 10 August 2013].

Klöckner, A., 2013a. *Welcome to PyOpenCL's documentation!* [Online] Available at: http://documen.tician.de/pyopencl/ [Accessed 10 August 2013].

Owens, J.D. et al., 2008. GPU Computing. *Proceedings of the IEEE*, 96(5), pp.879,899.

Feldman, M., 2012. *OpenCL Gains Ground On CUDA*. [Online] Available at: http://www.hpcwire.com/hpcwire/2012-02-28/opencl_gains_ground_on_cuda.html [Accessed 10 August 2013].

National Geographic Society, 2013. *The Development of Agriculture*. [Online] Available at: https://genographic.nationalgeographic.com/development-of-agriculture/ [Accessed 10 August 2013].

Kavraki, L.E., 2007. *Molecular Distance Measures*. [Online] Available at: http://cnx.org/content/m11608/1.23/.

Steve, H., 1994. Tri-linear Interpolation. In P. Heckbert, ed. *Graphics Gems IV*. 1st ed. San Francisco: Morgan Kaufmann. p.521.

Baker, M.J., 2013. *Maths - Transformations using Quaternions*. [Online] Available at: http://www.euclideanspace.com/maths/algebra/realNormedAlgebra/quaternions/transforms/index.htm [Accessed 11 August 2013].

Goodsell, D., 2000. *HIV-1 Protease*. [Online] Available at: http://www.rcsb.org/pdb/education_discussion/molecule_of_the_month/download/HIV-1Protease.pdf [Accessed 11 August 2013].

Bolton, E., Wang, Y., Thiessen, P.A. & Bryant, S.H., 2008. PubChem: Integrated Platform of Small Molecules and Biological Activities. *Annual Reports in Computational Chemistry*, 4, pp.217-41.

Intel, n.d. *Intel Core i7-3615QM Processor*. [Online] Available at: http://ark.intel.com/products/64900 [Accessed 11 August 2013].

NVIDIA, n.d. *GeForce GT 650M Specifications*. [Online] Available at: http://www.geforce.com/hardware/notebook-gpus/geforce-gt-650m/specifications [Accessed 11 August 2013].

Weisstein, E.W., n.d. *Distance.* [Online] Available at: http://mathworld.wolfram.com/Distance.html [Accessed 8 September 2013].

Shiffman, D., 2012. *Chapter 9. The Evolution of Code*. [Online] Available at: http://natureofcode.com/book/chapter-9-the-evolution-of-code/ [Accessed 10 August 2013].

Ouyang, X. & Kwoh, C.K., 2012. GPU Accelerated Molecular Docking with Parallel Genetic Algorithm. In *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*. Singapore, 2012. CPS.

National Center for Biotechnology Information, n.d. *Indinavir - Compound Summary (CID 5362440)*. [Online] Available at: http://pubchem.ncbi.nlm.nih.gov/summary/summary.cgi?cid=5362440 [Accessed 11 August 2013].