ACCELERATING MOLECULAR DOCKING and BINDING SITE MAPPING USING FPGAs and GPUs

BHARAT SUKHWANI

Dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

BOSTON

UNIVERSITY

BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Dissertation

ACCELERATING MOLECULAR DOCKING and BINDING SITE MAPPING USING FPGAs and GPUs

by

BHARAT SUKHWANI

M.S., The University of Arizona, 2005

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
2011

© Copyright by
BHARAT SUKHWANI
2010

Approved by

First Reader

Martin C. Herbordt, PhD
Professor of Electrical and Computer Engineering

Second Reader

Roscoe Giles, PhD
Professor of Electrical and Computer Engineering

Third Reader

Ayse Coskun, PhD
Assistant Professor of Electrical and Computer Engineering

Fourth Reader

Sandor Vajda, PhD
Professor of Biomedical Engineering

A man should look for what is, and not for what he thinks should be. Albert Einstein

Education is what remains after one has forgotten what one has learned in school. Albert Einstein

Acknowledgments

I would like to thank my advisor, Professor Martin Herbordt, for introducing me to the wonderful world of high performance computing. Also, I am deeply grateful to him for his help and guidance and for providing the intellectual, moral and financial support throughout the course of my PhD. His support and contributions played an important role towards the successful completion of my PhD.

I would also like to thank the members of my PhD defense committee Professors Roscoe Giles, Sandor Vajda and Ayse Coskun, for their valuable inputs which helped improve my dissertation. I am grateful for the time they took to carefully review my dissertation. I also want to thank all the professors whom I interacted with or received guidance from, either in the form of casual discussions or academic courses during my studies at Boston University.

I want to thank the members of the Structural Bioinformatics lab at Boston University for providing the source code for PIPER and FTMap programs and the immense help in understanding the code.

Special thanks to all my friends, and to my colleagues in the department. Particulary, I would like to thank Yongfeng Gu and Matt Chiu for the numerous technical and casual discussions as well as their help and suggestions. I also want to thank the friendly and helpful staff of the ECE department.

Heartfelt thanks to my loving wife who has kept-up with me all these years and has provided moral and emotional support, not just during my PhD but ever since we got to know each other. Last but not the least, I would like to thank my parernts for their constant support in all of my endeavors, including my PhD. Their support has always been crucial in everything I have done.

I dedicate this dissertation to my parents and my lovely wife.

ACCELERATING MOLECULAR DOCKING and BINDING SITE MAPPING USING FPGAs and GPUs

(Order No.)

BHARAT SUKHWANI

Boston University, College of Engineering, 2011

Major Professor: Martin C. Herbordt, PhD,
Professor of Electrical and Computer Engineering

ABSTRACT

Computational accelerators such as Field Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs) possess tremendous compute capabilities and are rapidly becoming viable options for effective high performance computing (HPC). In addition to their huge computational power, these architectures provide further benefits of reduced size and power dissipation. Despite their immense raw capabilities, achieving overall high performance for production HPC applications remains challenging due to programmability, lack of parallelism in existing codes, poor resource utilization, and communication overheads. In this dissertation, we present methods for the effective use of these platforms for the acceleration of two production molecular modeling applications: molecular docking and binding site mapping.

Molecular docking refers to the computational prediction of the structure of the intermolecular complex formed when two independent proteins interact. Binding site mapping, on the other hand, aims at finding the region on the surface of a protein that is likely to bind a small molecule with high affinity. Docking and mapping find application in drug discovery which involves docking-based screening of millions of drug candidates for a given protein target; mapping helps identify the site on the protein where the binding is likely to occur, thus limiting the docking-based search to a small region.

Both docking and binding site mapping are computationally very demanding, requiring many hours to days on a serial processor. This makes it impractical for biologists to run them interactively on their desktop computers; production docking and mapping programs typically run in batch on large clusters. In this dissertation, we present the FPGA and GPU based acceleration of the production molecular docking program PIPER and the production binding site mapping program FTMap, enabling desktop based molecular modeling solutions which are fast and cost effective as well as more power efficient.

The proposed FPGA-docking algorithms achieve multi-hundred-fold speedup of the code that represents over 95% of the original run-time, resulting in 36× overall speedup for small molecule docking. For effective docking of large molecules, we propose GPU accelerated docking algorithms which result in an overall speedup of 18×. The acceleration of mapping computations on FPGAs and GPUs poses further challenges for two reasons: the process is iterative, with relatively little computation per iteration, and a large fraction of the computation is serial. We address these issues on the FPGAs by creating highly customized, deeply pipelined processors. On GPUs, we introduce two new data structures that enable effective parallelization. The result using the GPUs is 6× to 28× speedup on different parts of the algorithm, with an overall speedup for FTMap of 13×. The FPGA-accelerated algorithms obtain 42× performance improvements on the core computations, resulting in an overall speedup of 30×.

Many of the proposed algorithms and hardware structures are general and can

be applied to a variety of other applications, both in the field of molecular modeling as well as other domains such as object recognition, n-body simulations and full-field biomechanics deformation and strain-measurement.

Contents

1 Introduction 1

1.1 The Problem . . . 1

1.2 Molecular Docking . . . 6

1.3 Binding Site Mapping . . . 8

1.4 High Performance Computing with Accelerators . . . 9

1.5 High Performance Molecular Docking and Binding Site Mapping . . . 15

1.6 Summary of Contributions . . . 19

1.6.1 Acceleration of Production Molecular Modeling Codes . . . 19

1.6.2 Potentials and Limitations of Application Accelerators . . . 21

1.6.3 Broader Application and Implications . . . 22

1.7 Organization of the Rest of the Thesis . . . 24

2 High Performance Computing Using FPGAs and GPUs 27

2.1 Overview . . . 27

2.2 Accelerator Based Computing . . . 28

2.3 Background . . . 31

2.4 High Performance Reconfigurable Computing . . . 33

2.4.1 FPGA Overview . . . 33

2.4.2 FPGA Architecture . . . 34

2.4.3 Sources of FPGA Performance . . . 37

2.4.4 FPGA Computing Models . . . 38

2.4.5 FPGAs for High Performance Computing . . . 41

2.4.6 FPGA Systems for High Performance Computing 46
2.4.7 Software Support for High Performance Reconfigurable Computing 53
2.4.8 Challenges with FPGA-based Design 56
2.5 High Performance Computing using Graphics Processors 58
2.5.1 Overview of GPUs 58
2.5.2 GPU Architecture 59
2.5.3 NVIDIA CUDA Programming Model 62
2.5.4 Challenges with GPU Computing 65
2.6 FPGAs versus GPUs 67
2.7 Summary 68

3 Molecular Docking and Binding Site Mapping 70

3.1 Overview 70
3.2 Molecular Docking Basics 72
3.3 Types of Docking 75
3.3.1 Overview of Docking Algorithms 75
3.3.2 Rigid Docking 79
3.3.3 Flexible Docking 85
3.3.4 Energy Minimization 90
3.4 Methods for Evaluating Docking Systems 95
3.5 PIPER Rigid Docking Program 96
3.6 Binding Site Mapping 102
3.6.1 Overview 102
3.6.2 Computational Mapping 104
3.6.3 FTMap 105
3.6.4 FTMap Energy Minimization Step 107

3.6.5 Energy Minimization vs. Molecular Dynamics 112

3.7 Analysis and Summary 114

4 Acceleration of Rigid Molecule Docking 117

4.1 Overview 117

4.2 Algorithms for FPGA Acceleration of Rigid Docking 118

4.2.1 Systolic Array for 3D Correlation 119

4.2.2 Supporting PIPER Energy Functions: Overview 122

4.2.3 Supporting PIPER Energy Functions: Augmented structure . 124

4.2.4 Energy Function versus Cell Complexity 129

4.2.5 Score Filtering on the FPGA 133

4.2.6 Supporting Large Ligands 135

4.2.7 Support for Different Energy Functions 138

4.3 Algorithms for GPU Acceleration of Rigid Docking 140

4.3.1 Direct Correlation 142

4.3.2 FFT Correlation 147

4.3.3 Scoring and Filtering 150

4.4 Divergence in Accelerated Rigid Docking Algorithms 155

4.5 Summary 157

5 Acceleration of Binding Site Mapping 158

5.1 Overview 158

5.2 Algorithms for FPGA Acceleration of Energy Minimization 160

5.2.1 Electrostatics Energy Evaluation on FPGAs: Overview 161

5.2.2 FPGA Pipelines for Electrostatics Energy Computation 163

5.3 Algorithms for GPU Acceleration of Energy Minimization 176

5.3.1 Energy Minimization on GPUs: Overview 177

5.3.2 Energy Minimization on GPUs: Initial Attempts with the Neighbor-lists 179
5.3.3 Improved Data Structures for Energy Minimization on GPUs . 182
5.3.4 Supporting Larger Neighbor-lists and Double Precision Arithmetic: Modified Assignment Table 189
5.4 Summary 193

6 System Design and Integration 195

6.1 Overview 195
6.2 Methods and Tools 195
6.3 System Integration of the FPGA-based Accelerators 200
6.3.1 System Architecture 200
6.3.2 Control Flow 201
6.3.3 Host-board Data Transfer 202
6.3.4 Board-FPGA Data Transfer 203
6.3.5 FPGA Controller FSM 205
6.3.6 Integration into the Production Code 207
6.4 System Integration of the GPU-based Accelerators 210
6.4.1 System Architecture: 210
6.4.2 Control Flow: 212
6.4.3 Integration into the Production Code 213
6.5 Summary 214

7 Results 215

7.1 Overview 215
7.2 Accelerated Molecular Docking 216
7.2.1 Performance Improvements 216
7.2.2 Errors Due to Reduced Precision 221

7.3 Accelerated Binding Site Mapping 223
7.3.1 Performance Improvements of the Energy Minimization Task . 224
7.3.2 Errors Due to Reduced Precision 227
7.4 Scaling to Multiple FPGAs and GPUs 230
7.5 Performance with respect to Cost and Power 231
7.6 Floating Point Utilizations 236
7.7 Summary 238

8 Conclusions and Future Work **239**

8.1 Summary 239
8.2 Observations and Conclusions 240
8.3 Future Directions 243
8.3.1 Improvements to the Current Work 243
8.3.2 Extension to Other Molecular Modeling Applications 245
8.3.3 Application to Other Domains 245

References **247**

Curriculum Vitae **258**

List of Tables

3.1 Sample of Rigid Docking programs. 82

4.1 Number of bits per voxel for computing various energy functions. . . 128

4.2 Limiting FPGA resource for different energy functions 130

4.3 Routing the partial scores to the total score grid 136

4.4 Resource utilization for piecewise correlation on Xilinx Virtex-II Pro VP70 FPGA. Adding support for piecewise correlation has very little overhead. 138

4.5 Direct correlation on a GPU; runtimes for 8 correlations of size 128^3 each. Correlation runtimes depend on the execution configuration as well as the ligand size. 145

7.1 Speedups from GPU on various per-rotation computations in rigid docking. 219

7.2 Speedups for different energy evaluation and force update steps of energy minimization. 224

7.3 FPGA speedups for self-energy and pairwise interaction computations. The runtimes shown are with 2 instances of computation pipelines, running at 185 MHz. 226

7.4 Comparison of the total power consumption for docking and energy-minimizing 16 probes using the FTMap program. 231

7.5 Cost of computation on different hardware platforms. The effective costs on the FPGA and GPU based systems are calculated assuming a 10× performance improvement compared to the CPU based system. 233
7.6 Floating point utilizations achieved on different architectures for the computations in production molecular docking and mapping applications. 237

List of Figures

1·1 The bird flu virus. Shown highlighted is Neuraminidase, a surface protein that is the main target for the immune system [NSc10] 3

2·1 Typical hardware set-up for an accelerator-based system (based on [Bha07]) 29
2·2 Typical usage scenario for an accelerator-based system 30
2·3 Configurable logic and interconnects inside an FPGA [DeH00] 35
2·4 Comparison of performance capabilities of an Intel quad-core processor and a Xilinx Virtex-5 FPGA (based on [Bol10] with some updates) . 37
2·5 Annapolis Micro Systems' Wildstar II Pro PCI board [II03] 47
2·6 Gidel's PROCe III system block diagram [Gid09a] 48
2·7 SGI RASC RC 100 blade [FL06] 49
2·8 SGI RASC software architecture [Sil04] 50
2·9 SRC Series H MAP processor [SRC10a] 51
2·10 XtremeData XD1000 [XDI07] 52
2·11 Floating point datapaths (a) interconnected IP cores, and (b) "fused" datapath using the Altera FPC. [Dai10] 55
2·12 Architecture of NVIDIA TESLA C1060 [NVI10] 60
2·13 Memory hierarchy of the NVIDIA GPUs [NVI08b] 61
2·14 Organization of threads into blocks and grid [NVI08b] 64

3·1 Interaction between two molecules (generated using Pymol [Pym10]) . 72

3·2 Different representations of molecular surface; (a) Grid, (b) Overlapping spheres [SKB92], and (c) Triangles [MST10] 75
3·3 (a) Protein-protein docking and (b) Protein-ligand docking 78
3·4 Lock and key model used by rigid docking programs 80
3·5 Iterative process of energy minimization 92
3·6 Various bonded and non-bonded interactions between atoms 93
3·7 Shape complementarity between two molecules (based on [VGH04]) . 99
3·8 PIPER program flow 100
3·9 Runtimes per rotation for different steps of PIPER rigid docking . . . 101
3·10 FTMap program flow 105
3·11 Profiling result of the FTMap program 106
3·12 Profiling of the energy minimization step of serial FTMap program, (b) Distribution of the energy evaluation time among the computation of different energy terms. 108
3·13 Neighbor lists 112

4·1 Structures to compute 3D correlations: (a) standard 1D systolic array, (b) extension to 2D correlation with delay lines, and (c) full 3D correlation with delay “planes”. 121
4·2 Three methods of combining multiple correlations in a single pass: (a) within a compute cell, (b) upon completion, or (c) integrated into the scoring function. 123
4·3 2D correlation pipeline modified to support complex correlations. The 3D extension is analogous. 124
4·4 Block RAM requirement increases with the increase in the complexity of the scoring function; the modified pipeline reduces this requirement by almost 40%. 125

4·5 Internals of a correlation compute cell for PIPER energy functions. . 126
4·6 Details of the end-of-row score combiner. It is pipelined to enable high operating frequency. 127
4·7 Cell operations and FPGA resource requirements for various energy functions. 129
4·8 Complexity of correlation cell versus number of cells along each dimension for various FPGA technologies. 131
4·9 Speedup of the FPGA accelerator over a single core against ligand size, for various energy functions; only the correlation task is evaluated here. 132
4·10 Score filtering; the result grid is divided into small cubes. The best score from each cube (shown as a red dot) is returned to the host. . . 134
4·11 Piece-wise correlation [Gu08]; scores from different quadrants must be routed appropriately. 135
4·12 Block diagram of piece-wise docking for computations with large ligand molecules. 137
4·13 Direct correlation on GPU; larger grid is stored on the device global memory while the smaller grid is copied to the shared memory of each multiprocessor. 143
4·14 Distribution of work on the GPU for direct correlation: (Left) Each thread block works on part of every 2D plane, (Right) Different 2D planes are assigned in their entirety to different blocks. 144
4·15 Scoring and filtering on the GPU: (a) Different coefficients are distributed among different multiprocessors, with each containing (b) one thread, or (c) multiple threads. 152

4·16 Correlation-only speedups for PIPER energy function (with 4 pairwise potential terms; 8 correlations). The horizontal axis is the edge-size of the ligand grid. 155

5·1 Data structures: (a) Neighbor-lists used in the serial FTMap code (b) Pairs-list used for the FPGA-accelerated FTMap 162

5·2 Electrostatics computations require two separate pipelines; shown are the two alternatives for organizing these pipelines. 163

5·3 Self-energy pipeline on the FPGA; For each input pair, the pairwise values are stored straight into the BRAMs whereas the partial atom-wise values are accumulated into the total values. 165

5·4 Generalized Born pipeline on the FPGA; The self energies are used to compute the atom Born radii which are then fed to the g-B pipeline. 168

5·5 Two parts of the generalized Born pipeline [Dai10]; (a) first part processes the atom-pairs to compute the pairwise energies and forces (b) second part processes individual atoms to compute the second component of the force. 170

5·6 Routing of atom pairs to multiple pipelines (a) separate routers and FIFOs (b) unified routing. 171

5·7 Complete electrostatics pipeline with two Eself and two g-B pipelines. 173

5·8 4-port BRAM constructed by replicating dual-ported BRAMs. 174

5·9 Data structures on the GPU: (a) Neighbor-lists lead to write conflicts (b) Pairs-list enables parallel writes. 179

5·10 Mapping the neighbor-lists onto GPU threads. Replicating the energy arrays to enable parallel updates. 180

5·11 Split pairs-lists: (Left) Forward list, (Right) Reverse list. 184

5·12 Assignment table for the GPU; atom-groups are mapped to the GPU thread-blocks in their entirety. 185
5·13 Multi-pass assignment table; threads process pairs from different groups in different passes. Some of the threads may remain idle during some of the passes (shown with -1 in the corresponding rows). 187
5·14 Threads compute partial energies in parallel and store in the shared memory. Master threads then perform the accumulations from the shared memory and store the accumulated value in the global memory. 188
5·15 Assignment table to support group-splitting; Group 0 has 13 pairs and there are only 5 threads per block – it is thus split into 3 sub-groups. With the number of atoms in the system = 30, the Global Ids start from 31. The second-level master (with Master = 1) has the corresponding Split Id = 31. Other master threads (Master = 0) do not perform second level accumulations. (The table shown is just an example; actual assignment tables contain tens of thousands of rows). 191

6·1 System architecture for FPGA-accelerated docking and mapping. . . 200
6·2 Flow of control between the software and the FPGA modules. 201
6·3 FPGA–on-board memory interface for (a) molecular docking and (b) energy minimization 203
6·4 Controller FSM on the FPGA (a) for molecular docking and (b) for energy minimization 206
6·5 System architecture for GPU-accelerated docking and mapping. . . . 211
6·6 Flow of control between the host and the GPU. 211

7·1 Speedups on different architectures for the correlation-only task. Speedups shown are for the PIPER energy functions with 4 pairwise potential terms (8 correlations). The reference baseline is the best correlation method on a single core. 217

7·2 End-to-end speedup of the PIPER program with 18 pairwise potential terms (22 correlations). The reference baseline is the original PIPER code using FFT correlations. 220

7·3 10 different protein-pairs from the Docking benchmark used for the evaluating the accelerated-PIPER code. 222

7·4 Distribution of errors in the results generated by the accelerated docking. Results shown are for 10 different receptor-ligand pairs from the docking benchmark; each was run for 5000 rotations. 223

7·5 Average error in the energy and force values after different number of minimization iterations. 228

7·6 Cost of docking using the PIPER program on different architectures. 234

List of Abbreviations

ACE	Analytic Continuum Electrostatics
ASIC	Application Specific Integrated Circuit
b	Bit
B	Byte
BCB	Bioinformatics and Computational Biology
BRAM	Block Random Access Memory
CAD	Computer-Aided Design
CHARMM	Chemistry at HARvard Molecular Mechanics
COTS	Commercial off-the-shelf
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DDR	Double Data Rate
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processor
EDA	Electronic Design Automation
FFT	Fast Fourier Transform
FIFO	First In First Out
FLOPs	Floating-point operations per second
FPC	Floating Point Compiler
FPGA	Field Programmable Gate Array
FSB	Front Side Bus
Gb	Gigabits, 2^{30} (10^9) bits
GB	Gigabytes, 2^{30} (10^9) bytes
GFLOPs	Giga FLOPS, 10^9 floating-point operations per second
GHz	Giga Hertz, 10^9 cycles per second
GPGPU	General-Purpose computing on Graphics Processing Units
GPP	General Purpose Processors
GPU	Graphics Processing Unit

GUI	Graphical User Interface
HDL	Hardware Description Language
HLL	High Level Language
HPC	High Performance Computing
HPRC	High Performance Reconfigurable Computing
IP	Intellectual Property
Kb	Kilobits, 2^{10} (10^3) bits
KB	Kilobytes, 2^{10} (10^3) bytes
L-BFGS	Limited-memory Broyden-Fletcher-Goldfarb-Shanno method
LSB	Least Significant Bit
LUT	Look Up Table
MAC	Multiplication and Accumulation
Mb	Megabits, 2^{20} (10^6) bits
MB	Megabytes, 2^{20} (10^6) bytes
MD	Molecular Dynamics
MHz	Mega Hertz, 10^6 cycles per second
MPP	Massively Parallel Processor
MSB	Most Significant Bit
PAL	Programmable Array Logic
PAR	Place And Route
PC	Personal Computer
PCI	Peripheral Component Interconnect
PCIe	PCI express
PDB	Protein Data Bank
PE	Processing Element
PLA	Programmable Logic Array
PLD	Programmable Logic Device
RAM	Random Access Memory
RTL	Register Transfer Level
SIMD	Single Instruction, Multiple Data
SIMT	Single Instruction, Multiple Thread
SRAM	Static Random Access Memory
TB	Terabytes, 2^{40} (10^{12}) bytes
TFLOPs	Tera FLOPS, 10^{12} floating-point operations per second
UI	User Interface
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLIW	Very Long Instruction Word

Chapter 1

Introduction

1.1 The Problem

"Scientists and researchers' insatiable need to perform more and larger computations has long exceeded the capabilities of conventional computers" [LG09]; this has led to increasing efforts in finding new and efficient computing platforms. Historically, such high performance computing has been done using dedicated supercomputers such as the CRAY-1 and the STAR vector machines. Starting from mid-1990s, however, commodity clusters have dominated the HPC domain. Though these architectures remain the predominant choice, the search for more power efficient and performance effective solutions continues. The President's Information Technology Advisory Committee's report in 2005 recommended that research and development investments be made to "design, prototype, and evaluate new hardware architectures that can deliver larger fractions of peak hardware performance on scientific applications" [Pre05]. One of the directions of recent research is the use of the application accelerators for compute-bound problems. Two such acceleration architectures that have gained widespread attention are Field Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs). FPGAs are commodity hardware which can be (re)configured into different logic circuits after the chip has been fabricated whereas GPUs are special-purpose hardware chips optimized for efficient graphics rendering. Their huge computational capabilities, combined with the opportunity for more power-efficient computing, has made these architectures attractive for accelerating a wide variety of computationally intensive applications.

Even though application acceleration has received much attention, the field is still in its infancy and most of the solutions are ad hoc. A number of high performance applications that can benefit from accelerator-based computing have still not been addressed, perhaps due to the challenges in designing accelerator based solutions. The problem is designing efficient solutions that result in overall high performance. Creating application accelerators poses numerous challenges such as the need to have a good understanding of both the application at hand and the accelerator hardware, the limited parallelism in the computations, the limited resources on the accelerators, architecture-aware redesigning of the underlying algorithms, the lack of appropriate programming models for the accelerators, designing efficient acceleration-cores, and minimizing the overheads of system integration. In the current work, we investigate the amenability of accelerator-based computing for HPC applications. In particular, we address these problems with respect to the acceleration of applications used for modeling the interactions between two molecules. Though these applications are both high-impact as well as require immense computational capabilities, their acceleration has largely not been addressed.

Modeling the molecular interactions is critical both to developing an understanding of the life processes as well as to performing effective drug discovery. The discovery and development of a new drug is a very expensive and time consuming process. Developing a new drug costs nearly a billion dollars and takes about 15 years. “The cost of drug development has risen markedly in the past 30 years, with studies now reporting values exceeding US $800 million. As these spiralling costs threaten to make the development of new drugs increasingly unaffordable ... efforts should be made to address this problem. All aspects of the drug discovery and development process should be examined for potential cost savings ...” [Raw04]. One of the reasons for such high cost is the computational complexity of the algorithms used in the process of drug discovery, leading to longer discovery times. Development of fast and more cost-effective methods for drug-discovery is a long-sought goal.

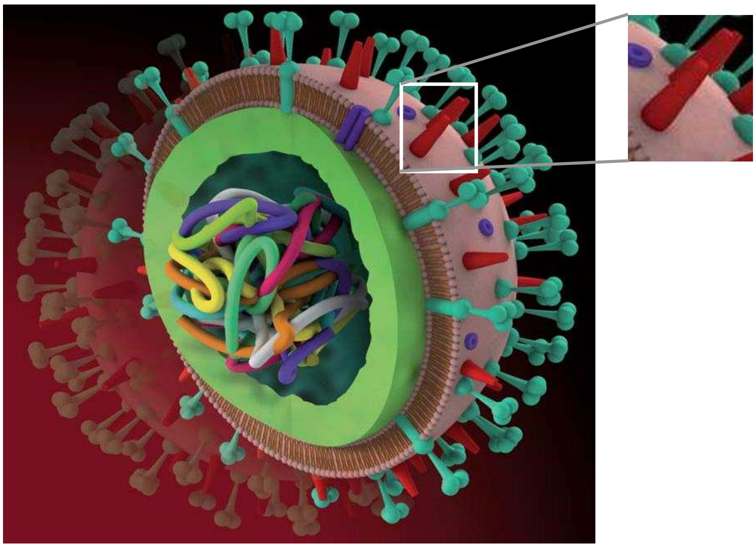


Figure 1·1: The bird flu virus. Shown highlighted is Neuraminidase, a surface protein that is the main target for the immune system [NSc10]

One of the main computations in drug discovery is modeling the interactions between a target protein and a candidate ligand. Proteins are linear polymers made of 20 different amino acids and folded into three-dimensional structures. They assist in a variety of essential biological processes including the immune system, the mechanical and structural support of the body, the transmission of the nerve pulses and the transportation of oxygen [BTS10]. The function of a protein is determined primarily by its three-dimensional structure. Ligands are small molecules that bind to the proteins and alter their functions. They can act as inhibitors and activators, selectively blocking or enhancing certain functions of the proteins. The behavior of the proteins, and hence the biological processes of life, can thus be controlled by binding certain ligands to the proteins. This is one of the fundamentals of drug design and discovery.

Discovering a drug to cure a disease involves finding the appropriate ligand that binds to the disease-causing protein and selectively inhibits its function without affecting other proteins. Consider the example of the bird flu virus shown in Figure 1·1. The red, tube-like structure shown highlighted is Neuraminidase, a surface protein that helps new viruses bud off from the cells [NSc10]. The function of a drug, in this case a Neuraminidase inhibitor (e.g. Tamiflu), is to bind to this protein and prevent the release of new viruses [NSc10].

Finding the appropriate ligand requires screening millions of drug candidates against the target protein. Experimentally determining the interactions between the two molecules in a wet-lab is both expensive as well as time consuming; computational methods are thus applied for faster screening of large ligand databases.

The process of computationally modeling the interactions between two molecules is called molecular docking. There are two main aims of docking: accurate structural modelling and correct prediction of the chemical activities [KDFB04]. Based on these, the docking algorithms can be classified into two categories: protein-ligand docking, which involves docking a small molecule into a pocket in a larger molecule, and protein-protein docking, which docks two large molecules. The former of these finds application in drug discovery while the latter helps understand the basic processes of life by predicting the structure formed when two proteins interact. In both these classes of docking, based on whether the molecules' flexibility is taken into consideration, the docking methods can be classified as rigid-body or flexible docking.

Most of the current state-of-the-art methods in the field of protein-protein docking perform rigid-body docking followed by the refinement of the docked structures [VK09]. For effective ligand docking, however, modeling the flexibilities in the ligand's side chain is essential for accurate predictions. An effective technique for this is to perform the energy minimization of the protein-ligand complex, allowing for the ligand side chains to move freely. Another promising approach in docking a ligand to a protein is the identification of the binding sites, the regions on the protein surface that contribute most towards the binding of the ligand to the protein, using small organic probes. This process is called binding site mapping and involves a combination of rigid-docking and energy minimization steps. Rigid docking and energy minimization, thus, play an important role in the current state-of-the-art molecular modeling techniques.

Both rigid-docking as well as energy minimization, however, are very time consuming. Docking a pair of typical-sized molecules (few-hundred residues) on a serial computer can take up to 30 hours. Though energy minimizing a protein-ligand pair typically takes less than a minute of compute time, minimization is performed on thousands of protein-ligand conformations, making the total runtime too long to be of any practical use. To keep the computational runtimes within reasonable limits, most high-end docking and mapping systems typically run on large clusters. One of the biggest challenges is to reduce the computational runtimes of these systems so as to enable desktop-based solutions as well as allow for more complex modeling of the chemical interactions for higher simulation accuracy.

The current dissertation investigates the problem of improving the performance of various molecular modeling applications using special purpose hardware, in order to provide the researchers with a more cost-effective, desktop based alternative to cluster-based solutions. In particular, we investigate the use of Field Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs) for effective acceleration of molecular docking and binding site mapping algorithms.

Field programmable gate arrays are a class of reconfigurable integrated circuits whose logic can be determined in the field, after the chip has been fabricated. FPGAs contain logic and interconnects that can be programmed in the field to generate different circuits. This feature allows them to be used for application acceleration since the hardware logic can be optimized for the computations at hand. Moreover, the configurable interconnects allow for high utilization of the available hardware resources, thus maximizing parallelism and hence the performance.

Graphics processing units are processor chips designed mainly for performing efficient graphics rendering. GPUs contain hundreds of parallel computation units on a single chip, providing very high computational capabilities. Though the interconnection among these processing cores are not as flexible as on the FPGAs, the vast raw computational power make them an attractive platform for accelerating a variety of applications.

In spite of the immense computational capabilities and the high flexibility of these architectures, achieving good performance for complex scientific applications in general, and docking and energy minimization computations in particular, presents many challenges. In the current work, we address these by significantly restructuring the computations, using highly-optimized, hardware-efficient pipelines on the FPGAs and complex data-structures to maximize parallelism and minimize communication on the GPUs.

1.2 Molecular Docking

A fundamental operation in biochemistry is the interaction of molecules through non-covalent bonding; molecular docking refers to the modeling of this interaction through computer simulation. Docking involves predicting the complex-structure that is formed between two interacting molecules. A basic computation in docking is to find the relative offset and rotation (pose) that gives the strongest interaction.

Based on the search criteria and the approximations applied, there exist a vast variety of docking algorithms. These include exhaustive search algorithms, molecular dynamics based docking algorithms, and heuristic algorithms such as those based on simulated annealing and genetic algorithms. Among the different classes of docking algorithms, rigid docking, a process wherein the interacting molecules are assumed to be rigid-bodies, is very widely used. Moreover, it also forms the basis for other docking algorithms such as fragment based flexible docking and solvent mapping. Rigid docking often involves exhaustive 6D search to evaluate billions of relative poses between the two interacting molecules. This is computationally very demanding, requiring many hours of runtime on a serial processor.

In this dissertation, we examine a variety of molecular docking algorithms with respect to their need for, and amenability to, acceleration. In particular, one of the algorithms addressed in this work is the grid-based, exhaustive search rigid docking, which is one of the most computationally demanding process among different docking computations. The aim of rigid docking is to exhaustively evaluate all possible poses (relative offset and rotation) between the two molecules being docked and return the ones that result in the most favorable interactions. The goodness of a pose is measured in terms of the shape and other properties of the two molecules. For each molecule, these properties are mapped onto a 3D grid. The docking process then involves holding one of these grids fixed (typically the larger one, called the receptor) while rotating and translating the other (the ligand) around it. The pose score is then computed as a correlation between the two grids. Typically, tens of thousands of rotations are performed. For molecule grids with edge size of N, the complexity of evaluating each rotation is $O(N^6)$. With the introduction of FFT-correlation to molecular docking, this complexity reduced to $O(N^3 log N)$. Even with this optimization, the absolute runtime on a serial processor is prohibitively large; for typical grid sizes, evaluating 10,000 rotations can take up to 28 hours of runtime on a modern processor. Often, more than 10,000 rotations are used in actual docking runs. With many ligands being docked to a given receptor, as is the case for drug-discovery, performing rigid docking on a serial processor is impractical; production docking settings typically use large clusters with thousands of nodes.

1.3 Binding Site Mapping

Binding sites are regions on a protein surface that are likely to bind a small molecule with high affinity. Mapping is the process of finding these sites using a set of standard small molecule probes. It is a relatively new technique used for performing drug discovery.

Mapping involves flexibly docking a variety of small molecules to the given protein and finding a consensus site that binds most of those probes. Due to the computational complexity of flexible docking, the process is often split into two steps: the first performs rigid docking between the protein and the probe; the second models the side chain flexibility by energy-minimizing the (few thousand) top scoring protein-probe complexes generated by the first step. Both these steps are computationally very expensive, requiring many hours of runtime per probe on a serial CPU.

In this dissertation, in addition to the acceleration of rigid docking, we also addresses the acceleration of the process of energy-minimization. Minimization is an iterative process and involves repeated evaluation of various bonded and non-bonded energies of a protein-probe complex, until some convergence criteria are met. Often, up to a thousand iterations can be performed, requiring runtimes of about 30 seconds per complex. With many thousand complexes to be minimized, the total runtime on a serial processor is typically many hours.

In addition to the binding site mapping, energy minimization is also used in various docking programs such as DOCK [MGBK93], DARWIN [TB00], RDOCK [LCW03] and EADock [GZM07]. Acceleration of energy minimization is thus highly desirable, as it will lead to faster molecular docking and quicker turn-around times in drug discovery.

1.4 High Performance Computing with Accelerators

Mainstream high performance computing has traditionally achieved performance in three ways: algorithmic advancements, improvements in the processor hardware or following the model of *throwing more hardware* at the problem. With the first two approaches having their limits, computationally intensive and HPC applications are often addressed using large clusters of microprocessors. In recent years, however, there have been some shifts from this approach and the accelerator-based high performance computing has gained some attention. Examples of these accelerators include FPGAs, graphics processors, Cell broadband engine, etc. Even though most of the high-end systems currently being used by the high performance computing world are based on collections of commodity microprocessors, there has been increasing interest in the accelerators from the HPC community.

The main reason for this shift in paradigm is the increasing power consumption of the serial microprocessors. Microprocessors have long been riding the bandwagon of Moore's law, with the increasing chip densities delivering ever increasing performance. Though this trend of doubling the number of transistors on the chip every 18 months still continues, power concerns have stagnated the operating frequencies at around 4 GHz. Due to this, the computer industry has moved towards parallelism for obtaining higher performance. The result is having multiple homogeneous processor cores on the same die. Though this provides a way of utilizing the chip resources efficiently, programming the applications so as to make use of this available parallelism is still a challenge – “Harnessing the power of multicore processors is one of the largest challenges facing the computer industry today [Ead07]”. Moreover, microprocessors, including the multi-core processors, are designed for general-purpose computing and are not optimized for high performance computing. For instance, microprocessors, unlike the graphics processors, devote a large portion of the chip area to cache rather than the processing cores. This is not particularly beneficial for the compute-bound HPC applications.

Another reason for the increasing popularity of the accelerators is the increase in their computational capabilities. Like microprocessors, other architectures such as FPGAs and GPUs also benefit from the increasing chip densities, making them capable of delivering unprecedentedly high computational payloads at a fraction of the cost of microprocessor based clusters. Modern FPGAs contain hundreds of embedded memories and hard multipliers, along with one or two embedded processors, making them suitable for a wide variety of high performance computing applications, even those requiring floating point computations. Modern graphics processors are capable of delivering peak floating point performance close to a tera-flop. Moreover, due to the multi-billion dollar gaming industry, GPU chips are manufactured and sold in large volumes, driving the per-chip cost very low.

Large cooling requirements for the microprocessor based clusters have also played a role in the growing popularity of FPGA and GPU based accelerators. Both these architectures have a much lower power per flop requirements compared to the large clusters. A single FPGA chip has typical power dissipation of about 20 watts. Even though the GPUs have higher power dissipation (around 180 watts for a 240 core GPU), they still have a much smaller power to core ratio than modern microprocessors. Microprocessor based clusters, on the other hand, dissipate hundreds of kilo watts to a few mega watts of power. Compared to the NVIDIA TESLA C1060 GPU, the RoadRunner supercomputer at Los Alamos National Labs [Lab10] delivers $1,600\times$ higher performance but dissipates $10,000\times$ more power. Ironically, the same supercomputer is considered one of the greenest among the current supercomputers. The high recurring cost of cooling these clusters is becoming an important criterion while selecting any new HPC solution. This has led to the development of GPU-based supercomputers [Top10, Tok10] and some FPGA-based research clusters [fHPRC10].

Advances in Accelerator Systems

Originally, FPGA and GPU based accelerators were treated mainly as peripheral devices, loosely coupled to the main processor via the PCI bus. The growing demand for accelerator-based computing has led to the advent of accelerator systems that are more tightly coupled to the main processors. One such example is the socket-based FPGA card wherein the FPGA sits in one of the processor sockets of a multi-processor motherboard and acts as a peer to the main processor. Examples include boards from XtremeData Inc. [XDI10] and DRC computers [DRC10]. More recently, processor manufacturers have been investigating designs with the accelerator sitting next to the processor on the same die. Examples include the Intel Larrabee architecture and the AMD/ATI Fusion. Both these processors contain a graphics processing engine alongside the main processor on a single die / package. Though these architectures are still in the development phase, they clearly indicate the recognition of the potential of accelerator-based HPC designs from the chip manufacturers and the growing support for the same.

Challenges with Accelerator-based Computing

Though accelerator based designs provide promising alternatives to the traditional cluster based computing, they do present some challenges that must be addressed. We discuss them here briefly. Some of these are elaborated upon in later chapters.

- **Amdahl's Law:** The first and foremost challenge with accelerator-based designs is Amdahl's law. Even with 80% of the computations accelerated, the overall system speedup remains limited to only 5x. In order to achieve higher speedups, it is thus important to move as much computation to the accelerator side as possible.

- **Architecture-aware design:** The computation mode and the data structures most suitable on serial processors are often not amenable to hardware implementation. Efficient mapping of the software algorithms on the hardware resources requires careful restructuring of the computations and reorganization of the data. Moreover, achieving high performance requires efficient utilization of the available hardware resources and the memory hierarchy.

- **Overhead:** The overhead of host-accelerator communication can sometimes nullify the benefit obtained from the accelerated computations. Partitioning the code and structuring the computations in a way so as to minimize this overhead is not always trivial.

- **Precision manipulations:** Accelerators often benefit from computations with reduced precision. For example, on the FPGAs, having lower precision enables more replication of the processing elements, thus increasing the performance. Reduced precision, however, can lead to errors in the final result. Obtaining results with tolerable errors requires carefully determining the precision requirements.

- **Limited resources:** Accelerators, unlike serial processors, aim to perform the computations in parallel and the achievable performance is directly related to the fraction of the computations that can *fit on the chip.* Resources on a chip, however, are limited and top-out at some point, leading to the need for having multiple passes through the accelerator. This results in reduced performance. Careful design of the accelerator with efficient use of the available resources is thus very critical.

- **System integration:** Integration of the accelerator module into the rest of the system is one of the most challenging tasks. If careful measures are not taken, all the achieved performance improvements can be easily lost in this step.

Cost of Accelerator-based Computing

There are various kinds of costs associated with computing. These include the cost of acquiring the hardware platform, the development cost, the cost for the power and the cost for maintaining the code and porting to newer-generation hardware. Here we briefly discuss these with respect to the FPGAs and GPUs.

- **Cost of ownership:** The cost of a high-end FPGA system is in the range of few-thousand to a few tens of thousands of dollars. This high cost is primarily due to the niche market served by these devices. GPUs, on the other hand, benefit from the large-volume gaming industry that drives the cost very low, with most GPUs costing somewhere between $300 to $1500. This is much lower than the cost of a multicore processor which can cost somewhere in the range of a few thousand dollars.

- **Power cost:** The cost of power for the FPGAs is about $1/4^{th}$ of that for a multicore processor while that for the GPUs is comparable to the multicore.

- **Development cost:** The cost of developing efficient algorithms on the FPGAs is higher compared to the serial processors and the GPUs. A large portion of the total efforts in the FPGA design is spent towards the integration of the accelerated routines into the rest of the system. This is mainly due to the lack of the standardized host-board interface. GPUs, on the other hand, do not suffer from this issue due to the availability of standardized interface APIs and high-level design languages. The actual development cost on these platforms is hard to quantify as it depends on a number of factors such as the complexity of the underlying algorithm and the extent of optimizations performed to improve the performance.

- **Cost of porting to new hardware:** Computer hardware undergoes rapid changes, with larger and improved chips becoming available every couple of years. This necessitates the porting of the algorithms to these new hardware to benefit from their improved capabilities. Porting on the serial processor simply involves re-compiling the code. Porting on the accelerator-based system, however, is not always so trivial.

  Porting to newer generation FPGA chips requires scaling the design to utilize the increased hardware resources. This can be as simple as replicating the pipelines or may require redesigning the pipelines for more efficient computation. GPU-accelerated programs, on the other hand, can benefit from the increased number of processor cores simply by changing some configuration parameters. This is due to the restrictive nature of the current GPU programming model. A more elaborate change to the GPU architecture, however, could require higher porting efforts.

In general, porting the FPGA and GPU accelerated designs to newer generation devices can require varying amount of efforts, depending on the extent of the change in the hardware.

1.5 High Performance Molecular Docking and Binding Site Mapping

Molecular docking and binding site mapping are important tools for modeling the molecular interactions. Their applicability and effectiveness in the complex fields of life sciences and drug discovery has called for high performance solutions that can keep the computation runtimes within tolerable limits. Due to this, the parallelization of these algorithms has been widely addressed, though mostly at a coarse level.

The computations in both molecular docking as well as mapping lend themselves well to distribution across a large number of processor nodes. For example, different rotations of a rigid-docking run are totally independent of each other and can be executed in parallel. Similarly, different degrees of freedom in flexible docking can be explored simultaneously. Parallelism at an even higher level is also possible; while screening large databases of ligands for discovering a potential drug candidate, the database can be distributed across different processor nodes and searched in parallel. In the case of mapping, where different protein-probe complexes are energy-minimized, similar coarse-level parallelism can be trivially attained; the minimization of different complexes are independent and can be executed simultaneously.

The large computation runtimes of these application, combined with the immense coarse-level parallelism afforded by them, has led to many parallel, MPI-enabled solutions, running on large clusters. Most production docking and mapping codes are executed on supercomputers, finishing the computation in few minutes. Examples of some public domain docking and mapping servers running on processor-clusters include ZDOCK [Wen10], ClusPro [SBL10a], FTMap [SBL10b], GRAMM-X [CfB10] and PIER [oPPS10]. Moreover, pharmaceutical companies employ their own proprietary in-house docking programs and servers.

The problem with the cluster based solution, though, is three fold: first is its high cost. Small academic labs and independent researchers cannot afford the huge cost associated with acquiring and maintaining such large supercomputers. In recent years, due to the large power dissipations, the cost of cooling these systems has also become significant. Secondly, researchers wanting to make certain algorithmic modifications to try and improve the results cannot do so; though the public domain solutions can help various researchers perform the docking task, incorporating any algorithmic modifications is not possible. The third problem relates to the logistics involved with using a shared resource: requesting processor time, submitting batch jobs and in case of a restart due to some errors or modifications in the parameters, waiting for the next allocation of the resources. Overall, this leads to long turnaround times. A desktop based solution is thus desirable.

Improving the performance of complex algorithms on a desktop-based solution involves either improving the computational efficiency of the underlying algorithm or speeding-up the computationally demanding tasks using some special-purpose processor. The latter is commonly referred to as accelerator-based design and is the point of the current discussion.

The use of accelerators to improve the performance of computationally intensive applications is both desired as well as promising. Even though computationally demanding applications have been traditionally addressed using clusters of microprocessors, the growing cost and power concerns call for more cost-effective alternatives. Among the different accelerator architectures, FPGAs and GPUs appear to be most promising and have been shown to achieve multi-hundred fold speedups on a variety of financial and scientific computations.

While accelerating docking and mapping tasks using heterogeneous parallel processors has clear and obvious benefits, there has been surprisingly little work thus far. One of the efforts of accelerating a docking program is the commercial product eHITS developed by SymBioSys [May08]. eHITS uses the IBM Cell Broadband Engine to accelerate the docking task. In another study, Servat, et. al. analyzed different parallelization strategies to port the FTDock protein-protein docking program on the Cell processor, reporting a speedup of 3× compared to a Power5 processor [SGAA$^+$08].

In our opinion, the only FPGA-based acceleration of a molecular docking program, other than our work, is the acceleration of the AutoDock program [Pec09]. The authors report implementing the genetic algorithm based docking on an FPGA, resulting in speedups in the range of 8× to 14×. The genetic algorithm implemented, however, is quite different from that used in the serial AutoDock program, resulting in some differences in the results [Pec09].

On the GPUs, we could not find any study that addresses the acceleration of the complete docking task. The only published work so far appears to be in a dissertation by Korb [Kor08]. There, the structure transformation and scoring function evaluation phases are accelerated. For the mapping algorithm, we could not find any published work that addresses its acceleration on any of the special-purpose processors.

In this work, we address the acceleration of molecular docking and binding site mapping at a fine-grained level. Though these computations are easily parallelizable across a large number of processors, accelerating them on a single chip poses a number of challenges. The first is simply the Amdahl's law. Even though both docking and mapping algorithms contain a large computational core, the rest of the computations also contribute significantly towards the total runtime. Their effect becomes even more severe once the most time consuming steps are accelerated. Achieving good overall performance, thus, requires accelerating as much computation as possible. We address this issue by moving most of the computations to the accelerator side and hiding the latencies of the remaining computations that are performed on the host.

The second challenge is to find and use the correct computation mode on the accelerator. The computation mode applied in serial docking algorithm is not very suitable for FPGA implementation. On the FPGAs, we use an entirely different model that results in very efficient hardware pipeline. On the GPUs, based on the problem size, two different computation modes are utilized.

The third challenge comes from the effective utilization of the available resources. A key method in obtaining high performance from accelerator-based designs is to achieve maximum parallelism through high utilization. In the case of the energy minimization computations, the data structures used in the serial code result in very poor utilization of the available processors on the GPU. On the FPGA, though high utilizations and parallelism can be achieved by replicating the processor cores, this is not feasible for the minimization task. The complexity of the energy minimization calculations and the use of high-precision result in large resources requirements for the processing core, thus forbidding any replications on the current generation FPGAs. We address these issues by using a new data structure on GPUs and very deeply pipelined, customized processors on FPGAs.

Our proposed methods and optimizations enable efficient, architecture-aware implementation of the docking and mapping computations on FPGAs and GPUs. For molecular docking, based on the problem size, our methods achieve speedups in the range of 22× to 1400× on the FPGA and 14× to 260× on the GPU for the core computations. For energy minimization, the performance improvement on the core computation is between 7× to 27× on the GPU and 42× on the FPGA. This, combined with the low power and cost, makes this solution very attractive.

1.6 Summary of Contributions

The contributions of this research can be classified into three broad categories (i) acceleration of production molecular modeling applications, (ii) investigation of the potentials and the limitations of the accelerator-based computing for HPC applications, and (iii) broader applications of the proposed algorithms and the implications of the corresponding research findings to other computations. We now describe these in more detail.

1.6.1 Acceleration of Production Molecular Modeling Codes

One of the main aims of the current research is the advancement of life sciences research by accelerating the production molecular modeling applications. In the current work, we analyze different molecular modeling algorithms and present efficient hardware algorithms for the computationally demanding steps. Moreover, we accelerate two highly used, production, molecular modeling software, obtaining significant overall speedup over the original software codes, while maintaining good agreement with original results.

(a) Analysis of the Docking Algorithms

We analyze different molecular docking algorithms with respect to their need and suitability for acceleration using FPGAs and GPUs. We find the most computationally intensive steps in these algorithms and investigate the potential performance benefits from, and the corresponding challenges in, accelerating them.

(b) Algorithms for Accelerating Docking Computations

We present efficient hardware algorithms on FPGAs and GPUs for accelerating the common computations in molecular docking. We perform significant restructuring of the underlying computations and the data structures to enable efficient mapping to these architectures, achieving one to two orders of magnitude speedups on the core computations. The proposed algorithms can be applied, either singly or in combination, to accelerate a variety of docking programs.

(c) Acceleration of a Production Rigid Docking Code

The production molecular docking program targeted in the current work is called PIPER [KBCV06]. PIPER rigid docking program has consistently performed well in the community wide molecular modeling experiment, CAPRI [JHM$^+$03]. Moreover, PIPER is used as the first step in protein docking and clustering program ClusPro [CGVC04] as well as the mapping program FTMap [BKC$^+$09]. Both ClusPro and FTMap are available for public use through web servers. Acceleration of PIPER program, thus, results in a great impact in the field of molecular modeling.

The current work achieves multi-hundredfold speedup on the computations that account for close to 95% of the total runtime. The overall end-to-end speedup varies from 20× to 37×.

On the FPGA, we achieve large speedups by restructuring the original computations and using a computation mode better suited for the hardware implementation. We propose an efficient hardware structure for the computation of multiple correlations required for PIPER. Additionally, we provide GPU computation kernels for both direct as well as FFT correlations, along with the acceleration of almost every other computation performed by the PIPER program.

(d) Acceleration of a Production Binding Site Mapping Code

The most time consuming computation in binding site mapping is energy minimization. While energy minimization can potentially benefit from acceleration, little to no work has been done in this area, especially none on a production code. The reason, perhaps, is the small amount of computation per iteration, making it difficult to achieve any speedup. In the current work, we address this by using efficient, custom-designed hardware pipelines on the FPGA and two new data structures for efficient work distribution on the GPU; our approach achieves speedups ranging from 15× to 100× on the core computation.

We integrate the accelerated docking and energy minimization routines in the production mapping program FTMap [BKC+09], resulting in overall speedup in the range of 12× to 27×.

(e) A More Power Efficient Solution

Due to the inherently lower power consumption of the FPGAs and the lower power-per-flop requirements of the GPUs, one of the accompanying contributions of this work is a low-power solution to the addressed problems. Our proposed solutions result in 92% power savings on the FPGAs and 44% savings on the GPUs.

1.6.2 Potentials and Limitations of Application Accelerators

Based on the structure of the underlying computation, the precision requirements and the computation mode, application accelerators display various potentials and limitations. In this work, we analyze the FPGA and GPU based accelerators with respect to various performance metrics.

(a) Viability of Floating Point Computations on Reconfigurable Architectures

Floating point computations have long been thought to be out of reach of the FPGAs. With the increasing chip sizes and capabilities, this has changed in the last few years. The current work shows that current generation FPGAs can achieve one to two orders of magnitude speedup on computations that are rich in floating point operations. Moreover, we also show how these speedups compare to those achieved on a high end graphics processor and contrast the relative merits of these two architectures.

(b) Analysis of the Achievable Floating Point Performance

Modern GPUs and multicore processors possess very high peak floating point capabilities. The utilizations on production applications, however, are often much smaller. Due to this, the peak performance is not always the best metric of the achievable performance. In the current work, we analyze the achieved floating point performance on these architectures, including that on a modern FPGA, for two different production molecular modeling applications. The result presents an interesting perspective on the relative merit of these architectures in terms of the performance on real applications.

(c) Analysis of the Computation Costs

We analyze the relative costs of performing the computations using accelerator based solutions as compared to the general purpose processors. We present an analysis in terms of the cost of ownership as well as the cost of power.

1.6.3 Broader Application and Implications

Even though the solutions proposed in this work are with respect to the molecular modeling applications, they can be employed in a variety of other applications.

Moreover, the research findings can be applied towards other applications with similar computations.

(a) General Computation Models and Data Structures on FPGAs and GPUs

The FPGA structures proposed in this work as well as the data structures used in the GPU design are generic and can be easily applied to other applications having similar computations.

Data Structure for efficient distribution of n-body problems: N-body problems, especially in the field of molecular modeling, are often arranged in terms of neighbor-lists. Mapping the neighbor-lists to a parallel architecture can result in large communication overhead as well as uneven work distribution. We propose a data structure that allows efficient and uniform distribution of work, leading to improved performance. Though we have used this data structure on a graphics processor, it is very generic and can be applied to other parallel architectures.

Hardware-friendly structure for multiple correlations: PIPER rigid docking performs multiple correlations and computes their weighted sum. Straightforward extension of the single-correlation systolic array to multiple correlations requires large computational resources. In this work, we present a new hardware-friendly systolic array for performing and combining multiple correlations in parallel. The proposed structure results in significant reduction in the hardware resource requirements compared to the other approaches. Moreover, the structure is very generic and can be applied to other applications requiring multiple parallel 3D correlations.

(b) Limits of the Direct Correlation

Correlation using the FFT is known to reduce the complexity from $O(N^6)$ to $O(N^3 logN)$. We find, however, that for certain small problem sizes and lower bit-precisions, parallel implementation of direct correlation can outperform FFT correlation. As the problem size increases, however, the increased complexity of direct correlation starts to overcome the benefits of the reduced-precision, with the FFT correlation ultimately surpassing direct correlation. This crossover point happens at different problem sizes on different architectures. The current research finds this crossover point for three different architectures: FPGAs, GPUs and multicore. In addition to molecular docking, this result can be of interest to other applications that perform 3D correlation as a core computation.

Divergence in accelerated rigid docking solutions: Since the primary computation in rigid-docking is 3D correlation and since different computation modes perform better on different problem sizes, there exists a divergence in the accelerated docking algorithms. Certain computation modes and hardware architectures are good for small-molecule docking whereas others are more cost-effective for protein-protein docking. In the current work, we find this divergence and identify the methods most suitable for the acceleration of different classes of rigid docking.

1.7 Organization of the Rest of the Thesis

The rest of the thesis presents different aspects of the design, integration and validation of the FPGA and GPU based accelerators for rigid docking and binding site mapping. We start with describing the applications and the hardware platforms in detail, followed by different possible mapping of these applications on the two architectures. Based on the computation mode chosen and the way the computation is structured, there are various possible ways of designing the accelerators, each presenting various design trade offs and performance limitations. We discuss these in detail and present the most optimal solutions on the two architectures. We then present our results in terms of the performance improvements obtained on these applications as well as an analysis of the precision errors.

The above topics are organized in eight different chapters. Chapter 2 gives an overview of performing high performance computing using FPGAs and GPUs. Here, we describe the two architectures, their programming models, various systems based on these architectures and the merits and challenges in designing accelerators using these architectures.

Chapter 3 discusses the molecular docking and binding site mapping algorithms in detail. It presents the different docking and mapping algorithms and their uses in the process of molecular modeling. Here, we analyze the computations in different docking algorithms and their amenability to acceleration. It also presents details about the computations involved in the two production docking and mapping applications being targeted for acceleration by the current work.

In chapter 4, we present our FPGA and GPU algorithms for the acceleration of rigid molecule docking. This chapter provides an in-depth discussion of the different design choices in designing the FPGA and GPU based accelerators for docking. We discuss the architectural trade offs of different designs on the FPGA and present our hardware-friendly systolic structure for performing multiple parallel correlations. We also discuss the different computation modes suitable on the two architectures and present a discussion about the divergence in the algorithms for accelerated docking.

Chapter 5 presents the acceleration of Binding site mapping using FPGAs and GPUs. In this chapter, we present our FPGA pipelines for the electrostatics energy computations and discuss various issues in the integration of the pipelines. We also present various different ways of mapping the energy minimization task on the GPU, followed by two different data structures that enable efficient distribution of work on the GPU.

In chapter 6, we present the details about the integration of the FPGA and GPU accelerated docking and minimization routines into production software. We present the overall system architectures and the control flow of the accelerator systems and discuss the integration issues specific to each application.

Chapter 7 presents our results from the acceleration of the two applications. The results are presented in terms of the performance improvements obtained on these applications as well as the precision errors in the integrated system. We also provide our results relating to the floating point utilizations on three different architectures.

Chapter 8 concludes this thesis and provides some future directions for this work.

Chapter 2

High Performance Computing Using FPGAs and GPUs

2.1 Overview

General purpose digital computers have been mainly based on the the von Neumann architecture where the instructions of a stored program are sequentially executed on a central processing unit. Traditionally, high performance computing has applied large clusters of such serial processors to expedite the execution of computationally demanding applications. In recent years, however, the use of accelerators in high performance computing has received widespread attention. Examples of some serious efforts in this direction include a special purpose machine for molecular dynamics simulations [SDDK07], the development of a high-end FPGA-based supercomputer [fHPRC10] and large supercomputers based on GPUs [Top10] and Cell BE [Lab10], to name a few.

Over the years, numerous different hardware architectures have been applied for application acceleration. Some of the more recent examples include FPGAs, GPUs, Cell Broadband Engine and of-course application-specific ICs. In this chapter, we talk about two such architectures: FPGAs and GPUs. We discuss in detail the various characteristics of these platforms, including their hardware architecture and the salient features as well as the potential of their application to high performance computing and the challenges involved therein. We also discuss the programming model for such systems and the hardware and software support from different vendors.

2.2 Accelerator Based Computing

Accelerator-based computing aims at moving the computationally demanding tasks of the application to a special purpose processor optimized to perform certain computations very efficiently. The main purpose of an accelerator is to speed-up the execution of certain complex computations that can benefit from parallel or other optimized implementations compared to running on a serial CPU. The use of accelerators to improve the overall system performance, however, is a non-trivial task. Designing such systems involves various challenges at the different steps of the design process, starting from the extraction of the computations to be accelerated, through the development of the acceleration coprocessor and finally in the integration of the accelerated subsystem into the rest of the system.

In order to be accelerated effectively, the computations must posses some specific characteristics. These include (a) some inherent parallelism or the ability to be transformed as such to benefit from parallel implementations, (b) high computation to communication ratio, (c) the use of bit-level or low-precision arithmetic, (d) simple computation kernel, and (e) uniform, non-divergent computations. The last three properties ensure small computational resource requirements, allowing for a large number of replications of the computation core and hence higher parallelism. Computations lacking one or more of these characteristics are often hard to accelerate. We faced some of these challenges in the current study and addressed them by restructuring the computations and the underlying data structures. We elaborate on these in chapters 4 and 5. Here we discuss the general set-up for accelerator-based computing and the typical usage scenario.

Figure 2·1 shows an accelerator system with two general purpose processors and

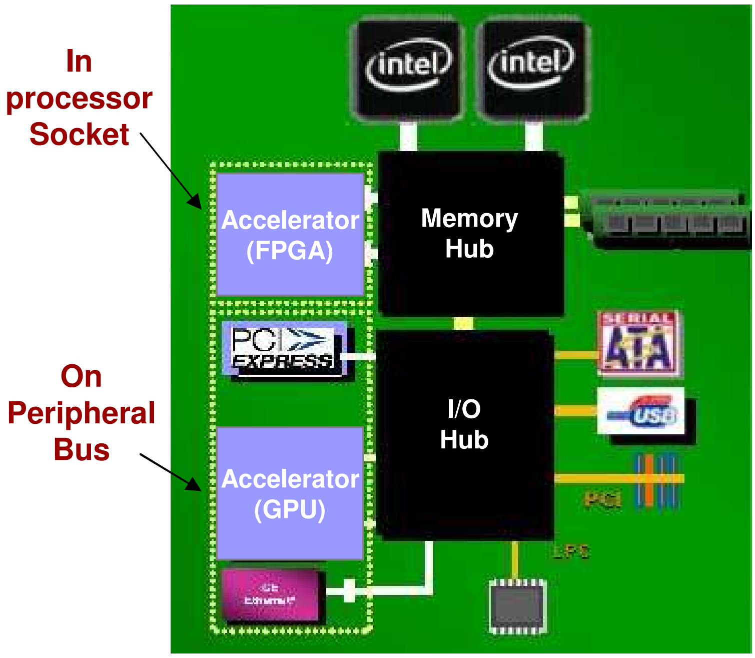


Figure 2·1: Typical hardware set-up for an accelerator-based system (based on [Bha07])

two accelerators: an FPGA board plugged into one of the processor sockets and a GPU board plugged into the peripheral bus. In general, an accelerator system may contain one or more FPGAs, one or more GPUs or a combination of both. GPU boards typically appear on the PCI or PCIe bus whereas the FPGA systems can be socket based or bus-based.

In the accelerator-based systems, the CPU and the accelerator board sit close to each other, inside the same box. This enables low-latency communication. As shown in the figure, the FPGA accelerator in the processor socket has direct access to the processors' memory through the memory hub. This provides high-speed communication between the host and the accelerator, with bandwidths of up to a few GB/s. The GPU system shown, however, accesses the main memory though the I/O hub. The multi-lane high speed PCIe bus enables fast communication between the processor and the GPU, with bandwidths similar to the socket-based systems.

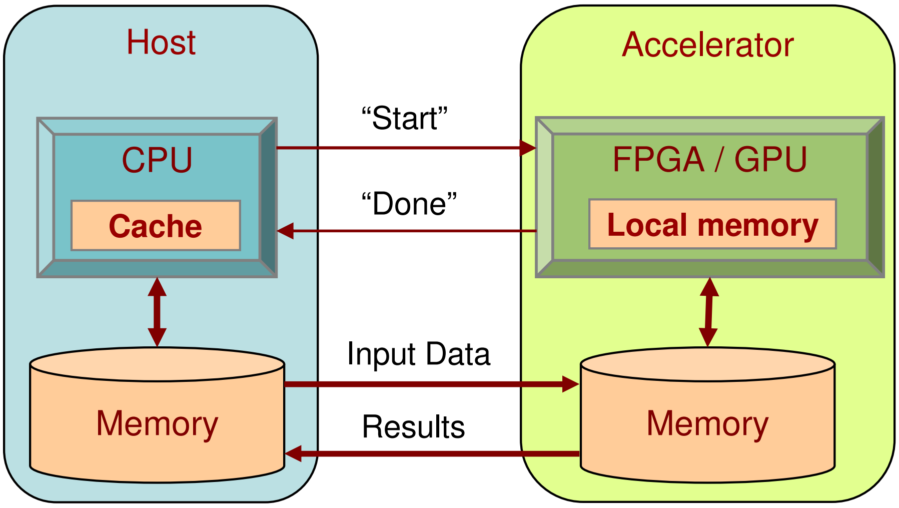


Figure 2·2: Typical usage scenario for an accelerator-based system

The typical scenario for computing with an accelerator-based system is shown in Figure 2·2. As shown, both the host and the accelerator have their own system memory. In addition, both contain a relatively small, low-latency local memory (or cache) to hold the working set. The accelerator acts as a coprocessor to the main processor, which offloads the computationally demanding tasks to the accelerator. The host explicitly transfers the required data and parameters from its memory to the accelerator memory and notifies the accelerator to start the computation. The host processor can either continue with the rest of the computations or wait for the accelerator to finish the computations. Once finished, the accelerator notifies the host processor, which then copies the results from the accelerator memory back to its main memory. The host-board transfer is typically done as DMA transfers whereas control signals are passed by writing to the hardware registers. Accelerator board vendors provide the software APIs and the hardware IP cores to facilitate these transfers.

2.3 Background

Field Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs) have gained widespread attention as application accelerators, both for everyday desktop computing as well as for high performance computing applications. The ability to exploit parallelism at various levels, along with very low power consumption, makes FPGAs particularly attractive for accelerating a wide variety of applications. GPUs, on the other hand, provide immense raw floating point compute capabilities for a fraction of the cost and power of a large cluster, making them suitable for applications requiring high floating point computations.

Field programmable gate arrays are commodity integrated circuits that can be "programmed" or reconfigured in the field. In other words, they contain logic and interconnects that can be programmed to generate different circuits. This is in contrast to other integrated circuits where the logic is fixed during fabrication and cannot be altered. The term reconfigurable computing refers to computing using such devices that can be reconfigured into different circuits. The idea of reconfigurable computing was first introduced in 1960 [Est60]. High performance reconfigurable computing (HPRC), however, has gained popularity only in the recent past. This is due to two main reasons. First, as the microprocessors are starting to hit the power wall, with frequencies stagnated around 4 GHz, alternative high performance computing (HPC) architectures are being explored. Second, recent advancements in FPGA chips have made them capable of delivering significant computational power relative to both ASICs and general purpose microprocessors. Current generation high-end FPGAs contain hundreds of embedded multipliers, hundreds of embedded multi-ported memories and high-speed I/O links, enabling high memory bandwidth and large computational throughput. Moreover, unlike few years ago when FPGAs could achieve clock frequencies only in few tens of mega-hertzs, modern FPGAs can achieve frequencies up to 500MHz. These evolutions in the FPGA technology have led to FPGAs being seen as a viable alternative for high performance computing [GRT$^+$06].

In addition to FPGAs, GPUs have also seen increasing interest from the HPC community. A graphics processing unit is a highly parallel, many-core processor specializing in computations related to graphics rendering. Most, if not all, computers now-a-days contain a graphics processing unit used for offloading the compute intensive task of pixel rendering. Due to the immense computational capabilities offered by the GPUs, they have long attracted attention from scientists and researchers who have insatiable need for computational power. Over the years, researchers and scientists have tried to map general purpose computing onto graphics processing units. This led to the coining of the term general purpose computing using graphics processing units (GPGPU). Until recently, graphics processor pipelines were very inflexible, performing fixed set of operations typically used for pixel computations. Due to this, using graphics processors for general purpose computing was a tedious task as it required laying out the computation in terms of pixel computations and using the graphics API. With the advent of more flexible graphics pipelines and high level languages such as NVIDIA CUDA [NVI08b] and AMD-ATI Stream [AA10] for programming the graphics processors, GPUs have gained widespread attention for general purpose computing. Their increased computational capabilities, relative ease of programming, easy integration, low cost and comparatively low power consumption have made them a very attractive platform for HPC applications.

2.4 High Performance Reconfigurable Computing

2.4.1 FPGA Overview

Field programmable gate arrays are a class of reconfigurable integrated circuits that contain logic and interconnects that can be programmed in the field to generate different circuits. High performance computing using FPGAs is, thus, often referred to as high performance reconfigurable computing (HPRC).

FPGAs differ from other members of the programmable circuits family such as Programmable Array Logic (PAL), Programmable Logic Array (PLA) and Programmable Logic Device (PLD) in two ways: first, FPGAs are more flexible in the types of functions that can be implemented and second, in addition to the programmable logic and interconnects, FPGAs contain a wide variety of embedded components such as embedded memory blocks, hard-wired multipliers and embedded processor cores. These features make the FPGAs suitable for a wide variety of applications.

Traditionally, FPGAs have seen application in the following areas:

- **Telecommunication and Networking:** One of the most common and widespread application of FPGAs is in the field of communications and networking. Network routers often utilize FPGAs to implement the routing tables and the network interface protocols.

- **Signal and Image Processing:** Due to their relatively simple computation kernels, reduced precision requirements and streaming-type computation mode, signal and image processing applications enjoy significant performance benefit from FPGAs implementation.

- **Glue Logic:** FPGAs see widespread application as “glue-logic” on printed circuit boards, to connect different chips on the board. Application of FPGAs for glue-logic allows for the inter-chip communication patterns to be modifiable even after the board has been manufactured.

- **Rapid Prototyping and Emulation of Integrated Circuits:** FPGAs have become the platform of choice for emulating the logic of integrated circuits before the actual chip is fabricated. The most common example is the emulation of microprocessor chips. FPGA emulation offers many benefits over software emulation approach. These include (i) faster emulation speed, (ii) the ability to perform tests on the real hardware, (iii) more accurate modeling of faults and interrupts in real-time, on real hardware, (iv) the ability to run full-blown operating system and real applications using the native instruction set.
- **Alternative to ASICs:** FPGAs are also widely applied in applications where the volume is not large enough to amortize the cost of an application specific IC. With their high flexibility and low cost, FPGAs provide a good cost-effective alternative.

2.4.2 FPGA Architecture

Original FPGAs are integrated circuit chips with millions of configurable logic gates in a sea of configurable interconnects. Figure 2·3 shows an abstract view of how the reconfigurable logic is implemented in an FPGA.

The main logic element in an FPGA is a look-up table (LUT). A look-up table is a combination of a 2^n bit configuration memory and a $2^n : 1$ multiplexor, where n is the number of inputs to the look-up table. The width of the configuration memory is 1-bit. FPGAs implement logic using typically 4 to 6 input look-up tables (LUTs). A 4 input LUT can implement a logic function of up to 4 variables. The logic is "programmed" by storing the value of the function for all the possible input combinations in the configuration memory. The logic functionality is achieved simply by selecting the appropriate output from the configuration memory using the multiplexor. The output can optionally be passed through the flip-flop at the output of the multiplexor, to build sequential logic (as shown inside the bubble in Figure 2·3).

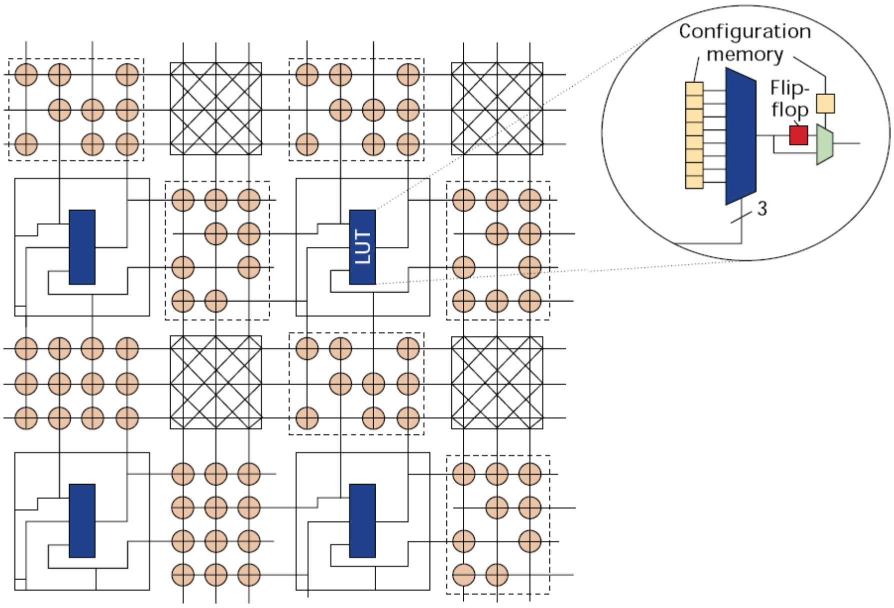


Figure 2·3: Configurable logic and interconnects inside an FPGA [DeH00]

As shown in Figure 2·3, the configurable interconnect in an FPGA is implemented using parallel rows and columns of interconnect wires, with a programmable switch at every junction where a vertical interconnect meets a horizontal interconnect.

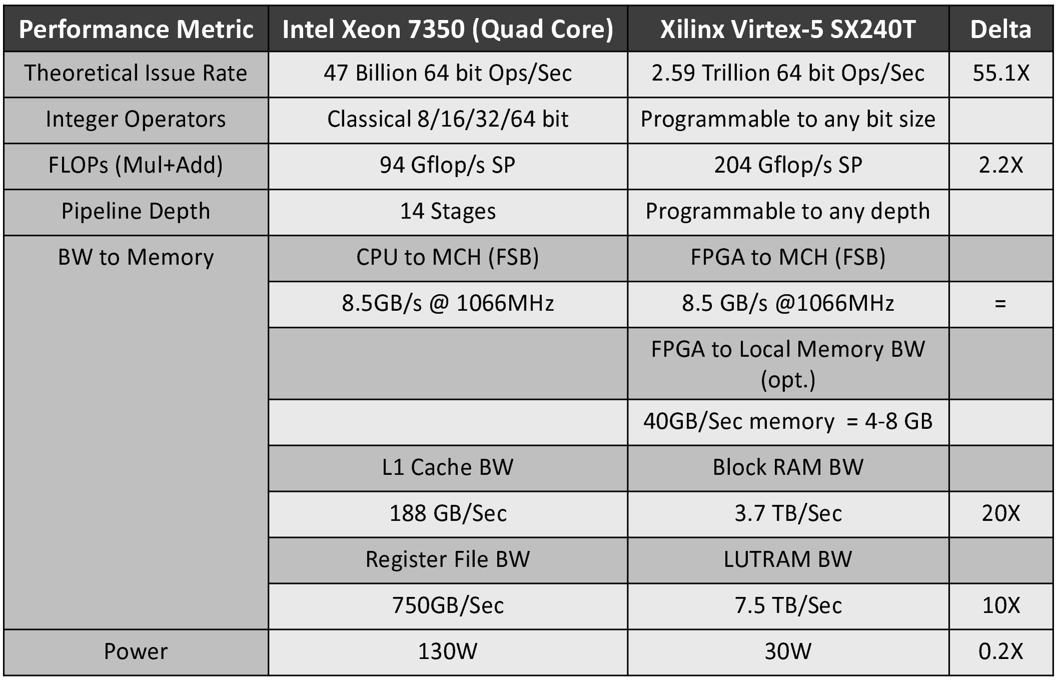
In addition to the reconfigurable logic and the interconnects, modern FPGAs also contain hundreds of hardwired DSP elements (such as multipliers) and independently addressable multi-ported memories, along with one or two embedded processor cores.

Due to their flexible architecture, FPGA fabrics are less dense and slower than ASICs. Their flexibility, however, often more than makes up for these drawbacks. The main features of the current generation high-end FPGAs can be summarized as follows:

- Millions of gate equivalents that can be used to implement variety of logic functions, distributed RAMs or shift registers.
- Tens of thousands of register elements
- Hundreds of hardwired DSP elements such as multipliers and accumulators
- Hundreds of independently addressable multi-ported memories
- Highly flexible, programmable interconnect between logic modules
- Embedded on-chip processors
- (Re)programmable in milliseconds
- Gigabit interfaces to off-chip devices
- Hundreds of I/O pins that allow bandwidths close to 50 Gbps to the off-chip memories
- Very low power dissipation, in the range 10 to 30 watts

Figure 2·4 compares a quad-core Intel Xeon 7350 processor and a Xilinx Virtex-5 FPGA with respect to some of these capabilities. As shown, FPGAs provide 55× higher computational capabilities and 10× higher on-chip bandwidth with only 20% of the power consumption.

Due to these immense independent resources, FPGAs can provide very high sustained computational throughput, resulting in orders of magnitude speedups over conventional microprocessors. Obtaining good performance from FPGAs, however, requires careful restructuring of the algorithms and efficient mapping to the underlying hardware resources. This is discussed in detail in Section 2.4.5.



Performance Metric	Intel Xeon 7350 (Quad Core)	Xilinx Virtex-5 SX240T	Delta
Theoretical Issue Rate	47 Billion 64 bit Ops/Sec	2.59 Trillion 64 bit Ops/Sec	55.1X
Integer Operators	Classical 8/16/32/64 bit	Programmable to any bit size	
FLOPs (Mul+Add)	94 Gflop/s SP	204 Gflop/s SP	2.2X
Pipeline Depth	14 Stages	Programmable to any depth	
BW to Memory	CPU to MCH (FSB)	FPGA to MCH (FSB)	
	8.5GB/s @ 1066MHz	8.5 GB/s @1066MHz	=
		FPGA to Local Memory BW (opt.)	
		40GB/Sec memory = 4-8 GB	
	L1 Cache BW	Block RAM BW	
	188 GB/Sec	3.7 TB/Sec	20X
	Register File BW	LUTRAM BW	
	750GB/Sec	7.5 TB/Sec	10X
Power	130W	30W	0.2X

Figure 2·4: Comparison of performance capabilities of an Intel quad-core processor and a Xilinx Virtex-5 FPGA (based on [Bol10] with some updates)

2.4.3 Sources of FPGA Performance

Over the years, FPGAs have been shown to achieve multi-fold speedups on a variety of engineering and scientific applications. The high performance achieved by the FPGAs can be attributed to the following main reasons:

- **Parallelism at different levels:** FPGAs can exploit parallelism at different levels; examples include bit-level computations, deep pipelining and replication of processing elements (PEs).

- **High payload to overhead ratio:** Unlike conventional microprocessors, most of the cycles on an FPGA are spent in executing payload instructions, with little to no control overhead. This is because the control is built into the logic being implemented on the FPGA.

- **High utilizations:** FPGAs can achieve high utilization of the available resources by replicating the processing elements across the entire chip. This results in increased parallelism and high throughput.

- **Specialized memory interface:** Unlike microprocessors where the interface to the memory is fixed, application-specific memory interfaces can be designed on the FPGA. Hundreds of independently addressable on-chip block-RAMs (BRAMs) can be used to provide fast and parallel single-cycle access to the required data.

- **Free operations:** Certain arithmetic and logic operations, when performed in hardware, translate into simple bit manipulations. Examples include bit-shifts, multiplication / division by powers of 2, bit-selection etc. While these operations still require at-least one cycle on conventional microprocessors, they come free on the FPGAs.

While most or all of the benefits mentioned above are also available in application specific integrated circuits (ASICs), the cost associated with the design and fabrication of an ASIC often cannot be justified for low-volume applications.

2.4.4 FPGA Computing Models

Developing efficient FPGA applications requires identifying the appropriate computing model that can map the application well on the hardware. The computing models for FPGAs can be considered analogous to the programming models for software in the sense that they both provide an abstraction of the machine on which the computation is performed. FPGA computing enables models with highly flexible fine-grained parallelism and associative operations such as broadcast and collective response.

In any computing model, in order to achieve maximal performance, it is essential to consider the underlying memory model. FPGA based systems typically provide a hierarchical memory model with memories at the following levels:

- On-chip registers and lookup tables, with memory bandwidths up to 1.5 TB/sec.
- On-chip block RAMs (BRAMs); high-end FPGAs have up to several megabytes of embedded BRAMs, with bandwidths up to 160 GB/sec.
- On-board SRAMs and DRAMs; sizes up to a few giga bytes can be supported, with typical bandwidths of 200 MB/sec to a few GB/sec.

A good FPGA computing model allows for creation of mappings that make efficient and maximal use of memories at one or more of these levels. The actual performance, of course, depends on the ability of the application to leverage this memory hierarchy and bandwidth. The current work heavily utilizes FPGA BRAMs for fast data access. A part of this work also uses the on-board memory for storage of different working sets and partial results.

Another critical factor for a good FPGA model is that code size translates into FPGA resource requirements. Best performance is achieved by having high resource utilizations, usually through large amounts of fine-grained parallelism. Performance degrades, however, if the computation does not entirely fit in the available resources and swapping and reconfiguration is required. Conditional computations also lead to poor performance since each branch of the conditional results in real hardware, whether or not the branch is taken. This results in low utilizations and hence lower performance. Careful restructuring and separation of computation, along with reconfiguration, however, can alleviate this situation.

In general, concepts such as “high utilization” and “deep pipelines” are certainly critical to efficient FPGA based design. These are, however, far removed from the application formulation at the start of most design processes. In an article in IEEE Computing in Science and Engineering [HGV$^+$08], we outlined several computing models that are useful during the initial design phase and map well on FPGAs. These are discussed briefly below.

- **Streaming:** Stream-based computing, as the name suggests, is characterized by streams of data flowing through the computation units. For problems with regular computation pattern, streaming can result in high throughput with very little control logic. Examples of streaming based HPRC applications include signal and image processing, string matching applications, sequence alignment and systolic correlation.

  Stream-based computations are a natural fit to FPGAs due to the support for multiple parallel streams, the flexibility to variously connect different streaming structures and high IO bandwidth for feeding those structures. Due to this, several high-level FPGA languages such as Streams-C [FGL01], A Stream Compiler (ASC) [Men06], SA-C [BHD$^+$02] and SCORE [DMC$^+$06] provide explicit support for streaming.

  All the FPGA designs presented in the current work are based on the streaming model, with data streaming through very deep pipelines, resulting in high throughput.

- **Associative computing:** Associative computing is characterized by operations such as broadcast, parallel tag checking, tag-dependent conditional computing, collective response and reductions. Associative computing benefits from FPGA based solutions due to the inherent support for parallel operations, broadcast and reduction, which can often be performed on FPGAs in a single cycle.

- **Standard hardware structures:** This model pertains to utilizing the appropriate standard hardware structure for the computation at hand. Examples of such structures include FIFOs, priority queues and systolic arrays. In signal and image processing applications, 1D systolic arrays are heavily utilized for computing correlations and convolutions. The current work uses an extended and modified version of such a systolic array to compute multiple, 3D correlations in parallel.

2.4.5 FPGAs for High Performance Computing

Field programmable gate arrays are widely considered as accelerators for compute-intensive applications. Traditionally, they have seen applications in the field of signal and image processing and network routers. In the last few years, however, FPGAs have seen widespread acceptance in the field of high performance computing for scientific, financial and military applications. This is mainly due to the following reasons: conventional microprocessors hitting the power wall, FPGAs growing to a size large enough to handle complex computations, continued evolution of FPGA architectures with embedded components and improved operating frequencies of modern FPGAs.

Power concerns in conventional microprocessors have played a major role, either directly or indirectly, in the start of paradigm shift in HPC from microprocessor to FPGAs. Though most HPC systems are still based on microprocessors, FPGA based alternatives are being actively explored. Until few years ago, microprocessors mainly depended on Moore's law for performance improvements, with ever increasing clock frequencies and transistor counts. In recent years, however, power density has hit a wall and frequency scaling has leveled-off. This has led the industry to resort to parallelism for performance improvements, i.e. to have multiple lower-frequency processor cores on the die instead of a single high-frequency core. Processors with

4 and 8 cores are already available and those with dozens of cores might soon be available. Using multicore processors (or manycore for more than 8 to 16 cores) for HPC, however, might not be suitable for fine-grained parallelism due to the large communication and synchronization overheads.

Another low power, high efficiency solution for HPC might be to use application specific ICs as computation engines. An example of such a system is the Anton supercomputer for Molecular Dynamics simulations [SDDK07]. It is a cluster of 512 ASICs specially designed for MD computations which offer very high fine-grained parallelism. For computations other than MD, however, the Anton supercomputer either cannot be applied or results in very poor performance. Clearly, the drawbacks of ASIC based systems are inflexibility and high development cost.

FPGAs provide a good trade off between the two extremes of microprocessors and ASICs. They are special purpose in the sense that they can be programmed to implement a design best suited to the computation at hand, providing fine-grained parallelism similar to ASICs. On the other hand, they are general purpose in the sense that they can be reconfigured to provide different logic functionalities. Even though FPGAs run at an order of magnitude slower clock rates than the microprocessors, their flexibility and fine-grained parallelism more than makes up for it.

The reasons for the high performance capabilities delivered by the FPGAs as outlined in section 2.4.3 can essentially be summarized into two main factors: (i) immense parallelism achievable when the entire FPGA fabric is utilized for replicating the computation engines, and (ii) the capability of the FPGA to execute payload instructions in each and every cycle. Both these factors result in large throughput and hence high performance.

Even though FPGAs offer very high potential performance for HPC applications, achieving good performance is not trivial. Obtaining high performance on an

HPC/FPGA application requires careful hardware-aware implementation and often algorithm restructuring in order to map it well on the underlying hardware.

In our article in the IEEE Computer Magazine's special issue on Reconfigurable Computing [HVG$^+$07], we outlined 12 guidelines that apply to almost any good HPC/FPGA design and can be followed to obtain good performance from FPGAs on high performance computing applications. These can be broadly classified into four categories:

Application Restructuring

- **Use an algorithm optimal for FPGAs:** Often, the algorithm most optimal for software implementation is not suitable for FPGA implementation. An example of this is direct correlation on FPGA versus FFT correlation in software. This restructuring is the central idea behind part of the current work.

- **Use a computing mode appropriate for FPGAs:** Similar to the algorithms, the programming models that are staples of serial computing often result in poor performance on FPGAs. For example, random access and pointer arithmetic are not very suitable for hardware implementation. Streaming, on the other hand, is a natural programming model for FPGAs. The current work benefits hugely from stream-based computing.

- **Use appropriate FPGA structures:** Digital logic contains structures and operations analogous to commonly used data structures in software. It is essential to use these hardware structures as opposed to translating the software structures into hardware.

- **Living with Amdahl's law:** Amdahl's law states that the total speedup on an application is limited due to the part that is left unaccelerated. In other words, obtaining high performance requires accelerating not just the most computationally intensive part but also the rest of the code.

Design and Implementation

- **Hide latency of independent functions:** Hiding communication latency by overlapping with computations is essential to obtaining good performance. Further, computations must be performed in parallel to the extent allowed by the algorithm and the available resources.

- **Use rate-matching to remove bottlenecks:** Hardware designs often contain some computation stages feeding data into other stages. These stages, depending upon their complexities, can take varying number of cycles to generate their output. To constantly feed the faster units with input data, multiple instance of the slower units must be instantiated.

- **Take advantage of FPGA-specific hardware:** FPGAs contain hundreds of special purpose hardware units such as block RAMs and multipliers. These must be efficiently utilized to get good performance. The current work heavily utilizes these hardware components for fast data access and efficient computation.

Arithmetic Operations

- **Use appropriate arithmetic precision:** Many HPC applications require only a few bits of precision. Unlike conventional microprocessors, FPGAs can leverage this reduced precision requirements and implement “narrow” datapaths, thus allowing for multiple replications and better performance.

- **Use appropriate arithmetic mode:** Certain arithmetic modes such as floating point require large computational resources and, if possible, should be avoided and replaced with well-tuned libraries and alternative arithmetic modes such as block floating point.

- **Minimize the use of high-cost arithmetic operations:** The relative costs of arithmetic functions are different on FPGAs than on microprocessors and restructuring the computations to account for this can substantially increase the performance.

System and Integration Issues

- **Create families of applications, not point solutions:** HPC applications are often complex and highly parameterized, with variations both at algorithmic and data format level. Developing solutions that can be applied across these variations amortizes development cost over a large number of uses as well as enables easy reuse for future variations. Developing efficient, parameterized hardware solutions, however, is a challenging task; the current work addresses this with a software-script that generates hardware modules (that change across different members of the application-family) as per the input parameters. These modules, along with the control modules that remain unchanged for a given family, provide the solution for a specific member of the application family.

- **Scale application for maximal use of FPGA hardware:** As a significant portion of the FPGA performance is obtained from parallelism, it is essential to instantiate as many processing elements as allowed by the available hardware resources. With ever increasing FPGA sizes, automated sizing to enable scaling across newer generation FPGAs is very important.

2.4.6 FPGA Systems for High Performance Computing

In order for FPGAs to be used for high performance computing, they need to be integrated with other system-level components such as memory modules, peripherals and host communication interface. Commercial off-the-shelf (COTS) FPGA systems are available from various vendors, differing in the type and the number of FPGAs present on the system, the amount of memory, and the board-host interface. Traditionally, FPGA systems had been based on boards that connect to peripheral buses such as PCI. Since these systems interface through the peripheral bus, the host-FPGA communication is relatively slow and can be a potential bottleneck for high performance computing. In recent years, with the rising demand of reconfigurable hardware for HPC applications, several vendors have released FPGA systems that sit on high speed bus or are more tightly coupled to the main processor and offer very high memory and host-board communication bandwidths. One class of solutions are based on high speed PCI express buses. These include boards from vendors such as Annapolis Micro Systems [Ann10], Nallatech [Nal10] and Gidel [Gid10], to name a few. Another class includes systems where the FPGA is integrated into the communication fabric of an SMP cluster. Examples include systems from SGI [SGI10] and Cray [Cra10a]. The most recent class of solutions include systems where the FPGA acts as a peer to the processor. In such systems, the FPGA board is plugged into one of the processor sockets of a multi-processor motherboard. Due to the tight coupling with the processor, FPGAs on these systems have very high speed access to the processor memory. Socket-compatible FPGA boards are available from vendors such as XtremeData [XDI10] and DRC [DRC10].

Below, we discuss some of these HPC/FPGA systems in detail.

(i) Annapolis Micro Systems' Peripheral Boards: Annapolis Micro Systems provides custom off-the-shelf PCI and PCIe plug-in boards with Xilinx FPGAs, along with other system level components such as large on-board memories and communication interfaces. These boards plug into the peripheral bus of standard PCs and act as coprocessors to the main processor.

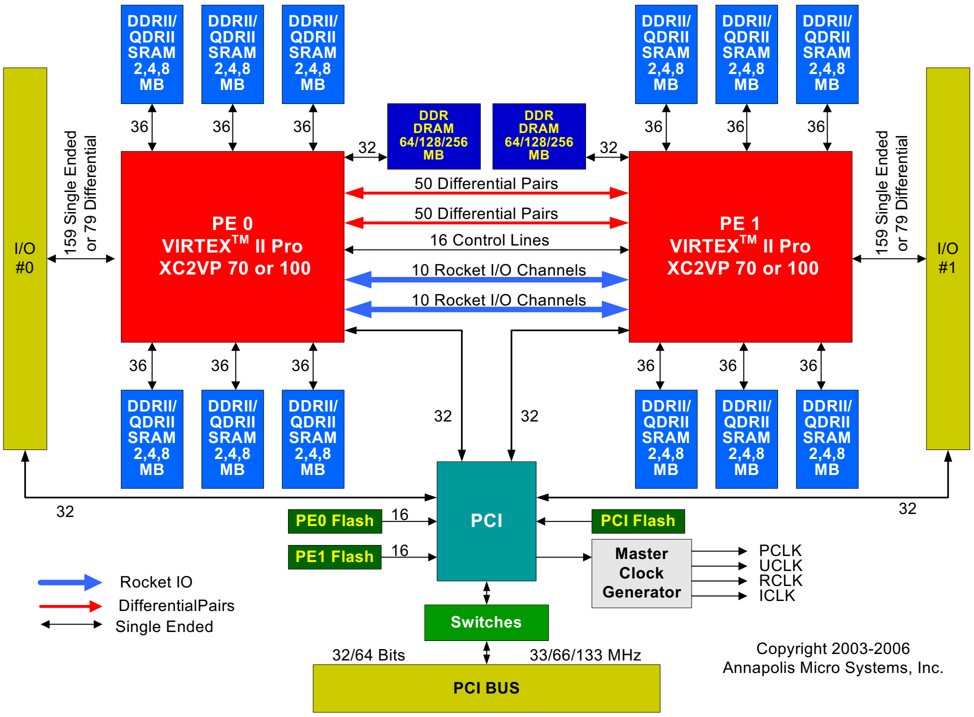


Figure 2·5: Annapolis Micro Systems' Wildstar II Pro PCI board [II03]

An example is the PCI based Wildstar II Pro board. The schematic of the system is shown in Figure 2·5. This board is a few generations old and contains two Xilinx Virtex II Pro FPGAs (VP 70 or VP 100), each connected with up to 48 MB on-board SRAM and up to 128 MB DRAM. The FPGAs communicate to each other through differential pins or RocektIO channels and to the host through the PCI bus. Annapolis Micro Systems also provides a newer generation PCIe based Wildstar 5 board.

(ii) Gidel PROCe III PCIe board: The PCIe plug-in FPGA boards provided by Gidel are based on Altera FPGAs. Figure 2·6 shows the block diagram of the PROCe III [Gid09a] board from Gidel.

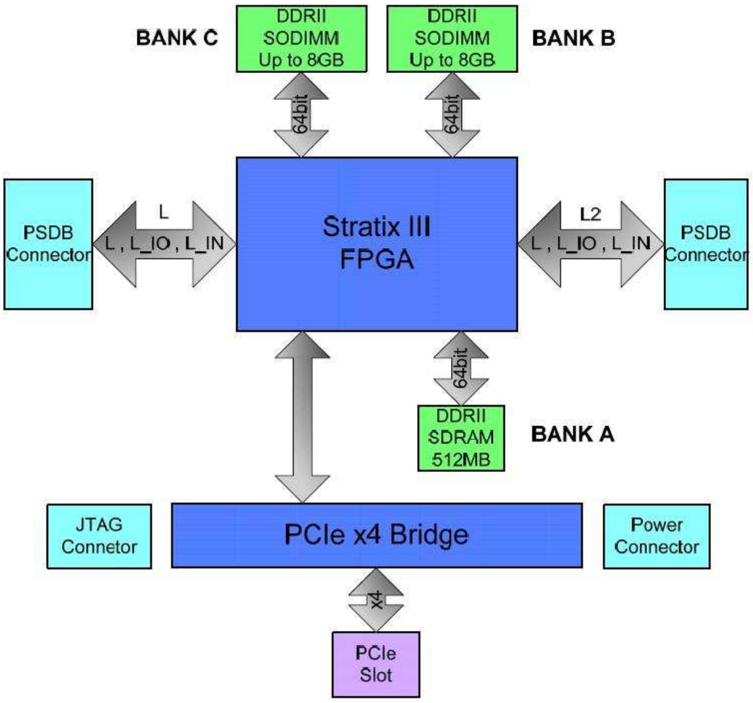


Figure 2·6: Gidel's PROCe III system block diagram [Gid09a]

The PROCe III board contains an Altera Stratix III FPGA and 512 MB of on-board memory. The total system memory can be expanded to up to 8.5 GB using the SODIMM sockets. The board connects to the host via a 4-lane PCIe bus, providing up to 8 DMA (Direct Memory Access) channels, with host-board communication bandwidths up to 550 MB/sec [Gid09a]. In addition, the PROCe III board provides fast interface to the on-board memory, with bandwidths up to 6 GB/sec. This makes it particularly suitable for applications with large working set that cannot fit on on-chip memory.

To design the host-board control and communication interface, Gidel provides a graphical user interface based software tool called the PROCWizard [Gid09b].

PROCWizard can be used for efficiently defining the on-board memory resources required by the application and the host-board and board-FPGA interfaces. This drastically simplifies the process of system integration.

(iii) SGI RASC Platform: Figure 2·7 shows the schematic of SGI's Reconfigurable Application Specific Computing (RASC) system [Sil04] based on the Altix [SGI08] architecture. A RASC blade contains two Xilinx Virtex 4 FPGAs, each with 5 DIMM slots that be used to connect up to 40 MB SRAM or 20 GB SDRAM. The FPGAs are connected to the host and the global shared memory through dual NUMALink buses, each providing a peak bandwidth of 6.4 GB/s. The connection between the FPGAs and the NUMALink is through the Scalable System Port (SSP) enabled via the TIO ASIC [Sil04]

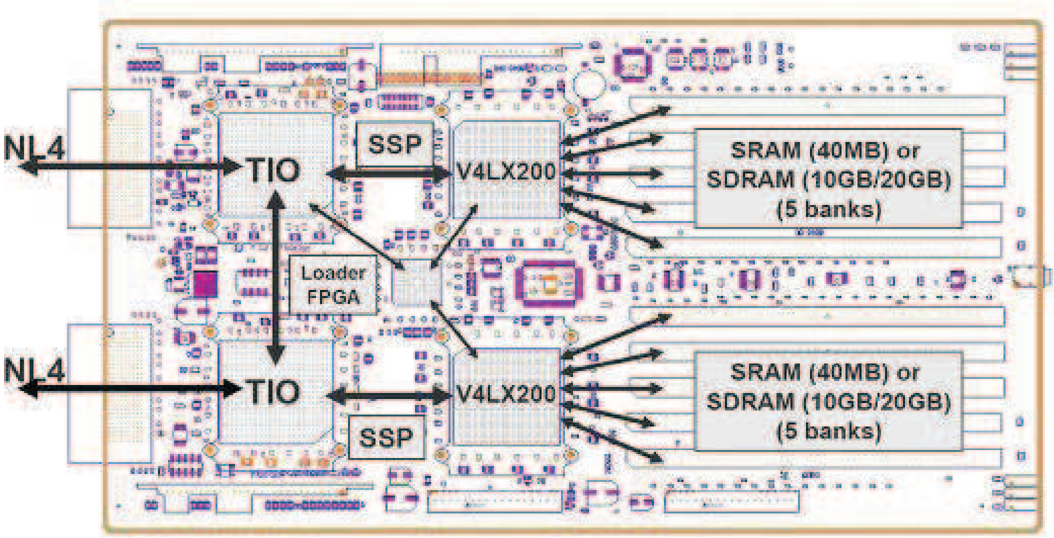

NL4
TIO
SSP
V4LX200
SRAM (40MB) or SDRAM (10GB/20GB) (5 banks)
Loader FPGA
NL4
TIO
SSP
V4LX200
SRAM (40MB) or SDRAM (10GB/20GB) (5 banks)

Figure 2·7: SGI RASC RC 100 blade [FL06]

The software architecture of the SGI RASC platform is shown in Figure 2·8. As shown, RASC provides a layered software architecture, allowing the users to access the FPGA using a high level API, with the OS being responsible for the device set-up and management and the data transfers. FPGA logic can be designed in conventional hardware languages such as VHDL or Verilog or using high level languages such as Handel-C [Agi10] or MitrionC [Mit10].

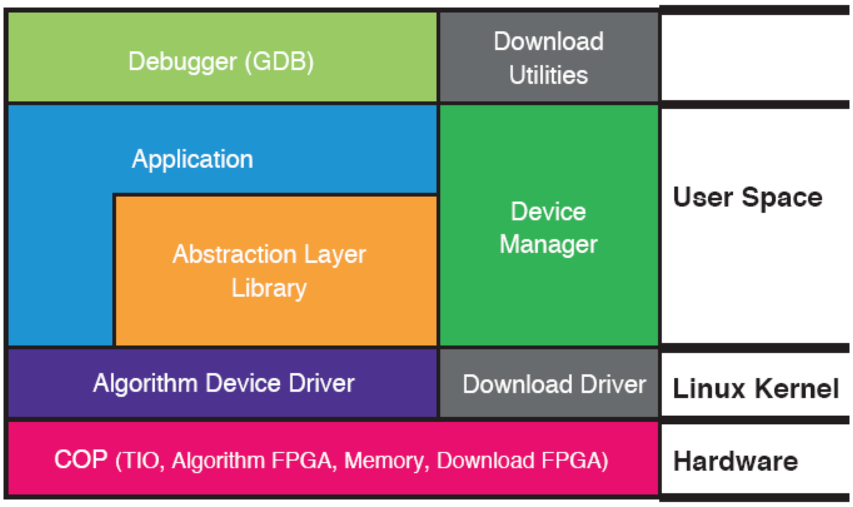


Figure 2·8: SGI RASC software architecture [Sil04]

(iv) Cray Supercomputers with FPGAs: Another example of FPGAs as a part of a message passing system is the Cray XD1 [Cra05]. Here, each chassis contains six nodes, with each node containing one Xilinx Virtex 4 FPGA along with two AMD Opteron microprocessors. The FPGA connects to the host via the HyperTransport bus (3.2 GB/s) and to other nodes via the RapidArray Interconnect System (4 GB/s). The message passing API is extended to execute computations on FPGAs. The newer generation FPGA solutions from Cray include the XT4 and the XT5$_h$ systems. Cray XT5$_h$ supports dual and quad core AMD Opteron processors, along with vector processors and FPGA accelerators [Cra10c]. The FPGA accelerators in both the XT4 and the XT5$_h$ are based on the Cray XR1 reconfigurable processing blade [Cra10b]. It contains two Reconfigurable Processing Units (RPUs) from DRC Computers [DRC10], each with one Xilinx Virtex 4 FPGA and up to 4

GB memory. In an RPU, the FPGA plugs into the socket of an Opteron processor, getting direct access to the main memory.

(v) SRC MAP Station: The HPRC solution provided by SRC Computers is called the Direct Execution Logic processor or the MAP processor [SRC10a]. The block diagram of a SRC Series H MAP processor is shown in Figure 2·9. It is a standalone box with two Altera Stratix II FPGAs for user logic and one for on-board control logic. The board also contains 64 MB of on-board SRAM and up to 2 GB on-board SDRAM, with memory bandwidths of up to 8.4 GB/sec.

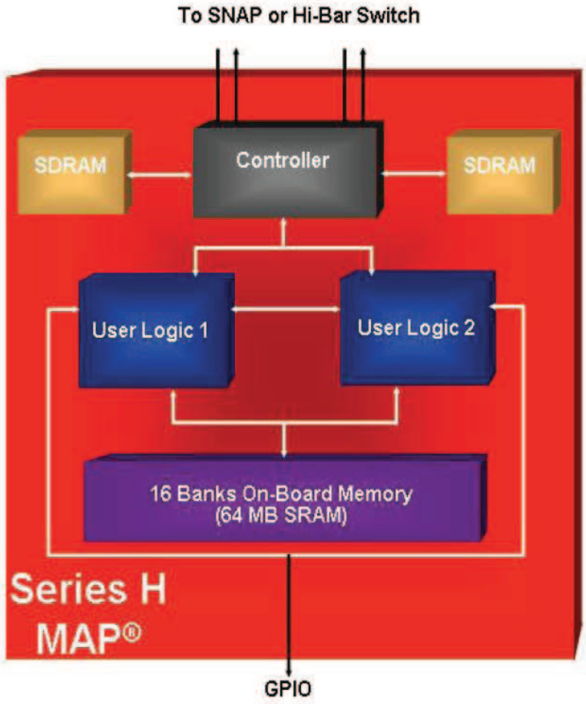


Figure 2·9: SRC Series H MAP processor [SRC10a]

A MAP box plugs into a microprocessor node called the MAP Station. Multiple MAP stations can be connected via Ethernet to build a high-end cluster, with multiple FPGA boxes, microprocessors and common memories connected through the

SRC Hi-Bar Switch bus.

The SRC MAP programming model is based on the SRC Carte Programming Environment, which is an FPGA C and Fortran programming environment [SRC10b]. A MAP compiler separates the C or Fortran code into software and hardware components and generates pipelined hardware for latter. These pipelines then get instantiated on the MAP processor. The MAP compiler also generates the interface code to manage data transfers and handshaking between the software and the hardware components.

(vi) XtremeData XD1000: Recently, some vendors have started providing FPGA boards that can plug into one of the processor slots on a multi-processor motherboard. One such solution is the XD1000 card from XtremeData Inc. [XDI07].

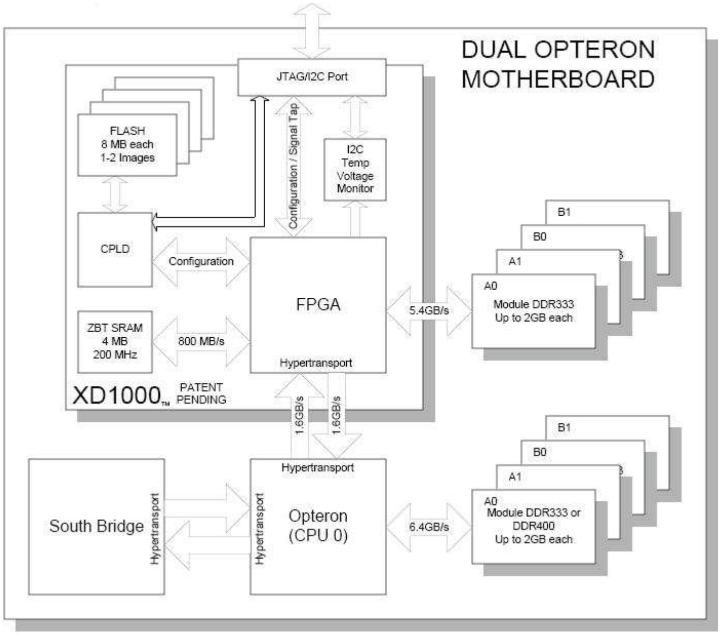


Figure 2·10: XtremeData XD1000 [XDI07]

Figure 2·10 shows the block diagram of the XtremeData XD1000 card plugged into one of the sockets of a dual Opteron motherboard. The FPGA card contains an Altera Stratix II FPGA, with 4 MB SRAM and up to 16 GB DRAM. The FPGA communicates with the Opteron processor through the HyperTransport bus at 3.2 GB/sec. In this way, the FPGA acts as a peer to the microprocessor, achieving high bandwidth communication to the host and the memories.

In terms of the programming environment, XtremeData provides a platform support package (PSP) with Impulse-C [Imp10]. Impulse-C is a high level C-like language for FPGA-based developments. Using Impulse-C, the FPGA design, interface and control logic, and host-board communication can be expressed in a C-like language and the Impulse CoDeveloper, a software-to-hardware compiler, generates the required logic and interface code in standard HDLs. The HDL code can then be synthesized using standard synthesis tools for mapping to the FPGA.

Of the HPC/FPGA systems discussed above, we have experience on the systems from Annapolis Micro Systems, Gidel, SGI and XtremeData. The current work, in particular, involves the use of boards from Annapolis Micro Systems, XtremeData and Gidel.

2.4.7 Software Support for High Performance Reconfigurable Computing

Designing and implementing HPRC solutions involves two main steps – designing the hardware coprocessor and integrating the coprocessor with the rest of the system. As discussed in the previous section, the software support for system integration is largely vendor specific and differs based on the HPRC platform used. The design and implementation of the acceleration core, however, is mostly independent of the hardware vendor, except for the use of the vendor specific IP cores.

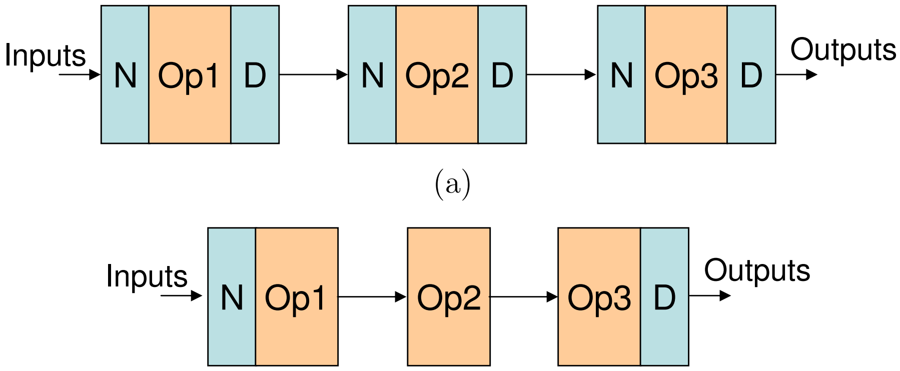
Traditionally, designing FPGA-based hardware has involved the use of hardware description languages such as VHDL and Verilog. In the last decade, however, a variety of high-level languages and “C-to-gates” compilers have been developed that can convert a serial C-like program into HDL code for the FPGA design. Some examples include the ASC [Men06], the Altera Floating Point Compiler (FPC) [Alt08], Handel-C from Agility Design Solutions [Agi10], MitrionC from Mitrionics [Mit10], CatapultC from Mentor Graphics [Men10b] and Impulse-C from Impulse Accelerated Technologies [Imp10]. Though these tools provide faster design solutions, in our experience, their use for the automatic conversion from a serial C-like implementation to an FPGA design results in inefficient hardware. Some of these tools, however, can be effective if used for generating only the computation pipeline and not the whole design. In this work, we utilize the Floating Point Compiler for generating the FPGA pipelines for the electrostatic computations in energy minimization. Below we present a brief discussion about this tool.

Altera Floating Point Compiler

The availability of large amounts of logic resources and hundreds of hardwired DSP components on the chip has enabled the current generation FPGAs to deliver high performance, even in the realm of floating point computations. Furthermore, certain CAD tools can be used for efficient generation of floating point datapaths for FPGA implementation. The Floating Point Compiler [Alt08] is one such tool developed by Altera corporation. It aims at reducing the hardware resource requirements by eliminating certain redundant transformations in a complex floating point computation.

FPGA vendors provide a variety of highly optimized IP cores for common floating point operations. Complex floating point pipelines on FPGAs can be implemented simply by interconnecting these IP blocks. This method, however, is still somewhat inefficient. This is because the floating point IP cores normalize their inputs before operating on them and denormalize the result before sending them to the output.

Connecting a chain of such IP cores leads to some redundant normalization and denormalization operations that could be avoided if the pipeline was hand-customized. This is shown in Figure 2·11 [Dai10]. As shown in Figure 2·11(b), some of the intermediate normalization/denormalization stages of Figure 2·11(a) can be removed without affecting the final result. This is the main idea behind the Altera Floating Point Compiler.



(a)

(b)

Figure 2·11: Floating point datapaths (a) interconnected IP cores, and (b) "fused" datapath using the Altera FPC. [Dai10]

The FPC is a C-to-HDL compiler that generates efficient floating point datapaths for computations described in standard C language. The generated datapath is unidirectional and contains no internal switching or control flow[Alt08]. The main idea of the compiler is the use of the "fused" datapath approach wherein many floating point operations are performed between each set of normalization and denormalization. This results in up to 50% reduction in logic utilizations and 50% reduction in latency compared to using the discrete IEEE754 operators[Alt08].

It is worth mentioning that even though the inputs to and outputs from the FPC are IEEE754 compliant, the formats used for the intermediate computations are different. Moreover, FPC uses different word formats during various floating point operations. This, combined with the non-associativity of the floating point operations, often results in there being a difference in the output from the datapath generated by the FPC and the output from the sequential C code [Alt08]. For certain iterative and highly precision-sensitive computations, this can lead to large overall errors. The FPGA pipelines for the electrostatic computations generated using the FPC display some such errors. Its effects on the final output are discussed in detail in Chapter 7.

2.4.8 Challenges with FPGA-based Design

Although FPGA based computing offers huge potential for HPC applications, it also poses a number of challenges and unsolved problems.

- **Programmability / Dual-expertise:** Efficiently mapping the application to the hardware often requires algorithm restructuring and precision analysis. Such algorithmic tuning requires domain knowledge and expertise. Efficient FPGA implementation, on the other hand, requires logic design expertise and the ability to program in hardware description languages such as VHDL and Verilog. It is often difficult, if not impossible, to find a confluence of these skills in a single person since scientists and application specialists are not trained in hardware design and vice-versa.

- **Jumping the frequency gap:** FPGAs typically operate at frequencies ranging from 50 MHz. to 500 MHz. This is an order of magnitude less than most microprocessors and obtaining speedup over software first requires jumping this frequency gap.

- **System/board level integration:** Integrating the FPGA design with the rest of the software code requires board level integration for data transfers and hardware setup. This often involves using vendor-specific APIs and makes migrating to future generation boards, possibly with a different architecture and from a different vendor, very cumbersome.

- **Synthesis and PAR time:** On current EDA tools, large designs with high resource utilizations may take many hours to even days for synthesis and place-and-route (PAR). This makes making design changes and debugging a very slow process. Even a small change in the design requires re-synthesizing the circuit. This results in long development and turnaround time.

- **Chip area limitations:** Even with large FPGA chips, resources top-out at some point. If the design at hand does not fit in the available chip area, computation must be performed in phases. This requires saving the partial results off-chip, reconfiguring the chip and combining the partial results. This often results in poor overall performance.

Some of the challenges mentioned above are topics of active research. For example, there have been significant efforts towards the development of C-like high level languages for hardware design. Some of the commercially available tools include Handel-C, [Agi10], MitrionC [Mit10], CatapultC [Men10b] and Impulse-C [Imp10]. Also, some research groups have focused on standardizing the host-to-board interfaces. Availability of such high level languages would enable application specialists to generate efficient hardware without the need to have a deep understanding of logic design principles and the underlying hardware. In our experience, however, most of the currently available high level design languages are not able to exploit the underlying hardware resources efficiently and suffer from poor performance.

2.5 High Performance Computing using Graphics Processors

2.5.1 Overview of GPUs

Graphics processing units, as the name suggests, were originally meant for performing graphics processing. The main purpose of a graphics processing unit is to act as a coprocessor for the CPU for offloading the compute intensive task of pixel rendering. A graphics card, containing the GPU chip along with some on-board memory, is a peripheral card that plugs into the PCI or PCIe slot of a standard PC. Data is transferred to the card from the CPU and the graphics processor processes and generates millions of pixels per second, which are then painted on the display device. Traditionally, graphics processors contained a fixed graphics pipeline, performing a fixed set of vertex and pixel computations on very low precision data (8-bits). Over the years, graphics processors have evolved in three main respects: first, their computational capabilities have increased significantly, with hundreds of parallel processor cores on a single chip; second, the graphics pipelines have become much more flexible in the range of computations that they can perform; and third, the data-path now support high-precision computations, including double precision floating point. Though the bulk of the graphics processors are still used for graphics rendering, these capabilities of modern graphics processors have made them see applications in high performance computing.

Even though the evolution of GPU architectures made them suitable for general purpose computing, their recent outbreak in the field of HPC applications is mainly due to a different factor: “The milestone of GPU development of the recent years is the appearance of the unified architecture-based devices [TS08]”. These GPUs implement a massively parallel design, enabling them to efficiently perform highly parallel scientific computations [TS08]. In addition, high level graphics programming languages such as Compute Unified Device Architecture (CUDA) [NVI08b] from NVIDIA and OpenCL [Khr10] enable easy and efficient mapping of computations on graphics processors as data-parallel computations, without the need to lay them out as pixel-computations. This has led to the application of graphics processor units to HPC applications, with many modern supercomputers using GPUs alongside a microprocessors in each node [Top10, Tok10]. Researchers have shown speedups ranging from two-fold to multi-hundred fold on a wide variety of scientific applications.

One of the reason for such high raw performance delivered by the GPUs is the cache, or the lack thereof. Unlike microprocessors, where 50% to 90% of the chip area is devoted to the on-chip cache to hide the memory latencies, GPUs have little to no cache. Most of the transistors on a GPU chip are used towards computation cores, with very little flow control logic. GPUs hide the memory latency by having a large number of active light-weight *threads* and very fast and efficient, single-cycle context switch between threads that are waiting for data and the threads that are ready to be executed. Moreover, the computation cores are simple, allowing for hundreds of cores on a single chip. This provides immense parallelism, both coarse grained, at the processor level, and fine grained, at the thread level.

2.5.2 GPU Architecture

Modern GPUs can be seen as massively parallel, SIMD machines, with hundreds of processing cores on a single chip, along with some on-chip memory. They integrate with the CPU via PCI and PCIe based peripheral cards which, in addition to the GPU chip, also contain up to a few giga bytes of on-board memory.

High-end GPUs are available mainly from two vendors; NVIDIA and AMD-ATI. Below we discuss the hardware architecture of a high end NVIDIA GPU, the Tesla

C1060, which is the GPU used in the current work. Tesla C1060 is a GPU board designed specifically for high performance computing applications. In fact, it does not provide a graphics output and hence cannot be applied to graphics rendering.

The smallest atomic unit of an NVIDIA GPU is a Scalar Processor (SP). It represents a single processor core that is capable of executing instructions independently. Eight streaming processors constitute a multithreaded Streaming Multiprocessor (SM) [NVI08b], which is the basic unit of replication on NVIDIA GPUs. Based on the size, an NVIDIA GPU can have one or more SMs, with up to a maximum of 30 SMs on current generation NVIDIA GPUs. The Tesla C1060 is the largest current generation NVIDIA GPU, with 30 SMs or 240 SPs (see Figure 2·12), capable of delivering a peak raw single precision floating point performance of 930 GFlops.

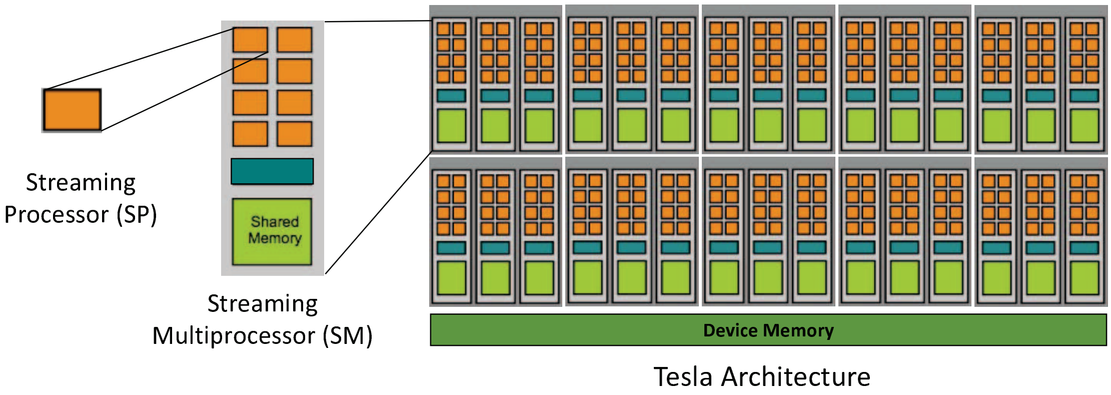


Figure 2·12: Architecture of NVIDIA TESLA C1060 [NVI10]

The eight cores in a streaming multiprocessor can be viewed as single-instruction-multiple-data (SIMD) cores. They share a single instruction unit and execute the instructions in lock-step. In addition to the eight SPs, each SM also contains two special function units (SFUs) for efficient computation of transcendental functions.

The memory hierarchy of NVIDIA GPUs is shown in Figure 2·13. Each SM contains four different types of memories:

- a small amount of low-latency, high-bandwidth *shared memory*. As the name suggests, the shared memory is shared by all the SPs on the SM. On current generation GPUs, the amount of shared memory per SM is 16K bytes,
- 16K local 32-bit registers. These registers are used for storing local variables used during program execution,
- a read-only constant cache, shared by all the streaming processors,
- a read-only texture cache, shared by all the streaming processors. This memory is accessed by the multiprocessors via a texture unit.

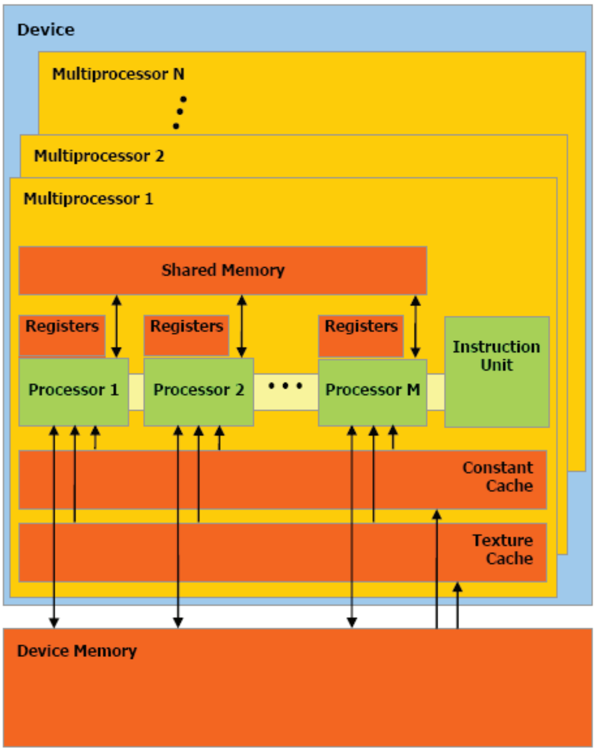


Figure 2·13: Memory hierarchy of the NVIDIA GPUs [NVI08b]

In addition to these memories, each SM can also access the on-board device memory, called the *global memory*. The Tesla C1060 contains 4 GB of global memory, accessible from all the multiprocessors. The bandwidth to the global memory, however, is an order of magnitude lower than that to the shared memory.

The data movement between the CPU (host) memory and the GPU (device) memory needs to be explicitly managed by the user. Data can be moved from the CPU memory to the GPU global memory using memory copy API from NVIDIA. The copy between the global and the shared memory needs to be explicitly performed by the GPU threads.

2.5.3 NVIDIA CUDA Programming Model

Traditionally, programming a graphics processor required the use of graphics APIs such as Microsoft's DirectX [Mic10] and OpenGL [Gro10]. With the growing interest of the HPC community in the use of graphics processors for computing, vendors such as NVIDIA and ATI introduced high level programming languages (CUDA and Stream respectively) to ease the use of graphics processors for general purpose computing. These languages enable easy and efficient mapping of computations on graphics processors as data-parallel computations, without the need to lay them out as pixel-computations. Below we discuss in detail the Compute Unified Device Architecture (CUDA) programming model from NVIDIA.

CUDA is a thread-based data-parallel programming model used for programming the NVIDIA GPUs. It is called the SIMT (single-instruction, multiple-threads) architecture, wherein multiple threads execute the same instruction stream in lock-step, on different data elements [NVI08b].

CUDA is based on standard C programming language, with extensions to support GPU constructs such as threads, thread organization, shared memory and data transfer between the CPU and the GPU memory. A CUDA program consists of a mix of standard serial C code that runs on the microprocessor and the parallel

CUDA *kernel* code that run on the GPU. The GPU kernel is *launched* from the CPU code using a configuration of threads that get mapped on the GPU multiprocessors. The multiprocessor maps each thread to one scalar processor, with multiple threads executing in parallel.

CUDA has three key abstractions: hierarchical arrangement of threads, the programmer managed shared memory and synchronization between threads [NVI08b].

- **Thread hierarchy:** In CUDA, threads are grouped into 1D, 2D or 3D *thread blocks.* Different thread blocks can then be grouped into a 1D or 2D *grid* of blocks (see Figure 2·14). An *execution configuration* defines the grid and block dimensions and is used while launching the GPU kernel. This configuration tells the thread scheduler how the threads are organized and determines the communication and synchronization between different threads.

- **Shared memory:** Each streaming multiprocessor contains a small local memory, called the shared memory, that is shared by all the threads within a thread block. The shared memory can be considered as a programmer managed cache and provides fast access to the data. The data from the host memory cannot be copied directly into the shared memory; it must go via the global memory. Once in the global memory, data must be explicitly copied into the shared memory by the threads. Shared memory provides an efficient way for threads within a multiprocessor to communicate or to efficiently reuse the data without having to access the slow global memory. The shared memory, however, is not persistent across kernel calls; the data in the shared memory is lost once a GPU kernel finishes and returns control to the CPU code.

- **Synchronization:** When a kernel is launched, the different thread blocks in the configuration are mapped on to the streaming multiprocessors (SMs) in

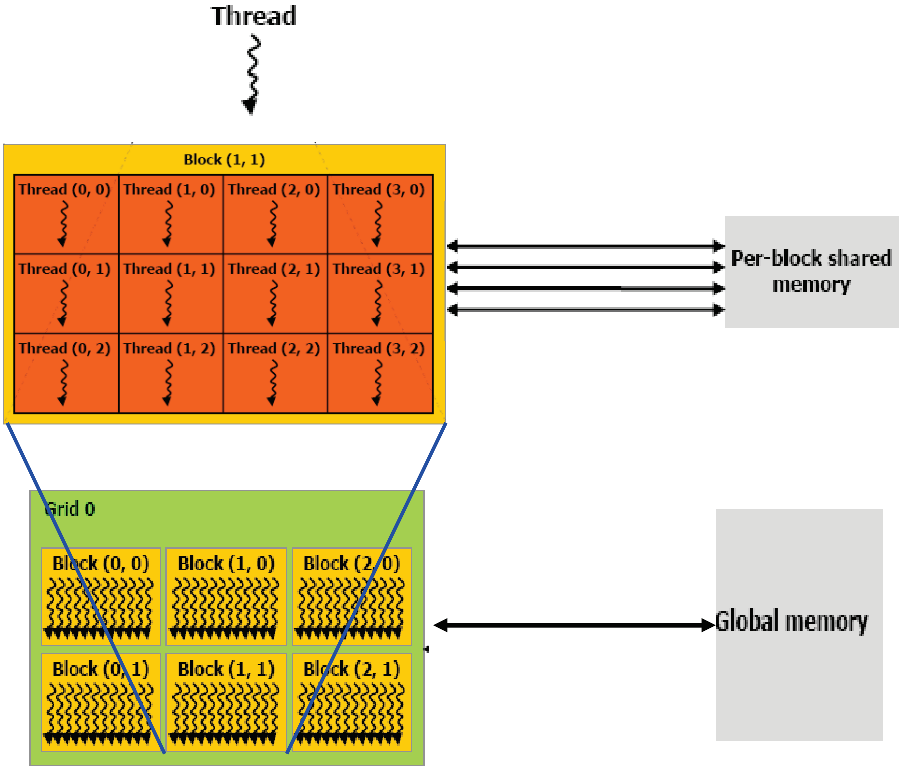


Figure 2·14: Organization of threads into blocks and grid [NVI08b]

their entirety; i.e. all the threads within a thread block will be executed by the processors in the same SM. Different thread blocks can get mapped to the same or different multiprocessors. Threads within a thread block execute concurrently and independently on a multiprocessor. They can be synchronized with each other using a simple synchronization primitive. Threads in different thread blocks, however, cannot be synchronized and are required to be independent of the threads in other thread blocks. The CUDA thread scheduler can schedule different thread blocks for execution in any order, in parallel or serially. Thread blocks can communicate only via the global memory, which is accessible to all the thread blocks. However, since the synchronization between different thread blocks is not defined within a kernel, multiple kernel launches will be required between the desired synchronization points.

As explained, CUDA requires explicit transfer of data between the CPU, the GPU global and the GPU shared memories. These data transfers constitute overheads and can be expensive. In order to achieve good performance, it is thus important to have large computation to communication ratio. Each thread must perform large amount of computations per datum fetched. In addition, since different memories provide different bandwidths, efficient use of the memory hierarchy is crucial. Where possible, data must be accessed from the shared memory and multiple access to the global memory must be avoided.

The configuration of threads and thread blocks is another important aspect of the CUDA programming. Since CUDA uses fast thread swapping to hide memory latency, having large number of threads is critical to achieving good performance. Similarly, it is beneficial to launch the kernel with large number of thread blocks. Thread blocks stream in pipeline fashion through the device. Having large number of thread blocks, thus, helps increase the device utilization and hence increases the overall performance.

2.5.4 Challenges with GPU Computing

Even though modern GPUs boast of very high raw computational capabilities, their relatively fixed architecture, memory hierarchy and communication and synchronization pattern limits the overall achievable performance. There are various challenges to achieving good performance from GPU computing. In particular, we experienced the following during the current work:

- **Distribution of work:** Distribution of work among different GPU threads needs to be explicitly done by the programmer. Naïve work distribution can lead to poor performance, mainly due to memory access conflicts and uneven processor utilizations. Efficient distribution of computations so as to achieve optimal performance is often non-trivial.

- **Data movement overhead:** GPUs incur large overhead for movement of data between the device and the host memory. If the computation per kernel is relatively small, this overhead can dominate the total runtime, resulting in poor performance.

- **Explicit copy to shared memory:** Even though the GPU shared memory provides low-latency access, the need to explicitly copy the data from the global memory nullifies any benefit if there is little or no data reuse.

- **Small shared memory:** Since the amount of shared memory per multiprocessor is relatively small, having a larger data set either requires accessing from the global memory or swapping; both of these deteriorate the performance.

- **Thread divergence:** As stated earlier, CUDA is a SIMT architecture, with multiple threads executing the same instruction stream in parallel. In case of divergence, i.e. threads branching to different targets, the threads get serialized, leading to degraded performance.

- **Thread communication and synchronization:** Since threads in different thread blocks cannot be synchronized and cannot communicate efficiently, certain computations are forced to be performed with a single thread block. A kernel launched with a single thread block will only utilize one of the multiprocessors, leading to heavy underutilization of the available processor cores. Having multiple kernel launches to enable synchronization, though possible, has two problems: first, it leads to large kernel launch overhead that cannot be justified unless the computation per kernel is very long, and second, upon kernel completion, the contents in the shared memory are lost and thus relaunching the kernel requires copying the data again from the global memory.

In the current work, we faced these challenges while trying to map our applications. We addressed these issues by redesigning the data structures to limit the communication overhead and to minimize serialization and maximize processor utilizations. In some cases, however, we found that having complex schemes to increase the processor utilizations leads to large overheads and eventually worse performance.

2.6 FPGAs versus GPUs

Even though both the FPGAs and GPUs are very promising for high performance computing, they are architecturally very different and display different strengths and weaknesses. Here we outline these differences briefly.

- **Processor cores:** One of the most important difference between the FPGAs and the GPUs is with respect to the processing elements. While the FPGAs provide the ability to design application-specific processor cores, just like an ASIC, the processors on a GPU are fixed. This ability to customize the processors enables the FPGAs to achieve higher parallelism through the use of bit-level and computation-specific optimizations.

- **Communication between the processors:** The highly configurable interconnect network on the FPGAs enables very flexible communication between the processors, customized to the requirements of the design at hand. GPUs, on the other hand, allow very restrictive communication and synchronization between the processors.

- **Operating frequencies:** Most current generation high-end FPGAs have a maximum clock frequency rating of 500 MHz, with frequencies around 100 to

200 MHz. being more realistic for large designs. The processors on a high-end GPU, on the other hand, operate at clock frequencies higher than a GHz.

- **Floating Point Units:** Currently, FPGAs do not contain any embedded floating point units; GPUs, in contrast, contain hundreds of optimized floating point units. Due to this, the peak floating point performance of a high-end GPU is 5× – 10× higher than that of an FPGA.
- **Power Consumption:** FPGAs have clear and definite advantage over the GPUs in terms of the power consumption. While the GPUs typically dissipate around 180 watts, the power consumption of current generation FPGAs is less than 30 watts.

2.7 Summary

The use of accelerators for addressing the computations in complex scientific problems is a relatively new technique and more importantly, has been proven effective only in the last decade or so. In particular, FPGA and GPU based systems have been shown to achieve two to three orders of magnitude chip-to-chip speedup over microprocessor based implementations.

Though the use of the computational accelerators for improving the performance of the computationally demanding applications is very promising, it is often non-trivial. As discussed in this chapter, achieving high performance from accelerator-based computing poses numerous challenges. The first among these is the Amdahl's law. To achieve any worthwhile overall performance improvement, at least 90% of the target application must lend itself to substantial acceleration. Secondly, it is important to avoid the implementational overhead; if careful design measures are not taken, losing the performance speedup achieved from the accelerated parts is rather easy. Another important criteria in the design of an accelerator system is the development time. If the development of an accelerator takes longer than it takes for a new process technology to arrive, the performance benefit would diminish drastically, making the acceleration unattractive.

In addition to the above challenges, implementing efficient acceleration routines requires careful consideration of a number of other factors such as high utilization of the available computational resources, minimum and efficient data transfer between the accelerator and the host, restructuring of the original computations and careful precision manipulations.

Chapter 3

Molecular Docking and Binding Site Mapping

3.1 Overview

Computational modeling of molecular activities is a vast and important field of research; molecular docking and binding site mapping are two such modeling methods. Molecular docking aims at predicting the interactions between two molecules and has been heavily used in the industry for designing new drugs [HMWN02]. Binding site mapping, on the other hand, is a relatively new and promising method for drug discovery. It helps identify the sites on the surface of a large molecule where the interaction with a small molecule is most likely to occur. The knowledge of the binding site then allows the localization of the search to a small region, enabling faster docking.

Drug activity is obtained by the molecular binding of a small molecule (the ligand) to the binding site of a large molecule (the receptor), usually a protein [TJK01]. The process of drug discovery, thus, involves finding the appropriate ligand molecule and orienting it in the binding site on the protein. This typically requires screening millions of candidate ligands from large ligand databases. Computational methods thus play an important role in fast and effective drug discovery. Among these, the key methodology of docking a small molecule to protein binding sites was pioneered in early 1980s and remains an active research area [KDFB04].

The need for effective docking can be exemplified by the examples of the drugs whose development was based on, or heavily influenced by, the use of docking methods; these include the HIV protease inhibitors and the peptide antigens for the MHC receptors [RVD95, KDFB04]. For the usefulness of molecular docking in drug discovery, however, accuracy and speed are two key factors; these requirements are often contradictory [BK03].

The two main aims of molecular docking studies are accurate structural modelling and correct prediction of activity [KDFB04]. Current docking algorithms employ a variety of efficient searching schemes; most methods, however, still have difficulty in identifying the correct solution from false positives [Rit08]. This is because organic molecules contain many conformational degrees of freedom and modeling them with sufficient accuracy is computationally challenging [KDFB04]. Modeling techniques can handle up to 30 degrees of freedom in a flexible ligand; modeling the full flexibility of even a small protein requires more than 1000 degrees of freedom [TJK01].

Due to these challenges, the docking task is generally split into multiple phases; the initial phase performs fast pruning of the solution set using approximate techniques such as rigid docking. This is followed by a refinement phase such as energy minimization that performs more accurate modeling of the chemical properties [KDFB04]. With few tens of seconds for evaluating each ligand, screening millions of ligands is still a time consuming process. The acceleration of molecular docking and mapping applications is thus highly desirable; faster docking and mapping algorithms would enable faster drug discovery and allow better modeling of complex interactions between the molecules. Our studies indicate that the computations involved in both docking and mapping task are amenable to parallelization and hardware acceleration. We discuss them in detail in later sections.

3.2 Molecular Docking Basics

Docking refers to the computational prediction of the structure of the intermolecular complex that is formed when two independent proteins interact. “Like all good scientific problems, the protein docking problem is easy to state but hard to solve” [Rit08]. The field of molecular docking is vast, with a variety of different docking applications and techniques been developed over the past 15 years [Kro03]. “The number of algorithms available to assess and rationalize ligand protein interactions is large and ever increasing” [TJE02]. Each year, hundreds of studies are published describing new and improved techniques for docking, with many review articles discussing the recent developments in the field and the shortcomings of the current methods.

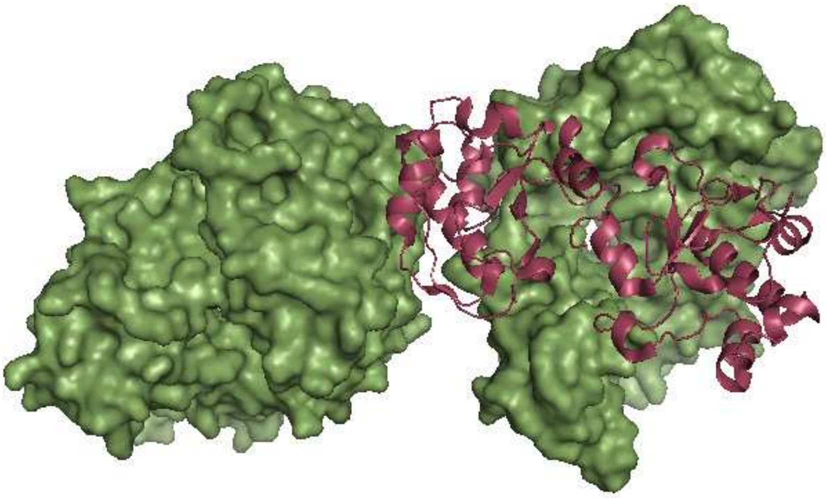


Figure 3·1: Interaction between two molecules (generated using Pymol [Pym10])

Starting from the atomic coordinates of the two molecules, docking aims at predicting their correct bound structure [HMWN02]. Interaction of molecules through non-covalent bonding is a fundamental operation in biochemistry (Figure 3·1). Modeling this process of molecular *docking* is critical both to evaluating the effectiveness of pharmaceuticals and to developing an understanding of life itself. In the former, millions of drug candidates may need to be evaluated for each molecule of medical importance. Computational experiments are faster and more cost effective and are therefore preferred over chemical [BK03].

Two proteins can interact in different relative orientations, forming different complex structures. Docking involves finding the relative offset and rotation (pose) that gives the strongest interaction. Different poses are evaluated based on some energy functions and the lowest energy poses are reported. Finding energetically favorable poses generally involves exhaustive search in the entire rotational and translational space and is thus computationally expensive. In addition, there are several other issues. First, biomolecules are flexible and undergo conformational changes upon binding. They contain flexible side chains that can flex and rotate around chemical bonds. Second, certain molecule pairs interact only when one (or both) flex, in a process known as induced fit. Modeling induced fit (in some cases) requires dynamic modeling, e.g., based on molecular dynamics. Third, the best docking pose for many molecule pairs can be determined via simple computations, but that of other pairs may be difficult even with the most sophisticated [KDFB04]. As a result, hierarchical methods are often used: an initial phase where candidate poses are determined and an evaluation phase where the quality of the highest scoring candidates is rigorously evaluated. Most often, the first step is referred to as docking and the second as scoring.

The first challenge in molecular docking is to reliably predict the conformations that closely resemble the native structure of the complex [Rit08]. There are two major computational limitations that affect the ability to generate near-native conformations: lack of appropriate scoring functions and the inability to correctly model the flexibility in the molecules [TJK01].

In order to predict the near-native structures, docking programs often generate a large number of candidate orientations that are further processed using complex scoring functions. Discriminating the *true-positive* conformations from the *false-positives* requires that the scoring functions accurately represent the physical phenomenon governing the interactions between the molecules. In order to reduce the computational run times, however, the scoring functions implemented in docking programs make various simplifying assumptions [KDFB04]. Extending the scoring function to account for terms such as solvation and entropy has been shown to generate better results, albeit at the cost of increased computational expense. Moreover, it has been observed that scoring functions perform differently on different target complexes [KDFB04], thus making it difficult to develop a good scoring function that performs well on a wide range of complexes. Finally, even though molecule binding is controlled by a combination of enthalpic and entropic effects, and either of them can be more dominant in a specific interaction, most scoring functions do not model the entropic effects [KDFB04].

The second challenge to the current docking programs is to accurately model the flexibility in the molecule's side chains. Molecules undergo significant conformational changes in the presence of other molecules and during docking [VC04]. The side chains on the molecules have various degrees of rotational and angular freedom. Accounting for these conformational changes is difficult both from the point of view of developing accurate models to represent these flexibilities and the computational power required to iterate through all the degrees of freedom. As stated earlier, one of the main applications of docking is to screen large databases of compounds for finding potential drug candidates. This process is called virtual screening, wherein each compound in the database (called the ligand) is docked against a target protein (called the receptor). In order for a docking program to be effectively applied for virtual screening, it must be very fast to evaluate millions of ligands in a reasonable amount of time. Even with very fast docking and scoring methods, modeling the full flexibility of both the ligand and the receptor molecules is prohibitively slow [KDFB04]. Due to this, most docking programs either consider both the molecules to be rigid or model the flexibility in only the ligand molecule. Ignoring the flexibility in the receptor molecule, however, sometimes leads to predictions different from the native structure.

3.3 Types of Docking

3.3.1 Overview of Docking Algorithms

Docking algorithms can be classified based on a variety of different criteria such as the representation of the molecules, the search strategy employed, the energy functions used to score the relative poses, the size of the interacting molecules and the amount of molecular flexibility modeled.

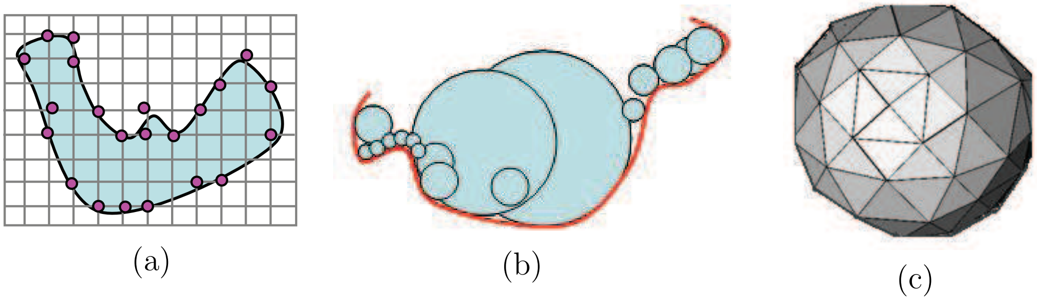

(a) (b) (c)

Figure 3·2: Different representations of molecular surface; (a) Grid, (b) Overlapping spheres [SKB92], and (c) Triangles [MST10]

Molecules can be represented in a number of ways; some commonly used representations include:

- **Grid-based representation:** Here, the surface of the molecule is represented as points on a 3-D grid [KDFB04] (Figure 3·2(a)). Each point on the grid represents whether the protein surface lies inside, outside, or on the grid point. The docking programs then match the shape of the two interacting molecules by computing 3D correlations between the two grids.

- **Sphere-based representation:** In this case, the shape of the molecule is represented as a set of overlapping spheres along the surface contour (see Figure 3·2(b)). Molecules are matched for shape complementarity by matching the distances between the spheres on the two molecules [SKB92]. This matching task is often represented as a graph-search problem [EK97].
- **Triangle-based representation:** Triangle-based representation of a molecule is shown in Figure 3·2(c). This representation is similar to the technique used in 3D graphics wherein the 3D shape is represented as a set of triangles [LHD88]. Shape matching is then done by projecting the triangles from one shape to those on the other, using a technique called geometric hashing [WR97].
- **Atomic representation:** In this representation, atoms are represented explicitly using their coordinates in 3D space. It is generally used only in conjunction with a potential energy function [TB00]. Moreover, due to the computational complexity of evaluating the pairwise interactions between the atoms, atomic representation is often used only during the final scoring phase [KDFB04].

Based on the search strategy, docking algorithms can be divided into the following categories:

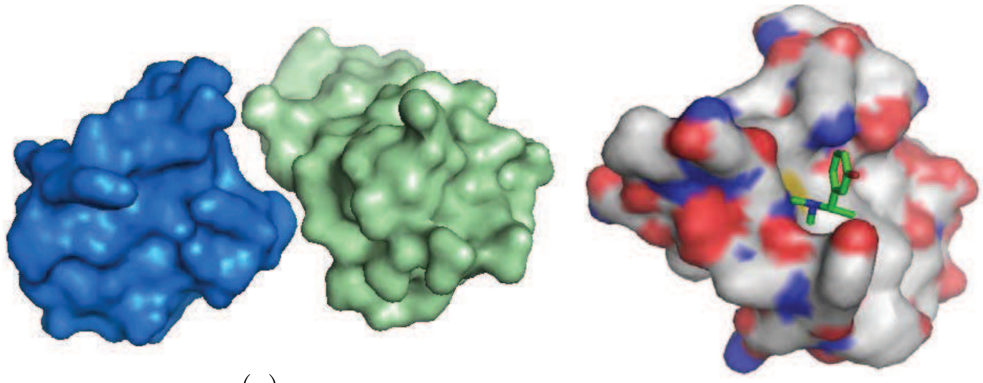
- **Exhaustive search algorithms:** Exhaustive search algorithms, as the name suggests, exhaustively evaluate all the different relative poses between the two molecules and find the best fit. Due to the huge combinatorial search, these algorithms tend to be computationally very expensive [HMWN02]. To limit the computational runtimes, these algorithms often use very simple scoring functions and are used as the first stage of hierarchical docking systems [KDFB04]. An example of the exhaustive search algorithm is the grid-based rigid docking algorithm using FFT.

- **Systematic search algorithms:** Systematic search algorithms are used to account for the flexibilities in the molecule. These algorithms systematically explore the molecule's degrees of freedom in a combinatorial fashion. Anchor-and-grow or incremental construction algorithm is an example of a systematic search algorithm [BK03].
- **Stochastic search algorithms:** Stochastic algorithms apply random perturbations to the two proteins and evaluate the resulting pose. The aim is to achieve at the near-optimal solution by randomly moving from one position to the next. Examples of stochastic search algorithms include Monte Carlo, Newton Raphson and genetic algorithms [BK03].

Docking computations are generally used to model one of the two types of interactions: between two proteins (protein-protein docking) or between a protein or other large molecule and a small molecule (small molecule docking or protein-ligand docking). The difference in the sizes of the interacting molecules leads to there being a divergence in optimizations, with docking codes sometimes specializing in one domain or the other.

- **Protein-protein docking:** In the case of protein-protein docking, both the interacting molecules are relatively large and are of comparable sizes (Figure 3·3(a)). This type of docking is used primarily to predict the structure of the protein-complex formed when two proteins interact. This helps answer the questions that are essential to understanding how the proteins function and hence the understanding of life itself. Some of the commonly used protein-protein docking programs include FTDock [GJS97], DOT [EMRP95], ClusPro (PIPER) [CGVC04], HADDOCK [DBB03] and HEX [RK00].

- **Protein-ligand docking:** Protein-ligand docking refers to the process of *fitting* a small molecule (ligand) into a *pocket* on the surface of a large molecule (protein; also known as the receptor) (Figure 3·3(b)). Here, the process involves finding the most optimal orientation of the ligand into a binding pocket on the protein. Often, the binding pocket is assumed to be known [HMWN02]. This type of docking finds application in the process of drug discovery wherein millions of candidate drug molecules (ligands) are evaluated against the target protein to find the one that binds with high affinity. Examples of popular protein-ligand docking systems include GOLD [JWG95], AutoDock [MGH$^+$98], DOCK [SKB92], GLIDE [RSF07], ICM [ATK94], and SITUS [CW02a]. Of these, DOCK and ICM can also perform protein-protein docking.


(a)

(b)

Figure 3·3: (a) Protein-protein docking and (b) Protein-ligand docking

Due to the difference in the sizes of the molecules involved, the methods used in protein-protein docking usually differ from those applied for protein-ligand docking. Protein-protein docking is often performed using exhaustive search methods. Most protein-protein docking programs start with a phase that treats the molecules as rigid bodies and generates a large number of docked conformations [VC04]. This is often followed by the steps of clustering the conformations based on their RMS deviations and ranking the clusters [CGVC04]. Protein-ligand docking, on the other hand, is mostly performed using stochastic or systematic search algorithms, or using molecular dynamics simulations [TJE02].

As stated earlier, modeling the full-flexibility in the molecules is computationally very expensive and leads to prohibitively slow algorithms. The process of molecular docking is thus divided into different steps. Based on these different steps, docking algorithms can be divided into two broad categories: rigid docking and flexible docking. Rigid and flexible docking systems are usually applied to different classes of problems; rigid docking is mostly applied for exhaustive search in protein-protein docking. Flexible docking, on the other hand, is used for orienting a flexible ligand in a receptor, often in a known binding site. Due to this, the underlying algorithms, representations and the computational structures are quite different. In the following subsections, we discuss both these classes of docking algorithms in detail and summarize some of the commonly used docking programs in each category.

3.3.2 Rigid Docking

The aim of molecular docking is to find the optimal conformation between the two interacting molecules. Two molecules can have in the order of billions of possible conformations [Rit08]. Exhaustively evaluating all of these conformations using complex energy functions that model the molecular properties accurately is computationally very demanding and requires prohibitively long computation runtime.

Moreover, modeling the inherent flexibility in the molecules further adds to the computational complexity of the problem. In order to limit the computational runtime, many docking programs assume, at least initially, a lock-and-key model wherein the two interacting molecules are considered to be rigid (see Figure 3·4). In other words, conformational changes in the molecules during binding are ignored. This process is called rigid docking. Though it does not account for the molecular flexibility, rigid docking still allows modeling of various force laws that govern the interaction between the molecules, including geometric, electrostatic, atomic contact potential, and others, while reducing the computational runtime significantly.

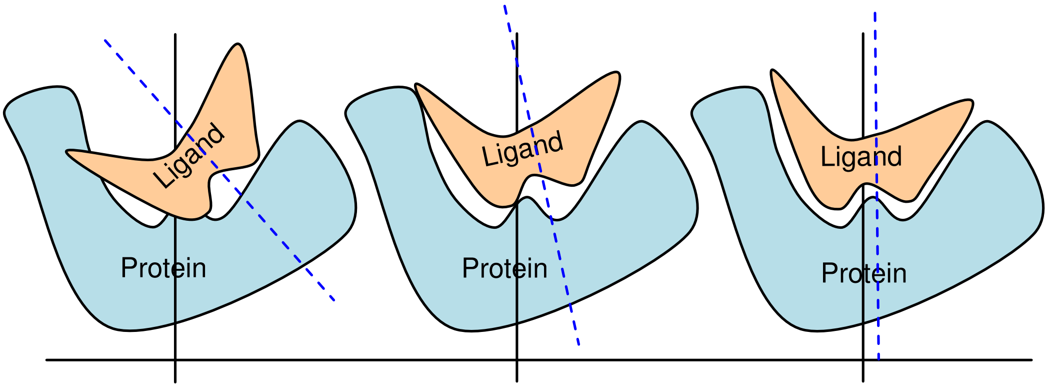


Figure 3·4: Lock and key model used by rigid docking programs

The main goal of rigid docking is to quickly find a few thousand top scoring conformations between the two interacting proteins using some relatively simple scoring functions. Most docking routines then subject these candidate conformations to further analysis where energy evaluations are performed using sophisticated scoring functions and the molecule flexibility is modeled, either explicitly or implicitly. In this respect, rigid docking can be considered as an initial filtering stage to limit the search space. It finds applications in many molecular modeling programs such as protein docking, clustering, and mapping.

Many rigid docking programs perform exhaustive search and return the top scoring conformations. To ensure that no true-positives are missed, the number of conformations retained is usually large. Clearly, the most desirable property of a rigid docking program is to limit the number of false-positives. This, however, requires the use of complex scoring functions, which in turn increase the computation runtimes. Over the years, various rigid docking programs have been developed that employ different scoring functions. Some of the more popular ones are discussed below.

Overview of Rigid Docking Systems

The most basic approach for rigid-docking is the FFT correlation based systematic search of the entire 6D space. Different FFT-based systems differ mainly in the energy function used to evaluate the docked pose. This is an important distinction since the computational expense as well as the accuracy of the algorithm depends on the complexity of the energy function [TJK01]. Too elaborate scoring functions tend to make the algorithm very slow whereas a very simple scoring function results in a large number of false positives. Original FFT-based rigid docking programs perform initial filtering using only simple geometric complementarity and use more complex energy functions in subsequent stages. In order to improve the discrimination sensitivity, rigid docking programs in the last 10 years have started incorporating more complex energy functions, such as electrostatics and desolvation, in the initial stage itself [VC04]. The increased program complexity is justifiable since it reduces the number of candidates for the subsequent stages.

In addition to the FFT-correlation technique, some other search schemes have also been applied to rigid docking. These include geometric hashing [SDINW05] and clique-detection [SKB92]. Unlike the FFT-based methods, however, none of these schemes perform exhaustive search of the entire conformational space.

In the current study, we focus on the FFT-based exhaustive rigid docking systems. Various FFT-based docking programs have been developed; some of these are publicly available with source code, whereas some are available in the form of program binary or web servers. Many rigid docking programs are proprietary and are available only through licensing.

A list of some of the popular FFT-based rigid docking programs is presented in Table 3.1. Of these, the FRED program [Sof10] performs docking based on the complementarity of the shapes of the two molecules [MAN$^+$03]. DOT [EMRP95] and FTDock [GJS97] programs model the electrostatic interactions in addition to the shape matching. In order to improve the discrimination sensitivity of the program, PIPER [KBCV06] and ZDOCK [CW02b] further extend the matching criteria to include the desolvation effects. The modeling of the desolvation in PIPER, however, is more complex and results in better discrimination of the native solutions compared to the ZDOCK program [Rit08]. The PIPER program is described in detail in section 3.5.

Table 3.1: Sample of Rigid Docking programs.

Program	Developed at	Energy functions used
DOT	San Diego Supercomputing Center	Electrostatics and van der Waals
ZDOCK	ZLab at Boston University	Shape complementarity Electrostatics and desolvation
PIPER	Structural Bioinformatics Lab at Boston University	Attractive vdW, repulsive vdW Electrostatics and desolvation
FRED	OpenEye Scientific Software	Gaussian function based shape complementarity
FTDock	Cancer Research UK	Shape Complementarity and electrostatics

Computations in FFT-based Rigid Docking

The computations performed by the different FFT-based rigid docking systems are very similar, differing mainly in the type and the number of energy functions used and thus the number of FFT correlations to be performed. The process starts by mapping the two molecules onto individual 3D grids. To evaluate the different poses between the two grids, the receptor grid is held fixed and the ligand grid is moved around it. With typical grid sizes of N = 128 in each dimension and the total number of angles around 10,000, 10^{10} relative positions are evaluated for a single molecule pair. Typically, the outer loop consists of the rotations while the inner loop scans the entire 3-axis translational space with a 3D correlation. Since the latter requires $O(N^6)$ operations, this type of exhaustive search was long thought to be computationally infeasible [KBO+82]. The introduction of the FFT to docking [KKSE+92] reduced the complexity of each 3D correlation to $O(N^3 \log N)$ for steric (shape only) models; further work expanded the method to electrostatic [GJS97] and solvation contributions [CW03]. Once all the relative poses between the two grids have been evaluated, the docking program selects and reports the top scoring poses.

The different computations involved in rigid docking can be listed as follows:

1. **Rotation:** This step involves rotating the ligand grid around each of its three Cartesian axes. Rotations can be performed at different angle increments. Smaller angle increments generate better docking results but require more rotations to be evaluated. Increment angles of 5 to 15 are typically used, though the user can provide any value. The process involves generating the rotation matrix for the current angle of rotation and multiplying it to the ligand grid to generate the co-ordinates of the rotated ligand. This is typically performed for thousands to tens of thousands of rotation angles.

2. **Charge Assignment:** In order to perform correlation, charges and potential values for different energy functions need to be assigned on the receptor and the ligand grids. Since the receptor is held fixed, the charge assignment for the receptor is done only once. For the ligand grid, values for different energy functions are re-assigned after each rotation. The process involves interpolation of space charges on fixed set of grid-points.

3. **FFT, modulation and IFFT:** FFT of the receptor grid is taken once and stored. For each rotation, FFT of the ligand grid is taken. The transformed grids are then modulated (multiplied) and an inverse FFT is taken to obtain the correlation scores for all the possible 3-axis translations for the current rotation. This is repeated for every energy function. Thus, for k energy functions, k forward and k reverse transform operations are required at every rotation. This step accounts for most of the computation runtime of rigid docking.

4. **Filtering of top scores:** For each rotation, a pre-specified number of top-scoring poses are reported. Often, many high scoring poses tend to appear close to each other. Simply reporting the high-scoring conformations would thus return all the poses near the global maxima. To report multiple distinct high scoring conformations, various complex filtering schemes can be used. One such approach is to divide the result space into different regions and report a pre-specified number of top scoring poses from each region. Another scheme includes excluding the conformations surrounding the current best conformation from consideration while reporting the next best. Both these schemes ensure that the reported conformations are dispersed over the entire set of conformations.

For each molecule-pair to be docked, the above steps are repeated for tens of thousands of rotations. With complex scoring functions requiring the evaluations of multiple 3D FFTs of typical sizes 128^3, the per rotation runtime can be as much as 10 seconds. With 10,000 rotations typically evaluated, this results in a total runtime of about 28 hours for a rigid-docking run.

3.3.3 Flexible Docking

As discussed previously, docking programs often model the receptor and the ligand molecules as rigid-bodies to limit the computational complexity. Biomolecules, however, are flexible and undergo conformational changes upon binding. Rigid docking has its limitation in that it does not account for this flexibility [BK03]. Relaxing the rigid-body approximation is often considered essential to improving the prediction of the docked complexes, especially for small molecule docking. The availability of faster computers and supercomputers has allowed some conformational flexibility to be taken into account. Many algorithms have been developed that consider the ligand to be flexible, though the receptor flexibility is still rarely explored [Kro03]. Furthermore, unlike rigid docking programs, most flexible docking systems do not perform an exhaustive search along the protein-surface. They assume that the ligand binding site is known in advance, either from a previous rigid docking run or through x-ray crystallography. Flexible docking is then performed on a small local region, e.g. the binding site, to find the relative orientation of the ligand in the binding site, accounting for the flexibility in the ligand side chains.

Algorithms that consider ligand flexibility can be classified into three broad categories: systematic, stochastic and deterministic [BK03].

- **Systematic search algorithms:** Systematic search algorithms combinatorially explore a grid of values for each degree of freedom. The number of evaluations needed increases rapidly with the increase in the degrees of freedom.

Some termination criteria are thus used to limit the search space. An example of systematic search algorithm is anchor-and-grow or incremental construction [BK03].

- **Stochastic algorithms:** Stochastic search algorithms aim to find the global energy minimum by making random changes along the ligand's degrees of freedom. A major problem with such algorithms is that convergence is not guaranteed. Multiple iterations are thus performed to improve convergence. Examples of stochastic algorithms include Monte Carlo methods and genetic algorithms [BK03].
- **Deterministic algorithms:** Deterministic search algorithms start from a known initial state and repeatedly move to a next state of equal or lesser energy, until some termination criteria are met. Such algorithms, when started again from the same initial state, always generate exactly the same final state, thus the name deterministic. The problem, however, with such algorithms is that they often get trapped in local minima. Molecular dynamics simulation is an example of deterministic search algorithm [BK03].

Overview of Flexible Docking Systems

There are a large number of flexible docking programs based on the above mentioned algorithms. Some of the more commonly used programs are as follows:

AutoDock [DSG90, MGH$^+$98] is a flexible docking program developed at the Scripps Research Institute. It uses simulated annealing based Monte Carlo simulation technique. The ligand is considered as a whole and is randomly placed in the binding site [BK03]. The algorithm aims to find the global minimum by performing several cycles of simulations at different temperatures, with each cycle consisting of tens of thousands of steps.

ICM [ATK94] is another popular flexible docking program based on Monte Carlo simulations. It samples the translational and rotational degrees of freedom using pseudo-Brownian motion and the ligand flexibility using biased probability moves [BK03].

GOLD [JWG95] is a flexible docking program that is also based on stochastic techniques but uses evolutionary algorithms to arrive at the global energy minimum. The technique comprises of carrying the fittest "individuals" to the next generation while discarding the not-so-fit ones [BK03].

FlexX [RKLK96] program falls in the category of systematic search algorithms. It performs docking of a flexible ligand into a rigid receptor using a technique called anchor-and-grow. The ligand flexibility is considered by dividing the ligand into a base fragment and a remainder fragment. The remainder fragment is further broken at each rotatable, acyclic single bond, creating many small fragments. The base fragment is rigidly docked into the receptor using pose clustering (triangle-matching) technique [LHD88] used in computer vision. A set of favorable placements of the base fragment are selected, to be extended further. To each placement, the rest of the ligand fragments are added using incremental construction approach, formulated as a tree search problem.

DOCK [KBO$^+$82, DSD$^+$86] is perhaps the most commonly used flexible docking program. It is a protein-ligand and protein-protein docking program developed at the University of California, San Francisco. Though DOCK also uses incremental construction method for flexible docking, it differs from FlexX in every respect: representation of the molecular surface, the ligand fragmentation, the method used for rigidly docking the fragments, and finally the reconstruction of the complex by joining the fragments.

Due to the popularity and wide acceptance of DOCK programs in the molecular modeling community, we studied the various versions of the DOCK program in detail. Since its initial version, DOCK has undergone a number of algorithmic changes. The first version of DOCK [KBO+82] performs protein-ligand docking and treats the interacting molecules as rigid bodies. Docking is performed using a geometric approach, accounting only for steric overlaps. The central idea is the representation of the molecular surfaces as sets of overlapping spheres (see Figure 3·2(b)) and matching the surface properties using distance-matching between the spheres on one molecule and those on the other. The final step involves optimizing the position of the ligand within the protein binding site.

The next version of DOCK considered the ligand flexibility while treating the receptor as rigid [DSD+86]. Unlike the original DOCK, here, the ligand is docked into a known binding site on the receptor, making the search local. Ligand flexibility is considered by partitioning it into a small set of large rigid fragments, each fragment being the largest that can be treated as rigid. Each rigid fragment is docked separately into the receptor binding site using the original DOCK algorithm. These fragments are then incrementally grouped based on the atom-atom distances between pairs of fragments. This results in many fragment groups; each group is then energy minimized. Energy minimization is an important step since it allows the modeling of induced fit between the ligand and the receptor [DSD+86]. It is described in detail in section 3.3.4.

The original distance-matching algorithm used by the DOCK program requires combinatorial search and is computationally very expensive. DOCK2 [SKB92] addresses this by formulating the distance matching as a graph problem, using bipartite graphs. This speeds up the sphere-matching process, allowing DOCK2 to perform protein-protein docking, which is prohibitively large for earlier versions of DOCK [SKB92]. DOCK2 also incorporates 3D grid-based scoring of the docked complex using atomic contact potential. DOCK 3.0 [MSK92] further extends this to include other scoring functions such as electrostatic energy and molecular mechanics interaction energy.

To further improve the computational runtimes, DOCK 4.0 replaces the bipartite graph algorithm with exhaustive matching based on single docking graph method [EK97] and clique detection [BK73]. In a single docking graph, each sphere-pair is represented as a node in the graph and the edges connect to other nodes with pairs of matching distances. The problem of shape matching then reduces to finding the cliques, the completely connected subgraphs within the undirected graph. The single docking graph can be represented as an adjacency matrix. The main advantage of this representation is that all the necessary distance comparisons are performed while constructing the adjacency matrix [EK97], which is a sparse matrix. This technique enables very efficient docking.

DOCK versions 4.0 [EMSK01] and 5.0 [MLP$^+$06] use the single docking graph algorithm to efficiently dock flexible ligands into protein binding sites, docking most test cases in a few seconds on the SGI R10000 CPU [EMSK01]. This enables the application of the DOCK algorithm for searching database of flexible ligands. Ligand flexibility is modeled using the anchor-and-grow algorithm, wherein the rigid portion of the ligand is docked first using geometric matching, followed by incremental addition of flexible side chains. The final step involves optimization of the bound complex using energy minimization. In terms of computational requirements, this step tends to dominate the overall runtime.

The computational complexity involved in the original DOCK algorithm makes it an interesting and ideal candidate for acceleration, leading to our detailed investigation into the algorithm and its computationally intensive steps. As discussed above, however, the current DOCK algorithm applies various simplifications and optimized graph algorithms during the search (docking) phase, making it very fast on a serial computer itself. Our runtime profiling of the DOCK program indicates that the docking phase requires only a few hundred milliseconds to a few seconds, with energy minimization requiring up to 100 seconds for minimizing 1000 orientations generated by the docking phase. Even though the energy minimization step dominates the total computation time, the time for a single minimization is only about 100 milliseconds. Moreover, minimization is an iterative process, with each iteration being dependent on the previous one. This makes it hard to parallelize. Further, our investigation into the energy minimization step of DOCK shows that the computations involved are highly conditional, with multiple branches based on certain parameters. This makes it challenging for parallelization and acceleration. We, therefore, did not pursue the acceleration of the DOCK program any further.

Energy minimization, however, is applied in a variety of flexible docking and mapping applications and thus its acceleration is certainly desirable. In the current work, we accelerate the energy minimization phase of a solvent mapping program, wherein it is used for modeling the side-chain flexibility of ligand molecules. This is discussed in detail in section 3.6.

3.3.4 Energy Minimization

Energy minimization is an iterative process which aims at computing the configuration of the atoms in a complex that corresponds to the minimum potential energy. It is used during molecular docking and solvent mapping to model the flexibility in the side chains of the ligand.

As discussed earlier, modeling the molecule flexibility during docking is computationally very demanding and docking programs often approach this in two steps. The first step quickly generates approximate docking poses between the two molecules using simple energy functions. This is followed by a refinement step that more accurately models the chemical activities between the molecules [VC04]. Energy minimization is one such refinement step, often performed on a local region, e.g. the binding site, to accurately orient a flexible ligand into the protein. During minimization, the larger molecule is held constant while the side chains of the ligand are free to move [SDV07], thus accounting for their flexibility.

Many docking programs perform energy minimization on the top scoring poses returned by the rigid docking step. Examples of some docking and mapping programs that employ energy minimization include DOCK [MGBK93], DARWIN [TB00], RDOCK [LCW03], EADock [GZM07], FTMap [BKC$^+$09], CSMap [LJY$^+$07] and MCSS HOOK [EWKH94].

Energy minimization performs repeated evaluation of the energy of the complex until some convergence criteria is met. During each minimization iteration, the total potential energy of the complex is computed. In order to evaluate the potential energy of the system rapidly, it is often represented using force-fields. A force-field represents each atom in a molecular system as a point charge and the total potential energy of the system as a sum of various two, three or four-particle interactions [Mat92]. Various force fields have been developed, with the more popular ones being CHARMM (Chemistry at HARvard Molecular Mechanics) [BBO$^+$83] and AMBER (Assisted Model Building and Energy Refinement) [CCB$^+$95]. Due to the popularity of the CHARMM force fields, energy minimization is often referred to as minimization of the CHARMM potential or simply as CHARMM minimization.

Minimization involves computing the potential energy of the complex at a point, updating the forces acting on the atoms, and adjusting the atom coordinates according to the total forces acting on them. Forces acting on the atoms are obtained by differentiating the potential energy function with respect to the atom coordinates.

This process of energy evaluation and of force and position updates is repeated for many iterations until the energy of the system converges to within a threshold (see Figure 3·5).

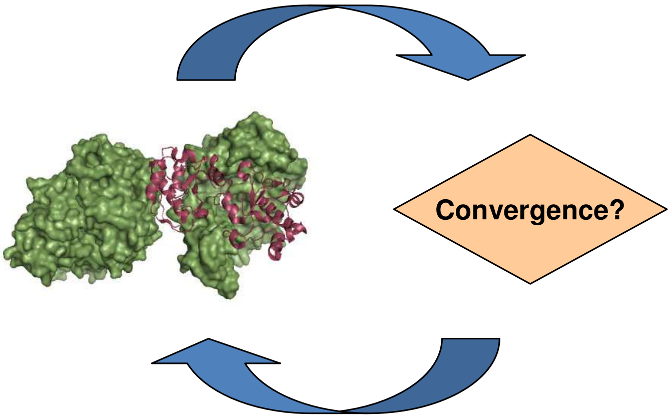


Figure 3·5: Iterative process of energy minimization

During minimization, the move to the next iteration can be made using one of many optimization approaches such as steepest descent, conjugate gradient, quasi-Newtonian, or Newton-Raphson. Depending on the method chosen, minimization requires computing the first and, in some cases, the second derivatives of the energy functions. The choice of iteration method also affects the rate at which the energy of the system converges.

Computations in Energy Minimization

In energy minimization, the system to be simulated consists of a number of atoms; the total energy of the system is a sum of various bonded and non-bonded energies of all the atoms (Equation 3.1).

$$E^{total} = \underbrace{E^{vdw} + E^{elec}}_{non-bonded} + \underbrace{E^{bond} + E^{angle} + E^{torsion} + E^{improper}}_{bonded} \tag{3.1}$$

The bonded energy terms represent the energies between 2, 3 and 4 different covalently bonded atoms. The non-bonded terms, on the other hand, represent energies between atoms which do not share a covalent bond but interact due to the electrostatic potential.

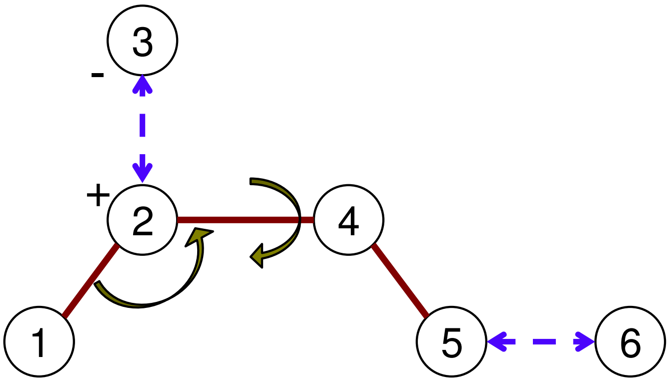


Figure 3·6: Various bonded and non-bonded interactions between atoms

The different terms of the total energy are as follows:

- **Electrostatic:** Electrostatic energy refers to the energy of an atom due to the interaction between its charge and the charges of the neighboring atoms. In other words, it is the energy of the atom due to the electrostatic field generated by other charged particles around it. In Figure 3·6, atoms 2 and 3 experience electrostatic energies due to the charges on each other.

- **van der Waals:** van der Waals energy represents the electrical interaction between charge-neutral particles (atoms 5 and 6 in Figure 3·6) and is caused due to the fluctuating polarizations leading to temporary charge dipoles.

- **Bond stretch:** This term represents the interaction between atom-pairs separated by one covalent bond and accounts for the energy due to the displacement from the ideal bond length [SDKF10] This is shown in Figure 3·6 between atoms 3 and 4.

- **Bond angle:** Bond angle term represents the energy between three covalently-bonded atoms and accounts for the change in the angle formed by them. It describes the angular vibrational motion between sets of three covalently-bonded atoms [SKLS$^+$10]. This is shown in Figure 3·6 as the angle between atoms 1, 2 and 4. Both the bond stretch and the bond angle terms are penalty functions and represent deviations from an ideal geometry. For a perfectly optimized structure, the sum of these two terms should be close to zero [SDKF10].

- **Bond torsion:** The torsion term models the presence of steric barriers between atoms that are separated by 3 covalent bonds [SDKF10] (atoms 1, 2, 4 and 5 in Figure 3·6). It represents the rotational motion around the middle bond, between the planes formed by the first three and the last three atoms [SKLS$^+$10].

- **Improper dihedral:** CHARMM force field has an additional bonded term called the improper dihedral. It is used to "maintain chirality and planarity" [SDKF10]; in other words, to restrict the geometry of the molecules.

Energy minimization involves the repeated evaluation of this expression, once during each minimization iteration. As stated earlier, moving to the next iteration requires moving the atoms in the direction of the least energy conformation. Thus at each iteration, the total force acting on each atom is also computed and the atoms are moved in the direction of those forces.

3.4 Methods for Evaluating Docking Systems

The overall goal of docking is to correctly predict the structure of the complex formed when two independent proteins interact. The effectiveness of docking systems is evaluated in terms of two criteria:

- Ability to discriminate the near-native structures from false positives [Rit08]: Complex structures generated by the docking systems are usually subjected to second level scoring algorithms. Many of the structures generated by the docking step are far from native solution and are eliminated in the scoring phase. The time spent on evaluating these false positive structures is, in some respect, loss of valuable computational resources. Better discrimination of near-native structures from false positives is, thus, a desirable feature of docking programs.

- RMS deviation of the docked complex from the experimentally determined native structure [JHM+03]: Prediction of complex structures close to the native structure is the most desirable quality of docking systems. The closer the prediction to the native structure, the better the docking results. The prediction accuracy can be measured in terms of the root mean square distance (RMSD) of the predicted structure from the experimentally determined complex structure.

There are, however, no specific ways of measuring these properties of the docking algorithms. Computing the RMSD requires that the experimentally determined structure of the complex be known. For most real-life docking, these structures are not known in advance and the aim of the docking process is to find that structure. One way of evaluating the effectiveness of the docking systems is through the use of docking benchmarks [CMJW03, MWP+05]. Docking benchmarks contain a list of undocked molecule pairs and their corresponding experimentally determined docked structure. One of the popular docking benchmarks is the Protein-Protein Docking Benchmark [MWP$^+$05]. It contains hundreds of protein-pairs and serves as a good resource for evaluating the docking systems. Docking systems, however, perform differently on different molecules. It has been shown that some docking systems perform exceptionally well on certain types of docking problems but are unable to predict the correct structures for other class of docking problems [VC04]. This makes evaluating the docking systems more difficult. A more widely accepted standard for evaluating the effectiveness of docking systems is the community wide experiment called the CAPRI challenge (Critical Assessment of PRedicted Interactions) [JHM$^+$03]. During this experiment, docking systems are presented with unpublished atomic coordinates of a variety of protein molecules and their predictions are compared against experimentally determined structure of the protein complex.

3.5 PIPER Rigid Docking Program

PIPER [KBCV06] is a state-of-the-art rigid docking program developed at the Structural Bioinformatics lab at Boston University [Vaj10]. Among the different FFT-based rigid docking programs, PIPER incorporates the most complex scoring functions; this improves its ability to discriminate the near-native conformations from the false positives. Due to this improved sensitivity, PIPER has consistently performed well in the CAPRI challenge. The use of the complex energy functions, however, leads to higher computational complexity and hence longer runtimes. Both these aspects of PIPER make it an ideal candidate for acceleration.

PIPER advances the art of rigid molecule docking by minimizing the number of candidates needing detailed scoring in the docking phases subsequent to rigid docking. A primary consideration in docking is preventing the loss of near-native solutions (false negatives); as a result, rigid molecule codes tend to retain a large number (thousands) of docked conformations for further analysis, even though only a few hundred will turn out to be true hits. “Improving these methods remains the key to the success of the entire procedure that starts with rigid body docking [KBCV06].” PIPER addresses this issue by augmenting commonly used scoring functions, such as shape and electrostatics, with a desolvation computed from pairwise potentials.

Even though there exist other rigid docking programs that employ other forms of desolvation energies, PIPER’s pairwise potential function results in better discrimination of near-native solutions. It has been shown that on complexes from the Protein-Protein Docking Benchmark [MWP$^+$05], PIPER results in up to 50% more near-native conformations compared to ZDOCK [CW02b], a rigid docking program that employs atomic contact potential (ACP) [ZVCD97] for modeling the desolvation energy [Rit08]. This improved sensitivity directly improves the performance of docking and discrimination programs that employ rigid-docking as the initial stage. For example, the first version of the multistage, protein-protein docking program ClusPro [CGVC04], that uses either DOT [EMRP95] or ZDOCK [CW02b] for initial rigid-docking phase, retains 20,000 or 2,000 conformations respectively for further evaluations. The second version of the ClusPro program, which uses PIPER for the rigid-docking phase, retains only 1000 conformations for further analysis, without the loss of any near-native conformation [Vaj10].

Pairwise potentials represent the interactions of atoms (or residues) on the interacting molecules. Different pairs of atoms have different values; these are empirically determined (and sometimes called knowledge based). For K atom types, there is a $K \times K$ interaction matrix; each column (or row) can be handled with a single correlation resulting in K forward and one reverse FFT. Since K is generally around 20 (and up to 160), and since the FFT dominates the computation, use of pairwise potentials could drastically increase run time. A fundamental innovation in PIPER is the finding that eigenvalue-eigenvector decomposition can substantially reduce this added complexity. In particular, “adequate accuracy can be achieved by restricting consideration to the eigenvectors corresponding to the P largest eigenvalues where $2 \leq P \leq 4$, and thus performing only 2 to 4 forward and one reverse FFT calculations”. In practice, however, up to 18 terms are sometimes used.

Like most other rigid docking program, PIPER performs FFT-based exhaustive 6-D search using 3-D grid representation of the shape and other properties of the interacting molecules. To exhaustively search the entire 6D space, PIPER performs tens of thousands of rotations of the ligand (small molecule) with respect to the receptor (larger molecule). For each rotation, PIPER's energy-like scoring function is computed to evaluate the goodness of fit between the receptor and the rotated ligand. It is defined on a grid and is expressed as the sum of P correlation functions for all possible translations α, β, γ of the rotated ligand relative to the receptor;

$$E(\alpha, \beta, \gamma) = \sum_{p} \sum_{i,j,k} R_p(i, j, k) L_p(i + \alpha, j + \beta, k + \gamma) \tag{3.2}$$

where $R_p(i, j, k)$ and $L_p(i + \alpha, j + \beta, k + \gamma)$ are the components of the correlation function defined on the receptor and the ligand grids, respectively.

For each rotation, PIPER computes the ligand energy function L_p on the grid and performs repeated FFT correlations to compute the scores for different energy functions. FFT correlation results in the scores for all the 3-axis translations for the current rotation. For each pose, the scores for different energy functions are combined to obtain the total score for the pose. Finally, a filtering step returns some number of top-scoring poses from different regions of the result-map.

PIPER Scoring Functions

The scoring function used in PIPER is based on three criteria: shape complementarity, electrostatic energy, and desolvation energy (through pairwise potentials). Each of these is expressed as a 3D correlation sum, and the total energy function is expressed as a weighted sum of these correlation scores:

$$E = E_{shape} + w_2 E_{elec} + w_3 E_{pair} \tag{3.3}$$

Shape complementarity refers to how well the two proteins fit geometrically (see Figure 3·7) and here is computed as a weighted sum of attractive and repulsive van der Waals (Pauli exclusion) terms, the latter accounting for atomic overlaps:

$$E_{shape} = E_{attr} + w_1 E_{rep} \tag{3.4}$$

Electrostatic interaction between the two proteins is represented in terms of a simplified Generalized Born (GB) equation [CW03]. The electrostatic energy is obtained as a correlation between the charge on the ligand grid and the potential field on the receptor grid.

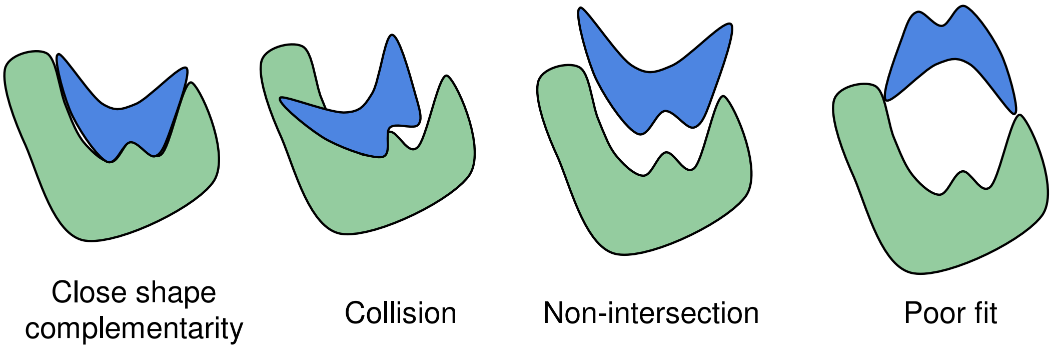


Figure 3·7: Shape complementarity between two molecules (based on [VGH04])

Desolvation is a measure of change in free energy when a protein-atom/water contact is replaced by a protein-atom/protein-atom contact. In PIPER, it is represented using pairwise interaction potentials, as previously discussed, through P correlation functions.

PIPER Program Profiling

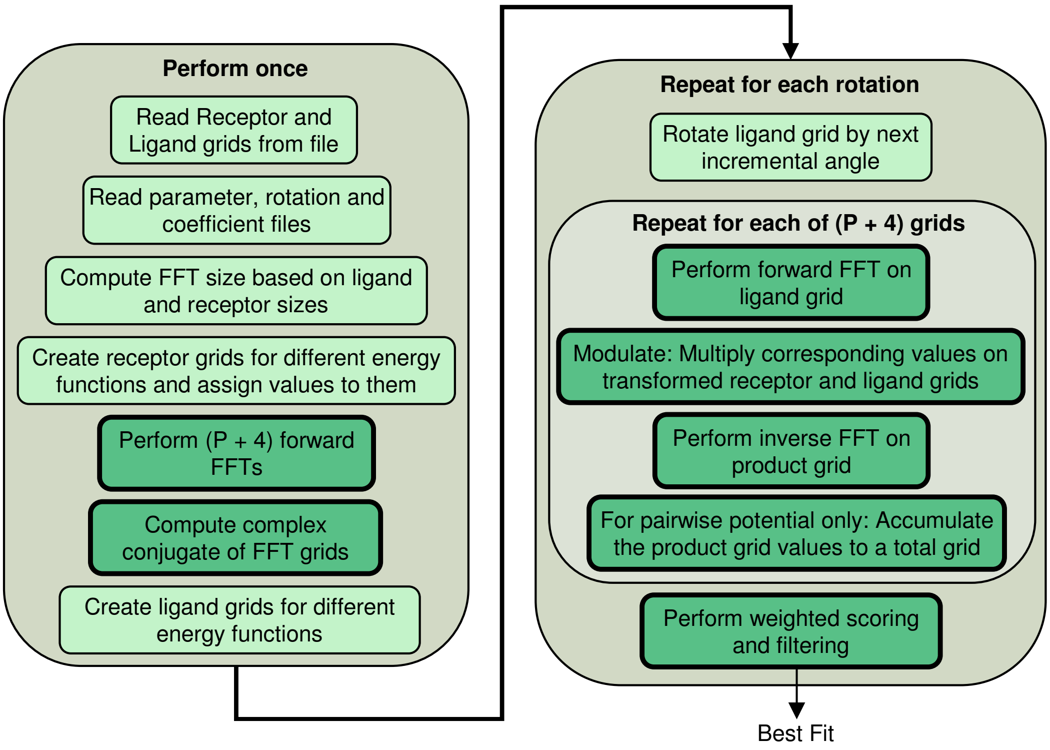

Perform once
Read Receptor and Ligand grids from file
Read parameter, rotation and coefficient files
Compute FFT size based on ligand and receptor sizes
Create receptor grids for different energy functions and assign values to them
Perform (P + 4) forward FFTs
Compute complex conjugate of FFT grids
Create ligand grids for different energy functions
Repeat for each rotation
Rotate ligand grid by next incremental angle
Repeat for each of (P + 4) grids
Perform forward FFT on ligand grid
Modulate: Multiply corresponding values on transformed receptor and ligand grids
Perform inverse FFT on product grid
For pairwise potential only: Accumulate the product grid values to a total grid
Perform weighted scoring and filtering
Best Fit

Figure 3·8: PIPER program flow

The computations performed by the PIPER program can be divided into one-time computations and per-rotation computations (see Figure 3·8). Since PIPER performs tens-of-thousands of rotations, with each rotation requiring up to 10 seconds, the one-time computation becomes a very small fraction of the overall runtime and can be ignored. The different per-rotation computations performed by PIPER can be listed as follows:

1. Rotation of the ligand grid
2. Assignment on the 3D grid
3. FFT correlation of the receptor and the ligand grids
4. Accumulation of pairwise potential terms to obtain the total desolvation score
5. Computing weighted scores of different energy functions
6. Filtering top scores for the current rotation

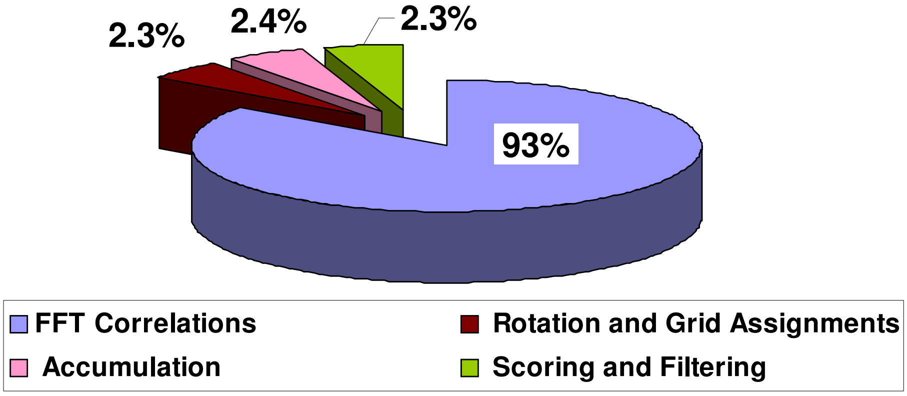


Figure 3·9: Runtimes per rotation for different steps of PIPER rigid docking

Figure 3·9 shows the distribution of the total per-rotation run time among these steps. As can be seen, most of the time is spent in computing the FFT correlations for different energy functions. This involves computing the forward FFT of the ligand grid, modulation of the two grids, and inverse FFT of the modulated grid.

In terms of the absolute runtime on a single processor, PIPER requires up to 10 seconds per rotation, resulting in total docking runtime of about 28 hours for a single molecule-pair. This is clearly prohibitively slow for any practical use. Production PIPER server runs on a large cluster with hundreds to thousands of processors.

In the current work, we propose the acceleration of the PIPER program using FPGAs and GPUs. Even though PIPER program affords immense coarse level parallelism (at the level of rotations), fine grained parallelism obtained from such acceleration can further improve the performance of the PIPER program and also provide a fast, desktop-based alternative to large clusters. Moreover, PIPER is a leading exhaustive rigid-docking docking program, with applications in other molecular modeling programs such as ClusPro [CKB^{+}07] and FTMap [BKC^{+}09]. Acceleration of PIPER, thus, has huge potential impact in the field of molecular modeling.

The current work accelerates all the steps depicted using dark green boxes (with bold outlines) in Figure 3·8. This includes some one-time steps as well as all the per-rotation steps, except for rotation of the ligand and generation of the rotated grid. These steps do not contribute significantly to the total per-rotation time. Moreover, these computations, which are performed on the host, are done in parallel with the computations on the FPGA or the GPU and thus their latencies are completely hidden.

3.6 Binding Site Mapping

3.6.1 Overview

Drug discovery is an expensive and time consuming process. Moreover, the chances of a candidate drug compound turning into an approved drug is one in 10,000 (0.01%) [Fou10]. “The well-documented high attrition rate of compounds in preclinical development and the high failure rate of investigational new drug applications in the clinic have motivated the reevaluation of high throughput screening (HTS) paradigm” [VG06]. The need for efficient and effective drug discovery methods is, thus, very pressing.

It has been observed that small organic fragments are able to characterize the "drug-like" effects more accurately than complex molecules. Due to this, the fragment-based drug-discovery methods are more effective [VG06]. Fragment based approaches dock a variety of small fragments in the protein binding site. The process starts with the identification of druggable binding sites on the protein surface [VG06]. Binding sites, or "hot spots", are regions on a protein surface that are major contributors to the total binding energy [LJY+07]. Furthermore, it has been widely understood that these sites bind a variety of small organic molecules; clustering of a number of such probes in a region on the protein can, thus, be a good indicator of druggability [VG06].

The ability to correctly identify druggable binding sites on a protein surface has been a long-sought goal in drug discovery. Experimental methods based on NMR and x-ray crystallography can identify the binding sites with high sensitivity [VG06]; these methods, however, are expensive. A variety of computational methods have been developed based on different approaches such as geometric, energy-based, and docking/mapping. The problem with the first two approaches, however, is that they identify various binding pockets without taking into account any measure of druggability [VG06]. As a result, they return a large number of potential binding sites on the protein surface, making it difficult to identify the relevant ones [MR96].

Docking/mapping based identification of the binding site is a relatively new and effective technique. Here, the binding site is identified by docking a set of small molecule probes on the protein surface. The consensus site where most of these probes cluster together is indicative of the potential binding site. This process is called mapping. Mapping is very effective in finding the active binding site on the protein surface. It has been shown that with at-least six different small molecule probes, the consensus site is always a major subsite of the active binding site [SKJK05].

3.6.2 Computational Mapping

Binding site mapping refers to the computational prediction of the regions on a protein surface that are likely to bind a small molecule with high affinity. Binding site mapping is a relatively new and effective technique used for performing drug discovery.

Discovering a new drug involves finding a site on a given protein which will bind a small molecule inhibitor with high affinity. Moreover, it also requires finding the appropriate small molecule inhibitor, or the ligand, that will bind to that site and alter the function of the protein, thus curing the disease. Drug discovery, thus, involves docking-based screening of millions of candidate ligands for a given protein. As discussed earlier, docking involves searching the entire protein surface for the binding site, often requiring exhaustive 3D search. This is a computationally demanding process, requiring many hours to days of CPU runtime. An important observation, however, is that certain regions on a protein surface, called "hot spots", are major contributors to the total binding energy between the protein and the ligand, and that they bind a wide variety of small molecule probes [BKC$^+$09, LJY$^+$07]. Thus, a hot spot on a protein surface can be found by docking some small molecule probes and finding a consensus region that binds most of these probes with high affinity. This process of finding the binding site using small probes is called binding site mapping or solvent mapping.

Binding site mapping benefits drug discovery since it finds the likely binding site on a protein surface using a standard set of small molecule probes, independent of the actual ligand. This significantly reduces the search space on the protein surface to be explored during the actual screening of the ligand database. During drug-screening, each ligand from the database can be docked on this local region or the hot spot region, without having to search the entire 3D space. This reduces the screening time significantly, enabling faster drug discovery. Though the identification of hot spots is also possible with experimental methods such as NMR or X-ray crystallography, such methods are very expensive and computational methods are explored as more cost-effective alternatives. Examples of computational mapping programs include CSMap [LJY+07], FTMap [BKC+09] and MCSS HOOK [EWKH94]. In the current work, we target the acceleration of the FTMap program, which is discussed in detail in the next section.

3.6.3 FTMap

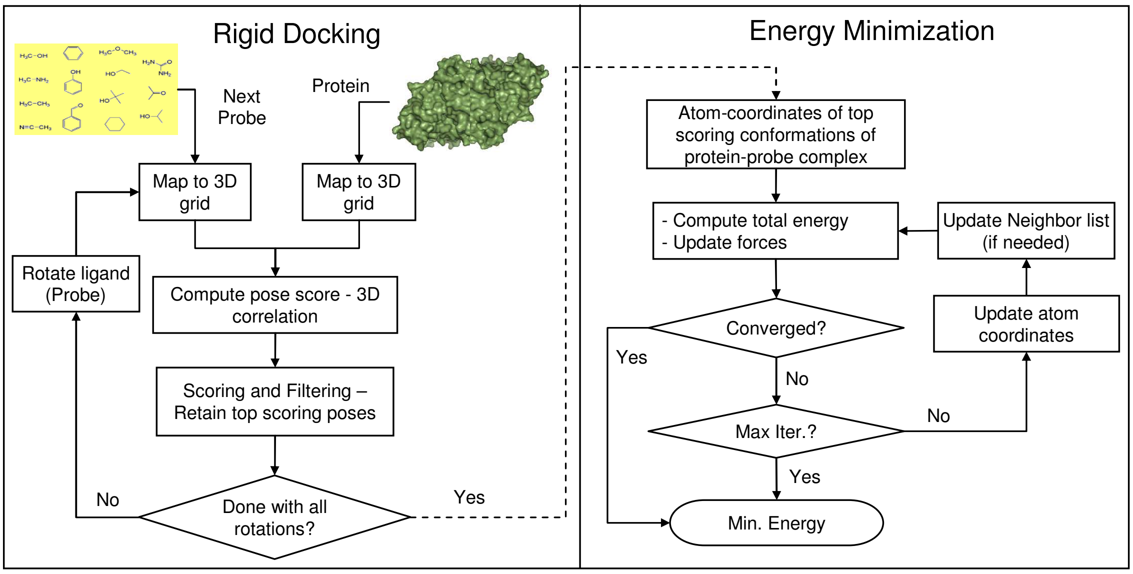


Figure 3·10: FTMap program flow

FTMap [BKC+09], or Fourier Transform based Mapping, is a program developed at the Structural Bioinformatics lab at Boston University [Vaj10]. As stated earlier, computational mapping of binding site involves flexibly docking a wide variety of small molecule probes to a given protein and finding the consensus region that binds most of these probes with high affinity. FTMap uses sixteen different small molecule probes for this task.

Due to the computational complexity of flexible docking, FTMap splits the mapping task into two steps. The first step performs rigid docking of the probes into the protein using the PIPER program. The second step accounts for the flexibility in the probe side chains by energy minimizing the top scoring protein-probe complexes returned by the first step. The overall flow of the FTMap program is shown in Figure 3·10.

Unlike the original PIPER program where tens of thousands of rotations of the ligand are evaluated, FTMap evaluates only 500 rotations of each probe with respect to the protein. For each rotation, it computes the pose score for all the possible translations using the FFT correlation scores for different energy functions. Four top scoring poses from each rotation are retained for energy minimization in the next step. FTMap, thus, performs energy minimization of 2000 top scoring poses between the protein and each probe. This results in 32,000 different protein-probe complexes to be energy-minimized. This makes the energy minimization step the most time consuming step of the FTMap program.

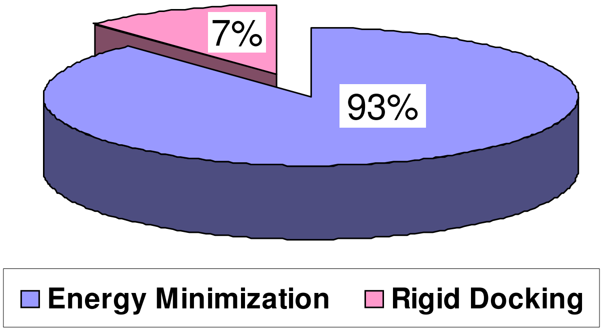


Figure 3·11: Profiling result of the FTMap program

Figure 3·11 shows the profiling results for the FTMap program. As shown, about 93% of the total FTMap runtime is spent in the energy minimization phase. In terms of the absolute runtime, energy-minimization of each protein-probe complex typically takes around 12 seconds on a single CPU, resulting in close to 7 hours for minimizing the 2000 complexes from each probe. With 16 probes to be mapped, the total FTMap runtime can be many days. The computations in FTMap, however, are highly parallel, at least at the coarse level. Different probes can be evaluated in parallel, independent of each other. Further, energy minimization of all the protein-probe complexes can also be performed in parallel.

Currently, FTMap runs on a 1024 node IBM BlueGene cluster, requiring only a few minutes to complete the computations. In the current work, we propose a desktop alternative to the supercomputer cluster, by accelerating the energy minimization step of FTMap on FPGAs and GPUs. This is done by parallelizing the computations at a fine-grained level. The coarse-level parallelism can still be applied to further improve the performance using a cluster of FPGAs or GPUs.

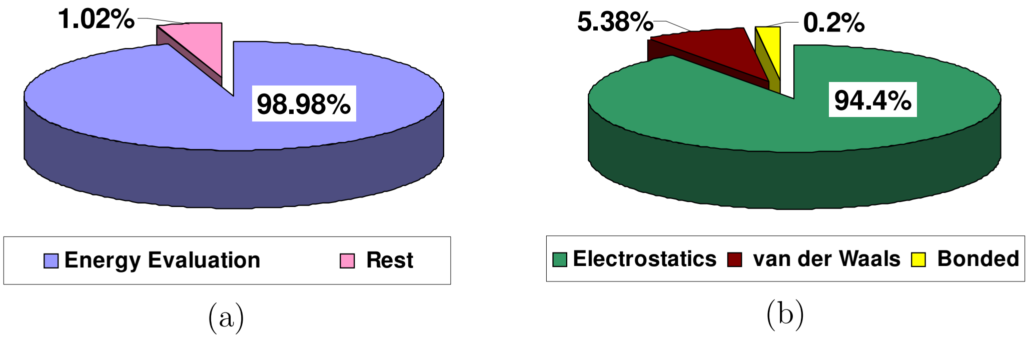
3.6.4 FTMap Energy Minimization Step

The free energy expression evaluated by FTMap during energy minimization is similar to Equation 3.1. FTMap implements CHARMM potential with electrostatics and solvation terms as implemented in CHARMM 27 [BBO$^+$83], with parameter set from CHARMM version 19 [BKC$^+$09].

FTMap performs minimization using the L-BFGS quasi-Newton optimization technique [LN89]. During the minimization process, the probe molecules are free to move, with the protein atoms held fixed [BKC$^+$09].

The runtime profiling for the energy minimization step of the FTMap program is shown in Figure 3·12(a). As shown, evaluation of different energy and force terms contributes towards almost 99% of the total time spent in the energy minimization step. Of the total time for energy evaluation, more than 94% is spent evaluating the electrostatics and another 5% in computing the van der Waals energy (Figure

3·12(b). Evaluation of the bonded energy terms constitutes only about 0.2% of the energy evalaution runtime.



(a) (b)

Figure 3·12: Profiling of the energy minimization step of serial FTMap program, (b) Distribution of the energy evaluation time among the computation of different energy terms.

FTMap Electrostatics and van der Waals Energy Models

The electrostatic energy of a solute with N charges can be decomposed into two components. The first component is a sum of N self-energy terms, each proportional to the square of the corresponding charge value. The self energy of a charge represents the electrostatic potential energy at the point where the charge is located, due to the charge itself. This effectively represents the energy required to assemble the charge. For a point charge, this energy is infinite [Ers70, Fit10]. Computing the self-energy, thus, requires that the charge be represented as a distributed charge. This is often done by distributing the charge uniformly over a small sphere [SK96, Bor20]. Note that the self energy of a charge depends only on its location and is independent of the other charges surrounding it. It does, however, depend on the solute geometry, or the solvent-inaccessible volume. Solvent has the effect of screening the interaction between charge pairs. Solvent inaccessible volume is, thus, defined as the volume that is inaccessible to the solvent and hence contributes to the atom self-energy [SK96]

The second component of the solute electrostatic energy is the sum of the $N(N-1)/2$ pairwise interaction terms, each proportional to the product of the two charges involved in the pair. Both the self-energy and the pairwise interaction energy terms depend on the geometry of the solute [SK96]. The total electrostatic energy of a solute is thus given as a sum of all the self-energies, E_i^{self} , and the pairwise interaction energies, E_{ij}^{int} [SK96] (see Equation 3.5).

$$E^{elec} = \sum_i E_i^{self} + \sum_{i<j} E_{ij}^{int} \tag{3.5}$$

For the computation of the electrostatic energy, FTMap employs the Analytic Continuum Electrostatics (ACE) model [SK96]. In the ACE model, the self-energy of an atom is represented as a sum of its Born self-energy in the solvent dielectric ϵ_s plus the sum of effective pairwise self-energy potentials, E_{ik}^{self}, due to all the other solute atoms (see Equation 3.6) [SK96].

$$E_i^{self} = \frac{q_i^2}{2\epsilon_s R_i} + \sum_{k \neq i} E_{ik}^{self} \tag{3.6}$$

The Born self energy represents the contribution to the particle's energy due to the field created by the charge of that atom itself. The pairwise self-energy E_{ik}^{self} of an atom i due to an atom k is the energy required to replace the solvent dielectric with the solute dielectric within the volume of atom k, in the presence of the electric field generated only by the charge of atom i [SK96].

To compute the pairwise portion of the self energy, ACE defines atom charges as Gaussian distributions. E_{ik}^{self} can then be computed by integrating the energy density of the electric field. For efficient computation, this is approximated as the sum of a short-range term, that approximates the Gaussian, and a long range term (first and second terms in Equation 3.7 respectively).

$$E_{ik}^{self} = \frac{\tau q_i^2}{\omega_{ik}} e^{-\left(\frac{r_{ik}^2}{\sigma_{ik}^2}\right)} + \frac{\tau q_i^2 \tilde{V}_k}{8\pi} \left(\frac{r_{ik}^3}{r_{ik}^4 + \mu_{ik}^4}\right)^4 \tag{3.7}$$

Here q_i represents the charge on atom i, τ is the difference in the dielectric constants of the solute and the solvent, r_{ik} is the distance between atoms i and k, $\tilde{V}_k$ is the size of the solute volume associated with atom k, ω_{ik} and σ_{ik} determine the height and width of the Gaussian that approximates E_{ik}^{self}, and μ_{ik} is an atom-atom parameter.

The pairwise interaction component of the total electrostatic energy (second term in Equation 3.5) is given by the generalized Born (GB) equation, which is the sum of Coulomb's law in a dielectric and the Born equation [STHH90]

$$E_{ij}^{int} = 332 \sum_{j \neq i} \frac{q_i q_j}{r_{ij}} - 166\tau \sum_{j \neq i} \frac{q_i q_j}{\sqrt{r_{ij}^2 + \alpha_i \alpha_j e^{-\left(\frac{r_{ij}^2}{4\alpha_i \alpha_j}\right)}}} \tag{3.8}$$

where α_i and α_j represent the Born radius for atoms i and j, respectively. These in turn depend on the total self-energy of the atom given by Equation 3.6.

The van der Waals energy of an atom is most commonly represented using the Lennard-Jones 6-12 potential (LJ). It is a potential that describes the interaction between two uncharged molecules or atoms. It is a function of the distance between the two interacting molecules and falls sharply as the distance increases. Due to this, most algorithms implement a cut-off based potential wherein the interaction energy is considered to be zero if the distance between the particles is larger than a cutoff. In its most general form, the LJ potential is given as

$$E_{vdw} = \sum_{non-bonded-pairs} \left(\frac{A_{ik}}{r_{ik}^{12}} - \frac{C_{ik}}{r_{ik}^6}\right) \tag{3.9}$$

where r_{ik} is the distance between the two atoms under consideration and A_{ik} and

C_{ik} are atom-atom constants.

FTMap computes the van der Waals energies of the atoms using a variant of the Lennard-Jones potential. The expression evaluated by FTMap for an atom-pair is as follows:

$$E_{vdw_{ik}} = eps_{ik} \left(\frac{rm_{ik}^6}{r_{ik}^{12}} - \frac{8rm_{ik}^6/r_c^6}{r_{ik}^6} + \frac{rm_{ik}^6}{r_c^{12}} \left[1 + \frac{2r_{ik}^6}{r_c^6} \right] \right) \tag{3.10}$$

where

$$eps_{ik} = eps_i.eps_k \tag{3.11}$$

and

$$rm_{ik} = (rm_i + rm_k)^2 \tag{3.12}$$

and eps_i and rm_i represent the van der Waals parameters of atom i, r_{ik} is the distance between atoms i and k and r_c is the cutoff distance.

Equations 3.7, 3.8 and 3.10 represent the main computations that need to be performed for all atom-atom pairs to evaluate the total electrostatic and van der Waals energies of a given conformation. In addition, the forces acting on the atoms are computed by computing the gradients of these energies with respect to the atom coordinates.

Performing all-to-all computations among all the atom requires $O(N^2)$ computations and is computationally very expensive. Moreover, atom-atom interactions fall sharply with increasing distance and can be ignored for atoms farther than a cut-off distance. FTMap, thus, computes the electrostatics and van der Waals interactions only among those atoms which are within a certain cutoff distance from each other.

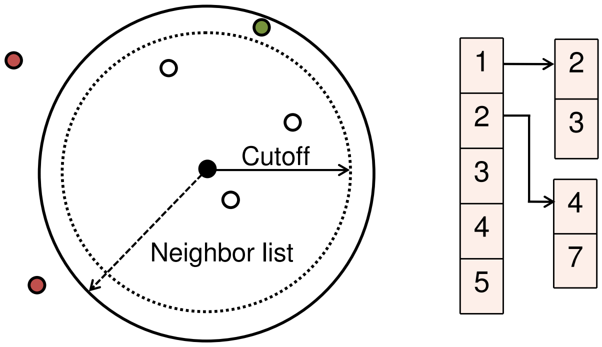


Figure 3·13: Neighbor lists

For efficient computation, atoms are arranged as neighbor-lists wherein each atom (called the first atom) has a list of atoms (called the second atoms) that are within a cut-off distance (see Figure 3·13). FTMap program cycles through the neighbor list of each first atom and computes its partial energy due to the second atoms in the list. Using symmetry, it also computes the partial energy of each second atom due to the current first atom.

3.6.5 Energy Minimization vs. Molecular Dynamics

Though the underlying computations in energy minimization seem superficially similar to those in molecular dynamics (MD), energy minimization differs from MD simulations in various ways. These distinctions can be classified into two categories – the geometry of the problem and the way the computation is performed.

- **Problem geometry:** From the point of view of the problem geometry, there are three differences. First, unlike MD where the system typically consists of millions of particles, energy minimization is often performed on a local region of the complex, resulting in only a few thousand atoms being simulated, and requiring only up to a few tens of thousands of atom-pair evaluations per iteration. Due to this, the computation per iteration is very small. Second, the neighborhood associated with each atom is much smaller compared to MD.

Finally, since minimization is a refinement step, simply to model the flexibility in atom side chains, the motions are small.

- **Computation structure:** With respect to the computations and the data structures, energy minimization differs from MD in four ways. First, unlike molecular dynamics where the movement of the atoms is based on Newtonian dynamic laws and produces a trajectory based on kinetic energy, minimization simply adjusts the atom coordinates so as to lower the total energy of the system [BBO$^+$83]. Forces acting on the atoms are computed and atoms are moved in the direction of the force using some optimization move. Second, even though the energy terms computed in minimization are similar to those in molecular dynamics, the actual energy expressions evaluated during minimization are often quite different. Third, unlike MD where cell-lists are heavily utilized to organize the computations, minimization routines often do not employ cell-lists at all. Finally, even though energy minimization, like MD, uses neighbor lists, the neighbor lists used in energy minimization are very differently populated compared to MD. In MD, the neighbor lists are uniformly and very heavily populated, with the neighbor list of each atom containing tens of thousands of atoms. The neighbor lists in energy minimization, however are very sparsely populated and very non-uniform, with most of the atoms containing only a few atoms in their neighbor lists while some containing a few thousand atoms. Moreover, unlike MD where the neighbor lists are updated very often, the neighbor lists in energy minimization are seldom updated.

Due to these differences, the acceleration of energy minimization is a very different problem than that of MD. Techniques such as efficient particle-filtering that are applied to accelerate the MD simulations do not benefit energy minimization computations. Additionally, the small computation per iteration makes the overall performance very sensitive to various data-movement and control overheads. Minimization of such overhead is, thus, central to obtaining performance improvements. Finally, a significant portion of the per-iteration computation is serial accumulation, that further limits the available parallelism. On FPGAs, we address these issues by utilizing stream-computing and deep pipelines; on GPUs, we change the underlying data structures for better mapping and increased parallelism.

3.7 Analysis and Summary

Molecular docking and binding site mapping are important tools for molecular modeling. They find widespread application in the field of protein structure prediction and drug discovery. Due to the computational limitations, however, accurate modeling of the chemical interactions is not generally possible. Most docking and mapping programs thus make various simplifying assumptions such as treating the molecules as rigid or neglecting certain interactions.

Of the various docking schemes, FFT-based rigid docking is arguably the most time consuming. With complex scoring functions requiring the evaluations of multiple 3D FFTs, the rigid-docking runtime can be close to 30 hours on a serial processor.

Moreover, rigid docking is at the heart of many molecular modeling applications. For protein-protein docking, rigid-docking is almost always performed as the first step. All the protein-docking programs that performed well in the community-wide CAPRI challenge [JHM+03] use rigid docking as the first stage [VC04], followed by further refinements. As an example, ClusPro [CGVC04], the protein-protein docking program that was the top performer in the Fourth CAPRI Evaluation Meeting challenge in 2009, performs rigid docking, followed by RMSD-based clustering and complex scoring. In addition, many flexible docking algorithms also perform rigid-docking at the inner loop. For example, fragment-assembly, an algorithm for modeling ligand flexibility, divides the ligand into small fragments, which are then individually docked using rigid-docking. Another molecular modeling application that uses rigid-docking is solvent mapping. The process involves flexibly docking a set of small molecule probes into a given protein. Some mapping programs achieve this by rigidly docking the probes into the protein, followed by energy minimizing the top scoring conformations.

Clearly, rigid-docking is a computationally intensive process that can benefit from acceleration. Further, its acceleration will also benefit a suite of other molecular modeling applications that are central to drug discovery and life sciences.

In addition to the rigid-docking, the acceleration of flexible docking is also critical. Accurate modeling of the interactions between the molecules requires modeling the flexibilities in (at least) the small molecule. A common technique to model these flexibilities is to energy-minimize the top scoring conformations returned by the rigid-docking phase, allowing the molecule side chains to move freely. Many docking programs either implement an energy-minimization routine of their own or call the CHARMM minimization routines [BBO$^+$83].

Among the other flexible docking programs, those based on systematic search, e.g. fragment-assembly, perform repeated rigid-docking and can benefit from the accelerated rigid-docking systems. Systematic search algorithms based on geometric hashing and clique detection are already very fast in the docking phase; most of their runtime is spent in the refinement steps such as energy minimization. The acceleration of energy minimization computations would, thus, also benefit these docking schemes.

Stochastic flexible docking schemes, however, are difficult to parallelize due to the small computation in each step. Moreover, achieving overall performance speedup on such systems requires moving almost all the computations to the accelerator side, which is often a difficult task.

In the current study, we target the acceleration of the computations in rigid docking and energy minimization. These accelerated routines can be applied to a variety of docking and mapping systems, either individually or in combination, to improve the overall system performance.

Chapter 4

Acceleration of Rigid Molecule Docking

4.1 Overview

We now present the FPGA and GPU algorithms for the acceleration of the rigid-docking program PIPER. As previously discussed, there exists a divergence in the protein-protein and protein-ligand docking algorithms, with docking programs sometimes specializing in one of these classes of docking. Our studies in the acceleration of rigid-docking indicate that a similar divergence exists in accelerated docking as well. Certain computation models suit well to the acceleration of protein-ligand docking but are not preferred for docking large molecules. PIPER docking program is applied to both protein-protein and protein-ligand docking. On accelerated PIPER, however, based on the molecule sizes, different acceleration schemes are preferred.

The main computation in PIPER rigid-docking is the evaluation of 3D FFT correlations for various energy functions and the weighted accumulation of the scores from these correlations. In the acceleration of these computations, we find that, based on the size of the interacting molecules, different optimizations can be performed. The first among these is replacing the multiple serial FFT correlations with parallel direct correlations on FPGAs and GPUs. Our studies indicate that compared to the FFTs, direct correlation is more amenable to efficient parallel implementation, both on the FPGAs and the GPUs. Secondly, the small hardware resource requirements for ligand docking allows us to perform multiple docking rotations simultaneously, resulting in improved performance. As a result, direct correlation based methods perform superior than FFT for protein-ligand docking.

The above optimizations, however, are attractive only for small molecule sizes. As the molecule size increases, the benefit of direct correlation starts to fade and the FFT-based method regains its dominance. Accelerated FFT routines on the GPUs thus provide more effective docking solutions for protein-protein docking.

4.2 Algorithms for FPGA Acceleration of Rigid Docking

As discussed in chapter 3, the basic computation involved in PIPER rigid-docking is 3D FFT correlation for different energy functions. The application of FFT correlation to docking revolutionized the field since it reduces the computational complexity from $O(N^6)$ to $O(N^3 logN)$, making it practical to exhaustively search the entire 6D space. FFTs, however, require complex double (or single) precision computations. Even if the input data has a very small range, thus requiring only a few bits for representation, computation of FFT expands this to 106 (or 48) bits precision. This is due to the multiplications with the complex exponential constants (the twiddle factors). Clearly, this is not amicable for FPGA acceleration, where one of the fundamental methods is to exploit the reduced precision to limit the resource requirements and duplicate the processing elements to maximize parallelism.

Efficient hardware structures for 1D and 2D direct correlation exist that can perform better than FFT on the FPGAs, especially if the data format requires few bits of precision. Performing direct correlation does not require the data format to be expanded to higher precisions since no frequency-domain transformation is done. Correlation simply performs element-to-element multiplications and accumulations. Such structures have been applied in the acceleration of signal and image processing applications using FPGAs. Previous work from members of our lab [VGH04, VGMH06] has shown that these structures can be extended to perform efficient 3D direct correlation on FPGAs that outperforms FFT correlation on host. They further showed that such algorithms, when applied to simple shape complementarity based rigid docking, can result in significant performance improvements.

In the current work, we extend the 3D direct correlation structures on FPGAs to perform multiple correlations as required of the PIPER docking program. We also find the limits of the correlation-based approach for docking on current generation FPGAs and GPUs. Since the FFT has the advantage over direct correlation in asymptotic complexity, it is natural to expect that as the problem size and the data precision increases, FFT will ultimately outperform direct correlation; the question is at what molecule size does this advantage begin to dominate over other factors, such as parallelism and precision.

In the current work, we also added support for the following: (i) pairs for large molecules as necessary for modeling protein-protein interactions; (ii) the efficient combining of a potentially large number of force models, (iii) handling charge reassignment after every rotation, (iv) changing the force models using a script for generating hardware modules for different energy functions.

Before we present these designs, we present a brief overview of the hardware structure for performing a single direct 3D correlation. In the subsequent sections, we show how this structure is extended to support multiple correlations and combining of scores for PIPER energy functions.

4.2.1 Systolic Array for 3D Correlation

Figure 4·1 shows the systolic 3D correlation array progressively formed starting from a 1D correlation array [VGMH06]. This structure is an extension of the McWhirther-McCanny correlation array described in [Swa87]. The systolic array performs direct correlation at streaming rate, generating one correlation score per cycle. The basic unit of the systolic array is a compute cell which takes two voxels and computes the voxel-voxel score. The compute cell then adds this score to the partial score from the cell on its left and outputs the updated partial score to the cell on its right. The operation of the compute cell can be written as

$$Score_{out} = Score_{in} + F(Voxel_L, Voxel_R) \tag{4.1}$$

where $Score_{in}$ is the score from the compute cell on the left and $F(Voxel_A, Voxel_B)$ is the function between the two voxels. For correlation, $F(Voxel_A, Voxel_B)$ translates to a product between the two voxels, making the compute cell a simple multiply-accumulate unit (Figure 4·1(a)). Voxels for the ligand grid are stored in the compute cells on the FPGA and the voxels of the receptor grid are streamed through it, generating one correlation score per cycle.

Figure 4·1(a) shows the systolic 1D correlation array consisting of pipelined compute cells. Computing the correlation scores between two arrays consists of three phases: loading, correlation and flushing. During the loading phase, the compute cells are loaded with the elements of one of the arrays by propagating them through the systolic array. This takes N_x cycles, where N_x is the length of the array that is being held in the compute cells. During the correlation phase, every clock cycle two things happen: one element of the other array is broadcast to every cell, and the partial scores computed by each cell are passed to the next cell. The total row score for the current cycle is generated by the last compute cell. The computation phase continues for M_x cycles and generates M_x correlation scores, M_x being the length of the array that is being streamed over the systolic array. Finally, the remaining $(N_x - 1)$ correlation scores are generated during the flushing phase, wherein zeros are broadcast to the compute cells for $(N_x - 1)$ cycles.

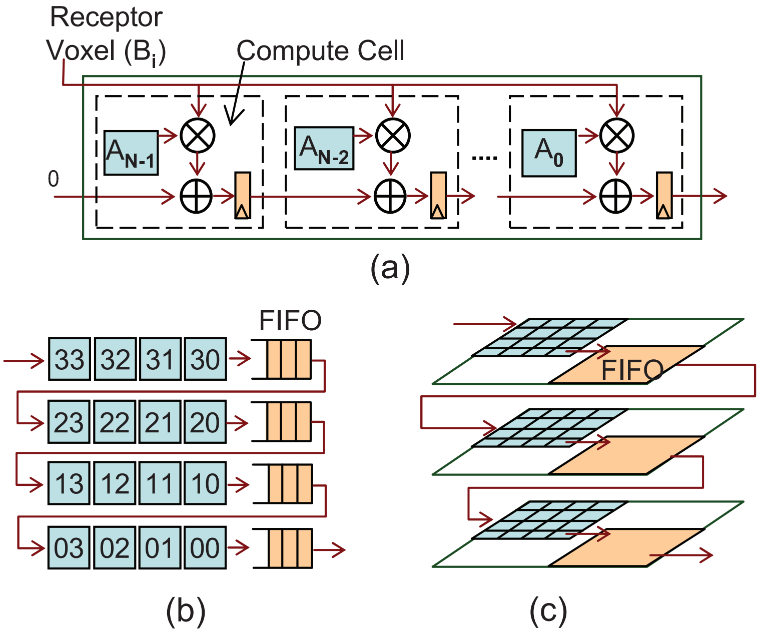


Figure 4·1: Structures to compute 3D correlations: (a) standard 1D systolic array, (b) extension to 2D correlation with delay lines, and (c) full 3D correlation with delay "planes".

Multiple 1D correlation arrays of Figure 4·1(a) can be connected to form a 2D correlation plane. The number of scores generated by each 1D correlation is $(N_x + M_x - 1)$, where N_x and M_x are the sizes of the two grids along the x-axis. To form a 2D correlation plane, the scores from different 1D rows need to be aligned. This is done by delaying each row score by $(N_x + M_x - 1)$ cycles before feeding it to the next row. A delay of N_x is inherently provided by the compute cells. To delay the scores by the remaining $(M_x - 1)$ cycles, 1D line FIFOs are used. Similarly, connecting multiple 2D correlation planes to form a 3D space requires plane FIFOs of size $(M_x + N_x - 1) \times (N_y - 1)$.

As is apparent from Figure 4·1, the number of compute cells in the systolic convolver is equal to the size of the 3D molecule grid that is being held in it. Since the compute cells require logic slices and multipliers, the size of one of the molecule (the smaller one) is limited by the amount of computation resources available on the FPGA. The larger molecule, however, is streamed in and its size is not affected by the size of the systolic array. There does exist a factor limiting its size, though. Note that the size of the FIFO is proportional to the size of the larger grid M_x. In addition, since the FIFOs are used to delay the correlation score, the width of the FIFOs depends on the number of bits the correlation score requires. The size of the larger molecule is, thus, limited by the amount of FIFO delay elements available on the FPGA. On typical high-end FPGAs, these FIFOs can be implemented using block RAMs. Although enough block RAMs are present to implement FIFOs for grids of quite large size, incorporating multiple correlations can pose a problem. This is discussed in the next section and a modified correlation pipeline is proposed.

4.2.2 Supporting PIPER Energy Functions: Overview

There are two obvious ways to extend the structure described in Section 4.2.1 to combine the multiple correlations required of PIPER: compute them singly or together. Neither is by itself preferred. The first method uses the same control structure as before, but for each different correlation, the FPGA is reconfigured to the appropriate data types and energy model. The scores must be saved off-chip and combined. That is, the k FFTs are replaced with k independent correlations, plus the overhead of reconfiguration and combining. The second method involves expanding the structure to perform k different correlations simultaneously. This method requires only a single pass through the large grid, and generates k independent correlation scores per cycle. Recall from Section 3.5, however, that the energy functions are weighted so that for k functions, the total score is given as

$$Score_{out} = Score_{in} + \sum_{i=1}^{k} w_i \times correlation\_score_i \tag{4.2}$$

Thus combining on-the-fly requires multiplications as well as additions, resulting in (perhaps) a substantially more complex compute structure. Combining can be done in three ways: within each compute cell, upon completion, or by integrating the weights into the scoring functions (see Figure 4·2).

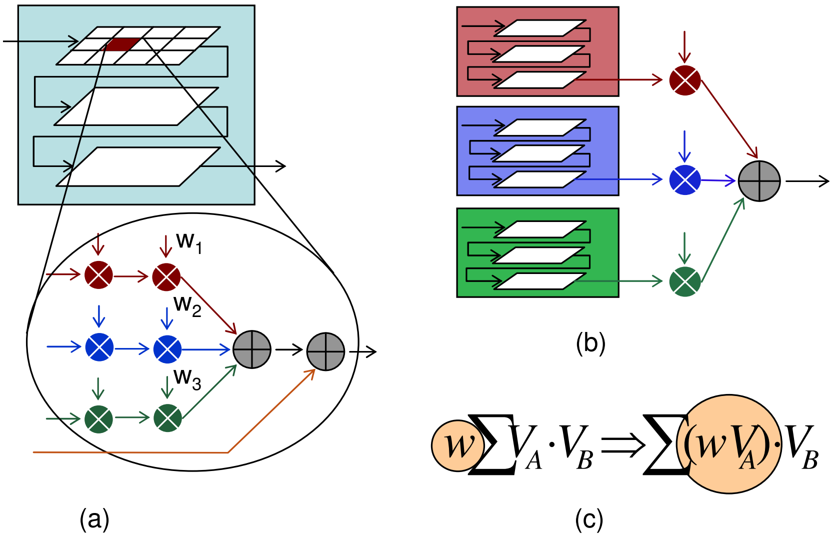


Figure 4·2: Three methods of combining multiple correlations in a single pass: (a) within a compute cell, (b) upon completion, or (c) integrated into the scoring function.

Combining within the compute cells requires that the weighted sums be computed within each one. This makes the compute cell more complex (see Figure 4·2(a)), with multiple multiplication operations and potentially a need to be pipelined. For each energy function, the first multiplier multiplies the two voxels to generate the score, which is then multiplied with the appropriate weight. Weighted scores of different energy functions are summed up and added to the weighted score from the previous cell. The problem here is the number of multipliers that this requires: $2k$ times the number of cells, or between 512 and 4000 additional multipliers. This is problematic for current FPGAs and would end up drastically reducing the number of compute cells that can fit on the FPGA and hence the size of the largest ligand grid that can be supported.

Combining on completion (see Figure 4·2(b)) means that we must propagate k independent running scores through the line and plane FIFOs of Figure 4·1; the width of the FIFOs must then be increased by $k\times$. Even with average sized grids, the block RAM requirements to implement the FIFOs are way over the available block RAMs on most FPGAs, making this approach impractical.

Integrating the weights into the grids (see Figure 4·2(c)) requires significantly increasing the precision throughout the entire system. This reduces the number of compute cells and thus the throughput. While a plausible solution, it is still not preferred.

4.2.3 Supporting PIPER Energy Functions: Augmented structure

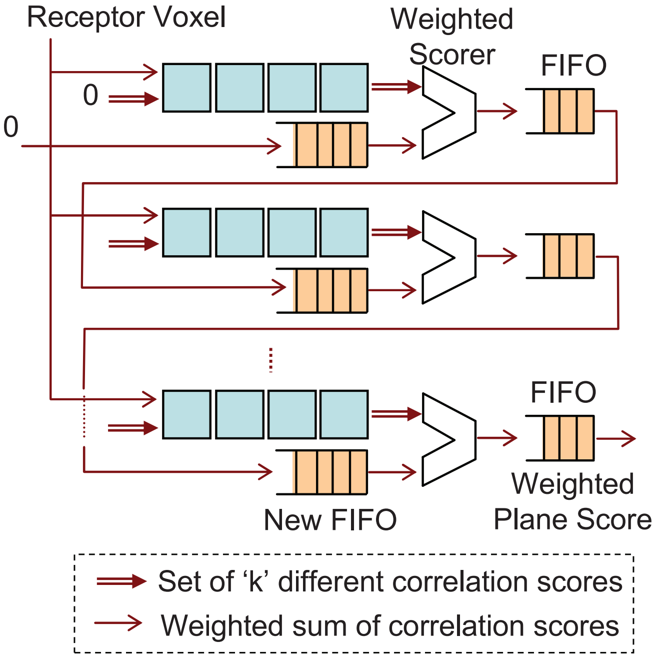

Receptor Voxel
Weighted Scorer
FIFO
0
0
FIFO
New FIFO
Weighted Plane Score
Set of ‘k’ different correlation scores
Weighted sum of correlation scores

Figure 4·3: 2D correlation pipeline modified to support complex correlations. The 3D extension is analogous.

To support multiple correlations for PIPER energy functions, we use a solution that is a hybrid of the options discussed in the previous section: we compute the correlations for all the energy functions simultaneously and combine the running scores once per row (see Figure 4·3). The resulting structure results in almost 40% savings in block RAM requirements compared to the solution in Figure 4·2(b). Figure 4·4 shows the block RAM requirements for different energy functions as per the structure of Figure 4·2(b). The right-most bar shows how this requirement is reduced by using the hybrid structure. The chart shown is for a ligand of size 8^3 and a receptor of size 60^3.

The proposed structure also results in almost 38% reduction in multipliers compared to the solution in Figure 4·2(a), for typical receptor-ligand grid sizes. Overall, it presents a balanced utilization of both the FPGA block-RAMs and the block-multipliers and thus enables supporting comparatively larger ligand as well as receptor grids.

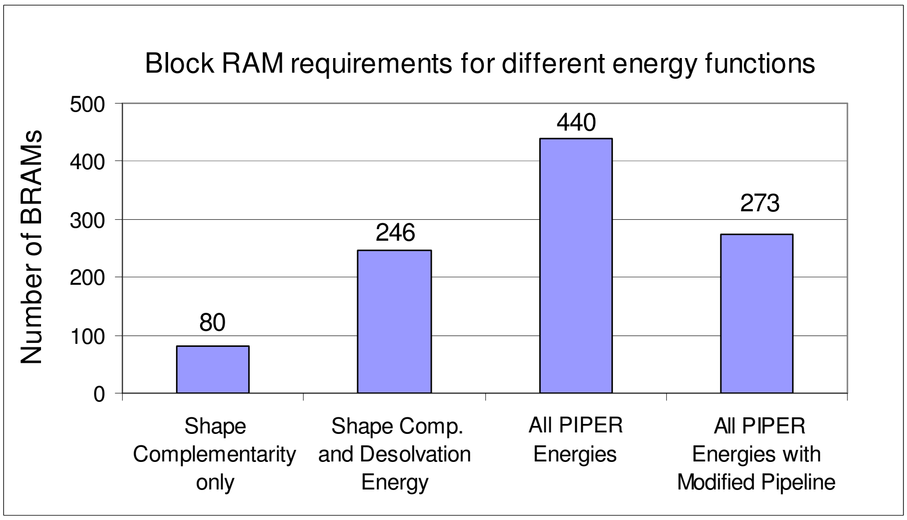


Figure 4·4: Block RAM requirement increases with the increase in the complexity of the scoring function; the modified pipeline reduces this requirement by almost 40%.

To obtain the new pipeline, we modify the correlation structure of Figure 4·1 in the following ways:

- **Modified compute cell.** The basic compute cell is extended to compute multiple correlation functions per cycle. Each cell in the modified systolic array performs k independent multiply-accumulate operations and outputs k independent partial correlation scores (see Figure 4·5).

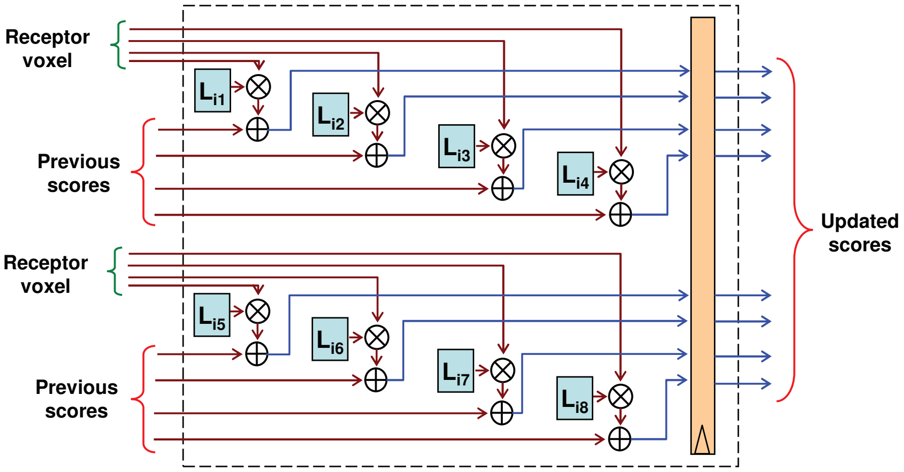


Figure 4·5: Internals of a correlation compute cell for PIPER energy functions.

- **Weighted Scorer.** At the end of each 1D correlation row, a new weighed scorer module is added. It takes k independent partial correlation scores (generated by the current 1D correlation row) and the partial weighted score from the previous row and computes a new partial weighted score. It also checks for saturation of individual scores, setting them to the positive or negative saturation value if needed. The partial weighted score is then sent to the line FIFO. Note that the FIFO now carries only one score as opposed to k. To obtain a high operating frequency, the computation is pipelined into 3 stages, as shown in Figure 4·6. The multiplexors in the first stage are used to select between the actual score or one of the saturation value.

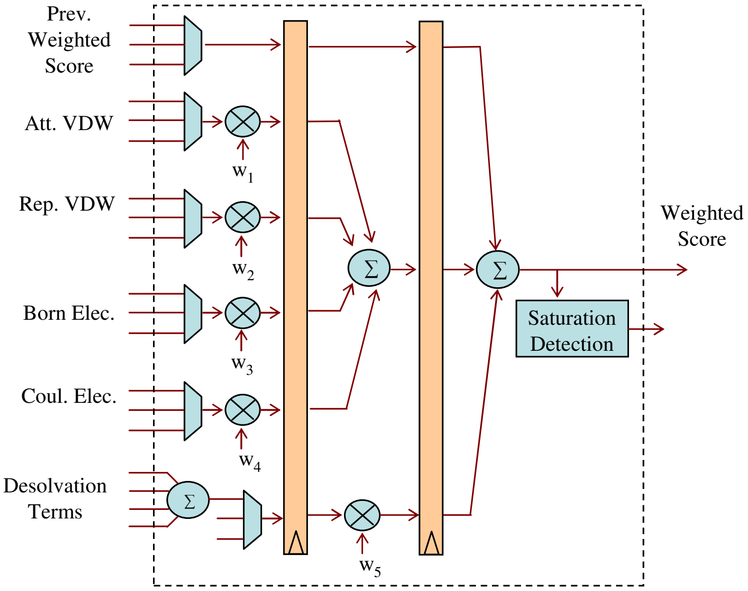


Figure 4·6: Details of the end-of-row score combiner. It is pipelined to enable high operating frequency.

- **New FIFO.** The scores propagating through the FIFOs are now weighted sums of individual correlation scores. The scores computed by the compute nodes are still the k individual correlation sums. Thus, compute cells cannot simply add the output of the FIFO to their current score. This distinction requires a modification in the existing pipeline and the compute cells. The compute rows no longer receive the partial correlation score from the FIFO; instead, a zero is fed into the first cell of each compute row. The weighted score from the FIFO of the current row is sent directly to the weighted scorer module at the end of the next row, where it is added to the partial weighted score of that row (as per Equation 4.2). In order to align this previous weighted score with the scores emerging from the current row, it must be sent through a new FIFO before it enters the weighted scorer. The length of this new FIFO is equal to the length of the 1D correlation row. For efficient implementation, this new FIFO is merged with the existing line FIFO. Also, the length of the combined FIFO needs to be adjusted to account for the delay through the pipeline stages of the weighted scorer.

- **New voxel data type.** In contrast to the earlier design, where each voxel represented only one value, the new voxel data at every grid point must represent energy values for different energy functions. To implement the PIPER energy functions, the voxel data is modified to contain the following five (or more) energy values: attractive van der Waals, repulsive van der Waals, Born electrostatics, Coulomb electrostatics, and P pairwise-potentials. The number of desolvation terms, P, is an input parameter.

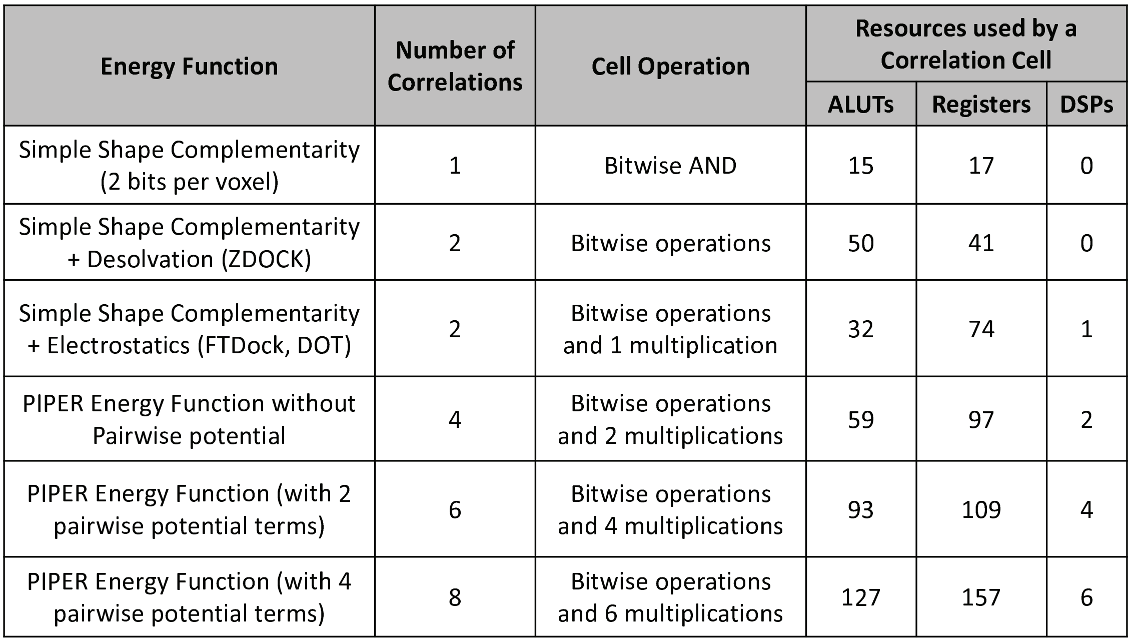
In the serial PIPER code, the voxels are represented using single precision floating point numbers. In our FPGA implementation, we use the fixed point numbers shown in Table 4.1 with no loss of precision.

Table 4.1: Number of bits per voxel for computing various energy functions.

Energy Term	Data Type	Number of bits	
		Receptor	Ligand
Attractive v. d. Waals	Integer	8	4
Repulsive v. d. Waals	Integer	4	4
Electrostatics	Fixed Point	9	9
Pairwise potentials	Fixed Point	9	9

4.2.4 Energy Function versus Cell Complexity

A fundamental observation here is that there exists a trade-off between the complexity of the energy functions implemented and the amount of potential performance improvements. The complexity of the correlation compute cell is directly related to the type and the number of energy terms implemented. This in turn affects the amount of parallelism that can be obtained from a given FPGA chip and hence the speedup. The FFT to correlation crossover point is, thus, dependent on the energy terms implemented.



Energy Function	Number of Correlations	Cell Operation	Resources used by a Correlation Cell		
			ALUTs	Registers	DSPs
Simple Shape Complementarity (2 bits per voxel)	1	Bitwise AND	15	17	0
Simple Shape Complementarity + Desolvation (ZDOCK)	2	Bitwise operations	50	41	0
Simple Shape Complementarity + Electrostatics (FTDock, DOT)	2	Bitwise operations and 1 multiplication	32	74	1
PIPER Energy Function without Pairwise potential	4	Bitwise operations and 2 multiplications	59	97	2
PIPER Energy Function (with 2 pairwise potential terms)	6	Bitwise operations and 4 multiplications	93	109	4
PIPER Energy Function (with 4 pairwise potential terms)	8	Bitwise operations and 6 multiplications	127	157	6

Figure 4·7: Cell operations and FPGA resource requirements for various energy functions.

For PIPER energy functions, with 8 correlations in parallel, we find that the crossover point occurs at ligand grid size of 16^3. For simpler energy functions, this boundary gets pushed further towards slightly larger molecule sizes. From another perspective, for a particular molecule size, the amount of performance speedup is larger for simple energy functions and it starts to fall as more energy terms are added.

To study the effects of different energy functions on the compute cell complexity and the resource requirements, we implemented the energy functions from different rigid-docking programs such as ZDOCK, FTDock, DOT and PIPER (with varying number of pairwise potential terms). Figure 4·7 shows the cell operation and the corresponding resources required for these.

As shown, for simple shape complementarity, with only 1 or 2 bits per voxel to indicate whether a grid point is inside or outside the molecule surface, the cell operation is a simple bitwise AND, requiring very little FPGA resources. This is an ideal case for FPGA acceleration as it allows immense bit-level parallelism and large number of PE replications, resulting in huge speedups over the serial code. As the complexity of the energy function increases, so does the resources per PE, reducing the number of PEs that can fit on the chip.

Table 4.2: Limiting FPGA resource for different energy functions

Energy function	**Number of Multipliers**	**Limiting resource**
Simple Shape Complementarity (2 bits)	None	Logic slices
Shape Complementarity (weighted sum of 2 terms)	$2N^2$ (for weights)	Logic slices
Shape Complementarity + Electrostatics	$N^3 + 3N^2$	Slices/Multipliers
PIPER energy function (4 pairwise potential terms)	$8N^3 + 5N^2$	Multipliers

Another interesting aspect here is that as the cell complexity changes, the factor that limits the overall potential performance on the FPGA also changes. This is shown in Table 4.2. For simpler energy functions, the FPGA slices present the limiting factor. As the energy functions become more complex, requiring large number of multiplication operations, the on-chip block multipliers become the critical resource. For our implementation of PIPER energy functions, with 4 pairwise potential terms, we utilize 100% of the on-chip multipliers and close to 90% of logic.

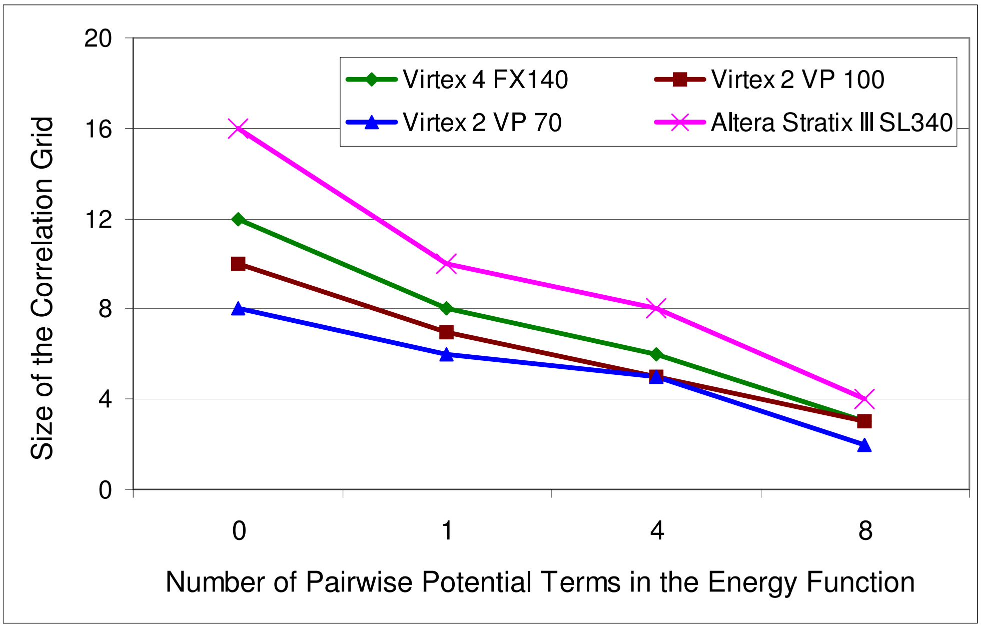


Figure 4·8: Complexity of correlation cell versus number of cells along each dimension for various FPGA technologies.

Figure 4·8 shows the correlation between the cell complexity and the largest molecule that can fit in its entirety, for different families of FPGA chips. For all the cases, we have the standard 4 terms of the PIPER energy function; the X-axis refers to the number of pairwise potentials computed in addition to those four. The Y-axis represents N, the size of one dimension of the 3D grid of computation cells. With one pairwise potential term, requiring 5 correlations in parallel, we can support ligand grid sizes up to 8^3 on a Xilinx Virtex IV FPGA. As more terms are added, this size becomes smaller until a point (with 8 pairwise terms; 12 correlations) where the molecule size that can fit becomes too small to be of any practical use. With

4 pairwise potential terms, which is typical of most PIPER runs, we can support ligand grids of up to 6^3 on a Xilinx Virtex IV FPGA and 8^3 on an Altera Stratix III FPGA. For larger ligand grids, multiple passes through the FPGA are required.

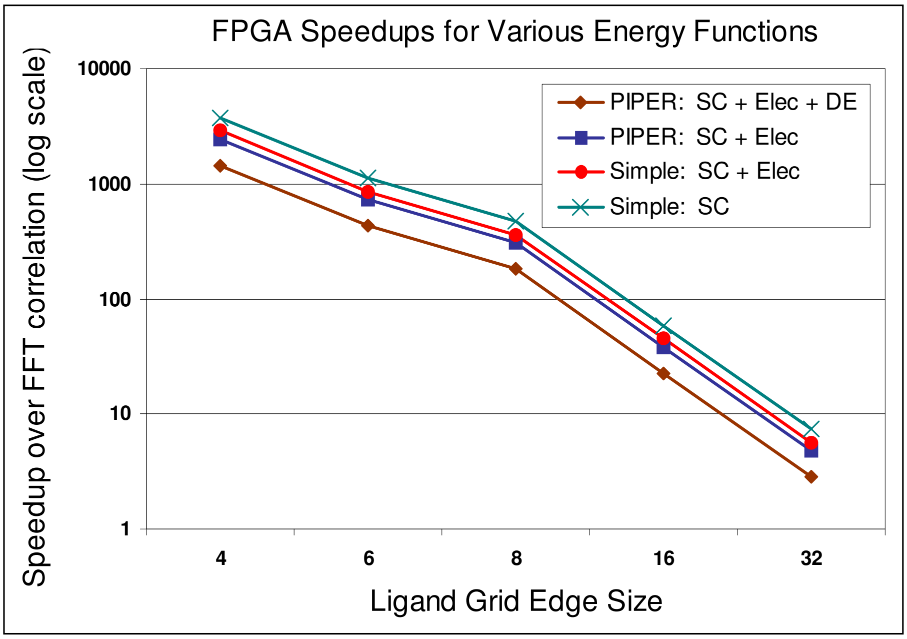


Figure 4·9: Speedup of the FPGA accelerator over a single core against ligand size, for various energy functions; only the correlation task is evaluated here.

The complexity of the compute cells translates directly into the amount of speedup that can be obtained from an FPGA implementation. Figure 4·9 shows the speedups for correlation only task for different energy functions as implemented in various rigid docking programs listed above.

The ligand size is varied from 4^3 to 64^3, keeping the correlation size fixed at 128^3. The results shown are on an Altera Stratix III FPGA, with the single core FFT correlation as the reference. For simple shape complementarity based docking ("Simple: SC" in the graph), with only two bits per voxel, a speedup of around 3800x can be achieved for a 4^3 ligand. As the energy function becomes more complex, requiring more logic resources per compute cell, the speedup starts to drop, though still around 1000x even with PIPER energy function with 4 pairwise potential terms ("PIPER: SC + Elec + DE" in the graph). For all the energy functions, speedups of 2 to 3 orders of magnitude can be obtained for ligand sizes up to 8^3. As the ligand size is further increased, the speedup starts to drop significantly, and ultimately around 32^3, the FPGA direct correlation concedes to the FFT correlation. The main reason for this drop in performance for larger ligands is the inability to fit the entire ligand grid on the FPGA. This requires swapping different parts of the ligand grid in and out of the FPGA pipeline and multiple passes through the receptor grid, leading to significant performance drop.

Overall, the performance of the FPGA-accelerated rigid-docking depends both on the complexity of the scoring function as well as the molecule size. For small molecule docking, even with complex energy functions requiring up to 8 correlations, speedups of 3 orders of magnitude can be obtained on the core computation.

4.2.5 Score Filtering on the FPGA

For each rotation, the correlation between the ligand and the receptor grid generates up to a few million scores. Docking programs typically save just a few top scores from each rotation. Maximum scores often cluster close to each other. To avoid reporting multiple local maxima, docking programs *filter* the scores by selecting the top scores from different parts of the result grid. Though the filtering step is computationally very simple and can be efficiently performed on the host, there are three main reasons why it is essential to accelerate the filtering step; first is simply the Amdahl's law. Obtaining good performance requires accelerating as much computation as possible. Second, performing filtering on the host requires that all the correlation scores be saved on the FPGA BRAMs or off-chip memory; filtering reduces this storage requirements to just a few memory elements. Finally, performing filtering on the host requires transferring a few megabytes of data from the FPGA to the host per rotation, leading to large data-transfer overhead.

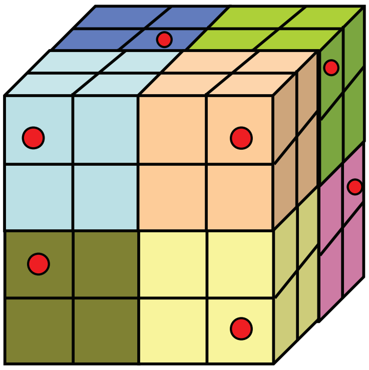


Figure 4·10: Score filtering; the result grid is divided into small cubes. The best score from each cube (shown as a red dot) is returned to the host.

The FPGA filtering scheme used here is identical to that described in [VGMH06]. The score grid is divided into different cubical regions and the score filter stores the best score from each cube (see Figure 4·10). In each cycle, the correlation systolic array generates a correlation score which is fed to the score filter. The score filter also receives the address of the current score. This address is generated by a controller and is simply the x, y and z coordinates of the current score in the result grid. The higher order bits of the score coordinates determine which cube of the result grid the current score belongs to. The score combiner compares the input score with the saved best score for the current cube and updates the score and its address if needed. Once all the scores for the current rotation have been generated, they are offloaded to the host, along with their addresses. The score address, along with the rotation number identifies the pose (relative offset and rotation) of the ligand with respect to the receptor.

4.2.6 Supporting Large Ligands

Implementing complex energy functions limits the size of the molecule that can fit on the FPGA chip. With PIPER energy functions, the molecule size on current generation FPGAs is limited to around 6^3 to 8^3. Molecular docking, however, often involves grids larger than 8^3, particularly for protein-protein docking. Even though with every new generation of FPGA chip, the molecule size that can fit gets larger, it is still desirable to be able to support larger molecules on any given chip. To enable this, we have implemented a scheme to compute correlation scores in pieces. Note that the receptor size can still be as large as before. We call this *piece-wise* correlation, as it involves loading different pieces of the ligand grid into the FPGA correlation cores and storing the partial scores in a score memory. The scheme requires multiple passes through the FPGA systolic array; for each ligand piece, all the three phases of systolic correlation (namely loading, correlation and flushing) are performed. During each pass, the entire receptor grid is streamed through the compute cells, generating the partial correlation scores.

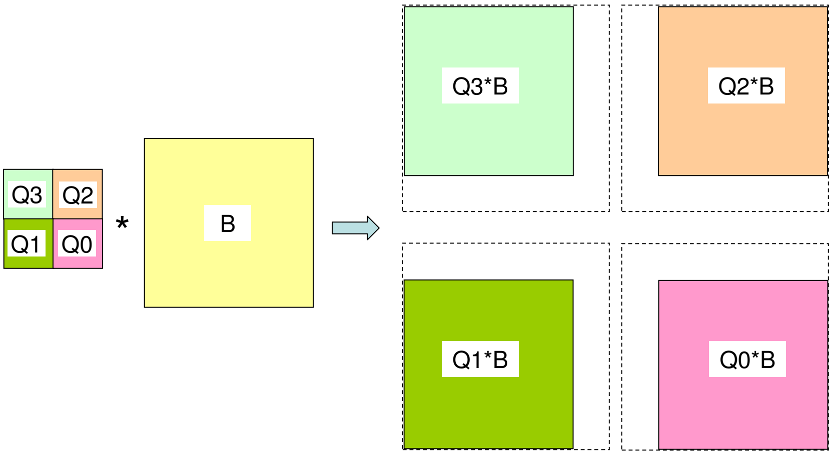


Figure 4·11: Piece-wise correlation [Gu08]; scores from different quadrants must be routed appropriately.

Note that, unlike the complete-ligand case where the score generated in each cycle is the total correlation score and can be sent straight to the on-chip score filter, the scores generated during piece-wise correlation are partial and must be accumulated into the total score. Based on the relative position of the piece in the total grid, the scores generated during piecewise correlation contribute towards different elements of the total score grid. Figure 4·11 shows this for a 2D correlation with the ligand grid divided into 4 quadrants. The relation between the total score and the partial scores can be written as follows:

$$S_{total}(i,j) = S_{Q0}(i,j) + S_{Q1}(i,j-Q_x) + S_{Q2}(i-Q_y,j) + S_{Q3}(i-Q_y,j-Q_x) \quad (4.3)$$

where Q_x and Q_y represent the x and y dimensions of the quadrant.

From Equation 4.3, the address of the final score to which a particular partial score contributes can be obtained as per Table 4.3. During each cycle, a partial score is generated and added to the appropriate address in the score memory.

Table 4.3: Routing the partial scores to the total score grid

Partial score from different quadrants	**Final score to which it contributes**
$S_{Q0}(i,j)$	$S_{total}(i,j)$
$S_{Q1}(i,j)$	$S_{total}(i,j+Q_x)$
$S_{Q2}(i,j)$	$S_{total}(i+Q_y,j)$
$S_{Q3}(i,j)$	$S_{total}(i+Q_y,j+Q_x)$

To support the above scheme, the following hardware modules have been added to the overall design:

- **New controller FSM:** A new controller FSM is needed for loading the ligand grid-pieces and routing the partial scores appropriately to the score memory.

It generates the address of the score in the score memory, fetches the partial score, adds the new partial score and updates the score memory. During the last ligand piece, the accumulated scores are sent to the score filter.

- **Ligand memory:** A ligand memory is added to store the entire ligand grid. This was not needed earlier since the ligand voxels were stored directly in the compute cells. For piecewise correlation, a different piece from this memory is loaded into the compute cells during each pass.
- **Score accumulator** adds the current partial score to the score fetched from the score memory.

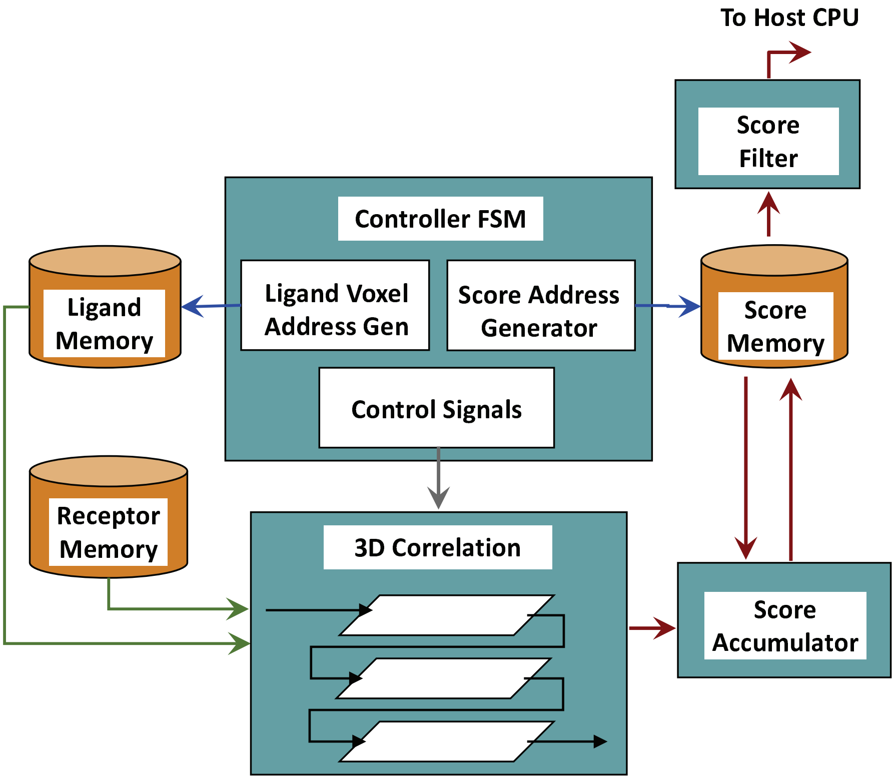


Figure 4·12: Block diagram of piece-wise docking for computations with large ligand molecules.

The entire scheme for piece-wise correlation is shown in Figure 4·12. It is worth noting that the logic overhead for supporting piece-wise correlation is minimal. Table 4.4 compares the logic utilizations for the correlation pipelines with and without the support logic for piecewise correlation. For this example, a simple compute cell is used with an $8 \times 8 \times 8$ on-chip array of cells. The designs in the first two rows both operate on an $8 \times 8 \times 8$ ligand; the difference is that the second has the overhead hardware for swapping. We see that the overhead for supporting piecewise correlation scheme is minimal. Also, the clock rate is virtually unchanged. The last two rows show the support required to operate on 16^3 and 32^3 ligands, respectively, keeping the number of hardware cells constant. Clearly, larger correlations can be supported without much increase in the resources required.

Table 4.4: Resource utilization for piecewise correlation on Xilinx Virtex-II Pro VP70 FPGA. Adding support for piecewise correlation has very little overhead.

Design	Resource Utilization		
	slices	flip-flops	LUTs
No piecewise support $8 \times 8 \times 8$ ligand	31009	22110	33522
Piecewise support $8 \times 8 \times 8$ ligand	31457	22429	34295
Piecewise support $16 \times 16 \times 16$ ligand	31519	22448	34405
Piecewise support $32 \times 32 \times 32$ ligand	31630	22475	34627

4.2.7 Support for Different Energy Functions

The core computation in rigid docking is essentially a correlation between two grids, each representing the voxels of the proteins to be docked. Different rigid docking programs differ primarily in the voxel data type and the number of energy functions used to evaluate the goodness of fit. To make the FPGA correlation coprocessor more useful and applicable to a wide range of docking programs, the compute cells must be able to handle these different energy functions. One way to do this is to insert the logic for multiple different energy functions in each compute cell and select the required subset at runtime. There are two disadvantages of this approach. First, generalizing the compute cells in this manner bloats the design, losing the potential performance speedup achievable with FPGAs. FPGAs achieve many-fold speedups over CPUs due to the optimized processing cores specific to the computation at hand. Generalization of the processing cores would result in significant reduction in the total performance gain. Secondly, incorporating any new energy function that becomes of interest in the future would require redesigning the compute cells. Although the overall control structure will still remain the same, addition of the new energy functions would require a hardware engineer to modify the HDL code manually. This is clearly undesirable and definitely unattractive to biochemists and molecular biologists.

To alleviate both these problems, we created an automatic HDL generator program that generates hardware components specific to a particular set of energy functions. The program, essentially a C script, generates VHDL modules for design components such as the compute cell and the score combiner that change with the energy functions. The type and the number of energy functions and the desired precision are specified in a simple text file which is the input to the HDL generation script. Note that the control structure remains unchanged and is not regenerated.

Generating accelerators for different correlation-based rigid docking program, thus, simply requires running the script and re-synthesizing the design. Although this still requires some knowledge of the CAD tools and downloading the new bit file to the FPGA, this entire process can be easily automated using a simple script.

4.3 Algorithms for GPU Acceleration of Rigid Docking

Direct correlation clearly has its limitations and as the size of the molecule increases, FFT correlation starts to catch up. On the FPGA, we notice that this crossover point happens almost directly on the small/large molecule boundary. Due to this, the FPGA direct correlation is very suitable for protein-ligand docking but not the preferred method for protein-protein docking. Moreover, performing 3D FFT on current generation FPGAs is not cost-effective, as it requires large amount of logic resources and leads to poor overall performance. To enable fast and efficient protein-protein docking, we investigated the use of the graphics processor. Our study indicates that even on the graphics processor, direct correlation is the preferred method for up to a certain molecule size. This is due to many reasons: direct correlation lends itself well to parallelization, multiple correlation scores can be computed together, multiple rotations can be scored in a single pass of the protein grid, and large data reuse amortizes the overhead of data fetch and kernel launch. As the molecule size increases, these benefits start to fade and the FFT correlation becomes the method of choice.

We, thus, present a set of GPU-based solutions that work well for both protein-protein and small-molecule docking domains. As described above, both the FFT and direct correlation are useful in the overall solution; the choice of implementation depends both on the size of the molecule grids as well as the complexity of the energy function.

In addition to the correlation, we also accelerate other docking steps on the graphics processors such as accumulation and scoring and filtering. Below we discuss the GPU algorithms for these computations. These were implemented on an NVIDIA TESLA C1060 GPU card using the CUDA programming model. Like in the case of the FPGA accelerated version, rotation and grid assignment is still done on the host but its latency is completely hidden.

As discussed in section 2.5.3, CUDA has three key abstractions: thread hierarchy, shared memory and synchronization. Efficient GPU programming requires taking into account these abstractions. In particular, the key considerations in obtaining good performance include:

- **Efficient distribution of computations:** Efficient distribution of work among different GPU threads so as to maximize parallelism and minimize memory access conflicts is of utmost importance. In CUDA, this is explicitly specified by the programmer and care must be taken in designing this distribution.

- **Execution configuration:** CUDA kernels are launched with a hierarchical organization of threads in thread blocks and grid of blocks. Even though it is in general a good idea to divide the computations among a large number of threads and thread blocks, the complex communication and synchronization can often lead to poor overall performance. It is, thus, important to find the complexity-utilization sweet-spot that results in the best overall performance.

- **Use of the memory hierarchy:** It is essential to utilize the GPU memory hierarchy efficiently; in particular, the use of the GPU shared memory to the extent possible.

- **Minimizing data transfers:** Transfer of data between the GPU and the host memory results in overhead that affects the overall performance. It is critical to minimize this transfer. There are two main components to minimizing the data transfer: moving as much computation to the GPU as possible and implementing any appropriate data filters before sending the data back to the host.

In the following subsections, we present our GPU designs for different components of the PIPER docking program. For each component, we discuss the design in terms of the above abstractions. We believe this to be the first published study of a production docking code accelerated with a GPU.

4.3.1 Direct Correlation

The inner loop to compute direct correlation between two molecule grids includes shifting one of the grids over the other, multiplying the corresponding voxels on the overlapping parts of the grids and summing the individual multiplication scores. On the GPU, we perform this by holding the smaller (ligand) grid constant and moving the larger (receptor) grid over it along the three translational axes. The outer loop repeats this for each rotation and for each energy function.

Data transfers and memory hierarchy: To perform correlation on the GPU, the ligand and receptor grids need to be transferred to the device memory. For the receptor grid, this is done only once. Since rotation and interpolation of the ligand grid is performed on the host, the ligand grid must be transferred to the GPU memory after each rotation. We hide this per-rotation transfer latency by overlapping the transfers with the computations on the GPU.

Since every multiprocessor needs access to both the ligand and the receptor grid, they either need to be stored in the device's global memory, accessible by all the multiprocessors, or duplicated in the local shared memory of each of the multiprocessor. Use of the shared memory is desirable because of its lower access latencies. However, since the receptor grids are large and the shared memory per multiprocessor is relatively small, it is not possible to copy these grids to the shared memories; rather, we store the receptor grids in the global memory (see Figure 4·13).

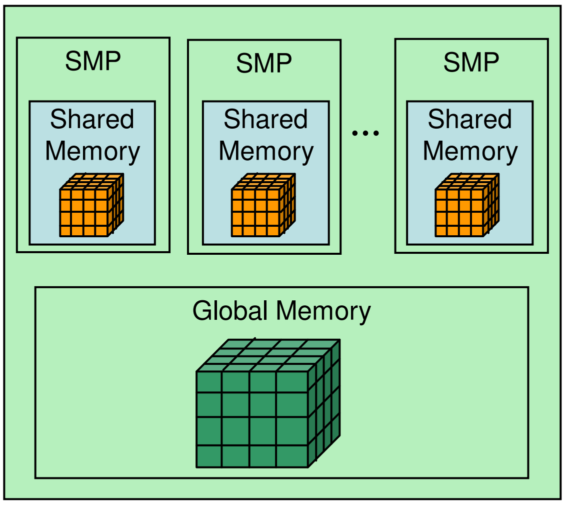


Figure 4·13: Direct correlation on GPU; larger grid is stored on the device global memory while the smaller grid is copied to the shared memory of each multiprocessor.

Ligand grids, however, are much smaller and hence we tried to store them variously in the devices' shared memory or constant cache, both of which provide much faster access compared to the global memory. We found that access times from the constant memory and the shared memory are identical, unless there is a cache miss on the constant memory, in which case, the data is accessed from the global memory. Both the shared and the constant memory, however, are small and thus limit the size of the largest ligand that can fit in its entirety. An important consideration is the complexity of the energy function used. With 4 pairwise potential terms, we can fit a ligand grid of size up to 7^3 in the shared memory and 8^3 in the constant memory. For larger ligand grids, we store the ligand in global memory, which significantly degrades the performance due to higher access latencies. Thus, here again exists a trade-off between the performance improvements and the problem size and complexity.

Work distribution and thread hierarchy: To distribute the task of computing the correlation scores for all the translations along the 3-axes, we represent the task as the 3D result grid that needs to be computed. Here, each grid point represents the correlation score for a translation. For different energy functions, different result grids must be computed, one for each. The total pose score is then a weighted sum of these different grids.

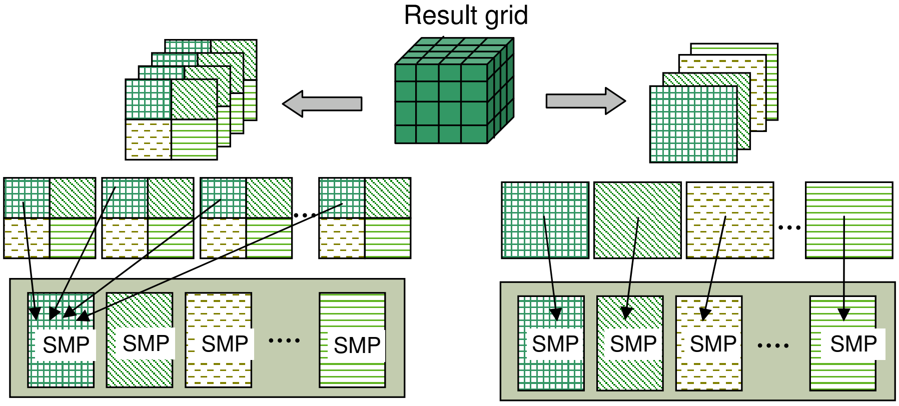


Figure 4·14: Distribution of work on the GPU for direct correlation: (Left) Each thread block works on part of every 2D plane, (Right) Different 2D planes are assigned in their entirety to different blocks.

The distribution of work on different GPU threads can now be seen as distributing the computation of different portions of the result grid across multiple threads and thread blocks. This can be performed in various ways. We tried two schemes: In both, we launch the kernel with a 2D array of thread blocks, each with a 3D array of threads. In the first scheme, each thread-block is responsible for computing a part of the 2D result plane for all the 2D planes in the 3D result grid (see Figure 4·14(a)). In the second distribution, we assign different 2D planes to different thread-blocks. The threads on each of those thread blocks compute a larger part of the 2D plane, but only for the planes assigned to the current thread block (see Figure 4·14(b)). Both distributions result in similar runtimes, though one or the other can have better performance for various non-cubic grids.

Table 4.5: Direct correlation on a GPU; runtimes for 8 correlations of size 128^3 each. Correlation runtimes depend on the execution configuration as well as the ligand size.

Ligand size	**Block size**	**Grid size**	**Cells updates per block**	**Cells updates per thread**	**Runtime (ms)**
8^3	2x2x2	16x16	8K	1024	1243
	4x4x4	16x16	8K	128	461
	8x8x8	**16x16**	**8K**	**16**	**245**
	8x8x8	8x8	32K	64	435
	8x8x8	4x4	128K	64	632
16^3	**8x8x8**	**16x16**	**8K**	**16**	**1650**
	8x8x8	8x8	32K	64	3120
	8x8x8	4x4	128K	256	4329

We noticed, however, that the execution configuration used to launch the GPU correlation kernel does affect the execution runtimes. Table 4.5 shows the runtimes for 8 parallel correlations for two different ligand sizes, using different hierarchies of threads and thread blocks. For each case, the correlation size equals 128^3. The best execution configurations are shown in bold.

Shown in Table 4.5 are different thread geometries and the corresponding amount of work done by each thread block and each thread. As can be seen, keeping the number of thread blocks (i.e. grid size) constant, as the number of threads per block (i.e. block size) increases, work per thread decreases and thread-level parallelism increases, resulting in improved performance. With constant number of threads per block (at 8^3 for the results shown), having more thread blocks yields better performance. Both these phenomena are attributes of the availability of large number of independent execution groups that can be efficiently swapped and scheduled by the thread scheduler to hide the memory access latencies.

Synchronization: Threads compute different portion of the result grid independent of each other. There is no dependency among different threads and hence no need for any synchronization during the kernel. Threads are automatically synchronized when the GPU kernel returns the control to the host.

Based on our experiments with different ligand sizes, we noticed the following peculiarities of direct correlation on the GPU with respect to the PIPER energy functions:

- Only a limited number of pairwise potential terms can be supported without swapping or storing the ligand on the global memory; more terms would result in larger memory requirements for storing the ligand grids. We could support ligand grid sizes of up to 8^3 when using 4 pairwise potential terms (8 correlations in total).

- Unlike FFT-based correlation, where the time to perform the correlation depends on the size of the padded FFT grid (which is equal to the sum of the ligand and receptor sizes), the direct correlation time depends mainly on the size of the ligand. This is because the size of the multiply-accumulate inner loop is proportional to the size of the ligand grid. Thus, for the same FFT size, direct correlation can perform significantly better for smaller ligands whereas FFT correlation runtimes remain unchanged.

- Smaller ligand grids allow us to perform a further optimization: we store voxels for multiple rotations of the ligand in the constant memory. This allows the correlation inner loop to compute multiple scores in each iteration. This yields two benefits; first, the loop overhead gets amortized over multiple rotations, resulting in effects similar to loop unrolling. More importantly, each receptor voxel, fetched from the global memory (which is not cached) gets used multiple times, thus reducing the overall fetch time by the number of rotations that can be computed at once. Since the global memory has higher access latency, reducing accesses to it results in significant performance improvements. For 4^3 ligand grids, we can perform 8 rotations in one iteration, achieving an additional speedup of 2.7x over direct correlation on GPU performed one rotation at a time.

- Direct correlation results in a performance improvement over FFT correlation only for ligand grids which can fit in the shared or constant memory. Accessing ligand from global memory results in significant performance loss.

- Direct correlation provides automatic filtering of the top scores since different multiprocessors, computing different regions of the result grid, can find and report their local best score. An extra step of finding the best of these local best scores, as well as flagging the cells for exclusion still needs to be performed by a master thread (see Section 4.3.3).

4.3.2 FFT Correlation

For protein-protein docking, direct correlation is not the preferred method. To enable fast docking of large grids, we revert to FFT based correlation.

Fast Fourier transforms are ubiquitous, with applications in digital signal and image processing, computational fluid dynamics, data compression and control systems. FFTs are used for a variety of computations such as solving partial differential equations, correlations and convolutions and long multiplications. Computing FFT is computationally expensive, requiring large number of floating point operations. Due to this, many highly tuned FFT libraries have been developed, with the most popular one being the FFTW [FFT10]. Due to their immense floating point capabilities, GPUs have naturally been explored to expedite the FFT computation. Recently, several studies have presented GPU acceleration of the FFT algorithm, including a GPU FFT library from NVIDIA [NVI08a]. We realized that some of these designs are highly tuned for NVIDIA GPUs and achieve good performance. We hence use one of these libraries instead of trying to implement one of our own.

FFT correlation of two grids involves 4 steps: computing the forward FFTs of the two grids, computing the complex conjugate of the transformed grids, modulation (multiplication) of the resulting grids, and an inverse FFT of the product grid. In our GPU accelerated docking code, we use the NVIDIA CUFFT library for computing the forward and inverse FFTs. While several other excellent FFTs for GPUs have been reported (see, e.g., [GLD+08, NOEM08]), we were not able to find another that is both publicly available and superior for this problem.

Complex conjugate and modulation of the grids are performed using GPU kernels, with the distribution of work in much the same way as direct correlation.

Data transfers and computations: The receptor grids for different energy functions are transferred once to the GPU memory and their forward FFTs are computed by calling the CUFFT library function. A separate kernel call computes the complex conjugate of the transformed grids on the GPU. No extra data transfer is required for this since the transformed grids are present on the GPU global memory, which is persistent across kernel calls.

The transformed receptor grids are held in the GPU global memory throughout the docking phase. This avoids the need to transfer large amounts of data from the host to the device for each rotation. This approach, however, is possible only if there is enough device memory to simultaneously store multiple grids of N^3 complex entries each. We find that our Tesla C1060 card easily fits all 22 grids ($4+P$; $P = 18$ desolvation grids) for a large grid size ($N = 128$).

For each rotation, the ligand grids for different energy functions are copied to the GPU and are transformed. Modulation of the transformed grids is performed by a subsequent kernel; again, no extra data transfer is required. The modulated grids are then inverse transformed using the CUDA FFT library. In the case of the desolvation grids, the inverse FFT is followed by an extra step to accumulate the scores of the different desolvation energy terms to obtain a total desolvation score. This accumulation is also performed on the GPU, yielding similar performance benefits as performing modulation on the device.

At this point, the correlation scores for different energy functions have been generated and can be transferred back to the host. This would require transferring up to 22 grids, each with around 2 mega floating point entries. Accumulation of desolvation grids on the GPU reduces this transfer to 5 grids. The transfer size can be further reduced significantly (to just a few floating point values) by performing scoring and filtering on the GPU.

Work distribution, thread hierarchy and synchronization: The distribution of work for the kernels and thread hierarchy and synchronization are similar to what was discussed for direct correlation. Due to the simple kernels for complex-conjugate, modulation and accumulation, however, the effect of different thread organizations is not as significant.

In addition to the support for larger grids and hence protein-protein docking, FFT based docking on the GPU also offers another benefit; in contrast with the direct correlation, FFT-based correlation performs individual correlations in serial order. This enables scalability to any number of correlations, thus allowing any number of pairwise potential terms in PIPER desolvation energy function. Addition of any new energy function is also straightforward. Further, unlike direct correlation on the GPU, where the size of the ligand grid is limited by the size of the constant memory, FFT correlation can support any ligand grid size, so long as the total FFT grid fits in the global memory.

4.3.3 Scoring and Filtering

For each rotation, after the correlation scores for the various energy functions have been computed, two steps remain: computing the total score for each translation (goodness of fit between the two molecules being docked) and selecting the translations with the best scores (filtering to find the best pose). Scoring simply involves computing the weighted average of the scores for the various energy functions. The PIPER filtering algorithm selects the top scoring conformations from different regions of the result grid. In PIPER, scoring and filtering is performed using multiple sets of weights (coefficients). For each set of coefficients, the number of scores returned per rotation is a parameter, with a default of one.

Performing scoring and filtering on the GPU results in two-fold benefit. First, and the more obvious, performance improvement comes from processing multiple scores in parallel on the GPU. This is achieved both at the thread-block level (different coefficients being processed on different multiprocessor) and the thread-level (different parts of the grids being processed by different threads).

The second performance improvement comes from the side effect of performing filtering on the device. Since grid modulation is performed on the GPU, performing filtering on the host would require transferring 5 modulated grids (each with N^3 elements) to the host for every rotation. With filtering on the GPU, this data gets reduced to only the few top scores per rotation, thus saving on the data transfer time.

Data transfers: Performing scoring and filtering on the GPU requires very little data to be transferred. This is because the modulated grids are already present on the GPU global memory; the only extra transfer required is to copy the scoring coefficients to the GPU. Since the GPU constant memory is cached and provides faster access, we use it to store the different coefficient sets. Coefficients are copied to the GPU constant memory only once and the transfer time is very small compared to the total runtime.

Work distribution, thread hierarchy and memory hierarchy: The work on the GPU is divided by distributing the K coefficient sets onto K different multiprocessors ($K = 6$ typically), each computing its respective best scores independent of the rest (see Figure 4·15(a)). In our initial naïve approach, we assign just a single thread to each thread block (running on each of the K multiprocessors). Thus, the kernel is launched with a total of K threads; each computes N^3 weighted scores and stores the top score in the device global memory (see Figure 4·15(b)). Since there is only a single thread per multiprocessor, the top score can be locally held in the register variable and there is no use of the shared memory.

This version results in significant slowdown compared even to the serial code. This is clearly due to the heavy underutilization of the processors. Since there is only one thread per block, only one of the processors are utilized in each of the K multiprocessors. For $K = 6$, only 6 of the available 240 processors are utilized. Moreover, note that the GPUs have no cache and the memory latencies are hidden by fast and efficient swapping of threads. In this case, there are simply not enough threads for the CUDA thread scheduler to swap, leading to worse performance than computing on the host.

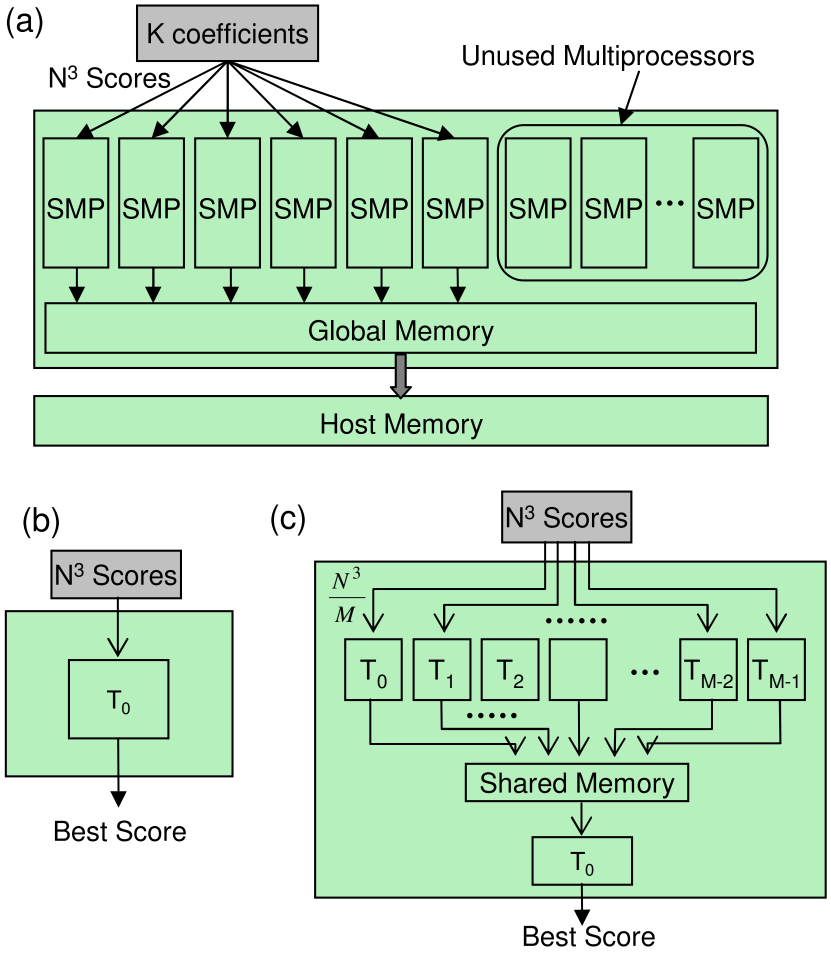


Figure 4·15: Scoring and filtering on the GPU: (a) Different coefficients are distributed among different multiprocessors, with each containing (b) one thread, or (c) multiple threads.

In our second scheme, we assign different coefficient sets to different thread blocks, with each thread block now containing M threads (M = 500). The correlation grids are divided equally among all the threads, with each thread computing N^3/M weighted scores and finding the best scores within its subset (Figure 4·15(c)). For each thread block, we allocate an array in the device shared memory. Each thread finds the best score within its subset and stores it, along with the corresponding score index, in the shared memory, independent of all the other threads. Once all the threads finish processing their subset of results, a synchronization barrier is reached. A master thread (thread 0) then processes the shared array to find the best of these best scores and stores it in the array of top scores in the device global memory. If the number of top scores to be reported per rotation (per coefficient) is greater than 1, then the master thread performs an additional task of flagging the cells neighboring the current best score so as to exclude them from consideration in the next iteration. This is done to avoid reporting multiple top scores from the same region.

Flagging of the neighboring cells presents a trade-off in the use of the memory hierarchy and the number of look-ups required to determine the exclusion. In the serial version, PIPER maintains an array of integers, with one entry to flag each of the N^3 result cells. Exclusion can then be determined in constant time simply by checking the flag at the array index equal to the cell index. In our GPU version, maintaining such a large array on the GPU shared memory is not feasible. We first tried to maintain a small array on the shared memory, containing only the indices of the cells to be excluded in the next iteration. Even though this scheme lets us use the fast shared memory, it requires each thread to traverse the entire array to check if the current cell is present and must be excluded. Overall, this results in a negative speedup, with GPU filtering performing 2-3x slower than filtering on the CPU. In our second approach, we store the N^3 array on the device global memory, with each thread reading only one array entry to determine exclusion. This improves the performance significantly, with GPU filtering now performing better than CPU filtering. Note that the size of the array in the global memory increases to $K \times N^3$ since all the coefficients are being processed in parallel.

Cell-flagging on the GPU presents an interesting trade-off: increased computation versus higher access latencies. Though in a part of the current work (discussed later) we find that it is often beneficial to perform redundant computations to avoid accesses to the slow global memory, that approach is clearly not effective here. In this case, the increased look-up time from the shared memory more than negates the benefit of the reduced access latencies.

Synchronization: In the filtering scheme presented above, each multiprocessor performs scoring and filtering for a different coefficient set. These are completely independent and do not require any synchronization. The threads within a particular thread block, however, must be synchronized before the master thread finds the global best score from the shared memory and flags the neighboring cells.

The simple synchronization scheme employed here comes with a price: underutilization of the available multiprocessors. Even though there are many threads within each thread block to fully utilize all the processors of a particular multiprocessor, only 6 of the 30 multiprocessors available on the Tesla C1060 GPU are utilized (since the kernel is launched with only 6 thread blocks, one for each of the 6 coefficients). Multiprocessor utilization can be improved by a more complex distribution of work for each coefficient across multiple thread blocks. This, however, is not likely to improve the overall performance. This is due to the fact that the CUDA architecture does not support synchronization among threads across different thread blocks and requires each thread-block to be able to execute independently and in any order. Synchronizing threads across multiple thread block would require multiple kernel launches as well as the use of the global memory for reduction. Since the filtering time is already a very small fraction of the total runtime, the overhead to support complex synchronization would nullify any potential performance benefit obtained from higher utilization.

4.4 Divergence in Accelerated Rigid Docking Algorithms

In the acceleration of rigid-docking programs, no single platform or algorithm is overall superior to the others. Based on the class of docking and the complexity of the underlying algorithm, different accelerators are preferred. Here we discuss the divergence in the methods for accelerated rigid docking and their relative price-performance benefits.

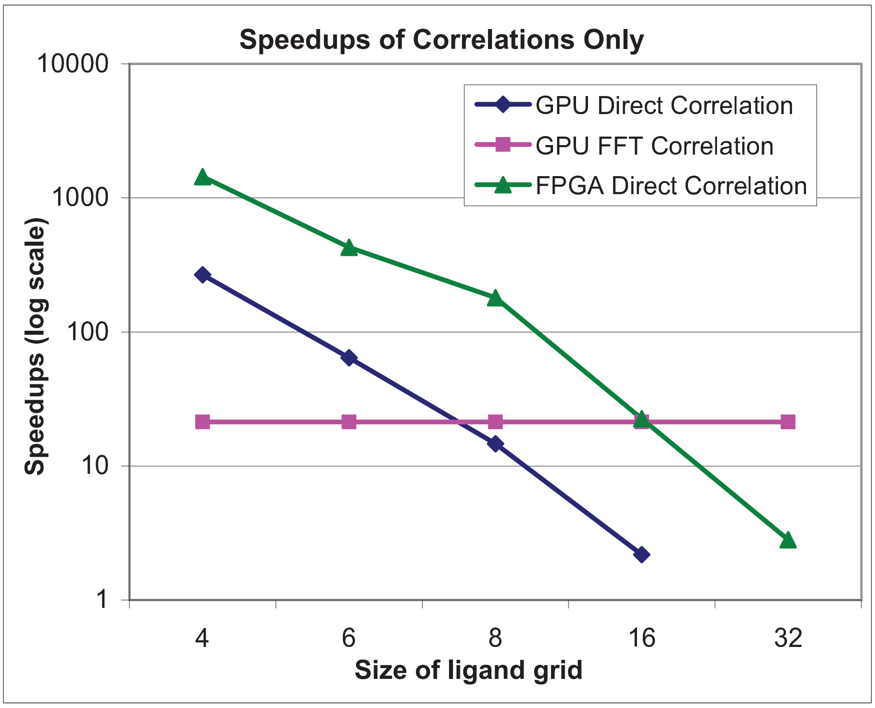


Figure 4·16: Correlation-only speedups for PIPER energy function (with 4 pairwise potential terms; 8 correlations). The horizontal axis is the edge-size of the ligand grid.

Figure 4·16 shows a comparison of FFT-based and direct correlation on the GPU, and the FPGA-accelerated direct correlation. The speedups shown are for the PIPER energy function with 4 pairwise potential terms (thus 8 correlations) with the single core FFT as the reference. The ligand grid size is varied from 4^3 to 32^3, with the total correlation size fixed at 128^3. For small grid sizes, direct correlation based method on both the FPGA and the GPU achieves two to three orders of magnitude speedup, whereas the FFT-based correlation on the GPU achieves a speedup of around 20×. As expected, the GPU-FFT maintains constant performance while the performance of the correlation-based method drops sharply with ligand size. For the GPU direct correlation, the drop in performance is due to two factors: the larger inner loop and the need to access the ligand grid from the global memory. For the FPGA direct correlation, the drop in performance is mainly due to the inability to fit the entire ligand grid on chip and thus the need to swap different parts of the ligand grid in and out of the FPGA pipeline.

Overall, for GPUs, direct correlation is better for small ligands, while the FFT maintains good performance throughout. The crossover point is at a ligand size of about 8^3. For the FPGAs, the crossover point with the GPU FFT is around 16^3.

An important question is what these results depict about the relative merit of these two architectures for rigid molecule docking and similar computations that involve 3D correlations. From the chart, it can be concluded that for small ligands, direct correlation is superior to the FFT, both on the FPGA as well as on the GPU. On the GPU, the crossover point lies around 8^3. The FPGA remains superior to the GPU for low precision correlations, and has better performance for ligand sizes up to about 16^3. Up to that size, the performance difference is substantial; FPGA direct correlation performs about an order of magnitude better than the GPU-direct correlation. Due to the Amdahl's law, however, this gap is much narrower for the overall rigid-docking including the rest of the computations. For all ligands larger than 16^3, the GPU version continues to give excellent performance, while the FPGA stops being cost-effective at around 25^3.

4.5 Summary

The computational complexity of rigid molecule docking as well as its widespread use in molecular modeling makes its acceleration highly desirable. Moreover, as we have shown, the computations in rigid docking are amenable to hardware acceleration, though after significant restructuring. The performance of both protein-protein and small-molecule docking can be improved significantly by mapping these computations to hardware. Nevertheless, the method of acceleration and the preferred hardware platform for these two classes of docking are very different. While the FPGAs achieve multi-hundred fold speedups on the core computations in small-molecule docking, they seem ineffective for docking large molecules. FFT correlation based methods on the GPUs, on the other hand, achieve good performance for both protein-protein and small molecule docking, though the speedups for small-molecule docking are somewhat inferior to that on the FPGAs. Putting these results into the users' perspective, the GPU-accelerated rigid-docking is clearly more cost-effective and achieves good performance for both the domains of rigid docking. For small molecule docking, however, FPGA offers even better performance, albeit at a higher cost.

Chapter 5

Acceleration of Binding Site Mapping

5.1 Overview

In this chapter, we present the FPGA and GPU algorithms for the acceleration of the binding site mapping program FTMap. The main task in binding site mapping is to flexibly dock a set of small molecule probes on a given protein. Like many other flexible docking programs, FTMap models the flexibility of the probe molecules in two steps. The first step rigidly docks the probes to the protein using the PIPER program and retains a few thousand top scoring protein-probe complexes for each probe. The second step then performs the CHARMM potential based energy minimization of these complexes. The acceleration of binding site mapping in general, and the FTMap program in particular, thus requires accelerating the process of rigid docking and energy minimization. Here we describe the acceleration of the energy computation phase of CHARMM potential minimization. The rigid docking phase is accelerated as discussed in the previous chapter. Since the grids for the small molecule probes (or ligands) used during mapping are very small, we perform accelerated docking of the probes using direct correlation, both on the FPGA as well as on the GPU.

Energy minimization is a widely applied routine in many molecular modeling algorithms. The underlying computation is similar to an N-body problem. Moreover, energy minimization is an iterative process, requiring many hundreds to even a thousand iterations to converge. This makes the computation very time consuming. With thousands of conformations between the protein and each probe to be minimized, the mapping task usually takes many CPU-hours. Obtaining coarse-level parallelism for the minimization task is trivial; each protein-probe complex is independent of the rest and can be minimized in parallel on different processors. This makes its acceleration using a cluster of processors an easy task.

While energy minimization can potentially benefit from fine-grained acceleration, and while this computation is superficially similar to the well-studied molecular dynamics, little or no work has been done in this area. Accelerating the minimization computation using a single chip is a challenging task. This is due to three main reasons. First, the time for minimizing a single protein-probe complex on a serial processor is already comparatively small, typically less than a minute. Secondly, minimization is an iterative process, with the computations in the current iteration being dependent on the results of the previous. Finally, there is comparatively little computation to be performed per iteration and that a substantial fraction of it is apparently serial. Due to this, the minimization operation affords little to no fine-grained parallelism.

Though one of the basic techniques in FPGA-based acceleration is the replication of processor-cores for parallel computation, this is not viable for the minimization computations. This is because the complexity of the energy minimization calculations leads to high resource requirements on the FPGA, allowing for very few, if any, replications. We address this using streaming through very deep pipelines, thus effectively performing many computations in parallel.

On the GPUs, the problem is a different one; though there are hundreds of processor cores on a GPU, their communication and synchronization pattern is relatively fixed. This, combined with the nature of the underlying computation, does not allow efficient utilization of the available processors. We address this by performing significant restructuring of the original data structures and introducing a new data structure to enable efficient, parallel computations on the GPU multiprocessors.

5.2 Algorithms for FPGA Acceleration of Energy Minimization

The main computation for energy minimization is the repeated evaluation of various bonded and non-bonded energy terms. Of these, the non-bonded electrostatic energy evaluation constitutes most of the computation. In the current work, we accelerate this computation by mapping to FPGA pipelines. Due to the highly precision-sensitive nature of the underlying computations, energy evaluation on a serial processor is performed using double precision floating point arithmetic. In the FPGA-accelerated electrostatic computation, we use single precision floating point arithmetic. The effects of the reduced precision are discussed in detail in chapter 7.

For long, FPGAs were considered effective only for bitwise and fixed point computations and not very suitable for floating point operations. This was mainly due to there being no dedicated floating point units on the chip. Building floating point arithmetic units from logic requires large amount of resources and, until recently, was not practically viable; this is mostly not true any longer. Even though FPGAs still do not contain any hardwired floating point units, they contain large amounts of logic resources and hardwired multipliers that can be used to build efficient floating point arithmetic units. Moreover, FPGA vendors provide customized IP cores for commonly used floating point arithmetic operations. As a result, modern FPGAs boast of high floating point capabilities, with the latest chips achieving up to 200 GFLOPs peak for single precision floating point arithmetic. This, along with the very high utilizations and the immense flexibility with respect to the selection of the computational cores and the data communication among them, makes them suitable for accelerating a wide variety of floating point intensive computations.

5.2.1 Electrostatics Energy Evaluation on FPGAs: Overview

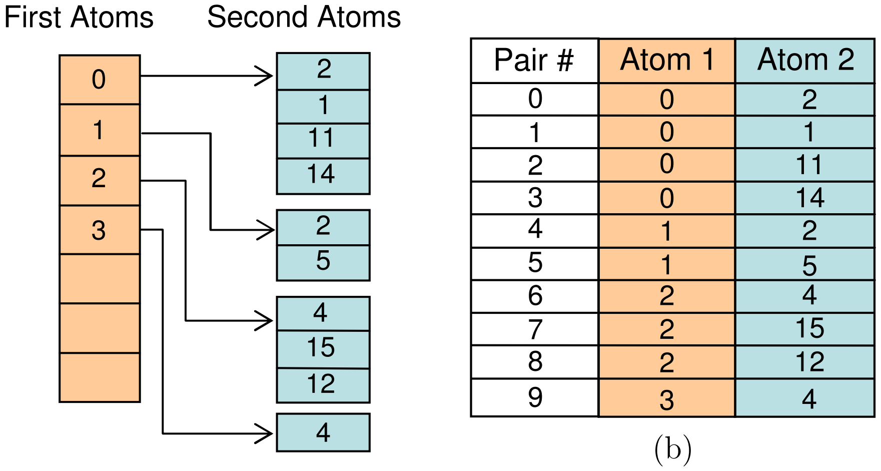
Evaluating the electrostatic energy of the system being simulated requires computing the sum of the electrostatic energies of all the atoms in the system, each of which is a sum of the contributions due to all the neighboring atoms. The main computation, thus, is repeated evaluation of interactions between pairs of atoms. In the serial FTMap program, the atoms are arranged in a neighbor list format (see Figure 5·1(a)). The program cycles through the neighbor list of each atom and computes the partial energies of the two atoms. Since the ACE electrostatic model implemented by FTMap is a sum of two terms (self energy term and generalized Born (g-B) term) and since the second term depends on the total accumulated value of the first term, the FTMap program cycles through the neighbor lists twice.

The above computations are mostly serial due to the accumulation of the partial energies of each atom into the corresponding total value. On the FPGA, however, these can be performed by streaming the atom pairs through deep pipelines and performing the accumulation at the end of the pipeline. This effectively results in many pairs being evaluated in parallel, though the accumulation is still being done in a serial order.

We divide the above computations among two separate FPGA pipelines, one for computing the self energy and the other for computing the pairwise energies (generalized Born). These pipelines can process atom-pairs in streaming fashion, generating and updating two energy values per cycle. Note that due to the dependencies, these two computations cannot be pipelined together and the first pipeline must complete processing all the atom pairs before the second one starts.

Data Structure on FPGAs: Pairs-list

The neighbor lists data structure used by the serial FTMap program is shown in Figure 5·1(a). A neighbor list essentially represents a pointer to a list of atoms. To better map this data structure to the FPGA's block RAMs, we replace the two dimensional structure of the neighbor lists with a one dimensional pairs-list, as shown in Figure 5·1(b). The pairs list, as the name suggests, is a list of atom-pairs, each containing the indices of the two atoms involved in the pair. These pairs can now be streamed through the FPGA pipelines, one per cycle. In every cycle, a feeder unit reads a pair of indices from the pairs list and fetches the coordinates and other parameters for the two atoms from the block RAMs. These are then fed into the pipelines which generate and update two energy values (for the two atoms in the current pair) per cycle.



Pair #	Atom 1	Atom 2
0	0	2
1	0	1
2	0	11
3	0	14
4	1	2
5	1	5
6	2	4
7	2	15
8	2	12
9	3	4

(a) (b)

Figure 5·1: Data structures: (a) Neighbor-lists used in the serial FTMap code (b) Pairs-list used for the FPGA-accelerated FTMap

Note that flattening the neighbor-list into a pairs-list increases the storage requirements. This can potentially be a problem if the neighbor lists are very dense, with tens of thousands of particles, as is the case for molecular dynamics computations. In energy minimization, however, the neighbor lists are very sparse and flattening does not increase the storage requirements significantly.

5.2.2 FPGA Pipelines for Electrostatics Energy Computation

Basic Design

As stated earlier, we divide the evaluation of electrostatics energy into two FPGA pipelines - one for computing the self-energies of the atoms (the Eself pipeline) and the second for computing the pair-wise interactions (the g-B pipeline). Since the evaluation of the pair-wise interactions requires the individual total self-energy of each atom, the self-energy computations must be finished before the pair-wise computations can begin. This leads to two possible solutions (see Figure 5·2).

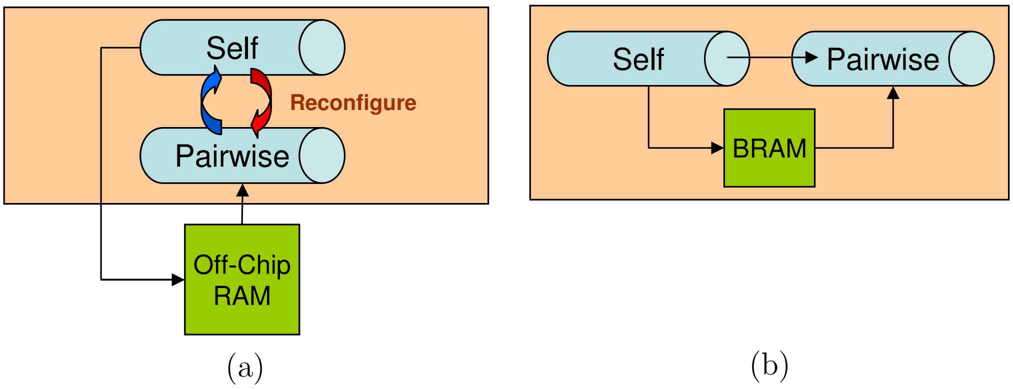


Figure 5·2: Electrostatics computations require two separate pipelines; shown are the two alternatives for organizing these pipelines.

The first is to utilize the entire FPGA for the self-energy pipelines and then reconfigure the FPGA into pair-wise interaction pipelines. The second scheme involves having both the pipelines present on the FPGA at the same time, but starting the pair-wise interaction pipeline only after the self-energy pipeline has finished evaluating the energies of all the atoms. This results, however, in part of the FPGA resources remaining idle at all times.

Though the first scheme yields better utilization of the FPGA resources and allows for more replication of pipeline instances for parallel evaluations, it is not the preferred solution. This is because the time per iteration for the energy evaluation is relatively small (a few milliseconds on serial computer and a few microseconds on an FPGA) and the time required for reconfiguring the FPGA for each iteration becomes a significant fraction of the total time, thus nullifying any performance benefits obtained by better utilization of the FPGA resources. Moreover, having multiple pipelines for the current computation runs into the problem of higher memory port requirements, thus requiring replication of various block RAMs for independent accesses by the different pipelines. This in turn requires broadcasting to multiple BRAMs as well as combining the partial values from those BRAMs, further adding to the complexity of the system.

Having both the pipelines present on the FPGA throughout the computation is thus preferred. The g-B pipeline starts only after the Eself pipeline has finished processing all the atoms. Effectively, this results in two streaming passes through the FPGA to compute the electrostatic energies of all the atoms.

The Self Energy Pipeline

Figure 5·3 shows the FPGA design for computing the self-energies of the atoms. The self-energy pipeline computes the atom self-energies along with some other quantities for updating the forces acting on the atoms.

The self-energy of an atom is the sum of the contributions due to all the neighboring atoms within a cutoff. At the head of the pipeline, a cut-off filter unit checks to see if the two atoms are within a cut-off distance. If so, it notifies the feeder unit so that it can fetch the remaining parameters for the two atoms and feed them to the pipeline.

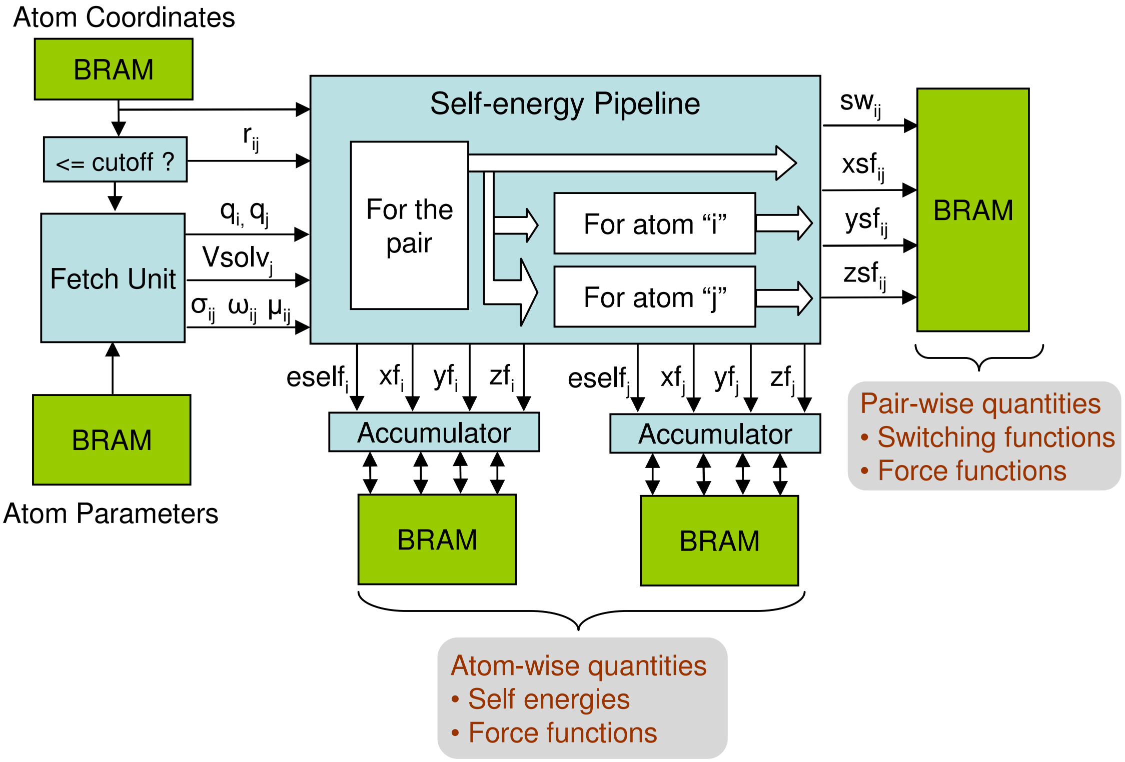


Figure 5·3: Self-energy pipeline on the FPGA; For each input pair, the pairwise values are stored straight into the BRAMs whereas the partial atom-wise values are accumulated into the total values.

In each cycle, the pipeline accepts one pair of atom coordinates and parameters and computes two sets of quantities: pair-wise values and atom-wise values (see Figure 5·3). Pair-wise values are those which are for a particular atom-pair and include a switching function [BBO$^+$83] and its derivative (sw and dsw), and the pair-wise force functions along each of the three axes (xsf, ysf and zsf). The switching function is used to represent the effect of distance dependent dielectric in the electrostatic energy expression. If the distance between the two atoms is larger than a cutoff, the switching function takes a value of 0, reducing the electrostatic energy between the atom-pair to zero. Similarly, the pair-wise force function along each axis depends on the switching function and its derivative as well as the difference of the atom-coordinates along that axis. The resulting forces acting on the two atoms along that axis are equal in magnitude and opposite in direction. The expressions for these various pair-wise quantities and their derivations are shown in Equations 5.1, 5.2 and 5.8.

Atom-wise values are for each atom in the pair and include partial self-energy of the atom (E_{ik}^{self} in Equation 3.6) and partial force-functions along the three axes (xf, yf, and zf) used to update the forces acting on the atom. The expressions for these atom-wise quantities, as evaluated by the self-energy pipeline, are shown in Equations 5.6 and 5.9. These partial values must be accumulated to obtain the total value for each atom. As shown in Figure 5·3, the self-energies and the force functions of the two atoms computed by the Eself pipeline are accumulated by the accumulators. The accumulated values are stored in block RAMs for use by the g-B pipeline. Pair-wise quantities such as switching functions and force functions are also stored in block RAMs and used by the g-B pipeline.

$$sw = const_3(nb_{cutoff}^2 - r^2)^2(nb_{cutoff}^2 - r^2 - 3(nb_{cot}^2 - r^2)) \tag{5.1}$$

$$dsw = const_{12}(nb_{cutoff}^2 - r^2)(nb_{cot}^2 - r^2) \tag{5.2}$$

$$exp = \omega_{ik} e^{-\left(\frac{r_{ik}^2}{\sigma_{ik}^2}\right)} \tag{5.3}$$

$$T_1 = \frac{\tilde{V}_k}{8\pi}\left(\frac{r_{ik}^3}{r_{ik}^4 + \mu_{ik}^4}\right)^4 \tag{5.4}$$

$$T_2 = 2sw(exp + T_1) \tag{5.5}$$

$$E^{self} = E^{self} - T_2 \tag{5.6}$$

$$T_3 = sw\left(-8T_1\frac{3\mu_{ik}^4 - r_{ik}^4}{r_{ik}^2(r_{ik}^4 + \mu_{ik}^4)} + \frac{4exp}{\sigma_{ik}^4}\right) - (T_2 \times dsw) \tag{5.7}$$

$$xsf = T_3(x1 - x2) \qquad ysf = T_3(y1 - y2) \qquad zsf = T_3(z1 - z2) \tag{5.8}$$

$$xf = xf - xsf \qquad yf = yf - ysf \qquad zf = zf - zsf \tag{5.9}$$

The computation of the electrostatic energy is generally performed with double precision floating point. On our FPGA Eself pipeline, we use single precision floating point arithmetic. As stated earlier, the floating point datapath is optimized using the Floating Point Compiler from Altera [Alt08]. It generates a 171 stage pipeline running at 125 MHz.

The Generalized Born Pipeline

Figure 5·4 shows the overview of the FPGA design for evaluating the generalized Born expression. The Generalized Born pipeline is used to compute the pair-wise interactions between the atom-pairs and to update the forces acting on them. As in the case of the self-energy pipeline, only those pairs whose distance is less than the cutoff need to be processed.

The Generalized Born pipeline uses atom self-energies generated by the Eself pipeline to compute the Born radius and its derivative associated with each atom (Equations 5.10 and 5.11). These in turn are used to compute the pair-wise interaction energies (Born and Coulomb energies) of the atoms (eBorn and eCoul in Figure 5·4). The expressions for eCoul and eBorn are given by the first and the second term of the g-B equation (Equation 3.8) respectively.

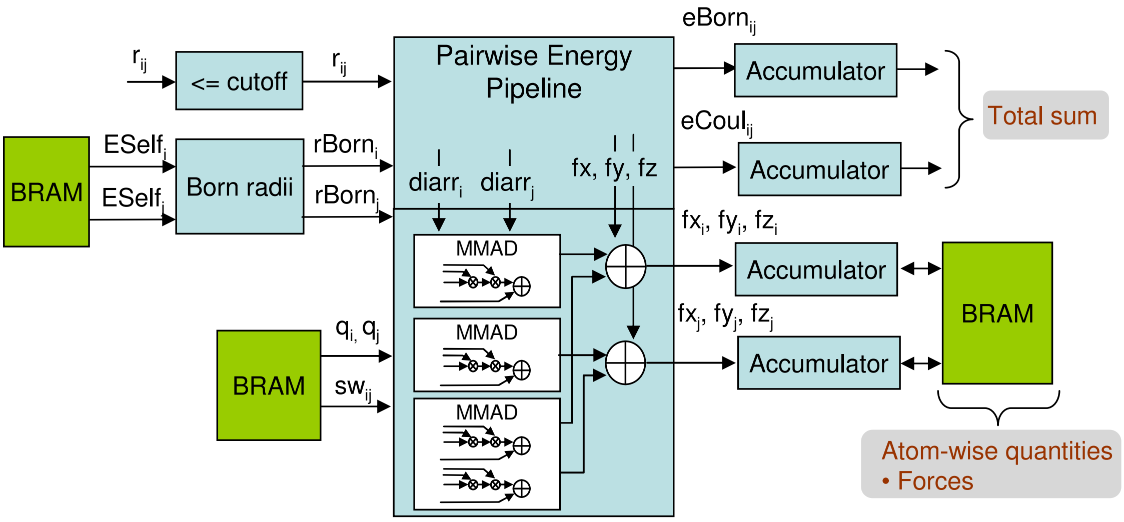


Figure 5·4: Generalized Born pipeline on the FPGA; The self energies are used to compute the atom Born radii which are then fed to the g-B pipeline.

$$rBorn_i = \begin{cases} 1/E_i^{self} & \text{if } E_i^{self} \geq 1/b_0 \\ b_0(2 - b_0 E_i^{self}) & \text{otherwise} \end{cases} \tag{5.10}$$

$$dBrdes_i = \begin{cases} rBorn_i/kE_i^{self} & \text{if } E_i^{self} \geq 1/b_0 \\ b_0^2/k & \text{otherwise} \end{cases} \tag{5.11}$$

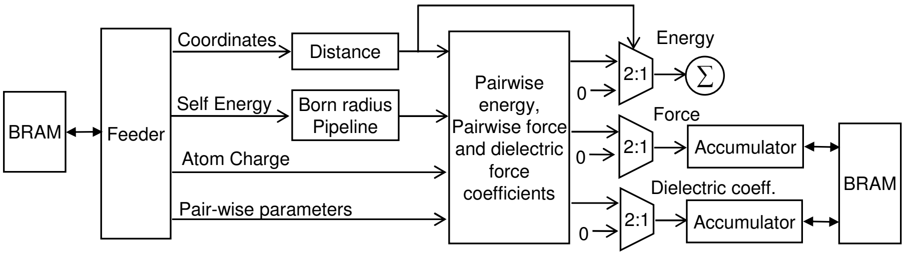
As shown in the figure, the Born and Coulomb electrostatic energies of different atoms are accumulated to generate the total electrostatic energy of the complex. Note that unlike the self-energies, we do not need the individual total Born and

Coulomb energies of each atom. Thus, the partial energy values generated by the g-B pipeline are accumulated into a single total value.

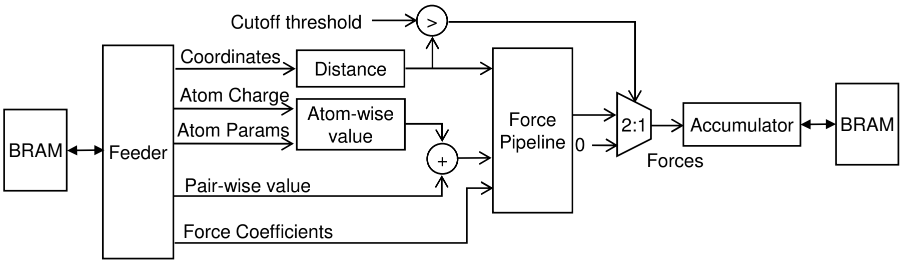
The g-B pipeline also computes the partial forces acting on each atom. These are computed using the pair-wise and atom-wise force functions generated by the self-energy pipeline. These partial forces must be accumulated to compute the total forces acting on each atom, which are then used to update the atom coordinates. To update the positions of the atoms, we need the individual forces acting on each atom; the accumulator thus stores the individual values into the force block RAMs. The total force values, along with the energy gradients, are then used to determine the new atom positions in the complex for the next minimization iteration.

The force expressions evaluated by the generalized Born pipeline contain a double summation. In other words, there is a dependency of some of the force calculations on the total accumulated values of the dielectric force coefficients being computed by the first half of the pipeline. This dependency is similar to the dependency of the g-B pipeline on the self-energy pipeline. Due to this, the g-B computations cannot be fully pipelined. The pairwise energy pipeline is thus split into two parts. The second part starts the computations only after the first pipeline has finished processing all the atoms. As a result of this, computing the electrostatic energies on the FPGA effectively requires three streaming passes through the pipelines.

The first part of the g-B pipeline is shown in Figure 5·5(a). It processes one atom-pair per cycle and computes the pairwise energies, pairwise component of the forces and the dielectric force coefficients for each atom in the pair. These are accumulated into the total values and stored in the FPGA block RAMs. The dielectric force coefficient of each atom also contains a constant value that must be added to the total value. Since a particular atom might appear in multiple pairs, a bit vector is used to ensure that the constant value is added only once.



(a)



(b)

Figure 5·5: Two parts of the generalized Born pipeline [Dai10]; (a) first part processes the atom-pairs to compute the pairwise energies and forces (b) second part processes individual atoms to compute the second component of the force.

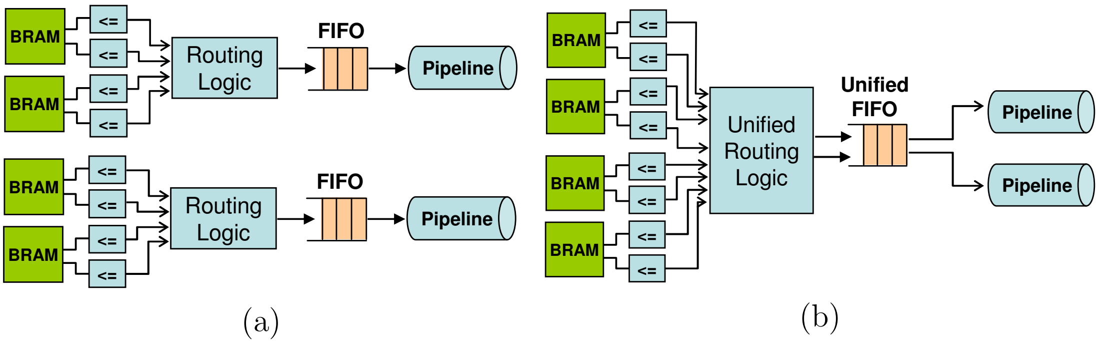
The second part of the g-B pipeline processes one atom per cycle (see Figure 5·5(b)). In each cycle, it reads the accumulated dielectric force coefficient for one atom from the block RAM and computes the total forces acting on the atom.

Like the Eself pipeline, the g-B pipelines also use single precision floating point arithmetic and are generated using the Altera Floating Point Compiler. The compiler generates a 213 stage pipeline for the first part and a 49 stage pipeline for the second part, both running at 125 MHz.

Integrating the Pipelines

Since the evaluation of different atom-pairs is independent of each other, it is desirable to have multiple pipelines that process different pairs in parallel and hence yield better performance. As per our initial synthesis results, we could fit two instances each of the Eself and g-B pipelines on an Altera Stratix III SE110 FPGA, allowing four energy updates per cycle. Below we present the integrated pipeline design with two instances of each of the self-energy and the pairwise energy pipelines. Having multiple pipelines, however, results in some added complexity in data routing as well as block RAM accesses and data storage. These are discussed below.

Feeding the pipelines: Having multiple parallel computation units results in the obvious requirement of load-balancing among them; in this case, routing the atom-pairs to the different pipelines. This routing becomes even more important if the computations are performed conditionally, leading to non-uniform flow of the inputs. As stated earlier, only those atom pairs that pass the cutoff distance test need to be processed by the Eself and g-B pipelines.



(a) (b)

Figure 5·6: Routing of atom pairs to multiple pipelines (a) separate routers and FIFOs (b) unified routing.

Thus, in order to feed the pipelines with one "valid" atom-pair per cycle, we need multiple cutoff filter units for each energy pipeline. This results in two complications. First, each cutoff filter requires two sets of atom-coordinates per cycle. These can be read from a single dual-ported block RAM. For multiple filters, we thus need to replicate the BRAMs, one for each filter. Second, there are multiple filters and each can potentially output a valid atom-pair every cycle. Since the energy pipeline can only process one of those per cycle, we need to add a mechanism for routing these pairs and storing them in a FIFO and possibly stalling the pipeline if the FIFO gets filled up. There are two possible routing solutions to this. In the first, there can be two separate sets of filters and FIFO combination, one for each of the pipeline instance. Each pipeline then gets fed from a separate FIFO (see Figure 5·6(a)). Another solution is to have a unified filter unit that reads multiple atom-pairs per cycle and outputs at most two valid pairs per cycle; the rest are stored in a unified FIFO (see Figure 5·6(b)). The advantage of this scheme over the first is that it results in better load-balancing and potentially fewer stalls during very random distribution of valid pairs in the input stream.

Overall, both these schemes result in increased complexity of the control logic and increased block RAM requirements. In order to avoid these pipeline complications, one possible solution is to process all of the atom-pairs present in the list without filtering with respect to the cutoff distance. Note that during the initial creation of the neighbor-lists on the host, a cut-off test is performed. Only those atom-pairs which are within a cutoff are added to the list. A second cutoff test is required on the FPGA to check for atoms that might have moved out of the cutoff radius due to position updates during minimization iterations, and to check for short-range cutoff. The change in position during minimization steps, however, is not radical [BBO^{+}83]. Due to this, most of the atoms are likely to pass the cutoff test on the FPGA. We noticed that, on average, only about 7-8% of the pairs fail the test. Moreover, since the total number of atom-pairs to be processed is only a few thousand (unlike molecular dynamics where millions of pairs are screened) unconditionally processing all the atoms does not lead to a huge amount of wasted work. We, thus, unconditionally process every atom-pair. A cut-off filter is still needed at the head of the energy pipeline to invalidate the energy and force updates at the end of the pipeline if the atom-atom distance is larger than the cutoff. This can be viewed as inserting a "bubble" into the pipeline whenever the distance is larger than the cutoff.

Block RAM requirements for the FPGA design: Figure 5·7 shows the integrated design for the electrostatic energy computations on the FPGA, with two instances each of the self-energy and pair-wise interaction pipelines.

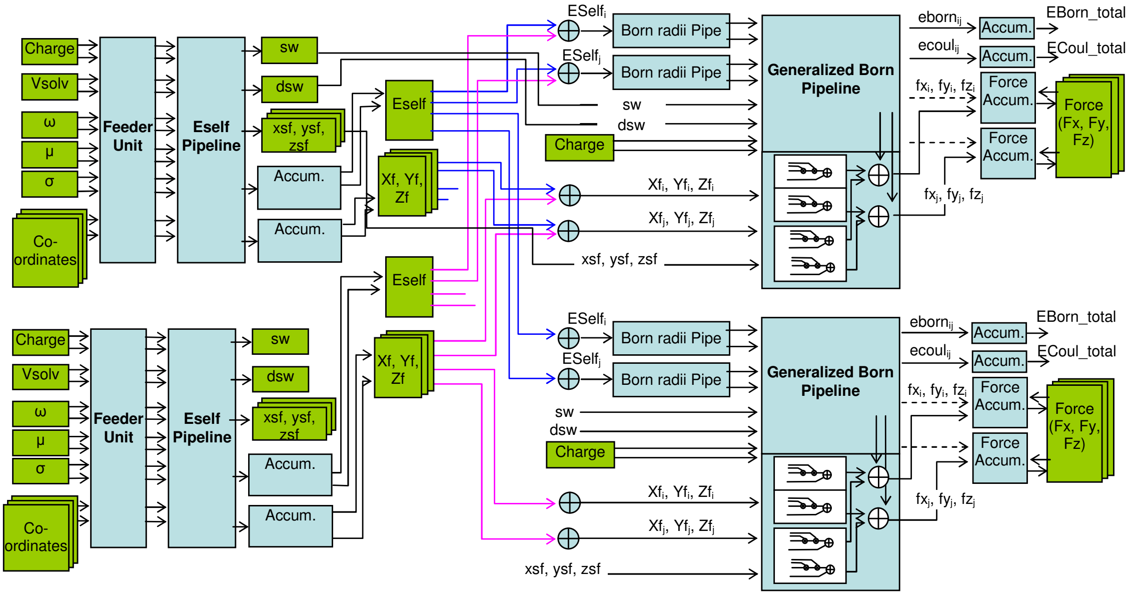


Figure 5·7: Complete electrostatics pipeline with two Eself and two g-B pipelines.

Two atom pairs are fetched from the pairs-list (of Figure 5·1(b)) in each cycle and one pair is fed into each of the pipeline instances. In order to feed these pipelines in parallel, some of the block RAMs must be replicated. Different block RAMs used in the design are shown as shaded blocks. The block RAMs for the atom coordinates and the other atom-wise and pair-wise parameters (atom charge, solvation volume, ω, μ, σ) are replicated to feed the two pipelines. The pair-wise parameters (ω, μ,

σ) are not symmetric and hence two sets of these are required per cycle. This is done by utilizing the dual-ported capability of the block RAMs. Similarly, atom charges and coordinates are replicated into two sets of dual-ported block RAMs, one for independently feeding each of the Eself pipelines.

Each of the Eself pipeline generates and stores self-energy and force function values (E^{self}, Xf, Yf, Zf) into its own set of block RAMs. An independent set of BRAMs is required for each pipeline since each performs two updates per cycle and FPGA block RAMs provide only two read/write ports. The force and energy values in the block RAMs of the two pipelines are, thus, partial and must be combined before being fed into the g-B pipeline. As shown in the figure, this can be done on the fly. Partial values from the two BRAMs are fetched and added and the sum is then fed into the g-B pipeline.

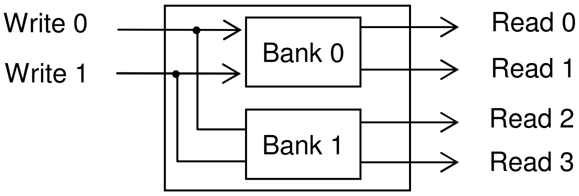


Figure 5·8: 4-port BRAM constructed by replicating dual-ported BRAMs.

The g-B pipeline requires two atom-wise quantities (E^{self}, Xf, Yf, Zf) per cycle, each of which is a sum of the values from the two sets of BRAMs. Thus, each g-B pipeline needs to access two locations in each BRAM. This results in 4 reads per cycle for each of the BRAMs. Since four read ports are not supported by FPGA block RAMs, we implement this by further replicating each of the block RAMs into two banks (see Figure 5·8). Each RAM requiring four read ports is implemented by instantiating two banks of dual ported RAMs. Writes are broadcast to both the banks and two reads are provided by each of the banks.

For the pair-wise quantities generated by the two Eself pipelines (sw, dsw, xsf, ysf, zsf), we could use a unified BRAM since each pipeline updates only one set of values per cycle. As shown in Figure 5·7, we still utilize independent sets of BRAMs for each pipeline. This is done due to two reasons. First, this allows the use of simple dual-ported block RAMs. Second, even though the pair-wise quantities are stored by the two pipelines in separate BRAMs, they need not be combined before being fed to the g-B pipeline. This is because these values are specific to a particular atom-pair and the distribution of pairs among the pipelines is the same in both the Eself and g-B pipelines. That is, if a particular pair is processed by the first Eself pipeline, it will always be processed by the first g-B pipeline. Finally, the total forces generated by the g-B pipeline are stored in dual-ported block RAMs. Again, each g-B pipeline has its own set of independent force BRAMs and updates two force values per cycle. These two sets of BRAMs are then combined to generate the total forces which are used to update atom positions for the next iteration.

Fixed Point Accumulation

As noted above, the energy and force values generated by the pipelines are partial and are accumulated into the total values. Since the generated values are in IEEE-754 single precision format, their accumulation poses some difficulties.

Since one partial energy value is generated per cycle, accumulation operation must either complete in one cycle or must be pipelined. Performing a single cycle floating point accumulation results in very poor operating frequencies. Pipelining the accumulation, however is not straightforward. This is because multiple partial values for a particular atom might be generated in consecutive cycles. This results in there being a need to accumulate on a value that is still in the pipeline. This is similar to a data hazard in a standard pipeline with a feedback and results in an erroneous total value. Though there have been some studies in the implementation of pipelined floating point accumulators on the FPGAs [SZ09, NZB09], those schemes are not suitable for the current work. We address this issue by converting the values into 64 bit fixed point representation before the accumulation. Accumulation is then done in fixed point and can be performed in a single cycle. The use of wide fixed point data format ensures that there is no loss of precision.

5.3 Algorithms for GPU Acceleration of Energy Minimization

Even though FPGA accelerated energy minimization results in very good performance speedups, the motivation for accelerating this using GPUs is three-fold: fast integration into the production system, comparatively easy modifications to accommodate any future changes to the energy model, and lower cost of ownership. The first two of these result from the availability of high level GPU programming languages and standard interface APIs such as NVIDIA CUDA and OpenCL. These are C-like programming languages wherein the computation is specified in a manner similar to standard C, thus enabling fast development and easy integration of the accelerated code into the rest of the code. Further, this allows the computational biologists to modify the code without having to interact with the hardware designer or the developer of the accelerator. For FPGAs, such standard APIs are mostly unavailable and programming is usually done in hardware description languages, making it hard to make even the slightest change.

The lower cost of ownership of the GPU solutions is a direct consequence of the volume of chips manufactured and sold. GPUs are at the heart of the multi-billion dollar gaming industry. They, thus, enjoy a very big market, driving the cost very low. FPGAs, on the other hand, form a very niche market, particularly for high performance computing, resulting in higher cost per chip.

Thus, to enable fast and more cost-effective desktop solution for energy minimization, we investigated the use of the GPUs. Below we present the algorithms for the GPU-accelerated energy minimization, as used in the FTMap program. Like in the case of the FPGA-accelerated energy minimization, evaluation of the bonded interactions is still performed on the host computer. In addition to the electrostatic interactions, however, here we also perform the van der Waals energy evaluations on the GPU, thus accelerating the entire non-bonded energy evaluations. This constitutes almost 99.8% of the total energy evaluation time. The rest 0.2% is the bonded interactions.

5.3.1 Energy Minimization on GPUs: Overview

The computations to be performed per iteration of energy minimization can be divided into six tasks: (i) computing the self-energies for all the atoms, (ii) computing the pairwise interactions energies, (iii) computing the van der Waals energies, (iv) computing the energy gradients (v) updating the forces acting on the atoms, and (vi) performing the optimization move and updating the atom-coordinates based on the force values. The original FTMap program performs these tasks using separate function calls. Since the GPU kernel launch has some overhead, and since some of the computations are common across multiple tasks, we combine them where possible.

Since the evaluation of the pairwise interactions requires the knowledge of the total self energies of all the atoms, these two computations cannot be combined. We therefore divide the six tasks into three GPU kernels: (a) computing the atom self energies and the corresponding energy gradients, (b) computing the pairwise interactions (which is a part of the electrostatic energy) and the van der Waals energies along with the energy gradients, and (c) updating the atom forces. The computation structures used by these kernels are similar and the techniques discussed here apply to all of these computations. Two computations - the optimization move and the atom-coordinate updates, are left on the host as they form a very small fraction of the total minimization runtime.

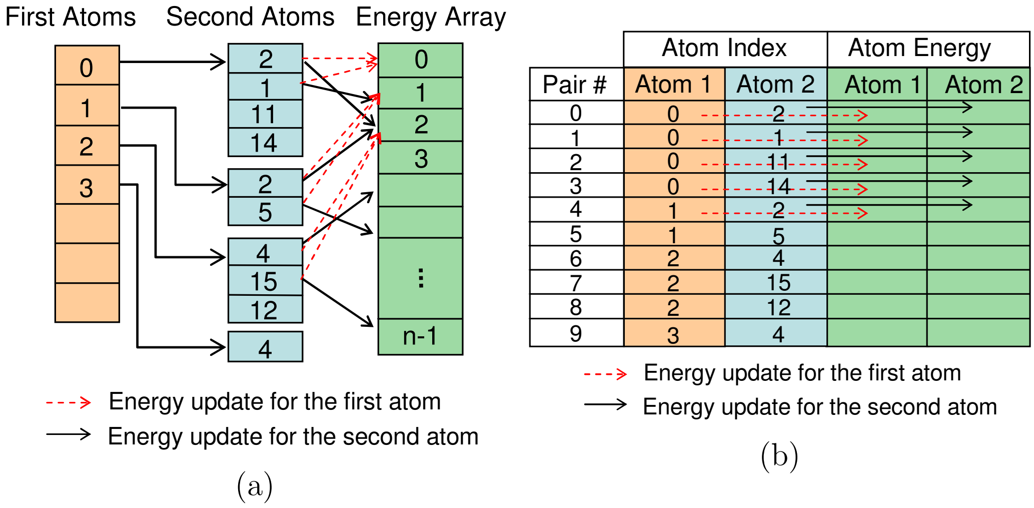
The main difficulty with accelerating the energy minimization computations is the small runtime per minimization iteration and a significant portion of it being serial accumulations. On the FPGA, we could address this by streaming the atom-pairs through very deep pipelines. This scheme, however, is not applicable on the GPUs since the GPU processors and communication structure is fixed and cannot be customized for the computation at hand. Even though GPUs contain hundreds of processor cores that can perform energy computations in parallel, the accumulation of these partial energies leads to serialization. Moreover, these accumulations must be performed in the slow global memory. This, combined with the restrictions in the synchronization of threads across different GPU multiprocessors makes it difficult to achieve good performance. We address this by restructuring the original neighbor-lists data structure and by introducing a new data structure.

Below we present three different schemes for mapping the energy computations onto the GPU processors. The first scheme uses the original neighbor-lists data structure and does not result in any performance improvements. We discuss the reasons for the difficulties in obtaining good performance from this and propose a modified data structure. The second scheme uses this modified data structure, leading to improved distribution of work among the GPU threads and better performance. This scheme, however, still requires serialized accumulations in the global memory and thus results in modest overall speedup. In the third scheme, we use a new data structure that enables efficient work distribution and multiple parallel accumulations from the shared memory, resulting in significant performance improvements.

5.3.2 Energy Minimization on GPUs: Initial Attempts with the Neighbor-lists

Difficulties with the Neighbor-lists

Serial FTMap uses neighbor lists to cycle through different atom-pairs and compute the partial energies. These partial energies are accumulated, as they are computed, into an array storing the total individual energy of each atom (see Figure 5·9(a)).



(a) (b)

Figure 5·9: Data structures on the GPU: (a) Neighbor-lists lead to write conflicts (b) Pairs-list enables parallel writes.

There are several reasons why the neighbor-list structure is not suited for mapping to the GPUs. First, we need the individual total self energies of all the atoms, not just the total self energy of the system. This requires multiple accumulations, one for each entry of the energy array. Second due to the random occurrences of the "second" atoms in the neighbor-lists, the energy array cannot be distributed into the shared memories of different GPU multiprocessors. Rather, it must be present in the GPU global memory, accessible from all the multiprocessors. And third, having the energy array in the global memory can potentially lead to write conflicts, since a particular "second" atom can be present in the neighbor-lists of more than one "first" atom (e.g. atom number 2 in Figure 5·9(a)).

Mapping the Neighbor-lists to the GPU Threads

We now present our first scheme that maps the neighbor-lists data structure onto the GPU threads. Though there are various ways to map this neighbor-list computation structure onto the GPU threads for parallel energy evaluations, most of them run into two serious problems: (i) memory conflicts during parallel updates from different threads and (ii) serialization during the accumulation of the partial energies into the energy array.

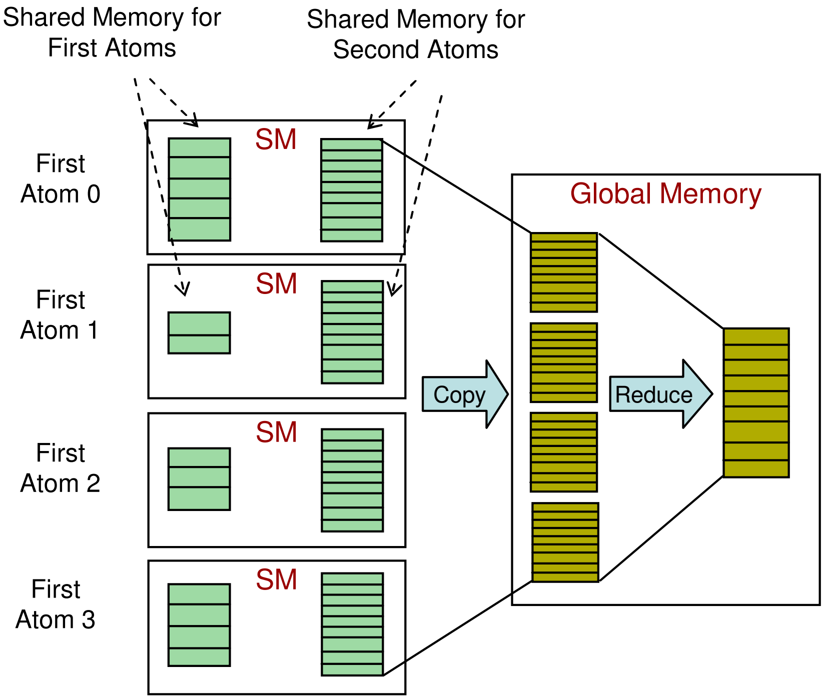


Figure 5·10: Mapping the neighbor-lists onto GPU threads. Replicating the energy arrays to enable parallel updates.

To enable parallel updates and accumulations on different GPU multiprocessors, we map only one “first” atom from the neighbor-lists onto a multiprocessor at a time. On each multiprocessor, we have two different energy arrays in the shared memory. The first array stores the partial energies of the currently mapped first atom, with one entry for its partial energy due to each second atom in its neighbor-list. The second array is to store the partial energies of the second atoms, with one entry for each atom in the system (Figure 5·10).

Different threads of a thread block compute the partial energy of the current atom due to one of the second atoms in its neighbor-list and that of the second atom due to the first. As the energies are computed by different threads, they are updated in these shared memory arrays. Note that since a second atom will appear in the neighbor list of a particular atom only once, no two threads will update the same shared memory location at the same time. This enables parallel, conflict free updates.

Once the entire neighbor-list of the current first atom is processed, a barrier is reached. A master thread (thread 0 from each block) then computes the total energy of the first atom by adding all the partial values in the first atom energy array. The energies in the second atom array, however, are for different second atoms and are only partial. As shown in Figure 5·10, analogous partial arrays are present on the shared memories of all the other multiprocessors. These must be combined to obtain the total energy of each of the second atom. This is done by copying the second atom arrays from the shared memories of the different multiprocessors to the global memory. The corresponding values from these arrays are then added to obtain a single array with the total energies.

Though this scheme allows parallel execution and updates, it has three problems. First, since only one “first” atom is processed by a multiprocessor, the GPU threads are heavily underutilized and the distribution of work on different multiprocessors is uneven. This is because the distribution of the atoms in the neighbor-lists is very non-uniform, with different “first” atoms having widely varying number of “second” atoms in their neighbor-lists, ranging from a few to a few hundred. Second, transferring multiple large second atom energy arrays from the shared to the global memory incurs high data transfer cost per iteration. Finally, accumulation from the global memory is slow. Overall this method results in poor performance and is not preferred.

5.3.3 Improved Data Structures for Energy Minimization on GPUs

Since the computation per iteration is very small, only a few milliseconds on a serial computer, obtaining high speed-up requires efficient distribution of work to maximize parallelism and reduce the communication cost. We now discuss two schemes that enable this. For efficient mapping of the work on the GPU threads and to enable conflict-free parallel energy updates, both these schemes use a modified data structure.

In the first scheme, we replace the neighbor-list structure with a pairs-list similar to the one used in the FPGA-accelerated energy minimization. It contains a list of atom-pairs that need to be processed, along with additional separate columns for storing the partial energies of the two atoms. This is shown in Figure 5·9(b). Different atom-pairs in this list are independent of each other and can be processed in parallel. We distribute these pairs equally among the different GPU threads. The pairs-list is stored in the GPU global memory. Each thread processes the pair assigned to it and stores the partial energies of the two atoms at the corresponding index in the pairs-list.

Once all the pairs have been processed, we accumulate these partial energies to compute the total energy of each individual atom. This needs to be done serially, mainly due to the unordered occurrences of the second atoms in the pairs-list.

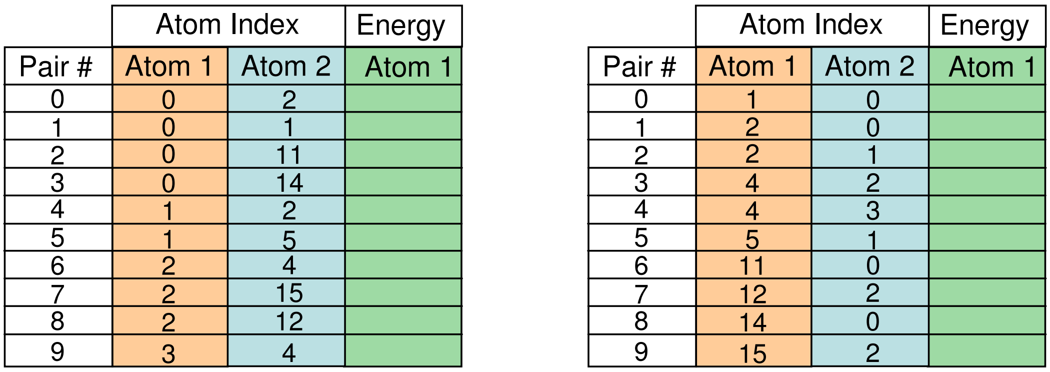
There are two alternatives for performing this serial accumulation: on the GPU using a single thread or on the host. On the GPU, since the energy values are stored in the global memory, accumulation requires multiple accesses to the slow global memory. Since the accumulation is done serially by a single thread, it turns out that the accumulation on the host outperforms the GPU. Accumulation on the host, however, requires transferring the two arrays of atom-energies from the GPU to the host memory in each iteration. Due to the relatively small amount of computation per iteration, the time for this transfer is actually larger than the computation time on the GPU.

Clearly, the limitation of the current scheme is the serial accumulations. Though it enables uniform distribution of work, parallel energy computations and conflict-free parallel updates, serialization during the accumulation of the partial energies limits the overall performance. With the accumulations performed on the host, this scheme results in an overall speedup of around 3x over the original serial code.

Split Pairs-lists and Static Mapping

To enable faster and parallel accumulations from the GPU shared memory, we further modify the data structure used in the previous scheme. In our second approach, we still use the pairs-list but make two changes in how the pairs get mapped to the GPU threads.

The first change is to split the pairs-list into two separate pairs-lists. Notice that the serialization during the accumulation is mainly due to the random occurrences of second atoms in the neighbor-lists (now the pairs-list). The first atoms still appear in an ordered fashion. Thus, to add determinism in how the atoms appear in the list, we split the lists into two separate lists and process each one separately.



	Atom Index		Energy
Pair #	Atom 1	Atom 2	Atom 1
0	0	2	
1	0	1	
2	0	11	
3	0	14	
4	1	2	
5	1	5	
6	2	4	
7	2	15	
8	2	12	
9	3	4	

	Atom Index		Energy
Pair #	Atom 1	Atom 2	Atom 1
0	1	0	
1	2	0	
2	2	1	
3	4	2	
4	4	3	
5	5	1	
6	11	0	
7	12	2	
8	14	0	
9	15	2	

Figure 5·11: Split pairs-lists: (Left) Forward list, (Right) Reverse list.

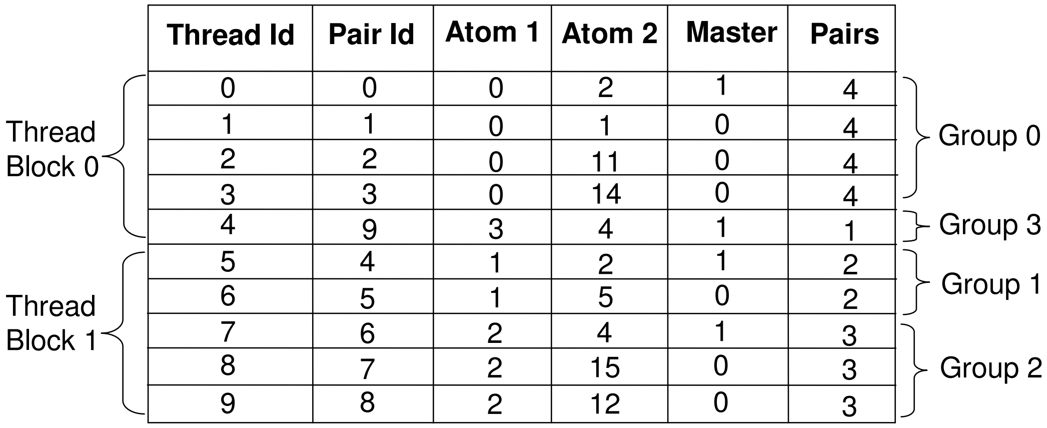
The first pairs-list is based on the original neighbor-list and is called the forward list. The second list is generated by reversing the original neighbor-list, i.e., by treating each second atom of the original neighbor list as a first atom for the reverse neighbor list. We call this second list the reverse list. While processing a list, only the energy of the first atom in each pair is computed and updated. This way, the energies of the first atoms (in the original list) get updated while processing the forward list and those of the second atoms (in the original list) while processing the reverse list. This is shown in Figure 5·11. Note that there is no column for storing the energies of the second atom in the pair.

The second modification involves statically mapping the pairs from the new pairs-lists onto the GPU threads. This comes from the observation that the pairs in the new lists can be grouped by the first atoms. This can be done since we now care only about computing the energies of the first atoms in the pair and not the second atoms.

These two changes allow better and more uniform distribution of atom-pairs on the GPU and enable parallel and much faster accumulations in GPU shared memory, as discussed next.

Assignment Tables for Static Mapping

Once we have the forward and reverse pairs lists, we statically distribute them to the GPU threads running on different multiprocessors. The central idea is to have all the partial energies that need to be accumulated together be computed on the same multiprocessor. The static mapping scheme does this by grouping together all the pairs in a list having the same first atom and maps the entire group onto the threads in the same thread block. Having all the pairs of a group mapped to the same thread block allows us to perform the accumulations in the shared memory, since all the partial energies are present within the same multiprocessor. Moreover, multiple parallel accumulations can be performed; within each multiprocessor, one of each of the group mapped to the thread block as well as on different multiprocessors.



Thread Id	Pair Id	Atom 1	Atom 2	Master	Pairs
0	0	0	2	1	4
1	1	0	1	0	4
2	2	0	11	0	4
3	3	0	14	0	4
4	9	3	4	1	1
5	4	1	2	1	2
6	5	1	5	0	2
7	6	2	4	1	3
8	7	2	15	0	3
9	8	2	12	0	3

Thread Block 0
Thread Block 1
Group 0
Group 3
Group 1
Group 2

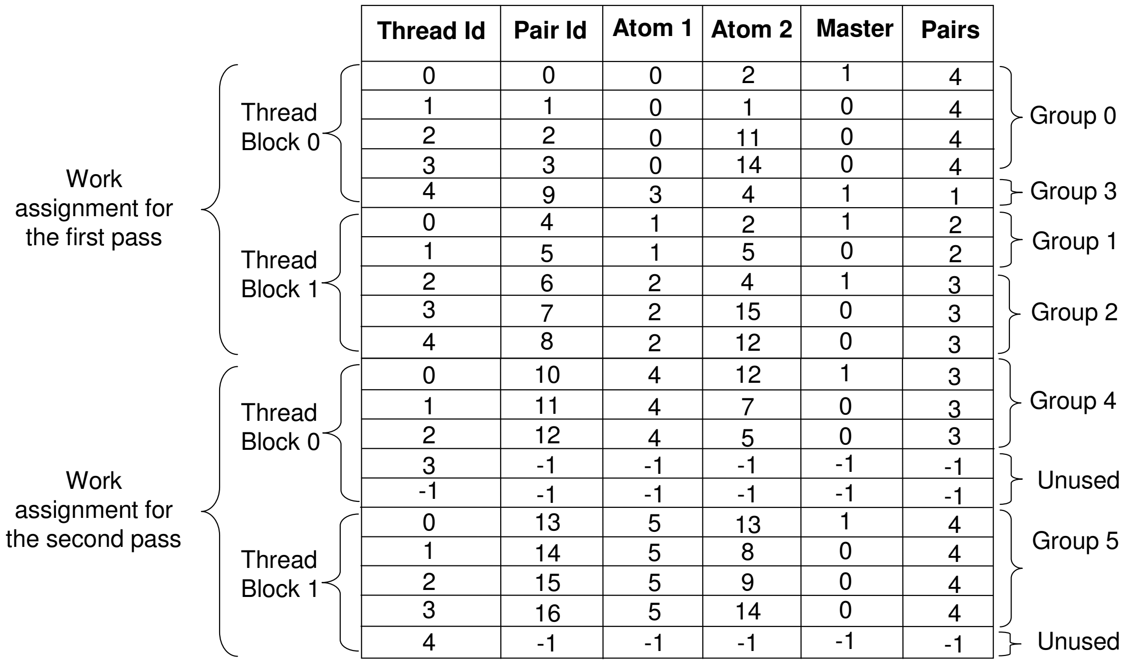
Figure 5·12: Assignment table for the GPU; atom-groups are mapped to the GPU thread-blocks in their entirety.

To determine this static assignment of work among the GPU threads, we generate a new data-structure called the assignment table. It is shown in Figure 5·12. The table contains one (or multiple) row per thread id; each row contains 5 fields: pair id, indices of the two atoms, a master field indicating if this thread is the first thread for this pairs-group, and the number of pairs in the group. The master thread field and the number of pairs in the group are used to accumulate the energies of the atoms in the shared memory.

The assignment table essentially tells which thread must work on exactly which atom-pair and which threads are responsible for the accumulations. Two assignment tables are generated, one for each of the forward and the reverse pairs-list. The generation of the assignment tables is a preprocessing step and is performed on the host computer. These assignment tables are then transferred to the GPU and stored in the GPU global memory. Note that this is done only once, at the beginning of the minimization process. There is no further data transfer per iteration, unless the neighbor list is updated, in which case we regenerate the assignment tables and transfer them to the GPU. This happens only a few times per 1000 minimization iterations; thus the preprocessing and transfer time is negligible.

As shown in Figure 5·12, the groups of pairs are mapped to the thread blocks in their entirety. More than one group can be mapped onto a particular thread block, provided there are enough threads to accommodate all the pairs of those groups. If the current thread block does not have enough threads left to accommodate the entire group, it is mapped onto the next available thread block (e.g. group 1). Unused spaces on the thread blocks are claimed by other smaller pair-groups (e.g. group 3).

In case the number of groups are larger than what can be mapped on the available thread blocks, the computation is performed in multiple passes, with the threads being responsible for more than one group of pairs. This is shown in Figure 5·13. This leads to more than one row per thread in the assignment table.



Thread Id	Pair Id	Atom 1	Atom 2	Master	Pairs
0	0	0	2	1	4
1	1	0	1	0	4
2	2	0	11	0	4
3	3	0	14	0	4
4	9	3	4	1	1
0	4	1	2	1	2
1	5	1	5	0	2
2	6	2	4	1	3
3	7	2	15	0	3
4	8	2	12	0	3
0	10	4	12	1	3
1	11	4	7	0	3
2	12	4	5	0	3
3	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
0	13	5	13	1	4
1	14	5	8	0	4
2	15	5	9	0	4
3	16	5	14	0	4
4	-1	-1	-1	-1	-1

Work assignment for the first pass
Thread Block 0
Thread Block 1
Work assignment for the second pass
Thread Block 0
Thread Block 1
Group 0
Group 3
Group 1
Group 2
Group 4
Unused
Group 5
Unused

Figure 5·13: Multi-pass assignment table; threads process pairs from different groups in different passes. Some of the threads may remain idle during some of the passes (shown with -1 in the corresponding rows).

Energy Computation using the Assignment Table

The GPU threads work in parallel on the different rows of the assignment table. Each thread works on the pair assigned to it in the assignment table, computing the energy of only the first atom. Energies computed by different threads are stored in an array in the GPU shared memory. The length of this array is equal to the number of threads in the thread block, with each thread storing the computed energy at the index equal to its local thread id (id within the block).

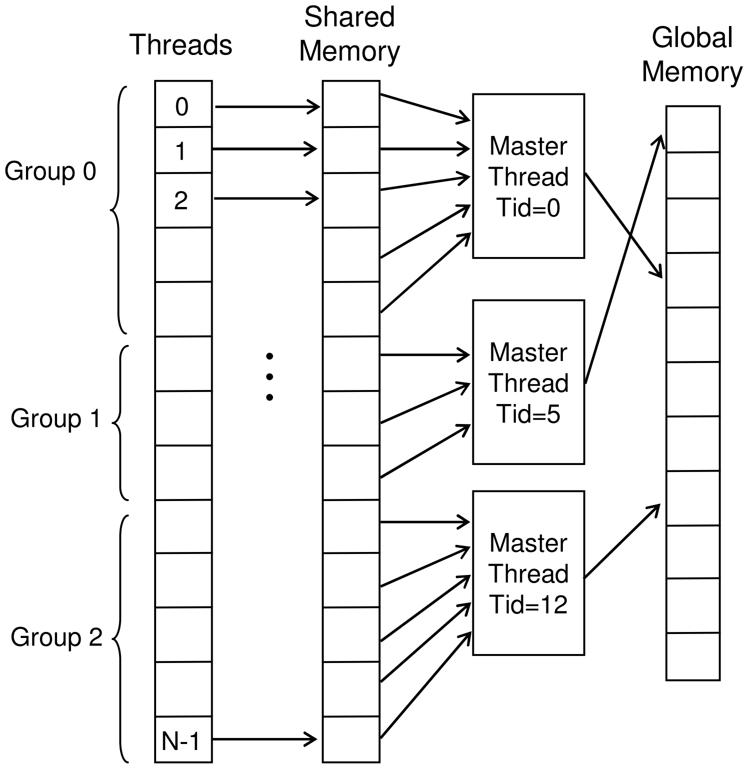


Figure 5·14: Threads compute partial energies in parallel and store in the shared memory. Master threads then perform the accumulations from the shared memory and store the accumulated value in the global memory.

Once all the threads finish processing their assigned pairs, the master threads execute the accumulation round (see Figure 5·14). Each master thread reads the number of atoms for the group associated with it and accumulates that many values from the shared memory, starting from its local thread id. This way, many threads perform accumulations in parallel and from the shared memory, resulting in significant speedup compared to the previous schemes. The master threads then store the accumulated values in the GPU global memory. If multiple passes are required, the above process is repeated for each pass.

Note that we can use this scheme only because we are computing and updating the energies of only the first atom. For the second atom, we repeat this process with the assignment table corresponding to the reverse pairs-list.

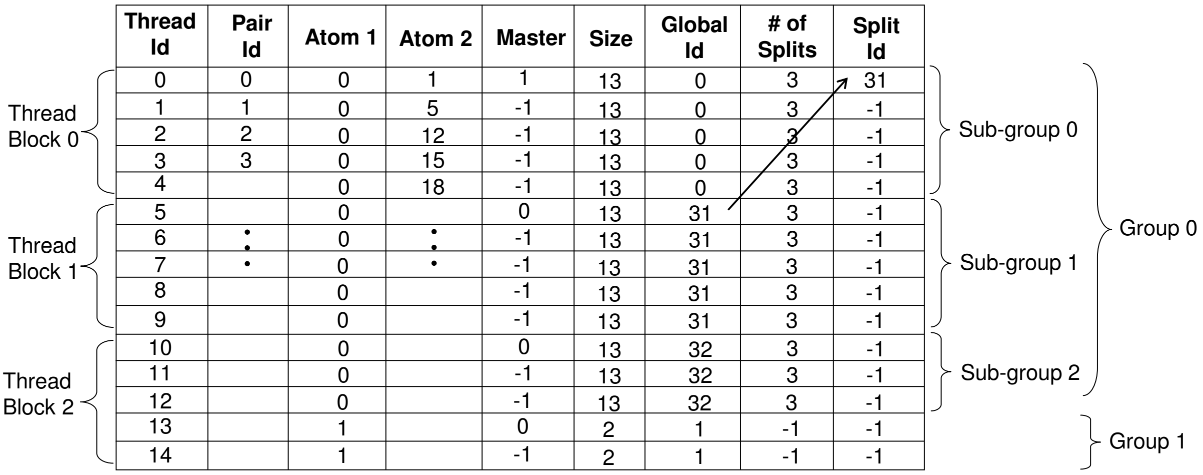
Processing the forward and reverse lists separately leads to repeating some of the computations. This can be avoided by storing those values in the GPU global memory during the kernel for the forward list and reusing them in the kernel for the reverse list. This, however, results in a slowdown due to the slower global memory access. Contrary to a previous case where accessing the global memory was preferred over multiple accesses to the faster shared memory, this is an example where redundant computation on the GPU is preferred as opposed to accessing the slow, global memory.

5.3.4 Supporting Larger Neighbor-lists and Double Precision Arithmetic: Modified Assignment Table

As stated above, the main purpose of the static assignment scheme is to ensure that all the pairs from the pairs-list that have the same first atom (i.e. the entire group) are mapped to the same GPU thread-block. This allows for efficient accumulations. This, however, puts a limit on the size of the neighbor lists. Since each second atom in the neighbor-list of a particular atom contributes to one pair in the group and since the above scheme requires one thread for processing each pair in the group, the maximum number of atoms in the neighbor-list of any atom is equal to the number of GPU threads per block.

Currently, CUDA allows a maximum of 512 threads per block. The actual number of threads per block is limited by the resources used by each thread. The neighbor-lists used in energy minimization are usually very sparse. For all of our test cases, the neighbor lists are smaller than 500. Moreover, with single precision floating point arithmetic, we can launch the kernel with 512 threads, thus accommodating all the groups. There can, nevertheless, be cases where the neighbor lists are larger than 512. Further, for energy minimization computations with double precision, the kernel resource requirements (registers and shared memory) become very large, limiting the number of threads per block to 400. This leads to situations where not all the second atoms of a first atom can stay on the same multiprocessor and the assignment table scheme discussed above cannot be applied.

To overcome the limitation on the size of the neighbor lists and to support the energy evaluations with double precision floating point arithmetic, we use a scheme for splitting the groups. Here, each group having more pairs than the number of threads per block is split into multiple sub-groups, each with its own split “pseudo” first-atom. These sub-groups can now be mapped to separate multiprocessors, as separate groups. The groups with pairs fewer than or equal to the number of threads per block remain unchanged and are mapped as before. The computation of the partial energies and the accumulation within each (sub)group is done as before. This is followed by an extra round of accumulation to gather the partially-accumulated energies from the sub-groups and compute the total energy. This is done using a separate GPU kernel.



Thread Id	Pair Id	Atom 1	Atom 2	Master	Size	Global Id	# of Splits	Split Id
0	0	0	1	1	13	0	3	31
1	1	0	5	-1	13	0	3	-1
2	2	0	12	-1	13	0	3	-1
3	3	0	15	-1	13	0	3	-1
4		0	18	-1	13	0	3	-1
5		0		0	13	31	3	-1
6	⋮	0	⋮	-1	13	31	3	-1
7		0		-1	13	31	3	-1
8		0		-1	13	31	3	-1
9		0		-1	13	31	3	-1
10		0		0	13	32	3	-1
11		0		-1	13	32	3	-1
12		0		-1	13	32	3	-1
13		1		0	2	1	-1	-1
14		1		-1	2	1	-1	-1

Thread Block 0
Thread Block 1
Thread Block 2
Sub-group 0
Sub-group 1
Sub-group 2
Group 0
Group 1

Figure 5·15: Assignment table to support group-splitting; Group 0 has 13 pairs and there are only 5 threads per block – it is thus split into 3 sub-groups. With the number of atoms in the system = 30, the Global Ids start from 31. The second-level master (with Master = 1) has the corresponding Split Id = 31. Other master threads (Master = 0) do not perform second level accumulations. (The table shown is just an example; actual assignment tables contain tens of thousands of rows).

The modified assignment table to support group-splitting is shown in Figure 5·15. There are three new fields in this table: “Global Id”, “Number of splits” and “Split Id”.

“Global Id” is used to indicate where in the global memory should the accumulated energy for this group (or sub-group) be stored. For the atoms which have not been split, the “Global Id” field takes the same values as “Atom 1” field in the original table. When a particular first atom is split into multiple sub-groups, then for the first sub-group, “Global Id” = “Atom 1” and for the subsequent sub-groups, “Global Id” = next available unassigned atom id (which is equal to the total number of atoms in the system (n_{atoms}) for the first time, n_{atoms}+1, n_{atoms}+2 and so on as more sub-groups are formed). For all the sub-groups, though, the value in the “Atom 1” column remains the same as before splitting, since that is the actual atom id of the first atom whose energy is being computed.

The remaining two new columns are used by the master thread in the second level accumulation kernel. These are used for gathering the partially accumulated energies of the different sub-groups from the global memory array and storing their sum back in the global memory at the appropriate first atom id. The thread responsible for the accumulation of the partial energies in the first sub-group is also responsible for this second level accumulation. “Number of splits” indicates how many sub-groups has the current first atom group been split into. This depends on the size of its neighbor list and the number of threads per block. This is how many entries of the global memory array that a second-level master thread must add together. For the groups that have not been split, this field takes a value of -1. The “Split Id” field stores the “Global Id” for the second sub-group for the current group. This is used to determine where in the global memory array should the second-level accumulation start from. A second-level master thread accesses “Num Splits”-1 partial energies from the global memory array, starting from the index equal to the “Split Id”, adds them to its own accumulated energy and stores the result back in the global memory at an index equal to the id of its “Atom 1” (or the “Global Id” since for the second-level masters, they will always be the same). The “Split Id” field is used only by the second-level master threads. For all the other rows, this field takes a value of -1.

In addition to the new fields in the assignment table, the columns previously present also take slightly different values. The “Size” column now indicates the number of second atoms in the split atom, not the total. Total is not stored anywhere and is not needed. In the new assignment table, the “Master” field can take one of 3 values: -1 (not a master) 0 (master) and 1 (second-level master). If a thread is a master (or a second-level master), it is responsible for accumulating the partial energies from the shared memory and storing in the global memory at an index equal to the “Global Id”. The threads which are marked as second-level masters additionally perform the second-level accumulation, as described above. This second level accumulation is done in a separate kernel that is called after the main energy computation kernel.

With the modified assignment table, we can support energy evaluations in double precision floating point arithmetic. Due to the flexibility of the scheme, there is no limit on the size of the neighbor-lists and no requirement on the minimum number of threads per block.

Another benefit of this scheme is the better utilization of the available multiprocessors. In the original assignment table, since the groups are required to be mapped on the multiprocessors in their entirety, each thread-block is required to have a large number of threads but only a few thread blocks are used. As a result, some of the multiprocessors remain idle. Splitting the groups enables the distribution of work across multiple multiprocessors. With this, different combinations of thread-block and grid sizes can be applied, allowing for improved processor utilizations and hence improved performance. Even though the double precision capabilities of the current generation GPUs are a fraction of that for the single precision, the better utilization helps offset most of it. As a result, the speedup achieved on GPU-accelerated energy minimization with double precision is comparable to that with single precision.

5.4 Summary

Binding site mapping is an effective technique for drug design and discovery. Its computational complexity, however, makes it prohibitively slow on a serial processor. The most computationally demanding step of mapping is the minimization of the CHARMM potential. In addition to mapping, CHARMM potential minimization is applied to other molecular modeling techniques such as flexible docking. The acceleration of energy minimization computations thus benefits various programs in the realms of both docking as well as mapping. The iterative nature of the computation, with little computation per iteration, as well as the high precision requirements, however, makes the acceleration of energy minimization a challenging task.

As shown in this chapter, we map the evaluation of the electrostatic energy between the particle-pairs on the FPGA. On the FPGAs, performing computations using floating point arithmetic results in high resource requirements and thus little parallelism. Nevertheless, by having custom-designed, deeply pipelined processing core, we can process the particle-pairs in a streaming fashion, effectively performing multiple computations in parallel. Compared to the original single-core implementation, our design on the FPGA achieves a speedup of 42× on the core computations. Note that this speedup is obtained with only a single FPGA pipeline. With the availability of larger FPGAs, more pipelines can be instantiated, leading to further improvements in performance.

Our GPU accelerated energy minimization routines accelerate both the electrostatic as well as the van der Waals energy evaluations. As discussed, efficient mapping of these computations on the GPU, so as to minimize data transfer overheads and improve processor utilizations, requires modifying the original neighbor-lists data structures. Our proposed assignment-table data structure performs static mapping of the particle-pairs on the GPU threads, enabling uniform distribution and improved parallelism. The result is a 14.5× performance improvement using single precision arithmetic and 11× using double precision.

Chapter 6

System Design and Integration

6.1 Overview

The primary goal of this research is the acceleration of production molecular docking and mapping systems to enable faster and more cost-effective molecular modeling. Accelerator design involves two main components: design and implementation of the coprocessor that accelerates the core computations and the integration of the coprocessor into the rest of the system. In previous chapters, we discussed the FPGA and GPU algorithms for the acceleration of various computationally intensive steps of docking and binding site mapping. In this chapter, we present the integration of these coprocessors into production software codes.

The current study involves the use of both FPGAs and GPUs for the acceleration of two different molecular modeling tasks. Since the system integration issues encountered in FPGA based designs are quite different from those in GPU based designs, we discuss these in two separate sections. In each section, we present the overall system architecture of the accelerator and discuss the issues specific to each of the molecular modeling task.

6.2 Methods and Tools

Creating a hardware-accelerated system involves many steps such as the identification of the computationally intensive tasks, initial design of the accelerated algorithms, verification of these designs for functional correctness, implementation of the design, simulation (or emulation) and verification, synthesis and place and route, timing analysis, and finally the system integration followed by the verification of the integrated system. Each of these steps requires the use of one or more CAD and software development tools. In this section, we present a brief discussion of the methods and tools used in this study.

Creating an accelerated system is a cumbersome task, with multiple iterations of design and implementation phases and tedious debugging. Moreover, improper initial analysis of the original software code can sometimes result in poor overall performance or incorrect results. In order to limit this iterative process and ensure functional correctness of the integrated system, we follow a systematic approach towards the design of the accelerated systems; this can be listed as the following steps:

- **Profiling of the serial code:** This step is used to identify the computationally intensive parts of the code. These parts become the candidates for acceleration.
- **Analysis of the computations:** The computationally intensive sections of the code are analyzed to evaluate the amenability to efficient hardware implementation and assess the potential for parallelization. This step also takes into account the Amdahl's law and determines if the overall system would benefit from hardware acceleration.
- **Code partitioning:** An important step in the design of the accelerated systems is to clearly identify the software/hardware partitions. In hardware-accelerated systems, explicit data transfer is required between the software and the hardware modules and often the data formats used in these modules are different, requiring data-type conversions. Due to this, having "fuzzy" bound-

aries with interleaving hardware and software modules leads to poor overall performance.

- **Design space exploration:** Most often, there are more than one possible ways to design the accelerated modules, with different solutions presenting different trade-offs. Initial design space exploration to identify the various ways of porting the software algorithm to the hardware structures and selecting the most suitable one reduces the wasted efforts for re-designing.

- **Design on-paper:** This step involves creating an initial hardware design on-paper to check the viability of the hardware implementation of the algorithm. This includes analyzing the resource requirements of the design (e.g. the FPGA block RAMs and multipliers, GPU shared memory etc.), bandwidth calculations and estimating data transfer latencies.

- **Design emulation:** VHDL/Verilog coding is tedious and time consuming and its simulation is a slow process. In order to quickly verify the functional correctness of the hardware design, we write a serial 'C' emulation for the designed hardware. This serves the purpose of verifying the design before the actual hardware description is written and also allows to perform precision analysis and compare the results for acceptable errors. A separate emulation code is required only for the FPGA design; on the GPU, emulation can be done simply by using the GPU emulation mode.

- **Hardware implementation:** Once the design has been emulated and verified, the next step is the actual hardware implementation. For the FPGA, this involves writing the hardware description in VHDL/Verilog. For the GPU, it involves writing multi-threaded, C-like code with explicit work distribution and data communication.

- **Behavioral simulation:** This step involves simulating the hardware design using a behavioral simulator. It is required only for FPGA-based designs and serves to ensure that the hardware implementation is behaviorally correct. The results generated by the simulation are compared against those generated by the emulation code.

- **Synthesis, place-and-route, and timing:** Again, this step is required only for FPGA-based designs and involves converting the hardware description into the physical gates mapping on the FPGA fabric. Often, behaviorally correct designs generate incorrect results after placement and routing due to incorrect timings. This step, thus, often requires performing timing-closures, involving multiple iterations of hardware debugging, and code rewriting. The GPU analogue of this step is compilation in the device mode and verifying on the real graphics board.

- **System integration:** This step involves addressing the system-level issues to integrate the hardware co-processor into the serial code. It is often one of the most time consuming step of the entire process as it requires the use of vendor-specific APIs for board-initialization, host-board communication and control.

- **Verification and validation:** The final step in the accelerator design is the verification and validation of integrated system against the original serial code.

Each of the above steps requires the use of some CAD tools and software development suites. The tools used in the current study are as follows:

- **Software language and compiler:** For the implementation of the serial profiling and emulation code, we use the ANSI-C language. The program is compiled using Microsoft Visual Studio IDE. For the development of the GPU code, we use the NVIDIA CUDA architecture. The GPU code is compiled using the nvcc compiler [NVI08c] from NVIDIA, integrated with the Microsoft Visual Studio C compiler.

- **Hardware description language:** For implementing the hardware designs on the FPGA, we use VHSIC Hardware Description Language (VHDL). For GPU implementation, we use the NVIDIA CUDA architecture.
- **CAD Tools:** VHDL code is written in either the Xilinx ISE [Xil10] or the Altera Quartus [Alt10] tool suite. The design is synthesized using the Synopsys Synplify Pro [Syn10] synthesis tool. Place-and-route is performed using the Xilinx or Altera PAR tools. For the simulation of the hardware design, we use the ModelSim [Men10a] functional simulator from Mentor Graphics. For the integration of the FPGA design into the accelerator board, Gidel's ProcWizard graphical user interface is used to set up the on-board memories, as well as the control and communication constructs.
- **Workstation:** The software compilers and IDEs and the hardware synthesis and PAR tools run on a Dell Workstation with quad-core Intel Xeon Harpertown X5472 processor @ 3GHz and 8GB RAM, running the Microsoft Windows XP operating system.
- **Hardware platforms:** The current work utilizes two different FPGA boards: the socket based XtremeData XD1000 system [XDI07] with an Altera Stratix II EP2S180 FPGA and the PCIe based Gidel PROCe III board containing an Altera Stratix III EP3SE260 FPGA. The GPU design is run on a PCIe based NVIDIA TESLA C1060 board with the TESLA T10 GPU and 4GB on-board memory.

6.3 System Integration of the FPGA-based Accelerators

In this section, we describe the system architecture and the integration for the FPGA based accelerators for molecular docking and energy minimization. The overall system architecture and the control flow is similar for both the applications, except for the specific control and communication structures, which are discussed separately.

6.3.1 System Architecture

The system-level view of the FPGA based accelerator is shown in Figure 6·1. It consists of two sides: the host side and the accelerator side. The host side consists of the CPU running the operating system, the software modules, and the interface APIs for communication between the board and the host. The accelerator side contains the PCIe board with the FPGA, the on-board DDR memories, and the PCI express interface.

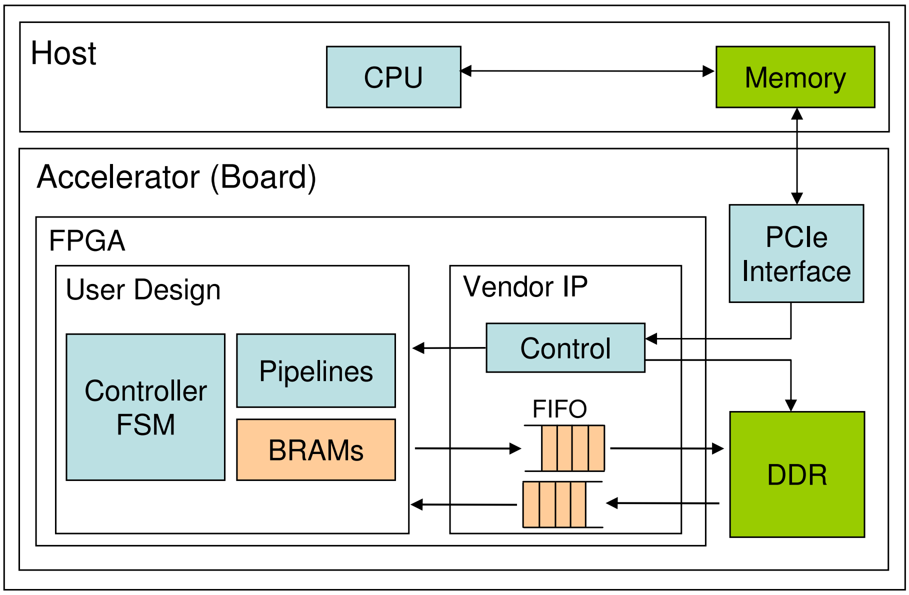


Figure 6·1: System architecture for FPGA-accelerated docking and mapping.

The FPGA contains two modules: the user design and the vendor specific IP cores. The user design implements the pipelines to accelerate the computations; for molecular docking, this includes the systolic arrays for multiple correlations; for energy minimization, it includes the pipelines for the electrostatics energy computations. In addition to the computation engines, the user design also includes block-RAMs for storing the input/output data as well as some parameters. A controller FSM is used to control the overall flow through the pipelines including the pipeline initialization and loading and the flow of the input and output data between the block RAMs and the pipelines.

The vendor IP cores serve the purpose of enabling the flow of data between the host and the on-board memory and the on-board memory and the FPGA.

6.3.2 Control Flow

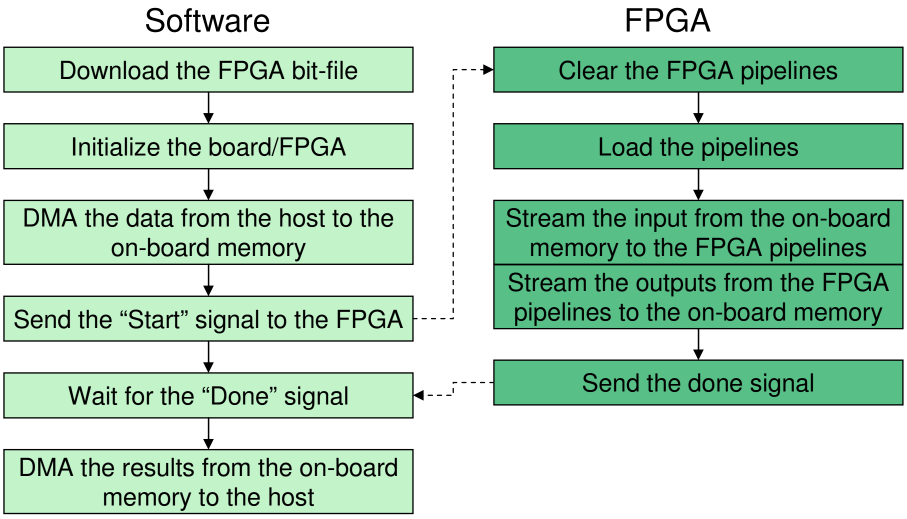

Software
Download the FPGA bit-file
Initialize the board/FPGA
DMA the data from the host to the on-board memory
Send the “Start” signal to the FPGA
Wait for the “Done” signal
DMA the results from the on-board memory to the host
FPGA
Clear the FPGA pipelines
Load the pipelines
Stream the input from the on-board memory to the FPGA pipelines
Stream the outputs from the FPGA pipelines to the on-board memory
Send the done signal

Figure 6·2: Flow of control between the software and the FPGA modules.

Figure 6·2 shows the overall sequence of steps for performing the computations on the FPGA-accelerated system. The left half shows the steps performed in software and the right half shows the steps after the control is transferred to the FPGA. The transfer of control is shown with the dashed lines.

The program starts with downloading the configuration bit file into the FPGA, followed with the board and FPGA initialization. The required data for the FPGA is prepared on the host and is DMAed to the on-board memory via the PCI express interface. A "start" signal is then sent to the FPGA and the control is transferred to the hardware module. Upon receiving the start signal, a controller FSM on the FPGA follows the steps of initializing and loading the pipelines, streaming the data through the pipelines and storing the results in the on-board memory. The controller FSMs for docking and energy minimization are different and are discussed later.

Once the FPGA engine finishes processing the inputs, it generates a "done" signal, triggering a notification to the host. The host then DMAs the results back from the accelerator on-board memory.

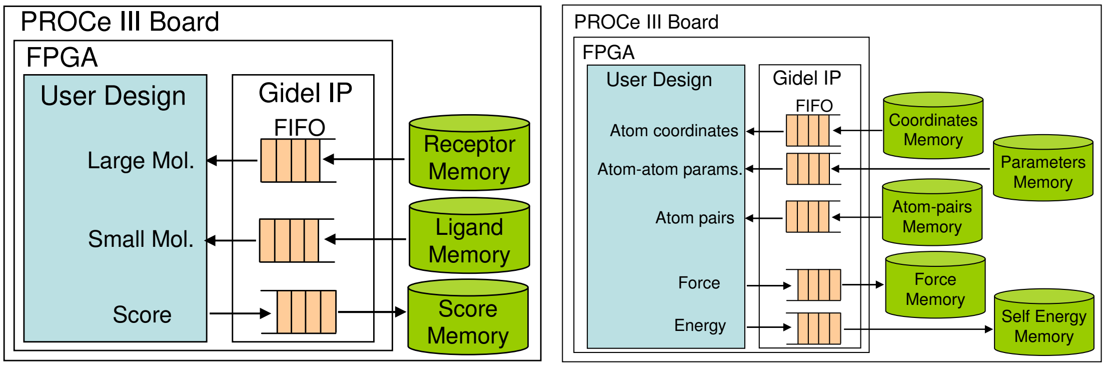
6.3.3 Host-board Data Transfer

The transfer of data from the host memory to the on-board DDR and vice-versa is done through the PCI express. This is dong using the DMA APIs provided by the vendor. In case the data format used by the FPGA is different from that on the host (as is the case for docking), the data type conversion must be performed. This can be done either on the host or on the FPGA. We perform this conversion on the host as a pre-processing step. This has the advantage that the converted reduced-precision fixed point data can be packed together for efficient transfer, thus saving on the DMA time. The packed data is then unpacked on the FPGA side before being fed into the FPGA pipelines.

For the molecular docking application, the scoring coefficients and the voxels of the large molecule are DMAed once and stored in the on-board memory. For each rotation, the voxels of the rotated small molecule are DMAed to the on-board memory and filtered scores are DMAed back to the host.

For the energy minimization, the atom-atom parameters are DMAed only once and stored in the on-board memory. However, the atom coordinates are DMAed from the host to the board and the computed forces and energies are DMAed back to the host in each iteration.

6.3.4 Board-FPGA Data Transfer



(a) (b)

Figure 6·3: FPGA–on-board memory interface for (a) molecular docking and (b) energy minimization

The use of the on-board memory is essential if the working-set size is larger than the amount of block RAMs available on the FPGA. Moreover, certain vendor APIs allow communication between the host and the FPGA only via the on-board memory. On the Gidel FPGA boards, transfer of data between the on-board memory and the FPGA modules is facilitated by Gidel's FIFO IP cores. The FIFO interface is shown in Figure 6·1. The vendor FIFO controller IP is responsible for the movement of data between the FIFO and the DDR. The user design simply interfaces to the FIFOs. The user design is responsible for generating the control signals for these FIFOs. In our design, the user design memory read / write signals and the control signals generated by the FSM are used to generate the appropriate FIFO control signals.

The FIFO interfaces used for the molecular docking and energy minimization FPGA designs are shown in Figure 6·3(a) and (b) respectively.

As shown in Figure 6·3(a), three different on-board memories are used for molecular docking: the receptor memory, the ligand memory and the score memory. Gidel PROCe III board contains three memory banks namely bank A, bank B and bank C. The receptor memory is instantiated in bank B and the ligand and the score memories are located in bank A. Receptor voxels are downloaded from the host memory once and stored in the receptor memory. For each rotation, the voxels of the rotated ligand are downloaded and stored in the ligand memory. These are then loaded in the FPGA processing units via the FIFO IPs. During the pipeline operation, the receptor voxels are streamed from the receptor memory to the FPGA and the generated scores are stored in the score memory.

For the energy minimization, five different memories are used (Figure 6·3(b)). The parameter memory is loaded with the atom-atom parameters from the host. This is done only once since the parameters do not change. There are 255 different atom types, leading to a 255 × 255 atom-atom parameter matrix. We notice, however, that this matrix is very sparse, with close to 99% of the values being zero or not used. We, thus, condense this matrix before transferring to the board and storing in the parameters memory.

Like the atom-atom parameters, the list of atom pairs is also downloaded only once, unless the neighbor lists on the software side are updated; this happens only once in a few hundred iterations.

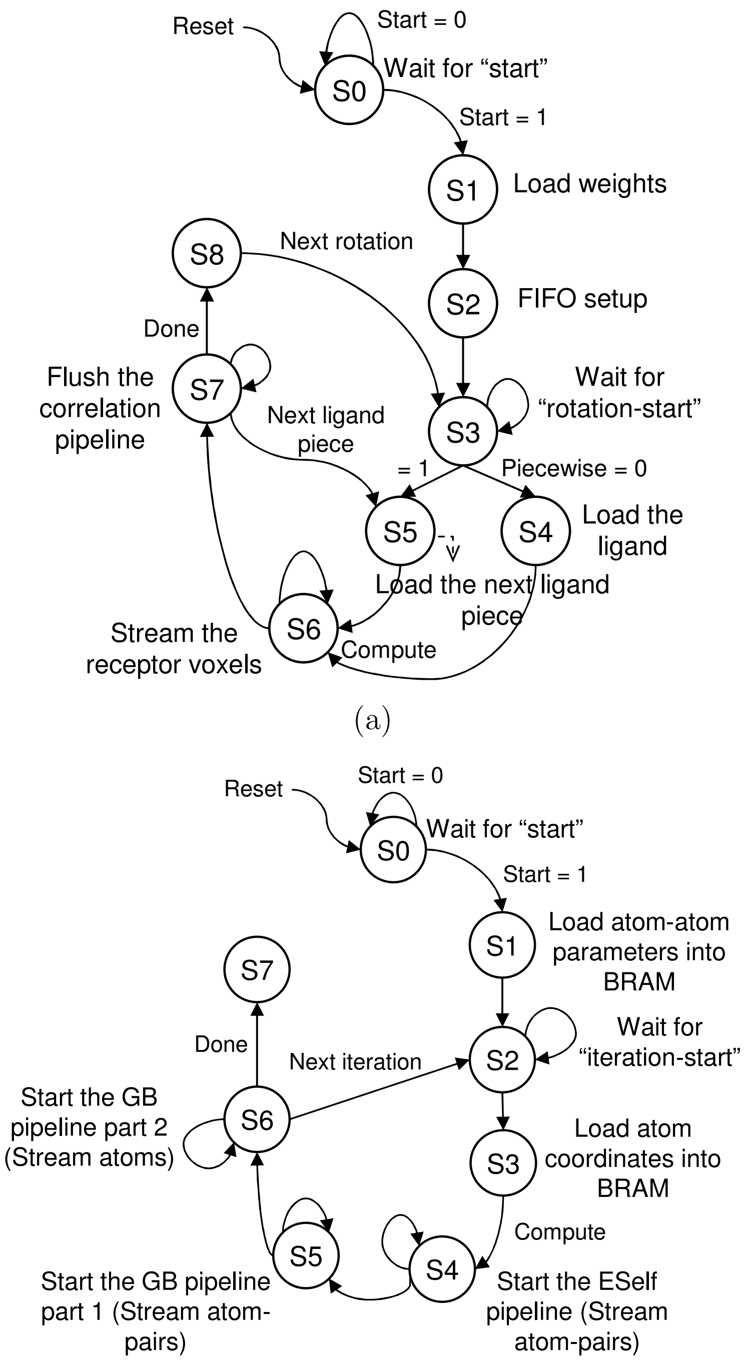
Atom coordinates are downloaded from the host in each iteration and stored in the coordinate memory. These are then transferred to the FPGA block RAMs. The electrostatics pipelines on the FPGA access the atom coordinates from the block

RAMs. The force and energy values generated by the pipelines are stored in the respective on-board memories before being DMAed back to the host.

6.3.5 FPGA Controller FSM

Once the required data is transferred from the host memory to the on-board DDR memory, the control is transferred to the FPGA by setting a "start" signal into a hardware register. This register is set-up using the Gidel's ProcWizard graphical user interface and its value is set by the software. Once the FPGA receives a high start signal, the controller FSM on the user design starts the sequence of loading the FPGA processing elements and streaming the inputs. The controller FSMs for molecular docking and energy minimization applications are shown in Figure 6·4(a) and (b) respectively.

The controller FSM for molecular docking waits in state $S0$ for the start signal to arrive. Upon receiving the start signal, it moves to state $S1$ and loads the scoring coefficients into the weighted scorers. This is followed by the FIFO setup state wherein the delays through the FIFOs are adjusted as per the sizes of the molecules. In state $S3$, the controller waits for the software to finish downloading the ligand grid for the current rotation to the FPGA and send the "rotation-start" signal. Depending upon whether piecewise correlation is performed, the controller moves to either the $S4$ or the $S5$ state, where the ligand voxels are read from the on-board memory and rippled through the compute cells, with the last voxel being streamed-in first. In the case of piecewise correlation, voxels of a part of the ligand are loaded. Once all the compute cells are loaded with the ligand voxels, the streaming of the receptor voxels starts (in state $S6$) and the correlation scores are generated, one per cycle. Once the entire receptor is streamed, the pipeline is flushed by streaming-in zeros in state $S7$. Upon completion of the flushing phase, the system either returns to state $S5$ and


Reset
Start = 0
Wait for "start"
S0
Start = 1
S1
Load weights
S8
Next rotation
S2
FIFO setup
Done
Flush the correlation pipeline
S7
Next ligand piece
Wait for "rotation-start"
S3
= 1
Piecewise = 0
S5
S4
Load the ligand
Load the next ligand piece
S6
Stream the receptor voxels
Compute
Reset
Start = 0
Wait for "start"
S0
Start = 1
S1
Load atom-atom parameters into BRAM
S7
Wait for "iteration-start"
Done
Next iteration
S2
Start the GB pipeline part 2 (Stream atoms)
S6
S3
Load atom coordinates into BRAM
Compute
S5
S4
Start the GB pipeline part 1 (Stream atom-pairs)
Start the ESelf pipeline (Stream atom-pairs)

(a)

(b)

Figure 6·4: Controller FSM on the FPGA (a) for molecular docking and (b) for energy minimization loads the next ligand piece (for piecewise correlation) or to the $S3$ state and waits for the "rotation-start" signal to arrive before loading the next rotated ligand into the compute cells.

The controller FSM for energy minimization is similar to that of molecular docking. After the start signal arrives, the atom-atom parameters are loaded into the FPGA block RAMs. The system then moves to state $S2$ wherein it waits for the host to finish downloading the updated atom coordinates and send the "iteration-start" signal. Once that signal arrives, the atom coordinates are loaded into the FPGA block RAMs. Electrostatics energy computation starts in state $S4$ wherein one atom-pair is streamed into the self-energy pipeline per cycle. Once all the atom pairs are processed, the second streaming pass starts wherein the first part of the GB pipeline processes one atom pair per cycle (state $S5$). In state $S6$, individual atoms are streamed through the second half of the GB pipeline and the atom forces are computed. After the forces of all the atoms have been computed, the system returns to state $S2$ and waits for the "iteration-start" signal before starting the next minimization iteration.

6.3.6 Integration into the Production Code

Integrating the FPGA accelerated routines into production software often requires elaborate changes and additions to the original software code. Changes to the original code include separating the hardware and software components into clear partitions and replacing the software function calls with calls to the hardware routines. Moreover, integration also requires adding new code for setting up the hardware modules, preparation of the data for the hardware routines and transfer of data to and from the accelerator board.

The amount and extent of modification required is dependent on the original code and the amount of restructuring and data-format manipulations performed for the hardware implementation. Here we discuss the changes made to the production molecular docking code PIPER and the production mapping code FTMap to integrate the FPGA accelerated subroutines.

Integration of both the accelerated docking and mapping requires extensive modification of the original code, though for different reasons. In the case of molecular docking, the change of the computation algorithm and the use of a different data format leads to extensive modifications. Recall that the serial PIPER code performs multiple FFT/IFFT pairs for generating the correlation sums of different energy functions. In the hardware, however, multiple correlation scores are generated in parallel using a combined systolic pipeline. Individual calls to the FFT/IFFT functions, thus, cannot simply be replaced with calls to the hardware routine. Instead, multiple software correlation calls need to be replaced with a single hardware call and the outer level control loop of the software program needs to be modified. Moreover, software docking employs single precision floating point arithmetic whereas hardware correlation is performed using fixed point arithmetic. This requires additional code to translate the input data to the format used by the hardware and the result to a format used in software.

Integrating the FPGA-accelerated energy minimization into the FTMap program does not require any floating to fixed point data format conversion since both the software and the hardware operate on floating point data. The data structures used by the hardware, however, are very different from those used in software. This requires addition of some pre-processing steps that converts the original neighbor lists data into a pairs-list. Moreover, additional pre-processing steps are required to compute the initial atom energies before the atoms are transferred to the FPGA. This is an $O(N)$ computation and is performed on the host in order to save on hardware resources.

Another important change to the energy minimization code is the merging of the neighbor lists. Software minimization code maintains two sets of neighbor lists, one called the nb-list and the other called the 1-4 list. These are processed by the software by separate functions. We noticed, however, that most of the computation performed by these functions are identical, with a very small divergent loop. For efficient computation on the FPGA, we merge these lists and process them in a single hardware pipeline, with a divergent path in a part of the pipeline. The appropriate computation path is selected using a multiplexor which is controlled using a bit encoded in the atom-pairs.

In addition to the above changes, both the PIPER and the FTMap code have been modified to include the following:

- **Initialization of the FPGA board:** This includes adding vendor specific APIs to set-up the board with the required memories, download the configuration bit-file, reset the system and initialize the co-processor with the required parameters.
- **Transfer of data:** Vendor APIs are added to DMA the data from the host memory to the on-board accelerator memory. Depending on the size of the data, one or more DMA calls are required.
- **Listening to the coprocessor:** The software needs to “listen” to the hardware coprocessor to determine when the coprocessor is done performing the required computations. This can be set-up either using an interrupt or by polling on a register whose value is set by the hardware.

Both the original and the integrated software are single threaded and run on a single processor core, with the accelerated routine running on a single FPGA. The design can, however, be easily parallelized on multiple FPGAs to utilize the coarse-level parallelism available in both the rigid docking and the energy minimization computations.

6.4 System Integration of the GPU-based Accelerators

The integration of the GPU-accelerated docking and mapping routines is relatively straightforward compared to their FPGA counterparts. This is mainly due to the more software-like memory model and the memory allocation system provided by the NVIDIA CUDA architecture. The main task in the integration of the GPU accelerated routines is the preparation of the appropriate data structures, allocation of the device memory and explicit transfer of data to and from the accelerator board. The first of these is done using simple C programming while the latter two are performed using the NVIDIA CUDA API functions.

6.4.1 System Architecture:

The system architecture of the NVIDIA GPU based docking and mapping accelerator is shown in Figure 6·5. The accelerator contains the graphics processing unit and the on-board memory called the device memory, along with the PCIe interface.

The host-board communication is performed via the device memory. Data is transferred from the host memory to the device memory via the PCIe interface, using the CUDA data transfer API. Once in the device memory, data can be loaded into the processors' explicitly-managed cache called the shared memory. The processors can also access the device memory directly, though it has higher latency.

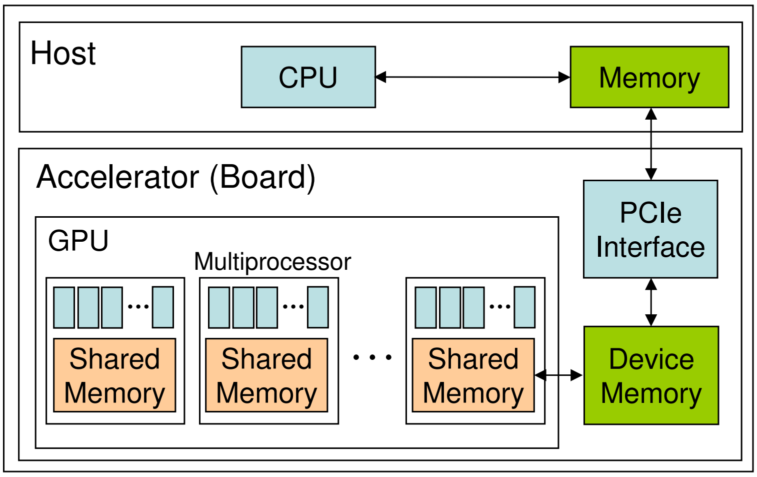


Figure 6·5: System architecture for GPU-accelerated docking and mapping.

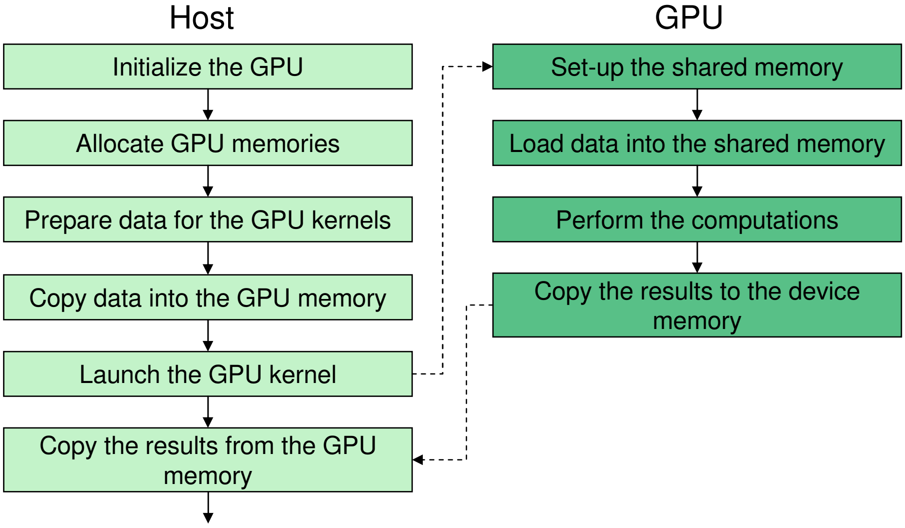

Host
GPU
Initialize the GPU
Allocate GPU memories
Prepare data for the GPU kernels
Copy data into the GPU memory
Launch the GPU kernel
Copy the results from the GPU memory
Set-up the shared memory
Load data into the shared memory
Perform the computations
Copy the results to the device memory

Figure 6·6: Flow of control between the host and the GPU.

6.4.2 Control Flow:

The sequence of steps required to perform the computations on a GPU-based system are shown in Figure 6·6. The first step includes recognizing the different GPU boards in the system and selecting and initializing the appropriate board. This is followed by the allocation of the required memories on the GPU. The organization of the data on the host memory is often different from what is most efficient on the GPU. This requires preparing the data arrays on the host before transferring them to the device. In our GPU accelerated docking and mapping codes, all the 2D and 3D arrays used by the software are flattened into 1D arrays before transferring to the GPU.

Once the required memories have been set up and loaded, the GPU kernel can be launched, in much the same way as a software function is called. One difference is that the GPU kernel must be "configured" with the appropriate number of threads and thread-blocks to be used.

On the GPU kernel, the first task is to allocate any shared memory used by the kernel and load them with the data from the device memory. CUDA does not provide any API for this transfer and it must be explicitly done using the GPU threads. The GPU kernel can then perform the intended computations and store the results in either the shared memory or directly into the device memory. If the results are stored in the shared memory, they must be explicitly copied into the device memory since the shared memories are not accessible from the host across the PCIe bus. Once the control is transferred back to the host, it can copy the results from the GPU device memory and continue with the rest of the program flow.

Note that if the GPU kernels are launched repeatedly, the transfer to the device memory need not be repeated unless the data has changed. This is because the GPU device memory is persistent across kernel calls. Shared memory, however, is not persistent and copying into the shared memory must be repeated in each kernel launch.

For the molecular docking, the receptor voxels are transferred once to the device memory and stored. For each rotation, the rotated ligand is copied into the device memory. The GPU kernel then copies it into the shared memory and performs the computation.

In the case of energy minimization, the assignment tables are transferred to the device memory only once, unless the neighbor-lists are updated. In each iteration, the atom coordinates are copied to the device memory and the energy computation kernels are launched. The computed energy and force values are copied back to the host in each iteration.

6.4.3 Integration into the Production Code

Integrating the GPU-accelerated routines into the production software code requires code changes similar to those for the integration of the FPGA routines. This include adding code for the GPU initialization, data preparation, memory allocation, data transfers and swapping the software function calls with the GPU kernel launches.

Integrating the GPU-accelerated docking into the PIPER code requires two extra steps compared to the FPGA-accelerated system. Recall that the docking solution on the GPU utilizes both direct correlation as well as FFT-based correlation. Depending on the size of the molecules, one is preferred over the other. A check on the molecule size is thus added into the integrated PIPER code and the appropriate docking routine is called based on the size. Moreover, for certain small molecule sizes, direct correlation based docking on the GPU evaluates multiple rotations together within the same kernel. Incorporating this requires additional change in the control flow of the program.

For the integration of the GPU-accelerated energy minimization into the FTMap program, the main step is the generation of the assignment tables from the neighbor lists. This preprocessing step is added to the FTMap program and is performed every time the neighbor lists are updated. Similar to the FPGA-system, the nb-list and the 1-4 lists are combined and are processed together in a single pass. Multiple calls to the energy-evaluation functions are thus replaced with a single call to the GPU kernel.

Like the FPGA-accelerated system, the GPU accelerated system also runs on a single core and utilizes a single GPU board. Parallelizing this to utilize multiple GPUs requires converting the code into a multithreaded version, with each process utilizing one graphics engine. This is relatively straightforward and would enable coarse-level parallelism. The application can scale to many GPUs since both PIPER and FTMap perform hundreds of independent rounds of docking / minimization.

6.5 Summary

In order for the accelerated subsystem to benefit the user community, its integration into the production software code is essential. The task of system integration, however, is not trivial and calls for careful consideration of a number of factors. If attention is not paid, the performance improvements obtained from the accelerated tasks can easily be lost in the integration step. In particular, the most important factors that must be addressed include the host-accelerator data transfer overhead and the modifications required in the original code for the software to interface with the hardware. Elaborate changes in the original code can sometimes be needed, including pre- and post-processing of the data. The choice of the appropriate software code as well as the right hardware platform, thus, play an important role in obtaining good overall performance.

Chapter 7

Results

7.1 Overview

In this chapter, we present our results from the FPGA and GPU acceleration of the molecular docking program PIPER and the binding site mapping program FTMap. The results are presented in terms of the performance improvements obtained on these programs and the precision errors resulting from the integrated systems. Furthermore, we discuss the significance of our work in terms of the power savings compared to performing the computations on a conventional processor and compare the costs of the accelerator-based solutions and their serial counterparts.

We also present our results relating to the floating point utilizations obtained for these applications on the two platforms, along with that on a multicore processor. This result is of interest since the graphics processors boast of very high raw floating point capabilities and it is interesting to observe as to how much of it is attainable on production applications. It also throws some light on the applicability of the FPGAs for the acceleration of floating point intensive applications.

We present the results for docking and mapping in separate sections. For both the applications, the FPGA designs were implemented on the Gidel PROCe III board with an Altera Stratix III EP3SE260 FPGA and 4.5 GB on-board memory. The board is housed in a PCIe x4 slot in a Dell Precision PWS670 workstation with a dual-core Intel Xeon processor @ 2.8 GHz and 2 GB RAM. The workstation is running 64-bit Microsoft Windows XP Professional SP3 operating system. The host code for the FPGA accelerated design runs on one of the cores of this dual-core processor.

The GPU-accelerated PIPER and FTMap codes run on an NVIDIA TESLA C1060 GPU card, containing the TESLA T10 GPU and 4 GB on-board memory. The GPU board is plugged into a PCIe x16 slot inside a Dell Precision T7400 workstation. The host processor is a quad-core Intel Xeon Harpertown X5472 processor @ 3GHz and 8GB RAM, running 64-bit Microsoft Windows XP Professional SP2 operating system. The host code for the GPU-accelerated systems uses a single core of the quad-core processor.

For performance comparisons, the serial reference code for both PIPER and FTMap is run on the machine that houses the GPU. The reference code is the original production version and is single threaded, unless specified otherwise, utilizing one of the cores of the quad-core Xeon Harpertown processor. Multicore versions of these codes are not available. The results on the multicore version, wherever reported, were obtained from our implementations.

7.2 Accelerated Molecular Docking

7.2.1 Performance Improvements

We present the performance improvements obtained on the correlation task that is being accelerated and the overall performance improvements on the PIPER program. The performance improvements on three different architectures are compared: FPGA, GPU and a quad-core CPU (using all 4 cores). The performance improvements are shown for different molecule sizes. On the FPGA, the correlation task is always performed as direct correlation whereas for the multicore and the GPU implementations, the graphs show the method that performs best (direct or FFT correlation) for the given molecule size.

Performance of the Correlation Task

Figure 7·1 shows the performance of the correlation task for the PIPER energy functions with four pairwise potential terms, thus 8 correlations together. Four pairwise terms are used due to two reasons: first, PIPER most commonly utilizes only 4 pairwise potential terms to represent the desolvation energy. Moreover, on our FPGA systolic correlation pipeline, we can support 4 pairwise terms without any swapping.

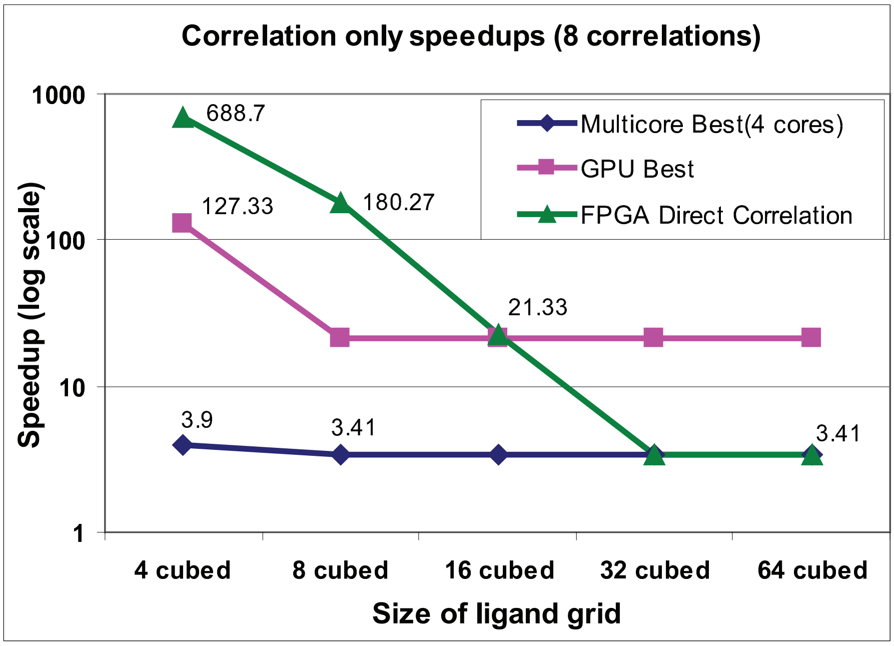


Figure 7·1: Speedups on different architectures for the correlation-only task. Speedups shown are for the PIPER energy functions with 4 pairwise potential terms (8 correlations). The reference baseline is the best correlation method on a single core.

The FPGA pipeline runs at a frequency of 105 MHz and can support ligands sizes up to 7^3. For larger ligands, multiple passes through the pipeline are required.

For the ligand size of 4^3, direct correlation performs better than the FFT correlation for the GPU, the multicore as well as the single core. The leftmost data points therefore use that method for all the technologies, including the single core reference. For larger ligand sizes, FFT correlation outperforms direct correlation.

As shown in the figure, for small ligand sizes, both FPGA and GPU achieve two orders of magnitude speedup on the multiple correlations required for the PIPER energy functions, with the FPGA performing superior to the GPU. For ligand sizes above 8^3, the GPU version consistently achieves a speedup of around 21x. The FPGA direct correlation continues to outperform the GPU and the multicore. The crossover point for the FPGA direct correlation is 16^3 with respect to the FFT correlation on the GPU and about 30^3 with respect to the FFT correlation on a four core processor. For ligand sizes larger than 32^3, direct correlation on the FPGA results in a negative speedup. This is mainly due to the inability to fit the ligand on the chip in its entirety. This requires swapping the ligand pieces in and out of the FPGA pipelines and multiple passes through the large receptor grid.

An interesting observation here is that the crossover between the FPGA and the GPU accelerated docking happens almost directly on the small/large molecule boundary. For small molecule docking, FPGA achieves multi-100 fold speed-up for the correlations, which accounts for 94.5% of the computation. FPGA docking, however, is clearly not suitable for large molecule docking where the size of the ligand is much larger. For large molecule docking, the GPU version achieves good performance consistently.

Speedups on Different Computations

Table 7.1 shows the speedups for various per-rotation computations in rigid docking, with 8 correlations per rotation. The speedups shown are for the GPU accelerated docking. On the FPGA, these steps are performed as part of the correlation pipeline and hence their individual performance improvements cannot be measured.

Table 7.1: Speedups from GPU on various per-rotation computations in rigid docking.

Task	Runtime (msec)		Speedup
	CPU	GPU	
Rotation and charge assignment	80	80	1x
Correlations	3600	160 – 13.5	22.5x – 267x
Accumulation of desolvation terms	180	1	180x
Scoring and filtering	200	30	6.7x
Total per rotation	4060	270 – 125	15x – 32x

Rotation and grid assignment are left on the host and thus have a speedup of 1. The speedup on the correlation task varies from 22x to 267x, depending on the size of the ligand and hence the chosen correlation method. As discussed earlier, GPU resources are underutilized during scoring and filtering, leading to the modest speedup. On the other hand, the computations in the accumulation of the desolvation term are relatively straightforward and afford very high speedup due to the inherent parallelism. Based on the size of the ligand, the overall speedup per rotation varies from 15x to 32x.

Performance of the Accelerated PIPER Program

Figure 7·2 shows the overall performance of the accelerated PIPER program on the three architectures. The reference baseline is the original serial PIPER code using FFT for computing the correlations. The graph is shown for 10,000 rotations, each with 18 pairwise potential terms, which is the maximum and the default in PIPER. The total number of correlations per rotation is thus 22.

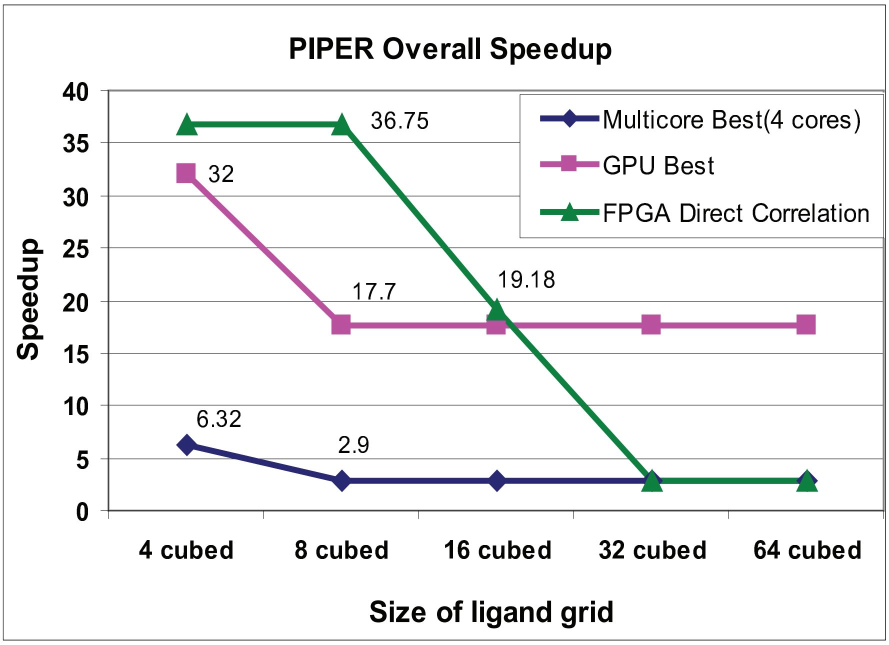


Figure 7·2: End-to-end speedup of the PIPER program with 18 pair-wise potential terms (22 correlations). The reference baseline is the original PIPER code using FFT correlations.

As can be seen, the speed-ups on the entire application are far more modest than on the correlation-only task. This is due to two factors. One is simply the Amdahl's law. The second is the large number of correlations required of each rotation. On the FPGA, this is performed using multiple passes. This affects the performance significantly. For small ligand grids, the limiting factor for the FPGA implementation is the host overhead (about 200 ms per rotation) which dominates the execution time for ligand sizes $\leq 8^3$.

In terms of the absolute runtime, the reference serial code on a single core requires 27.8 hours for the completion of 10,000 rotations. On the FPGA, this reduces to 45 minutes for ligands up to 8^3 and to 1.5 hours for a ligand of size 16^3. GPU accelerated PIPER achieves performance similar to the FPGA for small ligands (4^3). For sizes above 4^3, the runtime on the GPU is consistently around 1.7 hours.

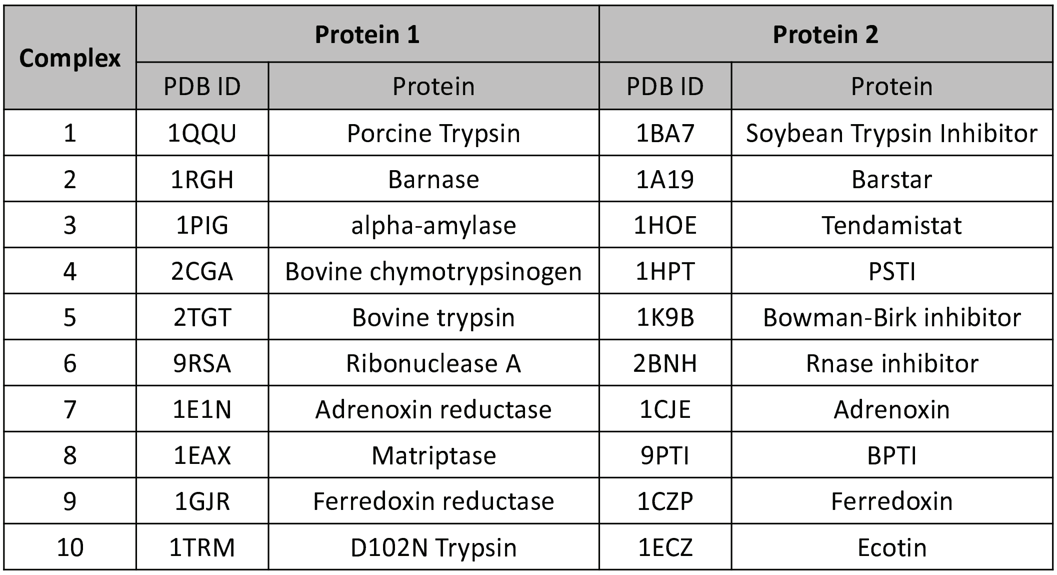
Another interesting observation here is that the crossover point of the FPGA with respect to the GPU is slightly higher for the integrated system. This is because the GPU performs filtering as a separate step, whereas on the FPGA it is pipelined with the correlation and so its latency is hidden. This increases the crossover point slightly.

Overall, the FPGA remains superior to the GPU for low precision correlations, and has better performance for ligand sizes less than about 16^3. The performance difference for the correlation alone (for these small ligands) is substantial; when the rest of the computation is included, however, the difference in speed-up is more modest. For all ligands larger than 16^3, the GPU version continues to give excellent performance, while the FPGA stops being cost-effective at around 25^3.

From a user's perspective, the GPU-accelerated version is clearly more cost-effective—with respect to an unaccelerated workstation—for both protein-protein and small molecule docking. If the user is interested only in small molecule docking, then the FPGA-accelerated version could be preferred. The performance difference, however, might not be great enough to justify the higher cost and the reduced flexibility of that technology. If, however, some of the overhead can be reduced, then the FPGA may find a clearer niche. Moreover, an efficient 3D FFT for FPGAs would shrink the performance gap for large molecule pairs.

7.2.2 Errors Due to Reduced Precision

The aim of molecular docking is to find the relative pose that yields the best interaction between the two molecules. The goodness of fit is measured in terms of the pose score. In order for the accelerated docking to be of use, it is important that the scores generated are close to the scores generated by the unaccelerated version. Since the ultimate aim is to find the best pose and not the pose score itself, the algorithm is not very sensitive to the variations in the individual score value. Both the FPGA and GPU accelerated docking algorithms, however, generate scores that are very close to those generated by the original software version.



Complex	Protein 1		Protein 2	
	PDB ID	Protein	PDB ID	Protein
1	1QQU	Porcine Trypsin	1BA7	Soybean Trypsin Inhibitor
2	1RGH	Barnase	1A19	Barstar
3	1PIG	alpha-amylase	1HOE	Tendamistat
4	2CGA	Bovine chymotrypsinogen	1HPT	PSTI
5	2TGT	Bovine trypsin	1K9B	Bowman-Birk inhibitor
6	9RSA	Ribonuclease A	2BNH	Rnase inhibitor
7	1E1N	Adrenoxin reductase	1CJE	Adrenoxin
8	1EAX	Matriptase	9PTI	BPTI
9	1GJR	Ferredoxin reductase	1CZP	Ferredoxin
10	1TRM	D102N Trypsin	1ECZ	Ecotin

Figure 7·3: 10 different protein-pairs from the Docking benchmark used for the evaluating the accelerated-PIPER code.

As discussed in section 3.4, protein-pairs from the Docking benchmark [MWP^{+}05] are often used to evaluate the effectiveness of protein docking programs. To measure the accuracy of our accelerated-docking program, thus, we docked 10 receptor-ligand pairs from this benchmark. The list of the protein-pairs used is shown in Figure 7·3. The docking phase for each pair was run for 5000 rotations and the best score from each rotation was compared against those generated by the unaccelerated software. Figure 7·4 shows the distribution of results in different error ranges. On average, 98.6% of the total results have errors less than 0.01% of the original score. At most one or two out of the 5000 scores have difference larger than 5% of the original score, with none having error larger than 10%. As per our understanding, this error is well within the acceptable range.

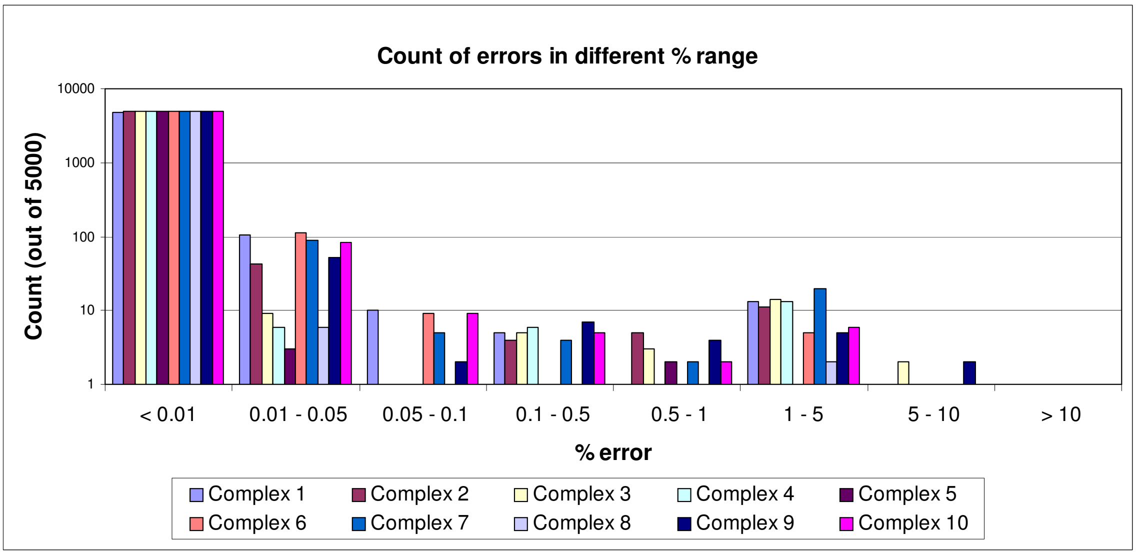


Figure 7·4: Distribution of errors in the results generated by the accelerated docking. Results shown are for 10 different receptor-ligand pairs from the docking benchmark; each was run for 5000 rotations.

7.3 Accelerated Binding Site Mapping

Binding site mapping consists of two phases: rigid docking of small molecule probes and energy minimization of the protein-probe complexes. In the accelerated FTMap, rigid docking is performed using the accelerated PIPER program, with performance speedups as reported in the previous section. Since FTMap operates on small molecule probes, the speedups in the docking phase are in the high end of the range of speedups presented earlier. For a probe size of 4^3, we obtain an overall speedup of 36x on the FPGA accelerated rigid docking and 32x on the GPU version, compared to the production, single core implementation of PIPER. Comparing against our FFT based multicore implementation of PIPER, running on a quad-core processor, the speed-up reduces to around 11x. On multicore, as in the case of the GPU and the FPGA, we observed that for small ligand sizes, direct correlation is faster than FFT. Comparing against our implementation of the direct correlation based PIPER on multicore, the speedup for the accelerated PIPER further reduces to 6x.

7.3.1 Performance Improvements of the Energy Minimization Task

Here we present the results from the acceleration of the energy minimization phase of the FTMap program. On the FPGA, we accelerate only the electrostatic energy evaluation whereas the GPU version accelerates both the electrostatics and the van der Waals computations. These results are thus presented separately.

Performance Improvement on GPUs

Table 7.2 shows the speedups achieved on various energy and force computations mapped onto the GPU kernels. The runtimes presented are for a single energy minimization iteration, which involves performing around 10,000 atom-atom computations for each of the energy terms. Force update kernel updates forces for the 2200 atoms in the complex. The GPU runtimes include the overhead for memory allocation, data transfers and kernel launch. The serial reference runs on a single core of a quad-code processor.

Table 7.2: Speedups for different energy evaluation and force update steps of energy minimization.

Computation	Runtime (msec)		Speedup
	CPU	GPU	
Self energies	6.15	0.23	26.7x
Pairwise interactions	2.75	0.19	17x
van der Waals	0.5		
Force updates	0.95	0.14	6.7x

As shown in the table, the computations of the pairwise interactions and the van der Waals energy have been combined into a single GPU kernel. This was done to reuse the computations common among them and to save on the kernel launch overhead. The speedup achieved on different computations are proportional to the complexity of the computation and hence the amount of work performed by each thread. For simpler computation such as the force updates, the communication overhead lowers the performance benefits. Overall, the GPU-accelerated energy minimization achieves a per-iteration speedup of around 14.5x.

We also measured the overall energy minimization times, including the computations that are performed on the host and all the overheads, for various different protein-probe complexes. The average time for minimizing 2000 conformations of a complex on the original single-core FTMap program is around 400 minutes. On our GPU accelerated version, the energy minimization time reduces to 32 minutes, representing an overall speedup of 12.5x for the energy minimization phase.

The speedups reported above are for a single precision implementation on the GPU. The original serial implementation, however, utilizes double precision floating point arithmetic. As discussed in previous chapters, in order to achieve results identical to the unaccelerated version, we implemented a double precision version of the GPU-accelerated energy minimization. Although the double precision performance of the current generation NVIDIA GPUs is $1/10^{th}$ of that for the single precision, our implementation achieves a performance similar to that of the single precision version. This is due to the improved distribution of work, as discussed in section 5.3. Using double precision arithmetic, we obtain an overall per-iteration speedup of around 11x.

Performance Improvement on FPGAs

The performance improvement obtained from an FPGA implementation is directly related to the number of pipeline replications possible on a given FPGA chip. The speedups obtained from the FPGA implementation of the self energy and the pairwise interaction computations are shown in Table 7.3. The speedups shown are post place-and-route estimates based on two pipelines each for the self energy and the pairwise interaction computations. The pipelines implement a single precision floating point datapath and operate at a frequency of 185 MHz. As shown, the FPGA pipelines obtain a potential speedup of two orders of magnitude for both the computations.

Table 7.3: FPGA speedups for self-energy and pairwise interaction computations. The runtimes shown are with 2 instances of computation pipelines, running at 185 MHz.

Computation	Runtime (msec)		Speedup
	CPU	FPGA	
Self energies	6.15	0.027	225x
Pairwise interactions and force updates	2.75	0.027	100x

The runtimes shown in the table do not take into account the host-to-board data transfer time as well as the FPGA logic required to control the pipelines and communicate with the host. Upon integrating the pipelines with the control logic and the vendor specific IP cores for communication with the host, we can only fit one set of pipelines on the Altera Stratix III FPGA. Moreover, the integrated design synthesizes at 125 MHz. The overall computation-only speedup with the integrated design thus reduces to 42x. Accounting for the host-board data transfer time and other overheads, the overall end-to-end speedup reduces to 27x. Note that this speedup is with only one set of pipelines on the FPGA. The computation is highly scalable to multiple pipelines on an FPGA. On newer generation FPGAs, we can potentially instantiate at-least 2 to 3 sets of computation pipelines. This would increase the overall speedup to around 60x compared to a single core implementation.

7.3.2 Errors Due to Reduced Precision

Obtaining higher performance from the FPGA and GPU accelerators warrants the use of reduced precision. Precision manipulations in the accelerators, however, often result in a difference in the final outcome compared to the original unaccelerated code. To ensure functional correctness, it is necessary to verify that this difference is within the acceptable tolerance. As stated earlier, FTMap uses double precision floating point arithmetic for energy minimization. The use of double precision floating point arithmetic is not viable on the current generation FPGAs. Our FPGA accelerated energy minimization pipelines use single precision arithmetic. GPUs, however, offer far higher double precision floating point capabilities, although it is still a fraction of those for the single precision. On the GPU, we accelerate the energy minimization computations using both the single precision and the double precision floating point arithmetic.

The use of single precision arithmetic for energy minimization has two effects. First, it results in some precision errors in the energy and force values computed per iteration. Secondly, since energy minimization is an iterative process, with the force values from the current iteration being used to generate the atom coordinates for the next, this error gets compounded as the computation progresses. Since the termination criteria for this iterative process is the convergence of the energy value within a certain threshold and since the threshold value is very small, the convergence in the single precision code happens much earlier than that in the double precision version, potentially converging at a different point. We discuss these differences below.

Convergence

Our experiments indicate that the protein-probe complexes that require around 700 iterations for convergence in double precision converge in only about 60-70 iterations when run using single precision. The point of convergence is sometimes close to the final convergence point, though in other cases, the convergence happens at a very different point. This early convergence clearly happens due to the lack of the high precision required to differentiate the energy of one iteration from the next within the threshold. In order to ensure that the convergence happens at the right place, we force the minimization to continue even after the energy values converge.

Error on the Energy and Force Values

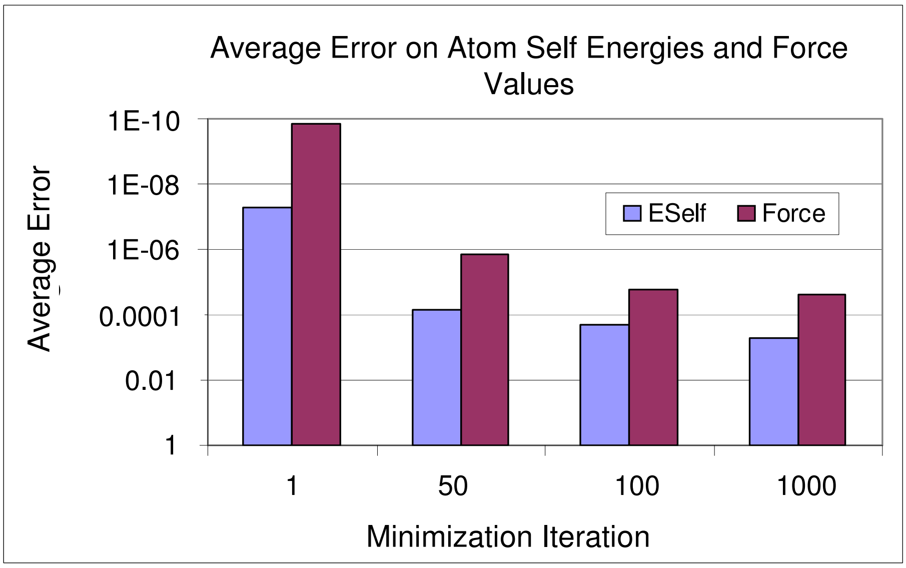


Figure 7·5: Average error in the energy and force values after different number of minimization iterations.

Figure 7·5 shows the average error on the atom self energy and force values after different number of minimization iterations. The errors shown are the differences between the the values computed by the accelerated energy minimization with single precision and those by the double precision serial code. For the first minimization iteration, the average error on the energy and force is around 5×10^{-8} and 1×10^{-10} respectively. As the computation progresses, the error gets compounded, with the error on the final values after few hundred iterations being around 2.5×10^{-5}.

Error on the Final Atom Coordinates

Though the errors on the final energy and force values are not significant, a more important measure of the accuracy is the final atom-coordinates after the minimization has converged. The goal of the FTMap program is to obtain the probe orientation in the protein that results in the least energy conformation. The final orientation is represented by the coordinates of the atoms after the minimization.

In our accelerated energy minimization routines, the final atom coordinate values have errors in the range of 1.24×10^{-3} Å, with a few atoms having errors as large as 5×10^{-1} Å. In FTMap, the atom coordinates are represented with a resolution of $1/1000^{th}$ of an Angstrom. This is the same format as required for the atom coordinates in the Protein Data Bank (pdb) file format [RCS10]. Moreover, FTMap uses multiple probes to determine the binding pocket. The binding pocket is determined by the region where most of the probes cluster together. The absolute position of each probe is, thus, not of utmost importance. An error in the range of 10^{-3} Å is thus within the acceptable range.

Errors with Double Precision

As discussed above, the use of single precision arithmetic results in some errors in the final atom coordinates and the energy and force values. Even though in our opinion these errors are within the acceptable range, to obtain the results exactly identical to the original software FTMap code, we implemented the energy minimization on the GPU using double precision arithmetic. Due to the non-associativity of the floating point operations as well as the different implementation of the floating point units, the use of similar precision can still result in some difference in the generated values. Compared against the serial version, the GPU-accelerated double precision energy minimization returns energy and force values that are identical up to 8 places of decimal. Moreover, the final atom coordinates have errors less that 10^{-5} Å. Since the pdb format represents the atom coordinates with a resolution of 10^{-3} Å, this error is not visible in the final probe conformation.

7.4 Scaling to Multiple FPGAs and GPUs

The performance results presented above for both molecular docking and energy minimization are using a single FPGA or GPU. It is worth mentioning, however, that both these applications afford high coarse-level parallelism and thus the proposed accelerators can be easily scaled to multiple FPGAs and GPUs, both on a single accelerator board as well as across multiple boards housed in different nodes. For the molecular docking, different accelerators can evaluate different rotations in parallel, independent of each other. Similarly, the accelerators for the energy minimization can operate on different protein-probe complexes. The production PIPER and FTMap codes currently run on large multi-node clusters and are thus MPI enabled. FPGA and GPU accelerated nodes can further improve the performance by providing fine-level parallelism at each node. Though we do not have the performance numbers for the multi-accelerator docking or mapping servers, we realize the potential performance improvements from the integration of the accelerated node into the production server and it is a part of the future work.

7.5 Performance with respect to Cost and Power

Here we present the performance comparison of the FPGA and GPU accelerated algorithms with the original software version in terms of the power consumption and the cost of the hardware. Even though the cost and power calculations are approximate, they provide a good indication of the relative benefits of the three architectures with respect to these parameters.

Power Consumption

In addition to the improved performance with respect to the computation times, the FPGA and GPU accelerated algorithms presented in this work provide benefits in terms of the reduced power consumption. As a result, the energy cost for performing these computations is much smaller compared to the original software version.

Table 7.4: Comparison of the total power consumption for docking and energy-minimizing 16 probes using the FTMap program.

System	Power rating	Computation time	Power dissipation (watt-hrs.) Host	Accelerator	Total	Savings
CPU	120/4 = 30	106 hrs.	3180	–	3180	–
FPGA	30	4 hrs.	120	120	240	92.5%
GPU	180	8.5 hrs.	255	1530	1785	44%

FPGA chips have very small power consumption compared to the current-generation microprocessors. A high-end FPGA chip typically dissipates 20 to 30 watts of power, much lower than that of a quad-core Intel Xeon Harpertown processor, which dissipates 120 watts. Even though the current high-end GPUs typically dissipate more power than a microprocessor, around 180 watts, they also deliver very high floating point capabilities. Due to this, both FPGAs and GPUs have much smaller power-per-flop requirements compared to a microprocessor.

Table 7.4 compares the power consumption for the FTMap program running on the FPGA, GPU and the single core of a quad-core Intel Xeon processor. The total power dissipation is presented as a product of the power rating and the time needed to perform the computation. For the accelerators, the total power includes the power consumed by the CPU for running the host program. Note that the power rating of the CPU is shown as $1/4^{th}$ of the actual power rating of the quad-core processor. This is done for a fair comparison since the serial code is running on only one of the 4 cores.

As shown in the table, the FPGA-accelerated FTMap dissipates less than 8% of the total power dissipated by the unaccelerated version, resulting in significant savings. Even though the power savings from the GPU-accelerated version are not as impressive as that from the FPGA version, it is still significant.

Cost of Computation

We now present a rough estimate on the cost of performing computations using the three architectures – FPGAs, GPUs and multi-core. The comparisons are presented in terms of the cost-per-hour on these platforms as well as the peak FLOPs-per-dollar. The cost-per-hour is calculated assuming two different lifetimes of the systems: two years and five years. These are shown Table 7.5. The cost for power is calculated using the lifetime and the power rating of the system and assuming a power cost of 16 cents per kilowatt-hour [U.S10].

As shown in the table, in terms of the absolute cost of computation per hour, GPU performs better than the CPU whereas the FPGA has a higher cost. Accounting for the improved computational performance, the effective cost, assuming a performance improvement of 10× from the accelerators, is much smaller for both the FPGA and the GPU. The effective computation cost on the GPU and FPGA based systems is respectively $1/15^{th}$ and $1/5^{th}$ the cost on a CPU.

Table 7.5: Cost of computation on different hardware platforms. The effective costs on the FPGA and GPU based systems are calculated assuming a 10× performance improvement compared to the CPU based system.

System	Hardware cost	Lifetime	Power cost	Total cost	Cost/hr. (cents)	Effective cost/hr.	FLOPs/$ (PFLOPS)
CPU	$2600	2 years	$336	$2936	16.7	16.7	2.1
FPGA	$7000		$84	$7084	40	4	0.65
GPU	$1500		$504	$2004	11.5	1.1	29.5
CPU	$2600	5 years	$841	$3441	7.9	7.9	4.4
FPGA	$7000		$210	$7210	16.5	1.6	1.6
GPU	$1500		$1261	$2761	6.3	0.6	53.4

In terms of the peak FLOPs per dollar, the GPU delivers about 12× −14× higher performance than a quad-core CPU, whereas the FPGA has 3× higher cost per FLOP compared to the same CPU. The main reason for the high cost on the FPGA based system is the high cost of the FPGA chip, which in-turn is the consequence of the low-volume production. Another important point here is that the FLOPs-per-dollar has been calculated based on the peak and not the sustained performance. As will be discussed in section 7.6, achieving utilizations close to the peak value is not trivial; FPGAs perform better in this respect compared to both the GPUs and the multicore, thus resulting in comparatively higher effective delivered FLOPs-per-dollar compared to the multicore.

To compare the relative costs of computation on different architectures for a production application, we plot the cost of a single docking run using the PIPER program (Figure 7·6).

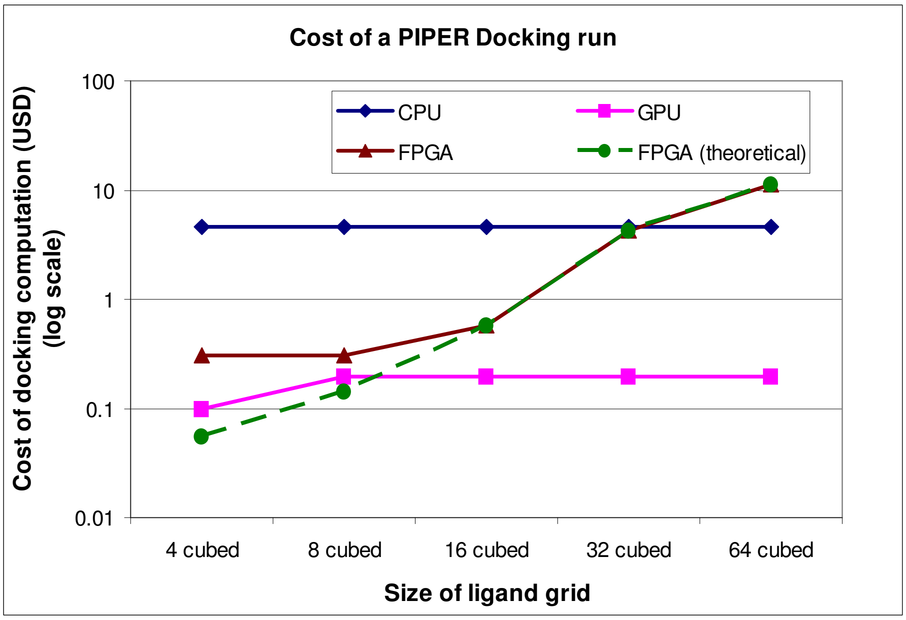


Figure 7·6: Cost of docking using the PIPER program on different architectures.

The computation cost is computed as a product of the runtime for 10,000 rotations and the cost of computation per hour on the corresponding architecture (6^{th} column in Table 7.5). Overall, GPU-based docking is the most cost effective solution, across all ligand sizes. The FPGA solution is more cost-effective than the serial version for ligand sizes up to 32^3. Even though the speedup for smaller ligand sizes is higher on the FPGA than on the GPU, the need to perform certain computations on the host increases the total runtime, and hence the computation cost. The line labeled "FPGA (theoretical)" plots the cost on the FPGA if the remaining computations are also moved to the FPGA. In that case, the FPGA has lower cost than the GPU for sizes up to 8^3.

Development Cost and Portability

Finally, we present a brief discussion on the development costs for the different platforms and the efforts needed to port these solutions to future-generation devices. The development cost can be measured in terms of the number of designer/programmer hours required to develop the solution. Another factor to account for is the amount of reuse of the developed algorithms, since the development cost gets amortized over the number of users.

Due to the mature tools and interfaces and the availability of efficient high-level languages, the development costs is the least for the CPU-based solutions. Developing a multi-core implementation, however, has some added costs associated with it, especially for applications that do not afford high parallelism. For example, even though we did not implement a multi-core version of the energy minimization computations, we anticipate that the efforts required for the same would be very high.

The C-like programming model for the GPUs and the standardized IO interface enable efficient development with relatively lower efforts. Obtaining efficient designs on the FPGAs, however, requires the use of low-level hardware description languages which are more cumbersome than high-level languages. This, combined with the lack of a standardized IO interface, leads to higher development costs for the FPGA-based solutions. With respect to the docking and mapping solutions presented here, even though the development cost is much higher than that on a single-core CPU, the FPGA and GPU accelerated designs are general and can be applied to a variety of docking programs. This amortizes the cost, since it benefits the large community of users of the docking and mapping programs.

The rapid change in the technology requires that the accelerated designs be ported to newer-generation devices every few years. Porting the accelerated routines to future devices involves changes at two levels: scaling the accelerator core to benefit from the increased resources available on the new chip and changing the control and interface code to enable the integration into the new board.

For the FPGA based designs, scaling the design to larger chips mainly involves replicating the pipelines to utilize the available resources. For both the docking and energy minimization routines, the coarse-level parallelism allows multiple pipelines to operate independently, thus enabling easy scaling.

The board-level interface for the FPGA design remains mostly unchanged, unless the design is moved to a board from a different vendor. Our current designs run on the PCIe board from Gidel. The newer generation boards from Gidel use the same interface, thus requiring no change in the interface code. Due to the non-standardized interface across the accelerator boards from different vendors, moving the FPGA design to a board from a different vendor, however, would require significant change in the interface logic, both in the FPGA code as well as the host software.

In the case of our GPU-accelerated routines, the block-independent distribution of the computations allows our designs to easily scale to larger chips; more processors available on newer chips can be utilized simply by changing the kernel configuration, i.e. the number of threads and thread-blocks with which the kernel is launched. Moreover, the standardized CUDA interface from NVIDIA remains unchanged across different boards. Thus, porting our current designs to the newer-generation FPGA and GPU systems seems relatively straightforward, at-least for the near-future.

7.6 Floating Point Utilizations

In this section, we present our results relating to the floating point utilizations achieved from the different architectures on production docking and mapping computations. The computations involved in both the applications addressed in the current study are high on floating point requirements. In terms of the architectures addressed in this study, the graphics processors clearly possess very high raw floating point capabilities, with the NVIDIA TESLA GPU providing a peak single precision performance of 936 GFLOPs. Multicore processors also contain multiple specialized, vector floating point units. The Intel 4 core i7 processor @ 3 GHz provides a peak performance of 96 GFLOPs. Even though FPGAs have long been considered as computation engines for integer operations, modern FPGAs provide floating point capabilities comparable to and sometimes higher than high end multicore processors. Assuming a multiplier-to-adder ratio same as that required for performing the FFT computations, the Altera Stratix III SE340 FPGA can deliver a peak performance of 72 GFLOPs.

Table 7.6: Floating point utilizations achieved on different architectures for the computations in production molecular docking and mapping applications.

Architecture	Computation	Single precision performance		Utilization
		Peak	Achieved	
Stratix III SE260 FPGA	Docking	72 GFLOPs	–	–
	Energy minimization	72 GFLOPs	50 GFLOPs	70%
TESLA C1060 GPU	Docking (128^3 FFT)	936 GFLOPs	38 GFLOPs	4%
	Energy minimization	936 GFLOPs	6 GFLOPs	0.6%
4-core Intel i7 @ 3 GHz	Docking (128^3 FFT)	96 GFLOPs	8 GFLOPs	8%
	Energy minimization	96 GFLOPs	–	–

For real life applications, however, obtaining utilizations close to the peak is often challenging. Table 7.6 shows the utilizations obtained on different architectures for the different computations performed in the current study.

As can be seen, even though the GPUs offer high peak performance, the utilization on a production application is rather small. This is mainly due to the comparatively rigid architecture, with fixed communication and synchronization patterns. FPGA, on the other hand, achieves very high utilization due to the close to full utilization of the hardware fabric and a customized communication pattern that allows keeping the pipelines filled at all times. The utilization on a multicore processor is somewhere between that of the FPGA and the GPU, though for a highly tuned library such as the FFTW, the 8% utilization is clearly very low.

7.7 Summary

In this chapter, we discussed our results from the acceleration of the computations in molecular docking and binding site mapping. In terms of the performance improvements, our FPGA and GPU accelerated routines for different computations in docking achieve multi-hundred fold speedups, with the overall end-to-end speedup in the range of 18× to 36×. The computation-only speedup for the mapping task varies from 6× to 42× on these architectures, resulting in the overall speedup of 13× on the GPU and 30× on the FPGA.

Moreover, we provided a discussion about the development cost and the portability of the accelerated code. We also presented our results in terms of the savings with respect to the power and the cost of computation. On average, the proposed algorithms result in 44% to 92% reduction in the power consumption, while costing less than $1/10^{th}$ of the original cost.

Finally, we presented a discussion about the floating point utilization on the FPGAs, GPUs and a multi-core processor for the two production molecular modeling codes. Based on our results, of the three architectures, the FPGAs achieve the highest utilizations; the multi-core processor comes a distant second.

Chapter 8

Conclusions and Future Work

We conclude this thesis by summarizing our work on the acceleration of molecular modeling applications and listing some possible future work in this direction. We also present our reflections on performing high performance computing using accelerators. In particular, we list some of the lessons learnt related to the potential and the effective use of FPGAs and GPUs for application acceleration.

8.1 Summary

In this research, we started out with the goal of accelerating complex algorithms in the field of molecular modeling, to enable cost-effective, desktop-based solutions. In particular, we focused on two such applications, namely molecular docking and binding site mapping, and two accelerator architectures, FPGAs and GPUs. We analyzed various docking algorithms and production docking systems with respect to the core computations and their amenability to acceleration. Moreover, we developed a set of hardware-accelerated algorithms for various computations involved in these applications, achieving significant overall performance improvements. The proposed solutions provide an additional benefit in terms of their reduced power dissipation. We also demonstrated the viability of accelerating the floating-point intensive computations using reconfigurable hardware and discussed the appropriate computation model for the same.

Moreover, in the course of our research, we discovered various aspects of accelerator based design, in general, and with respect to the computations in molecular modeling algorithms, such as the need for algorithmic restructuring to enable efficient mapping and the importance of efficient utilization of the available resources. Finally, we also analyzed the effectiveness of various accelerator platforms in delivering high floating point performance on real-life applications, thus shedding some light on the relative merit of these architectures in delivering sustained throughput.

8.2 Observations and Conclusions

Some of the key lessons learnt from the acceleration of the two production codes are:

- **Algorithmic restructuring is crucial:** The computation models and the data structures that are most optimal for serial implementation are not always best suited for mapping to the accelerators. Efficient mapping to hardware requires hardware-aware restructuring of the computations and the data structures.

- **High performance comes from high utilization:** To achieve high parallelism and maximum performance, it is important to utilize all the available hardware resources. On the FPGAs, it amounts to replicating the computation units to use the entire chip area. On the GPUs, it involves efficient distribution of the work among the different multiprocessors.

- **High utilization is difficult to achieve:** While accelerating real-life applications, achieving high utilization is a non-trivial task. This is particularly true for the GPUs. Non-uniform distribution and divergent code can lead to very poor utilization of the available processors, leading to poor performance. Even though replicating the processing units on the FPGAs can result in high utilizations, other issues such as limited bandwidth and other limiting resources can prevent this. This points us back to the first point – algorithmic restructuring. To enable uniform distribution and use the available resources efficiently, so as to improve the utilizations and hence the overall performance, significant restructuring of the original computations is required.

- **Streaming is good for FPGAs:** Streaming computation model is very efficient for FPGA-based designs. Designing deep pipelines and performing the computations in a streaming fashion results in high utilizations and high effective parallelism.
- **Use of the memory hierarchy:** Efficient use of the memory-hierarchy on the GPUs is very important in achieving good performance. Performing computations from the global memory is very inefficient; shared memory must be used wherever possible.
- **GPU programming is (not) trivial:** Our experience shows that getting an application working on the GPUs is rather easy. Achieving good performance, however, is not so much.

To conclude, we present the following remarks about the acceleration of molecular modeling applications and the use of FPGAs and GPUs for high performance computing:

- **Molecular docking computations are good candidates for acceleration:** Based on the current research, we conclude that molecular docking algorithms, in the domains of both rigid and flexible docking, have huge computational needs and can benefit from acceleration. Their acceleration, however, is a non-trivial task. Though these algorithms exhibit immense coarse-level parallelism, extracting fine-grained parallelism is challenging. Moreover, even though the flexible docking algorithms tend to be more computationally demanding, their small computation core, combined with the iterative nature, makes them hard to accelerate. In general, achieving overall high performance for docking applications requires moving all the computations to the accelerator side, something that is often hindered by the limited accelerator resources.

- **FPGAs demonstrate high potential for the acceleration of molecular modeling:** We conclude that the FPGAs provide a viable and effective solution for the acceleration of complex molecular modeling applications, including those involving large floating point computations. Moreover, FPGAs currently have a lot of slack – in terms of the frequency, the power budget and the technology. For example, current generation FPGAs typically operate at a frequency of 200 MHz and do not contain any dedicated floating point units. Advancements in these areas would further improve the potential of the FPGAs. In our opinion, the biggest challenge facing this technology is the lack of appropriate tools and programming models and the large overhead associated with system integration. Currently, obtaining an efficient design requires the use of low-level hardware description languages such as VHDL and Verilog. Moreover, system integration requires the use of vendor-specific APIs. Due to these aspects, developing FPGA-based solutions requires very large efforts and development time. Though there exist a variety of high-level *C-to-gates* languages, we believe that most result in inefficient hardware and thus poor performance. There is, hence, a pressing need for efficient, high-level development tools and a standardized host-accelerator interface. Until such a tool is developed, the use of low-level HDLs is probably the right choice.

- **GPUs provide a cost-effective HPC solution:** GPUs possess very high floating point capabilities and are suitable for accelerating a wide variety of applications, including those in molecular modeling. The main advantage of GPUs, apart from their huge computational capabilities, is the availability of standard, high-level programming models and the support for system integration. This has made the GPUs receive widespread attention in the last few years. Though the speedups for reduced-precision computations are not as impressive as those on the FPGAs, the low cost, combined with the relative ease of development, makes the GPU-based solutions an attractive HPC alternative.

8.3 Future Directions

Here we list some of the future directions in this research. We have categorized them as the possible improvements to the current work as well as its extension towards the acceleration of other molecular modeling tasks. We also list some of the other domains which can potentially benefit from the accelerated routines proposed in this work.

8.3.1 Improvements to the Current Work

The current work can be improved in the following ways:

- **Optimizations to the hardware code:** In this work, we did not put any emphasis on hand-tuning our FPGA designs; for example, no manual place-and-route and reduction of the fan-out was done to improve the operating frequency. In particular, the pipeline for the docking computations runs at about 100 MHz, which is slow for the FPGA generation it is running on. This low frequency is mainly due to the global broadcast of the receptor voxels, leading to a long critical path. We believe that a frequency of around 200 MHz is easily achievable by inserting repeaters (latches) to shorten the critical path. This would further improve the performance by a factor of two.

- **Moving more computations to the accelerator side:** Due to the limited hardware resources, only the electrostatics computation of the energy minimization is currently done on the FPGA. van der Waals and bonded computations, as well as the optimization move are still performed on the host. Similarly, on the GPU, bonded computations and the optimization move is performed on the host. This requires downloading the updated atom coordinates from the host to the accelerator in each cycle. This transfer time is a significant portion of the total time. Moving these computations to the accelerator would, thus, improve the overall performance significantly.

- **Moving to newer generation devices:** As discussed earlier, our current FPGA designs have been implemented on two generations old chip. The limitations of these chips prevent us from performing all the computations on the FPGA and from having multiple computation pipelines. Moving to the larger, newer generation chips would result in improved performances.

  For the GPUs, even though our design utilizes the newest generation TESLA chip. The TESLA architecture poses certain memory and other limitations that affect the performance. These issues have supposedly been addressed in the next-generation FERMI architecture. It would be interesting to see the effects of these improvements on the overall system performance.

- **Scaling to multi-accelerator systems:** Our FPGA and GPU designs currently run on a single node, achieving fine-grained parallelism. Both docking and mapping applications, however, provide immense coarse-level parallelism. Scaling our designs to multi-accelerator systems is thus an obvious and trivial extension.

8.3.2 Extension to Other Molecular Modeling Applications

Some of the possible extensions of the current work and the future directions in the acceleration of other molecular modeling applications include:

- **Application to other algorithms:** The first possible extension is to apply the hardware-accelerated routines to other docking and mapping algorithms. The techniques developed are general enough to benefit a variety of other docking algorithms such as flexible docking, fragment-based docking and general energy-minimization.

- **Acceleration of other types of docking:** Molecular docking is a vast field, spanning a large number of optimizations and search algorithms; many of these can potentially benefit from FPGA and GPU based acceleration. In particular, spherical-polar coordinates based exhaustive search and geometric-hashing based docking seem interesting.

- **Acceleration of other computations:** To improve the discrimination sensitivities, docking systems apply various refinement algorithms on the results generated from the docking steps. An example is the RMSD based clustering and ranking. Though these steps are not as computationally intensive as docking itself, their acceleration can potentially improve the overall performance and can be looked-at.

8.3.3 Application to Other Domains

Although the focus of the current research has been the acceleration of molecular modeling applications, the proposed solutions are general and can be applied towards the acceleration of a variety of applications. For example, the proposed hardware structure for performing efficient 3D correlations can benefit applications in the domains of object recognition and full-field biomechanics deformation and strain-measurement. Similarly, the proposed data structures for the energy-minimization computation can be applied to other algorithms involving all-to-all computations such as distance based clustering and wireless sensor networks.

References

[AA10] AMD-ATI. ATI Stream Technology, 2010.

[Agi10] Agility. HandelC Language Reference Manual. `http://www.agilityds.com/literature/HandelC_Language_Reference_Manual.pdf`, 2010.

[Alt08] Altera. Altera Floating Point Compiler, Version 0.2, 2008.

[Alt10] Altera. `http://www.altera.com/support/software/sof-quartus.html`, 2010.

[Ann10] Annapolis. Annapolis Micro Systems. `http://www.annapmicro.com/`, 2010.

[ATK94] R. Abagyan, M. Totrov, and D. Kuznetsov. ICM - a new method for protein modeling and design: applications to docking and structure prediction from the distorted native conformation. *Journal of Computational Chemistry*, 15(5):488–506, 1994.

[BBO$^+$83] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. CHARMM: a program for macromolecular energy, minimization, and dynamics calculations. *Journal of Computational Chemistry*, 4:187–217, 1983.

[Bha07] A. V. Bhatt. Accelerating with Many-cores and Special Purpose Hardware. In *Proceedings of the IEEE Conference on Field Programmable Logic and Applications*, 2007.

[BHD$^+$02] W. Böhm, J. Hammes, B. Draper, M. Chawathe, C. Ross, R. Rinker, and W. Najjar. Mapping a single assignment programming language to reconfigurable systems. *Journal of Supercomputer*, 21(2):117–130, 2002.

[BK73] C. Bron and J. Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.

[BK03] N. Brooijmans and I. D. Kuntz. Molecular Recognition and Docking Algorithms. *Annual Review of Biophysics and Biomolecular Structure*, 32:335–373, 2003.

[BKC$^+$09] R. Brenke, D. Kozakov, G. Chuang, D. Beglov, D. Hall, M. R. Landon, C. Mattos, and S. Vajda. Fragment-based identification of druggable hot spots of proteins using Fourier domain correlation techniques. *Bioinformatics*, 25(5):621–627, 2009.

[Bol10] I. Bolsens. Programming Customized Parallel architectures in FPGA. In *Reconfigurable Architecture Workshop*, 2010.

[Bor20] M.Z. Born. *Physics*, 1:15, 1920.

[BTS10] J. M. Berg, J. L. Tymoczko, and L. Stryer. Biochemistry: Protein Structure and Function. `http://www.ncbi.nlm.nih.gov/bookshelf/br.fcgi?book=stryer&part=a280`, 2010.

[CCB$^+$95] W. D. Cornell, P. Cieplak, C. I. Bayly, I. R. Gould, K. M. Merz, D. M. Ferguson, D. C. Spellmeyer, T. Fox, J. W. Caldwell, and P. A. Kollman. A Second Generation Force Field for the Simulation of Proteins, Nucleic Acids, and Organic Molecules. *Journal of the American Chemical Society*, 117:5179–5197, 1995.

[CfB10] The University of Kansas Center for Bioinformatics. GRAMM-X Protein-Protein Docking Web Server v.1.2.0. URL: `http://vakser.bioinformatics.ku.edu/resources/gramm/grammx/`, 2010.

[CGVC04] S. R. Comeau, D. W. Gatchell, S. Vajda, and C. J. Camacho. ClusPro: an automated docking and discrimination method for the prediction of protein complexes. *Bioinformatics*, 20(1):45–50, 2004.

[CKB$^+$07] S. R. Comeau, D. Kozakov, R. Brenke, Y. Shen, D. Beglov, and S. Vajda. ClusPro: Performance in CAPRI rounds 611 and the new server. *Proteins: Structure, Function, and Bioinformatics*, 69(4):781–785, 2007.

[CMJW03] R. Chen, J. Mintseris, J. Janin, and Z. Weng. A Protein-Protein Docking Benchmark. *Proteins*, 52(1):88–91, 2003.

[Cra05] Cray. Cray XD1 Supercomputer. `www.cray.com/products/xd1`, 2005.

[Cra10a] Cray. Cray Inc. `http://www.cray.com`, 2010.

[Cra10b] Cray. Cray XR1 Reconfigurable Processing Blade. `http://www.cray.com/Assets/PDF/products/xt/CrayXR1Blade.pdf`, 2010.

[Cra10c] Cray. Cray XT5$_h$ Supercomputer. URL: `http://www.cray.com/Assets/PDF/products/xt/CrayXT5hBrochure.pdf`, 2010.

[CW02a] P. Chacon and W. Wriggers. Multi-Resolution Contour-Based Fitting of Macromolecular Structures. *Journal of Molecular Biology*, 317:375–384, 2002.

[CW02b] R. Chen and Z. Weng. Docking Unbound Proteins Using Shape Complementarity, Desolvation, and Electrostatics. *Proteins. Structure, Function, and Genetics*, 47:281–294, 2002.

[CW03] R. Chen and Z. Weng. A novel shape complementarity scoring function for protein-protein docking. *Proteins: Structure, Function, and Genetics*, 51:397–408, 2003.

[Dai10] H. Dai. Free Energy Minimization Accelerated with FPGAs. In *Master's Thesis, Boston University*, 2010.

[DBB03] C. Dominguez, R. Boelens, and A. M. J. J. Bonvin. HADDOCK: a protein-protein docking approach based on biochemical and/or biophysical information. *Journal of the American Chemical Society*, 125:1731–1737, 2003.

[DeH00] A. DeHon. The Density Advantage of Configurable Computing. *IEEE Computer*, 33(4):41–49, 2000.

[DMC$^+$06] A. DeHon, Y. Markovskya, E. Caspia, M. Chua, R. Huanga, S. Perissakisa, L. Pozzia, J. Yeha, and J. Wawrzynek. Stream computations organized for reconfigurable execution. *Microprocessors and Microsystems*, 30(6):334–354, 2006.

[DRC10] DRC. DRC Computer Corporation. `http://www.drccomputer.com/`, 2010.

[DSD$^+$86] R. L. DesJarlais, R. P. Sheridan, J. S. Dixon, I. D. Kuntz, and R. Venkataraghavan. Docking flexible ligands to macromolecular receptors by molecular shape. *Journal of Medicinal Chemistry*, 29(11):2149–2153, 1986.

[DSG90] A. J. Olson D. S. Goodsell. Automated docking of substrates to proteins by simulated annealing. *Proteins: Structure, Function, and Bioinformatics*, 8(3):195–202, 1990.

[Ead07] D. Eadline. The Multicore Programming Challenge. `http://www.linux-mag.com/id/4312`, 2007.

[EK97] T. J. A. Ewing and I. D. Kuntz. Critical evaluation of search algorithms for automated molecular docking and database screening. *Journal of Computational Chemistry*, 18:1175–1189, 1997.

[EMRP95] L. F. Ten Eyck, J. Mandell, V. A. Roberts, and M. E. Pique. Surveying Molecular Interactions with DOT. In *Proceedings of the ACM/IEEE Supercomputing Conference*, page 22, 1995.

[EMSK01] T. J. Ewing, S. Makino, A. G. Skillman, and I. D. Kuntz. DOCK 4.0: search strategies for automated molecular docking of flexible molecule databases. *Journal of Computer-Aided Molecular Design*, 15(5):411–428, 2001.

[Ers70] R. E. Ershov. Self-energy of a "smeared" charge. *Russian Physics Journal*, 13(6):813, 1970.

[Est60] G. Estrin. Organization of Computer Systems – The Fixed Plus Variable Structure Computer. In *Proceedings of Western Joint Computer Conference*, pages 33–40, 1960.

[EWKH94] M. B. Eisen, D. C. Wiley, M. Karplus, and R. E. Hubbard. HOOK: A program for finding novel molecular architectures that satisfy the chemical and steric requirements of a macromolecule binding site. *Proteins: Structure, Function and Genetics*, 19:199–221, 1994.

[FFT10] FFTW. Fftw. `http://www.fftw.org/`, 2010.

[FGL01] J. Frigo, M. Gokhale, and D. Lavenier. Evaluation of the streams-C C-to-FPGA compiler: an applications perspective. In *Proceedings of the International Symposium on Field Programmable Gate Arrays*, pages 134–140, 2001.

[fHPRC10] NSF Center for High-Performance Reconfigurable Computing. Novo-G reconfigurable supercomputer. `http://www.chrec.org/facilities.html`, 2010.

[Fit10] R. Fitzpatrick. URL `http://farside.ph.utexas.edu/teaching/em/lectures/node56.html`, 2010.

[FL06] H. Fouquet-Lapar. Highly scaleable RASC implementations on SGI 4700/RC100 systems. In *Proceedings of the Reconfigurable Systems Summer Institute*, 2006.

[Fou10] Alzheimer's Drug Discovery Foundation. The Drug Discovery Process. `http://www.alzdiscovery.org/index.php/alzheimers-disease/`, 2010.

[Gid09a] Gidel. PROCe III Data Book Version 1.0, 2009.

[Gid09b] Gidel. PROCWizard User's Manual, 2009.

[Gid10] Gidel. Gidel Ltd. `http://www.gidel.com/`, 2010.

[GJS97] H. Gabb, R. Jackson, and M. Sternberg. Modelling protein docking using shape complementarity, electrostatics, and biochemical information. *Journal of Molecular Biology*, 272:106–120, 1997.

[GLD+08] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High performance discrete Fourier transforms on graphics processors. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[Gro10] Gold Standard Group. OpenGL - The Industry Standard for High Performance Graphics. `http://www.opengl.org/`, 2010.

[GRT+06] M. Gokhale, C. Rickett, J. Tripp, C. Hsu, and R. Scrofano. Promises and pitfalls of reconfigurable supercomputing. In *Proceedings of the 2006 Conference on the Engineering of Reconfigurable Systems and Algorithms*, pages 11–20, 2006.

[Gu08] Y. Gu. FPGA Acceleration of Molecular Dynamics Simulations. In *PhD Dissertation, Boston University*, 2008.

[GZM07] A. Grosdidier, V. Zoete, and O. Michielin. EADock: docking of small molecules into protein active sites with a multiobjective evolutionary optimization. *Proteins*, 67(4):1010–1025, 2007.

[HGV⁺08] M. C. Herbordt, Y. Gu, T. VanCourt, J. Model, B. Sukhwani, and M. Chiu. Computing Models for FPGA-Based Accelerators. *Computing in Science and Engineering*, 10(6):35–45, 2008.

[HMWN02] I. Halperin, B. Ma, H. Wolfson, and R. Nussinov. Principles of Docking: An Overview of Search Algorithms and a Guide to Scoring Functions. *Proteins: Structure, Function, and Genetics*, 47:409–443, 2002.

[HVG⁺07] M.C. Herbordt, T. VanCourt, Y. Gu, B. Sukhwani, A. Conti, J. Model, and D. DiSabello. Achieving High Performance with FPGA-Based Computing. *IEEE Computer*, 40(3):50–57, 2007.

[II03] Wildstar II. Annapolis Micro Systems, Inc. WILDSTAR II Hardware Reference Manual, 2003.

[Imp10] Impulse Accelerated Technologies. Impulse CoDeveloper C-to-FPGA Tools. `http://www.impulseaccelerated.com/products_universal.htm`, 2010.

[JHM⁺03] J. Janin, K. Henrick, J. Moult, L. Eyck, M. Sternberg, S. Vajda, I. Vakser, and S. Wodak. CAPRI: a Critical Assessment of PRedicted Interactions. *Proteins*, 51(1):2–9, 2003.

[JWG95] G. Jones, P. Willett, and R. C. Glen. Molecular recognition of receptor sites using a genetic algorithm with a description of desolvation. *Journal of Molecular Biology*, 245(1):43–53, 1995.

[KBCV06] D. Kozakov, R. Brenke, S. R. Comeau, and S. Vajda. PIPER: An FFT-based protein docking program with pairwise potentials. *Proteins: Structure, Function, and Bioinformatics*, 65(2):392–406, 2006.

[KBO⁺82] I. Kuntz, J. Blaney, S. Oatley, R. Langridge, and T. Ferrin. A geometric approach to macromolecule-ligand interactions. *Journal of Molecular Biology*, 161:269–288, 1982.

[KDFB04] D. B. Kitchen, H. Decornez, J. R. Furr, and J. Bajorath. Docking and scoring in virtual screening for drug discovery: methods and applications. *Nature Reviews Drug Discovery*, 3:935–949, 2004.

[Khr10] Khronos Group. OpenCL - The open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`, 2010.

[KKSE^{+}92] E. Katchalski-Katzir, I. Shariv, M. Eisenstein, A. Friesem, C. Aflalo, and I. Vakser. Molecular surface recognition: Determination of geometric fit between proteins and their ligands by correlation techniques. *Proceedings of the National Academy of Sciences of the United States of America*, 89:2195–2199, 1992.

[Kor08] O. Korb. Efficient Ant Colony Optimization Algorithms for Structure- and Ligand-Based Drug Design. In *PhD Dissertation, University of Konstanz*, 2008.

[Kro03] R. T. Kroemer. Molecular modelling probes: docking and scoring. *Biochemical Society Transactions*, 31(5):980–984, 2003.

[Lab10] Los Alamos National Laboratories. RoadRunner. `http://www.lanl.gov/roadrunner/`, 2010.

[LCW03] L. Li, R. Chen, and Z. Weng. RDOCK: refinement of rigid-body protein docking predictions. *Proteins*, 53(3):693–707, 2003.

[LG09] James Larus and Dennis Gannon. Multicore Computing and Scientific Discovery. *The Fourth Paradigm*, pages 125–129, 2009.

[LHD88] S. Linnainmaa, D. Harwood, and L. S. Davis. Pose Determination of a Three-Dimensional Object Using Triangle Pairs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(5):634–647, 1988.

[LJY^{+}07] M. Landon, D. R. Lancia Jr., J. Yu, S. C. Thiel, and S. Vajda. Identification of Hot Spots within Druggable Binding Regions by Computational Solvent Mapping of Proteins. *Journal of Medicinal Chemistry*, 50:1231–1240, 2007.

[LN89] D. C. Liu and J. Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45(3):503–528, 1989.

[MAN^{+}03] M. R. McGann, H. R. Almond, A. Nicholls, J. A. Grant, and F. K. Brown. Gaussian docking functions. *Biopolymers*, 68(1):76–90, 2003.

[Mat92] A. M. Mathiowetz. Dynamic and Stochastic Protein Simulations: From Peptides to Viruses. `http://www.wag.caltech.edu/publications/theses/alan/`, 1992.

[May08] M. May. PlayStation Cell Speeds Docking Programs. `http://www.bio-itworld.com/issues/2008/july-august/simbiosys.html`, 2008.

[Men06] O. Mencer. ASC: a stream compiler for computing with FPGAs. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 15(9):1603–1617, 2006.

[Men10a] Mentor Graphics. `http://model.com/`, 2010.

[Men10b] Mentor Graphics. Catapult C Synthesis Datasheet. `http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis/upload/Catapult_DS.pdf`, 2010.

[MGBK93] E. C. Meng, D. C. Gschwend, J. M. Blaney, and I. D. Kuntz. Orientational sampling and rigid-body minimization in molecular docking. *Proteins: Structure, Function, and Bioinformatics*, 17(3):266–278, 1993.

[MGH+98] G. M. Morris, D. S. Goodsell, R. S. Halliday, R. Huey, W. E. Hart, R. K. Belew, and A. J. Olson. Automated docking using a Lamarckian genetic algorithm and an empirical binding free energy function. *Journal of Computational Chemistry*, 19(14):1639–1662, 1998.

[Mic10] Microsoft. Microsoft DirectX. URL `http://www.microsoft.com/games/en-US/aboutGFW/pages/directx.aspx`, 2010.

[Mit10] Mitrionics. Mitrion C. `http://www.mitrionics.com`, 2010.

[MLP+06] D. T. Moustakas, P. T. Lang, S. Pegg, E. Pettersen, I. D. Kuntz, N. Brooijmans, and R. C. Rizzo. Development and validation of a modular, extensible docking program: DOCK 5. *Journal of Computer-Aided Molecular Design*, 20(10):601–619, 2006.

[MR96] C. Mattos and D. Ringe. Locating and characterizing binding sites on proteins. *Nature Biotechnology*, 14:595–599, 1996.

[MSK92] E. C. Meng, B. K. Shoichet, and I. D. Kuntz. Automated docking with grid-based energy evaluation. *Journal of Computational Chemistry*, 13(4):505–524, 1992.

[MST10] Molecular Surface Triangulation. URL:`http://www.cacr.caltech.edu/~sean/projects/stlib/html/mst/`, 2010.

[MWP+05] J. Mintseris, K. Wiehe, B. Pierce, R. Anderson, R. Chen, J. Janin, and Z. Weng. Protein-Protein Docking Benchmark 2.0: an update. *Proteins*, 60(2):214–216, 2005.

[Nal10] Nallatech. Nallatech. `http://www.nallatech.com/`, 2010.

[NOEM08] A. Nukada, Y. Ogata, T. Endo, and S. Matsuoka. Bandwidth intensive 3-D FFT kernel for GPUs using CUDA. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.

[NSc10] Bird Flu. `http://www.newscientist.com/topic/bird-flu`, 2010.

[NVI08a] NVIDIA. CUDA CUFFT Library, Programming Guide, Version 2.0, 2008.

[NVI08b] NVIDIA. NVIDIA CUDA Programming Guide, Version 2.0, 2008.

[NVI08c] NVIDIA. The CUDA Compiler Driver NVCC, Version 2.0, 2008.

[NVI10] NVIDIA. NVIDIA TESLA C1060. `http://www.nvidia.com/object/product_tesla_c1060_us.html`, 2010.

[NZB09] K. Nagar, Y. Zhang, and J. Bakos. An Integrated Reduction Technique for a Double Precision Accumulator. In *Proceedings of the Third International Workshop on High-Performance Reconfigurable Computing Technology and Applications*, pages 11–18, 2009.

[oPPS10] Skaggs School of Pharmacy and San Diego Pharmaceutical Sciences, University of California. PIER Server. `http://abagyan.ucsd.edu/PIER/`, 2010.

[Pec09] I. Pechan. Implementing a Global Optimization Algorithm Related To Bioinformatics with a High-Performance FPGA. In *Proceedings of the Sixteenth PhD Mini-Symposium, Budapest University of Technology and Economics*, pages 60–61, 2009.

[Pre05] President's Information Technology Advisory Committee. Computational Science: Ensuring Americas Competitiveness. URL: `http://www.nitrd.gov/pitac/reports/20050609_computational/computational.pdf`, 2005.

[Pym10] Pymol. `http://www.pymol.org/`, 2010.

[Raw04] M. D. Rawlins. Cutting the cost of drug development? *Nature Reviews Drug Discovery*, 3:360–364, 2004.

[RCS10] RCSB. `http://deposit.rcsb.org/adit/docs/pdb_atom_format.html`, 2010.

[Rit08] D. W. Ritchie. Recent Progress and Future Directions in Protein-Protein Docking. *Current Protein and Peptide Science*, 9:1–15, 2008.

[RK00] D. W. Ritchie and G. J. L. Kemp. Protein Docking Using Spherical Polar Fourier Correlations. *Proteins: Structure, Function, and Genetics*, 39(2):178–194, 2000.

[RKLK96] M. Rarey, B. Kramer, T. Lengauer, and G. Klebe. A fast flexible docking method using an incremental construction algorithm. *Journal of Molecular Biology*, 261(3):470–489, 1996.

[RSF07] M. P. Repasky, M. Shelley, and R. A. Friesner. Flexible ligand docking with Glide. *Current Protocols in Bioinformatics*, 8(12), 2007.

[RVD95] R. Rosenfeld, S. Vajda, and C. DeLisi. Flexible Docking and Design. *Annual Review of Biophysics and Biomolecular Structure*, 24:677–700, 1995.

[SBL10a] Boston University Structural Bioinformatics Lab. ClusPro protein-protein docking server. `http://cluspro.bu.edu/`, 2010.

[SBL10b] Boston University Structural Bioinformatics Lab. FTMap protein mapping server. `http://ftmap.bu.edu/`, 2010.

[SDDK07] D. E. Shaw, M. M. Deneroff, R. O. Dror, and J. S. Kuskin. Anton, a special-purpose machine for molecular dynamics simulation. In *Proceedings of the 34th annual International Symposium on Computer Architecture*, pages 1–12, 2007.

[SDINW05] D. Schneidman-Duhovny, Y. Inbar, R. Nussinov, and H. J. Wolfson. Geometry-based flexible and symmetric protein docking. *Proteins: Structure, Function, and Bioinformatics*, 60(2):224–231, 2005.

[SDKF10] R. Stote, A. Dejaegere, D. Kuznetsov, and L. Falquet. `http://vit-embnet.unil.ch/MD\_tutorial/`, 2010.

[SDV07] M. Silberstein, J. Damborsky, and S. Vajda. Exploring the binding sites of the haloalkane dehalogenase DhlA from Xanthobacter autotrophicus GJ10. *Biochemistry*, 46(32):9239–9249, 2007.

[SGAA$^+$08] H. Servat, C. Gonzlez-Alvarez, X. Aguilar, D. Cabrera-Benitez, and D. Jimnez-Gonzlez. Drug Design Issues on the Cell BE. In *High Performance Embedded Architectures and Compilers*, pages 176–190, 2008.

[SGI08] SGI. SGI Altix Family. `http://www.sgi.com/products/servers/altix/`, 2008.

[SGI10] SGI. Silicon Graphics International. `http://www.sgi.com/`, 2010.

[Sil04] Silicon Graphics, Inc. Extraordinary Acceleration of Workflows with Reconfigurable Application-Specific Computing from SGI. White Paper, 2004.

[SK96] M. Schaefer and M. Karplus. A Comprehensive Analytical Treatment of Continuum Electrostatics. *Journal of the Physical Chemistry*, 100(5):1578–1599, 1996.

[SKB92] B. K. Shoichet, I. D. Kuntz, and D. L. Bodian. Molecular docking using shape descriptors. *Journal of Computational Chemistry*, 13(3):380–397, 1992.

[SKJK05] T. Shiraki, T. S. Kodama, H. Jingami, and N. Kamiya. Rational discovery of a novel interface for a coactivator in the peroxisome proliferator-activated receptor gamma: theoretical implications of impairment in type 2 diabetes mellitus. *Proteins*, 58(2):418–425, 2005.

[SKLS$^+$10] K. Schulten, L.V. Kale, Z. Luthey-Schulten, E. Tajkhorshid, A. Aksimentiev, and C. Chipot. `http://www.ks.uiuc.edu/Research/namd/`, 2010.

[Sof10] OpenEye Scientific Software. FRED: Fast Rigid Exhaustive Docking. `http://www.eyesopen.com/products/applications/fred.html`, 2010.

[SRC10a] SRC. Series H MAP Processor. `http://www.srccomp.com/techpubs/docs/SRC_MAP_69226-BF.pdf`, 2010.

[SRC10b] SRC. SRC Carte Programming Environment. `http://www.srccomputers.com/techpubs/carte.asp`, 2010.

[STHH90] W. C. Still, A. Tempczyk, R. C. Hawley, and T. Hendrickson. Semianalytical treatment of solvation for molecular mechanics and dynamics. *Journal of the American Chemical Society*, 112(16):6127–6129., 1990.

[Swa87] E. Swartzlander. Systolic signal processing systems. Marcel Dekker, Inc., 1987.

[Syn10] Synopsys. Synplify Pro. `http://www.synopsys.com/tools/implementation/fpgaimplementation/fpgasynthesis/pages/synplifypro.aspx`, 2010.

[SZ09] S. Sun and J. Zambreno. A floating-point accumulator for FPGA-based high performance computing applications. In *Proceedings of the International Conference on Field-Programmable Technology*, pages 493–499, 2009.

[TB00] J. S. Taylor and R. M. Burnett. DARWIN: a program for docking flexible molecules. *Proteins*, 41(2):173–191, 2000.

[TJE02] R. D. Taylor, P. J. Jewsbury, and J. W. Essex. A review of protein-small molecule docking methods. *Journal of Computer-Aided Molecular Design*, 16(3):151–166, 2002.

[TJK01] M.L. Teodoro, G.N. Phillips Jr., and L.E. Kavraki. Molecular Docking: A problem with thousands of degrees of freedom. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 960–965, 2001.

[Tok10] Tokyo Institute of Technology. TSUBAME Grid Cluster with CompView. `http://www.gsic.titech.ac.jp/`, 2010.

[Top10] Top500. Tianhe-1, China's first Petaflop/s scale supercomputer. `http://www.top500.org/blog/2009/11/13/`, 2010.

[TS08] B. Tukora and T. Szalay. High Performance Computing on Graphics Processing Units. *Pollack Periodica*, 3(2):27–34, 2008.

[U.S10] U.S. Energy Information Administration. Average retail price of electricity. `http://www.eia.doe.gov/electricity/epm/table5_6_b.html`, 2010.

[Vaj10] S. Vajda. `http://structure.bu.edu/`, 2010.

[VC04] S. Vajda and C. J. Camacho. Protein-protein docking: is the glass half-full or half-empty? *Trends in Biotechnology*, 22(3):110–116, 2004.

[VG06] S. Vajda and F. Guarnieri. Characterization of protein-ligand interaction sites using experimental and computational methods. *Current Opinion in Drug Discovery and Development*, 9(3):354–362, 2006.

[VGH04] T. VanCourt, Y. Gu, and M. C. Herbordt. FPGA acceleration of rigid molecule interactions. In *Proceedings of the IEEE Conference on Field Programmable Logic and Applications*, 2004.

[VGMH06] T. VanCourt, Y. Gu, V. Mundada, and M. Herbordt. Rigid molecule docking: FPGA reconfiguration for alternative force laws. *Journal on Applied Signal Processing*, 2006:1–10, 2006.

[VK09] S. Vajda and D. Kozakov. Convergence and combination of methods in protein-protein docking. *Current Opinion in Structural Biology*, 19(2):164–170, 2009.

[Wen10] Z. Weng. ZDOCK Server. `http://zdock.bu.edu/`, 2010.

[WR97] H. J. Wolfson and I. Rigoutsos. Geometric Hashing: An Overview. *IEEE Computational Science and Engineering*, 4(4):10–21, 1997.

[XDI07] XDI. XD1000 Development System. `www.xtremedata.com`, 2007.

[XDI10] XDI. XtremeData, Inc. `http://www.xtremedata.com/`, 2010.

[Xil10] Xilinx. `http://www.xilinx.com/tools/logic.htm`, 2010.

[ZVCD97] C. Zhang, G. Vasmatzis, J. L. Cornette, and C. DeLisi. Determination of atomic desolvation energies from the structures of crystallized proteins. *Journal of Molecular Biology*, 267:706–726, 1997.

CURRICULUM VITAE

BHARAT SUKHWANI

Electrical and Computer Engineering
Boston University
8 Saint Mary's Street, ECE, Boston, MA
02215 USA

Phone: (508) 410-7480
`bharats@ieee.org`
`http://people.bu.edu/bharats/`

Education

- **PhD**, Electrical Engineering, Boston University, Boston, MA. June 2010
  Dissertation Title: GPA 3.96/4.0
  Accelerating Molecular Docking and Binding Site Mapping using FPGAs and GPUs
  Molecular Docking and Binding Site Mapping are two commonly used techniques in drug discovery. Due to their computational complexities, these algorithms require many hours of CPU runtimes and are usually run on large clusters. We design FPGA and GPU algorithms for accelerating the computations in these applications, leading to significant performance improvements, thus enabling desktop-based alternatives. The accelerated solutions have the added benefit of reduced power consumption. Moreover, the designed algorithms are general and can be applied to a variety of other applications.
- **M.S.**, Electrical and Computer Engineering, The University of Arizona 2005
  Tucson, AZ.
- **B.E.**, Electronics Engineering,
  Thadomal Shahani Engineering College, University of Mumbai 2001
  Mumbai, India

Professional Experience

- **Research Assistant**, Computer Architecture and Automated Design Lab
  Boston University, Boston, MA. Oct. 2009 – Jun. 2010
  Sep. 2007 – Jun. 2009, Sep. 2005 – Aug. 2006
- **Research Intern**, IBM T. J. Watson Research Center Jun. 2009 – Sep. 2009
  Yorktown Heights, NY.
- **Research Intern**, Microsoft Research Jun. 2007 – Aug. 2007
  Redmond, WA.
- **Teaching Fellow**, ECE Department Sep. 2006 – Apr. 2007
  Boston University, Boston, MA.
- **Research Assistant**, Digital VLSI Design Lab Jan. 2004 – Dec. 2004
  Department of Electrical and Computer Engineering
  The University of Arizona, Tucson, AZ.

- **Research Assistant**, Artificial Intelligence Lab Sep. 2002 – Dec. 2004
  Department of Management Information Systems
  The University of Arizona, Tucson, AZ.
- **Software Programmer**, Mastek Ltd Aug. 2001 – Jun. 2002
  Mumbai, India.
- **Presenter and Reviewer** 2006 – Present
  Reviewed papers for international conferences and journals.

Publications in Refereed Journals

- **B. Sukhwani** and M.C. Herbordt (2010). "FPGA Acceleration of Rigid-Molecule Docking Codes", *IET Computers & Digital Techniques*, 4(3), pp. 184-195.

- M.C. Herbordt, Y. Gu, T. VanCourt, J. Model, **B. Sukhwani** and M. Chiu (2008). "Computing Models for FPGA-Based Accelerators", *Computing in Science & Engineering*, 10(6), pp. 35-45.

- M.C. Herbordt, J. Model, **B. Sukhwani**, Y. Gu and T. VanCourt (2007). "Single Pass Streaming BLAST on FPGAs", *Parallel Computing: Special issue on High-Performance Computing Using Accelerators*, 33, pp. 741-756.

- J. M. Wang, **B. Sukhwani**, U. Padmanabhan, D. Ma and K. Sinha (2007). "Simulation and Design of Nanocircuits with Resonant Tunneling Devices", *IEEE Transactions on Circuits and Systems, Part I*, 54(6), pp. 1293-1304, June 2007.

- M.C. Herbordt, T. VanCourt, Y. Gu, **B. Sukhwani**, A. Conti, J. Model and D. DiSabello (2007). "Achieving High Performance with FPGA-Based Computing", *IEEE Computer*, 40 (3), pp. 50-57, March 2007.

Publications in Refereed Conference Proceedings

- **B. Sukhwani** and M.C. Herbordt (2010), "Fast Binding Site Mapping using GPUs and CUDA", *Proceedings of the Ninth International Workshop on High Performance Computational Biology (HiCOMB'10)*, April 2010.

- **B. Sukhwani** and M.C. Herbordt (2009), "FPGA-based Acceleration of CHARMM-potential Minimization", *Proceedings of the Third International Workshop on High-Performance Reconfigurable Computing Technology and Applications (HPRCTA'09)*, November 2009.

- **B. Sukhwani**, Matt Chiu, Md. Ashfaq Khan and M.C. Herbordt (2009), "Effective Floating Point Applications on FPGAs: Examples from Molecular Modeling", *Proceedings of the 2009 Workshop on High Performance Embedded Computing (HPEC'09)*, September 2009.

- **B. Sukhwani** and M.C. Herbordt (2009), "Accelerating Energy Minimization using Graphics Processors", *Proceedings of the 2009 Symposium on Application Accelerators in High Performance Computing (SAAHPC'09)*, July 2009.

- M.C. Herbordt, **B. Sukhwani**, Matt Chiu and Md. Ashfaq Khan(2009), “Production Floating Point Applications on FPGAs”, *Proceedings of the 2009 Symposium on Application Accelerators in High Performance Computing (SAAHPC’09)*, July 2009.
- **B. Sukhwani** and M.C. Herbordt (2009), “GPU Acceleration of a Production Molecular Docking Code”, *Proceedings of the Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU’09)*, ACM International Conference Proceeding Series, v383, pp. 19-27, 2009.
- **B. Sukhwani** and M.C. Herbordt (2008), “FPGA Acceleration of Production Rigid Docking Code”, *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL’08)*, September 2008.
- **B. Sukhwani**, A. Forin and N. Pittman (2008), “Extensible On-Chip Peripherals”, *Proceedings of the IEEE Symposium on Application Specific Processors (SASP’08)*, July 2008.
- M.C. Herbordt, Y. Gu, T. VanCourt, J. Model, **B. Sukhwani** and M. Chiu (2008), “Computing Models for FPGA-Based Accelerators with Case Studies in Molecular Modeling”, *Proceedings of the Reconfigurable Systems Summer Institute (RSSI’08)*, July 2008.
- **B. Sukhwani**, A. Forin and N. Pittman (2008), “I/O Subsystem of eMIPS Extensible Processor”, *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’08)*, April 2008.
- M.C. Herbordt, J. Model, Y. Gu, **B. Sukhwani** and T. VanCourt (2006), “Single Pass, BLAST-Like, Approximate String Matching on FPGAs”, *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’06)*, April 2006.
- **B. Sukhwani** and J. M. Wang (2005), “A Stepwise Constant Conductance Approach for Simulating Resonant Tunneling Diodes”, *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS’05)*, May 2005.
- **B. Sukhwani**, U. Padmanabhan and J. M. Wang (2005), “NanoSim: A Step Wise Equivalent Conductance based Statistical Simulator”, *Proceedings of the conference on Design, Automation and Test in Europe (DATE’05)*, March 2005.

Other Publications

- **B. Sukhwani**, A. Forin and N. Pittman (2007), “Extensible On-Chip Peripherals”, *MSR-TR-2007-120, Microsoft Research, Redmond, WA*, September 2007.
- **B. Sukhwani**, T. VanCourt, J. Model and M. C. Herbordt (2007), “Environments for Implementing Scalable Families of High Performance Applications”, *Boston University Annual Science and Engineering Research Symposium*, April 2007.

Honors and Awards

- Summer Fellowship, IBM T. J. Watson Research Center 2009
- Summer Fellowship, Microsoft Research 2007
- Dean's Research Fellowship, College of Engineering, Boston University 2005 – 2006
- Graduate Tuition Scholarship, The University of Arizona 2002 – 2003
- Sir Ratan Tata Trust Scholarship for Undergraduate Studies, Mumbai, India (50 out of 8000 students selected) 2000
- Outstanding Performance Award in Analytical section of the Graduate Record Examination (GRE) (Top 10 worldwide) March, 2001
- Ranked 25th among 1722 students of the University of Mumbai BE (Electronics) degree examination 2001

Technical Competence

- Operating Systems: Windows, Linux, Mac OS X
- Programming Languages: C, C++, JAVA, Perl, SQL, UML
- Hardware Description Languages: VHDL, Verilog HDL, SystemC, Handel-C
- Parallel Programming: OpenMP, MPI, NVIDIA CUDA
- Engineering Tools: MATLAB, Xilinx ISE, Altera Quartus, ModelSim, ChipScope, Cadence, LaTeX

Last updated: July 19, 2010