Springer Series in Advanced Microelectronics 55

B. Sharat Chandra Varma
Kolin Paul
M. Balakrishnan

Architecture Exploration of FPGA Based Accelerators for BioInformatics Applications

Springer

Springer Series in Advanced Microelectronics

Volume 55

The Springer Series in Advanced Microelectronics provides systematic information on all the topics relevant for the design, processing, and manufacturing of microelectronic devices. The books, each prepared by leading researchers or engineers in their fields, cover the basic and advanced aspects of topics such as wafer processing, materials, device design, device technologies, circuit design, VLSI implementation, and subsystem technology. The series forms a bridge between physics and engineering and the volumes will appeal to practicing engineers as well as research scientists.

More information about this series at http://www.springer.com/series/4076

B. Sharat Chandra Varma
Kolin Paul · M. Balakrishnan

Architecture Exploration of FPGA Based Accelerators for BioInformatics Applications

Springer

B. Sharat Chandra Varma
Department of Electrical and Electronic Engineering
The University of Hong Kong
Hong Kong
Hong Kong

Kolin Paul
Department of Computer Science and Engineering
Indian Institute of Technology Delhi
New Delhi, Delhi
India

M. Balakrishnan
Department of Computer Science and Engineering
Indian Institute of Technology Delhi
New Delhi, Delhi
India

ISSN 1437-0387 ISSN 2197-6643 (electronic)
Springer Series in Advanced Microelectronics
ISBN 978-981-10-0589-3 ISBN 978-981-10-0591-6 (eBook)
DOI 10.1007/978-981-10-0591-6

Library of Congress Control Number: 2016931344

Printed on acid-free paper

This Springer imprint is published by SpringerNature
The registered company is Springer Science+Business Media Singapore Pte Ltd.

To the Almighty
To my family and friends

Preface

Rapid advances in VLSI technology have enabled fabrication of billions of transistors on a single chip. Technology scaling has allowed more than one processor core to be integrated in a single chip. The current computing systems, including desktops, laptops, and mobile phones have many-core processor chips. Software applications are parallelized to execute on multiple processing units/cores concurrently to reduce the overall execution time. Although sophisticated compute systems have been developed for fast execution, software applications belonging to certain domains take a very long time (days to months) for producing results.

Bioinformatics is one such domain in which applications take a long time to execute. It is also a multidisciplinary research field consisting of computer science, mathematics, and statistics, which deals with storing, analyzing, and interpreting large biological data. The recent advancements in biological research has resulted in generation of large amounts of digital data. Most of the bioinformatics algorithms are both data intensive and compute intensive. Speedups can be obtained using specialized hardware along with the existing processor architectures.

General-purpose processors are often augmented with hardware accelerators for compute intensive application to reduce the computation time. Typical hardware accelerators consist of a large number of small execution units which facilitate parallel execution. Very often they represent transformation of loops in a sequential code (temporal iteration) to spatial unrolling of the loop (spatial iteration) to reduce the computation time through concurrent execution. Some of the popular hardware accelerators are GPUs, FPGAs, and CELL processors. The accelerators are programmable and hence can be used for a variety of related applications. FPGA-based accelerators are known to be effective in speeding up certain kinds of applications when compared to other accelerators. Since FPGAs are configurable, they can be customized to implement a variety of processing elements as accelerators. But speedup may not always be possible as FPGAs run at slower clock frequency vis-a-vis processors and the resources available in the FPGA might not be sufficient for the implementation of a significant number of copies of the processing elements. FPGAs with heterogeneous mix of coarse grained hard blocks along with

programmable soft logic, can facilitate implementation of a much larger number of processing elements and thus achieve higher speedups.

FPGAs have evolved and the hardware blocks useful for many applications are being implemented within the FPGA fabric as Hard Embedded Blocks (HEBs). Introduction of HEBs in FPGA can significantly increase the performance of FPGA-based accelerators. Modern FPGAs contain specialized embedded units like memory units, array multipliers, DSP computation units, etc. In fact many FPGAs have embedded processor cores. Based on the application a matching FPGA with the right HEBs is chosen. For example in Xilinx Virtex-4 FPGAs, the SX-series has only LUTs, in the LX-series there are DSP units as HEBs and the FX-series have Power-PC embedded in them. The user can choose the best suited FPGA architecture according to his/her needs. It is easy to predict that many more such hard blocks will be embedded into future FPGAs.

The evaluation approach to identify and incorporate HEBs is complex as there are many parameters and constraints such as area, granularity routing resources, etc., which need to be considered in an integrated manner to get efficient implementation. Incorporating HEBs in FPGA involves a clear tradeoff as this may occupy a significant area and hence may reduce the configurable logic. Further, they may not be usable for many applications. On the other hand, they may give very significant speedups for certain applications. Clearly, the challenge is to identify kernels that are useful for a class of applications and justify designing customized HEBs that are effective in significantly speeding up an important class of applications. There is a need to develop a methodology to explore the FPGA fabric design space to evaluate the nature and number of HEBs based on other constraints.

This book presents an evaluation methodology to design future FPGA fabrics incorporating hard embedded blocks to accelerate applications. This methodology is useful for selection of blocks to be embedded into the fabric and for evaluating the performance gain that can be achieved by such an embedding. The use of the methodology is illustrated by designing FPGA-based acceleration of two important bioinformatics applications: Protein docking and Genome assembly. This book explains how the respective HEBs are designed and how hardware implementation of the application is done using these HEBs. The impact of use of HEBs on accelerating these two applications is shown. The methodology presented in this book may also be used for designing HEBs for accelerating software implementations in other domains as well.

We thank Prof. Dominique Lavenier for letting us collaborate with the SYMBIOSE lab, IRISA, Rennes, France. We are also grateful to him for providing us guidance on this project. We thank Pierre Peterlongo for his valuable inputs on Mapsembler.

Hong Kong — B. Sharat Chandra Varma
New Delhi, India — Kolin Paul
New Delhi, India — M. Balakrishnan

Contents

About the Authors

B. Sharat Chandra Varma is Research Associate in the Department of Electrical & Electronic Engineering at The University of Hong Kong, Hong Kong. He obtained his Ph.D. from IIT Delhi, New Delhi. He completed his B.E. (2003) from Visvesvaraya Technological University Karnataka and M.Sc. (2007) from Manipal University, Karnataka. His research interest areas include re-configurable computing, FPGA, hardware-software co-design, hardware accelerators, and computer architecture. Dr. Varma has over 10 years of research and industrial experience and has several journal publications to his credit.

Kolin Paul is presently Associate Professor at the department of Computer Science and Engineering at Indian Institute of Technology Delhi, New Delhi. His academic degrees include M.Sc. (1995) from Jadavpur University and Ph.D. (2002) from Bengal Engineering College, Calcutta, India. Dr. Paul has published several papers in refereed journals and conference proceedings.

M. Balakrishnan is Professor in the CSE Department at IIT Delhi, New Delhi, India. He completed his Ph.D. in 1984 from IIT Delhi. He has 16 journal articles, 63 refereed conference publications, one book chapter and two patents to his credit. Dr. Balakrishnan has been a part of many research projects and consultancies from leading EDA/VLSI companies. He is a reviewer for major international journals, a member of key academic bodies. He has supervised 8 Ph.D.'s, 3 MS(R), and 78 M.Tech students. His areas of specialization include behavioral and system level synthesis, system level design and modeling, computer architecture, hardware-software co-design, embedded system design, and assistive devices for the visually impaired.

Acronyms

API	Application Program Interface
ASIC	Application Specific Integrated Circuit
CAD	Computer Aided Design
CLB	Configurable Logic Block
DNA	DeoxyriboNucleic Acid
DSE	Design Space Exploration
DSP	Digital Signal Processing
ECS	Electrostatic Complementarity Score
FFT	Fast Fourier Transform
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FTDock	Fourier Transform Dock
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HEB	Hard Embedded Blocks
LUT	Lookup Table
MSA	Multiple Sequence Alignment
NGS	Next Generation Sequencing
OLC	Overlap Layout Consensus
PC	Personal Computer
PCB	Printed Circuit Board
PDB	Protein Data Bank
PE	Processing Element
RAM	Random Access Memory
SCS	Shape Complementarity Score
SIMD	Single Instruction Multiple Data
SOC	System On Chip
VEB	Virtual Embedded Block
VLSI	Very Large-Scale Integration
VPR	Versatile Place and Route

Chapter 1
Introduction

Abstract Application-specific integrated circuits (ASICs) are specialized custom-designed circuits which are developed to carry out desired tasks efficiently in hardware. Often, microprocessors are preferred over ASICs, since they give flexibility to users. They allow the same hardware to be used for a variety of applications. Still for applications requiring very high speed computation and/or very low energy ASICs have been preferred over software solutions. Microprocessors are typically based on Von Neumann architecture, which allow execution of stored programs (Von Neumann, IEEE Ann. Hist. Comput. 15(4): 27–75, 1993). For implementing a specific application, the user writes software programs to specify the sequence of tasks that gets executed within the processor. Rapid advances in VLSI technology have enabled fabrication of billions of transistors on a single chip. Technology scaling has allowed number of transistors to double every 18 months in accordance to Moore's law (Moore, Prod. IEEE 86(1):82–85, 1998). This technological advancement has led to design and development of faster and energy-efficient hardware. Availability of faster processors enabled software based solutions to replace hardware solutions over increasingly larger domain. In the past few years, frequency scaling of processors has saturated due to thermal limitations and the integrated circuit (IC) designers are focusing on gaining speedups by running more operations concurrently in hardware; either on multi-core processors or on specialized hardware.

1.1 Background

The current computing systems including desktops, laptops, and mobile phones have many-core processor chips. In synergy with this approach, general purpose processors are often augmented with hardware accelerators for compute intensive applications to reduce the computation time. Typical hardware accelerators consist of a large number of small execution units which facilitate parallel execution. Very often they represent transformation of loops in a sequential code (temporal iteration) to spatial unrolling of the loop (spatial iteration) to reduce the computation time through concurrent execution. Some of the popular hardware accelerator architectures are based on GPUs and CELL processors. FPGAs with hardware reconfigurability is another

B.S.C. Varma et al., *Architecture Exploration of FPGA Based Accelerators for BioInformatics Applications*, Springer Series in Advanced Microelectronics 55, DOI 10.1007/978-981-10-0591-6_1

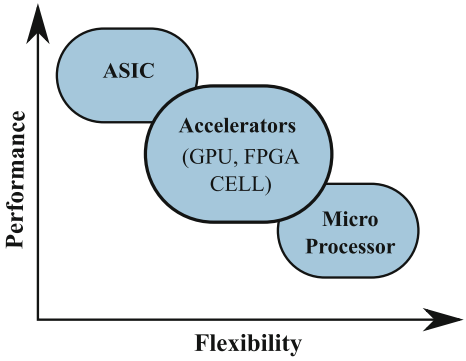

Fig. 1.1 Performance versus flexibility of compute systems

interesting option for building hardware accelerators. FPGA-based accelerators are programmable and hence can be used for a variety of related applications. Although sophisticated compute systems have been developed for fast execution, software applications belonging to certain domains still take very long time (days to months) for producing results. Moore's law has driven semiconductor industry to implement very efficient larger circuits on a single chip [1]. This has enabled design of better compute systems than the traditional Von Neumann based systems [2].

Software solutions are parallelized to execute on multiple processing units/cores concurrently to reduce the overall execution time. The performance versus flexibility/programmability of various compute systems is shown in Fig. 1.1. The ASICs have the least flexibility, but give the best performance both in terms of time and energy. The hardware accelerators come below ASICs in terms of performance. They are all programmable and hence can be used for various applications, but yet require "accelerator-specific coding" to reap the full benefits of the underlying hardware architecture. For example, the NVidia GPUs require CUDA implementation, and CELL processors require special multithreaded code to run on them whereas HDL coding has to be done to implement the hardware. Researchers are working towards a unified language like OpenCL which allows easy programmability as well as portability between different architectures [3]. The microprocessors are easy to program, and hence are more popular. Depending on the application characteristics, specific accelerators may outperform other accelerators. Selecting a suitable accelerator and doing an efficient implementation/programming is an important design decision when performance as a key objective. For floating point applications, GPUs or CELL may be preferred and for custom accelerator designs, FPGAs are preferable.

FPGA based accelerators are known to be effective in speeding up certain kind of applications when compared to other accelerators [4, 5]. Since FPGAs are reconfigurable, they can be customized to implement a variety of processing elements as accelerators. Systolic array based hardware implementations are naturally suitable for mapping onto FPGAs because of simple cell structures and local communication. Further its high degree of parallelism coupled with pipelined implementation with local communication mapped onto short interconnects can give very high speedups

over software solutions. Custom hardware design and deep-pipelines, when implemented in FPGAs also provide performance benefits. But speedup always may not be possible as FPGAs run at slower clock frequency vis-a-vis processors and the resources available in the FPGA may not be sufficient for the implementation of a significant number of instantiation of the processing elements to exploit the available degree of parallelism. Recent FPGAs contain a heterogeneous mix of coarse grained hard blocks along with programmable soft logic, facilitating implementation of a much larger number of processing elements. This can translate into much higher speedups as well.

FPGAs have evolved and the hardware blocks useful for many applications are being implemented within the FPGA fabric as hard embedded blocks (HEBs). Introduction of HEBs in FPGA can significantly increase the performance of FPGA-based accelerators. Modern FPGAs contain specialized embedded units like memory units, array multipliers, DSP computation units, etc. In fact many FPGAs have embedded processor cores. Based on the application requirements, a matching FPGA with the right HEBs is chosen. For example in Xilinx Virtex-4 FPGAs, the SX-series has only LUTs, in the LX-series there are DSP units as HEBs and the FX-series have PowerPC embedded in them [6]. The user can choose the best suited FPGA architecture according to his/her needs. It is easy to predict that many more such hard blocks will be embedded into future FPGAs [7, 8].

FPGA-based accelerators have been successful in speeding up bioinformatics applications which take a long time to execute [9, 10]. Bioinformatics is a multidisciplinary research field consisting of computer science, mathematics, and statistics which deals with storing, analyzing and interpreting large biological data. The recent advancement in biological research has resulted in the generation of large amounts of digital data. Most of the bioinformatics algorithms are both data intensive as well as compute intensive. Two important bioinformatics applications of our interest are protein docking and genome assembly.

1.1.1 Protein Docking

Study of 3D structure of both ligand and protein together helps in drug design. Molecular docking is a process by which two molecules fit together in a 3D space. Docking tries to predict the structure of the complex formed from these two molecules. This process takes a very long time to be done chemically and hence computer software simulations are used. The protein docking program ranks the protein–ligand complex based on scoring functions. Scoring functions predict the strength of the complex. The high scoring drug molecules from a large library of drug molecules are then selected as probable drug molecules. A library may consist of more than one million drug molecules [11].

FTDock is an open-source protein docking software application, which is used to study protein–protein binding as well as protein–ligand binding [12]. A correlation function is used to do the scoring. The correlation function is expensive to compute

in software and hence fast Fourier transform (FFT) is used to reduce the number of multiplications. Even though the application uses FFTW library, which is an efficient implementation of FFT, it takes large amount of time to execute [13]. For example docking of two proteins, barnase (1a2p)[1] and barstar (1a19) using FTDock application took 21 h to execute on a desktop PC with Intel Core 2 duo E4700, 2.6 GHz processor with 4 GB RAM.

1.1.2 Genome Assembly

Genome is the set of all genes in an organism. The study of genome enables us to understand the various functions in an organism. It also helps identifying defects and diseases caused due to small changes in the genes known as mutation. The genome is represented by a long stream of alphabets, "A", "C", "T" and "G". These alphabets represent deoxyribonucleic acids (DNA), which encode the genes.

Sequencing technologies are used to determine the order of nucleobases in the DNA. Primitive sequencing methods were very slow and were very expensive. Next-generation sequencing (NGS) technologies produce large amounts of data at very low cost [14]. The NGS machines take a biologically processed sample and generate large number of sequences of fixed length up to 450 known as "**reads**". These reads are a part of the original long genome. The computation challenge is to construct the original genome from the reads produced by the NGS machine. In de novo genome assembly, the genome is constructed using the overlap information available in the reads. Like most other bioinformatics applications, genome assembly is also data dominated. For example, the size of uncompressed human genome of length three billion base pairs is 3 GB and is 750 MB when compressed. The reads contain overlap information and hence the size of input file given to assembly softwares for assembling human genome is often more than 30 GB. Even though compute infrastructure ranging from server racks to cloud farms exist for solving these problems, the time taken is enormous. For example assembly of human genome using PASHA software took around 21 h on a 8-core workstation with 72 GB memory [15].

Clearly, there is a need to accelerate these applications, as they take significant amount of time to execute. Software applications can be accelerated using hardware accelerators based on existing FPGAs. Further speedups may be obtained by using new reconfigurable fabrics with custom-designed HEBs. The methodology to identify and evaluate appropriate HEBs to be incorporated is complex as there are many parameters and constraints like area, granularity, routing resources, etc. that need to be considered in an integrated manner to get an efficient implementation. Design space exploration (DSE) of such future fabrics is critical to the design of these HEBs and in turn design of FPGA-based accelerators.

[1] The protein structures are denoted by their four letter code in the Protein Data Bank.

1.2 DSE of Application Acceleration Using FPGAs with Hard Embedded Blocks

The design of new hardware for performing a set of tasks involves area, delay and power trade-offs. Considering application acceleration as the main goal, many more design choices are added into this pool. The design choices should have a balance between the various factors and a trade-off consensus is required to identify the best suited design for a particular domain or set of applications. Figure 1.2 shows the flow diagram for carrying out DSE for carrying out application acceleration. A typical flow used for accelerating application using FPGAs is shown on the left and the design of HEBs is shown on the right. The goal of the DSE is to design new reconfigurable fabrics incorporating HEBs to achieve better performance over existing FPGAs.

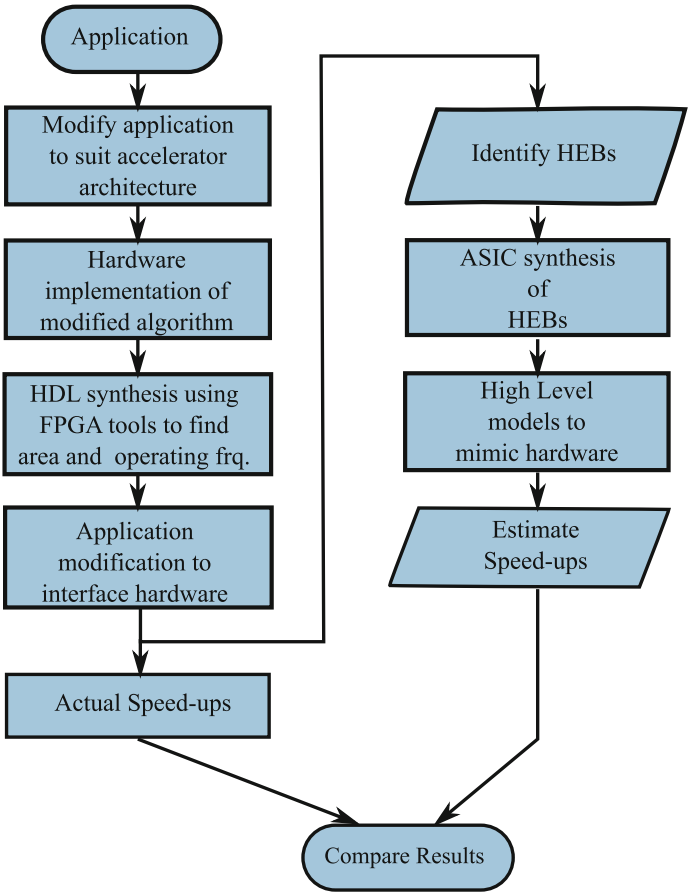

Fig. 1.2 DSE of FPGA-based accelerators

Initially, accelerator-specific changes are done to the chosen algorithm. The kernels are identified and implemented in hardware using HDLs. The application is modified to interface with these kernels. This defines the data transfers between the software and the hardware platform. Actual speedups can be measured on existing platforms (boards/cards having existing FPGAs).

Performance estimation is often needed to predict the performance on platforms that are yet to be designed. Using these estimates, probable HEBs have to be identified and evaluated. ASIC synthesis of these HEBs has to be carried out to estimate the area occupied and to predict the operating frequency. For some of the applications, estimating speedups is straight forward and hence analytical models can be used, whereas for others it is difficult to build analytical models and hence simulation models have to be used to mimic hardware behavior. System-level simulation models are used to estimate the speedups using fabrics incorporating HEBs. These simulation models can be built at various levels of abstraction to carry out DSE.

1.2.1 Challenges in Design Space Exploration

The key issues to be addressed for DSE of FPGAs with HEBs are:

1. Application modification to suit accelerator architecture
2. HEBs identification
3. FPGA design incorporating these HEBs
4. Application mapping on hardware
5. Performance estimation

Applications are usually designed to perform efficiently on a generic system. When accelerators are used, accelerator-specific modifications help in getting performance benefits. For example, in FPGAs, data types and bit width chosen can have significant impact on the performance. Fixed point computation units occupy significantly less resources than floating point units and hence number of concurrent executions can be increased using the same hardware resources resulting in increased performance. Based on the memory available in the accelerator architecture, modifications that trade-off "local" memory with "remote" memory can be carried out on the application to get better speedups. Applications can also be modified to take advantage of the HEBs, as they use less amount of silicon area and run at significantly higher clock rates when compared to implementations based only on configurable logic, i.e., CLBs/LUTs in case of Xilinx FPGAs.

Another important factor influencing performance is the accelerator architecture design. Incorporating HEBs in FPGA involves a clear trade-off as this may occupy significant area and hence reduce space available for configurable logic. Further, they may not be usable for many applications. On the other hand, they may give very significant speedups for specific applications. Clearly the challenge is to identify kernels that are useful for a class of applications and justify designing customized HEBs that are effective in significantly speeding up this class of applications. The right choice

of the number of units and the type of units will not only be able to give performance benefits but also cost benefits. The various factors affecting this choice are the application/s computation requirements, internal memory requirements, external memory bandwidth and the application's post-processing requirements. Efficient mapping of the application also helps in arriving at suitable values for some of these parameters.

Application mapping on hardware is a typical hardware–software co-design problem, where the user with the aid of tools decides the portions of the application to be run in software and the portions that should be run in hardware to get maximum speedups under resource constraints. The kernels that take most of the execution time in software have to be identified by profiling the application. Memory profiling has to be done to estimate the communication delay between host and the FPGA. Memory profiling also helps in deciding the use of internal memories (BRAMs) available in the FPGA that effectively creates a memory hierarchy. Wherever possible, multiple copies of the kernels have to be executed in parallel in order to get speedups. It is in this context that HEBs have to be efficiently used in the mapping process to achieve speedups. The kernels which share data have to be placed close to each other to avoid communication delays. Mapping efficiency can be assessed by using performance estimation.

Performance estimation is critical as HEBs are typically an expensive solution. It is well known that hardware–software partitioning can be efficiently explored using performance estimation. Performance estimation can be carried out by building simulation models at various levels of abstraction. Typically, the models at higher level of abstraction take considerably less amount of time than the ones at lower levels and thus can be used to explore most of the design space. On the other hand, the lower level models give more accurate performance numbers when compared to higher level models. The right choice of parameters to be studied at different levels of abstraction based on accuracy and simulation time will have significant impact on design turn-around. In the context of HEBs, performance estimates help in the modification of application, identification and design of HEBs, as well as mapping applications onto FPGAs incorporating HEBs.

Use of HEBs can increase the operating frequency of the hardware implementation. Custom HEBs do a dedicated operation, but occupy less silicon chip area compared to CLB/LUT implementation and hence the FPGA resources can be used optimally for the rest of the operations. Typically, the coarser HEBs are IP cores with less programmability like the PCI interface logic, which are common to a wide range of applications. The finer grained HEBs like BRAMs provide more flexibility, so that they can be used as small modules as well as built into large modules to meet the design requirements. A right combination of the HEBs along with the LUTs and switches can provide performance benefits without losing the programmability (flexibility) of FPGAs. Keeping all these key factors into consideration, there is a need to develop a methodology to explore the FPGA fabric design space to evaluate nature and number of HEBs based on other constraints.

In rest of the chapters we consider application and accelerator architecture in tandem and show a methodology to evaluate accelerators using FPGAs with custom HEBs. We use bioinformatics as the application domain, since the domain consists of applications having very high computational requirements.

References

1. Moore, G.: Cramming more components onto integrated circuits. Proc. IEEE **86**(1), 82–85 (1998)
2. Von Neumann, J.: First draft of a report on the EDVAC. IEEE Ann. Hist. Comput. **15**(4), 27–75 (1993)
3. Stone, J., Gohara, D., Shi, G.: Opencl: a parallel programming standard for heterogeneous computing systems. Comput. Sci. Eng. **12**(3), 66–73 (2010)
4. Che, S., Li, J., Sheaffer, J., Skadron, K., Lach, J.: Accelerating compute-intensive applications with GPUs and FPGAs. In: Symposium on Application Specific Processors, 2008. SASP 2008, pp. 101–107 (2008)
5. Chung, E., Milder, P., Hoe, J., Mai, K.: Single-chip heterogeneous computing: does the future include custom logic, FPGAs, and GPGPUs? In: Annual IEEE/ACM International Symposium on Microarchitecture (MICRO) (2010)
6. Xilinx: Xilinx FPGAs, ISE. http://www.xilinx.com (2015)
7. Beauchamp, M.J., Hauck, S., Underwood, K.D., Hemmert, K.S.: Embedded floating-point units in FPGAs. In: ACM/SIGDA International Symposium on FPGAs (2006)
8. Yu, C.W., Smith, A., Luk, W., Leong, P., Wilton, S.: Optimizing coarse-grained units in floating point hybrid FPGA. In: International Conference on FPT (2008)
9. Jain, A., Gambhir, P., Jindal, P., Balakrishnan, M., Paul, K.: Fpga accelerator for protein structure prediction algorithm. In: 5th Southern Conference on Programmable Logic, 2009, SPL (2009)
10. Tang, W., Wang, W., Duan, B., Zhang, C., Tan, G., Zhang, P., Sun, N.: Accelerating millions of short reads mapping on a heterogeneous architecture with fpga accelerator. In: Annual IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 184–187 (2012)
11. Zhu, T., Cao, S., Su, P.C., Patel, R., Shah, D., Chokshi, H.B., Szukala, R., Johnson, M.E., Hevener, K.E.: Hit identification and optimization in virtual screening: practical recommendations based on a critical literature analysis. J. Med. Chem. **56**(17), 6560–6572 (2013)
12. Sternberg, M.J.E., Aloy, P., Gabb, H.A., Jackson, R.M., Moont, G., Querol, E., Aviles, F.X.: A computational system for modeling flexible protein-protein and protein-DNA docking. In: Proceedings of the 6th International Conference on Intelligent Systems for Molecular Biology, pp. 183–192 (1998)
13. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. In: Proceedings of the IEEE (2005)
14. Ayman, G., Kate, W.: Next-generation sequencing: methodology and application. J. Invest. Dermatol. **133** (2008)
15. Liu, Y., Schmidt, B., Maskell, D.: Parallelized short read assembly of large genomes using de Bruijn graphs. BMC Bioinform. **12**(1), 354–363 (2011)

Chapter 2
Related Work

Abstract In this chapter we discuss the related work, where we present some of the ideas and implementations reported by other researchers working in areas closely related to this field. First we describe the various accelerator architectures and then, discuss FPGA based accelerators. We describe the FPGA architecture as well as the EDA tool flow followed while exploring HEBs in FPGAs. We discuss "bioinformatics" domain and the two important applications belonging to this domain. We show how these applications have benefited by FPGA-based acceleration.

2.1 Accelerator Architectures

Accelerators have become very popular in the community in the past decade. Popular implementations are based on ASICs, FPGAs, GPUs, and CELL add-ons. Heterogeneous accelerators consist of a mix of accelerators which are connected to the control processor. Each of the approaches listed here have their own strengths and weaknesses.

Chung et al. [22] study the importance of heterogeneous architectures. They evaluate different architectures by running various benchmarks and predict the performance of future architectures using scaling factors [43]. They predict that the future systems will include a mix of various architectural features in order to have a balance between power and speed. Venkatesh et al. [85] predict that the future chips will have large amounts of "Dark Silicon". As the operating frequency of IC is increased, the chip gets heated up and some parts of the IC have to be shut down to avoid permanent damage. The parts which are shut-down are known as dark silicon. The authors propose reconfigurable "conservation cores" that operate at lower frequencies during the shut-down of the faster cores. Even though these conservation cores have less operating frequency, they carry out specialized computations parallel and thus provide performance benefits over chips that do not have them.

In ASICs, hardware implementation of multiple copies of the compute intensive kernels is done. ASIC implementation of the kernels will give higher performance compared with other accelerators in terms of power, speed, and area. Kuon et al. [50] have studied the gap between designs implemented in FPGA and ASIC. For 90 nm

B.S.C. Varma et al., *Architecture Exploration of FPGA Based Accelerators for BioInformatics Applications*, Springer Series in Advanced Microelectronics 55, DOI 10.1007/978-981-10-0591-6_2

technology, they report that the FPGA implementations are 5× slower and occupy 35× more area compared to ASIC implementations. They also report that dynamic power of FPGA implementations is 14 times that of ASIC. They also report that by using the HEBs in FPGAs, these ratios can be decreased. Even though ASICs provide high performance they are not popular as accelerators because ASIC implementations have long "design time" and do not offer much flexibility once the implementation is done. One more disadvantage of ASICs as accelerator is that the cost of producing them at low volumes is very high. As flexibility offered by ASICs is low, the demand for such specific accelerator chips is low but also their "life" is also low due to changes in standards as well as applications. FPGAs are cheaper compared to ASICs as they offer a lot of flexibility due to their reprogrammability. The hardware can be reconfigured for different kernel implementations. These advantages make FPGA one of the most popularly used accelerators.

Recently GPUs and CELL as accelerators are being extensively used as accelerators. These accelerators have a large number of floating point arithmetic units as cores which take advantage of single instruction multiple data (SIMD) processing. Many researchers have compared various accelerator architectures [16, 32, 35, 46]. Most of them report that GPUs and CELL are useful for floating point computations and FPGAs perform well on fixed point or integer computations. Further, FPGAs outperform other accelerators in some domains including cryptography.

2.1.1 Challenges in Designing Accelerators

The accelerators differ mainly on parameters such as price, power, productivity, performance, and ease of programming. Flexibility is an important aspect while deploying accelerators. FPGAs give more "configuration" flexibility vis-a-vis GPUs and CELL, but run at relatively lower clock speeds. The power consumption of GPUs is relatively high and can pose problem in cluster deployment.

Sanjay et al. [70] discuss the various issues related to the design of accelerators. They report that the main concern with accelerators is the lack of standard models in both architecture and software programming. This creates portability, scalability, and software management issues. A common programming language which allows ease of programming and standard interface design between host and accelerator to allow ease of data transfers across memory hierarchies can solve these problems.

The major obstacle for wider use of accelerators is that they require extra programming effort, which leads to long learning curves and hence result in increase in design time as well as cost. As the hardware of GPUs, CELL, and multicore CPUs are fixed and not designed from the perspective of bioinformatics applications, the software needs to be "tailored" and "partitioned" to utilize the resources efficiently for getting high performance. Each of the accelerators provides different programming interfaces and extensions. For designing an accelerator using FPGAs, we have to do a custom design of hardware.

2.2 FPGA-Based Acceleration

FPGAs are integrated circuits consisting of an array of logic blocks interconnected by routing resources which are programmable. Depending on their architecture they can be programmed either once or many times. The popular FPGA players presently in the market are Xilinx and Altera, which are based on SRAMs and thus easily reprogrammable [4, 92]. The other kind of FPGAs are based on anti-fuse, EPROM based [72]. Many other architectures based on emerging technologies have been proposed by many research groups [27, 83, 95]. The SRAM-based FPGAs are reprogrammable and hence are used as custom accelerators for various time-consuming applications. Knowing the underlying architecture of FPGAs, will help design better accelerators.

2.2.1 FPGA Architecture

A simple FPGA architecture is as shown in Fig. 2.1 [13]. The configurable logic blocks (CLBs) as well as input/output (I/O) blocks are programmable and all these are interconnected through programmable routing channels. A basic CLB is shown in Fig. 2.2. It consists of LUT, a flip flop and a mux. The mux is used to choose between direct output from the LUT or the registered LUT output. The respective output for an input combination is stored in the LUT. A combination of such CLB units is used to construct any desired logic circuit. A connect box is used to connect the CLB outputs to the routing channels. A connect box is used to interconnect various routing resources.

SRAM-based FPGAs are programmed using bitstreams. FPGA design flow is shown in Fig. 2.3. The flow shows the various steps involved in hardware implementation that culminates in generation of bitstream starting from HDL or schematics. High-level design languages like C are also being used to describe hardware. The

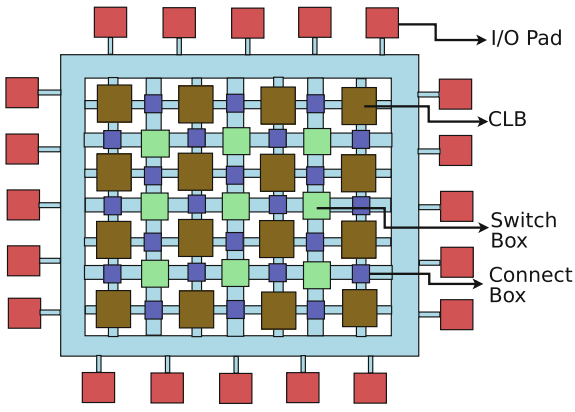

Fig. 2.1 FPGA architecture

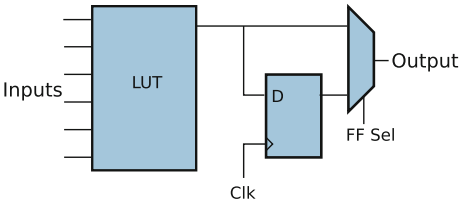

Fig. 2.2 Basic CLB structure

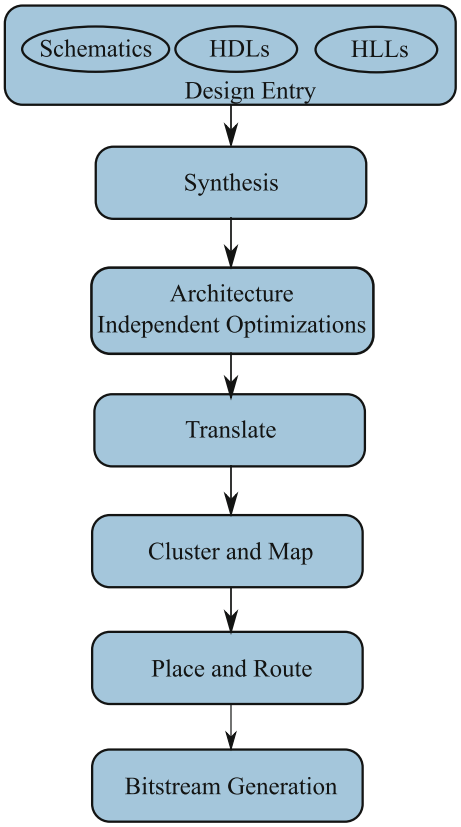

Fig. 2.3 FPGA design flow

synthesis tools do optimizations on the hardware described and convert it into gate level net-list. The translate tool packs these gates into LUTs. The place and route tool places the LUTs in an optimized way so as to have a less critical path. Typically, this is achieved by placing the LUTs representing the combinational logic (between two registers) close to each other. The routing tool defines the interconnectivity between the LUTs. This tool optimizes the use of long and short wires such that minimum number of switch blocks is used. Then a bitstream to program each of the LUTs and the interconnect are generated. The FPGA is programmed using this bitstream.

Lots of research have been carried out on FPGA architecture both by academia and industry. Several research groups have reported results on deciding the optimal LUT size [1, 34, 51, 75]. Larger LUTs reduce the interconnect delays but occupy more area and are slower because of the internal circuitry needed to decode the output. The optimal LUT size as reported by many researchers lies between 4 and 6. It is mostly dependent on the size of clustering of CLBs and the channel width. The present commercial FPGAs have LUT sizes up to 8. Xilinx has six input fracturable LUTs, whereas Altera has eight input fracturable LUTs [4, 92]. As the name suggests, "fracturable LUTs" can be broken and used as two smaller size LUTs.

Advances in FPGA-based reconfigurable computing (RC) technology over the past decade has allowed researchers to exploit performance, power, area, cost and versatility advantages in FPGA devices as compared to conventional microprocessor and ASICs [50]. More than a thousand-fold parallelism can be achieved for low-precision computation [37, 38]. Previously FPGAs were used for glue logic between incompatible systems, but now as technology has advanced much more logic can be implemented on an FPGA. Hence FPGAs are being used as standalone computing/logic units.

2.2.2 FPGA-Based Accelerators

Custom processors can be augmented to a base processor for accelerating computations. Xtensa processor from Cadence allows instruction set configurability [15]. Processors with custom instruction set can be developed using the tools. Custom accelerators are usually implemented on FPGAs as they are easily adaptable. Taking design decisions manually from a large number of choices based on complex trade-offs including reuse, area and gain is cumbersome and time consuming. Computer-aided design (CAD) tools have been developed that select optimal combination of accelerators by throughly searching the entire design space [71]. These tools reduce design time and make it easier for the designer to develop custom accelerators for different kind of applications hence reusing the same setup and reducing the cost. Recently many soft-core processors like Leon are available in public domain [33]. These are widely being used as multiprocessor systems on FPGAs. As the source code for these processors is available, the designer can modify the existing instruction set according to application requirements to improve performance.

Typically, FPGA cards or boards are plug-in cards that are plugged into slots available on the motherboard of computers. The boards contain one or more FPGAs. The FPGAs are connected to each other by direct wired connections. User constraints provided to the tools ensure that the hardware components in different FPGAs are connected to the right pins. The communication interfaces between host and FPGA are predefined during board design. Some boards use separate FPGAs for managing data through the interfaces. The current FPGA boards use PCIe interfaces and can be plugged into the PCIe slots available on CPUs [2, 42, 93]. Hardware implementation is done in HDLs. The bit files to program the FPGAs are generated using respective

FPGA tools. Typically FPGAs are programmed using these bit-files via application program interfaces (APIs) which wraps around a device driver. As device capacity of the FPGA changes, the API has to be modified to account for the granularity of functions that are implemented on them.

Even though FPGA-based accelerators give promising results, their use is not wide spread as they pose significant challenges due to limitations of application development tools and design methodologies [52]. The construction of accelerators consumes a huge amount of time and is very laborious. Tarek El-Ghazawi et al. did a research study focusing on investigating key challenges [28]. The authors tried to identify limitations of existing FPGA tools. In the report they discuss limitations that are based upon each of the four application development phases namely "Formulation," "Design," "Translation," and "Execution".

Critical design decisions are not made through algorithm design, architecture exploration and mapping but rather mostly on ad hoc basis. This causes costly iterations in the design flow. Many tools exist for designing accelerators from hardware description languages (HDLs) or HLLs, but need expertise in hardware for exploiting the FPGA resources properly. HLLs do not aid the designer to specify the concurrency required to fully exploit the features of FPGA-based systems. The lack of good co-design tools and languages is a concern and this leads to manual design partitioning. The manual design and partitioning is specific to the interfaces present in the accelerator card/board and hence it reduces portability and interoperability across platforms. OpenCL is one such language which attempts to solve this interoperability issue [82]. Many commercial vendors are supporting the use of this language [4, 67, 92]. Even though the EDA tools using this promises easy implementation, manual intervention is still required to get benefits for the specific accelerator [78].

Translation time is a major hurdle to FPGA synthesis. The place and route tools take a long time and sometimes are not very efficient and require designer interference to pack more logic. The execution challenges are that of run-time support for debugging. Most tools don't provide this feature. The tools for run-time reconfiguration are also not very efficient.

2.2.3 Hard Embedded Blocks in FPGAs

Typically, FPGA implementation occupies more silicon area and run at slower clock frequencies vis-a-vis ASICs. To implement large circuits using configurable logic in the FPGA may some times need more than one FPGA. Use of multiple FPGAs require more printed circuit board (PCB) space and more circuitry to manage the I/Os between the FPGAs. Synchronization is a matter of concern. To alleviate this problem, many of the commonly used sub-circuits in a design are available as HEBs in FPGAs. As these HEBs are custom designed and not implemented as CLBs in the FPGA, they occupy less silicon area and have higher operating frequency. Figure 2.4 shows a simplistic diagram of a modern FPGA consisting of DSP units as HEBs. As shown in the figure, additional HEBs are provided along with the CLBs and routing

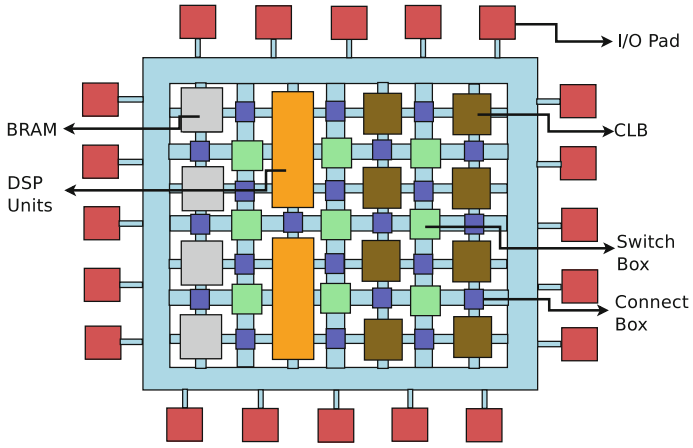

Fig. 2.4 Modern FPGA architecture

resources in the FPGA. Many a times, special routing resources are also provided for the HEBs. Modern FPGA contains memory controllers, PCIe controllers, FIFO controllers, serial transceivers, clock management, AES encryption units, ethernet MAC blocks, etc. as HEBs [92].

The HEBs in FPGAs can be broadly classified as fine-grained HEBs and the coarse-grained HEBs. The hardware blocks which are smaller in size and are used frequently are usually implemented as HEBs. A hardware design using a number of such fine-grained HEBs will be beneficial in terms of area, power and delay. These fine grained HEBs are typically distributed all over the area of the FPGA, and thus they can be closer to the blocks using them and hence reduce the interconnect delays. Some amount of configurability is provided to these blocks, so that a collection of them can be used to build larger hardware blocks. The coarser grained HEBs are hardware circuits which occupy significant amount of silicon area if they were implemented using the CLBs. Since these blocks are large, the configurability is less. As these blocks are pre-routed and do not use the interconnect blocks in the FPGA, they take less area and can run at high clock frequencies. Table 2.1 shows implementations of two hardware kernels on a Xilinx Virtex-6 FPGA. These were generated using Xilinx core-generator [90]. It can be observed that the equivalent silicon area used is significantly less compared to CLB only implementations. For example, the 64-bit signed multiplier unit uses 4265 slices, whereas the same implementation requires 16 DSP units whose equivalent area in terms of slices is 60, i.e., nearly two orders less.

Some of the fine-grained HEBs available in modern FPGAs are the block RAMs and carry-save arithmetic blocks. Almost all the applications require memory to store the intermediate results. These intermediate results are either stored in registers or in memory blocks. The advantage of storing data in the registers is that they can be written and read by many compute blocks, but they occupy more silicon area.

Table 2.1 Resources occupied by cores using only slices and using DSP units [92]

Core	Slices	DSP	Eq. slices
64-bit signed multiplier	4256	0	4256
	0	16	80
48-bit cordic unit	9138	0	9138
	0	12	60

Bit-level operations can be easily done on registers. A disadvantage of using registers is that in FPGA implementations consisting of a lot of wide bit-width registers, a lot of routing resources will be needed and thus may cause the operating frequency to reduce. The memory blocks are preferred when the data from the block is needed by very few compute units. Multi-ported memories are typically used in such scenarios, but they occupy more area and hardware support to keep a check on the "read after write" (RAW) and "write after read" (WAR) hazards. Modern FPGAs have memory blocks as HEBs. Embedded block RAMs were proposed and studied by Wilton et al. [66, 86–88]. The authors propose various ways to build memories as HEBs. Memories of various sizes have to be built based on the application. Choosing the right size of memory block is an important issue in FPGAs. Larger blocks will cause the memory to be unused, whereas the smaller blocks will cause routing congestion. Due to unavailability of multiple ports, larger blocks may sometimes cause the data-starvation in the compute units. The authors propose design of smaller memory blocks so that they can be combined to form larger blocks of memory. They also show that hardware implementations can benefit from logic circuits implemented using these memories as LUTs. Xilinx FPGAs provide 18 and 32 kb embedded block RAMs running at 550 MHz [89]. Altera also provides embedded memories named as MRAMs [5]. They are configurable and can be used to build larger memory blocks.

Various enhancements to the FPGA architecture have been done to improve the hardware implementation of arithmetic operations. The carry chains are provided along with the CLBs as HEBs to build efficient adders. The adder implementations typically require 2 LUTs to implement logic for carry and the sum. The use of carry chain reduces this as well as the interconnect delays. Parandeh-Afshar et al. [69] propose a way to improve the performance of carry-save arithmetic in FPGAs using generalized parallel counters. They show 1.8× improvement in energy consumption. Many of the FPGAs provide digital signal processing (DSP) units as HEBs [3, 91]. These DSPs units are configurable and can be used for performing addition, multiplication as well as multiply accumulator operations. The newer DSP units also support other bit-level operations. Beauchamp et al. [9] have proposed floating point units as HEBs. The authors implement both single-precision and double-precision floating point units as HEBs in FPGAs and show 55 % area benefits and 40.7 % increase in clock frequency. Flexible embedded floating point unit, which can be used both as single-precision and double-precision floating point units, has been proposed by Chong et al. [21].

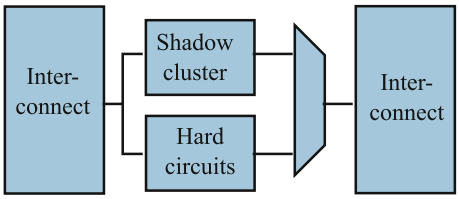

Fig. 2.5 Shadow clusters for performance benefits [44]

Configurable hard blocks have advantages as they can be used in wider applications without wasting the silicon area. The fabrics which have HEBs are expensive since the user has to pay for the cores available for HEBs even if they do not use them. This also effectively reduces the reconfigurable area for the same silicon area, if the HEBs are not used. Jamieson et al. [44] propose shadow circuits to improve the performance of designs without losing the flexibility of the FPGAs. The basic idea is shown in Fig. 2.5. Here the shadow clusters and the hard circuits are separated using the mux. The whole block is incorporated as HEB in FPGA fabric. The key idea is to use the silicon area efficiently when the hard circuits can not be directly used. The shadow circuits enable partial use of HEBs and thus increases the overall fabric area usage. They also implement CAD tools to support this type of mapping.

A proper methodology is needed to evaluate the HEBs to be incorporated in the FPGA fabric, in order to use the silicon area efficiently. Tool-flows to evaluate HEBs are discussed in more detail in Chap. 3.

One of the key areas which could benefit by using FPGA-based acceleration is bioinformatics. Applications in bioinformatics domain are compute intensive. Data-level parallelism can be exploited by hardware accelerators to speedup software implementations. In the next section we review some important concepts in the bioinformatics domain and introduce the need and scope of accelerators for two key algorithms.

2.3 Bioinformatics

Bioinformatics is an interdisciplinary subject consisting partly of molecular biology and partly of computers. It deals with use of computers to analyze and organize biological data. This field has advanced so rapidly that nowadays chemists, mathematicians, statisticians, and computer engineers work as a team to solve the problems [24]. DNA sequencing costs have been reduced by increasing throughput by the use of massively parallel DNA sequencing systems. These systems are capable of determining the sequence of huge numbers of different DNA strands at one time. These "next generation sequencing" (NGS) systems allow millions of reads to be gathered in a single experiment [40]. These sequencing techniques have led to exponential increase in data. NGS aspires to find solutions in genetic analysis. Understanding and analyzing this large volume of data is of immense importance to biologists. This

large volume creates challenges in acquisition, management and analysis of data. The major research areas in bioinformatics are sequence alignment, gene structure prediction, construction of phylogenetic trees, protein folding and drug design [23]. Many software tools have been developed to aid biologists in their respective fields [6, 31, 49, 59, 74]. Computer scientists and biologists are working together to overcome the challenges of rapid interpretation of the data. We describe two important bioinformatics applications that we address in this book, namely protein docking and genome assembly.

2.3.1 Protein Docking

Study of inter-molecular interactions has various applications in biology, especially in the process of drug design. An important and time-consuming part of drug design process is the "virtual screening", which is a computational technique used to find the drug molecules which binds well with a particular protein molecule from a large library of probable drug molecules [47, 60]. "Protein docking" is a software application which is used to do this screening process. The docking application scores the drug molecules based on the binding parameters. The docking program ranks the protein–ligand complex based on scoring functions. Scoring functions predict the strength of the complex. The highly ranked drugs or drug conformations are then checked to see if they are suitable for human beings. Even though the computer-based virtual screening is faster than chemical processes, most of the docking tools perform computationally intensive calculations and take large amount of time to execute on CPUs. The medical field is gradually moving towards personalized medicines and hence there will be a need to do a virtual screening on a large scale. Jenwitheesuk et al. [45] have proposed that computational screening of small drug-like molecules against multiple proteins will increase the efficiency for finding drugs for drug-resistant diseases. Implementing such a design for drug design will take significant amount of time and accelerating docking program becomes very important for making such a system pragmatic.

An example of protein complex is shown in Fig. 2.6. The diagram shows a ligand which is bound to the protein. Here the protein–ligand complex is in a state where overall free energy of the system is minimum. Docking will help find the orientation of the ligand so as to give such minimized free energy state system.
There are three types of docking [36]:

1. Rigid molecular docking
2. Flexible docking
3. Combination of both rigid and flexible docking

Rigid docking is a type of modeling in which the internal geometry of the interacting elements are kept fixed and relative orientations are varied. Changes in the internal geometry might occur during the formation of complex and docking which takes this into consideration is called flexible docking. The combination of these two is also

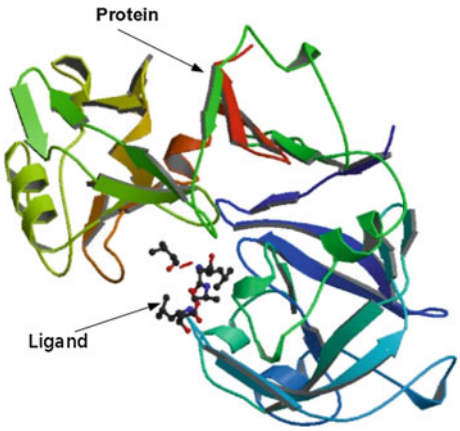

Fig. 2.6 Biological assembly image for docked complex [30]

used, where a small subset of possible conformational changes are taken into account to reduce the computation time. Some of the tools used for docking are FTDock [81], Autodock [62], PIPER [49] and Hex [26], ZDock [17], F2Dock [7].
Typical docking application involves the following steps:

1. Discretize the molecules
2. Rotating the molecules
3. Shape complementarity score calculations
4. Electrostatic complementarity score calculations

The discretization process involves placing the molecule in a 3D grid and noting the position of each atom in that molecule. This process is done for both the proteins in case of protein–protein docking, and both ligand and protein in case of protein–ligand docking. The total grid size is dependent on the size of molecule. The grid should cover the whole of the larger molecule. Resolution of the grid decides the accuracy of the experiment. Smaller grid resolution gives more accurate results but require more computations vis-a-vis large grid resolution. The discretized molecule is stored as a 3D matrix.

To study the different orientations of the molecules and to study their binding, systematic rotation of the protein or ligand molecules is carried out. Typically, larger molecule's position/orientation in the grid is kept constant and the smaller molecule is rotated around the axes. The various scores are calculated for each of the rotated smaller molecule. Here also, the resolution of rotation is chosen based on the accuracy required. For example for a resolution of 6°, there are 54,000 rotations possible where for 12° rotation 13,500 rotations are possible. Some of the approaches calculate other scores based on bonding.

Shape complementarity score and electrostatic complementarity score are calculated using correlation functions. For electrostatic complementarity the Coulombic forces on each of the atom by other atom is stored in the 3D matrix. These correlation functions are mostly convolution of the two matrices representing the

discretized matrices. The convolution function 'C' of two matrices 'A' and 'B' is given by (2.1).

$$C(i, j, k) = \sum_{l=1}^{L} \sum_{m=1}^{M} \sum_{n=1}^{N} A(l, m, n) * B(i - l, j - m, k - n) \tag{2.1}$$

The calculation of the convolution function on general purpose processors is expensive as it involves lots of multiplications. In many of the docking applications, FFTs are used to reduce the number of multiplications [7, 49].

2.3.1.1 Need for Speedup and Use of Accelerators

Even though a significant number of multiplications are reduced by use of FFT to $\log_2(N)$ vis-a-vis direct convolution which has N^3 multiplication, 3D-FFT still takes a very long time to execute on processors. The run-times for the various docking applications are shown in Table 2.2 which are reported by Dave et al. in [73]. The authors compare the run-time of Hex docking application [26] which is their own implementation, with the run-times of two other docking applications—ZDock [17] and PIPER [49]. The docking was carried out for Kallirein A/BPT1 complex. The protein molecules grid size was 128 and the ligand molecules grid size was 32. The processor used was Intel Xeon processor and the GPU used was Nvidia GTX-285. The ZDock application makes use of convolution function directly. ZDock does some optimizations for carrying out convolution function on sparse matrices. Piper [49] uses 3D-FFT for accelerating convolution function, whereas Hex docking application uses 3D-FFT in spherical coordinates to accelerate computations. The run-times of using 1D-FFT is also shown. It can be seen that there is significant reduction in execution time when GPUs are used. For example Hex docking computing 3D-FFTs gives speed-ups of about 2.7×. The GPUs carry out the FFT computation in parallel on the multiple floating point units and hence exploit the parallelism available in the application. Bharat et al. have attempted to accelerate PIPER docking application using FPGAs [10]. They report that for smaller sized molecules (grid size <16) FPGAs perform better than GPUs. They also report that GPUs perform better for

Table 2.2 Run-times of various docking programs [73]

Docking	Docking	Run-time (s)	
Software	Type	CPU	GPU
ZDock	3D	7172	–
Piper	3D	468,625	26,372
Hex	1D	676	15
	3D	224	84

docking of larger molecules. They report that FPGA implementation based on direct correlation function is better in FPGAs and FFT-based implementations are better suited for GPUs.

2.3.2 Genome Assembly

The basic building block of all organisms is the *cell*. All the cells (irrespective of the size of organism) have a nucleus which carries a genetic material known as *deoxyribonucleic acid* (DNA). DNA holds the hereditary information and is responsible for the controlling and functioning of the organism. DNA is made up of four bases: adenine (A), guanine (G), cytosine (C), and thymine (T). Adenosine (A) pairs with thymine (T), and cytosine (C) pairs with guanine (G) forming *base pairs* (bp). This pairing is due to weak hydrogen bonding and is the basis for DNA replication. A segment of a DNA molecule can be written using the first letter of the bases it contains (eg., ...TACGTAG...).

The complete set of all genes along with noncoding deoxyribonucleic acid (DNA) in an organism is called a *genome*. The study of genomes of various organisms is known as genomics. It has a lot of applications in medicine, biotechnology, anthropology, forensics and synthetic biology. Also, comparative study of different genomes is helpful for evolutionary studies. Increasingly, genomics is also being used to study the contribution of genes in many diseases and is aiding in the development of personalized drugs. Hence, genome construction is very important, which helps considerably in the study of various biological processes in an organism.

DNA sequencing technology helps in generating the data needed for construction of genomes. Recently, next-generation sequencing (NGS) platforms are being used for DNA sequencing. These platforms generate short fragments called "*reads*" of length ranging from thirty-five to few hundreds of base pairs. These reads are part of a large genome containing millions of base pairs (the size of the human genome $= 3 \times 10^9$ bp). The NGS platforms generate large amounts of data at very low cost and at a greater speed when compared to older platforms [65].

The large amount of data has posed many challenges for computer scientists who develop softwares to analyze these data. New algorithms and data structures have been proposed to speed up the analysis [8, 58]. Databases have been created and statistical analysis programs have been developed for retrieving specific information from this data. *Sequence assembly* is a computational biology problem where the reads generated from the NGS machine are used to build the whole genome.

An example of assembly is shown in Fig. 2.7. A biological sample is preprocessed and given to a sequencing machine, which generates a set of short reads. These short reads are assembled to construct the genome. The 'T' in the read CTGTGTGT*T*, is an error as the exact match to the genome at that position was supposed to be 'C'. The error could be identified as the frequency of occurrence of 'C' at that position is more than that of 'T'. The error can occur during the sequencing process by the sequencing machine. Error can also occur while assembling the genome from the

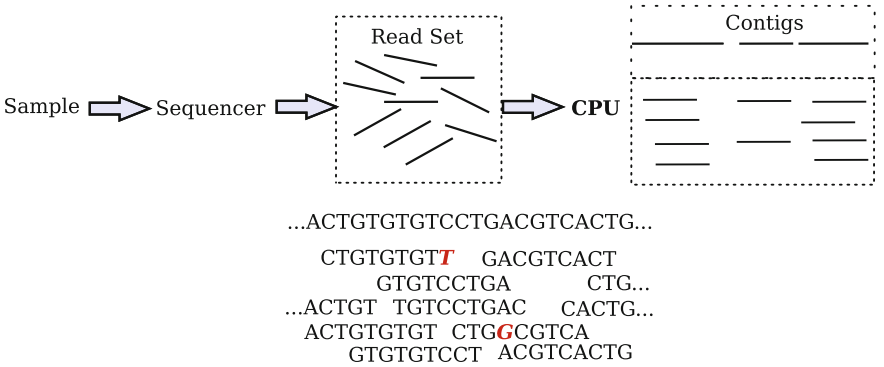

Fig. 2.7 NGS assembly: the DNA sample is given to sequencer, which generates read-set. The read-set is processed in CPUS to generate contigs

short reads, where the read is falsely mapped to a particular location of the genome. Due to these errors, genome assembly problem is more difficult to solve than the well studied shortest super-string problem [65].

2.3.2.1 Genome Assembly Softwares

Many types of software have been developed to do assembly [61]. Algorithms are modified in order to alleviate some of the complexities involved in the assembly and to execute efficiently on processors. Assembly software solutions can be divided into two categories; mapping based comparative assembly and de novo assembly. In the former method, assembly is done by mapping the reads to an already pre-existing reference genome. Even though the genomes of a particular organism contain lots of similarities, there are certain dissimilar regions which makes each organism unique. These dissimilar regions are of interest to biologists as they show particular behavior unique to that individual organism. Mapping the reads to a pre-existing reference genome might cause this uniqueness to be destroyed and hence the software assemblers allow certain amount of mismatches and gaps. Some of the mapping based assembly programs are SOAP [55], MAQ [54], Bowtie [53], and RMAP [79].

The later method is called de novo assembly where the information is extracted from the reads and their overlaps. De novo assembly is a type of assembly process where genome is constructed without using a reference genome. It is the only way to construct a genome if the reference genome does not exist. As the shorter reads have less overlap information, the reads are generated with much more coverage in order to construct the genome. The overlap information from the reads is used to construct the contiguous consensus sequences known as *contigs*. The genome is constructed using these contigs using a process known as scaffolding. Some of the de novo assembly software programs are Velvet [94], Edena [39], PerM [18], BFAST [41] and Minia [20]. De novo assembly takes more computational time than

Table 2.3 Run-time of various de novo genome assembly softwares [12]

Software application	Number of cores	RAM	Run-time
Hyda [63]	48	512 GB	14 h
Abyss [77]	12	4 GB each	13 h
Allpaths-LG [14]	48	512 GB	151–215 h
Velvet [94]	32	300–500 GB	3.5 weeks
Celera [64]	32	256 GB	9.5 days
SOAP de novo [57]	24–32	110–150 GB	48–72 h
Newbler [76]	12	130 GB	18 h
Ray [11]	256	768 GB	36–72 h
Monument [19]	16	140 GB	1 day

mapping based assembly. Since the mapping based assembly includes a pre-existing reference genome during mapping, the assembly process is biased and hence in certain situations bioinformaticians prefer to use de novo based assembly.

Clearly, like most of the bioinformatics applications, NGS assembly is also data dominated. Even though compute infrastructure ranging from server racks to cloud farms exist for solving these problems, the time taken is enormous. For example assembly of human genome using PASHA software took around 21 h on a 8-core workstation with 72 GB memory [56]. The run-times of various assembler programs as reported by Bradnam et al. [12] are shown in Table 2.3. The table shows the number of cores used, maximum RAM usage in most of the cases (authors report RAM available in the system in some cases) and the run-time. The run-times were reported for assembling snake, fish and bird genomes. They attempt to construct the genome for the first time and since they do not have any reference genome they use de novo genome assembly. It can be seen that the run-time taken by different softwares vary from 10 h to 3.5 weeks. Each of the assembler softwares have their own pipeline/flow to construct the genome, starting from processing of the biological sample. The selection of assembler application is also based on the sequencing machine used for generating the reads. For example, 454-sequencing machine provide a tool called Newbler which is a proprietary application that works best for the data it produces [76]. As most of the applications take significant amount of time to execute, there is a need to accelerate them.

2.3.2.2 FPGA-Based Acceleration

Fernandez et al. [79] have reported FPGA acceleration of NGS mapping [29]. They show speedups up to 4× over RMAP software. The reads are stored in registers in the FPGA and the reference sequence is streamed through shift-registers for comparison. Multiple reads are compared with the reference genome sequence and hence parallelism is exploited to get speed-ups over software application. Knodel et al. [48]

have also accelerated NGS short read mapping. They use a similar approach used by Fernandez et al. [29] and show that the performance is comparable to many software applications like Bowtie [53]. Tang et al. [84] report 42× speedup over software PerM [18] using FPGAs. PerM uses comparative assembly. They construct a pipeline of Processing Elements (PEs) in FPGAs. The reference sequence is compared with the reads in the PEs. The number of mismatches is counted and if it is less than a threshold, the position of the read is reported. Pairwise sequence comparison is executed in parallel and thus speedups are obtained software implementations. Olson et al. [68] has also shown acceleration of short read mapping on FPGA. The authors show 250× improvement over BFAST software [41] and 31× when compared to Bowtie [53]. All short sequences of fixed length (eg., 22) known as "seeds" in the reference sequence are indexed in a table. Each of the "seeds" present in the "reads" are searched in the index table. If the seed is found in table, Smith–Waterman algorithm is used to compare the read with the indexed portion of the reference sequence [80]. The hashing and the Smith–Waterman algorithm is implemented in FPGA. Speed-ups over software is obtained because of the hash function, which reduces the comparison and due to the fast implementation of dynamic programming (Smith–Waterman algorithm) on FPGAs. The Convey Computer firm has developed the Convey GraphConstructor (CGC), which use FPGAs to accelerate de novo assembly [25]. They show speedups of 2.2× to 8.4×. There is very little work reported in literature on FPGA acceleration of de novo genome assembly. It can be seen that there is a need to accelerate many assembly applications.

2.4 Summary

In this chapter we have discussed the various accelerator architectures. We have discussed FPGA-based accelerators in detail. We have described the basic architecture of FPGA and how they have evolved to include HEBs. We report some of the research works related to HEBs. We have introduced some basic concepts of bioinformatics and describe protein docking and genome assembly which are two important bioinformatics applications. We have discussed works related to accelerating these applications specifically using FPGAs. With this background, the DSE is explained in the next few chapters.

References

1. Ahmed, E., Rose, J.: The effect of LUT and cluster size on deep-submicron FPGA performance and density. IEEE Trans. Very Large Scale Integr. VLSI Syst. **12**(3), 288–298 (2004)
2. Alpha-Data: Alpha-Data FPGA Boards. http://www.alpha-data.com/ (2015)
3. ALTERA: Altera dsps. http://www.altera.com/technology/dsp/dsp-index.jsp (2015)
4. Altera: Altera FPGAs. http://www.altera.com (2015)

5. ALTERA: Altera mrams. http://www.altera.com/technology/memory/embedded/mem-embedded.html (2015)
6. Altschul, S.F., Madden, T.L., Schffer, A.A., Zhang, J., Zhang, Z., Miller, W., Lipman, D.J.: Gapped blast and psiblast: a new generation of protein database search programs. Nucleic Acids Res. **25**(17), 3389–3402 (1997)
7. Bajaj, C., Chowdhury, R., Siddavanahalli, V.: F2Dock: fast Fourier protein-protein docking. IEEE/ACM Trans. Comput. Biol. Bioinform. **8**(1), 45–58 (2011)
8. Baker, M.: Next generation sequencing: adjusting to data overload. Nat. Methods 495–499 (2010)
9. Beauchamp, M.J., Hauck, S., Underwood, K.D., Hemmert, K.S.: Embedded floating-point units in FPGAs. In: ACM/SIGDA International Symposium on FPGAs (2006)
10. Bharat, S., Herbordt, M.C.: GPU acceleration of a production molecular docking code. In: GPGPU (2009)
11. Boisvert, S., Laviolette, F., Corbeil, J.: Simultaneous assembly of reads from a mix of high-throughput sequencing technologies. J. Comput. Biol. **17**(11), 1519–1533 (2010)
12. Bradnam, K.R., Fass, J.N., Alexandrov, A., Baranay, P., et al.: Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species. GigaSci. **2**(1) (2013)
13. Brown, S., Rose, J.: Architecture of FPGAs and CPLDs: a tutorial. IEEE Des. Test Comput. **13**, 42–57 (1996)
14. Butler, J., MacCallum, I., Kleber, M., Shlyakhter, I.A., Belmonte, M.K., Lander, E.S., Nusbaum, C., Jaffe, D.B.: ALLPATHS: de novo assembly of whole-genome shotgun microreads. Genome Res. **18**(5), 810–820 (2008)
15. CADENCE: Tensilica customizable processors. http://ip.cadence.com/ipportfolio/tensilica-ip (2015)
16. Che, S., Li, J., Sheaffer, J., Skadron, K., Lach, J.: Accelerating compute-intensive applications with gpus and fpgas. In: Symposium on Application Specific Processors, 2008. SASP 2008 (2008)
17. Chen, R., Li, L., Weng, Z.: ZDOCK: an initial-stage protein-docking algorithm. Proteins **52**(1), 80–87 (2003)
18. Chen, Y., Souaiaia, T., Chen, T.: PerM: efficient mapping of short sequencing reads with periodic full sensitive spaced seeds. Bioinformatics **25**(19), 2514–2521 (2009)
19. Chikhi, R.: Monument assembler. http://www.irisa.fr/symbiose/people/rchikhi/monument.html (2015)
20. Chikhi, R., Rizk, G.: Space-Efficient and Exact de Bruijn Graph Representation Based on a Bloom Filter. In: Raphael, B., Tang, J. (eds.) Algorithms in Bioinformatics. Lecture Notes in Computer Science, vol. 7534, pp. 236–248. Springer, Berlin Heidelberg (2012)
21. Chong, Y.J., Parameswaran, S.: Flexible multi-mode embedded floating-point unit for field programmable gate arrays. In: ACM/SIGDA International Symposium on FPGAs (2009)
22. Chung, E., Milder, P., Hoe, J., Mai, K.: Single-chip heterogeneous computing: does the future include custom logic, FPGAs, and GPGPUs? In: Annual IEEE/ACM International Symposium on Microarchitecture (MICRO) (2010)
23. Cohen, J.: Bioinformatics an introduction for computer scientists. ACM Comput. Surv. **36**, 122–158 (2004)
24. Cohen, J.: Computer science and bioinformatics. Commun. ACM **48**, 72–78 (2005)
25. Convey Computer: Convey GraphConstructor. http://www.conveycomputer.com (2015)
26. David, Ritchie, W., Ritchie, D.: Hex 6.0 user manual protein docking using spherical polar fourier correlations
27. Devadoss, R., Paul, K., Balakrishnan, M.: p-qca: a tiled programmable fabric architecture using molecular quantum-dot cellular automata. J. Emerg. Technol. Comput. Syst. **7**(3), 13:1–13:20 (2011)
28. El-Ghazawi, T., George, A.D., Gonzalez, I., Lam, H., Merchant, S., Saha, P., Smith, M., Stitt, G., Alam, N., El-Araby, E., Holland, B., Reardon, C.: Exploration of a Research Roadmap for Application Development and Execution on Field-Programmable Gate Array (FPGA)-Based Systems. Technical Report, Defense Technical Information Center (2008)

29. Fernandez, E., Najjar, W., Harris, E., Lonardi, S.: Exploration of short reads genome mapping in hardware. In: International Conference on FPL, pp. 360–363 (2010)
30. Fujinaga, M., Chernaia, M.M., Tarasova, N.I., Mosimann, S.C., James, M.N.: Crystal structure of human pepsin and its complex with pepstatin. Protein Sci. **4** (1995)
31. Gabb, H.A., Jackson, R.M., Sternberg, M.J.E.: Modelling protein docking using shape complementarity, electrostatics and biochemical information. J. Mol. Biol. **272** (1997)
32. Gac, N., Mancini, S., Desvignes, M., Houzet, D.: High speed 3D tomography on CPU, GPU, and FPGA. EURASIP J. Embed. Syst. **2008**, 5:1–5:12 (2008)
33. Gaisler: LEON processors. http://www.gaisler.com/index.php/products/processors/ (2015)
34. Gao, H., Yang, Y., Ma, X., Dong, G.: Analysis of the effect of LUT size on FPGA area and delay using theoretical derivations. In: Sixth International Symposium on Quality of Electronic Design, 2005. ISQED 2005, pp. 370–374 (2005)
35. Grozea, C., Bankovic, Z., Laskov, P.: Facing the multicore-challenge. chap. FPGA vs. Multicore CPUs vs. GPUs: Hands-on Experience with a Sorting Application, pp. 105–117. Springer-Verlag, Berlin, Heidelberg (2010)
36. Halperin, I., Ma, B., Wolfson, H., Nussinov, R.: Principles of docking: an overview of search algorithms and a guide to scoring functions. Proteins Struct., Funct., Bioinf. **47**(4), 409–443 (2002)
37. Hauck, S., DeHon, A.: Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation. Morgan Kaufmann, Systems on Silicon (2007)
38. Herbordt, M.C., VanCourt, T., Gu, Y., Sukhwani, B., Conti, A., Model, J., DiSabello, D.: Achieving high performance with FPGA-based computing. Computer **40**(3), 50–57 (2007)
39. Hernandez, D., François, P., Farinelli, L., Østerås, M., Schrenzel, J.: De novo bacterial genome sequencing: millions of very short reads assembled on a desktop computer. Genome Res. **18** (2008)
40. Hert, D.G., Fredlake, C.P., Barron, A.E.: Advantages and limitations of next-generation sequencing technologies: a comparison of electrophoresis and non-electrophoresis methods. Electrophoresis **29**(23), 4618–4626 (2008)
41. Homer, N., Merriman, B., Nelson, S.F.: BFAST: an alignment tool for large scale genome resequencing. PLoS ONE **4** (2009)
42. Inc., D.: Digilent FPGA Boards. https://www.digilentinc.com/ (2015)
43. ITRS: The International Technology Roadmap for Semiconductors. http://www.itrs.net (2015)
44. Jamieson, P., Rose, J.: Enhancing the area efficiency of fpgas with hard circuits using shadow clusters. IEEE Trans. VLSI Syst. **18**(12), 1696–1709 (2010)
45. Jenwitheesuk, E., Horst, J.A., Rivas, K.L., Voorhis, W.C.V., Samudrala, R.: Novel paradigms for drug discovery: computational multitarget screening. Trends Pharmacol. Sci. **29**(2), 62–71 (2008)
46. Kapre, N., DeHon, A.: Performance comparison of single-precision SPICE model-evaluation on FPGA, GPU, cell, and multi-core processors. In: International Conference on Field Programmable Logic and Applications, 2009. FPL 2009, pp. 65–72 (2009)
47. Kitchen, D.B., Decornez, H., Furr, J.R., Bajorath, J.: Docking and scoring in virtual screening for drug discovery: methods and applications. Nat. Rev. Drug Discov. **3**(11), 935–949 (2004)
48. Knodel, O., Preusser, T., Spallek, R.: Next-generation massively parallel short-read mapping on FPGAs. In: IEEE International Conference on ASAP, pp. 195–201 (2011)
49. Kozakov, D., Brenke, R., Comeau, S.R., Vajda, S.: PIPER: an FFT-based protein docking program with pairwise potentials. Proteins (2006)
50. Kuon, I., Rose, J.: Measuring the gap between FPGAs and ASICs. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **26**(2), 203–215 (2007)
51. Kuon, I., Rose, J.: Area and delay trade-offs in the circuit and architecture design of FPGAs. In: Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays. FPGA '08, pp. 149–158. ACM, New York, NY, USA (2008)
52. Kuon, I., Tessier, R., Rose, J.: FPGA architecture: survey and challenges. Found. Trends Electron. Des. Autom. **2**, 135–253 (2008)

53. Langmead, B., Trapnell, C., Pop, M., Salzberg, S.: Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biol. **10** (2009)
54. Li, H., Ruan, J., Durbin, R.: Mapping short DNA sequencing reads and calling variants using mapping quality scores. Genome Res. **18**(11), 1851–1858 (2008)
55. Li, R., Li, Y., Kristiansen, K., Wang, J.: SOAP: short oligonucleotide alignment program. Bioinformatics **24**(5), 713–714 (2008)
56. Liu, Y., Schmidt, B., Maskell, D.: Parallelized short read assembly of large genomes using De Bruijn graphs. BMC Bioinf. **12**(1), 354–363 (2011)
57. Luo, R., Liu, B., Xie, Y., Li, Z., Huang, W., Yuan, J., He, G., Chen, Y., Pan, Q., Liu, Y., Tang, J., Wu, G., Zhang, H., Shi, Y., Liu, Y., Yu, C., Wang, B., Lu, Y., Han, C., Cheung, D.W., Yiu, S.M., Peng, S., Xiaoqian, Z., Liu, G., Liao, X., Li, Y., Yang, H., Wang, J., Lam, T.W., Wang, J.: SOAPdenovo2: an empirically improved memory-efficient short-read de novo assembler. Gigascience **1**(1), 18 (2012)
58. Mardis, E.R.: Next generation DNA sequencing methods. Ann. Rev. Genomics Hum. Genet. **9**, 387–402 (2008)
59. McCollum, J.M., Peterson, G.D., Cox, C.D., Simpson, M.L.: Accelerating gene regulatory network modeling using grid-based simulation. Simulation **80**(4–5), 231–241 (2004)
60. McInnes, C.: Virtual screening strategies in drug discovery. Curr. Opin. Chem. Biol. **11**(5), 494–502 (2007)
61. Miller, J.R., Koren, S., Sutton, G.: Assembly algorithms for next-generation sequencing data. Genomics **95**(6), 315–327 (2010)
62. Morris, G.M., Huey, R., Lindstrom, W., Sanner, M.F., Belew, R.K., Goodsell, D.S., Olson, A.J.: J. Comput. Chem
63. Movahedi, N., Forouzmand, E., Chitsaz, H.: De novo co-assembly of bacterial genomes from multiple single cells. In: 2012 IEEE International Conference on Bioinformatics and Biomedicine (BIBM), pp. 1–5 (2012)
64. Myers, E.W., Sutton, G.G., Delcher, A.L., Dew, I.M., Fasulo, D.P., et al.: A whole-genome assembly of Drosophila. Science **287**(5461), 2196–2204 (2000)
65. Nagarajan, N., Pop, M.: Sequence assembly demystified. In: Nature Reviews Genetics, pp. 157–167 (2013)
66. Ngai, T., Rose, J., Wilton, S.J.E.: An sram-programmable field-configurable memory. In: Proceedings of the IEEE Custom Integrated Circuits Conference, 1995, pp. 499–502 (1995)
67. NVIDIA: Nvidia GPGPU. http://www.nvidia.com (2015)
68. Olson, C., Kim, M., Clauson, C., Kogon, B., Ebeling, C., Hauck, S., Ruzzo, W.: Hardware acceleration of short read mapping. In: IEEE Symposium on FCCM, pp. 161–168 (2012)
69. Parandeh-Afshar, H., Verma, A., Brisk, P., Ienne, P.: Improving fpga performance for carry-save arithmetic. IEEE Trans. Very Large Scale Integ. VLSI Syst. **18**(4), 578–590 (2010)
70. Patel, S., Hwu, W.W.: Guest editors' introduction: accelerator architectures. IEEE Micro **28**(4), 4–12 (2008)
71. Pothineni, N., Kumar, A., Paul, K.: Exhaustive enumeration of legal custom instructions for extensible processors. In: VLSID '08: Proceedings of the 21st International Conference on VLSI Design, pp. 261–266. IEEE Computer Society, Washington, DC, USA (2008)
72. QuickLogic: QuickLogic FPGAs. http://www.quicklogic.com/ (2015)
73. Ritchie, D.W., Venkatraman, V.: Ultra-fast FFT protein docking on graphics processors. Bioinformatics **26**(19), 2398–2405 (2010)
74. Rizk, G., Lavenier, D.: Gassst: global alignment short sequence search tool. Bioinformatics **26**(20), 2534–2540 (2010)
75. Rose, J., Gamal, A.E., Member, S., Sangiovanni-vincentelli, A.: Architecture of field-programmable gate arrays: the effect of logic block functionality on area efficiency. Proc. IEEE **25**, 1217–1225 (1990)
76. Sequencing, R.: 454 Sequencing. http://www.454.com/products/analysis-software/ (2015)
77. Simpson, J., Wong, K., Jackman, S., Schein, J., Jones, S., Birol, I.: ABySS: a parallel assembler for short read sequence data. Genome Res. **19**, 1117 (2009)

78. Singh, D.P., Czajkowski, T.S., Ling, A.C.: Harnessing the power of fpgas using altera's opencl compiler. In: Hutchings, B.L., Betz, V. (eds.) FPGA, pp. 5–6. ACM (2013)
79. Smith, A.D., Xuan, Z., Zhang, M.Q.: Using quality scores and longer reads improves accuracy of solexa read mapping. BMC Bioinf. **9**, 128 (2008)
80. Smith, T., Waterman, M.S.: Identification of common molecular subsequences. J. Mol. Biol. **147**(3), 195–197 (1981)
81. Sternberg, M.J.E., Aloy, P., Gabb, H.A., Jackson, R.M., Moont, G., Querol, E., Aviles, F.X.: A computational system for modeling flexible protein-protein and protein-DNA docking. In: Proceedings of the 6th International Conference on Intelligent Systems for Molecular Biology, pp. 183–192 (1998)
82. Stone, J., Gohara, D., Shi, G.: Opencl: a parallel programming standard for heterogeneous computing systems. Comput. Sci. Eng. **12**(3), 66–73 (2010)
83. Tabula: Tabula FPGAs. http://www.tabula.com/ (2015)
84. Tang, W., Wang, W., Duan, B., Zhang, C., Tan, G., Zhang, P., Sun, N.: Accelerating millions of short reads mapping on a heterogeneous architecture with fpga accelerator. Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 0, pp. 184–187 (2012)
85. Venkatesh, G., Sampson, J., Goulding, N., Garcia, S., Bryksin, V., Lugo-Martinez, J., Swanson, S., Taylor, M.B.: Conservation cores: reducing the energy of mature computations. SIGARCH Comput. Archit. News **38**(1), 205–218 (2010)
86. Wilton, S.J.E.: Embedded memory in fpgas: recent research results. In: 1999 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, pp. 292–296 (1999)
87. Wilton, S.J.E., Rose, J., Vranesic, Z.: The memory/logic interface in fpgas with large embedded memory arrays. IEEE Trans. Very Large Scale Integr. VLSI Syst. **7**(1), 80–91 (1999)
88. Wilton, S.J.E., Rose, J., Vranesic, Z.G.: Architecture of centralized field-configurable memory. In: Proceedings of the 1995 ACM Third International Symposium on Field-programmable Gate Arrays. FPGA '95, pp. 97–103. ACM, New York, NY, USA (1995)
89. XILINX: Xilinx brams (2015)
90. XILINX: Xilinx core generator. http://www.xilinx.com/tools/coregen.htm (2015)
91. XILINX: Xilinx dsps. http://www.xilinx.com/products/technology/dsp/ (2015)
92. Xilinx: Xilinx FPGAs, ISE. http://www.xilinx.com (2015)
93. XtremeData: XtremeData FPGA Boards. http://www.xtremedata.com/ (2015)
94. Zerbino, D.R., Birney, E.: Velvet: algorithms for de novo short read assembly using De Bruijn graphs. Genome Res. **18**(5), 821–829 (2008)
95. Zhang, W., Jha, N.K., Shang, L.: A hybrid nano/cmos dynamically reconfigurable system—part i: Architecture. J. Emerg. Technol. Comput. Syst. **5**(4), 16:1–16:30 (2009)

Chapter 3
Methodology for Implementing Accelerators

Abstract Many software implementation of applications have been accelerated using FPGAs. Compute intensive kernels from the application are implemented in hardware and executed in parallel to achieve speedup over softwares. FPGA consists of CLBs (consisting of LUTs and flip-flops) and interconnects which are programmable. HEBs implement performance critical components efficiently vis-a-vis their implementation using configurable logic and thus improve the performance. Adding of HEBs to FPGA fabrics may not always give performance benefits, as they occupy significant amount of chip area and sometimes may not be usable due to limitations of the memory bandwidth. A proper methodology to design HEBs and estimate the expected performance gain would be a necessary component of any design methodology. It is expected that more HEBs will be embedded into FPGAs and such a methodology will aid in building efficient reconfigurable fabrics. In this chapter, we describe a methodology to design accelerators using FPGAs with custom-designed HEBs.

3.1 Design Space Exploration

Typically, hardware accelerators are used to reduce the execution time of software implementations. When the accelerators have fixed architecture like the GPUs and CELL, the application acceleration problem involves efficient mapping of the application to the architecture. In FPGA-based accelerators, one more dimension is added to the problem due to reconfigurable nature of FPGA fabric. This makes the hardware–software codesign problem hard to solve as there are lots of design choices to be evaluated.

Speedups are achieved by identifying the compute intensive kernels in the application and implementing them as hardware blocks in FPGAs. Sophisticated hardware implementations can be done using the HEBs available in modern FPGAs. Accelerator designs having hardware implementations using HEBs take less area and give better speedups vis-a-vis designs not using them. For example, many applications which involve large bit-width multiplications perform better when DSP units are used. DSP units are available as HEBs in most of the modern FPGAs. HEBs do

B.S.C. Varma et al., *Architecture Exploration of FPGA Based Accelerators for BioInformatics Applications*, Springer Series in Advanced Microelectronics 55, DOI 10.1007/978-981-10-0591-6_3

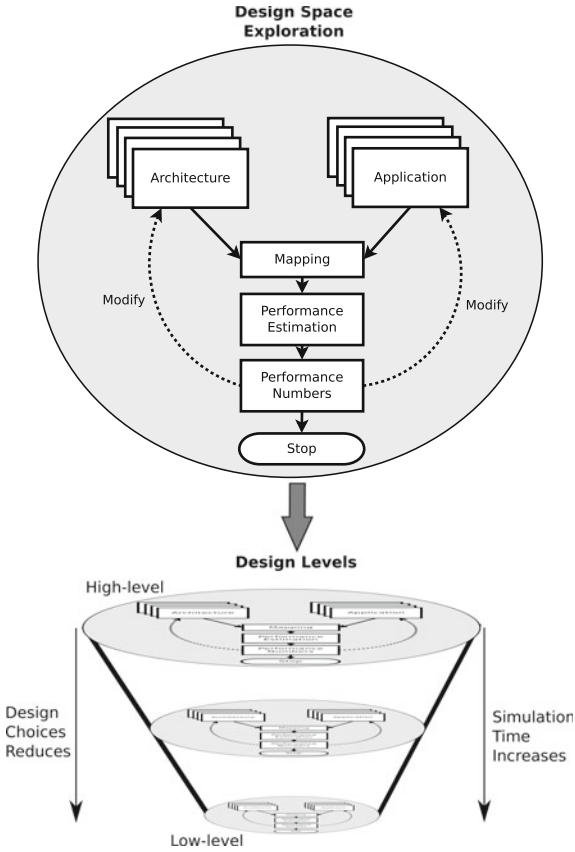

Fig. 3.1 Y-chart for design space exploration [5]

reduce the available configurable logic (if we consider the chip area as fixed), but still vastly outperform. For example, the area equivalent of one DSP48 unit available as HEB in Xilinx FPGA is around five slices, whereas an equivalent multiplier implementation on a FPGA (without DSP unit) will require much more than five slices. A number of different HEBs can be added to the FPGA in order to get performance benefits. Identifying the right HEBs based on the application is very important.

Methodologies have been developed and many tools are available to design system on chips (SoCs). These SoCs are typically designed for a specific set of applications. The Y-chart approach shown in Fig. 3.1 is commonly used for design of SOCs [5]. A similar approach can be used for designing FPGAs with custom HEBs. This approach has been evolved for achieving the design goals which could be optimization of power, energy, performance, or a combination of them using specific trade-off factor. Extensive design space exploration is carried out using the different parameters to come up with the best-suited architecture. The system design with incremental changes is repeated at different levels of abstraction. At higher levels of abstractions, many design choices can be evaluated with ease as the time taken for the evaluation

is small. As we go down the abstraction level, the simulation time dominates and thus limits the range of design parameters that can be explored.

In the Y-chart method, an architecture of a base system is built in the beginning. The architecture represents the complete system including the interfaces with the external devices. In a system containing an FPGA board, the communication interfaces between the host and the FPGA board like PCIe are well-defined. The size of the FPGA is fixed and so the number of CLBs, the interconnects, I/O pins, etc., available is known to the designer. An approximate hardware implementation is done on the FPGA which is incrementally improved in the later stages.

The application set is the input to the design space for which an optimal FPGA fabric has to be designed. Usually, the application set belongs to a particular domain having similar characteristics. The application characteristics are identified from the set. The application is profiled at different stages to get the estimates of the time taken by different functions to identify the kernel.

Mapping the application onto the architecture is the most difficult part of the design space exploration. This is a typical hardware–software codesign problem where the parts that have to be executed in software and the parts that have to be executed in hardware have to be identified. The memory access patterns have to be studied based on the memory hierarchy available in the whole system. The communication delay for the data transfer between the interfaces available between the processor and the hardware has to be considered.

Performance estimation is done by building models for each component in the system. During the initial design phases analytic models are useful and faster to use. Building analytical models for certain type of systems is very difficult and so simulation at different levels of abstraction can be used. For example, analytical models for the cache is very difficult and so cache simulators are used for estimating time taken for accesses accross memory hierarchy. Simulations also help to check the correctness of the system. The trade-offs in the quality of output can be studied by these simulations. Simulations at higher level of abstractions typically have some error in the performance numbers, as they generally use lumped delay models. The higher level simulations are fast and thus it is possible to explore many more different design choices. As we go down the abstraction level, the time taken for performance estimates increases, reducing the range of design choices that can be explored.

Performance numbers are obtained from the performance estimates. Based on these numbers, the architecture changes are carried out to meet the design goals. The application modifications are also done based on these numbers. Various decisions on the quality of the output are also taken during this phase. The bit-width optimizations are done both for reducing the communication delay and reducing the time taken for execution. The decision on type representation for different data elements (integer, fixed point, floating point, double floating, etc.) are also taken based on the performance numbers along with quality of output. Both application and hardware designs are changed to accommodate such modification in the data types. The bandwidth requirements, communication as well as memory, for the particular architecture and data are computed. The bandwidth study influences scalability of the system, as well as helps to identify the modules that are likely to be starved or bandwidth limited.

The need and size of FIFOs to connect the modules to reduce starvation are also obtained from these performance numbers. The performance numbers at different levels of abstraction provide insight to make the right trade-offs to obtain an optimal design.

3.2 Tools for Carrying Out Design Space Exploration

Various tools, each specialized in certain functionality are used to carry out design space exploration. We have used some open source tools as well as proprietary tools for evaluating HEBs in FPGA. The proprietary tools from companies require the designer to adapt their flow for designing and implementing hardware in FPGAs. We discuss the tools and the tool flows in the following subsections.

3.2.1 Profiling

The first step in mapping applications to hardware is profiling. Profiling the application enables us to analyze the bottlenecks in the program execution. It also aids in program optimization. Profiling tools help in identifying the program phases which in turn help in modifying the underlying algorithms. The small fraction of the program which takes most of the execution time can be identified. Typically, these are small portions of the code that are executed multiple number of times. Identification of such code segments generally drives hardware–software partitioning. These tools also enable the designer to know the communication overheads in different phases. Ideally, profiling done on the actual platform gives better insights for program optimization and hardware–software partitioning, but such exploration is typically infeasible as the actual hardware is to be built. Typically, profiling is done starting from very low level granularity and the profile data is analyzed for performance on yet to be designed systems.

Even though profiling tools run on CPUs, the profile data is very useful in designing hardware accelerators. Compute intensive kernels are identified using profiling tools. To achieve speedups, multiple copies of these compute intensive kernels are implemented in hardware as processing elements (PEs). Memory bandwidth available between the FPGA and the CPU limits the supply of data to the PEs which in turn limits the number of PEs that can be kept busy. Profiling tools are used to analyze the memory requirements for a single kernel and this is used to estimate the number of PEs that can be kept busy for a given bandwidth. For significantly high memory bandwidths, the number of PEs that can be implemented in hardware is limited by the FPGA hardware resource constraints. This can further be increased using multi-FPGA boards.

Some FPGA implementations consist of deep pipelines which reduces the requirement of very high bandwidth. In such cases too, the profiling tools will help in deciding the data required by one pipeline stage. This can be used to do retiming in the hardware design, which may reduce the FPGA resources.

We used gprof [1], VTune [4], PIN [6] for profiling the applications. GNU prof also known as gprof is a profiling tool that can be used to determine the parts of a program that take most of the execution time. This profiler works with gcc and g++. The program executable has to be made using the '-pg' option when compiler is being used. When this program is executed, a file with name 'gmon.out' is produced. The time taken by the executable thus generated is typically more than the normal executable as the profile data is collected. The command gprof is run to get the details of the profiled data. It also generates a call graph showing the dynamic hierarchy of the program parts. The profile data lists the various functions that are executed along with the time taken and the number of times the function was executed.

VTune and PIN are proprietary tools from Intel. VTune uses PIN tool in the background for getting the profile data. The tool has a good GUI and displays the call graph. The profile data is also displayed along with call graph. The time taken by function and number of times it is executed is shown. The assembly instructions and the number of times they are executed is also shown. Many options can be chosen for instrumenting the profile data. For some of the options the code is executed multiple number of times to show the exact execution trace. PIN is a dynamic binary instrumentation tool, which adds certain instructions in the code to enable profiling. Many optimization techniques are used to reduce the run time of the tool as well as reduce its memory overhead.

3.2.1.1 ASIC Synthesis

The HEBs are identified by analyzing the profile data. The HEBs have to be implemented in hardware and ASIC synthesis has to be done to get the exact area and timing information about the modules. We modeled the HEB using hardware description languages; VHDL and Verilog. This code was simulated and checked for correctness using Modelsim simulator [2]. The ASIC synthesis tools first convert the HDLs into their proprietary intermediate formats. The optimization is carried out in the intermediate level code and then the intermediate code is broken down into standard cells/gates provided in the form of library. The library contains different variations of the gates (as well as conditions) and based on the design goals the right library elements are chosen and the intermediate code is mapped to such hardware. Lots of optimizations are carried out after the mapping based on timing, fan-outs, and other parameters. The tool gives the area and the critical path of the circuit, which limits the clock period. We have used Synopsys Design Compiler [8] tool for doing the ASIC synthesis to obtain the clock period and the area of the HEBs.

3.3 Design of FPGAs with Hard Embedded Blocks

The HEBs have to be created in the FPGAs and the hardware description should capture the HEBs. The proprietary tools from Xilinx and Altera do not directly provide such tools to include HEBs in their flow. Versatile place and route (VPR) tool from University of Toronto have developed a methodology and associated tools for doing the evaluation of HEBs [7]. Virtual embedded block (VEB) is another methodology from Imperial College London that can be used for evaluating potential HEBs [3].

3.3.1 VPR Methodology

The VPR tools provide the complete tool flow for evaluating FPGAs with heterogeneous blocks. The design flow entry is using VHDL. The architecture of the FPGA is defined in an architecture description file. Here the size of LUTs, the interconnects, and available resources can also be defined. This description as well as the VHDL code is given using ODIN tool for synthesis. The tool does mapping of the intermediate code onto the resources available in architecture file. The HEBs supported were multipliers. If other HEBs have to be included, the tool has to be modified for optimization. The T-VPACK tool does the packing of the design into LUTs and does the clustering of the LUTs to optimize mapping on FPGA. The VPR tool then places and routes the interconnect between the packed LUTs. This will give the critical path and the resources occupied by the design. This can further be used in the flow to get the performance numbers for application acceleration.

3.3.2 VEB Methodology

The VEB flow is different from VPR. The VEB methodology uses already existing tools and evaluates inclusion of HEBs in existing commercial FPGAs. The Xilinx tool flow (ISE) is used starting from the design entry to the place and route steps. The HEBs are modeled as dummy virtual blocks in the whole design. The VHDL code has to be modified to use these dummy blocks. The dummy blocks area and critical path should be equivalent to the HEBs delay which is obtained from ASIC synthesis. If it is not possible then the area can be separately synthesized to get area estimates and timing can be separately synthesized with different dummy blocks. The dummy blocks are introduced into the design as black boxes which are either hard macros or relationally placed macros (RPMs) in the Xilinx flow. RPMs have flexibility in shaping the HEB. Hard macros fix the shape of the HEBs. The design with such hard macros is placed and routed to obtain the area and critical path of the whole design. These values can be used further in the flow for estimating performance of

the application. We have used VEB flow in our experiments as we wanted to evaluate application acceleration for current FPGAs with HEBs. In the presented work, the HEBs designed are not derived from existing HEBs. We created dummy blocks using the ASIC synthesis results. Also, actual devices with the proposed HEBs have not been fabricated and performance gains reported are on the basis of the area and delay obtained from synthesis.

3.4 Methodology for Designing FPGAs with Custom Hard Embedded Blocks

We follow a similar approach as the Y-chart approach for the system design, but our focus is on implementing custom HEBs in FPGAs. The system is evaluated for designing FPGA-based accelerators incorporaing HEBs. The various parameters that effect the design like the number of HEBs and the bandwidth requirement for using the HEBs efficiently is analyzed. The methodology to evaluate HEBs in FPGAs

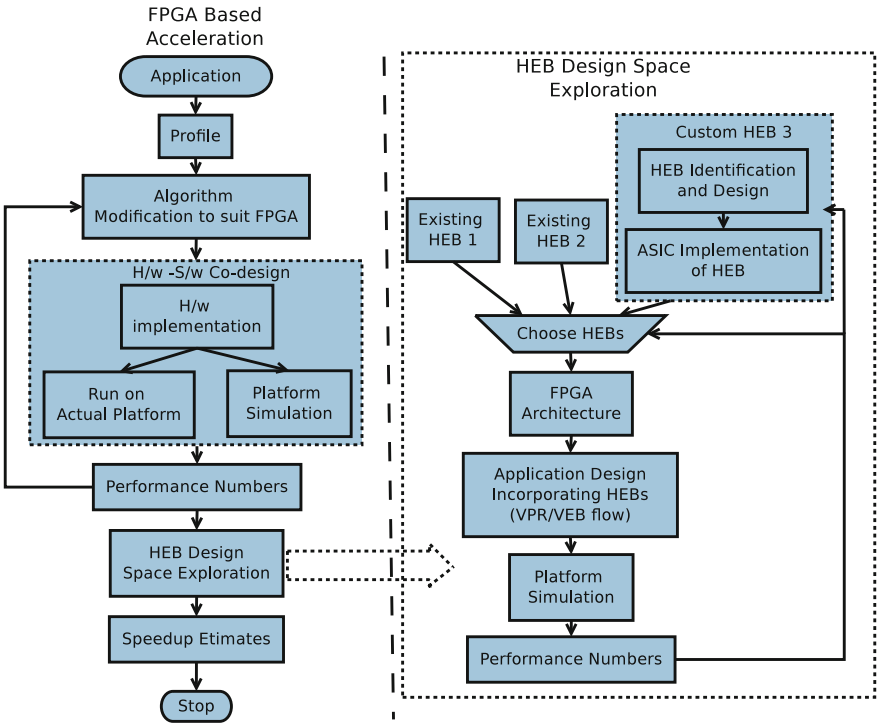

Fig. 3.2 Methodology to design hard embedded blocks in FPGA in the context of FPGA-based accelerators

for accelerating an application is shown in Fig. 3.2. The flow diagram on the left depicts the generic flow for accelerating applications using FPGAs. The flow diagram on the right shows the corresponding flow for design and evaluation of HEBs in FPGAs. Here the design of HEBs is done to maximize speedup over software under resource constraints. Profiling of an application is done to identify various kernels at different levels of granularity. Granularity can have a significant impact on the external memory bandwidth as some of the data accesses can be converted from external memory accesses to internal memory accesses. This would typically push the partition to larger granularity. This needs to be balanced against "reuse" potential of kernel among various applications as well as "internal" memory capacity. Typically in popular FPGA platforms (like Xilinx), these internal memory blocks would be implemented using BRAMs. The part of the sequential code that takes significant amount of time during execution is chosen as a kernel. HEB is designed based on the kernels extracted from the application and is coded using a hardware description language (HDL). The ASIC synthesis of the hardware description of the kernels can be carried out for different performance parameters like area, speed, and power. In our present methodology, we have not considered power as a parameter but is part of our future work.

The profile data of the application which has to be accelerated is generated and the data access patterns are analyzed for the inputs to the HEBs. There can be many ways to implement the application using the HEBs. This also depends on the amount of concurrency and pipelining stages achievable including overlap of I/O and computation time. This essentially corresponds to the FPGA architecture constrained design space to be explored using the selected HEBs.

Performance evaluation is carried out on each type of design implementation using the HEB(s). For a particular implementation of the application kernel using HEBs, the memory bandwidth requirement is analyzed. The number of HEBs which can be incorporated in a particular device would be constrained by the device size. Based on memory bandwidth requirement, and area constraints, an optimal number as well as combination of HEBs is chosen for the fabric. This becomes an important factor as the HEBs occupy significant area of the FPGA and can be used only for a specific set of function(s)/operation(s). If the memory bandwidth cannot support effective utilization of large number of HEBs, then there would be a lot of chip area that would not have been effectively utilized. The architecture description of the FPGA is built considering all of these design considerations. The application can be mapped to the new fabric using any of the flows like VPR or VEB and the application acceleration can be estimated. The area and the speed at which design works are obtained and compared with the implementation of the application on existing hardware accelerators like FPGA without HEBs as well as software only implementation.

3.4.1 Performance Estimation of Application Acceleration

The placed and routed design gives the clock period which can be used for performance estimates. The VEB methodology is used to obtain these parameters. Analytical models can be built to calculate the overall execution time of the application on systems with FPGAs incorporating these HEBs. Typically, such FPGA accelerators would be built on separate cards with standard interfaces (like PCIe) to the processor. Analytical models also have to capture the communication delays between the different modules and the delays in the interfaces between host and the accelerator cards. For some of the systems, it is difficult to build analytical models where the speedups are significantly dependent on input parameters. System simulation is the only option for such designs. The system simulation can be done by modeling in system-level modeling languages such as System-C. We have used analytical model for "Docking" application and system level model for "Genome Sequencing" application.

3.5 Summary

A methodology to evaluate effectiveness of HEBs while exploring design space will be helpful in designing future fabrics. In this chapter, we have proposed a method to explore and evaluate HEBs for FPGA-based accelerators. Our exploration can be used to estimate the number of HEBs to be embedded for effective acceleration. This methodology can be used to improve the quality of HEBs. Clearly, external memory bandwidth, area constraints that limit the number of HEBs as well as internal memory size constraint are important parameters to be considered. Our exploration considers all the above constraints to predict the speedup for the FPGAs incorporating these HEBs. In the subsequent chapters, we discuss acceleration of two bioinformatics applications. We evaluate custom HEBs for these two applications using our methodology.

References

1. Graham, S.L., Kessler, P.B., Mckusick, M.K.: Gprof: a call graph execution profiler. SIGPLAN Not. **17**(6), 120–126 (1982)
2. Graphics, M.: ModelSim Simulator. http://www.mentor.com/products/fpga/model (2015)
3. Ho, C.H., Leong, P.H.W., Luk, W., Wilton, S.J.E., Lopez-Buedo, S.: Virtual embedded blocks: a methodology for evaluating embedded elements in FPGAs. In: IEEE Symposium on FCCM (2006)
4. Intel: Vtune Profiler. http://software.intel.com/en-us/intel-vtune/ (2015)
5. Kienhuis, B., Deprettere, E.F., Wolf, P.V.D., Vissers, K.A.: A Methodology to Design Programmable Embedded Systems—The Y-Chart Approach. In: SAMOS, pp. 18–37. Springer-Verlag, London, UK (2002). http://dl.acm.org/citation.cfm?id=646466.691571

6. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '05, pp. 190–200. ACM, New York, NY, USA (2005)
7. Luu, J., Kuon, I., Jamieson, P., Campbell, T., Ye, A., Fang, W.M., Rose, J.: VPR 5.0: FPGA cad and architecture exploration tools with single-driver routing, heterogeneity and process scaling. In: ACM/SIGDA FPGAs (2009)
8. Synopsys: Synopsys Design Compiler. http://www.synopsys.com (2015)

Chapter 4
FPGA-Based Acceleration of Protein Docking

Abstract In this chapter, we discuss FPGA-based acceleration of a commonly used protein docking application. The docking application is compute intensive and involves floating-point computations. We used the methodology discussed in Chap. 3 to accelerate the application. Accordingly, we explain the identification of kernel from the software implementation and their FPGA implementation to get performance benefits.

4.1 FTDock Application

There are many software docking applications as discussed in Chap. 2. We chose FTDock application, an open-source implementation of rigid-body docking and show that it can be accelerated using FPGAs. Most of the FFT-based docking applications are provided as executables as open-source. For understanding the application and accelerating it, we need the code and hence we chose FTDock software, which provides the C-code under general public license (GPL). FTDock application performs rigid-body docking on two molecules. FTDock outputs multiple predictions that can be screened using biochemical information. FTDock implements shape complementarity function developed by Katchalski-Katzir et al. [1] and an electrostatics complementarity Fourier correlation function defined by Gabb et al. [2]. The block diagram is shown in Fig. 4.1. The inputs to the application are two molecules, a large molecule '*A*' and a small molecule '*B*'. These molecules are typically in protein data bank (PDB) file format. PDB format is a text file format which describes the 3D structure of the molecules. This file describes the coordinates of the atoms that are part of the protein. These files are preprocessed to remove the hydrogens and extra oxygen at the carboxyl terminal. Each molecule is discretized and placed in a 3D grid of the definite grid size (cuboid). The number of unit-cubes in the grid will determine the memory size, as the data in each grid unit has to be stored. The FFT word length will depend mainly on this grid size (length of the edge of the cube). This becomes an important factor which determines the speedup as this data has to be sent for computing through a bus.

B.S.C. Varma et al., *Architecture Exploration of FPGA Based Accelerators for BioInformatics Applications*, Springer Series in Advanced Microelectronics 55, DOI 10.1007/978-981-10-0591-6_4

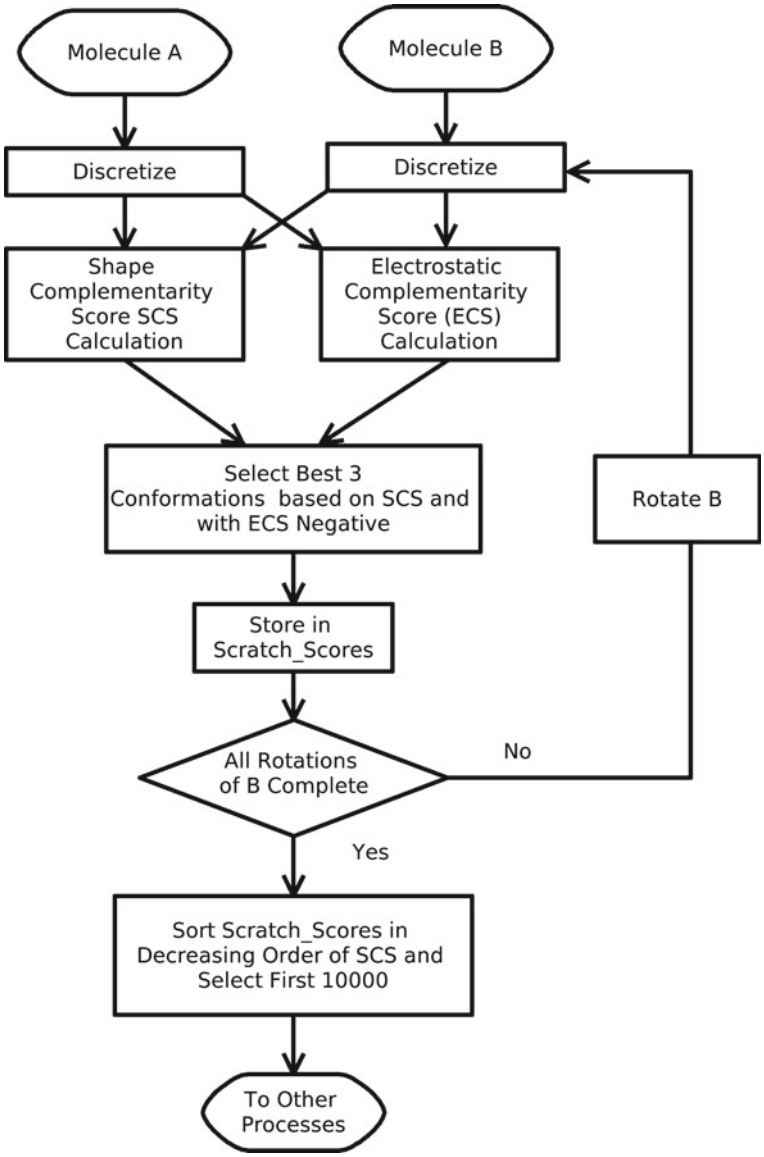

Fig. 4.1 Block diagram of FTDock

The grid span is calculated by (4.1).

$$\text{grid span} = 1 + \text{diameter of molecule 1} + \text{diameter of molecule 2} \tag{4.1}$$

This grid span is divided by a definite grid size to get the actual number of grid units. Shape complementarity score (SCS) and electrostatic complementarity score (ECS) are calculated on the discretized grid. The electrostatic complementarity is used as a binary filter and the ones with positive scores are neglected. The best three conformations based on SCS with negative ECS are selected and stored. The molecule '*B*' is rotated and then discretized. The scores are calculated on the rotated molecule '*B*'. The process is repeated till all rotations of molecule '*B*' are finished. The stored conformations are arranged in decreasing order of SCS and the best 10,000 values are selected, which are used by other applications for further processing and filtering.

4.1.1 Shape Complementarity

Geometric surface recognition is used in finding the shape complementarity score. The block diagram is shown in Fig. 4.2.

Each of the molecules '*A*' and '*B*' are placed in a 3D grid of N ∗ N ∗ N, of definite grid size (default 7 Å). Every node is assigned a value based on the (4.2) and (4.3).

$$A_{l,m,n} = \begin{cases} 1 & \text{for surface,} \\ \rho & \text{for core,} \\ 0 & \text{outside.} \end{cases} \tag{4.2}$$

$$B_{l,m,n} = \begin{cases} 1 & \text{inside molecule,} \\ 0 & \text{outside molecule.} \end{cases} \tag{4.3}$$

1.5 Å surface layer on molecule '*A*' is used to score a node on the surface. A grid node within 1.8 Å is considered within the core. ρ has a default value of -15. ρ actually defines the degree of acceptable surface overlap. The application calculates the score by superimposing grids as shown in Fig. 4.3 and translating the movable grid '*B*' by shifts of α, β, and γ angles and the correlation function based on shape complementarity is calculated using (4.4).

$$C_{\alpha,\beta,\gamma} = \sum_{l=1}^{N} \sum_{m=1}^{N} \sum_{n=1}^{N} A_{l,m,n} B_{l-\alpha,m-\beta,n-\gamma} \tag{4.4}$$

A negative score indicates that there is overlap between the molecules and higher score shows better surface complementarity. The score calculation has $\Theta(N^6)$ complexity as there are N^3 multiplications for N^3 transformations of α, β, and γ. This complexity is reduced significantly to $\Theta(N^3 \log N)$ by using FFTs.

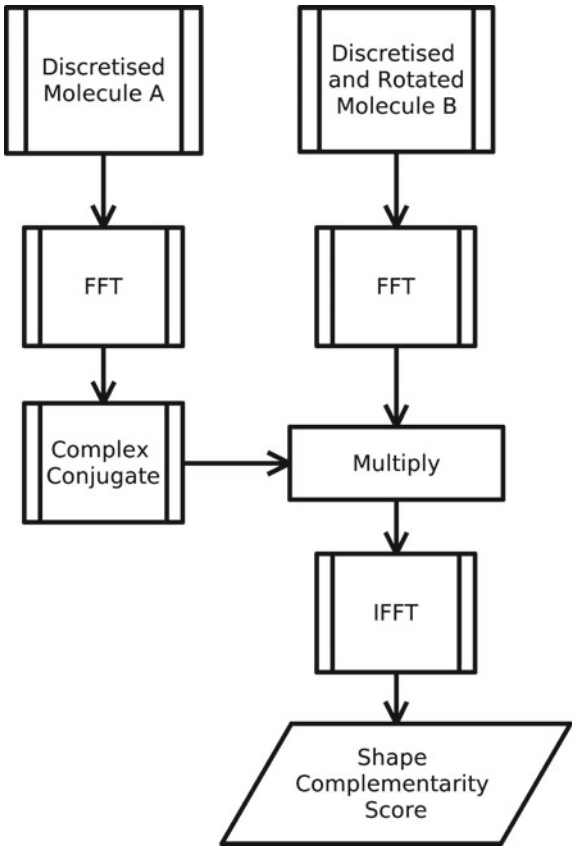

Fig. 4.2 Block diagram for calculating shape complementarity score

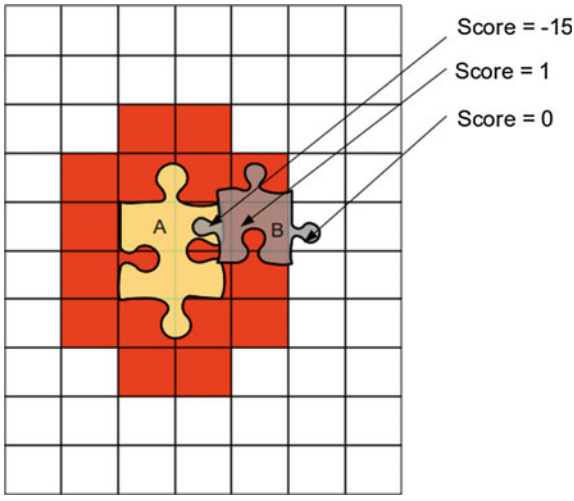

Fig. 4.3 Shape complementarity score calculation after superimposing grids

The correlation function is calculated using FFT by using the (4.5) and (4.6).

$$\mathrm{FFT}(C) = \mathrm{FFT}^{*}(A) * \mathrm{FFT}(B) \tag{4.5}$$

$$C = \mathrm{IFFT}[\mathrm{FFT}(C)] \tag{4.6}$$

where $\mathrm{FFT}^{*}(A)$ is the FFT of the complex conjugate of matrix 'A'. Molecule 'B' is rotated about one of its Euler angles until all orientations are captured. The rotation is done around the geometric center of the mobile molecule in steps of α. The results are accurate for smaller angles of α, as score calculation will be done at more points, and chances of getting the best orientation increases. But as α decreases, the number of FFTs which has to be evaluated increases and hence the computation time increases. The default value of α is 12° in the FTDock application. There are $360 * 360 * 180/\alpha^3$ orientations. For example, there are 13,500 orientations for 12° rotation. Many of these orientations are degenerate and are removed by using (4.7) [3],

$$\alpha = \cos^{-1}\left(\frac{t_r\left(R_1 * R_2^T\right) - 1}{2}\right) \tag{4.7}$$

where R_1 is the rotation matrix of the first orientation, R_2^T is the transpose of the rotation matrix of the second orientation, and t_r is the matrix trace. Thus for 12° rotation, there are 9240 nondegenerate rotations.

4.1.2 Electrostatic Complementarity

Electrostatics also plays a very important role in the docking process. A simple Coloumbic model is used to calculate the electrostatic complementarity. Specific charges are assigned to each atom of the protein and then placed in a grid using Table 4.1 [2].

Table 4.1 Charges used in Coulombic electrostatic fields [2]

Peptide backbone	Charge	Side-chain atoms	Charge
Terminal-N	1.0	Arg-N^{η}	0.5
Terminal-O	−1.0	Glu-N^{ε}	−0.5
C^{α}	0.0	Asp-O^{δ}	−0.5
C	0.0	Lys-N^{ζ}	1.0
O	−0.5	Pro-N	−0.1
N	0.5		

The electric field at each node of the grid is calculated using (4.8).

$$\phi_i = \sum_{j=1} \frac{q_j}{\varepsilon\left(r_{ij}\right) r_{ij}} \tag{4.8}$$

where ϕ_i is the electric field strength at node i, q_j is the charge on atom j, and r_{ij} is the distance between i and j.

$\varepsilon(r_{ij})$ is given by (4.9) [4].

$$\varepsilon(r_{ij}) = \begin{cases} 4 & \text{when } r_{ij} \leq 4\,\text{Å} \\ 38r_{ij} - 224 & \text{when } 4\,\text{Å} < r_{ij} < 8\,\text{Å} \\ 80 & \text{when } r_{i,j} \geq 8\,\text{Å} \end{cases} \tag{4.9}$$

Considering docking of two molecules, a large molecule 'A' and a small molecule 'B', charges are assigned to molecule 'B' and discretized to get $q_{l,m,n}$ using (4.10).

$$B_{l,m,n} = q_{l,m,n} \tag{4.10}$$

'A' is calculated using (4.11).

$$A_{l,m,n} = \begin{cases} \phi_{l,m,n} & \text{grid excluding core,} \\ 0 & \text{inside core} \end{cases} \tag{4.11}$$

The electrostatic complementarity score is calculated using (4.12).

$$C_{\alpha,\beta,\gamma} = \sum_{l=1}^{N} \sum_{m=1}^{N} \sum_{n=1}^{N} A_{l,m,n} B_{l-\alpha,m-\beta,n-\gamma} \tag{4.12}$$

Fourier transform is used to reduce the time complexity, and hence to do faster computation. Similar to shape complementarity, score calculation following (4.13) and (4.14) are used.

$$\text{FFT}(C) = \text{FFT}^*(A) * \text{FFT}(B) \tag{4.13}$$

$$C = \text{IFFT}[\text{FFT}(C)] \tag{4.14}$$

where $\text{FFT}^*(A)$ is the FFT of the complex conjugate of matrix 'A'. The electrostatic score is used as a binary filter and the orientations with positive scores are rejected even if they have good shape complementarity score as they are not favorable.

4.2 Profiling Results

We took an open-source implementation of rigid-body docking called FTDock [5]. The application was written in *C* language. We used VTune [6] to profile the FTDock application, running the docking application for different ligand–protein combinations. Table 4.2 shows the profiling results of FTDock application averaged over 10 protein–ligand pairs with grid sizes varying between 128 and 232 and with 12° angle step for rotation. The application was executed on a PC with Intel Core 2 duo E4700, 2.6 GHz processor with 4 GB RAM. The ***forward FFT*** and ***inverse FFT*** take 94.17 % of the total time after removing the overhead due to the profiling tool. The ***discretise*** function took 2.81 % and ***electric field*** calculation 2.41 % of the total execution time.

4.2.1 Need to Speedup

The evaluation of FFT is compute intensive as it contains floating-point multiplications. The FTDock application has to do 3D FFT for each orientation of the smaller molecule followed by IFFT to get the score. This process takes considerable time to execute on CPU, and hence there is need for accelerating the FFT application. For example, the FFT and IFFT evaluation in docking of barnase (1a2p) and barstar (1a19) with grid size $232 * 232 * 232$ took 17 h on a PC with Intel Core 2 duo E4700, 2.6 GHz processor with 4 GB RAM. Calculations are independent for each rotation, so thread-level parallelism can be extracted to get speedup in software implementation.

Table 4.2 Profiling results of FTDock application averaged over 10 protein–ligand pairs with various grid sizes

Function	Total execution time in %	Normalized execution time removing VTune overhead
Forward FFT	45.44	47.22
Inverse FFT	45.18	46.95
Discretize structure	2.69	2.81
Electric field	2.32	2.41
Electric point charge	0.45	0.47
Rotate structure	0	0
FFT create plan	0.01	0.01
_mcount(VTune Overhead)	3.91	–

4.2.2 *Earlier Attempts to Speedup*

Recently, many researchers have reported accelerator implementations for bioinformatics applications [7, 8]. Bharat and Herbordt [9], Sukhwani and Herbordt [10] have studied the acceleration of PIPER protein docking application using FPGAs and GPUs. PIPER docking requires a computation of a 3D correlation function. In order to accelerate the correlation computations in software implementations, fast fourier transform (FFT) is used. The code was modified for FPGA and direct correlation functions were used instead of FFT computations. GPU implementation with the modified code with only the correlation function gave lower speed when compared to FFT-based implementation. This was because the GPUs perform well for large number of floating-point computations. Even the multicore performance was better for FFT-based code. Harald Servat et al. have ported FTDock application on CELL-BE architecture [11].

4.3 Choice of Single Precision

The original code of FTDock uses FFTW library, which uses double precision as default [12]. Double-precision floating-point arithmetic on FPGAs clearly consumes a large amount of resources thus limiting the achievable parallelism under resource constraints. The output of FTDock is a selection of first 10,000 orientations of ligand arranged in the decreasing order of the shape complementarity score. The selection of actual conformations with high scores is important whereas the actual value of the score calculated is not significant. Given this property of the application, we ran several experiments to determine whether reduced precision produces acceptable results.

Table 4.3 shows the results containing the orientations selected in double precision and missing in single precision. Four pairs of different proteins were chosen. FTDock was run with both double-precision and single-precision arithmetic. The first "n" best confirmations were chosen in both cases and compared. Wherever there was a difference it was noted down. Column 1 in the Table 4.3 includes the values of "n" whereas the other column indicates the difference in case of different proteins. Note a zero difference implies no loss due to single-precision arithmetic. From the results, we can see that the ligand orientations which were selected using double precision and which were not selected during the single-precision run were the ones with very low shape complementarity score. This is because they appear only when we choose 10,000 shapes. The implication is that the results would not vary much by using single-precision floating-point arithmetic for calculating the FFTs instead of double precision. The fact that reduced precision does not impact the quality of result allows us to exploit parallelism to build a hardware accelerator for the application using single-precision floating-point arithmetic. This is because with single-precision arithmetic, current FPGA devices provide adequate resources to support large number of concurrent computation.

Table 4.3 Results of mismatches between single-precision and double-precision floating-point arithmetic selection

Selection range	2ka-5pti	1tgn-4pti	2ptn-6pti	2jel-1poh
0–100	0	0	0	0
101–200	0	0	0	0
201–500	0	0	0	0
501–700	0	0	0	0
701–1000	0	0	0	0
1001–2000	0	0	2	0
2001–5000	7	4	7	7
5001–7000	9	4	10	6
7001–9000	10	6	10	9
9001–10,000	221	214	184	192

4.4 FPGA Resource Mapping

The 3D-FFT of the rotated and discretized smaller molecule has to be multiplied with the complex conjugate of the 3D-FFT of the larger molecule. Complex conjugate of the FFT is done only once so we chose to do it on the host PC. The multiplication of the FFT results is done on the host PC as we wanted to utilize most of the FPGA resources for the hardware acceleration of the part of the software code which was taking more time to execute.

As it is evident from the profiling data, the FFT computation is the most compute-intensive portion of the application. Hence, we decided to implement this in hardware in the design that we now describe.

We consider the accelerator setup where, a board has an FPGA which gets data from host PC through PCIe bus. For estimating the speedup for FTDock, we choose four different Xilinx FPGAs, and calculate the speedup for each of the FPGAs. The Table 4.4 contains the resources available in each of the FPGAs.

An efficient way to compute DFT is by using Fast Fourier Transform which has a computational complexity of O(N log N). An N point 3D-DFT is given by (4.15).

$$F(n_1, n_2, n_3) = \sum_{k_3=0}^{N_3-1} \sum_{k_2=0}^{N_2-1} \sum_{k_1=0}^{N_1-1} f(k_1, k_2, k_3) e^{ik_1 w_1} e^{ik_2 w_2} e^{ik_3 w_3} \tag{4.15}$$

Table 4.4 FPGA resources [13]

FPGA	DSP 48E1 slices	18 kb block RAMs
XC6VSX475T	2016	2128
XC6VSX315T	1344	1408
XC6VHX565T	864	1824
XC6VLX75T	288	312

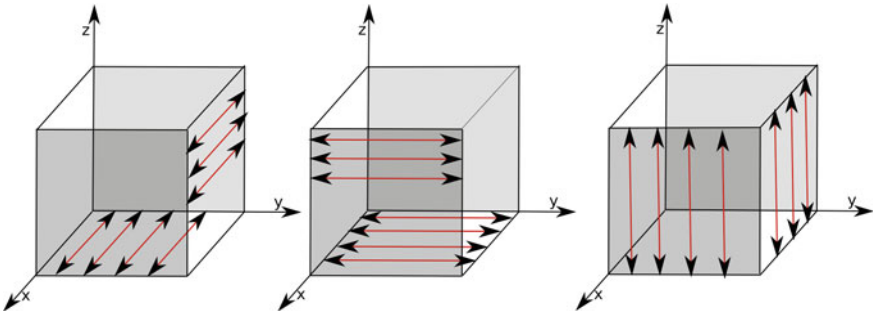

Fig. 4.4 3D-FFT calculation using 1D-FFT

where $w_1 = 2\pi\, n_1/N_1$, $w_2 = 2\pi\, n_2/N_2$ and $w_3 = 2\pi\, n_3/N_3$. 3D-FFTs are calculated using 1D-FFT with some variants of Cooley Tukey algorithm [14]. Single-dimension Fourier transform in each of the three dimensions of the 3D matrix is used to calculate 3D-FFT as shown in Fig. 4.4.

We use simple Algorithm 4.1 to calculate the 3D-FFT.

Algorithm 4.1 Algorithm for calculating 3D-FFT

```
begin
  for i = 1 to N step 1 do
    for j = 1 to N step 1 do
      FFT1(:,i,j) = 1D-FFT{x(:,i,j)};
    end
  end
  for i = 1 to N step 1 do
    for j = 1 to N step 1 do
      FFT2(i,:,j) = 1D-FFT{FFT1(i,:,j)};
    end
  end
  for i = 1 to N step 1 do
    for j = 1 to N step 1 do
      FFT3(i,j,:) = 1D-FFT{FFT2(i,j,:)};
    end
  end
```

In the algorithm, 'x' is the input 3D-matrix. FFT1 and FFT2 are 3D-matrices which store the intermediate values. The x(:,i,j) returns the array of all values in first dimension with second dimension 'i' and third dimension 'j'.

We give the performance estimate for implementing this algorithm on FPGA. For implementing this on FPGA, we chose the following model: PCIe bus transfers the required data from the CPU for computation and multiple cores compute on the FPGA concurrently. The bus being a shared resource, we set up a pipeline for data I/O as described later in Sect. 4.5. 3D-FFT requires two N^3 3D-matrices as input and it generates a 3D matrix as an output. Single-precision arithmetic is used and each data of matrix is represented by 32 bits (4 Bytes). For $N = 256$, a total of $256^3 * 4$ Bytes

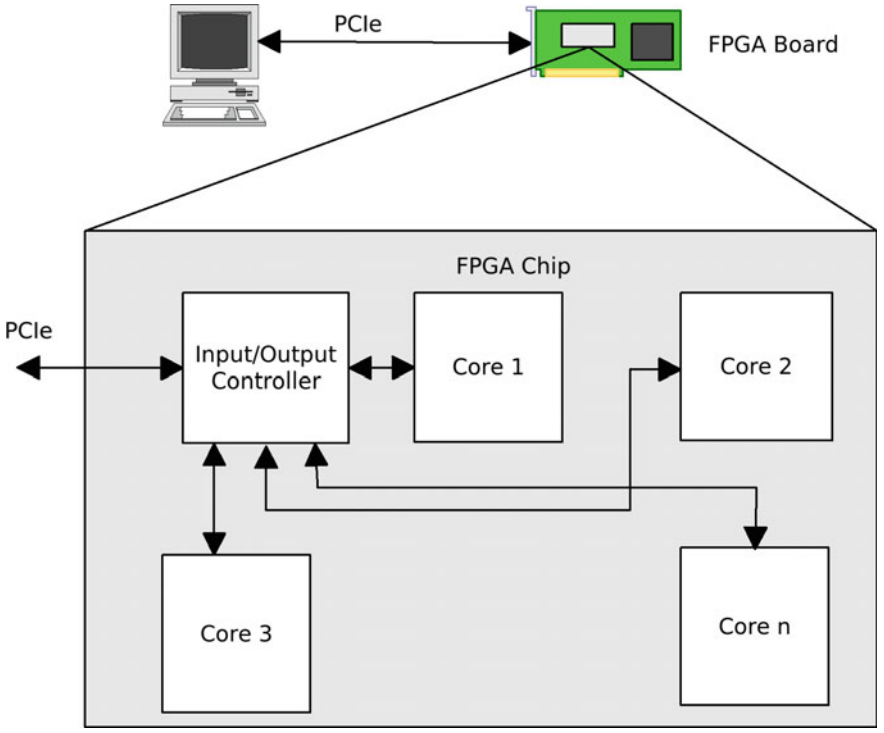

Fig. 4.5 Mapping FFT cores onto FPGA

= 67,108,864 Bytes ~68 MB are needed for storing one matrix. As it is not possible to store whole of the matrix on the FPGA and while performing the calculations, we have designed a model in which the FPGA computes on the data sent by the host CPU and sends it back to CPU after computation. The block diagram is as shown in Fig. 4.5.

We have attempted to accelerate the 3D-FFT part of the FTDock application. The input/output controller controls the input, and sends the required input data to particular floating-point FFT core. The FFT values which are computed are transferred back to CPU. The multiplication of the complex conjugate is done on the CPU. The same process of transferring data from the CPU to FPGA and transmitting data back to CPU after computation is repeated for calculating IFFTs. Sorting and selection of ligand conformations is done on the CPU itself.

Xilinx Core Generator provides 3 types of FFT core implementations for burst I/O. They are radix-4 burst I/O, radix-2 burst I/O, and radix-2 burst I/O lite. We chose the burst I/O mode as we wanted to exploit the DMA transfer from host PC to FPGA through PCIe interface. Table 4.5 shows the resources used by each of the cores for computing a 256-point FFT. The computation time estimate is also provided by the core generator for running the core at 550 MHz [13].

Table 4.5 FFT core resource utilization estimate [13]

FPGA resources	Radix-4, Burst I/O		Radix-2, Burst I/O		Radix-2, Lite I/O	
Mode	256 pt FFT	128 pt FFT	256 pt FFT	128 pt FFT	256 pt FFT	128 pt FFT
DSP slices	40	40	12		6	6
18 kb block-RAMs	16	16	8		6	6
Computation time in µs	2.565	1.402	4.045	2.036	5.726	2.729

Table 4.6 Number of cores that can be instantiated

FPGA	Radix-4	Radix-2	Radix-2 Lite
XC6VSX475T	45	151	302
XC6VSX315T	30	100	201
XC6VHX565T	19	64	129
XC6VLX75T	6	21	43

Table 4.6 shows number of cores which can be instantiated assuming 90 % of DSP slices and Block RAMs are available for mapping these cores.

We use the instantiation of the cores to estimate the performance of FPGA-based acceleration.

4.5 Estimation Results

In order to use a pipelined model with as few stalls as possible, we have used the timing model in which the input data is sent from the host in burst mode to each of the cores. The computing cores start computing as soon as they have received their respective data. The data is transmitted to the host PC as soon as the computation is over. There can be two cases,

1. $(N_c - 1) * T_b < T_{\text{comp}}$
2. $(N_c - 1) * T_b > T_{\text{comp}}$

where

$T_b = T_i = T_o$ is the time taken to transfer required data for one core from host PC to FPGA board
(T_i) or in the reverse direction (T_o). The transfer uses the PCIe bus interface in DMA mode.
$T_{bi} = T_b$ is the time taken to transfer the required data for *i*th core.
T_{comp} is the time taken by one core to compute on an input data set,

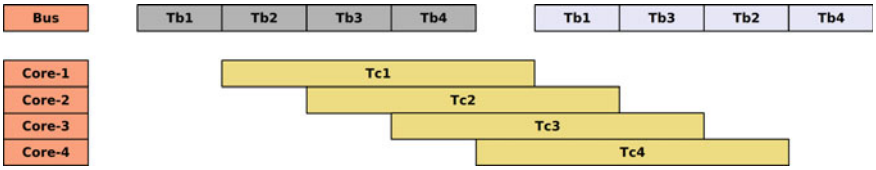

Fig. 4.6 Timing diagram for case 1: $(N_c - 1) * T_b < T_{\text{comp}}$

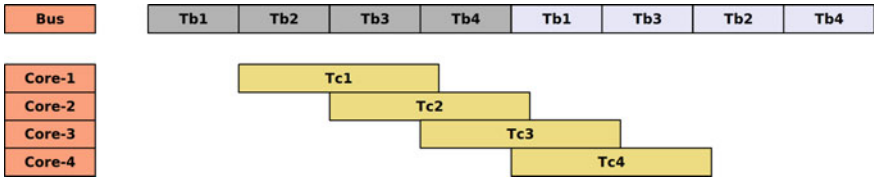

Fig. 4.7 Timing diagram for case 2: $(N_c - 1) * T_b > T_{\text{comp}}$

$T_{ci} = T_{\text{comp}}$ is the time taken for computation on 'ith'core,
and N_c is the number of cores that are instantiated.

The timing diagram for $N_c = 4$ in both the cases are shown in Figs. 4.6 and 4.7. Clearly, the two cases correspond to the shared resource for being a bottleneck or otherwise. This can be extended to any number of cores.

Taking PCIe bandwidth to be 1 GBps for data transfer between host CPU and FPGA board, the total time for execution of one 'n' point FFT can be given by (4.16).

$$\text{Total Time}_{\text{(n point 3D-FFT)}} = \begin{cases} \lceil \frac{k}{N_c} \rceil * \left[N_c * T_b + T_{\text{comp}} + T_b \right] & \text{when } n * T_b < T_{\text{comp}}, \\ \lceil \frac{k}{N_c} \rceil * \left[N_c * T_b * 2 \right] & \text{when } n * T_b > T_{\text{comp}} \end{cases} \tag{4.16}$$

where k is the total number of 'n' point FFT required and N_c is the number of cores.

The $T_b = 1024 * 10^{-9}$ for 256-point 3D-FFT, and $512 * 10^{-9}$ for 128-point FFT, which is calculated using the PCIe bandwidth.

Based on (4.16), we estimate the time for calculating one 256-point FFT for each FPGA. The results are given in Table 4.7. The overall speedup is calculated by (4.17),

$$\text{maximum speedup} = \frac{p}{1 + f \cdot (p - 1)} \tag{4.17}$$

where p is the speedup of part of the code that is accelerated, which in our case is 3D-FFT and f ($0 < f < 1$) is the fraction of time spent in the part that was not improved. In our implementation, p is the speedup in 3D-FFT calculation, i.e., time taken in software divided by the time taken by the FPGA hardware. The fraction of code which is not accelerated is 0.0583 (5.83 %). The results containing estimated overall speedup of FTDock application are tabulated in Table 4.8. The speedup using different FPGA devices is plotted in Fig. 4.8. From the graph, we can see that as the number of cores that can be instantiated increases, the speedup increases.

Table 4.7 Computation time of one 3D-FFT in milli seconds

Software execution time		FPGA device	Radix-4, Burst I/O		Radix-2, Burst I/O		Radix-2 Lite, Burst I/O	
256 pt FFT	128 pt FFT		256 pt FFT	128 pt FFT	256 pt FFT	128 pt FFT	256 pt FFT	128 pt FFT
2005.2398	1028.885	XC6VSX475T	217	27	208	26	206	26
Software executed on PC with Intel Core 2 duo E4700, 2.6 GHz processor with 4 GB RAM		XC6VSX315T	225	28	211	26	208	26
		XCVHX565T	238	30	217	27	212	26
		XC6VLX75T	318	41	249	31	232	29

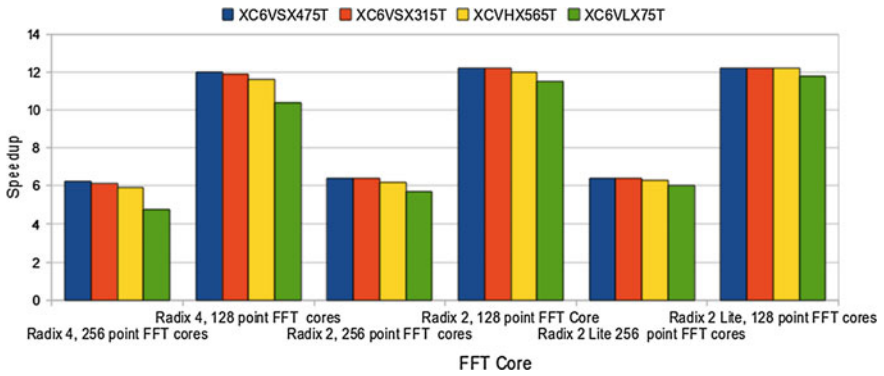

Fig. 4.8 Plot of speedup for different FPGA devices

Table 4.8 Overall speedup achieved using FPGA-based acceleration compared to software

FPGA	Radix 4, Burst I/O		Radix 2, Burst I/O		Radix 2 Lite, Burst I/O	
–	256 pt FFT	128 pt FFT	256 pt FFT	128 pt FFT	256 pt FFT	128 pt FFT
XC6VSX475T	6.24×	12x	6.4×	12.2×	6.4×	12.2×
XC6VSX315T	6.1×	11.9×	6.4×	12.2×	6.4×	12.2×
XCVHX565T	5.9×	11.6×	6.2×	12×	6.3×	12.2×
XC6VLX75T	4.8×	10.4×	5.7×	11.5×	6×	11.8×

Results show speedup of around 6× for 256-point FFT and 12× for 128-point FFT using single FPGA device.

From the results, we estimate a maximum speedup of 6.4× for 256-point 3D-FFT and 12.2× for 128-point 3D-FFT, when compared to software application run on a desktop PC with Intel Core 2 duo E4700, 2.6 GHz processor with 4 GB RAM. The speedups are less in 256-point FFT when compared to 128-point FFT because the data required by 256-point FFT is more and hence T_b is high, which is the data transmission time. Note that the results use a single FPGA device and it is not uncommon to have accelerator boards with 16 or more devices.

4.6 Summary

In this chapter, we discussed FPGA-based acceleration of protein docking application. We profiled the application and found that 3D-FFT was taking most of the time. The change in quality of the output by use of single precision was analyzed. FPGA implementation of the 3D-FFT algorithm was done. We developed an analytical model to predict the speedups. Estimating performance before implementation will help the designer take early decisions in the design cycle. This will not only help in

choosing the right FPGA but also help in design for better performance. We illustrate this in the next chapter by accelerating genome assembly, which is yet another commonly used bioinformatics application.

References

1. Katchalski-Katzir, E., Shariv, I., Eisenstein, M., Friesem, A.A., Aflalo, C., Vakser, I.A.: Molecular surface recognition: determination of geometric fit between proteins and their ligands by correlation techniques. In: Proceedings of the National Academy of Sciences of the United States of America, vol. 89
2. Gabb, H.A., Jackson, R.M., Sternberg, M.J.E.: Modelling protein docking using shape complementarity, electrostatics and biochemical information. J. Mol. Biol. **272** (1997)
3. Lattman, E.E.: Optimal sampling of the rotation function. Acta Crystallogr. B **28**, (1972)
4. Biopolymers The theory of ionic saturation revisited. **24**(3), 427–439 (1985)
5. Sternberg, M.J.E., Aloy, P., Gabb, H.A., Jackson, R.M., Moont, G., Querol, E., Aviles, F.X.: A Computational system for modeling flexible protein-protein and protein-DNA docking. In: Proceedings of the 6th International Conference on Intelligent Systems for Molecular Biology, pp. 183–192 (1998)
6. Intel: Vtune Profiler. http://software.intel.com/en-us/intel-vtune/ (2015)
7. Integrating FPGA Acceleration into HMMer.: Parallel Comput. **34**
8. Chen, Y., Schmidt, B., Maskell, D.L.: A reconfigurable bloom filter architecture for BLASTN. In: ARCS (2009)
9. Bharat, S., Herbordt, M.C.: GPU Acceleration of a production molecular docking code. In: GPGPU (2009)
10. Sukhwani, B., Herbordt, M.C.: Acceleration of a production rigid molecule docking code. In: FPL (2008)
11. Servat, H., González-Alvarez, C., Aguilar, X., Cabrera-Benitez, D., Jiménez-González, D.: Drug design issues on the CELL-BE. In: HiPEAC'08: Proceedings of the 3rd International Conference on High Performance Embedded Architectures and Compilers (2008)
12. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. In: Proceedings of the IEEE (2005)
13. Xilinx: Xilinx FPGAs, ISE. http://www.xilinx.com (2015)
14. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex fourier series. Math. Comput. **19**(90), (1965)

Chapter 5
FPGA-Based Acceleration of De Novo Genome Assembly

Abstract De novo Assembly is a method in which the genome is constructed using the reads without using reference sequence. It is the only way to construct new genomes. This method is also used when reference genome is available because the construction is unbiased. The genome assembly involves large amounts of data and string comparison and hence takes significant time to execute. In this chapter, we show achieved speedups over software implementations using FPGA-based accelerators.

5.1 Application

De novo assembly can be divided into three categories; Greedy, overlap layout consensus (OLC), and de Bruijn graph-based assemblers. The softwares like PCAP [1] and TIGER [2] that use greedy approaches make use of the overlap information for doing the assembly. In the greedy method, the pairwise alignment of all reads is done and the reads with the largest overlap is merged. This process is repeated till a single lengthy sequence is obtained.

The OLC method is a graph-based method where an overlap graph is constructed from the reads. Some of the software assemblers based on OLC are Edena [3] and CABOG [4]. The reads become the node and edges show the overlap information. The nodes are placed in the form of a graph. Multiple sequence alignment (MSA) is done with the reads having more than two edges. Based on this, consensus sequence is constructed and sequencing errors are removed. A "***Hamiltonian path***" (path which visits every node exactly once) in the graph is used to construct the contiguous sequences (contigs). Later, the whole genome is constructed using these contigs.

Figure 5.1 shows an example for assembly using OLC method. A graph is constructed with the reads as the nodes and the overlap region as edges. For example, ATCGGTCGAT has an edge to node GTCGATCAGT, since they overlap. After construction of the whole graph, Hamiltonian path (path which connects all nodes) has to be found to construct the genome. Whenever there are more paths available between two nodes, MSA is carried out to remove the errors. Multiple sequences are arranged in a manner where there is maximum overlap between the sequences. In the figure,

B.S.C. Varma et al., *Architecture Exploration of FPGA Based Accelerators for BioInformatics Applications*, Springer Series in Advanced Microelectronics 55, DOI 10.1007/978-981-10-0591-6_5

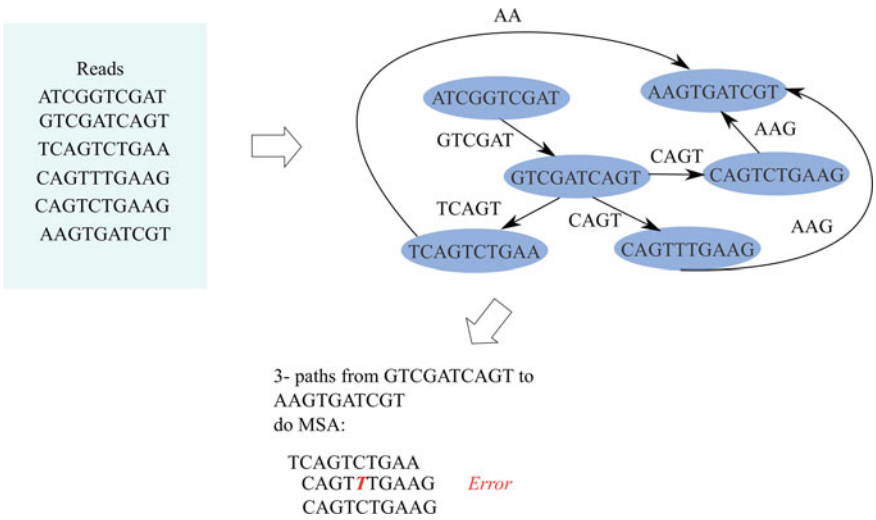

Fig. 5.1 Example showing OLC-based assembly

there are three paths between GTCGATCAGT and AAGTGATCGT. A consensus is reached after MSA is done by looking at the majority occurrence. In the figure, "*T*" is an error and the consensus sequence is found to be TCAGTCTGAAG.

The de Bruijn graph assembly also uses a graph where the nodes are k-mers. A "***k-mer***" is a sequence of "k" base pairs. Some of the popular software assemblers based on de Bruijn graph are PASHA [5], Velvet [6] and Euler [7]. All the reads are broken into respective k-mers, i.e., all substrings of length "k." A graph is constructed with $(k-1)$-mers as nodes and the k-mers as edges. This graph contains all the overlap information contained in the reads for a particular k-mer length. Due to errors in the reads, there can be a chain of nodes that are disconnected, i.e., they do not converge into the graph. These are called "***tips***". The errors can also cause the graph to have redundant paths that have same starting and ending point, i.e., the paths converge back into the graph. These are called "***bubbles***". These tips and the bubbles are removed using heuristics and sometimes with sequence comparisons. "***Euler path***" (path which visits every edge exactly once) is used in this de Bruijn graph to construct the contigs, which in turn is used for constructing the whole genome. The de Bruijn graph consists of k-mers and since the number of k-mers in a genome is large, the graph is also large. Since there is a need to store these k-mers in RAM for constructing the graph, the softwares use large amounts of RAM (more than 150 GB for human genome).

The de novo genome construction from NGS data is complex due to the following reasons:

1. A very large amount of data has to be processed. Most of the algorithms need processors with large amount of RAM for processing and storing the data. If the RAM usage has to be reduced, data partitioning has to be done to keep the "data

of interest" closer to the processor in the memory hierarchy. This process would involve swapping of data across the memory hierarchy and thus lead to increase in execution time.
2. There are some common sequences in the genome called repeats. Identifying the reads which form these repeats is nontrivial. Some plant genomes include more than 80 % of repeat sequences.
3. The sequencing machine has a constraint on length of reads. If the read length is less than repeat it is almost impossible to detect which portion of the genome the read came from.
4. The data generated by the sequencing machines are not fully accurate and contain errors. The genome constructed may be erroneous if the programs do not take error correction into consideration.

The choice of assembly software is mainly dominated by the quality of assembly, speed of assembly and the RAM needed for the execution. Many techniques are used for error correction and improving the quality of assembly [8, 9]. Considering speed as the criteria, graph-based assemblers are preferred over greedy assemblers. In the graph-based assemblers, de Bruijn graph-based assemblers have become more popular as they are faster than OLC-based assemblers. This is because, finding Hamiltonian path (path which visits every node exactly once) in a directed graph in OLC-based assemblers is a NP hard problem, while finding the Euler path is easier (solvable in polynomial time) [10]. Also, MSA of the reads used in OLC method for removing errors is both compute intensive and memory intensive when compared to the techniques used in de Bruijn graph-based assemblers.

5.1.1 Related Work with FPGA-Based Acceleration

Several groups have attempted to accelerate NGS short read mapping using FPGAs, where the genome is constructed by mapping the short reads to an already existing genome. Tang et al. [11] have accelerated short read mapping and achieved 42× speedup over software PerM [12]. Olson et al. [13] has also shown acceleration of short read mapping on FPGA. The authors compare their results with BFAST software [14] and show 250× improvement and 31× when compared to Bowtie [15]. Fernandez et al. [16] and Knodel [17] have also accelerated NGS short read mapping. The Convey Computer firm has developed the Convey Graph Constructor (CGC), which use FPGAs to accelerate de novo assembly [18]. They show speedups of 2.2× to 8.4×. We attempt to accelerate de novo genome assembly using FPGAs. As it is not fair to compare mapping-based assembly with de novo assembly, as both are different and have their own advantages and disadvantages, we compare our hardware implementation with existing novo software-based implementation.

5.2 Approach

We chose to accelerate de Bruijn graph-based assembly, as they take less amount of time to execute when compared to OLC-based assemblers. We use the fact that there is a lot of redundancy in short read sequencing. Landel and Waterman [19] describe the use of redundancy for getting good quality assembly. This redundancy helps in providing coverage as well as eliminating errors encountered during sequencing of these short reads.

De Bruijn graph-based software assemblers takes several GBs of RAM space while executing [20]. Efficient implementation of de Bruijn graph-based assemblers on an FPGA is difficult due to the memory resource constraints in current FPGAs. We model a hybrid approach where we implement a part of OLC-based assemblers on FPGA to remove the redundancy present in the reads. We run the de Bruijn graph-based assembler in software on the reduced set of reads from the FPGA. The key innovation is to use a (parallel) hardware implementation to remove the "redundancy" in the input data and use the state-of-the-art highly computationally intensive de Bruijn graph-based Velvet software [6] to build the consensus sequence. We choose to accelerate Velvet software since it is the most commonly used de novo genome assembly software and it takes significant amount of time to execute on general purpose processors. One of the features which was needed by our approach was error correcting feature included in the software package and since it was available in Velvet, we chose to use it.

A high-level design of our approach is shown in Fig. 5.2. The reads are passed through Redundancy Remover Unit (RRU) implemented in FPGA, which acts as preprocessor. The RRU is constructed using processing elements connected in series. All the processing elements form a pipelined structure and hence execute parallely. Each of the processing elements stores a sequence. Reads are passed through each of the processing element. The processing element checks for overlap region at its ends and if the overlap region is greater than a threshold value, it is extended. These extended sequences form the intermediate contigs which are given to Velvet for constructing the final contigs.

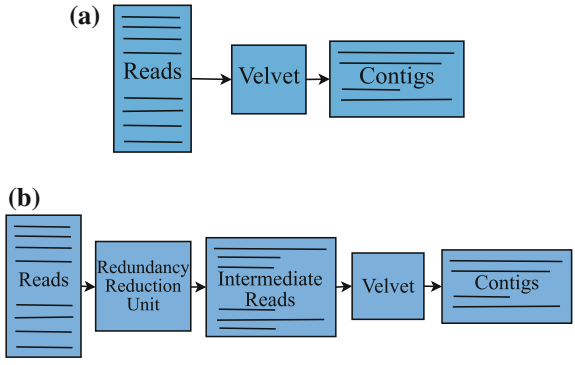

Fig. 5.2 Velvet flow and FPGA-based approaches. In our approach, we generate intermediate contigs which are given to Velvet for further processing

This approach allows us to effectively use the FPGA resources for removing redundancy in the reads.

5.2.1 Algorithm

The main thrust in our approach is to find the overlap region between reads and store the overlap region only once. This can be done using a greedy approach. A read from the read set is picked which is called "starter" sequence. The starter sequence is checked for extension with the rest of the reads in read set. This process has to be repeated until the starter does not extend, meanwhile removing the reads from read set which extend the starter. After many such iterations, we are left with a reduced read set and a extended starter. This extended starter which cannot be further extended is stored as an "***intermediate contig***". This process has to be repeated by picking a read from the remaining read set, so that we store the overlap information only once. After checking with all the reads, if it does not extend, it can be made an intermediate contig. If it gets extended, the read causing the extension is removed. An example of a single contig construction is shown in Fig. 5.3. In this example, the read 5 is made a starter. The extension in each round is shown. During the first round, read number 4 and 6 extend the starter. Similarly in round 2, reads 3 and 7 extend the starter. In round 3, all the reads are used up for extensions and the particular contig is constructed. For this ordering of reads and choice of starter, 3 rounds were required for the construction of contig. We use this idea for constructing intermediate contigs.

To implement this in parallel, we can start by picking a small subset of reads (multiple starters) and start comparing and checking if they can be extended by the reads left in the remaining read set. After single iteration, the starters which did not get extended can be removed and put in intermediate-contigs set, as there is no chance for them to get extended in further iterations. The removed starters are replaced with new reads from the remaining read set for next iteration.

As we do not consider error checking while extending, we call the contigs generated from our approach as "intermediate contigs," which can be further processed by other tools like Velvet. The overall flow diagram is as shown in Fig. 5.4. This approach reduces the size of the input to the Velvet software, as shown in Fig. 5.5, and thereby giving speedups compared to software. There may also be reduction in the RAM usage due to smaller input file.

Velvet is a de Bruijn-based graph assembler. As explained in Sect. 5.1, the k-mers are used to construct the de Bruijn graph. In Fig. 5.5, an example construction of a de Bruijn graph is shown with k-mer size of 5. The top part of the figure shows the velvet flow and the bottom half shows how the same de Bruijn graph is obtained using the FPGA-based approach. The Velvet breaks the reads into their respective k-mers. For example, the first read shown in the figure TCATGTAAG is broken into TCATG, CATGT, ATGTA, TGTAA, and GTAAG. De Bruijn graph is constructed using a combined set of all the k-mers obtained from whole of the read set as shown in the figure. In the FPGA-based approach, the intermediate contigs become the input to

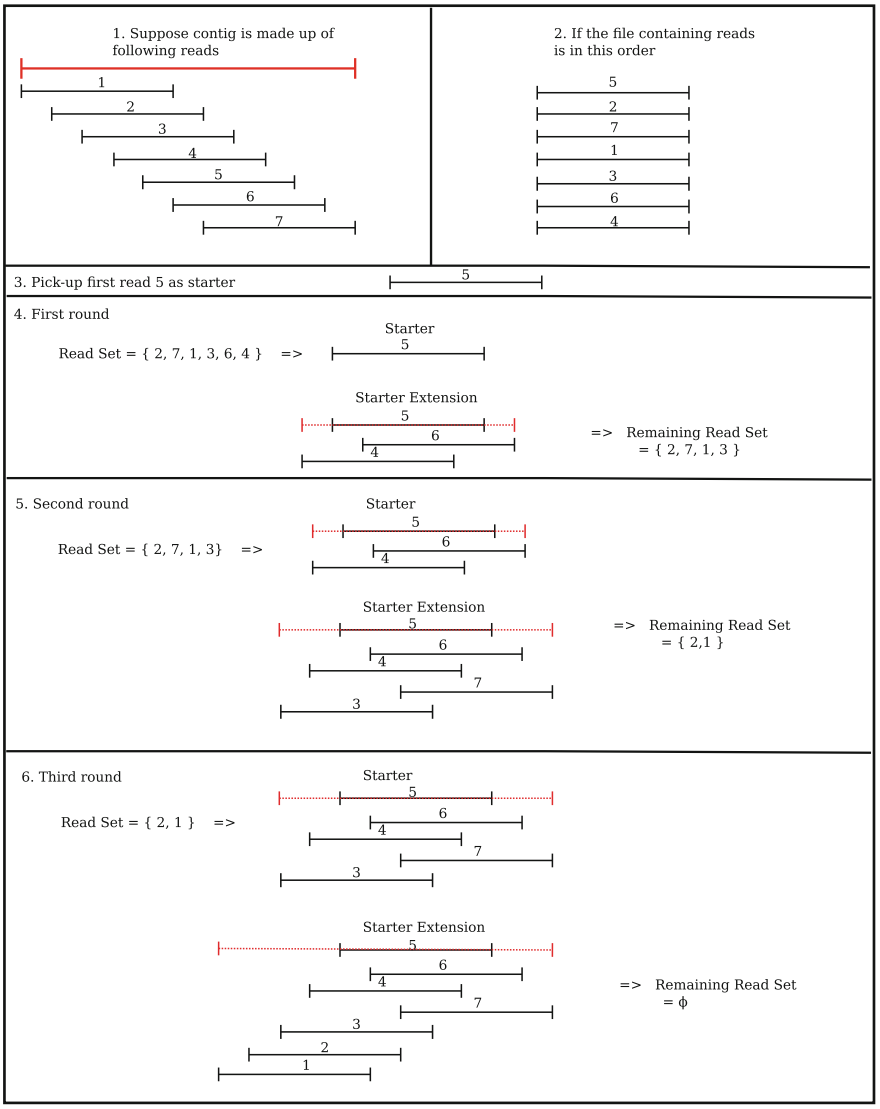

Fig. 5.3 Example showing the construction of contig using our approach. In each round starter is extended. Contig is constructed from the reads in 3 rounds

velvet software which constructs the de Bruijn graph from the k-mers obtained from the intermediate contigs. The figure shows that the same de Bruijn graph is obtained as the end result using both software only approach and FPGA + software-based approach. Due to repeated occurrences of k-mers, the graphs thus constructed using these two methods may not be identical. Quality of assembly has been studied using these two approaches, which is explained in the results section.

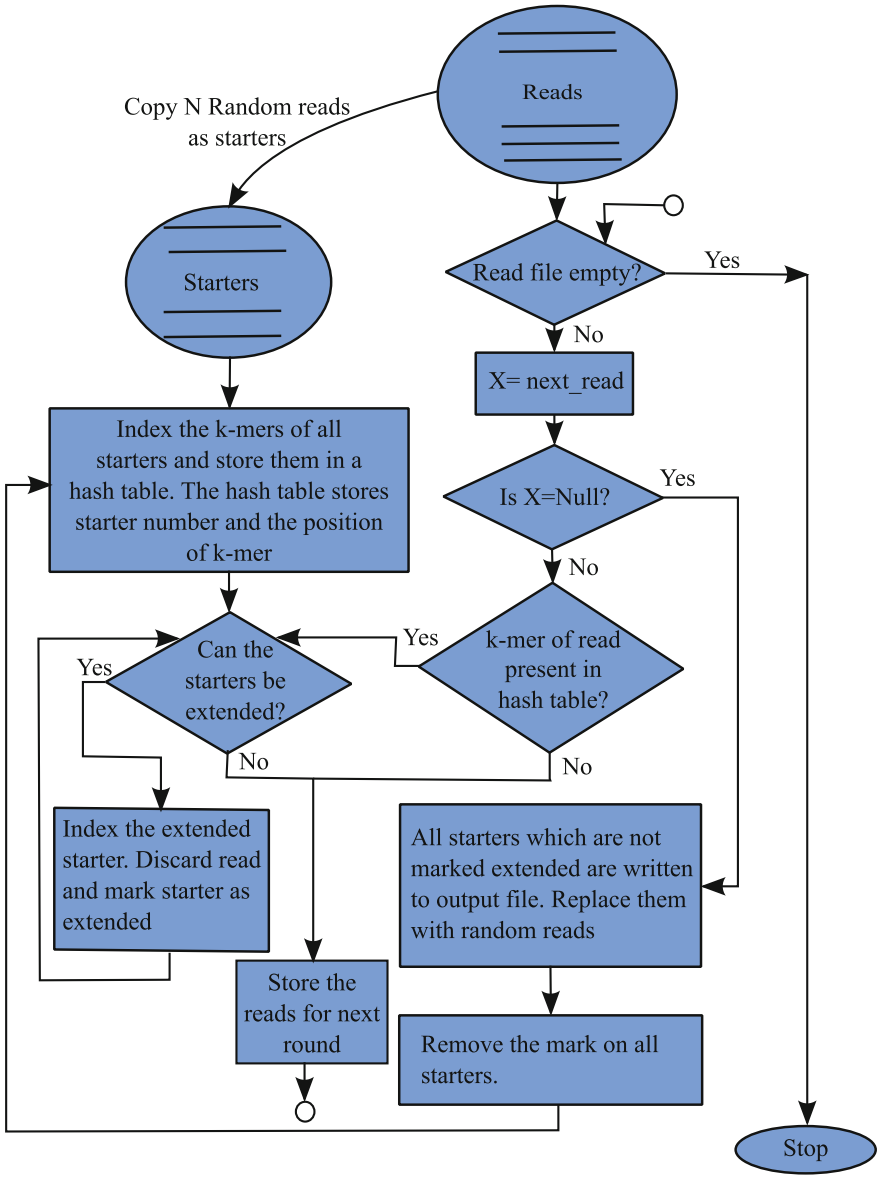

Fig. 5.4 Software flow for estimation. Mapsembler was used for feasibility test and estimation

To study the benefits of our approach, we used an open source software known as Mapsembler [21], which does targeted assembly. It takes NGS raw reads and a set of input sequences (starters). The software determines if the starter is read coherent, i.e., starter is a part of the original sequence. The neighborhood of the starter is given as output if the starter is read coherent.

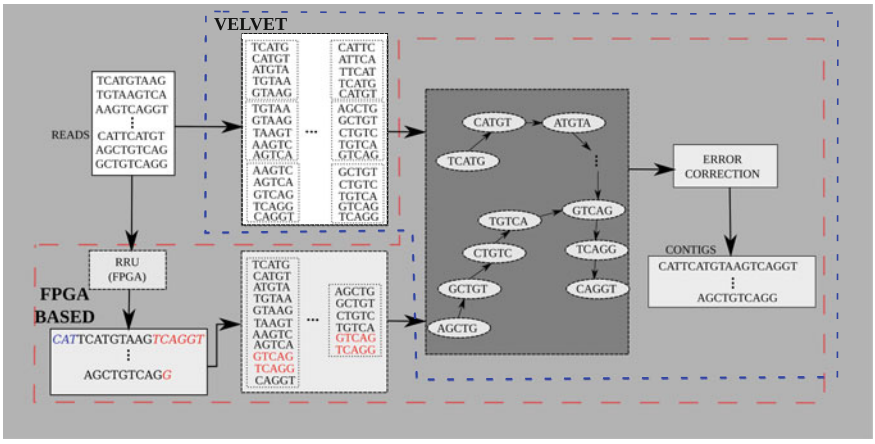

Fig. 5.5 FPGA-based de novo assembly

The algorithm is described in Algorithm 5.1. All the k-mers in all of the starters are indexed and stored in a hash table "I." The hash table consists of starter number and the corresponding position of the k-mer in that particular starter. A read is taken from the NGS read set and the respective k-mers are searched in the hash table. If the k-mer is already hashed, the corresponding starters are tried for extension with the reads.

This high-level model based on Mapsembler was used to study its effectiveness in removing redundancy. This model was also used for conducting experiments with Velvet software on various data sets. From these experiments, we verified that the time taken by Velvet software was dependent on the input file size. Removing the redundancy by our approach did not cause significant loss in the quality of output. We also studied the quality by varying the mismatches allowed during extension. Even though the software model was essentially done to study the initial benefits of our approach, we would also like to mention here that this approach takes very long time to execute, as the reads are compared serially with the starters. In fact, the time taken on a dual core desktop computer is more than the Velvet software time in most of the cases.

5.2.2 *Algorithm to Architecture*

To take advantage of the FPGA architecture we do a streaming design where processing elements are connected in series. Each of the processing elements (PE) stores a sequence called starter. The PEs are connected in a series. "N" starters in the corresponding "N" processing elements are populated with "N" random reads. A read from remaining read set is streamed through the PEs. In each PE, the read is checked

Algorithm 5.1 Redundancy removal using Mapsembler

```
1: procedure RRUMAPSEMBLER(Read-set R)
2:     pick N random reads and store as starter s ∈ S
3:     delete these reads from R
4:     for each starter s ∈ S do
5:         index all k-mers of s in index-table I
6:         starter → extendedFlag = 0
7:     end for
8:     if |R| ≠ ∅ then
9:         for each read r ∈ R do
10:            for each k-mer k in r do
11:                if k indexed in I then
12:                    if r extends corresponding s then
13:                        s = extended(s)
14:                        set starter → extendedFlag = 1
15:                        delete r from R
16:                    end if
17:                end if
18:            end for
19:        end for
20:        for each starter s ∈ S do
21:            if starter → extendedFlag = 0 then
22:                store s as intermediate contig in IC
23:                replace s with random read r ∈ R
24:            else
25:                starter → extendedFlag = 0
26:            end if
27:        end for
28:    end if
29:    store all starters to IC
30:    Return IC
31: end procedure
```

if it can extend the starter. If extended, the starter is updated with the extended starter. This process is continued till all the reads are exhausted. We use the term "***round***" frequently in the rest of the chapters which means that all the reads from the read set are compared once with current set of starters and tried for extensions. After each *round*, the starters which are not extended are replaced with new random reads from the remaining read set. These nonextended starters of the current *round* are stored in an output file. This process removes redundancy in the reads. The redundancy reduction process is repeated several times till there are no more reads. The remaining reads along with all the nonextending starters from previous rounds constitute the *Intermediate Reads*. These intermediate reads are stored in an output file. The intermediate reads are less in number and longer in length. This output file is given as input to Velvet software for removing errors and generating contigs.

In order to get better performance, we do the hardware implementation of the redundancy removal unit using FPGAs. The proposed hardware model differs significantly form the software model. The hash-based searching of k-mers in the

software model is not implemented in this hardware model due to memory resource constraints. The reads are compared with the starter ends and tried for extension. In each cycle, the read is shifted and checked if it can extend the starter.

For example:

$$\begin{cases} Cycle1 \\ Starter - \quad ACTGTCGTGTCTGC \\ Read - \quad\quad TGTCGTGTCTGCGC \\ Cycle4 \\ Starter - \quad ACTGTCGTGTCTGC \\ ShiftedRead - \quad\quad TCGTGTCTGCGCTG \\ Extd.Starter - ACTGTCGTGTCTGCGCTG \end{cases}$$

In this example, the starter gets extended after four cycles, where a perfect match is found after shifting the read by 4 bps.

The extension phase is expensive as it is a long process. To avoid this long delay in the extender, we add a filter which eliminates reads with no probable extensions. The number of cycles needed for extension is equivalent to the difference of read length and k-mer length. From the software implementation, it is observed that for a single round, the number of reads used for extension are very small when compared to reads that extend the starters. To take advantage of this feature, we propose a *prefilter* block. The prefilter is added before the extension phase. The prefilter compares the signature of the reads with the signature of the starter. This signature is called the "*read vector*" and is constructed by encoding the 4-mers in binary format. 4-mer was chosen for signature because the vector width would be 256 corresponding to 4^4. If we choose a signature with more than 4-mer, then the signature will become much more lengthy and hence would require large memory for its storage and larger amount of resources for doing the prefilter.

An example of construction of the read vector is shown in Fig. 5.6. In the example shown, the read vector for read AAAAAAAGGGGG is "100100...001." Each bit represents a particular 4-mer. It is set if the 4-mer exists else it is reset. Only one bit is stored if there is repetition of the 4-mer. The construction of read vectors has to be done only once, as it does not change during the whole process of assembly. We first implemented this in software and found that it was taking significant amount of time and so we implemented this in hardware. The details of this implementation are explained in Sect. 5.3.

The processing element consists of the prefilter and the extender part. Each processing element has to store the starter sequence. As the starter keeps increasing in length, it becomes difficult to store the whole starter inside the processing element, due to limited resources available in the FPGA. In order to alleviate this problem, we decided to store the left end and right end of the starter in the processing element, and reconstruct the starters in the host. The length of the starter ends stored in processing element is equivalent to the read-length, thus allowing extensions at the ends.

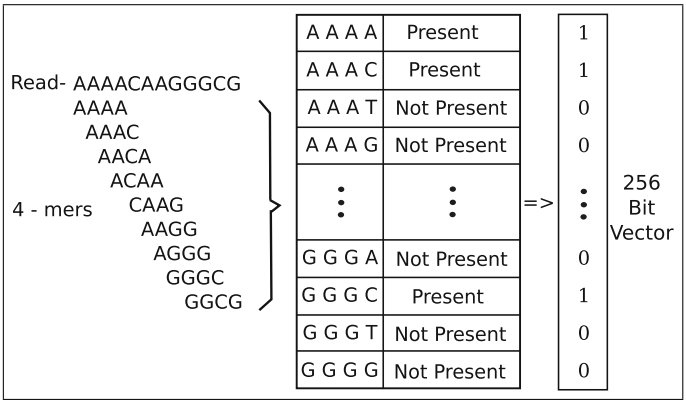

Fig. 5.6 Example showing construction of read vector

With this approach, the reads which are completely covered by the starter and do not extend the starter are not eliminated. We observed that this does not cause significant difference in the speedup. By using this approach, we gain two advantages:

1. The memory resource usage is reduced as we store only the left end and the right end of the starter.
2. We need not reconstruct the vectors for the starter ends used in the prefilter. This is because the extension is caused by the read for which read vector is already available. If starter is extended, the starter end and its corresponding vector are replaced by the read and its vector. This saves considerable amount of time, as construction of vectors for each extension would be very expensive, both in terms of resource usage and execution time.

The clock cycles required by each processing element to process a read varies widely. The number of cycles is highly dependent on the read which is being processed. Due to rejection by the prefilter, there could be lot of data generated in very few cycles for the next processing element, or the next processing element could be waiting for the extension phase of the processing element. Due to this there is irregularity in the time which processing element can start processing the reads. To keep the processing elements busy for most of the time, we have introduced FIFOs in between the processing elements. As the implementation is done on FPGAs, the BRAMs were used for the FIFO implementation for effective resource usage.

5.3 Hardware Implementation

The overall block diagram is shown in Fig. 5.7. It shows the FPGA board interface with the host. We use AlphaData board for hardware implementation [22]. The board has a PCIe bridge which is used for data transmission between host and vice versa.

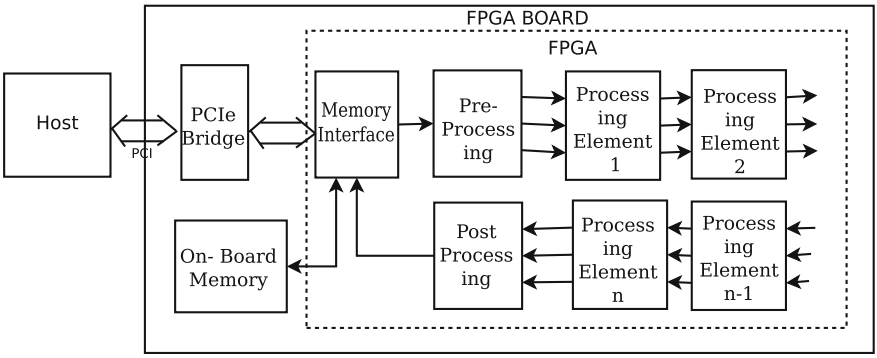

Fig. 5.7 Diagram of FPGA board. The RRU unit is implemented on an FPGA board connected to host through PCIe interface

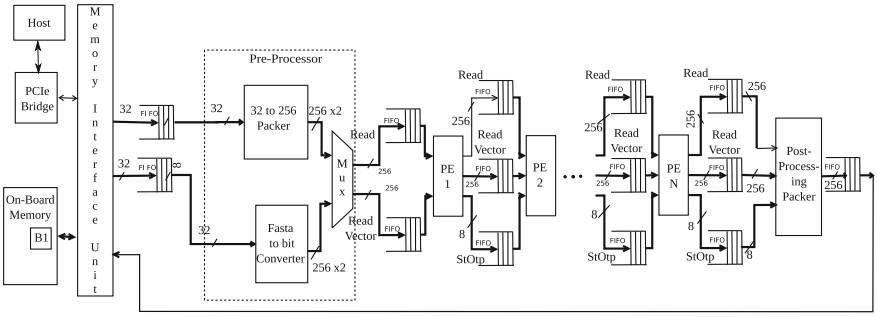

Fig. 5.8 Block diagram of hardware implementation [23]

The Memory interface unit connects the onboard memory and the PCIe bridge. In our design, we use the *Memory Interface* unit to send data to a "*Preprocessor*." From the preprocessor, a series of *Processing Elements* (PEs) are connected through a set of FIFOs. The FIFOs are not shown in Fig. 5.7 for clarity. The last PE is connected to a "*Post Processor*" connected back to memory interface unit. The expanded diagram showing different stages including the FIFOs is shown in Fig. 5.8.

The read set in FASTA input file format is sent from the host to the FPGA board through PCIe bus. For initializing the starters, we encode the most significant three bits of the read. The fourth bit is used for marking the read that it has extended as a starter. In order to reconstruct the starters, the starter and the position of the extension is sent as output through the FIFOSet. Reconstruction of starters is done in software.

5.3.1 FASTA File to Bit File Converter

The *FASTA to bit converter block* reads data from the input buffer and encodes the base pairs in binary format; 'A' as "00," 'C' as "01," 'T' as "10," and 'G' as "11." It also generates the read vector. The block diagram is as shown in Fig. 5.9. The FASTA to bit converter has an input BRAM which stores a part of the FASTA file. This block has two parts: the sequence coder and bit sequence generator which generates the read vector. The sequence coder reads data from the input FIFO and encodes the base pairs in binary format and removes the comments. Sequence coder is implemented as a state machine. The state diagram is shown in Fig. 5.10. The bit sequence generator is made up of a 256-bit shift register, a 256-bit register, and a control unit as shown in Fig. 5.11. The two-bit sequence code from the sequence coder unit is pushed into the shift register by shifting two bits to the left. The first eight bits are taken as an address to set the bit on the second 256-bit register. This register is read after a single read is encoded completely, which is known by the "*seqValid*" signal coming from the sequence coder unit. Control unit controls the shift operation and generates "*FifoWr*" signal for writing the read vector and the read when it is ready.

This binary conversion from ASCII is done only once and the rest of the units use the binary format for further processing. During the next rounds, this state machine remains idle and so "mux" and "control" is used for selecting the required FIFO.

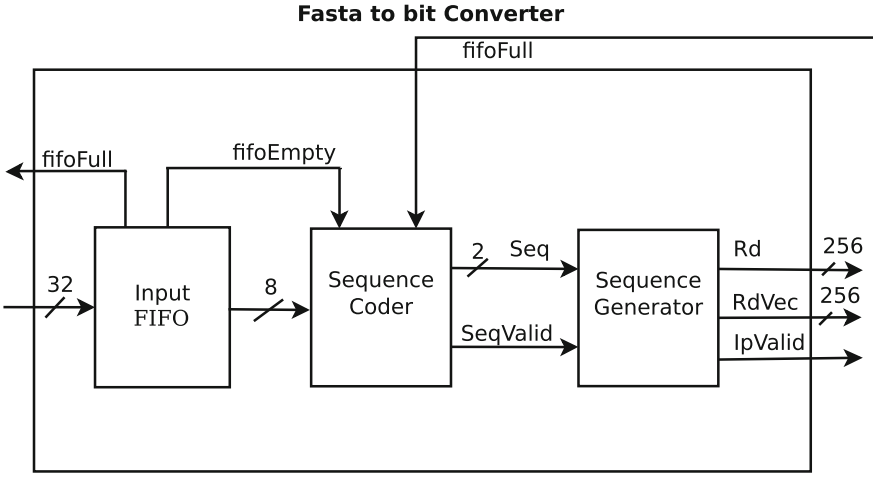

Fig. 5.9 FASTA to bit converter

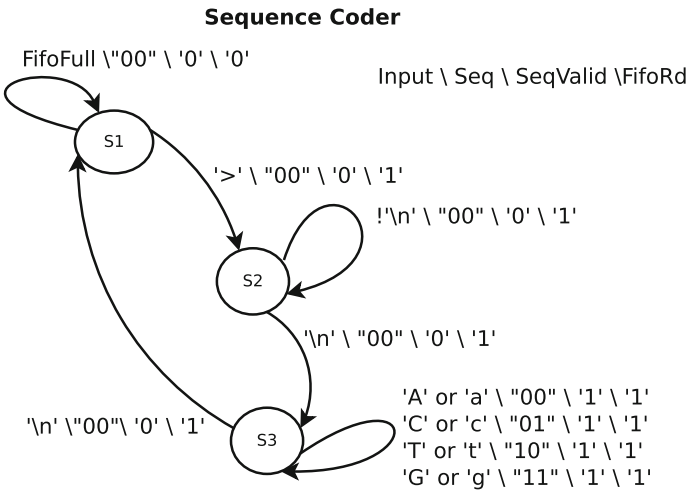

Fig. 5.10 State diagram for sequence coder

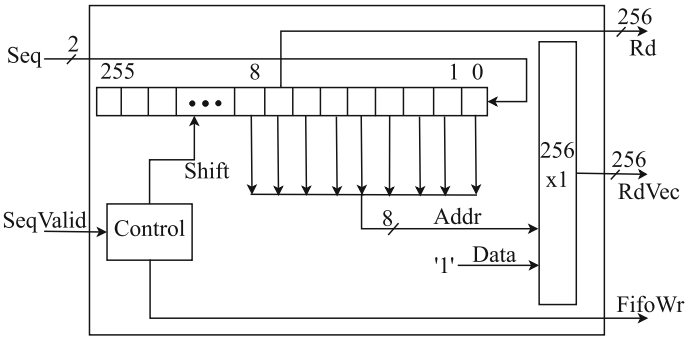

Fig. 5.11 Bit sequence generator

5.3.2 Processing Element Design

The preprocessing block is followed by a series of *Processing Elements* (PE). The PE contains two parts: the *Prefilter* and the *Extender* as shown in Fig. 5.12. A read and corresponding read vector is taken from the remaining read set and compared with the starters for extension. Each processing element stores the information of a single starter. The starter will be saved as an intermediate contig if it does not extend in the current round. Due to memory constraints in the FPGA, we store only the right end and the left end of the string for extension. Each of the starters have two parts: each of length equivalent to the read length. The left end is called *starterLeft* and the right end is called the *starterRight*. First "*n*" reads, considering that the reads

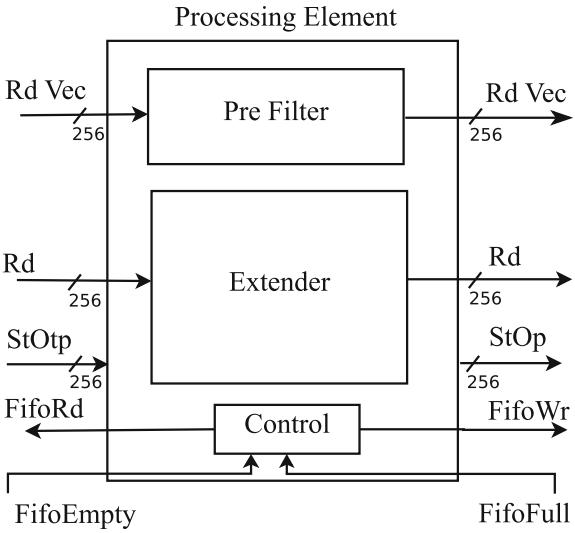

Fig. 5.12 Processing element

are arranged randomly, are copied to starterLeft and starterRight. Similarly, we store *starterVectorRight* and *starterVectorLeft* used for prefiltering.

5.3.3 Prefilter Design

The prefilter design is shown in Fig. 5.13. In the prefilter, a logical "*AND*" is done between read vector and starterVectorLeft and stored in TempL register. Similarly, logical "*AND*" is done between read vector and starVectorRight and stored in TempR register. The one-counter block counts the number of '1's in the TempL and TempR register. If the number of '1's is greater than a set threshold, the read is passed to the extender; else, it is passed to the FIFO for the next processing element to evaluate it. The one-counter block lies in the critical path and hence defines the clock period of operation. We implemented two versions of the one-counter block: one using Wallace tree and the second using the carry chain available in the Xilinx FPGA slices. The Wallace tree is built out of 6:3 compressors as the Xilinx Virtex 6 FPGA has six input LUTs. For the 256-bit implementation, we need a Wallace tree of 258 bits using the 6:3 compressors. The tree is built in five stages and requires 100 $(43 + 24 + 14 + 10 + 9)$ 6:3 compressors with a carry adder at the final stage. The second implementation is done by adding each bit using the carry chain available in the FPGA. This implementation took less time, but slightly more area when compared to the Wallace tree implementation.

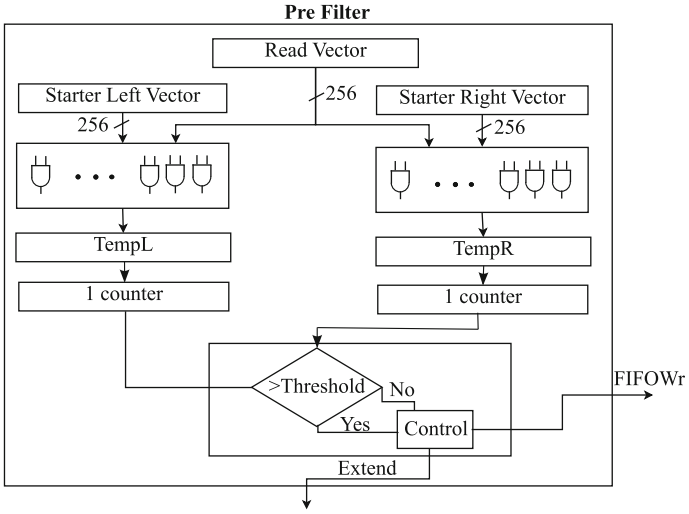

Fig. 5.13 Prefilter design

5.3.4 Extender Design

The extender design is shown in Fig. 5.14. We use a "maskL" and "maskR" register for masking the corresponding bits in the starter after shifting the read. In the beginning, these registers are initialized with twice read length on the right with '1's and rest of the bits are set to '0." The following operations are done in the extender:

$$\begin{cases} tempL = (starterLeft\,AND\,maskL)\,XNOR\,shiftedRdL \\ tempR = (starterRight\,AND\,maskR)\,XNOR\,shiftedRdR. \end{cases}$$

The corresponding scores, scoreL and scoreR, are calculated from tempL and tempR, respectively, using modified one-counter. A modified one-counter is needed as we are encoding the base-pair in two bits. An example is shown in Fig. 5.15. Here, we see that the total score should be calculated by checking 2 consecutive 1s. For calculating this, we modify the Wallace tree implemented in prefilter block. The Wallace tree consists of compressors and since FPGA has 6 input LUTs, we used 6:3 compressors. The LUTs store the corresponding outputs. The interconnect tree is built to build the adder.

In our case, since we need to measure only consecutive 1s, we store the appropriate values in the LUTs of the first stage of the Wallace tree implementation instead of the sum. For example, we store output as 011 (decimal 3) instead of 100 (decimal 8) for 11 11 11 in the 6:3 compressor LUT. This says that three matches are found. Similarly, for 01 10 11 output stored in the compressor LUT is 001 and 10 for 11 11 01. So, the sum (score) gives the exact matches of the two sequences.

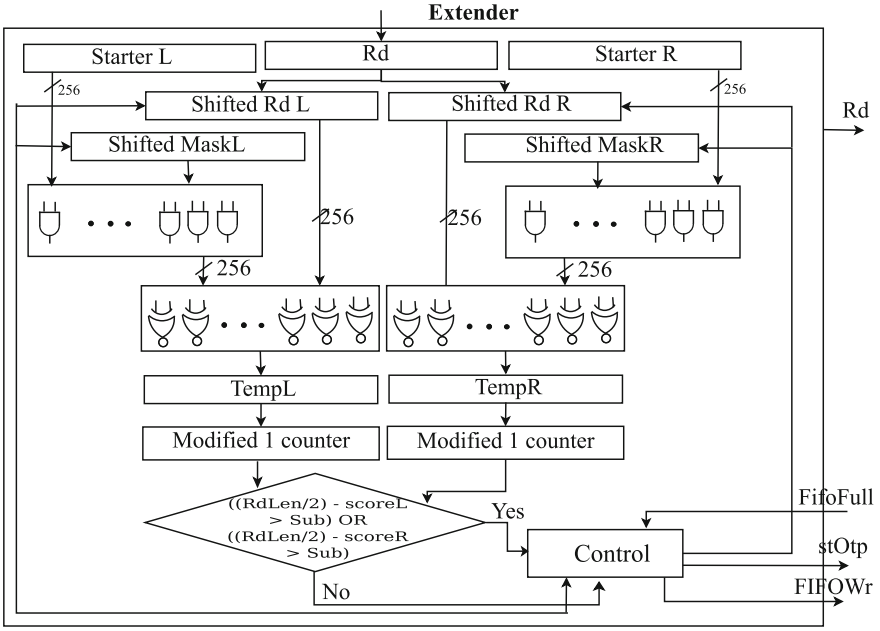

Fig. 5.14 Extender design

```
                    TCGTGTCGTCTTG          -- Starter
                      CGTCGTTGTACTA        -- Shifted Read
00 00 00 00 00 00 11 11 11 11 11 11 11     -- MASK
                      AND
 10 01 11 10 11 10 01 11 10 01 10 10 11    --Starter
-------------------------------------------------------------
 00 00 00 00 00 00 01 11 10 01 10 10 11    --temp
                    X-NOR
00 00 00 00 00 00 01 11 10 01 11 10 10 11 10 00 01 10 00
                                           --shifted read
-------------------------------------------------------------
11 11 11 11 11 11 11 11 11 11 10 11 10     --tempScore
```

Fig. 5.15 Example of score calculation in extender. Here score = 11

if((readLength/2) − scoreL) is less than allowed substitutions, the starter is extended on the left side or if((readLength/2) − scoreR) is less than allowed substitutions, the starter is extended on the right side. The starterLeft and starterVectorLeft are replaced by the read and read vector, respectively, if the starter is extended on the left side. Similarly, starterRight and starterVectorRight are replaced by the read and read vector respectively, if the starter gets extended on the right side. This step of counting the number of '1's can be eliminated if the allowed substitution is set to zero. For this, only an "*XOR*" operation of between (shifted read "*AND*" with mask) and starter has to be done and checked if it is equivalent to zero for extension. This

saves lots of resources and also from the experiments conducted we found that the quality of results is better with threshold set to "0." When the threshold is zero, the implementation does not need a comparator and hence the resources occupied were reduced by a considerable amount. If a starter is not extended on either of the two sides, then shiftedReadL and shiftedReadR are shifted to right and left, respectively, by two bits as base-pair is encoded with two bits.

The maskL and maskR are changed as follows:

$$\begin{cases} maskL = maskL\, AND\, (maskL \ll 2) \\ maskR = maskR \gg 2. \end{cases}$$

This process is repeated till the read is shifted (readLength $-$ k-mer length), as we do not extend starter if there are less than "k" matches. The reads that do not extend any of the starters are put in the next FIFOSet for further processing by the next PEs.

5.4 Results and Discussion

Zhang et al. [20] have done a comparison of de novo assembly software approaches. The authors have provided scripts for generating the read set from the genome. We used these scripts to generate the read files for *E. coli*, swinepox, and human influenza. For evaluating our approach, we generated the single-ended read set with 100× coverage for read-length 36 and 75 and 1 % error rate similar to what was reported by Zhang et al. [20]. For the software only flow time, Velvet software was run using the read set directly on a desktop computer with Intel (R) Core (TM) 2 Duo CPU E4700 running at 2.60 GHz with 4 GB RAM.

5.4.1 Resource Utilization and Operating Frequency

We have implemented the RRU on FPGA and obtained the clock period and FPGA resource utilization after running place and route tools provided by Xilinx ISE 14.1 [24]. We use these parameters to estimate the speedups for running the Velvet on the output of RRU after each round. From place and route tools, the maximum clock frequency for the whole of the design was found to be 200 MHz. We get better performance by using multiple clocks. The sequence coder and generate units were able to run at a maximum frequency of 350 MHz on Virtex-6 FPGA. The maximum frequency of operation for the rest of the units was 200 MHz. The hardware implementation was done on Alpha-Data board having Xilinx Virtex-6 (XC6VSX475T) FPGA with speed-grade 1.

Table 5.1 Resource utilization

Component	Slices	BRAM
preprocessor	559	–
In-FIFO	17	32
FIFO-set	244	8
Post-processor	200	–
Out-FIFO	93	15
PCI-interface	2047	15
Others	35	–
PE-RdLen-36	380	–
Threshold-0		
PE-RdLen-36	939	–
Theshold-11		
PE-RdLen-75	643	–
Theshold-0		
PE-RdLen-75	1175	–
Theshold-11		

The resource utilization obtained from ISE module level utilization, for the different units is shown in Table 5.1. Here we considered design which does not allow any substitution. The resources occupied by processing element varies depending on the read length. When the threshold value of the prefilter is set to "0," the one-counter is removed and thus number of slices occupied is significantly reduced. Table 5.2 shows the number of PEs that could be implemented on Xilinx Virtex-6 (XC6VSX475T) FPGA which has 74,400 slices. This table also shows the estimates of number of PEs on a larger Xilinx Virtex-7 (XC7V2000T) device which has 305,400 slices, considering 97 % resource utilization. This is actually an underestimate as many slices from various units get combined during the synthesis flow and more logic can be realized on the device. The threshold in this table also refers to the prefilter threshold.

Table 5.2 Number of PEs on Xilinx devices

Xilinx Device	XC6VSX475T	XC7V2000T
PE-RdLen-36	110	467
Threshold-0		
PE-RdLen-36	58	247
Threshold-11		
PE-RdLen-75	78	330
Threshold-0		
PE-RdLen-75	48	206
Threshold-11		

If this prefilter threshold is zero, the resource usage is less and hence the number of PEs which can be implemented on the device increases. We did experiments and found that for swinepox, threshold value of 11 gives better reduction. If the threshold is high many of the reads that can be extended are not sent to the extender and if reduction is low, many of the reads that do not extend the starter are passed to the extender. The maximum number of cycles a read spends in a processing element is equivalent to the length of the read, assuming it has passed the prefilter stage. Using this parameter, we chose the FIFO depth to be same as the length of the read. Most of the time, two consecutive reads will not extend the same starter and hence the PEs are busy most of the time. This was verified using profiling.

Note that for design with larger number of PEs where multiple FPGAs would be required, we have not considered inter FPGA transfer time in our estimates. We assume FPGAs are connected in series and the data is streamed from the host, through the FPGAs and finally, back to the host.

5.4.2 Speedups over Software

Figure 5.16 shows the graphs of the speedups at different rounds for swinepox with read length 36 using PEs varying from 30 to 3000 and using Velvet software [23].

The graphs show maximum speedups in the range of 5.2× to 11.9× for swinepox over Velvet software. We also observe that the speedups reach a maximum and then start to decrease with increasing number of processing elements. The reason for this is that the utilization of the PEs goes down after a peak and hence the time for read and write cycles dominate.

We have considered the worst case time by setting the threshold value for the prefilter to be zero. Figure 5.17 shows the reduction in base pairs after each round for swinepox genome. It can be observed that the compression saturates after certain number of rounds. The rate of reduction is higher when more number of PEs are used since compression is being done at a faster rate.

The reduction in size of the input file in terms of base pairs to Velvet software is shown in Fig. 5.18. For a larger genome like *E. coli*, we need to have more processing elements to get significant speedups. The maximum speedups are tabulated in Table 5.3.

The speedups in each case first increases, reaches a peak, and then tapers off. The initial increase can be attributed to the high reduction of input file size during the initial rounds. After these initial rounds, the redundancy removal is more limited and so the time taken by Velvet is almost constant. The FPGA processing time is incremental in nature and hence goes on increasing after each round. Even though there is not much redundancy removal during the later rounds, the hardware unit takes at least as many cycles as the number of reads and writes in each PE.

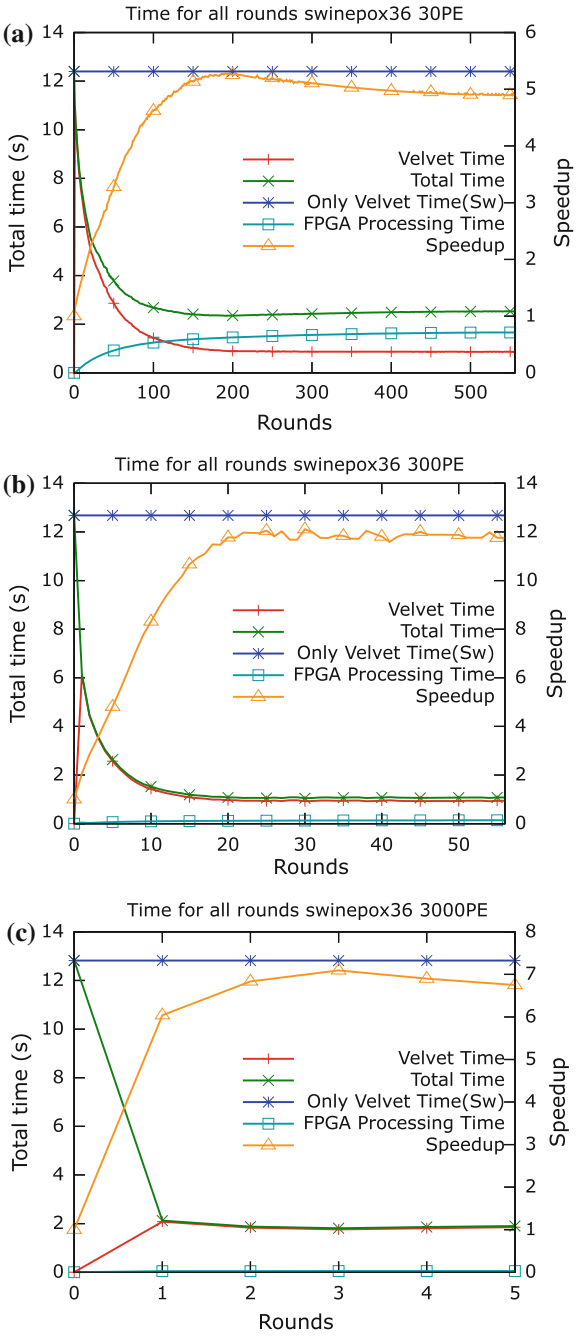

Fig. 5.16 Speedups over Velvet software after each round for swinepox read-length 36. **a** Speedups 30 PE. **b** Speedups 300 PE. **c** Speedups 3000 PE

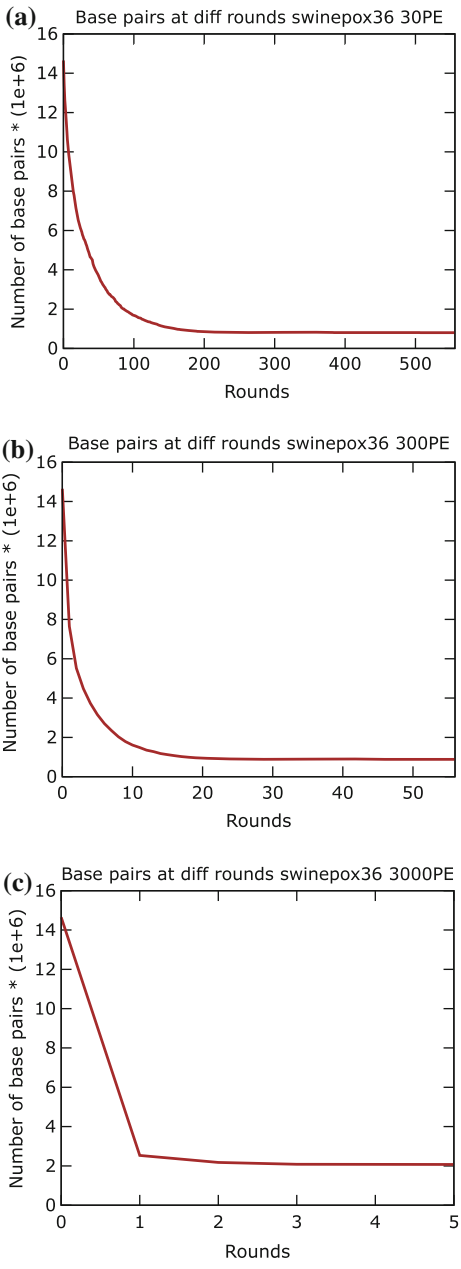

Fig. 5.17 Number of base pairs after each round for swinepox read-length 36. **a** Input Size 30 PE. **b** Input Size 300 PE. **c** Input Size 3000 PE

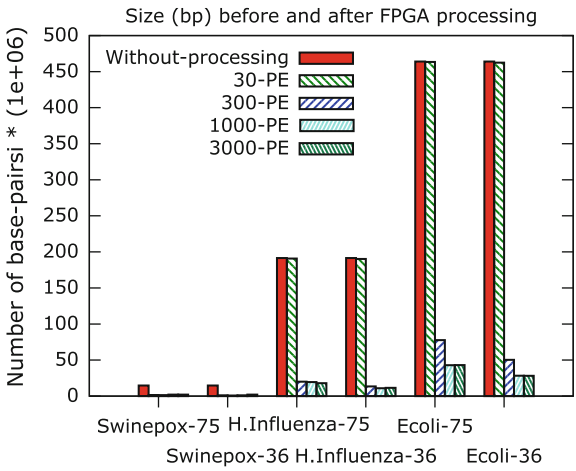

Fig. 5.18 Input sizes (bp) before and after FPGA processing with threshold 0

Table 5.3 Maximum speedups over Velvet software

Sample	Swinepox	Swinepox	H. Influenza	H. Influenza	*E. coli*	*E. coli*
Read-length	75	36	75	36	75	36
PE\Size	30.8 MB	48.4 MB	449 MB	729.7 MB	1.2 GB	2.1 GB
30 PE	3.5×	5.2×	1.1×	1.09×	1.2×	1.09×
300 PE	13×	11.9×	3.2×	3.6×	2.5×	2.1×
1000 PE	6.5×	10.5×	6.8×	6.0×	4.4×	5.02×
3000 PE	4.8×	7×	6.8×	10×	6.5×	9.2×

From these results, we can determine a termination criterion to get maximum benefits from FPGA processing. For this, we keep track of reduction in the total size of data set in each round and if this decrease is too little (less than threshold), we stop further rounds.

5.4.3 Quality

There are many factors which affect the quality of assembly. The quality is dependent on the sequencing machine. After the sequencing, the quality of assembly is dependent on many parameters that are used in the assembly algorithms. Mostly, the input parameters to the assembler like k-mer length and number of mismatches allowed can significantly affect the quality of the output. The most popular metrics to measure quality are the maximum length of the contigs and the "***N50***". N50 is the minimum length of the contig such that summing up the length of only those contigs whose length is more than N50 cover 50 % of the genome.

Table 5.4 Quality of assembly

Sample	Swinepox 75		*E. coli* 36	
Parameters	N-50	Max contig	N-50	Max contig
FPGA based\Velvet	119,046	119,046	14,988	100,485
30 PE	119,046	119,046	14,988	100,485
300 PE	102,563	102,566	14,988	100,485
1000 PE	119,046	119,046	15,344	100,485
3000 PE	119,046	119,046	15,351	100,485

The quality of Velvet output using these metrics for different PEs is tabulated in Table 5.4. From the various experiments conducted, we observed that by not allowing mismatches during extension, there was no (significant) loss in quality of output as shown in results.

From the results, we find that the speedup is dependent on the nature and size of input data. For a fixed number of PEs, the speedup first increases and then tapers down with larger number of rounds as FPGA processing time starts dominating. Maximum speedup increases with number of PEs and reduces after reaching peak. We estimate speedups up to 13× using our hybrid approach.

From the results it can be seen that the key factor limiting speedup is the number of PEs in an FPGA. For larger genomes, more number of PEs will be useful in reducing the overall execution time. Since rate of compression per round increases with more number of PEs, this would directly influence the overall speedups. Since we have done an efficient hardware implementation, the number of PEs can be increased by using multi-FPGA boards. The results using multi-FPGA boards for further reducing the execution time are included in Chap. 7.

5.5 Summary

In this chapter, we studied the FPGA acceleration of de novo genome assembly. We accelerate the application by doing preprocessing of the input using FPGAs. The speedups are obtained by reduction of the input to the Velvet software. The intermediate contigs were generated in hardware which was given to software to remove the errors and construct the contigs. The algorithm modification was done to do an efficient implementation in FPGAs. Prefilters were used to decrease the execution time. The effect of the algorithm modification on the quality of output is analyzed. In the next chapter, we discuss how we adopted our methodology to further accelerate protein docking and genome assembly using FPGAs with custom-designed HEBs.

References

1. Huang, X., Wang, J., Aluru, S., Yang, S.P., Hillier, L.: PCAP: a whole-genome assembly program. Genome Res. **13**(9), 2164–2170 (2003)
2. Wu, X.L., Heo, Y., El Hajj, I., Hwu, W.M., Chen, D., Ma, J.: TIGER: tiled iterative genome assembler. BMC Bioinf. **13** (2012)
3. Hernandez, D., François, P., Farinelli, L., Østerås, M., Schrenzel, J.: De novo bacterial genome sequencing: millions of very short reads assembled on a desktop computer. Genome Res. **18** (2008)
4. Miller, J.R., Delcher, A.L., Koren, S., Venter, E., Walenz, B.P., Brownley, A., Johnson, J., Li, K., Mobarry, C., Sutton, G.: Aggressive assembly of pyrosequencing reads with mates. Bioinformatics **24**(24), 2818–2824 (2008)
5. Liu, Y., Schmidt, B., Maskell, D.: Parallelized short read assembly of large genomes using de bruijn graphs. BMC Bioinf. **12**(1), 354–363 (2011)
6. Zerbino, D.R., Birney, E.: Velvet: algorithms for de novo short read assembly using de Bruijn graphs. Genome Res. **18**(5), 821–829 (2008)
7. Pevzner, P.A., Tang, H., Waterman, M.S.: An Eulerian path approach to DNA fragment assembly. Proc. Nat. Acad. Sci. **98**(17), 9748–9753 (2001)
8. Koren, S., Schatz, M.C., Walenz, B.P., Martin, J., Howard, J.T., Ganapathy, G., Wang, Z., Rasko, D.A., McCombie, W.R., Jarvis, E.D., ED., J., Phillippy, A.M.: Hybrid error correction and de novo assembly of single-molecule sequencing reads. Nat. Biotechnol. **30**(7), 693–700
9. Salmela, L., Schröder, J.: Correcting errors in short reads by multiple alignments. Bioinformatics **27**(11), 1455–1461 (2011)
10. Compeau, P.E.C., Pevzner, P.A., Tesler, G.: How to apply de Bruijn graphs to genome assembly. Nat. Biotechnol. **29**, 987–991
11. Tang, W., Wang, W., Duan, B., Zhang, C., Tan, G., Zhang, P., Sun, N.: Accelerating millions of short reads mapping on a heterogeneous architecture with FPGA accelerator. In: Annual IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 184–187 (2012)
12. Chen, Y., Souaiaia, T., Chen, T.: PerM: efficient mapping of short sequencing reads with periodic full sensitive spaced seeds. Bioinformatics **25**(19), 2514–2521 (2009)
13. Olson, C., Kim, M., Clauson, C., Kogon, B., Ebeling, C., Hauck, S., Ruzzo, W.: Hardware acceleration of short read mapping. In: IEEE Symposium on FCCM, pp. 161–168 (2012)
14. Homer, N., Merriman, B., Nelson, S.F.: BFAST: an alignment tool for large scale genome resequencing. PLoS ONE 4 (2009)
15. Langmead, B., Trapnell, C., Pop, M., Salzberg, S.: Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biol. **10** (2009)
16. Fernandez, E., Najjar, W., Harris, E., Lonardi, S.: Exploration of short reads genome mapping in hardware. In: International Conference on FPL, pp. 360–363 (2010)
17. Knodel, O., Preusser, T., Spallek, R.: Next-generation massively parallel short-read mapping on FPGAs. In: IEEE International Conference on ASAP, pp. 195–201 (2011)
18. Convey Computer: Convey Graph Constructor. http://www.conveycomputer.com (2015)
19. Lander, E.S., Waterman, M.S.: Genomic mapping by fingerprinting random clones: a mathematical analysis. Genomics **2**(3), 231–239 (1988)
20. Zhang, W., Chen, J., Yang, Y., Tang, Y., Shang, J., Shen, B.: A practical comparison of de novo genome assembly software tools for next-generation sequencing technologies. PLoS ONE **6**(3) (2011)
21. Peterlongo, P., Chikhi, R.: Mapsembler, targeted and micro assembly of large NGS datasets on a desktop computer. BMC Bioinf. **13**(1) (2012)
22. Alpha-Data: Alpha-Data FPGA Boards. http://www.alpha-data.com/ (2015)
23. Varma, B.S.C. Paul, K., Balakrishnan, M., Lavenier, D.: FAssem: FPGA based acceleration of de novo genome assembly. In: 2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 173–176 (2013)
24. Xilinx: Xilinx FPGAs, ISE. http://www.xilinx.com (2015)

Chapter 6
Design of Accelerators with Hard Embedded Blocks

Abstract In modern FPGAs, the digital circuitry common to many applications are being embedded as hard embedded blocks to utilize silicon area efficiently. Use of HEBs improves the overall performance of hardware implemented using FPGA, as the interconnect delays are also reduced. In this chapter, we discuss how HEBs are designed using the methodology described in Chap. 3. We study the impact of HEBs on the execution time of the two bioinformatics applications; Protein docking and Genome assembly. In Chaps. 4 and 5, we had studied the acceleration of these applications using currently available FPGAs. In this chapter, we discuss the identification and design of respective HEBs to get performance benefits. We also show how we can estimate application speedups using these future FPGA fabrics incorporating these HEBs.

6.1 Acceleration of FTDock Using Hard Embedded Blocks

The FTDock application is a docking application which we intend to accelerate using HEBs in FPGAs. As discussed in Chap. 3, by profiling the application it was found that the most time consuming kernel in the application is the three-dimensional Fast Fourier Transform (3D-FFT) calculation. The 3D-FFT is also popularly used in many other scientific applications in various domains like image processing, bioinformatics, and molecular dynamics. Typically, 3D-FFT computation takes significant part of the execution time in many of these applications. Thus, in order to speedup these applications, it becomes necessary to accelerate 3D-FFT computation. In this section, we discuss acceleration of 3D-FFT using hard embedded blocks, which in turn can be used to accelerate the docking application.

6.1.1 Related Work

Roesler and Nelson [13] and Beauchamp et al. [2] have shown area and performance improvement over fine-grained implementations by including coarse-grained floating

B.S.C. Varma et al., *Architecture Exploration of FPGA Based Accelerators for BioInformatics Applications*, Springer Series in Advanced Microelectronics 55, DOI 10.1007/978-981-10-0591-6_6

point units in FPGAs. Roesler et al. [13] show 2.2 GFlops can be achieved by implementing embedded floating point units in Xilinx XC2V6000 FPGA. Beauchamp et al. [2] use VPR tools to evaluate the floating point unit. The authors also propose embedded shifters which are used for doing IEEE-compliant floating point multiplications. They report 54.2 % reduction in area and 33.4 % increase in speedups when compared to 18×18 multipliers which are already available as HEBs.

Flexible embedded floating point unit is proposed by Chong et al. [3]. The authors report a configurable floating point HEB, which can be used as single precision or double precision based on the need. Their idea is inspired by shadow circuits by Jamieson et al. [10]. Since configurable HEBs provide more usability, they can be used for variety of applications. All these approaches have considered floating point unit implemented as hard embedded blocks for better performance. They project the speedup by mapping some benchmark circuits on the proposed new FPGA containing these HEBs.

Ho et al. [8] report the impact of adding floating point coarse-grained core in an FPGA. They show 2.5× improvement in operating frequency and 18× benefits in area. The coarse-grained block consists of floating point multipliers, adders, and word-blocks. The bus-based interconnect is used to connect these blocks. The bus can be connected to the fine-grained blocks in the FPGA. The reported parameters that were varied were the number of floating point units (both adders and multipliers), number of input-output buses, number of feedback paths, bus width, and the number of such coarse-grained blocks. These parameters were used for DSE. Based on the applications, these parameters were varied to get an optimal design. The ASIC synthesis of these blocks is done using Synopsys design compiler using standard cell libraries. They use VEB method to evaluate the HEBs. They evaluate both single- and double-precision floating point units as HEBs.

Yu et al. [20] try to identify best combination of floating point adders, multipliers, and word blocks for embedding them onto FPGAs based on performance and area. Common subcircuits needed for different applications are implemented as HEBs. They also evaluate merging of floating point units to get performance benefits. They employ multiple types of floating point units by forming combinations based on application characteristics and embed them in FPGAs. They discuss the impact of routing resources for large HEBs.

We also evaluate performance of hybrid FPGAs but our main goal is to evaluate a specific HEB as hardware accelerator.

6.1.2 FPGA Resource Mapping

We use the Algorithm 4.1 described in Chap. 4 which is based on computing 1D-FFT to calculate the 3D-FFT. The 1D-FFT is calculated on each of the three dimensions to get the result.

The input/output controller controls the input and sends the required input data to a particular floating point FFT core. The FFT values which are computed are transferred

back to CPU. The same process of transferring data from the CPU to FPGA and transmitting data back to CPU after computation can be used for calculating IFFTs.

3D-FFT requires n^3 size 3D matrix as input and generates a 3D matrix as an output. Single-precision floating point arithmetic is used and each data of matrix is represented by 32 bits (4 Bytes). As it is not possible to store the whole 3D matrix inside the FPGA for large n (e.g., n = 256, a total of 256^3 * 4 Bytes = 67108864 Bytes), we have designed a data usage model that is described now.

CPU transfers the required data for computation to the DRAM. The data required for calculating 1D-FFT is copied to BRAMs within the FPGA. Multiple cores compute single-dimension FFT on the FPGA concurrently. The computing cores start computing as soon as BRAMs receive their respective data in full. The data output is transmitted back to the DRAM as soon as the computation is over.

The timing diagram to calculate the total time for computing the 3D-FFT is shown in Fig. 6.1. The total time for execution of one n-point 3D-FFT using two-port DRAM is given by (6.2), where Tb is the time taken to transfer required data for one 1D-FFT from external memory to BRAM and Tc is the computation time of 1D-FFT. For our HEB, Tc is given by (6.1), where t is the clock period of the butterfly.

$$\mathrm{Tc} = t(n/2 * \log_2 n + 7) \tag{6.1}$$

The total time for 3D-FFT thus obtained is divided by the number of banks on the onboard memory to get actual computation time, as this unit can be replicated for each bank of onboard memory.

$$\mathrm{T_{total}} = n * n * 3 * \mathrm{Tb} + \mathrm{Tc} + \mathrm{Tb} \tag{6.2}$$

Based on (6.2), we estimate the time for calculating N point 3D-FFT for different FPGA devices with varying resources. This is compared with the time taken for software executed on desktop PC with Intel Core 2 duo E4700, 2.6 GHz processor with 4 GB RAM.

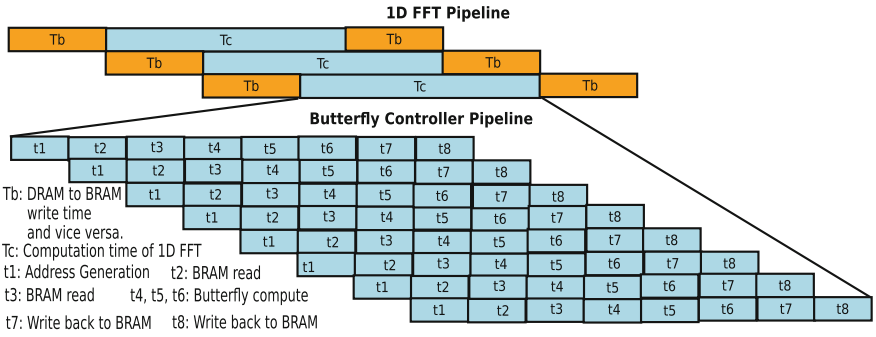

Fig. 6.1 Timing diagram for 3D-FFT

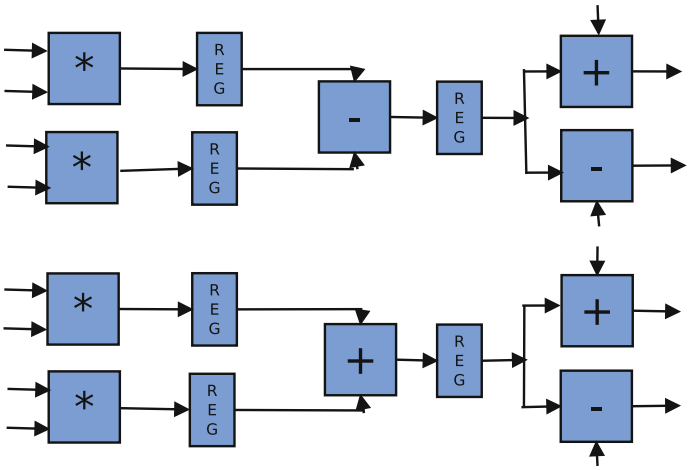

Fig. 6.2 Butterfly design

6.1.3 Hard Embedded Block Design Space Exploration

Methodologies to evaluate heterogeneous embedded blocks in FPGAs have been reported by two groups as explained in Chap. 3. We used VEB methodology as it allows using proprietary tools for evaluating the new custom-designed fabrics with existing FPGAs. The HEB design and the application design involving FPGAs incorporating the HEBs is discussed in the following subsections.

6.1.3.1 Hard Embedded Block Design

Usually, many applications involve calculation of FFTs of different lengths. As butterfly is a common module for FFT of any size, we decided to make the butterfly computation as a HEB to exploit its generic use. The butterfly design is as shown in Fig. 6.2. For implementing the butterfly, unlike other implementations reported in Sect. 6.1.1, we consider the single-precision floating point units in pipelined manner. The butterfly HEB takes three clock cycles to complete the operation.

Flachs et al. [4] have implemented a floating point arithmetic unit used in their processor. From their implementation, the maximum clock period for one FPU is 800 MHz. According to ITRS [9], the area of design is halved when a feature size is scaled by 0.7×. As we do not change the frequency of operation of FPU, we divide the area by a factor of 4 for scaling the FPU from 90 to 40 nm. We use their model to estimate the frequency and the area of the butterfly unit as this is one of the good FPU implementations which is highly optimized for processor design. We estimate that the area occupied by one butterfly HEB unit is 0.815 mm^2.

For VEB methodology, we need to estimate the area of the HEB in terms of slices. Nachiket [11] has estimated that the area of Virtex-6 LX760 FPGA having 118,560 slices is 471 mm^2. The author uses LUT area to estimate the area occupied by slices. Using this area estimates, we find that the area occupied by one butterfly HEB is equivalent in area to 210 Virtex-6 slices. Such an equivalence is useful in performing the trade-off and configurable logic bocks.

6.1.3.2 Application Design Using Hard Embedded Blocks

The FFT is implemented using the butterfly HEB. 1D-FFT design is as shown in Fig. 6.3. The FFT consists of two dual-port memories connected to the butterfly. The controller is implemented in a pipelined manner with eight stages. The first stage is the address generation stage where the BRAM read address is generated. The next two stages are the read stages where data from BRAM is read. The next three stages are computation stage followed by two write stages where the computed data is written back to the BRAM.

Ideally, for fast operation, FFT computation using butterfly requires a quad-port memory. Since implementation of quad-port memory using distributed RAM is not area efficient, we decided to use dual-port memory to store the data. This dual-port memory was implemented using BRAMs and was designed using logic core provided by Xilinx ISE. In order to compute FFT using dual-port memories, two cycles are needed for both read as well as write operations. We designed the controller to generate the required addresses for computing the FFT. Simulation was done using Modelsim and synthesis and place and route was done using XST to obtain the minimum clock period. The various parameters like area and the clock period of

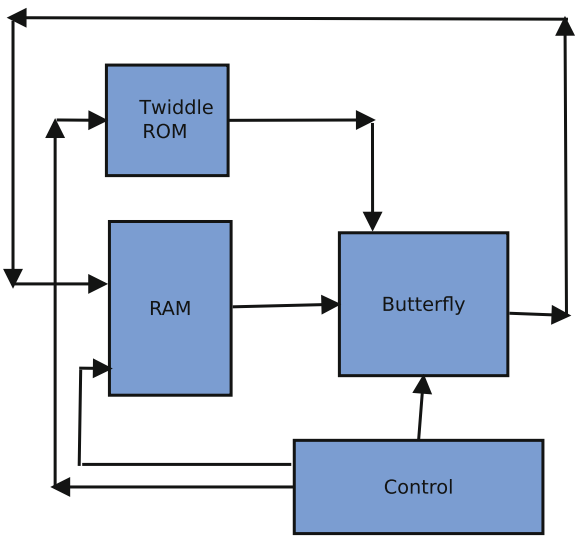

Fig. 6.3 1D-FFT design

core using butterfly HEB when compared with Xilinx core are reported in Table 6.1. We can see that the FFT with butterfly HEB occupies lesser area compared to the Xilinx core which uses DSP HEB. The number of cycles taken by FFT using butterfly HEB is less compared to one with Xilinx core. This is in spite of Xilinx core being a highly optimized implementation.

6.1.4 Results and Discussion

We compare our HEB with the implementations using existing HEB in the devices. We use Xilinx FFT core 7.1 for comparing this. The FFT core uses DSP slices which are in fact HEBs already incorporated in the existing devices. As in FFT using butterfly HEB, here also we use the same model, where data is read from DRAM to BRAM and the core uses the data in BRAM for the computation. To make our comparisons realistic, we choose Radix 2 butterfly design provided by core generator in Xilinx ISE. All speedups reported are with respect to software execution using highly optimized FFTW [5] libraries run on desktop PC with Intel Core 2 duo E4700, 2.6 GHz processor with 4 GB RAM. The FFTW library is a C subroutine library which can be used to compute mutidimensional discrete Fourier transform, for both real as well as complex data.

In order to do design space exploration of the HEBs, we first estimate the number of FFT cores that can be instantiated for different devices. For the Xilinx core, this limit could come from one of the three following factors:

- Number of BRAMs available in the device
- Number of DSP units available in the device
- Constraint of logic blocks

For the butterfly HEB, this limit comes either due to number of slices or BRAMs. In our case, the limit came due to the BRAMs available. The estimated number of cores for devices for 1024-point and 2048-point FFT are listed in Table 6.2.

The number of banks in the available DRAM is also varied to see the effect of memory bandwidth on the HEBs. Basically, the number of HEBs that can be effectively used may be constrained by the memory bandwidth. As we consider existing BRAMs which have fixed maximum frequency of operation, we increase the bandwidth by increasing the number of DRAM banks. While increasing the number of butterfly HEBs, we see to it that the area of all the HEBs is not more than the area of all the DSP units present on the device. Thus for each device, we have seen that the configurability of FPGA is not compromised. The speedups over software version using FFTW library for 2048-point 3D-FFT on XC6VSX475T device is listed in Table 6.3 with the number of Xilinx cores equal to number of HEB cores [17]. It can be seen that as bandwidth increases, the number of cores that can be instantiated increases and hence the speedup. After reaching a bandwidth of 6 GBps per bank, the number of HEB cores which can be effectively used saturates due to limited amount of BRAM resources on the FPGA. This causes the speedup

Table 6.1 Area–time report of 1D-FFT using Xilinx core and 1D-FFT core using butterfly HEB

FFT (pt)		Slices	BRAM	DSP/butterfly HEB	Equivalent slices	Computation clock		Cycles
						ns	MHz	
2048	HEB	149	12	1	359	3.165	315.956	11,271
	Xilinx core	658	13	12	718	3.381	295.77	19,684
1024	HEB	144	6	1	354	3.37	296.736	5127
	Xilinx core	686	7	12	746	3.449	289.939	9427
512	HEB	117	5	1	327	3.26	306.748	2311
	Xilinx core	649	6	12	709	3.398	294.291	4546
256	HEB	121	5	1	331	3.189	313.578	1031
	Xilinx core	640	6	12	700	3.379	295.946	2225
128	HEB	114	5	1	324	3.259	306.843	455
	Xilinx core	635	6	12	695	3.383	295.596	1120
64	HEB	109	5	1	319	3.276	305.25	199
	Xilinx core	615	6	12	675	3.368	296.912	591
32	HEB	107	5	1	317	3.321	301.114	87
	Xilinx core	519	6	12	579	3.491	286.451	591

(from Xilinx ISE after P&R)

Table 6.2 Number of 1D-FFT cores that can be instantiated on different devices

FFT length	1024				2048			
Design using	Xilinx core		FFT core with butterfly HEB		Xilinx core		FFT core with butterfly HEB	
FPGA family	Number	Limit	Number	Limit	Number	Limit	Number	Limit
XC6VSX475T	108	Slices	177	BRAM	81	BRAM	88	BRAM
XC6VSX315T	71	Slices	117	BRAM	54	BRAM	58	BRAM
XC6VHX565T	72	DSP	152	BRAM	70	BRAM	76	BRAM
XC6VHX250T	48	DSP	84	BRAM	38	BRAM	42	BRAM
XC6VLX760	72	DSP	120	BRAM	55	BRAM	60	BRAM
XC6VLX75T	16	Slices	26	BRAM	12	BRAM	13	BRAM

Table 6.3 Speedups over software for 2048-pt 3D-FFT on XC6VSX475T device

Bandwidth GBps per bank	Speedup using butterfly HEB	Speedup using Xilinx core using DSP
9	1953×	1522×
8	1953×	1522×
7	1953×	1522×
6	1854×	1446×
5	1545×	1218×
4	1236×	989×
3	927×	761×
2	618×	533×
1	309×	304×

to saturate. When the bandwidth is 9 GBps per bank, the speedups compared to software version using butterfly HEBS was 1953× and using Xilinx core was 1522×. The speedups over software version for different FFT lengths are shown in Fig. 6.4. Here the total combined area of all the Xilinx cores is equivalent to the combined area of all the HEB cores. It can be observed that with normalized area, our HEBs give comparatively higher speedup compared to Xilinx core which uses DSP HEB. For example, for 1024 point 3D-FFT, speedups obtained over software version was 2833× using butterfly HEB and 1031.72× for Xilinx core using DSPs. This was due to the fact that the number of 1D-FFT cores which could be instantiated on the fabric with butterfly HEB was 49 while 26 1D-FFT Xilinx cores using DSPs could be instantiated in the equivalent area.

The implementations reported in Sect. 6.1.1 focused on reducing the critical path which mostly lies in the HEB. Even though all have reported significant reduction in area, the importance of the reduction is not shown. But in the case of application acceleration, area occupied by the resource, along with the critical path becomes an important factor as the number of units that can be instantiated decides the speedup. An open source protein docking application FTDock [14] is used as a case study and is accelerated using the designed FPGA architecture. FTDock application performs rigid-body docking on two molecules. Profiling was done to identify the kernel. It was found that the 3D-FFT computations take up 94.1 % of the total execution time. The overall speedup of 17× was obtained using butterfly HEBs when compared to software only version executed on desktop PC with Intel Core 2 duo E4700, 2.6 GHz processor with 4 GB RAM. The speedup is limited to around 17× as the 94.1 % of the application execution time is used by FFTs which is accelerated using FPGA. Since rest of the application (5.9 %) that runs on processor is serial and cannot be parallelized, we do not expect any changes in speedups even if newer architectures like Intel I7 having up to 6 cores are used.

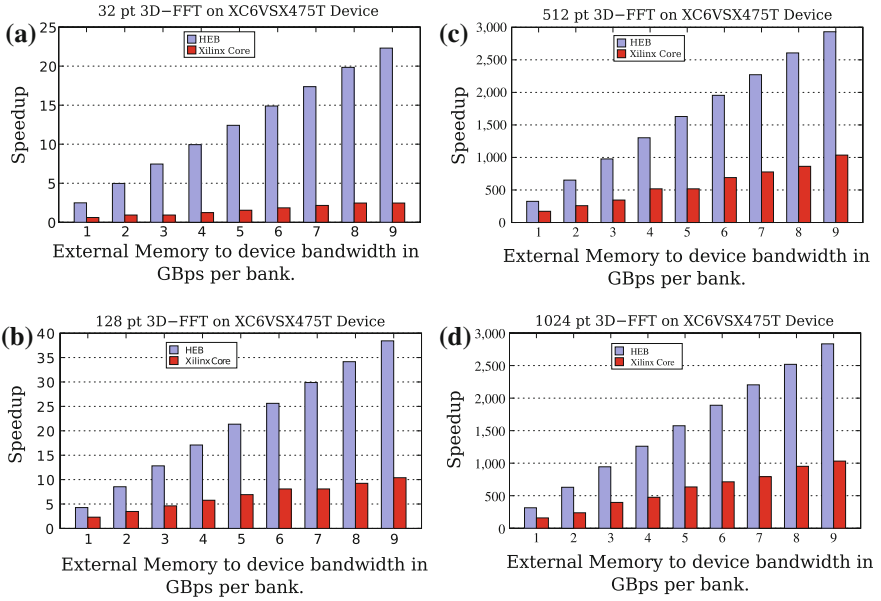

Fig. 6.4 Speedups for 3D-FFT with different number of points on XC6VSX475T device. **a** 32-pt 3D-FFT. **b** 128-pt 3D-FFT. **c** 512-pt 3D-FFT. **d** 1024-pt 3D-FFT

6.2 Acceleration of Genome Assembly Using Hard Embedded Blocks

In Chap. 5, we had discussed a novel method to accelerate de novo assembly using FPGAs. In this section, we show further speedups by designing HEBs customized to accelerate de novo assembly using FPGAs. Using the HEBs, we show that speedups of up to 11× can be obtained using FPGAs containing the HEBs. To the best of our knowledge, this is the first attempt to accelerate de novo genome assembly using HEBs in FPGAs.

Since genomics has usage in various fields like personalized drugs, metagenomics, etc., the usage of custom HEBs can be justified. Also our HEB can be used to find Hamming distance between two vectors. Hamming distance calculation is commonly used in digital communication and in many encryption–decryption algorithms [6]. It very often comes in the critical path of these applications. Using HEBs can reduce the execution time. Custom HEBs, due to their dedicated application-specific design, occupy less silicon chip area compared to LUT implementation and hence the FPGA resources can be used optimally for the rest of the operations.

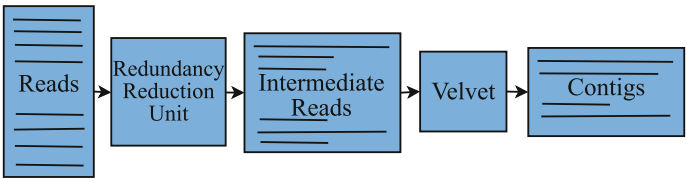

Fig. 6.5 FPGA-based approach to accelerate Velvet

6.2.1 Algorithm—Recap

We first summarize the application described in Chap. 5. To reduce the execution time of Velvet software, we generate ***intermediate contigs*** using a redundancy removing unit (RRU) implemented on an FPGA. The block diagram is shown in Fig. 6.5. These intermediate contigs store the overlap information only once. The intermediate contigs generated by the FPGA are given to the velvet software to generate the contigs.

The RRU design consists of processing elements connected in a series. Each processing element stores a sequence known as starter. The reads are streamed through each of the starters stored in the PEs to check if they extend the starter. A read extends a starter on the right if starter's right end matches with the read's left end. Similarly, a read extends the starter on the left if starter's left end matches with the right end of the read.

'N' first reads from read set *'R'* are stored as starters in 'N' processing elements. The $(N+1)$th read is checked with each starter for extension. If the starter is extended, the read is marked as extended and discarded. Thus, each read is tried with all the starters for extension. One such iteration of comparing reads with one set of starters is termed as a "round". If the read does not extend any of the starters, it is stored for comparison for the next round in remaining read set *'RemR'*. The starters which do not get extended in the current round become the intermediate contigs, as they do not get extended in further rounds. The starters which became intermediate contigs are replaced by first few reads of *'RemR'*. The *'RemR'* is now made new read set *'R'*. This process is repeated with the new read set *'R'* till the number of reads in the remaining read set *'RemR'* is less than "N," the number of PEs.

6.2.2 FPGA Resource Mapping

The hardware implementation of the RRU was done on an FPGA board. The algorithm is slightly modified to make it suitable for the hardware implementation. Due to hardware resource constraints, only the left and right ends of the starter equivalent to length of reads is stored in the PE. The process of extension is time consuming, as it involves comparison of read with the starter at each position. To reduce the time,

the we have proposed to store a "read vector" with each read in order to quickly check if there is a possibility of extension by the starter. The main idea is to find the number of matching 4-mers. If this number is greater than a threshold, the read is tried for extension. The read vector is a 256-bit vector whose position depicts an existence of a 4-mer. The starters left vector and starters right vector are stored along with the starters. A logical 'AND' between read vector and starter's vectors marks the matching 4 mers. Now the number of matched 4-mers can be found by counting the number of '1's in the result. This modification is called *"pre-filtering"* which saves a lot of time by bypassing the extension process for all but a small number of starters. The PE design is shown in Fig. 5.12. Each PE consists of a prefilter, an extension unit, and a control unit. The reads which pass the prefilter are passed to the extender. In each cycle, the extender shifts the read by one base pair and compares with the starter. The extender updates the starter if extended, marks the read, and sends it to the next PE. The control unit activates the FIFO control signals as and when required. Each PE is preceded by a FIFO to store the intermediate results. The FIFOs are added to keep the processing elements busy, as the PEs take varying number of clock cycles to process a single read.

The hardware implementation was done using Xilinx ISE. After synthesis, we found that the 256-bit 1-counter came in the critical path of the design which decides the operating frequency. In order to increase the operating frequency, we have designed a "256-bit 1-counter HEB" to do this operation. The basic function of 256-bit 1-counter HEB is to add all the bits in the 256-bit vector. To get more speedups, we have to implement more number of PEs on a single chip. As each PE is preceded by FIFO in our design, it was important to reduce area occupied by the FIFOs. We used BRAMs which are popular in current FPGAs as HEBs. The FIFO controller which was built around the BRAMs also takes significant area and hence we used the existing FIFO controllers in the Xilinx FPGA to optimize the overall design both in terms of area and performance.

6.2.3 Hard Embedded Block Design Space Exploration

We use the methodology shown in Fig. 3.2 for design space exploration. The application is modified to suit the FPGA accelerator technology. From this modified design, we do the synthesis using standard FPGA design tools and find the critical path and resource usage. Using the results of such an analysis, HEB has to be identified. Usually, the logic which comes under the critical path is made an HEB if the next delay path is considerably less than the critical path delay. The granularity is either decided by the number of times the "time critical part" is instantiated in the design in the case of fine-grained HEB or by using the resource occupancy of the modules in the design in case of coarse-grained HEB. In our approach, the HEB is designed using virtual embedded block (VEB) methodology [7]. The application is designed using the HEBs and the resource usage and critical path are obtained using ISE synthesis and place and route tools. A high-level implementation mimicking the hardware was done to identify the speedups.

6.2.3.1 Hard Embedded Block Design

We use VEB methodology for evaluating the 256-bit 1-counter HEB [7]. The VEB methodology can be used for performance evaluation of future fabrics which contain many heterogeneous HEBs. ASIC synthesis of the HEB block has to be done using ASIC synthesis tools. The area and the delay from these tools will be used to model the dummy block in Xilinx ISE. In VEB methodology, the HEB is introduced as dummy block in the design. The delay of the dummy block is made equivalent to the delay of the HEB. For area analysis, the area of the dummy block is designed in such a way that is equivalent to the area of the HEB. The dummy block is modeled as a hard macro or a relationally placed macro (RPM) using Xilinx design tools.

The top-level design has to be modified to infer these HEBs as components instead of the logic blocks. This design has to go through synthesis as well as place and route tools of Xilinx ISE to get the exact operating frequency numbers and the area estimates. Using the operating frequency, the overall execution time using an FPGA with such an embedding has to be estimated for different genomes. This can be done in two ways. In the first method, the VHDL model without HEB can be used for simulation with different input sets. The VHDL model without the HEB ensures a functionally correct and working model. This model can be tweaked to extract the exact number of cycles taken by the FPGA with embedded HEBs. The second method is to implement a high-level design in software to mimic the hardware. The simulation time is a crucial factor for studying the effects of such a design, as the input size is very large (For *E. coli* bacteria the input size with 100× coverage is 2 GB). The simulation time could take up days for analyzing such an embedding. The VHDL model would take very long time as it is dependent on the simulation tool on which it is run. The simulation tool overhead is also a matter of concern. The high-level software model is a faster method to estimate the overall execution time and we discuss this in the next chapter.

We designed a 256-bit 1-counter HEB for accelerating de novo genome assembly. The 256-bit 1-counter is a unit which takes a 256-bit vector and gives the number of '1's in the vector as an output. The adder adds each bit to obtain the number of '1's in the input. For example if the input vector is "00010101" the output is "0011" (decimal-3) in binary. This 256-bit 1-counter is used in the prefilter section of the RRU. Many experiments were carried out to implement an efficient 256-bit 1-counter. We designed a Wallace tree with 6:3 compressors. We chose 6:3 compressors as they would fit exactly in three 6-input LUTs inside the FPGA. We also implemented 256-bit 1-counter by adders by writing a behavioral code using commonly used shift-add method. The design which uses a tree-based implementation gave significantly better results. The Fig. 6.6 shows the adder tree for 8-bits. For example, at the bottom level of 8-bit adder, four 2-bit adders are used to add in parallel. Similarly, outputs are added at the next level in chunks of two till we find the overall sum, which is the required result.

The adder was designed using VHDL and synthesized using Synopsys Design Compiler tool [15] using UMC [16] 90 nm library. The area of the 256-bit 1-counter was 11983 μm^2 and the operating frequency was 185 MHz. We scaled this to 40 nm,

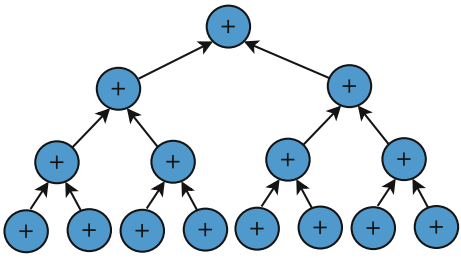

Fig. 6.6 Adder design

using ITRS scaling factors, where area is halved for 0.7× scaling in technology for the same frequency of operation [9]. The area of the 256-bit 1-counter was found to be equivalent to area of 1 Virtex-6 FPGA slice using the slice area from [11]. In the VEB design, the HEB has to be represented in terms of slices in an existing FPGA. A single slice with 256 inputs will need special routing resources. Instead, we have used the slice area as 6 slices to accommodate 256 inputs. This is an underestimate as a design using 6 times its area can in fact have better operating frequency.

6.2.3.2 Application Design Using Hard Embedded Blocks

We used Mapsembler [12], an open-source software implemented in C language to mimic the hardware implementation. Counters were introduced in the model to give the number of clock cycles taken by the design with HEB for a particular input. The time taken by the FPGA with HEB as well as the output from the FPGA is obtained from the high-level model. This output was fed to the Velvet software to calculate the overall execution time for the sample input. The data transfer delays were picked from the design on Alphadata board [1]. The overall time taken by the RRU and Velvet after each round of processing is calculated and compared with the time taken by Velvet using the unprocessed input. This is because the absence of such an accelerator would imply that Velvet processes the complete input stream.

6.2.4 Results and Discussion

We have implemented the RRU on Xilinx Virtex-6 SX475T FPGA using the Xilinx ISE tools [19]. The design was validated on an AlphaData FPGA board with Xilinx Virtex-6 SX475T FPGA. The design for the 256-bit 1-counter HEB was done using the VEB methodology using the Xilinx tools. For estimating speedups of fabrics with HEBs, we use the critical path obtained by the VEB method. It is increasingly common to see FPGA development boards containing multiple FPGAs. We estimate the speedups for an FPGA board with 5 such VIrtex-6 FPGAs [18].

The area and the timing results from a single FPGA were used to estimate the speedups using 5-FPGA board. The communication overhead due to multiple FPGAs has not been considered. We compare the time which Velvet takes without the preprocessing and compare it with the sum of the time taken by FPGA preprocessing and the time taken by Velvet to process the "reduced" FPGA output. The software version was run on a computer with Intel Core2 duo E4700 processor running at 2.6 GHz with 4 GB RAM and Ubuntu operating system. The reads were generated synthetically using the scripts provided by Zhang et al. [21]. The results are shown for Swinepox virus. The reads of length 36 and 75 were generated with 100× coverage.

Table 6.4 gives the number of PEs that can be designed on an printed circuit board with 5 Virtex-6 SX475T FPGAs. Our objective is to see the effect of our HEBs as well as FIFO controller HEB which exists in modern FPGAs. The number of PEs which can be put on the board increases as we use HEBs as they take lesser area compared to the 256-bit 1-counter implementation using logic blocks. The clock frequency is also considerably increased. The operating frequency increased by a factor 2, when 256-bit 1-counter HEBs were used. The number of 256-bit 1-counter HEBs used is twice the number of PEs used as each PE requires two such 256-bit 1-counters. For read length 75, the number of slices used increases and hence the number of PEs is less when compared to the number of PEs when read length is 36.

The graphs of the speedups using read length 36 is plotted in Fig. 6.7. In this figure, the reduction in the size of input (base pairs) is also shown. The rate of base pair reduction with each round decreases as the number of base pairs that get used up in the extensions are more in the beginning. For example, it can be seen in Fig. 6.7a that there is reduction from $14 * 10^6$ bp to almost $1 * 10^6$ bp after 10 rounds and reduces very slowly after that. During the later rounds, only those reads are left which mostly do not extend any of the starters and become the intermediate contigs.

The FPGA processing time for each round is plotted in Fig. 6.8 for *E. coli* genome assembly with read length 36 and 100× coverage containing 0.1 % errors. We see significant reduction in FPGA processing time when HEBs are used. For example, at the 200th round, we see that the processing time is 375 s when neither of the HEBs were used, where as the time reduced to 150 s when both the HEBs were used.

From the graphs it can be observed that the speedups increase when specialized HEBs are incorporated. For example, it can be observed from Fig. 6.7b, c that the speedups increase from 9× to 11×. By using the in-built Xilinx FIFO controllers, the speedup is increased and when 256-bit 1-counter HEBs are used, the speedups further goes up. This increase in speedups can be attributed to the increase in number of PEs which can be fitted on the FPGA and the increase in operating frequency, when HEBs are used. For longer reads, the speedups can be increased by increasing the number of FPGAs used. Similarly for larger genomes, the number of FPGAs have to be increased in order to increase the number of PEs used. As the FPGAs become larger, there will be higher speedups possible due to the FPGA's capacity to accommodate more number of PEs.

The Velvet software has not been particularly optimized to execute on a multiple CPUs (nodes), but as it is implemented using a multi-threaded design, the execution time can be reduced by running on a multi-core machine. Newer architectures like

Table 6.4 Area and operating frequency of the RRU

Read length	1 Virtex-6 FPGA (SX475T)							
	Without 256-bit 1-counter HEB				With 256-bit 1-counter HEB			
	Without FIFO-controller HEB		With FIFO-controller HEB		Without FIFO-controller HEB		With FIFO-controller HEB	
	No. of PEs	Clock Freq.	No. of PEs	Clock Freq.	No. of PEs	Clock Freq.	No. of PEs	Clock Freq.
36	60	98 MHz	73	98 MHz	80	185 MHz	103	185 MHz
75	50	98 MHz	60	98 MHz	63	185 MHz	79	185 MHz
5-FPGA board								
36	300	98 MHz	365	98 MHz	400	185 MHz	515	185 MHz
75	250	98 MHz	300	98 MHz	315	185 MHz	395	185 MHz

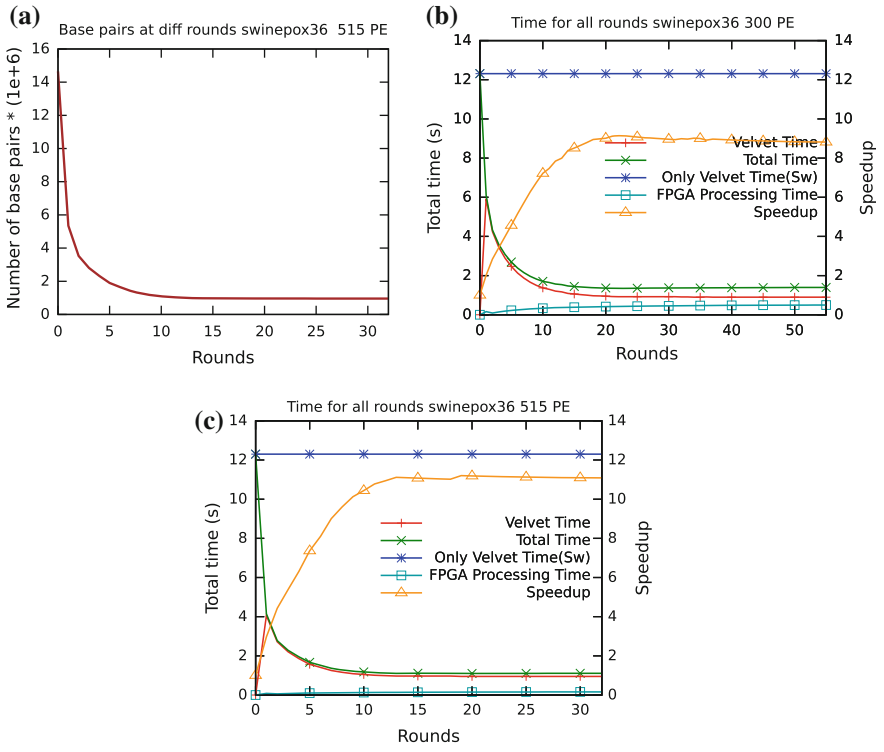

Fig. 6.7 Speedups and number of base pairs after each round for Swinepox read length 36. **a** Input size with 256-bit 1-counter HEB and with FIFO controller HEB. **b** Speedups without 256-bit 1-counter HEB. **c** Speedups with 256-bit 1-counter HEB and with FIFO controller HEB

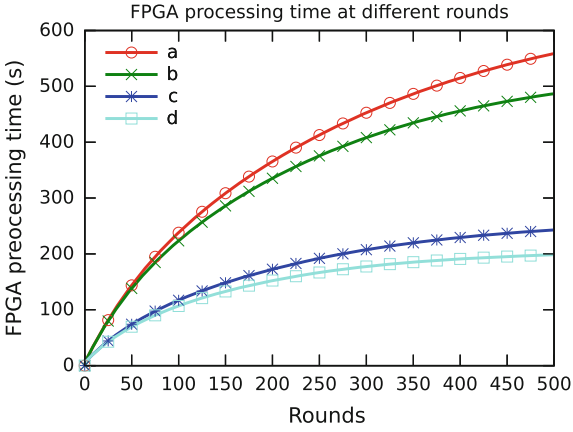

Fig. 6.8 FPGA processing time for each round for *E. coli* genome assembly. Key: **a** without any HEBs, **b** with only FIFO controller HEBs, **c** with only 1-counter HEBs, **d** with both the HEBs

Intel I7 have up to 6 cores and this would speed up the execution of Velvet vis-a-vis processors containing fewer cores. From the results, we can infer that the speedups are limited by the Velvet software which is used to construct the contigs from the intermediate contigs. Using efficient hardware design methods, the pre-processing time in hardware can be reduced, but the speed-ups will still be limited by the Velvet software needed to construct the contigs. Thus we expect slight reduction in speedups on newer platforms with higher number of cores.

6.3 Summary

In this chapter, we showed acceleration of 3D-FFT and Genome assembly by incorporating HEBs in FPGAs.

The butterfly unit was implemented as an HEB for the 3D-FFT acceleration. FFT has critical role in many fields and many applications which use FFTs can be accelerated using such fabrics. The 256-bit 1-counter HEB can be used in many other applications including cryptography and digital communications, where hamming distance has to be calculated. The 256-bit 1-counter we designed takes a 256-bit vector, but can easily be configured for using it for larger vector inputs.

The time taken by compute-intensive applications can be reduced significantly by using FPGA-based accelerators. To evaluate HEBs, simulation models were developed. Performance estimates using these models were used to design the HEBs. For protein docking application, analytical models could be easily used for performance estimates. For de novo genome assembly, the speedups cannot be easily predicted with analytical models, as the number of "rounds" cannot be predicted for a sample input. Hence, we used high-level simulation models for performance estimates. As performance estimates are tightly integrated in our methodology for DSE, we show that choosing the right parameters at different levels of abstraction helps reduce the overall time for DSE. We describe the use of high-level models for DSE for accelerating de novo genome assembly in the next chapter.

References

1. Alpha-Data: Alpha-Data FPGA Boards. http://www.alpha-data.com/ (2015)
2. Beauchamp, M.J., Hauck, S., Underwood, K.D., Hemmert, K.S.: Embedded floating-point units in FPGAs. In: ACM/SIGDA International Symposium on FPGAs (2006)
3. Chong, Y.J., Parameswaran, S.: Flexible multi-mode embedded floating-point unit for field programmable gate arrays. In: ACM/SIGDA International Symposium on FPGAs (2009)
4. Flachs, B., Asano, S., Dhong, S., Hotstee, P., Gervais, G., Kim, R., Le, T., Liu, P., Leenstra, J., Liberty, J., Michael, B., Oh, H., Mueller, S., Takahashi, O., Hatakeyama, A., Watanabe, Y., Yano, N.: A streaming processing unit for a CELL processor. In: IEEE International Solid-State Circuits Conference (2005)
5. Frigo, M., Johnson, S.G.: The Design and Implementation of FFTW3. Proc. IEEE (2005)
6. Haykin, S.: Communication Systems, 5th edn. Wiley Publishing (2009)

7. Ho, C.H., Leong, P.H.W., Luk, W., Wilton, S.J.E., Lopez-Buedo, S.: Virtual embedded blocks: a methodology for evaluating embedded elements in FPGAs. In: IEEE Symposium on FCCM (2006)
8. Ho, C.H., Yu, C.W., Leong, P., Luk, W., Wilton, S.: Domain-specific hybrid FPGA: architecture and floating point applications. In: International Conference on FPL (2007)
9. ITRS: The International Technology Roadmap for Semiconductors. http://www.itrs.net (2015)
10. Jamieson, P., Rose, J.: Enhancing the area efficiency of fpgas with hard circuits using shadow clusters. IEEE Trans. VLSI Syst. **18**(12), 1696–1709 (2010)
11. Kapre, N.: Spice2 a spatial parallel architecture for accelerating the spice circuit simulator. PhD dissertation, California Institute of Technology, Pasadena, California (2010)
12. Peterlongo, P., Chikhi, R.: Mapsembler, targeted and micro assembly of large NGS datasets on a desktop computer. BMC Bioinform. **13**(1) (2012)
13. Roesler, E., Nelson, B.E.: Novel optimizations for hardware floating-point units in a modern FPGA architecture. In: 12th International Conference on FPL (2002)
14. Sternberg, M.J.E., Aloy, P., Gabb, H.A., Jackson, R.M., Moont, G., Querol, E., Aviles, F.X.: A computational system for modeling flexible protein-protein and protein-DNA docking. In: Proceedings of the 6th International Conference on Intelligent Systems for Molecular Biology, pp. 183–192 (1998)
15. Synopsys: Synopsys Design Compiler. http://www.synopsys.com (2015)
16. UMC: UMC, 90 nm Libraries. www.umc.com/ (2015)
17. Varma, B.S.C., Paul, K., Balakrishnan, M.: Accelerating 3D-FFT using hard embedded blocks in FPGAs. In: 2013 26th International Conference on VLSI Design and 2013 12th International Conference on Embedded Systems (VLSID), pp. 92–97 (2013)
18. Varma, B.S.C., Paul, K., Balakrishnan, M.: Accelerating genome assembly using hard embedded blocks in FPGAs. In: 2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems, pp. 306–311 (2014)
19. Xilinx: Xilinx FPGAs, ISE. http://www.xilinx.com (2015)
20. Yu, C.W., Smith, A., Luk, W., Leong, P., Wilton, S.: Optimizing coarse-grained units in floating point hybrid FPGA. In: International Conference on FPT (2008)
21. Zhang, W., Chen, J., Yang, Y., Tang, Y., Shang, J., Shen, B.: A practical comparison of de novo genome assembly software tools for next-generation sequencing technologies. PLoS ONE **6**(3) (2011)

Chapter 7
System-Level Design Space Exploration

Abstract In this chapter, we discuss the design space exploration carried out to accelerate de novo genome assembly using FPGAs. It is well known that as systems become more complex, one moves up the abstraction level for design space exploration through simulation. This is essential for managing complexity. Normally, higher abstraction levels imply faster and wider design space exploration but come at the price of lower accuracy. The focus of this chapter is on high-level design space exploration. The exploration was carried out at three levels. In this chapter, we propose modeling at three different levels: high-level algorithm model using 'C' followed by cycle-accurate model using System-C, and finally RTL component model using VHDL. We classify the parameters that can be studied using these three models.

7.1 Introduction

The problem of accelerating applications is challenging as there are a large number of design choices and each of them have to be evaluated to develop a near-optimal design. First step is to arrive at a suitable algorithm which is efficient in terms of complexity. Typically, this translates into minimizing the number of primitive operations like arithmetic, logical, memory accesses, etc. The other important design decision is the choice of hardware for accelerating such applications. Many of these applications are run on general-purpose processors along with some kernels and interfaces implemented as custom hardware or reconfigurable accelerators. These accelerators require application-specific coding and thus imply significant investment of design effort. It is important to analyze the speedups before actually building and testing them on the actual hardware. A system-level simulation of the platform along with an accelerator model helps in designing better accelerators.

B.S.C. Varma et al., *Architecture Exploration of FPGA Based Accelerators for BioInformatics Applications*, Springer Series in Advanced Microelectronics 55, DOI 10.1007/978-981-10-0591-6_7

7.2 Design Space Exploration

The main goal of the design space exploration was to accelerate Velvet software, which takes significant amount of time to execute on general-purpose processors. Profiling of the Velvet software was carried out and it was found that the compute-intensive kernels in Velvet software cannot be directly implemented on FPGA. This is because the hardware resources available even on the largest FPGA are not adequate. Further, the direct implementation would require storing of the *de Bruijn* graph which itself requires large amounts of memory. The next strategy was to investigate if we could do preprocessing of the reads and then give the output to Velvet software for constructing the contigs. The contigs can further be used by other software tools for constructing the genome. To investigate this, we used an existing open-source targeted assembly software called Mapsembler [7]. The key idea was to use a hybrid approach based on both OLC method and *de Bruijn* graph-based de novo genome assembly. As the reads in the *de Bruijn* graph method are broken into respective k-mers before processing, storing the read-only once would reduce the input size for Velvet processing. The various parameters that were explored using the three models are shown in Fig. 7.1. The C model was essentially used to test the feasibility. We then implemented the VHDL model of the core to get the area and delay (clock period as well as number of clock cycles) required for performance estimation.

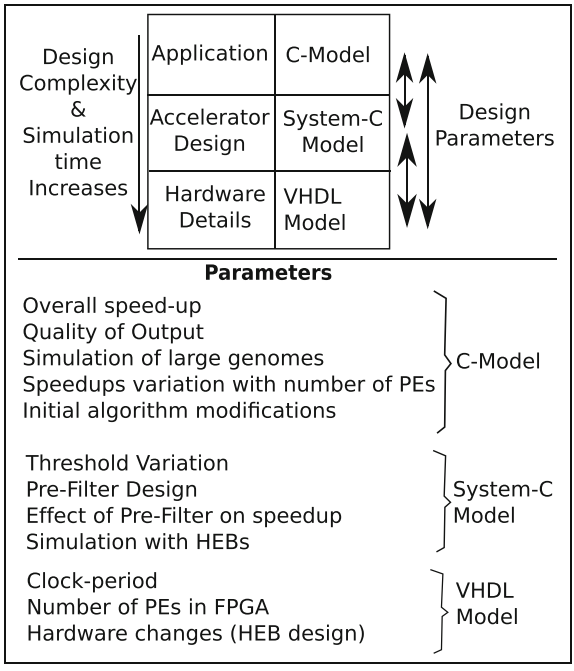

Fig. 7.1 Block diagram

Table 7.1 Three models: simulation time for *E. coli* and human influenza genomes

Model\Genome	Simulation time	
	H. Influenza	*E. coli*
C	50 s	2.5 min
System-C	3 h 40 min	1 h 13 min
VHDL	200 h	601 h

Further, a cycle-accurate System-C model was developed to know the exact number of processing cycles required as it would translate to possible concurrency-based resource availability.

The simulation time taken for the three models developed are shown in Table 7.1. Clearly, there is almost two-order increase in speed using a higher abstraction model vis-a-vis lower level model. The platform used was a CPU with 2 GHz Intel Core i7-3537U processor with Ubuntu 13.04 OS and 8 GB RAM. VHDL simulation was done using ModelSim [1]. The time taken for simulation reported here is for *E. coli* genome and human influenza virus genome with read length 36 and for hardware platform with 250 processing elements (PEs). From the table it can be observed that VHDL simulations for doing the design space exploration are not feasible. The genome length of human influenza virus is around 13,588 base pairs and the *E. coli* genome length is around $4.6 * 10^6$ base pairs. Note that human genome's length is around 3 billion base pairs and clearly larger genomes are not feasible to use VHDL simulations for design space explorations. Even though some of the parameters like clock period and the resources utilized can be obtained from VHDL model, whole system simulation is not feasible. C-model can be used for very-high- level design-space exploration to design the major architecture features. System-C model can be used for complete system simulation and thus can be used to decide multiple design parameters.

7.2.1 C-Implementation

The Mapsembler software was used as a C-model to do the design space exploration at algorithmic level. The Mapsembler does targeted assembly, i.e., it will try to extend a smaller set of sequences known as "starters" with sequences in a larger "read-set." For doing assembly using Mapsembler, we take the first 'N' reads from the read set and seed them as starters. The rest of the reads in the read set are compared with the starters one by one for extension. We precisely repeat the core of the algorithm presented in Chap. 5.

In order to check if reads have completely extended the starters, multiple iteration of comparison of the reads with one set of starters has to be done. The reads which extend the starters are removed from the read set as their information is captured

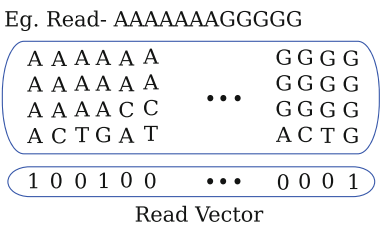

Fig. 7.2 Read vector construction

in the starter. The starters that do not extend in the current round, will not extend in further rounds and hence have to removed. These removed starters are stored in the output file known as "intermediate contigs." At the end of each round, the nonextended starters are removed and replaced with a new set of reads as starters. The iterations are repeated till the number of reads from the read set is less than the number of starters. In Mapsembler, to fasten this process, the k-mers of the starters are indexed in a hash table. Extension is carried out only if any of the k-mers present in the read is already available in the hash table. The intermediate contigs generated after multiple iterations of extending the starters. The overall block diagram of our approach is shown in Fig. 5.2. The preprocessing of RRU in the FPGA as discussed in Chap. 5 and the intermediate contigs generated are given to host where Velvet software is run for generating the contigs.

The main advantages of the C-model is that not only it is fast but also it is easy to make changes without much debugging. The key elements which could be tested using this model were:

I Check the feasibility of the algorithm;
II Quality of the output;
III Performance numbers; and
IV Modification to the algorithm.

Starting from this model, the potential benefit of hardware implementation was assessed. The resources needed for storing the starters was estimated. We realized that there would be a memory resource constraint for storing the starters. Based on such an analysis, it was decided to store only the left and the right part of the starter equivalent to the read length.

The quality of the output is an important aspect of our study, as we were changing the input given to Velvet software. In the initial work, we allowed extensions with predefined number of mismatches. When the output of the Velvet was compared with the original output (reads were directly given to Velvet), the N-50 numbers were small and the maximum length of the contig was small. When the number of substitutions allowed was reduced to zero, i.e., no substitutions were allowed, there was little or no change in the N-50 and maximum contig length.

To get the performance numbers, we introduced counters for counting the number of clock cycles taken by each starter. As these starters function parallelly in hardware, the starter with maximum clock cycles was taken for computing the delay and thus the performance. The clock period was obtained from the VHDL model. This enabled

us to estimate speedups very early in the DSE. Various experiments were done to observe the variation of performance with the number of starters used.

In the algorithm, after each iteration of comparing starters with the read set, the starters that do not get extended are replaced by reads from the remaining read set. Based on the experimental study, it was found that the compression or the reads that extend the starters reduces after some iterations. This is because the reads that participate in the extension get utilized in the first few rounds. The number of iterations required to get good speedups varies with the input read set and the number of starters used. So we decided to stop the process of iterating based on a threshold value. After each iteration, the number of reads that participate in extension is stored and compared with the value in the previous iteration. We exit the redundancy removal unit when the difference in these values is less than the threshold.

We also found that compared to the number of reads that pass through the starter and that do not extend the starter are significantly more than the reads that actually extend the starters. The process of extension in hardware would involve comparing the read with the starter at each position. The number of clock cycles spent on trying for extension would be equivalent to length of the read. There is a lot of time wasted in doing unnecessary work. To overcome this, we came up with a strategy where we used "signatures" along with reads. This study is easier with a model closer to hardware and a System-C model was implemented for this purpose. The hardware VHDL model was used for getting the actual clock speed and resource usage.

7.2.2 Hardware Implementation

The hardware implementation was done using VHDL as explained in Chap. 5. The hardware model was used for

I Modification to the algorithm to suit the target hardware (FPGA);
II Compute performance numbers;
III Predict the performance numbers for variants of the fabric (non-existing as of now).

The algorithm was modified to suit the FPGA implementation. The reads are given to a file-coder implemented in hardware, which does the binary coding of the reads. The nucleotide in the reads are binary coded using two-bits; 'A' as "00," 'C' as "01," 'T' as "10," and 'G' as "11." This saves communication delays and also the FPGA resource usage.

A prefilter block is designed to save the time spent on extension for nonextending reads. The prefilter quickly checks if the read is likely to extend the starter. If the read passes the prefilter, it is evaluated for extension, else it is sent to the next processing element containing a different starter. A signature of the read termed as "read-vector" is stored along with the read. The read vector is a 256-bit vector storing the information of the 4-mers available in the read. The position in the vector tells the presence of the particular 4-mer. The particular bit is set if that 4-mer exists in the

read. As the read is copied as starters at the beginning of each iteration, the starter also stores the vector both for the left and the right ends of the starter. We term this "starter left vector" and the "starter right vector" for the left starter and the right starter. An example of a read vector is shown in Fig. 7.2. In the prefilter, a logical "AND" of read vector and starter left vector is computed. The number of matching 4-mers in the starter and the read is found by counting the number of '1's in this result. If the number of '1's is greater than a threshold, it is passed to the extender. Similarly, the same operation is performed with the right vector in parallel to save time. In the extender, the read is shifted and checked if it can be extended. The model with fixed mismatches will require more hardware as some more comparisons are needed. As, it was found from the C-model that mismatches caused the quality of output to deteriorate, we reduce the FPGA resource usage. The construction of the read vector is a one-time process and so is done along with binary coding of the reads.

The processing elements are connected in series preceded by the coding unit. As the processing units take irregular number of clock cycles for their operation, FIFOs are implemented between the processing elements.

The hardware model was used to get the actual operating frequency and the resource usage. The critical path of the design was in the 1-counter in the prefilter. The operating frequency without the HEBs was 98 MHz. For doing a design space exploration of HEBs, we consider design

I without any HEBs.
II with only FIFO controller as HEB (already such FIFO controllers exist in modern FPGAs).
III with only 1-counter HEBs.
IV with both the FIFO controller as well as 1-counter HEBs.

VEB methodology was used for building the model with HEBs. The VEB methodology can be used to get the area and the critical path estimates using the HEB design. In this methodology, the HEB is implemented as an ASIC to get the area and the time taken from input to output. The HEB block is implemented as a dummy block in FPGA with delay between the pins equivalent to the HEB. A separate area and delay model can be used for analysis.

The 1-counter HEB was built using synopsys design compiler tools and the critical path and area of the design was obtained [8]. The dummy blocks with equivalent area and delay was constructed as a Xilinx Relationally Placed Macro (RPM). These RPMs replace the actual blocks in the FPGA design and the critical path and resources occupied in the FPGA were evaluated. The area benefits provide more number of processing elements to be embedded into the FPGA and hence more speedup. As this model would not allow the analysis of speedups with HEBs, it was required to implement the system in a high-level description. System-C model was used to study the benefits of the HEBs.

7.2.3 System-C Implementation

The System-C model was required to get to a model which had more details and was nearer to hardware implementation. It is easier to model a parallel system using System-C than a C-implementation. Even though System-C model runs slower than the C-model, it makes performance analysis more accurate and easier. For analyzing the speedups for a co-design problem like this which is input data dependent and resource dependent, a System-C model is beneficial. We also wanted to study the benefits of embedding an hardware accelerator in the FPGA. As such a system does not exist, this model provides a mechanism to do performance analysis. The System-C model is similar to the hardware model as shown in Fig. 7.3. Each of the processing element is modeled as a "sc_method" as shown in Fig. 7.4. Counters are added for displaying the clock cycles. The System-C model was basically used for

I Study of the algorithm with the current design.
II Performance numbers with existing system.
III Performance numbers for future fabrics with HEBs.

The System-C model was used for studying the variation of the preprocessor output with the threshold used in the prefilter. The C-model had an implementation of hash tables for making it work faster, so it was not very accurate. The measurement of the clocks and use of the prefilter could be done using the System-C model. System-C model was used for observing the finer details along with getting more accurate measurements.

The performance numbers were generated by using cycle accurate model. The clock cycles obtained from the model were then added with the software time taken by the Velvet software for constructing the contigs from the intermediate contigs.

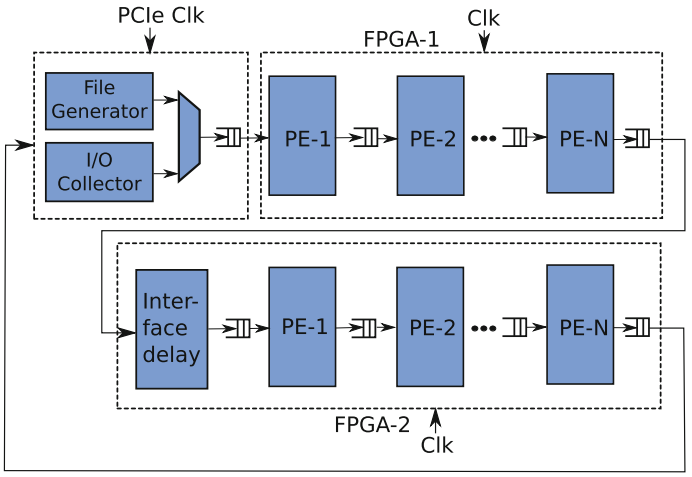

Fig. 7.3 Overall system design using System-C

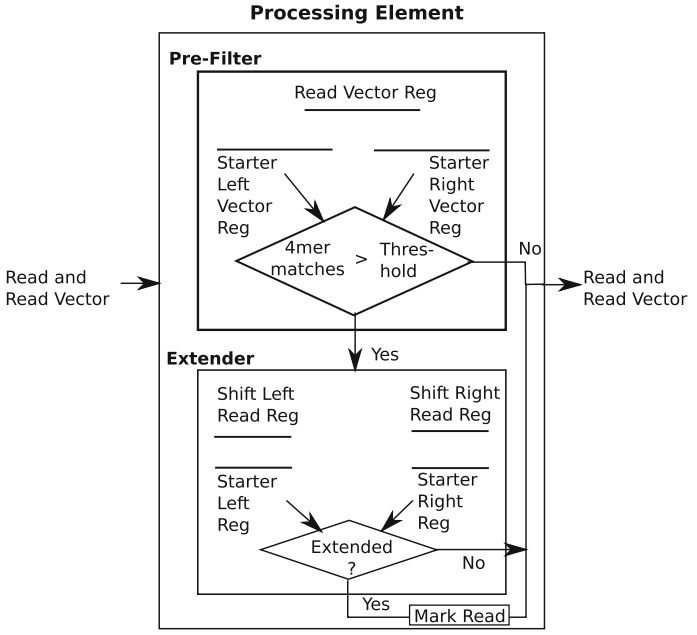

Fig. 7.4 Processing element design (system level)

This gave more realistic numbers for the speedups. There was no need to modify the system-C model for incorporating HEBs as the functionality remains the same without the HEB and with the HEB. The delay computation based on clock period and number of clocks were changed accordingly for estimating the overall performance.

7.3 Multi-FPGA Implementation

Bioinformatics applications deal with very large amount of data and there is considerable data parallelism available in the applications. Most of the times, parallelism available in the application cannot be exploited due to limited hardware resources. Modern FPGA-based accelerator cards contain more than one FPGA. Applications can be ported on these multi-FPGA cards to take advantage of the large number of resources available in the board. More concurrent execution units can be implemented to achieve higher speedups over software-only implementations. Typically, linear speedups using multiple FPGAs is not realized due to constraints on the I/O pins as well as delays due to interconnects. I/O pin constraints implies multiplexing for achieving data transfer. Recently, FPGAs have seen major increase in I/O pins and this also promotes assessment of multiple FPGA solutions for increasing performance.

We developed a System-C model to study application acceleration on multi-FPGA boards. We investigate the benefits of using multi-FPGA boards with FPGAs having custom HEBs in them. Again, FPGAs with HEBs will be able to implement number of processing elements on a single FPGA while simultaneously executing with a faster clock as HEBs reduce the critical path. We model the whole system containing host, interface, and FPGA board with multiple FPGAs using System-C. First, we map the application onto such fabrics including the required delays to estimate the overall execution time. The model was also used to study the memory bandwidth requirements for the application.

Multi-FPGA board application partitioning has been studied by Jing et al. [6]. The authors discuss methods basically aimed at energy efficiency. Many authors have reported studies on multi-FPGA interconnectivity topologies [2, 3]. Application mapping on multi-FPGA configuration has been studied by Sushil Chandra Jain et al. [5]. There has been work related to time multiplexing the interconnections between the FPGAs to achieve performance [4]. Our goal was to estimate the performance of board with multiple FPGAs that contained specialized blocks embedded in them. We study acceleration of de novo genome sequencing using such multi-FPGA boards.

7.4 Results and Discussion

The results reported here are based on the three models explained in Sect. 7.2. The performance numbers were obtained for single-FPGA board as well as multi-FPGA implementations. The number of FPGAs were varied from 1 to 4. The System-C model was executed on a system with Intel core I7 processor with 8 GB ram, running Ubuntu 13.04 operating system.

7.4.1 Single-FPGA

The three models were tested for correctness using sample inputs. The VHDL model was validated using a PCIe, alpha-data FPGA board having Virtex-6 FPGA. A comparison of various algorithms is shown and the perl scripts to generate reads are provided by Zhang et al. [11]. Similar to their input, we also generated synthetic reads. The reads were generated from the swinepox genome sequence. The input to the perl scripts were, the number of reads, read length and the error percentage. The coverage of the reads for our experiments was taken to be 100× and the number of reads was calculated based on the length of the genome for different read lengths. An error of 0.1 % was introduced into the reads.

The results from the three models were obtained and compared with the Velvet software [9]. For the speedups, the Velvet software time was compared with the preprocessing time and the postprocessing time using Velvet. Simulation using C-model is faster compared to other models and hence it is used for analyzing the speedups

with larger genomes like *E. coli*. It was also used for refinements in the algorithm as discussed previously. The C-model is faster due to the hash table implementation in Mapsembler. Since hardware implementation does not have the hash table implementation, System-C model was required to study the exact number of cycles saved by using the prefilter. The VHDL model was needed to get the exact area numbers and the critical path which decides the operating frequency. The VHDL model was also used to get the operating frequency and area estimates for the designs with HEBs.

Using C-model it is observed that the speedup first increases with the number of rounds and reaches a peak. After attaining this peak it starts to taper down. The speedup initially increases as the compression of the reads is initially quite high. During the later rounds, the reads which are left are mainly disjoint reads. The tapering of the speedups after attaining the peak can be attributed to the fact that effective work done on compression is reduced and the time is wasted for the read write time and the time spent in comparing the reads with the starter when there are very few extensions possible.

The VHDL model was implemented using Xilinx ISE [10]. The Xilinx synthesis and place and route tools were used to find the operating frequency of the circuit. It was found that the critical path of the design was in the 1-counter section of the prefilter. We implemented the 1-counter as an HEB using the VEB methodology. Synopsys design compiler tool was used for ASIC synthesis of the 1-counter block [8]. The 1-counter is built using a tree structure where bit wise addition is carried out at each level.

The operating frequency and the number of processing elements that could be fitted onto Xilinx Virtex-6 SX475T and Virtex-7 XC7A200T devices is tabulated in Table 7.2. The number of PEs that can fit in the FPGA increases and hence contribute to the speedups. We evaluated an already existing FIFO controller HEB and the 1-counter HEB. Even though the frequency of operation of the design with no HEB and with FIFO HEB is same the number of processing elements that can be implemented increases due to the benefits from the resource usage, as the HEBs occupy less area when compared to the area occupied by logic in LUTs. Similarly, the number of processing elements using both HEBs (FIFO controller and the 1-counter HEBs) in Xilinx Virtex7-XC7A200T FPGA is 247 when compared to design without HEBs which is 143.

Table 7.2 Resource usage and operating frequency obtained from VHDL model

Device	Feature	No HEB	FIFO HEB	1-counter HEB	Both HEBs
Virtex-6	No. of PEs	60	73	80	103
SX475T	Op. freq MHz	98	98	185	185
Virtex-7	No. of PEs	143	175	191	247
XC7A200T	Op. freq MHz	98	98	185	185

The variation of threshold with the compression and total time taken for processing when no HEBs are present and when HEBs are present are shown in Fig. 7.5a–c. This is estimated using the System-C model. The results shown were obtained for simulation of a single round of preprocessing using reads of Swinepox genome with read length 32 and 100× coverage and 0.1 % error. The compression increases with further rounds. It is observed that the time taken reduces after a certain threshold value. In Fig. 7.5a, after threshold value of 13 the time taken for preprocessing reduces. This means that the 4-mer matches in the reads and starter should be at-least 13 for getting better performance both in terms of time and compression. The time reduces as the cycles used by the prefilter is very small compared to the time spent in the extender by the reads that do not extend the starter.

The preprocessing time with varying "host to board data transfer bandwidths" variation is shown in Table 7.3. The preprocessing time is taken for swinepox with read length 32 for a single round. It can be observed that after 5 GBps, the preprocessing time saturates to 54.17 μs for no prefilter and 14.8 μs with prefilter with threshold 11. As the threshold for the prefilter is increased, the preprocessing time reduces as the time spent in the extender is avoided.

The time taken by the processing element with HEBs present is less as the operating frequency is more when HEBs are used. The variation of compression with varying thresholds for the designs without using HEBs and with HEBs is shown in Fig. 7.5c. The compression increases when HEBs are used because the number of processing elements that can be implemented on the device increases. As the processing elements increase, the read which can extend a particular starter need not wait for next iterations for finding its corresponding starter for extending. So, the work done per iteration on compression increases. The results shape the architecture and an optimal design can be implemented with such an exploration.

7.4.2 Multi FPGA

The various multi-FPGA topologies were studied using the System-C models are shown in Fig. 7.6. The 1-FPGA board has a PCIe interface to transfer data from the host to the board. In the 2-FPGA board, the PCIe interface connects the two FPGAs. The FPGAs are connected to each other using 300 I/O pins directly. The FPGAs are connected through 150 pins in the 3-FPGA and 4-FPGA boards. The inter-FPGA delay is assumed to be two cycles. We also evaluate performance for a 4-FPGA board which has two FPGAs clustered and connected to another cluster of two FPGAs through PCIe interface as shown in Fig. 7.6. In such an architecture, the PCIe bandwidth is shared equally between the host connectivity and for the inter-FPGA connectivity.

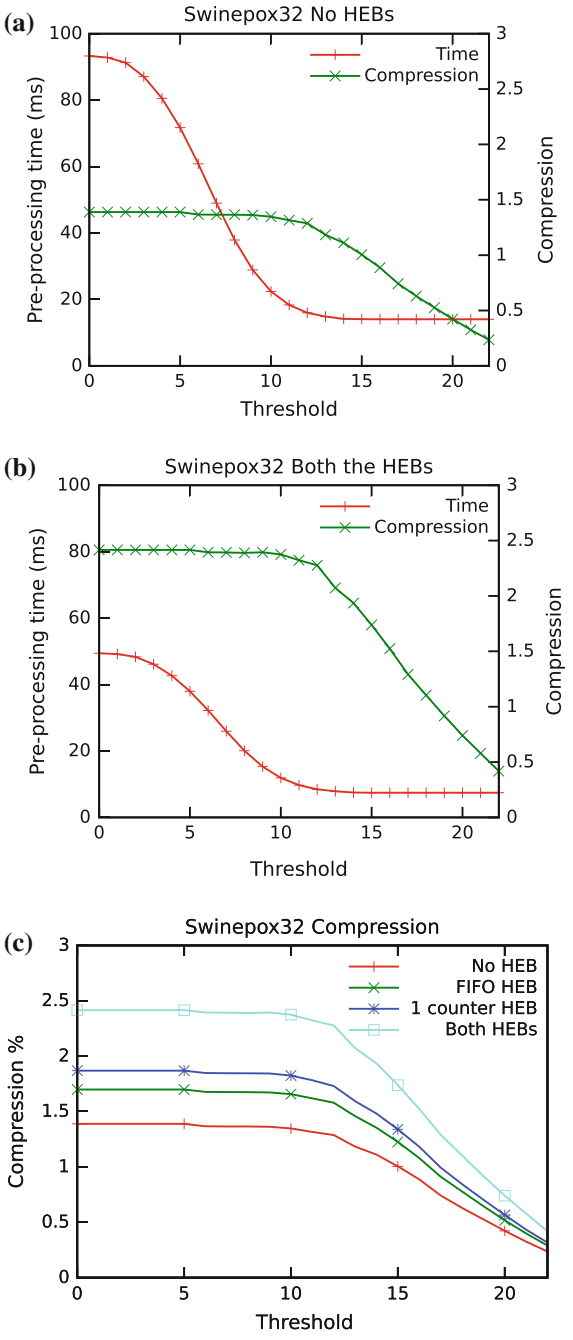

Fig. 7.5 Compression of reads. **a** Compression and time with varying threshold for single round without HEBs. **b** Compression and time with varying threshold for single round with HEBs. **c** Compression with varying threshold

Table 7.3 Variation of preprocessing time with host to board data transfer bandwidth

Host-board bandwidth in GBps	Preproc. time in µs for threshold 0	Preproc. time in µs for threshold 11
1	89.301	49.779
2	66.874	27.352
3	59.4	19.877
4	55.66	16.137
5	54.117	14.822
6	54.17	14.822

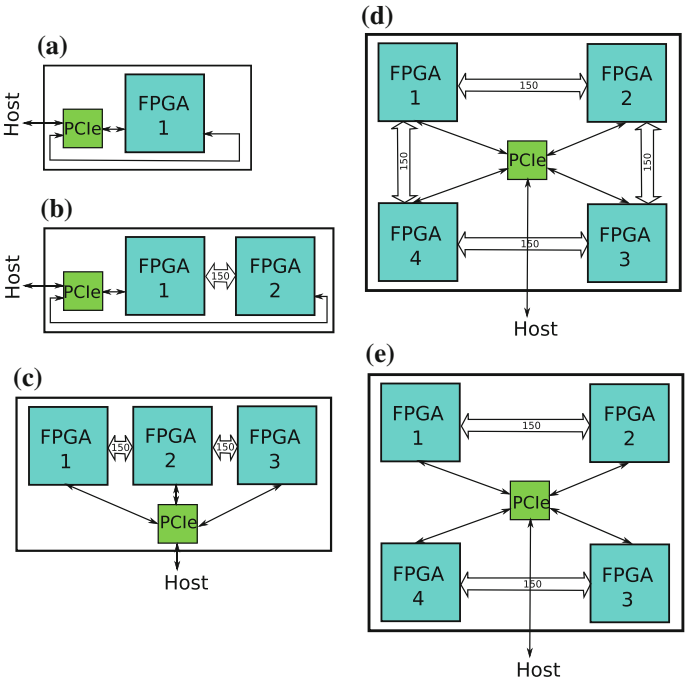

Fig. 7.6 Multi-FPGA topologies

The Velvet software is accelerated using the approach discussed in the previous chapters. The preprocessing is mapped to the FPGA. In the multi-FPGA design, we consider a system where the host supplies the reads to the FPGA board through PCIe interface. The PEs are implemented in the FPGAs. FIFOs are introduced in between the PEs. The PEs in two FPGAs are connected through FIFOs as shown in Fig. 7.7.

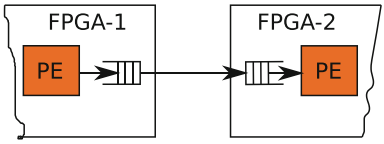

Fig. 7.7 Connectivity between PEs in multi-FPGA board

If 'N' PEs can fit in a FPGA, the data reaches to $(N + 1)$th PE through two FIFOs. The loss of data due to interconnect delays between the pins of different FPGAs is avoided by using the FIFOs. We assume that clocks of the FPGAs on a single board are synchronized.

The compression after various rounds for FPGAs without HEB vis-a-vis with HEBs is shown in Fig. 7.8a. The compression is an important factor which decides the overall speedups as compression increases, the overall speedup increases as the data to be processed by Velvet software decreases. The overall speedups for assembly of swinepox genome with 0.1 % error is 9× obtained using no HEBs and 12× using both FIFO controller HEBs and 1-counter HEBs. The compression in the reads and the FPGA preprocessing time for different FPGA boards is shown in Fig. 7.8b, c respectively. It is interesting to note that the time taken for preprocessing by 1-FPGA board is less than 2-FPGA board in the beginning rounds even though number of FPGAs in 2-FPGA board is greater. This is due to the interconnect delays introduced between the FPGAs in the same board. This does not affect the overall speedup as the compression from the 2-FPGA board is more than the 1-FPGA board. But, as the number of FPGAs increase, the rate of compression per round increases and hence the overall speedup increases as more PEs can be implemented in hardware.

The connectivity between the FPGAs in a single board is also important. It can be seen from Fig. 7.9 that the direct connectivity between FPGAs reduces the execution time. In the board with PCIe connectivity and 2-FPGA clusters, the bandwidth gets shared for the host communication and inter-cluster connectivity. For the genome assembly, a multi-FPGA board with FPGAs connected directly through pins is more favorable. Even though such systems provide more speedups, scalability beyond certain number of FPGAs would be an issue due to PCB design constraints.

7.5 Summary

The methodology described in Chap. 3 was used to accelerate de novo genome assembly. Simulation models at various levels of abstraction were used for design space exploration. Based on the simulation time, we have studied the various parameters that could be explored at different levels of abstraction. Iterative refinement of the algorithm driven by the architecture of FPGAs including HEBs has been presented. In future, as new technologies develop and reduce the transistor size, it is expected

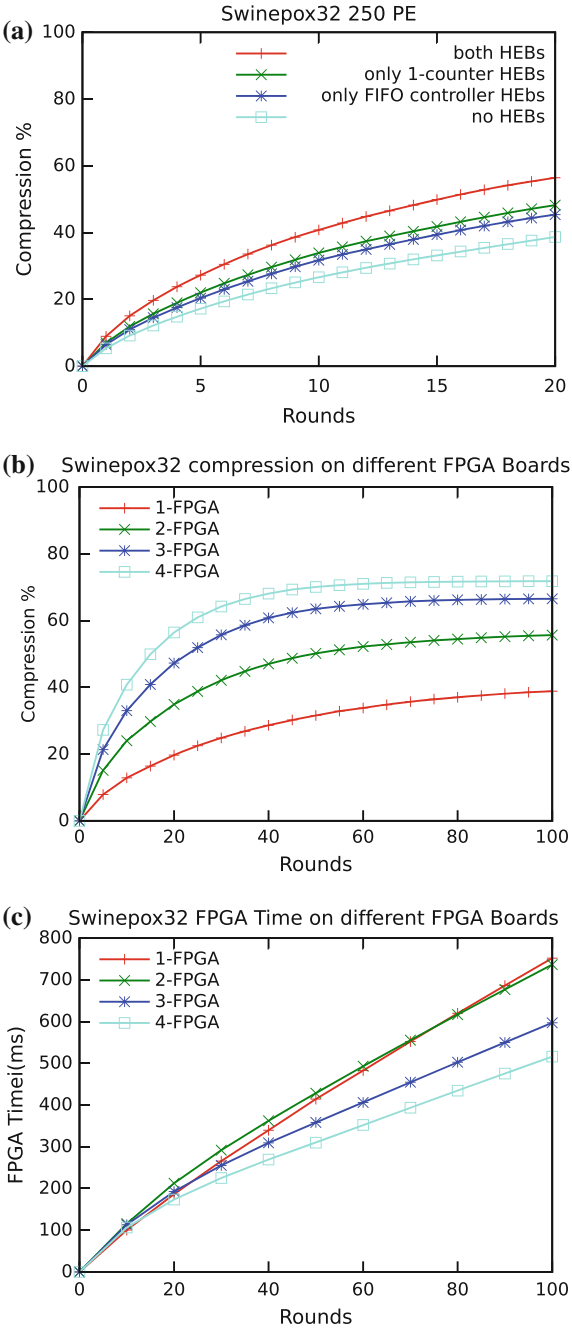

Fig. 7.8 Compression and preprocessing time with muti-FPGA board. **a** The compression comparison with HEBs and without HEBs for 4-FPGA board. **b** The compression on different FPGA boards. **c** The preprocessing time on different FPGA boards with both HEBs

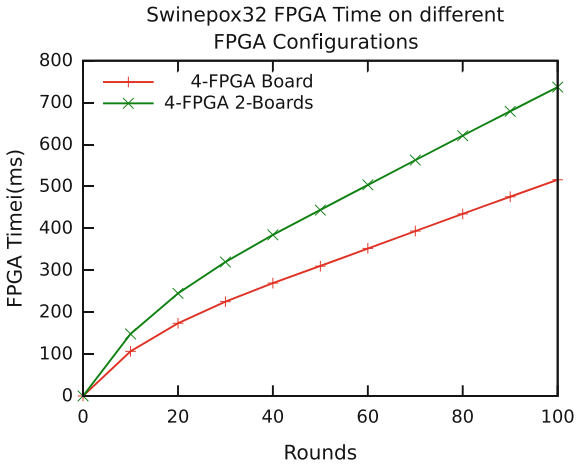

Fig. 7.9 The FPGA preprocessing time for 4-FPGA boards with different connectivity with both HEBs

that the number of HEBs that can be implemented on a single high-end FPGA itself would increase. An evaluation framework such as one presented here with high-level design tools will be useful for an optimal design within the given constraints.

References

1. Graphics, M.: ModelSim Simulator. http://www.mentor.com/products/fpga/model (2015)
2. Hauck, S., Borriello, G.: Pin assignment for multi-FPGA systems. In: Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, 1994, pp. 11–13 (1994)
3. Hauck, S., Borriello, G.: Logic partition orderings for multi-FPGA systems. In: Proceedings of the Third International ACM Symposium on Field-Programmable Gate Arrays, 1995. FPGA '95, pp. 32–38 (1995)
4. Inagi, M., Takashima, Y., Nakamura, Y.: Globally optimal time-multiplexing in inter-FPGA connections for accelerating multi-FPGA systems. In: International Conference on Field Programmable Logic and Applications, 2009. FPL 2009, pp. 212–217 (2009)
5. Jain, S.C., Kumar, A., Kumar, S.: Hybrid multi-FPGA board evaluation by permitting limited multi-hop routing. Des. Autom. Embed. Syst. **8**(4), 309–326 (2003)
6. Jing, C., Zhu, Y., Li, M.: Energy-efficient scheduling on multi-FPGA reconfigurable systems. Microprocess. Microsyst. **37**(6–7), 590–600 (2013)
7. Peterlongo, P., Chikhi, R.: Mapsembler, targeted and micro assembly of large NGS datasets on a desktop computer. BMC Bioinf. **13**(1) (2012)
8. Synopsys: Synopsys Design Compiler. http://www.synopsys.com (2015)
9. Varma, B.S.C., Paul, K., Balakrishnan, M.: High level design approach to accelerate de novo genome assembly using FPGAs. In: 2014 17th Euromicro Conference on Digital System Design (DSD), pp. 66–73 (2014)
10. Xilinx: Xilinx FPGAs, ISE. http://www.xilinx.com (2015)
11. Zhang, W., Chen, J., Yang, Y., Tang, Y., Shang, J., Shen, B.: A practical comparison of de novo genome assembly software tools for next-generation sequencing technologies. PLoS ONE **6**(3) (2011)

Chapter 8
Future Directions

New high-performance applications demand high compute power. These applications have always driven hardware designers to come up with new hardware architectures to execute them efficiently. On the other hand, today's connected world with complex but standard interfaces require extensive software components to be an integral part of any solution. Thus, we are moving toward an era where software and hardware would have to coexist. High performance could be achieved by hardware accelerators augmenting processors. FPGAs as reconfigurable hardware would be able to provide flexible platforms. Design of efficient hardware accelerators becomes essential for meeting the application demands. The recent trends clearly show that huge performance gains can be obtained using FPGAs only by implementing some core functionality as HEBs.

On one hand, speedups can be obtained by mapping computation flow to custom hardware. On the other hand, speedups can also be obtained by changing the application to better suit the computing environment where it is run. In future, usage of custom accelerators to reduce execution time will be common. These custom accelerators should have the right balance between soft (flexibility/programmability) and the hard (custom HEBs) in order to perform better than the existing systems while keeping the traditional flexibility associated with FPGAs.

In this book, we showed a methodology for evaluating benefits of accelerators using FPGAs with HEBs. The methodology allows modifications to be done to the application as well as to the FPGA fabric with a view to evaluate the overall application performance on the system. We used the methodology for accelerating two important bioinformatics applications; protein docking, and de novo genome assembly. The applications were modified to suit the accelerator architecture. The methodology was also used for designing HEBs specific to the application; the butterfly HEB for FTDock application and the 1-counter HEB for de novo genome assembly. The performance estimates were predicted for these fabrics containing the respective accelerator HEBs. Analytical models and high-level simulation models were used to carry out the DSE.

B.S.C. Varma et al., *Architecture Exploration of FPGA Based Accelerators for BioInformatics Applications*, Springer Series in Advanced Microelectronics 55, DOI 10.1007/978-981-10-0591-6_8

The FTDock application was profiled and the most time-consuming part was found to be three-dimensional Fast Fourier Transform (3D-FFT). The algorithm to carry out 3D-FFT was implemented in hardware. Following our methodology, we modified the FTDock application to suit the FPGA architecture. The software version was using double-precision floating point data types. For FPGA implementations, double-precision floating point computations take up large amount of resources and hence we studied the impact of changing the data types to single precision. Results showed very minimal degradation in the output, when single-precision floating point arithmetic was used. We implemented multiple hardware units to carry out FFT computations on FPGA. We show speedups of up to 12× can be achieved using FPGA-based accelerators. We also used our methodology to come up with a custom HEB to achieve more speedups. We showed speedups of upto 17× over FTDock application can be achieved using FPGAs incorporating 'butterfly' HEBs.

Velvet which is a de novo genome assembly software implementation was profiled and found that direct hardware implementation of the software may not possible in FPGAs due the resource constraints. We proposed a method to reduce the input size given to Velvet to achieve speedups, without degrading the output. This preprocessing of reads was done on hardware and the output was given to Velvet to achieve overall reduction in execution time. A streaming based hardware implementation was done, which is suitable to FPGAs. Pre-filters were designed to further reduce the overall execution time. The application for accelerating Velvet was designed using "256-bit 1-counter" HEB and the operating frequency of the FPGA implementation increased from 98 to 185 MHz. High-level models were used to do the algorithm to architecture mapping. The major limiting factor for speedup is Velvet software execution time used for construction of contigs from the intermediate contigs generated by FPGAs. Since, Velvet uses de Bruijn graphs which have very high memory requirement, they could not be implemented in FPGAs. Higher speedups may be possible, if de Bruijn graphs can be constructed in FPGAs. This may be possible by developing new FPGA architectures with significantly larger internal memory or by developing blocks in FPGA having very tight integration with external memory.

Our proposed methodology was used for the selection of blocks to be embedded into the fabric and for evaluating the performance gain that can be achieved by such an embedding. Using our methodology, we showed acceleration of bioinformatics applications using reconfigurable fabrics with accelerator HEBs. As, our methodology considers application details, an in-depth analysis of the application will have to be done to accelerate it. We have considered both application characteristics and accelerator architecture in tandem for achieving speedups for the two applications considered. Since both applications and architectures keep evolving, our methodology can be easily adopted to accelerate other applications by developing new accelerator architectures.

Even though the Design Space Exploration presented in this book is FPGA centric, it can be extended to design heterogeneous accelerators. It can be easily predicted that future chips will contain blocks of varying granularity. The Design Space Exploration presented in this book can be extended to such accelerators where there is a mix of blocks with varying granularity. The coarse-grained units may also execute

instructions and can also be a part of the exploration. Such units in accelerators will add one more dimension to the Design Space Exploration, which would be based on a retargetable compiler for such fabrics. A systematic exploration methodology such as ours can be extended to design future chips, which will allow the use of silicon area efficiently by providing programmability without compromising much on the performance.

The major directions in which this work can be extended is as follows:

1. **Automating Design of Custom Accelerators**
   We have studied and shown that reconfigurable fabrics augmented with hard embedded blocks provide performance benefits. The methodology can be extended for automatic generation of the hard embedded blocks in FPGAs. The process of automatic generation of these blocks has many challenges. The first challenge is to come up with profiling tools to asses the software applications from a particular domain. Present profiling tools generate data for a architecture on which the tool is installed. The tool which takes architecture description as an input and gives the profile information for that specific architecture would be beneficial. Work that has been done for extending the instruction set with custom instructions in ASIP space could be useful as a starting point.
   Coupled with this, automated generation of inputs for high-level synthesis tools enable generation of good, configurable coarse-grained blocks to be embedded in the FPGA. There are challenges at the low-level hardware design which has to be studied. Tools have to be developed for placement of these hard embedded blocks and more importantly a suitable interconnect architecture. These have to be done in conjunction with the design of distributed memory architecture of the fabric. These coarse-grained hard embedded blocks reduce the flexibility offered by the reconfigurable fabric. There has been little work in defining the flexibility parameter in the context of tradeoffs involved with the hard embedded blocks. This metric would enable systematic exploration of the fabric design space.
2. **Heterogeneous Reconfigurable Systems**
   There is a need to develop new systems with heterogeneous cores to process and analyze big data. Such systems should be architected to allow scalability and be able to cater to the requirements of various applications. Reconfigurable accelerators provide flexibility that is critical for evolving applications. Designing effective models for performance estimation for such systems is a challenge. The problem is complex as both software application and the hardware parameters can be varied simultaneously. This work can be used for architecting these systems and developing new ways to carry out such a design space exploration. The tasks involved in this process will be to develop simulators and high-level models for doing the performance estimates. Once the architecture is decided, building the actual platform and doing an efficient hardware implementation would be a challenge. The energy and power consumption of such devices will have to be studied and modifications to the systems have to be done. Tradeoffs based on different metrics will also have to be evaluated.

GPGPUs have now been established as very effective platforms for applications with significant SIMD kernels. Partitioning and mapping applications on architectures with CPU cores, GPUs as well as reconfigurable logic is becoming a reality. Effective performance enhancements with a power budget would require use of HEBs as a part of reconfigurable logic and thus the need for exploration of such heterogeneous design space.

Index

B.S.C. Varma et al., *Architecture Exploration of FPGA Based Accelerators for BioInformatics Applications*, Springer Series in Advanced Microelectronics 55, DOI 10.1007/978-981-10-0591-6