Partial Reconfiguration Implementation on Fluid Dynamics Computation Using an FPGA

Mohamad Sofian bin Abu Talip

A dissertation submitted in partial fulfillment of
the requirements for the degree of

DOCTOR OF PHILOSOPHY

School of Science for Open and Environmental Systems
Graduate School of Science and Technology
Keio University

September 2013

Preface

The field of high performance scientific computing lies at the crossroads of many disciplines and skill sets. Scientific computation from an application context makes some acquaintance with physics and engineering sciences. Then, problems in these application areas are solve using scientific processes, and the use of computers for numerical analysis to produce quantitative results. An efficient implementation of the practical formulations of the application problems requires some understanding of computer architecture, both on the CPU level and on the level of parallel computing.

One of the high performance computing (HPC) applications is computational fluid dynamics (CFD). In aerospace industry, CFD is used as a common design tool. It presents scientific computation methods to analyze fluid behavior for designing aircraft components such as engines and wings. Therefore, software packages for CFD are needed for aeronautical engineers and researchers. However, enormous floating-point calculations cause a long execution time required to simulate complete aeronautics configurations. It remains as a bottleneck in the design flow of new structures for the aircraft design. Thus, reducing the total execution time for aerodynamics analysis is one of the important challenges of current research in this field.

Recent advances in Field Programmable Gate Array (FPGA) technology make reconfigurable computing using FPGAs an attractive platform for accelerating scientific applications. The readily availability and high-power efficiency of high-density FPGAs make them attractive to the HPC community. Since their invention in the mid-1980s, FPGAs have been used to accelerate high performance applications on custom computing machines. Under such circumstance, a new type of computational systems is being focused for allocating a part of scientific operations to dedicated hardware in order to achieve both low-cost and high-performance.

In this thesis, two CFD codes are studied: UPACS (*Unified Platform for Aerospace Computational Simulation*) and FaSTAR (*Fast Aerodynamics Routines*) software packages. The problems of these codes are hard to be executed in parallel machines because of their irregular and unpredictable data structure. In addition, a single FPGA is not enough for the software packages because the whole modules are very large. Exploiting reconfigurable hardware with their advantages to make up for the inadequacy of the existing high performance computers has gradually become the solutions. Instead of using a large number of chips, partially reconfigurable hardware available in recent FPGAs is explored for these applications.

With the above aim, this thesis explores scientific computation of CFD applications and implements the target subroutines in FPGA by utilizing partial reconfiguration technology. The goal of this work is to achieve high performance compared to microprocessor execution and to clarify the relationships among hardware resources utilization, configuration time and performance according to the evaluation results.

UPACS developed by JAXA (*Japan Aerospace Exploration Agency*) is one of CFD packages to simulate compressible flow using multi-block grids. MUSCL (Monotone Upstream-centered Schemes for Conservation Laws) scheme in UPACS is chosen as a target subroutine, since it is used twice in core routine of UPACS. Partial reconfiguration is applied to the flux limiter functions (FLF) in MUSCL. Two types of partially reconfigurable design are implemented that are static and dynamic reconfigurations. Four FLFs are implemented for Turbulence MUSCL (TMUSCL) and eight FLFs are for Convection MUSCL (CMUSCL). In statically reconfigurable design, the implementation has successfully reduced the resource utilization by 44% to 63%. Total power consumption was also reduced by 33%. Configuration speed was improved by 34 times faster as compared to full reconfiguration method. In dynamically reconfigurable design, the implementation has successfully reduced resources utilization by 60%. Total power consumption was also reduced by 29%. Configuration speed was improved by 15 times faster compared to fully reconfiguration method. Both implementations also achieved at least 17 times speed-up compared with the software execution.

FaSTAR, another CFD package developed by JAXA, supports several solvers and adopts unstructured mesh as its grid form. The advection term computation module in FaSTAR is chosen as a target subroutine, which a time-consuming and large function. Therefore, a partially reconfigurable flux calculation scheme that would fit in a single FPGA was proposed. The flux computational module was developed and five flux calculation schemes were implemented as reconfigurable modules, these were: Roe, HLLE, HLLEW, AUSM$^+$-up, and SLAU. The implementation of this module has the advantages of saving up to 62.75% of resource and increasing the configuration speed by a factor of 6.28. Performance evaluation also shows that 2.65 times more acceleration was achieved compared to the Intel Core 2 Duo at 2.4 GHz.

Finally, we summarize our proposed implementation method, utilizing the partial reconfiguration technique for saving hardware resources and achieving faster performance in comparison with software execution. Based on the above, we discuss how the current work could be explored further in order to develop the scientific applications of FPGAs.

Acknowledgments

In the name of Allah, the Beneficent, the Merciful.

Many great people has helped and assisted me in so many ways during my study. I take this opportunity to express my gratitude to the people who have been instrumental in the successful completion of this thesis.

First and foremost, I would like to express my deepest appreciation to my supervisor, Professor Hideharu Amano, for his continuous support and guidance throughout my study. I have been fortunate to have such a great advisor who has outstanding skills and immense knowledge. He has always guide me to the right path when I faltered in my research.

Furthermore I would also like to acknowledge with much appreciation the crucial role of my doctoral committee members. I am grateful to Professor Hideo Saito, Professor Issei Fujishiro, and Associate Professor Hiroaki Nishi for their careful reviews and valuable comments. Their constructive criticism and excellent advice have significantly improved this thesis.

It has been a wonderful experience to participate in the Aerotech group and collaborate with researchers from other institutions. My sincere thanks are due to Mr. Naoyuki Fujita (JAXA) for constant support and encouragement, Asst. Prof. Yasunori Osana (University of the Ryukyus) for detailed reviews on my manuscript before each submission. I am also grateful to Mr. Kenta Inakagata (Renesas), Mr. Takayuki Akamine (Toshiba) and Mr. Mao Hatto, all of their creative vision and constructive ideas have been crucial to the success of this research. Special thanks to Ms. Yamada Naoko (Toshiba) for fruitful discussions and generous support to teach me about partial reconfiguration methodology. I would also like to wish best of luck to Mr. Naru Sugimoto and Mrs. Dipikarani Mishra to continue the journey of exploration in this research group.

I would also like to thank all Amano lab members, past and present, for providing me daily inspirations and so much fun while studying abroad. I would like to extend my deep thanks to my fellow doctoral candidates: Mrs. Amila Akagic, Mr. Zhang Hao and Mr. Takaaki Miyajima for sharing pleasures and sorrows as research assistant of GCOE program. Thank you for exchanging ideas and providing me with many insightful technical discussions.

I appreciate the financial supports from Ministry of Higher Education Malaysia and University of Malaya, Malaysia for give me a great opportunity to pursue my study in Japan. I would also like to thank Keio Leading-Edge Laboratory (KLL) of Science & Technology for providing research grant during three years of my study. This work was also supported in part by a Grant-in-Aid for the Global Center of Excellence for High-Level Global Cooperation for Leading-Edge Platform on Access Spaces from the Ministry of Education, Culture, Sport, Science and Technology in Japan.

In addition, I am also very grateful to my friends who have shared enjoyable time at Keio University. Thank you to all my fellow Malaysians and other international students who have shared our precious experience in Japan. I will always remember and greatly value our friendship forever.

Last, but not least, I wish to express my heartfelt thanks to my parents, parents-in-law, my lovely wife and my two daughters for all of the love and sacrifice that they have done for me. Their support has been a great source of encouragement and strength.

Mohamad Sofian bin Abu Talip
Yokohama, Japan
August 2013

Contents

List of Tables

List of Figures

Abbreviations and Acronyms

AES	Advanced Encryption Standard
ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
AUSM	Advection Upstream Splitting Method
BUFG	Global Clock Buffer
BUFH	Horizontal Clock Buffer
BUFIO	Input Output Buffer
BUFR	Regional Clock Buffer
CDR	Convection Diffusion Reaction
CFD	Computational Fluid Dynamics
CLB	Configurable Logic Block
CMT	Clock Management Tiles
CMUSCL	Convection MUSCL
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DCM	Device Clock Manager
DRAM	Dynamic Random Access Memory
DRC	Design Rule Check
DSP	Digital Signal Processing
EEPROM	Electrically Erasable Programmable Read Only Memory
EPROM	Erasable Programmable Read Only Memory
FaSTAR	Fast Aerodynamics Routines
FIFO	First-in First-out
FLF	Flux Limiter Function
FPGA	Field Programmable Gate Array
FSB	Front Side Bus
GFLOPS	Giga Floating-point Operations Per Second
GNU	GNU's Not Unix
GPP	General Purpose Processor
GPU	Graphics Processing Unit

GUI	Graphical User Interface
HDL	Hardware Description Language
HLL	Harten-Lax-van Leer
HLLE	Harten-Lax-van Leer-Einfeldt
HLLEW	Harten-Lax-van Leer-Einfeldt-Wada
HLS	High Level Synthesis
HPC	High Performance Computing
I/O	Input Output
IBUF	Input Buffer
ICAP	Internal Configuration Access Port
ICC	Inter Connect Controller
IOB	Input Output Block
IP	Intellectual Property
JAXA	Japan Aerospace Exploration Agency
JTAG	Joint Test Action Group
LBM	Lattice Boltzmann Method
LSI	Large Scale Integration
LUT	Look Up Table
MAC	Multiplication and Accumulation
MAP	Map
MDGRAPE	Molecular Dynamics Gravity Pipe
MFGS	Matrix Free Gauss Seidel
MMCM	Mixed Mode Clock Managers
MPGA	Mask Programmable Gate Array
MPI	Message Passing Interface
MUSCL	Monotone Upstream-centered Schemes for Conservation Laws
NACA	National Advisory Committee for Aeronautics
NGD	Native Generic Database
OBUF	Output Buffer
OTN	Optical Transport Network
PAL	Programmable Array Logic
PAR	Place and Route
PC	Personal Computer
PCI-X	Peripheral Component Interconnect Extended
PDB	Parameter Data Base
PE	Processing Element
PLA	Programmable Logic Array
PLD	Programmable Logic Device

PLL	Phase Locked Loop
PMCD	Phase Matched Clock Divider
PR	Partial Reconfiguration
PROM	Programmable Read Only Memory
RAM	Random Access Memory
RM	Reconfigurable Module
ROM	Read Only Memory
RP	Reconfigurable Partition
RTL	Register Transfer Level
SDR	Software Defined Radio
SDRAM	Synchronous Dynamic Random Access Memory
SIMD	Single Instruction Multiple Data
SLAU	Simple Low-Dissipation Scheme of AUSM-Family
SO-DIMM	Small Outline Dual In-line Memory Module
SoC	System-on-Chips
SPARC	Scalable Processor Architecture
SRAM	Synchronous Random Access Memory
TCK	Test Clock
TFLOPS	Tera Floating-point Operations Per Second
TMUSCL	Turbulence MUSCL
UART	Universal Asynchronous Receiver Transmitter
UPACS	Unified Platform for Aerospace Computational Simulation
VLSI	Very Large Scale Integration
XPE	Xilinx XPower Estimator

Chapter 1

Introduction

1.1 Background

Until the early 2000s, general purpose single-core CPU-based systems were the processing systems of choice for high performance computing (HPC) applications. They replaced exotic supercomputing architectures because they were inexpensive, and performance scaled with frequency in line with Moore's Law. Presently, HPC industry is going through another historical step change. General-purpose CPU vendors changed course in the mid-2000s to rely on multicore architectures to meet high performance demands. The technique of simply scaling a single-core processor's frequency for increased performance has run its course, because as frequency increases, power dissipation escalates to impractical levels.

The shift to multicore CPUs forced application developers to adopt a parallel programming model to exploit CPU performance. Even using the newest multicore architectures, it is unclear whether the performance growth expected by the HPC end user can be delivered, especially when running the most data-intensive and computing-intensive applications [1]. CPU-based systems augmented with hardware accelerator as co-processors are emerging as an alternative to CPU-only systems [2].

This has opened up opportunities for accelerators like Graphics Processing Units (GPUs) [3], Field-Programmable Gate Arrays (FPGAs) [4], and other accelerator technologies [5], to advance HPC to previously unattainable performance levels. HPC comprises a class of systems typically used by scientists, engineers, and analysts. This sysmtems simulate, model applications and analyze large quantities of data. Typical systems range from server farms to big supercomputers. Table 1.1 shows a summary of different industries and HPC applications. These applications are data-intensive and computing-intensive. There are in constant need of increased computing power and bandwidth to memory.

Nowadays, CFD (Computational Fluid Dynamics) has growth attention from HPC community. CFD has been widely utilized in the design and optimization of fluid flow applications. In aerospace industry, CFD is a cost-effective design tool for aircraft components such as jet engines and wings [6]. It presents methods to solve and analyze problems of the physical phenomena of fluids involving

Table 1.1: Typical applications for high performance computing.

Industry	Sample Applications
Government Labs	Climate modeling, Nuclear waste simulation, Disease modeling and research
Defense	Video, audio, data mining and analysis for threat monitoring, Pattern matching, Image analysis for target recognition
Financial services	Options valuation and risk analysis of assets
Geosciences and engineering	Seismic modeling and analysis, and reservoir simulation
Life sciences	Gene encoding and matching, Drug modeling and discovery

fluid flow on discrete space and time. Thus, software packages for CFD are needed for aeronautical engineers and researchers.

Ground test facilities do not exist in all flight regimes covered by such hypersonic flight. No wind tunnels that can simultaneously simulate the higher Mach numbers and high flow-field temperatures to be encountered by trans-atmospheric vehicles, and the prospect for such wind tunnels in the twenty-first century is not encouraging. Hence, CFD has become the major player in the design of such vehicles [7]. In addition, compressible flow simulations are of vital importance to the aerospace engineering community, which will always seek higher resolution and more accuracy creating a demand for faster codes and making the use of high performance computing strategies invaluable.

On the other hand, FPGAs continue to be a platform of choice for designing programmable systems. Due to their inherent flexibility, FPGAs have been used in programmable solutions such as serving as a prototype vehicle as well as being a highly flexible alternative to application-specific integrated circuits (ASIC) [8,9]. Advancements in silicon, software, and IP (Intellectual Property) have proven FPGAs to be the ideal solution for accelerating applications on high-performance embedded computers and servers [10, 11].

Presently, reconfigurable systems using FPGAs have been utilized for acceleration of specific applications including bio-informatics, digital image processing, finance and others [12–14]. Even though the early reconfigurable systems did not focus on large scale numerical scientific application, the use of FPGAs for such areas has been growing remarkably because of the rapid performance improvement of modern FPGAs with a large number of configurable logic blocks, memory blocks and embedded multipliers. However, although some research works using FPGAs achieved significant speed-up ratio to the software [15, 16], targets were simple programs rather than practical software packages.

There is a new technology in FPGA design called partial reconfiguration, which is the process of changing a portion of reconfigurable hardware circuitry while the other part is still running. In FPGA, partial reconfiguration is the modification of an operating design by loading a partial configuration file [17]. FPGA technology provides the flexibility of on-site programming and re-programming without going through re-fabrication with a modified design. Partial reconfiguration takes this flexibility one step further, allowing the modification of an operating FPGA by loading a partial bit file. After a full bit file configures the FPGA, partial bit files can be downloaded to modify reconfigurable regions in the FPGA without compromising the integrity of the applications running on those parts of the device that are not being reconfigured.

They have been few attempt to make use of partial reconfiguration technology in embedded and aerospace applications. Recently, in computer security applications, partial reconfiguration is applied in AES (Advanced Encryption Standard) algorithm implementation [18]. It also had been used to accelerate video processing in driver assistance system [19]. Also, a few researches had been done on an aerospace applications using partial reconfiguration method. LaMeres *et al.* [20] designed and prototyped the computing architecture which dynamically reconfigures the system depending on the environment. Another work has proposed the SoCWire architecture verification, test and results on network on chip for safe dynamic partial reconfiguration in spaces applications [21].

In the meantime, a typical simulation platform in the aeronautics industry consists of a CFD specific software application, normally written in a high-level language. Although it is a convenient tool for aerodynamics analysis, it sometimes takes several days or weeks when an analytical area grows larger [22]. This is mainly caused by low parallel processing efficiency accompanied with pointer links and a complicated memory access patterns. In CFD package, the increasing demands for accuracy and simulation capabilities produce an exponential growth of the required computational resources.

Our approach in this work is to make use of partial reconfiguration technology available in recent FPGAs to accelerate and improve performance of CFD software packages in a reconfigurable hardware.

1.2 Objective

The objective of this thesis is to study CFD applications on FPGAs using partial reconfiguration. The study is based on the viewpoint that future CFD applications would integrate FPGA as a heteregenous architecture or as an accelerator. With all of the FPGA advantages compared to CPU, exploring partial reconfiguration methodology would be an added advantage to CFD researchers. To achieve this goal, two CFD software packages provided by JAXA (Japan Aerospace Exploration Agency) are studied: UPACS (Unified Platform for Aerospace Computational Simulation) and FaSTAR (Fast Aerodynamics Routines) software packages.

In UPACS, the goal is to improve the performance by using FPGAs as a reconfigurable platform. However, the whole UPACS package is too large to implement in a single FPGA, and often larger than the size of the target platform. Morishita *et al.* tried to implement core subroutines in UPACS which has complicated memory access into FPGAs [23]. Although the performance can be improved, it is found that the target requires too vast resources to implement on a small number of FPGAs. Inakagata *et al.* also have tried multiple FPGAs as an option to resolve this problem [24]. However, we want to save the number of FPGAs to be used, and also to reduce power by introducing partial reconfiguration provided in recent FPGAs. Since UPACS consists of various solvers, not all of them are needed to solve a target application. In this study, we introduce this mechanism into flux limiter functions (FLF) in MUSCL (Monotone Upstream-centered Schemes for Conservation Laws) scheme available in UPACS package.

In FaSTAR, total required hardware often becomes large if all functions of the package are implemented on FPGAs. We can avoid it by providing a subset design with only required functions in advance. However, the number of designs to be prepared in advance becomes large for complicated application package. Previously, Akamine *et al.* have proposed out-of-order mechanisms to cope with unstructured mesh for efficient execution of FaSTAR [25]. By introducing the partial reconfiguration, we can quickly prepare the appropriate design for the user. This trial is the first step of using partial reconfiguration technique, and it has a possibility to extend the application of the partial reconfiguration to a large scientific application package. In addition, our objective is to improve the performance of target subroutine in FaSTAR by implementing in FPGA. Utilizing partial reconfiguration is a promising approach to achieve this goal with a small amount of hardware.

1.3 Contribution

In this thesis, we have studied fluid dynamics computation on an FPGA using partial reconfiguration technique. Our main contributions can be highlighted as follows:

1) In UPACS explorations, we introduce a reconfigurable flux limiter functions in MUSCL scheme. Core routine in UPACS consumes 70% of the total execution time, and only MUSCL scheme is used twice. Moreover, MUSCL scheme contribute to 23.6% from that fraction. Therefore, speed-up of MUSCL scheme has been a priority before the other subroutines. We make an analysis of 6 flux limiter functions (FLF) which populate the design space of MUSCL scheme, and data dependencies between them. Besides, we also pipeline the MUSCL datapath to increase the throughput of the system when processing a stream of data. Two designs are implemented that are static reconfiguration [26] and dynamic reconfiguration [27]. Quantitative comparison of resource utilization, power consumption, and configuration speed between partial and full reconfiguration are performed. We also analyze the performance speed-up over a CPU execution.

2) In FaSTAR explorations, we study an advection term computation core routine. Advection term subroutine has consumed more than 30% from the total time of FaSTAR execution. Therefore, we introduce a partially reconfigurable flux calculation schemes in advection term computation to save the hardware resources. We make an analysis of 5 flux calculation schemes and deploy in reconfigurable region. In addition, we introduce a single stage pipeline in static module to provide multistage computations. In each reconfigurable module, data level parallelism is introduced to reduce total clock cycles per operation. Quantitave comparison of hardware resource utilization and configuration speed is performed. We also analyze the performance speed-up achieved compared to software over CPU execution.

1.4 Thesis Organization

The thesis is organized as illustrated in Figure 1.1. This chapter gave a brief overview of high performance computing, CFD, FPGA as well as partial reconfiguration technology. Thesis objectives and contributions were also highlighted. The next chapter gives an overview about computational fluid dynamics. Conventional systems used to executed CFD applications are discussed with several examples. Then, is followed by explanation regarding UPACS and FaSTAR software packages. Chapter 3 is about FPGA and partial reconfiguration as well as examples of CFD applications utilized FPGAs technology. Chapter 4 studies target subroutine in UPACS that is MUSCL scheme and their implementation in an FPGA using partial reconfiguration. Performance evaluations are performed and discussed. Chapter 5 studies target subroutine in FaSTAR that is advection term computation and their implementation in an FPGA using partial reconfiguration. Performance evaluations are also presented and analyzed. Finally, Chapter 6 summarizes our proposed reconfigurable designs for CFD applications. Based on the above, we make a discussion on the expanding of partial reconfiguration technology to scientific computing mainly to CFD applications.

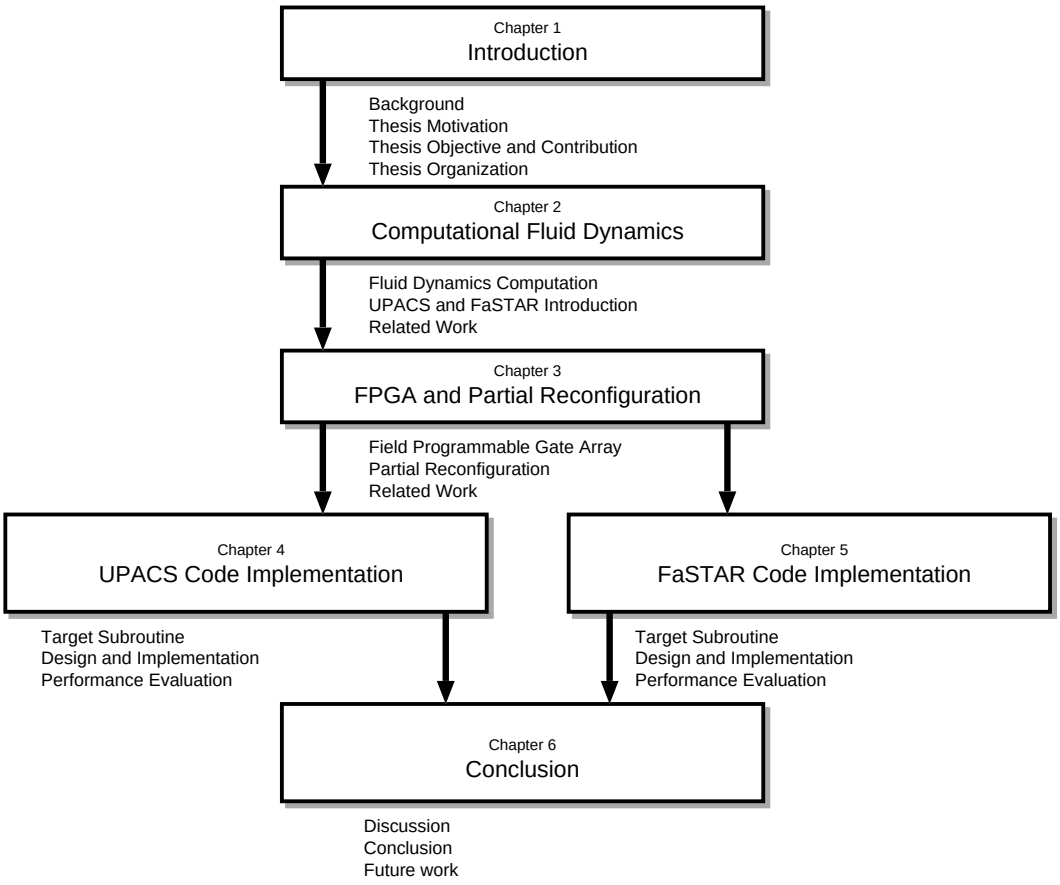

Figure 1.1: Thesis organization

Chapter 2

Computational Fluid Dynamics

Modeling, simulation and optimization using computing tools are the core approach nowadays in the whole discipline of science complementary to experiment and theory. In recent years, numerical analysis has been used in areas of scientific computations, and the complexity of the algorithm is increasing.

Computational fluid dynamics (CFD) is one of the key areas of numerical analysis. Today, CFD codes are becoming a commodity used regularly for engineering analysis and even in the design process. Commercial codes have now been steadily employed, not only in industry but also in academia and government laboratories. Indeed, CFD is further reinforced by the rapid advancement of ever faster and larger memory computer processors, making computation of complex geometry and flow physics affordable. Hence, CFD no longer belongs to experts, but in fact is leveraged largely by practitioners.

This chapter provides an overview of CFD. Then, following the conventional system of CFD applications, a few example systems are given. After that, we introduce the UPACS and FaSTAR software package. Finally, this chapter concludes with a summary.

2.1 Overview

CFD is the science of predicting fluid flow, heat transfer, mass transfer, chemical reactions, and related phenomena by solving the mathematical equations, which govern these processes using numerical methods. Thus it provides a qualitative and quantitative prediction of fluid flows by means of mathematical modeling, numerical methods and software tools. Fluid flows encountered in everyday life include heating, ventilation and air conditioning of buildings, combustion in automobile engines, other propulsion systems and etc. In addition, we had many researchers contributing to fluid mechanics and its solutions, mathematical models and formulating the models across the centuries. We are now living in a time wherein the software packages have exploited all this work with the advancement of computational speed.

2.1.1 Analytical Fluid Dynamics

Before the high-speed computers of today arrived, the usual processes followed by researchers were using manual or hand calculations with some approximations. With some assumptions and approximations, researchers generally made a free body diagram of the target object, simplified the complex 3-D geometry into simple 1-D or 2-D analysis, and ended up doing an integration of these equations. Researchers also set up the constraints in the form of initial and boundary conditions to find constants for integration. Finally, values at discrete points are obtained, where could end-up having results getting plots.

Now for a simple case it might be easy to apply this method but not in cases of complex problem. However, not just the complex geometry is a problem but also more often there are limited in defining the exact physical conditions in terms of concerned mathematical equations. To solve the general equation on a real geometry is a challenge. If we do not want to face this, one can have another approach, called Experimental Fluid Dynamics (EFD).

2.1.2 Experimental Fluid Dynamics

Here researchers have a scale down model, wherein create a prototype model and perform appropriate experiments. It will test the model under conditions that would reflect exactly what would happen in reality. It is often called wind tunnel experiments. Further a lot of probe points are introduced for data collection thereby introducing disturbances in the flow itself. Also it is not so always easy to make an exact prototype of the real problem, and there are problems with cost and feasibility too.

So we can see that both methods have some limitations, analytical approach with respect to complex geometries and physics capturing while experimental approach with issues in time, cost and feasibility.

Thus, we focus on CFD wherein we have the mathematical model whereas the physics clearly defined, and these physics equations that are usually the partial differential equations, are solved through numerical methods. With the assumptions that some higher order terms are neglected we can use computer to solve them and end-up having a CFD result, a result that is numerical solution of our physics. This has many advantages. We have a low cost simulation, and we are able to do a lot of analysis within a short period of time, as we do not have to actually make a physical model of our study. With approximations in some parameters, we can study the problem using the high-speed computer, which previously takes about hours or days time to obtain the results. So in brief, with the CFD approach we can have a low cost solution and reliable information as compared to any other approach. Also although CFD does not replace the measurements completely, the amount of experimentation and the overall cost can be significantly reduced. Table 2.1 summarizes the comparison of EFD and CFD to solve flow problems.

Table 2.1: Comparison of EFD and CFD for fluid dynamics.

EFD	CFD
Quantitative acquisition of flow phenomena using measurements • at a limited number of points and time instants • for a laboratory-scale model • for a limited range of problems and operating conditions Error sources: measurement errors, flow disturbances by the probes	Quantitative prediction of flow phenomena using CFD software • for all desired quantities • with high resolution in space and time • for the actual flow domain • for virtually any problem and realistic operating conditions Error sources: modeling, discretization, iteration, implementation

2.2 CFD Process

Overall CFD process is a 3 step procedure:

- pre-processing
- numerical solution
- post-processing

Pre-processing step starts with determining the equations to be solved. Then, geometry is considered to define the domain of interest, which is then divided into segments, in the mesh generation step and the problem is set up defining the boundary conditions upon generating a computational mesh. However, it depends upon the desired output of the simulation and the capabilities of the solver.

Once the problem is set up defining the boundary conditions, we solve it with software on a computer. As mentioned in the previous section, it can also be done by hand calculations, but would take long time. Nevertheless, user intervention is necessary in this step. Users need to set under-relaxation factors and input parameters, whilst an understanding of discretization methods and internal data structures is required in order to supply mesh data in an appropriate format and to analyze output.

The raw output of the solver is a huge set of numbers corresponding to the values of each field variable at each point of the mesh. In post-processing step, this must be reduced to some meaningful subset and manipulated further to obtain the desired main outputs. We may analyze them by means of color plots, contour plots, and appropriate graphical representations and we can generate reports. Commercial packages routinely provide plotting tools to visualize the flow and analysis tools to extract and manipulate data.

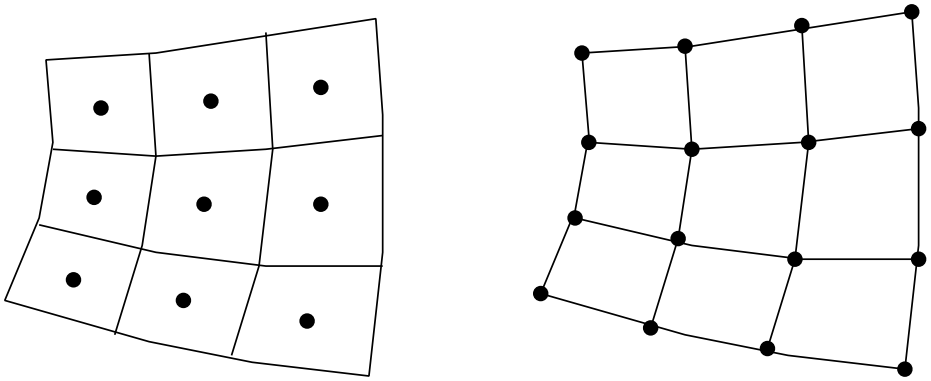

Figure 2.1: Cell-centered storage on the left and cell-vertex storage on the right.

2.2.1 Mesh Structure

The purpose of the grid generator is to decompose the flow domain into control volumes. The primary outputs are cell vertices and connectivity information. Precisely the location nodes are relative to the vertices depending on the solver usage, for example, cell-centered or cell-vertex storage as shown in Figure 2.1. The unknowns may be located at vertices of the cells, or at the centers of the cells. These two multigrid methods differ in the location of the nodes in the grids and in the transfer operators.

Structured meshes are those whose control volumes can be indexed by (i, j, k) for $i = 1, ..., n_i$, $j = 1, .., n_j$, $k = 1, ..., n_k$, or by a group of such blocks, that is multi-block structured grids. Structured grids can be Cartesian or curvilinear, usually body-fitting. In the former, grid lines are always parallel to the coordinate axes. In the latter, coordinate surfaces are curved to fit boundaries of a target object. The grid is described as orthogonal if all grid lines cross at 90°. Such examples include Cartesian grids and cylindrical or spherical polar grids.

Unstructured meshes can accommodate completely arbitrary geometries. Unlike structured grids, the cell at location n in memory may have no physical relation to the cell next to it at location $n + 1$. This means that an unstructured grids allows a lot of freedom in constructing a CFD grid. However, there are significant penalties to be paid for this flexibility, both in terms of data structures and solution algorithms. Grid generators and plotting routines for such meshes are also very complex.

2.2.2 Simulation Process

To understand the simulation process and the steps involved in it let us consider an example of a flow through a pipe bend. Figure 2.2 gives a series of the steps that would be involved in its analysis. For a fluid flow through a pipe bend, we have the geometry built up, separated into smaller fragments/segments, called a mesh. With this mesh we actually define our probe-points where we want the analysis to be done. We then define the boundary conditions to get a unique solution with a computer. The results obtained give us a lot of data along these probe points that are then post-processed with visualization tools to analyze the results.

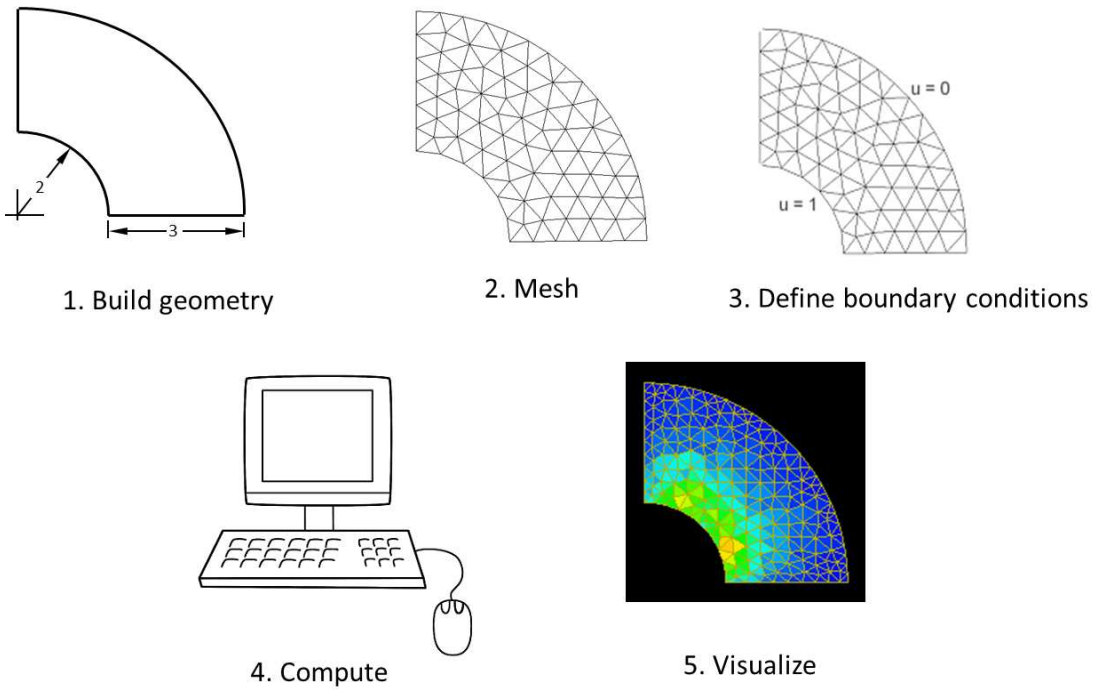

Figure 2.2: Example of CFD simulation process of a flow through pipe bend.

As shown above, it turns out that selection of model construction and a numeric solution are needed for the analysis of fluid. In CFD, according to a problem, a new analysis program is created each time in many cases, and there was usually a problem that construction of simulation environment also takes time. However, in order to solve this problem, JAXA is developing a general-purpose CFD packages called UPACS and FaSTAR.

2.3 UPACS

UPACS (Unified Platform for Aerospace Computational Simulation) is a CFD software package developed by JAXA [28,29]. It is an application with a highly scalable environment. Mainly, UPACS includes the following features:

- Multi-block structured grid
  In order to realize complicated form, flexible analytic space is realizable by preparing two or more analytic spaces separately rather than using only a single analytic space, along with connection information.

- Parallel computing by MPI
  By specifying the number of processors in advance, it is possible to perform parallel processing in units of blocks. There is also benefit when the MPI is used, architecture independence is high.

- Separate processing of the block and numerical solution
  Process of numerical solution performs only one block in UPACS. When performing processing on different blocks, it is performed by passing it to the block processing function. Because of this structure, a new block is easy to be added in the calculation method.

- Encapsulation of the calculation procedure and data structure
  Calculation procedures and variables are encapsulated. The calculation is performed by calling the process in the form of a function from the outside. In addition, it has a structure that can be easily modified when it becomes necessary to change the code thereby.

- Developed in Fortran 90
  It is possible to incorporate the idea of object-oriented, and is intended to maintain scalability.

As stated above, UPACS can add various numeric solutions, but selection of a solution can be simply performed simultaneously using a GUI tool called PDB (Parameter Data Base) Editor by users. Table 2.2 shows examples of parameters which are available and can be chosen using PDB Editor.

2.4 FaSTAR

FaSTAR (Fast Aerodynamics Routines) is another CFD software package developed by JAXA to simulate compressible flow using unstructured grid [30]. FaSTAR is an application with high extensibility that can be used in parallel computing environment. FaSTAR has the following features:

- Unstructured grid
  Polygons and cubes, such as a triangle and a quadrangle, and the polyhedron can be put in order freely, and complicated form can be realized. Grid data is automatically generated, and it can perform a simulation in high accuracy, without accuracy correction.

- Parallel computing by Zoltan and MPI
  In the environment of a supercomputer or a cluster computer, it comes out by specifying the number of processors beforehand to carry out parallel processing per grid area. When MPI is used, there is also a merit that architecture independence is high. It is necessary to divide the whole grid into two or more domains in that case, and a division algorithm called Zoltan plays the role.

- Development by Fortran 90
  It is possible to take in an object-oriented idea and, thereby, the extensibility and conservativeness of FaSTAR are improved.

Table 2.2: Examples of the calculation parameters used in UPACS.

Solutions	**Available Subroutines**
Governing Equation	Euler Equation Navier-Stokes Equation Reynolds Averaged Navier-Stokes Equation
Turbulent Flow Model	Baldwin-Lomax Spalart-Allmaras
Convective Term Scheme	Roe Scheme AUSMDV Scheme
Flux Limiter Function	no limiter van Leer van Albada minmod superbee Hemker-Koren
Time Integration	Euler Explicit Method 3^{rd} Runge-Kutta Method Jameson Method Matrix-Free Gauss-Seidel Method
Implicit Method Scheme	1^{st} Euler Method 2^{nd} Euler Method Crank-Nicolson Method

- Separation of the process division of a numeric solution and a grid
  In FaSTAR, the scheme, which reads grid data, and the scheme, which performs a numeric solution, are separated. Therefore, it is easy to change or add a new numeric solution.

- Encapsulation of a data structure and a computational procedure
  A variable and a computational procedure are encapsulated and it calculates by calling processing in the form of a function from the exterior. In case this needs change, it has the code structure which can be understood easily.

As stated above, FaSTAR can add various numeric solutions with a user defined parameter file, and selection of a solution can simply be performed. Table 2.3 shows the solution which is available in FaSTAR. By combining these solutions, the user can perform the simulation without adding correction to a code.

Table 2.3: Examples of the calculation parameters used in FaSTAR.

Solutions	Available Subroutines
Governing Equation	Euler Equation Navier-Stokes Equation Reynolds Averaged Navier-Stokes Equation
Turbulent Flow Model	Spalart-Allmaras Model (SA) Menter k-ω Shear Stress Transport Model (SST) Automatic Wall Processing Capability
Inviscid Flux Calculation Scheme	Roe Scheme HLLE Scheme HLLEW Scheme AUSM$^+$-up Scheme SLAU Scheme
Flux Evaluation	Least-Square Law Green-Gauss Law
Flux Limiter Function	Hishida Limiter Venkatakrishnan Limiter Bart-Jespersen Limiter minmod Limiter
Time Integration	LU-SGS(regular/unsteady, local/global time stepping)
Convergence Acceleration	Multigrid (FAS:Full Approximation Storage) Krylov Law (GMRES)

2.5 Conventional Systems

Improvement in the speed of application has mainly been conventionally performed by a supercomputer, a cluster type computer, and ASIC (Application Specific Integrated Circuit). Moreover, in the past few years, improvement using graphics processing unit (GPU) also gains attention from CFD research community. In this section, we will explain about these conventional systems as well as example systems, which realize improvement in the speed of application. Then, it follows by discussion on GPU as a platform for CFD applications acceleration.

2.5.1 Supercomputer

One of the high-speed techniques of a numeric simulation is using supercomputer. Since the definition of supercomputer changes with times, it is difficult to give a definition uniformly. Typically, a supercomputer is a computer at the frontline of current processing capacity, particularly in terms of speed of calculation.

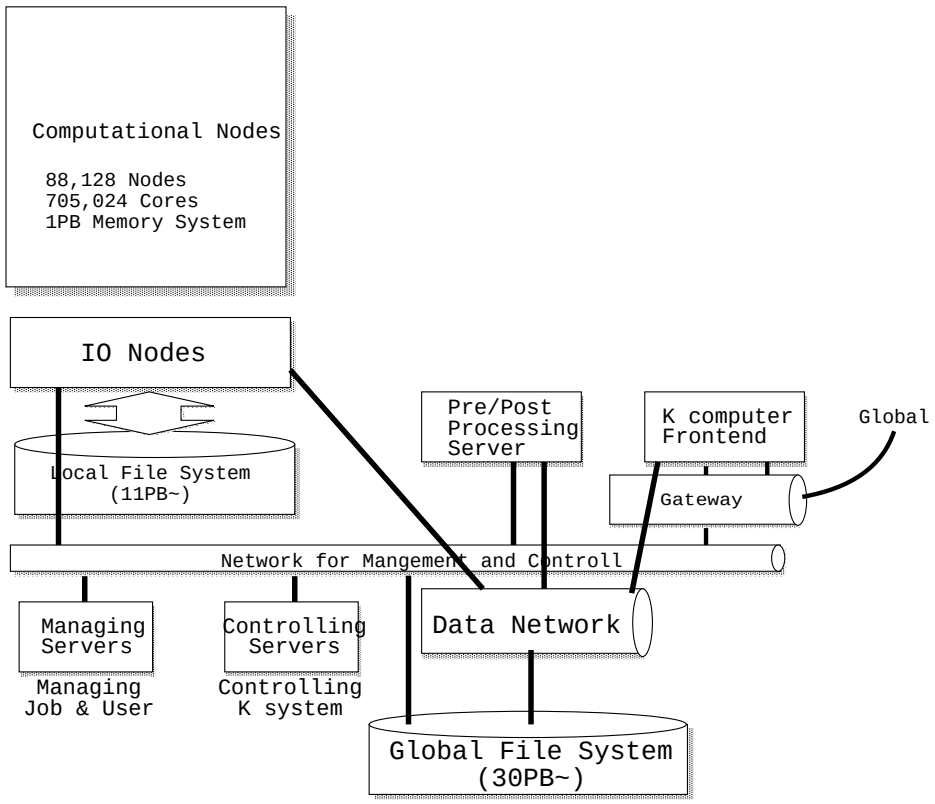

Figure 2.3: K computer system configuration.

One example of a supercomputer is K computer [31]. System configuration of the K computer is shown in Figure 2.3. The K computer is a distributed memory supercomputer system consisting of 82,944 compute nodes and 5,184 I/O nodes, two kinds of files systems, control and management servers, and front end servers. Each compute node is mainly composed of CPU, a set of memory with 16 gigabytes, and an LSI interconnect between the nodes as shown in Figure 2.4. A CPU is SPARC64 VIIIfx developed by Fujitsu Ltd. and it has 8 cores and a 6 MB L2 cache shared by the cores on an LSI, which operates at a clock frequency of 2 GHz. Its peak performance and performance per electricity unit are 128 GFLOPS and 2.2 GFLOPS/Watt, respectively. Each core has four floating-point multiply-and-add execution units, two of them are operated concurrently by a SIMD instruction, and 256 double precision floating point data registers are suitable for scientific and engineering computations by extending the architecture from the original SPARC architecture [32]. The new designed interconnection network, named Tofu [33], is a six-dimensional mesh/torus network and is used as the data communication network among compute nodes. The LSI for Tofu, named Interconnect Controller (ICC), is directly connected to a single CPU and each ICC has ten routes connecting ten adjacent nodes [34].

One example of CFD execution using supercomputer has been reported by Matsuo *et al.* [22]. A parallel supercomputer called Numerical Simulator III (NS-III) is used, and various CFD applications are performed and evaluated. The system used Fujitsu PRIMEPOWER HPC2500 as main

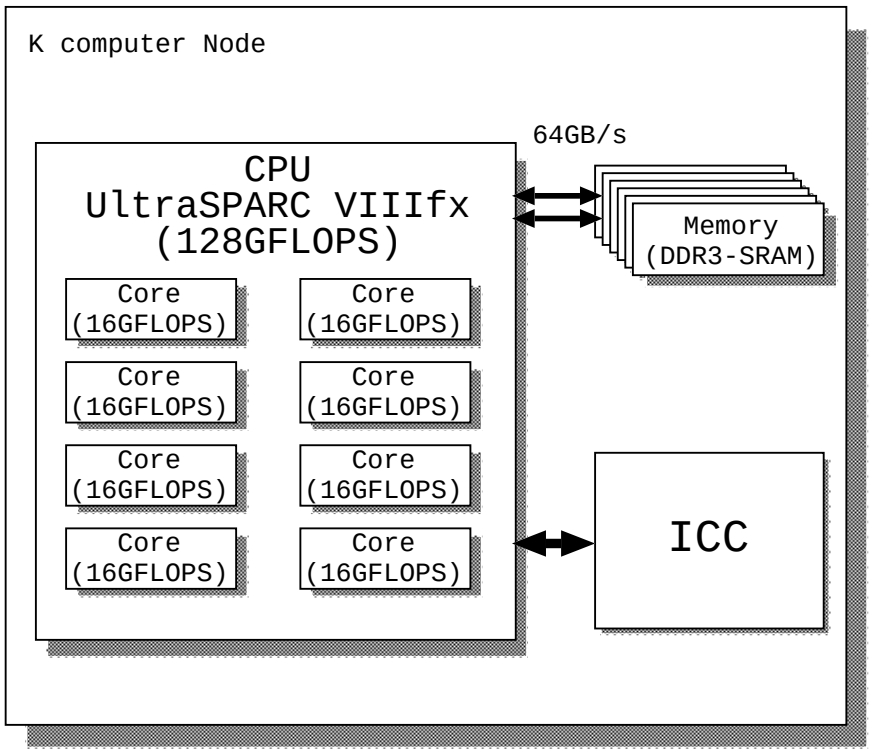

Figure 2.4: Node composition of K computer.

compute engine. Figure 2.5 shows a general view of this system. It has computing capability of 9.3 Tflop/s peak performance and 3.6 TB of user memory, with about 1,800 scalar processors for computation. Another example is performance analysis of CFD application on Nehalem and Westmere based supercomputers [35]. The target is Cart3D CFD application aimed at conceptual and preliminary design of aerospace vehicle with complex geometries. Performance evaluation using two different analysis tools is performed and reported.

However, operational costs and installation area are needed for a supercomputer. The case of K computer previously quoted as an example, for the development, requires 1,120 $\times$ 100 million yen expense and for a maintenance cost, every year is about 80 $\times$ 100 million yen. An installation area about 20,000 m^2 is needed. Moreover, if the application to operate is not suitable for the architecture of the supercomputer, the effective performance may become low.

2.5.2 Cluster

Numerical analysis is also usually executed using cluster computing. A computer cluster consists of loosely connected or tightly connected computers that work together so that in many respects they can be viewed as a single system. The components of a cluster are usually connected to each other through a fast network, with each node running its own instance of an operating system. Computer clusters emerged as a result of convergence of a number of computing trends including the availability of low cost microprocessors, high speed networks, and software for high performance computing.

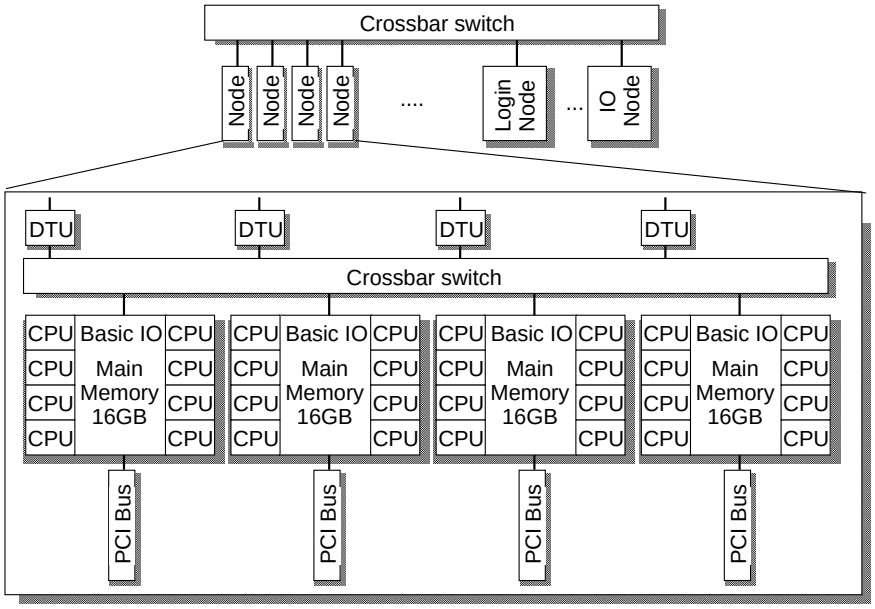

Figure 2.5: A general view of PRIMEPOWER HPC2500.

An example of computer cluster architecture is proposed by Sun Microsystems [36]. The Sun cluster architecture aims to deliver a cost-effective clustering solution that meets all the important user needs. From a physical perspective, a Sun cluster consists of one or more servers that work together as a single entity to cooperatively provide highly available access to applications, system resources, and data of the user community. Each server can itself be a symmetric multiprocessor with multiple CPUs. This architecture enables a potentially large pool of hardware resources to manage as a single system, resulting in significantly lower administration costs, dramatically improved management, and increased flexibility. Global networking of Sun cluster architecture is shown in Figure 2.6.

Xiao *et al.* has proposed an efficient method to parallelize CFD applications on clusters called auto-CFD [37]. Auto-CFD is a pre-compiler which transforms Fortran CFD sequential programs to efficient message passing parallel programs running on clusters. Various types of flow field scale and up to 6 processors are empirically analyzed.

Despite all the advantages of cluster computing, several issues are remaining in concern. Network issues are among others, which is easy for saturation and lossy transmission. In addition, it is hard to distribute workload equally to each node. All these problems might lead to the system which cannot achieve an optimal performance.

2.5.3 ASIC

An application specific integrated circuit (ASIC) is an integrated circuit (IC) customized for a particular use, rather than intended for general-purpose use. Since ASIC is a dedicated communication circuit to specific application, it can realize a high speed, low area, and a low power consumption system.

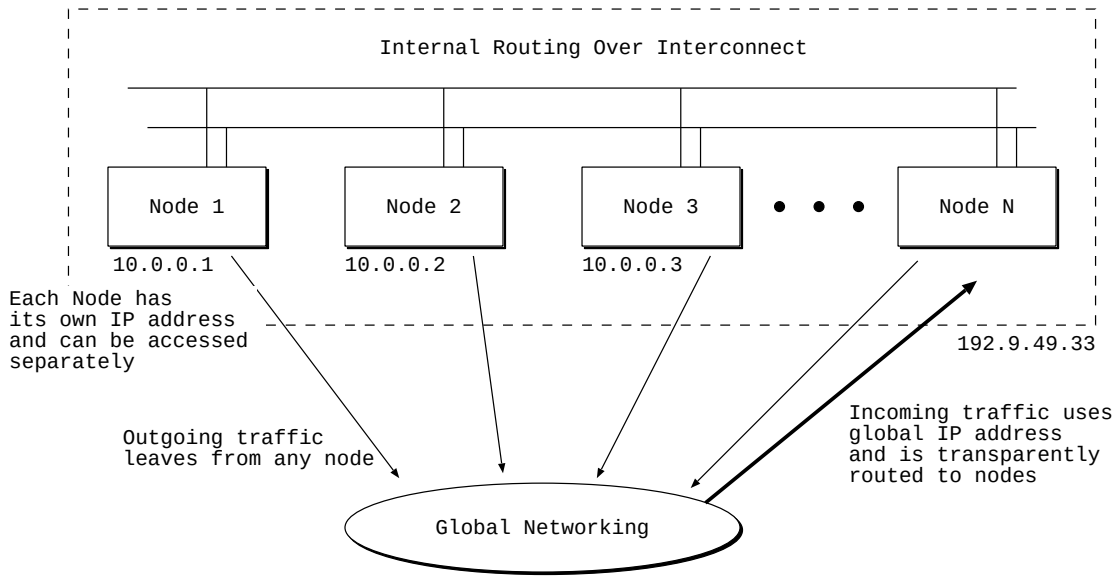

Figure 2.6: Global networking in Sun cluster.

MDGRAPE-3 (Molecular Dynamics Gravity Pipe) is an example of the system using ASIC, which was built in order to calculate the power committed among all the atoms in a molecular dynamics simulation at high speed [38]. The chip currently used by MDGRAPE-3 is an ASIC specialized in those simulations. The composition of MDGRAPE-3 is shown in Fig. 2.7. Fig. 2.7 (1) shows a board of MDGRAPE-3 and (2) a system-wide block diagram.

The MDGRAPE-3 chip functions as the core LSI of the system, and it performs the force calculations. The chip exhibits a peak performance equivalent to 180 GFLOPS at 250 MHz. By employing the broadcast memory architecture, this chip can enclose 20 parallel pipelines, and it performs 720 equivalent floating-point number operations per cycle. The dimensions of the chip are 15.4 mm $\times$ 15.4 mm, and it has been fabricated by Hitachi using 130 nm technologies. It employs a single +1.2 V power supply. The power consumed is less than 0.1 W/GFLOPS, which is considerably lower than that of typical modern general-purpose processors.

In order to communicate with the host computer, an FPGA is installed on the board. It also controls the chip, thermal sensors, and so on. The board is connected to the host by a 10 Gbps serial communication link with a 4-lane 2.5 Gbps I/O through an InfiniBand cable. The host computer has an interface card with an FPGA attached to a PCI-X bus.

However, there are a few problems with ASIC. Once the chip is fabricated, the design cannot be changed anymore. Moreover, the design cost for ASIC is very expensive mainly for the fabrication process. From the CFD application point of view, an ASIC produced is just a special-purpose computer for analysis. Therefore, it has very low chance to enter the market for mass production.

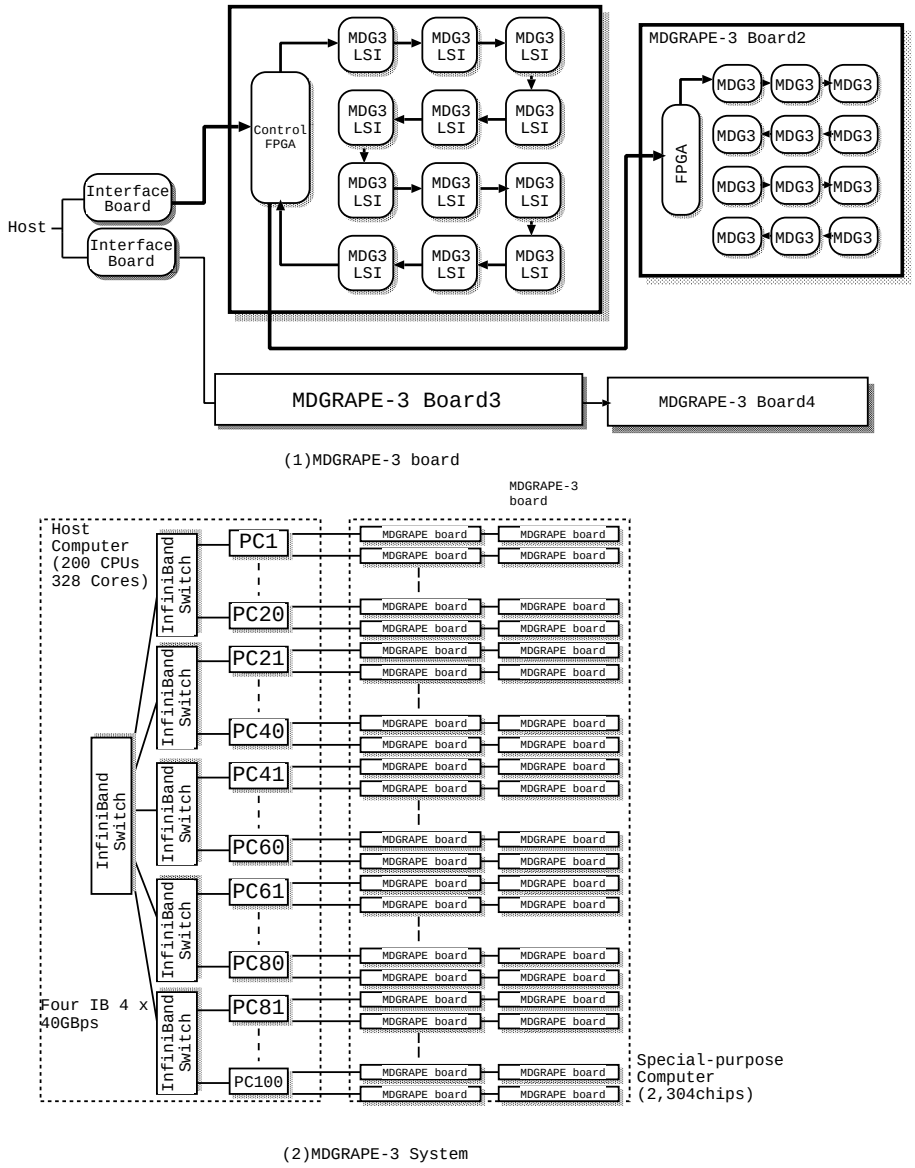

Figure 2.7: Composition of MDGRAPE-3.

2.5.4 GPU

A graphics-processing unit (GPU) is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display. Previously, GPUs are used in embedded systems, mobile phones, personal computers, workstations and game consoles. Lately, GPU has been used widely for many applications in recent years including computational finance and climate weather [39, 40]. Eventually, GPUs contain operation core of hundreds massively parallel processors, and can perform large-scale parallel processing.

Bailey *et al.* has accelerated Lattice Boltzmann fluid flow simulations using GPUs [41]. Lattice Boltzmann methods (LBM) are used for the computational simulation of Newtonian fluid dynamics. They reported improvement upon prior single-precision GPU LBM results by increasing GPU mul-

tiprocessor occupancy, resulting in an increase in maximum performance. This implementation has outperformed single-precision quad-core CPU utilizing OpenMP. Another study has tried to solve parabolic problems using multithread and GPU [42]. They presented an efficient computational scheme for solving parabolic partial differential equations on multithreading and GPU accelerator. Convection-diffusion-reaction (CDR) solver was proposed to quickly solve a big problem on the GPU accelerator. The implementation illustrated 2.5 times faster than those on sequential code with the problem size of 400×400.

Multi-GPU also has become the platform of choice for researchers. Liu *et al.* presented a multi-GPU platform which can solve the incompressible 2D Navier-Stokes equations efficiently [43]. They design a tile structure to distribute the whole computation domain evenly to multi-GPU. Each GPU only solves the equations in a limited area but maintains the whole computation area within its own device memory. In the experiments, single, double and triple GTX 260 graphics cards are installed to construct the platform with CUDA 2.3. With a triple GPU configuration, 270 speed-up is achieved compared to the CPU version without including the memory copy cost. A similar configuration is also about 2.1 faster than a single GPU version.

Furthermore, studies also had been reported on using GPU as a cluster. A study had been reported trying to implement automatic resource scheduling for parallel stencil applications on GPU clusters [44]. This work is to address the problem of time-consuming programming part in stencil application development. They developed an automatic code generation tool to produce a parallel stencil application with latency hiding. The dataflow compiler determines a data decomposition policy for each application, and generates source code that overlaps the stencil computations and communication. They demonstrated two types of overlapping models, a CPU-GPU hybrid execution model and GPU-only model. CFD benchmark computing 19 point 3D stencil is used to evaluate the scheduling performance, which results in 1.45 TFLOPS in single precision on a cluster with 64 Tesla C1060 GPUs.

However, GPU is designed for specific purpose of image processing and graphics. For a reason, it is specializing in stream processing. In stream processing, when there is no data dependence between processing data, high performance can be obtained. In contrast, it is not suitable for high dependable data processing [45].

2.6 Summary

In this chapter, we have discussed about computational fluid dynamics and their applications in high performance scientific computing. CFD advantages compared to analytical and experimental fluid dynamics are discussed. CFD procedure is explained as well as how it is executed. Our intention in this chapter was to emphasize solely on CFD to provide clear explanation on what applications we try to tackle.

Then, we introduced UPACS and FaSTAR CFD software packages. Both packages are developed by JAXA for fluid flow simulations. Both are crucial and important tools in the design of aircraft components such as jet engines and wings. Features of UPACS and FaSTAR are presented and the advantages are explained. Simulation flow and subroutine available in both software packages are also given and discussed. This is mainly to introduce the software packages that we used in this research as well as mapped onto an FPGA.

Finally, we have presented conventional systems on how CFD are executed for the past decade. It includes supercomputer, cluster computer, ASIC as an accelerator and even recently massive parallel processors in GPU. In addition to these entire platforms, example systems as well as their merits and drawbacks are given and discussed. The intention is also to keep aware about the other systems, and provides some insight into the direction it may go in the future.

Chapter 3

FPGA and Partial Reconfiguration

This chapter gives an introduction of FPGA. FPGAs have become one of the most popular implementation media for digital circuits since their invention in 1985. The ability to implement any circuit simply by being appropriately programmed becomes the key point of their popularity. End users can directly configure the final logic structure without the use of an integrated circuit fabrication facility.

3.1 FPGA

FPGAs consist of a large array of Configurable Logic Blocks (CLBs); Digital Signal Processing (DSPs) blocks; Block RAM, and Input/Output Blocks (IOBs). CLBs and DSPs are similar to a processor arithmetic logic unit (ALU) but programmable. It can be programmed to perform arithmetic and logic operations like "add", "multiply", "subtract", and "compare". Unlike processors, which has fixed ALU designed in a general-purpose manner to execute various operations, the CLBs can be programmed with the operations needed by an application. This results in increased computing efficiency.

3.1.1 History

In order to show the history of FPGAs clearly, we need to track back to the evolution of programmable devices. They are general-purpose chips that can be configured for a wide variety of applications. Programmable devices have played a key role in designing digital hardware.

3.1.1.1 PROM

Programmable Read-Only Memory (PROM) is the first type of programmable device used widely. A PROM consists of an array of read-only cells. A logic circuit can be implemented by using the PROM address lines as the circuit inputs, and data lines as outputs.

There are two basic versions of PROMs: mask-programmed and field-programmable. For mask-programmed devices, only the manufacturer can program them while field-programmable devices

can be programmed by the end user. The Erasable Programmable Read-Only Memory (EPROM) and the Electrically Erasable Programmable Read-Only Memory (EEPROM) are two variants of the PROM.

3.1.1.2 PLD

The structure of PROMs is best suited for the implementation of computer memories. Another type of programmable device, Programmable Logic Device (PLD) is designed specifically for implementing logic circuits. The most basic version of a PLD is the Programmable Array Logic (PAL) consisting of a programmable AND-plane followed by a fixed OR-plane. A more flexible version of the PAL is the Programmable Logic Array (PLA) which connects both planes with programmable switches.

3.1.1.3 MPGA

Both types of PLDs described above can only implement small logic circuits due to the simple two-level structure. Their interconnection structure would grow impractically large if the number of product terms was increased.

The solution to this problem is to design programmable devices consisting of an array of uncommitted elements that can be interconnected according to users specification. They are known as Mask-Programmable Gate Arrays (MPGAs). MPGAs provide a general structure that allows the implementation of much larger circuits.

Since MPGAs are mask-programmable, they require significant manufacturing time and incur high initial costs. In 1985, Xilinx introduced Field-Programmable Gate Arrays (FPGAs). Like an MPGA, an FPGA consists of an array of uncommitted elements that can be interconnected in a general way. The interconnections between the elements are also user-programmable. Low cost hardware prototypes are built using such FPGAs. Since the early 1990s, many different FPGAs have been developed by a number of companies.

The FPGA is generally distinguished by two features. First, instead implementing gates physically, the logic is implemented with look up tables. Second, most FPGAs use static RAM cells to hold the configuration information as opposed to permanent ways of earlier devices. This allows the FPGA to be configured and reconfigured after the device has been installed in a product.

3.2 Architecture of an FPGA

All FPGAs consist of three fundamental components: Processing Units; Input/Output; and Interconnect. Figure 3.1 shows the basic model of an FPGA. From the high-level view, the FPGA looks like a network of processors.

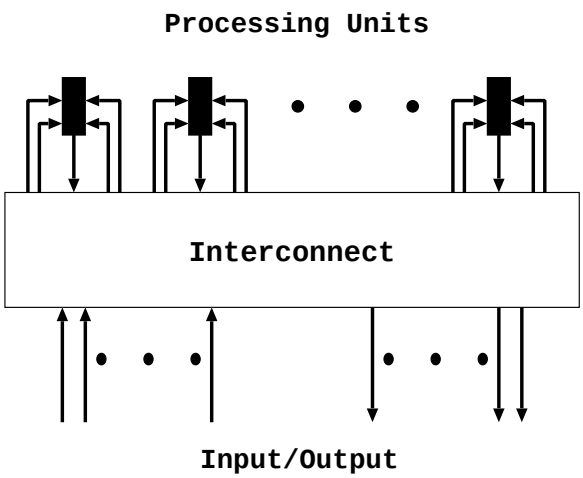

Figure 3.1: Basic model of FPGAs.

However, an FPGA differs from a conventional multiprocessor in several ways:

- Granularity
  Conventional FPGAs have single bit processing elements, each of which is controlled independently.
- Instruction Control
  Conventional FPGAs are configurable with a single instruction resident per processing element. Changing instructions is slow compared to the instruction processing rate in a general purpose processor.
- Static Interconnect
  In conventional FPGAs, an interconnect is purely static. It connects sources and sinks by locking down a path through the switching network.

Figure 3.2 shows a generic FPGA architecture. A circuit is implemented in an FPGA by programming each logic block so as to take a role of a small logic portion. Each I/O block is programmed for either an input pad or an output pad. The programmable routing is configurable to make all the necessary connections between logic blocks and from logic blocks to I/O block.

3.2.1 Commercially Available FPGAs

Several companies: Xilinx; Altera; and Actel have introduced a number of different types of FPGAs. In this subsection, we will focus on Xilinx FPGAs, not only because it is the first company that introduced FPGA but also because it is the biggest vendor of FPGA nowadays, and its FPGAs are most widely used.

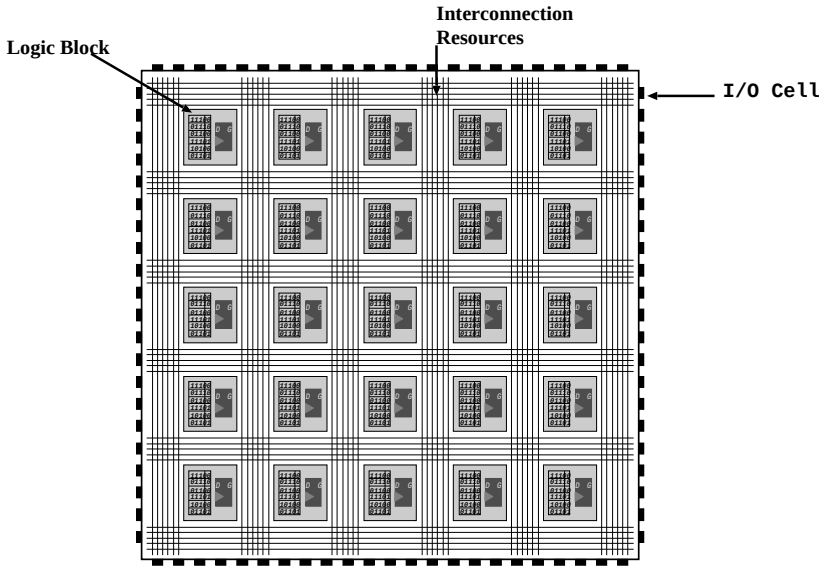

Figure 3.2: Schematic of an FPGA.

Commercial FPGAs can be classified into three groups based on their routing architecture: an island-style; row-based; and hierarchical. Xilinx FPGAs adopt an island-style. Figure 3.3 shows the island-style layout, where blocks are arranged in an array with vertical and horizontal routing.

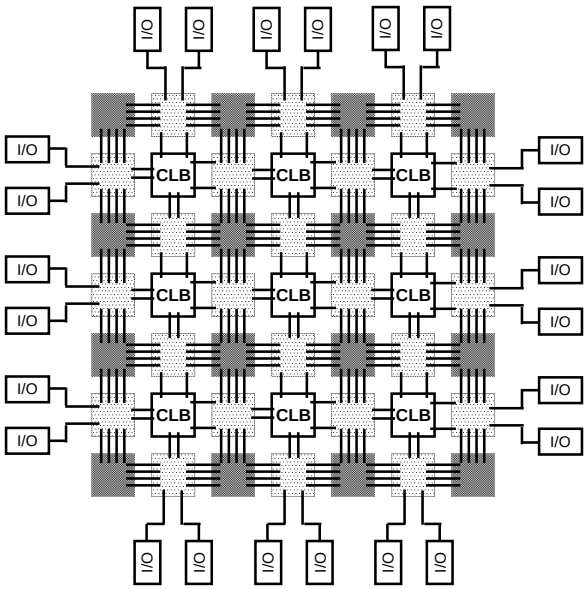

Figure 3.3: Island-style interconnects in commercial FPGA.

Early FPGAs consist of an array of similar programmable logic elements interfaced to interconnection elements. Today, modern FPGAs consist of a heterogeneous fabric of programmable elements connected by a switch-based network. As mentioned above, in many cases, an FPGA was used for prototyping an ASIC. It was rarely used for high performance applications. However, recent technology has allowed FPGAs to evolve remarkably by increasing the amount of resources, such as

slices; DSPs; and Block RAMs. It also improved to work with high clock frequency.

Table 3.1 shows Virtex series FPGAs released from 2002 to 2011. Figure 3.4 shows the amount of hardware resources of slice; DSP; and Block RAM in each generation of Virtex FPGAs.

Table 3.1: List of Virtex series FPGAs.

Series	Process Size	Year Release
Virtex-II Pro	130 nm	2002
Virtex-4	90 nm	2004
Virtex-5	65 nm	2006
Virtex-6	40 nm	2009
Virtex-7	28 nm	2011

3.2.2 Virtex 6 FPGA

The Xilinx Virtex 6 FPGA provides advanced features in the FPGA market. Virtex 6 FPGAs are the programmable silicon foundation that delivers integrated software and hardware components to enable designers to focus on innovation. The Virtex 6 family contains three distinct sub-families:

- LXT: High performance logic with advanced serial connectivity
- SXT: Highest signal processing capability with advanced serial connectivity
- HXT: Highest bandwidth serial connectivity

Each sub-family contains a different ratio of features to most efficiently address the needs of wide variety of advanced logic designs. In addition to the high performance logic fabric, Virtex 6 FPGAs contain many built-in system level blocks. Built on a 40 nm state-of-the-art copper process technology, Virtex 6 FPGAs are a programmable alternative to the custom ASIC technology.

3.2.2.1 Look-Up Table

Function generators in Virtex 6 FPGAs are implemented as 6-input Look-Up Tables (LUTs). There are six independent inputs and two independent outputs for four function generators in a slice. The LUTs can implement an arbitrarily defined 6-input Boolean function. Each LUT can also implement two arbitrarily defined 5-input Boolean functions, as long as these two functions share common inputs.

In practice, a Hardware Description Language (HDL) is used to describe the digital circuit, and then synthesis tools are used to map the textual description into LUTs. The designer never defines logics in LUTs directly. The important consideration for a designer is representing a circuit efficiently to utilize available resources.

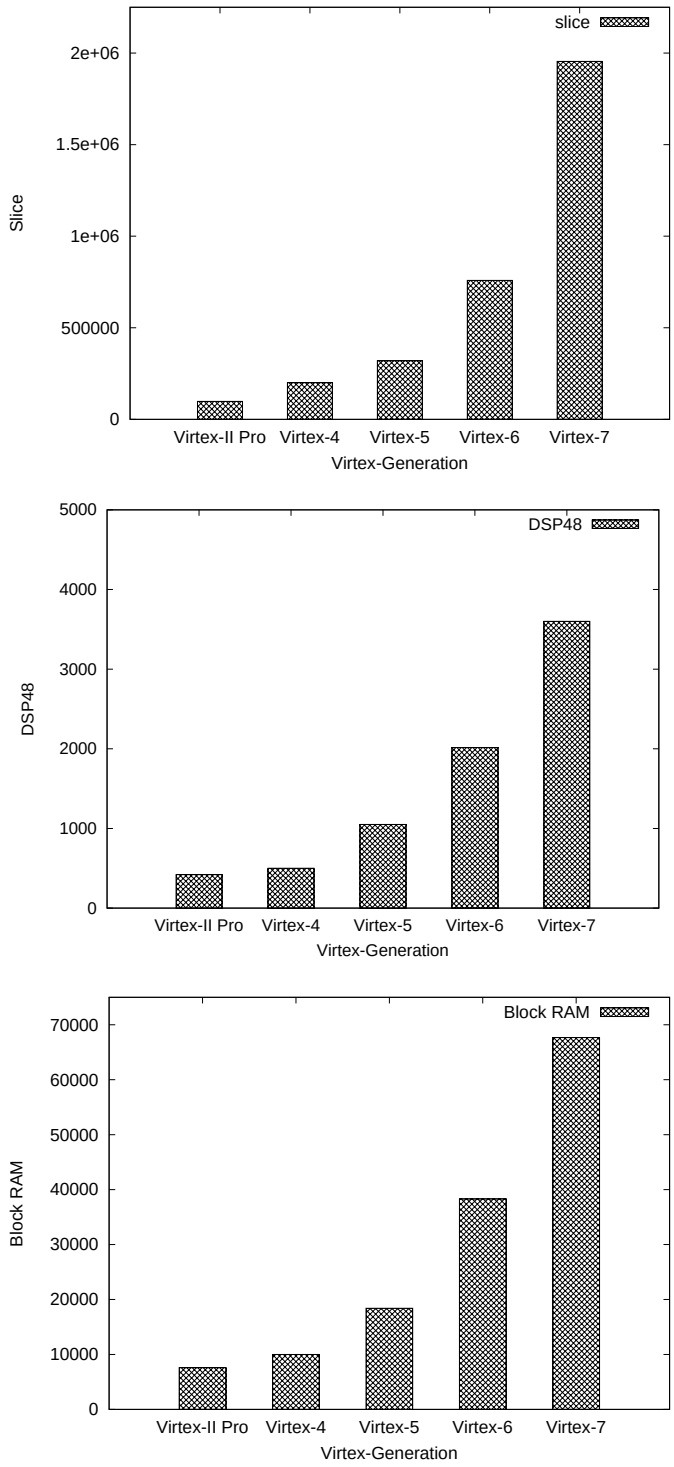

Figure 3.4: The amount of resources in Virtex FPGAs.

Table 3.2: Logic resources of one CLB in Virtex 6 FPGA.

Slices	LUTs	Flip-Flops	Arithmetic and Carry Chains	Distributed RAM	Shift Registers
2	8	16	2	256 bits	128 bits

3.2.2.2 Slice

Every slice contains four logic-function generators (or look-up tables); eight storage elements (D flip-flop); wide-function multiplexers; and carry logic. In addition to Boolean logic, a slice can be used for arithmetic and storing data. Some slices are connected in such a way that they can be used for data storage as distributed RAMs. This is accomplished by combining multiple LUTs in the slice.

In addition to logic and memory, slices can be used as shift registers. A shift register is capable of delaying an input with x clock cycles. Using a single LUT, data can be delayed up to 32 clock cycles. Cascading all four LUTs in one slice, the delay can increase to 128 clock cycles. This is useful for small buffers instead of using a large block RAMs.

3.2.2.3 CLB

The Configurable Logic Blocks (CLBs) are the main logic resources for implementing both sequential and combinatorial circuits. Each CLB element is connected to a switch matrix for accessing the general routing matrix as shown in Figure 3.5. A CLB element consists of a pair of slices. These two slices do not have direct connections to each other, and each slice is organized as a column. Each slice in a column has an independent carry chain. For each CLB, slices in the bottom of the CLB are labeled as SLICE(0), and slices in the top of the CLB are labeled as SLICE(1). Figure 3.6 shows a row and column relationship between CLBs and slices. Table 3.2 summarizes the logic resources in one CLB.

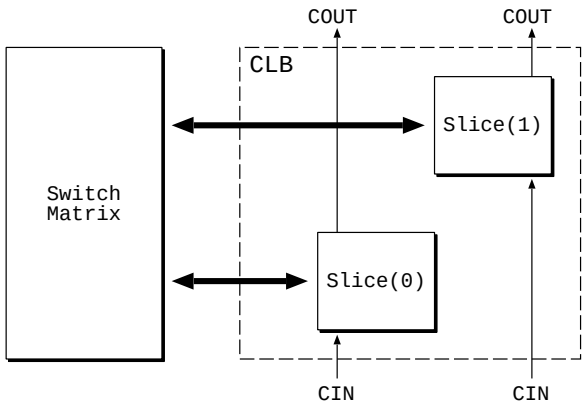

Figure 3.5: Arrangement of slices within the CLB.

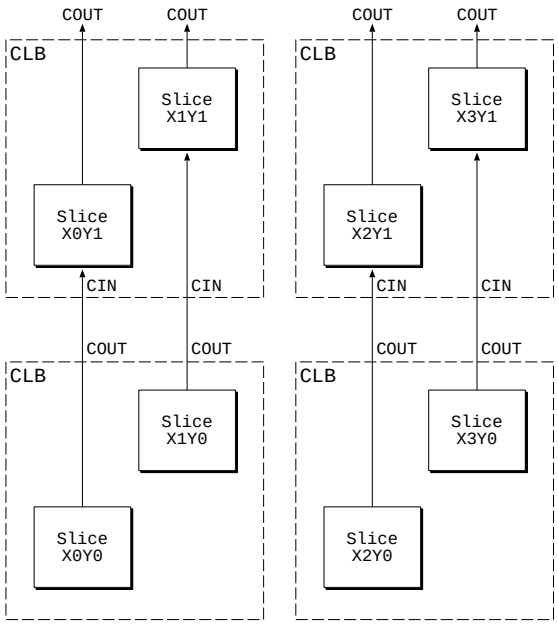

Figure 3.6: Row and column relationship between CLBs and slices.

3.2.2.4 Block RAM

Block RAM is a dedicated random access memory grouped together in 36 Kbits blocks in Virtex 6 FPGAs. Every Virtex 6 FPGA has 156 to 1064 dual-port block RAMs. Each block RAM has two completely independent ports that share nothing but the stored data. The clock controls each memory access, read and write. All inputs; data; address; clock enables; and write enables are registered. Nothing happens without a clock. The input address is always clocked, retaining data until the next operation. An optional output data pipeline register allows higher clock rates at the cost of an extra cycle of latency. During a write operation, the data output can reflect either the previously stored data, the newly written data, or remain unchanged. One common use of block RAMs in FPGA design is for FIFOs or data queues.

3.2.2.5 DSP Slices

DSP applications use many binary multipliers and accumulators implemented in dedicated DSP slices. All Virtex 6 FPGAs have many dedicated, full custom, and low power DSP slices. In Virtex 6, the DSP slices are known as DSP48E1 (48 bit DSP element) slices. Figure 3.7 shows DSP48E1 diagram available in Virtex 6 FPGA.

Each DSP48E1 slice consists largely of dedicated 25×18 bit two's complement multiplier and a 48 bit accumulator. They can operate at 600 MHz in maximum. The multiplier can be dynamically bypassed, and two 48 bit inputs can be connected into a single-instructions-multiple-data (SIMD) arithmetic unit, or a logic unit that can generate any one of 10 different logic functions of the two operands.

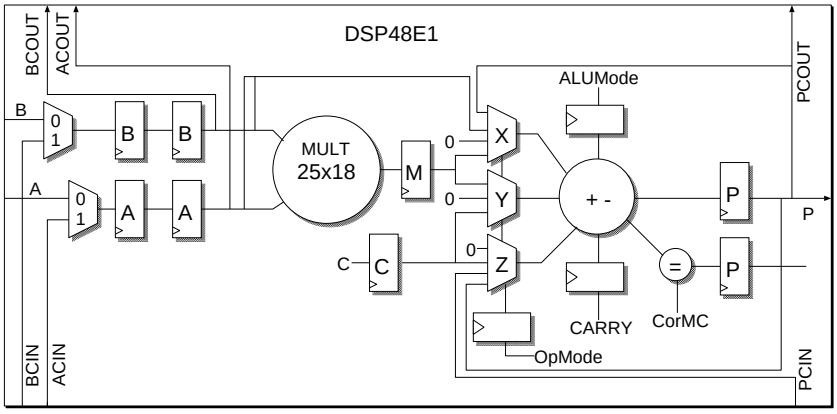

Figure 3.7: DSP slice in Virtex 6 FPGA.

The DSP48E1 slices provide extensive pipelining and extension capabilities that enhance speed and efficiency of many applications, even beyond digital signal processing, such as wide dynamic bus shifters, memory address generators, wide bus multiplexers, and memory-mapped I/O register files. The accumulator can also be used as a synchronous up/down counter.

3.2.2.6 Clocks

In FPGA designs, it is common to operate different cores with different frequency clocks. In traditional design, any clocks needed would have to be generated off-chip and connected as input to the system. However, in recent FPGA designs, it is possible to generate a wide range of clocks from a single to a few clock sources. The FPGA is bisected, and on each half, a clock region spans 40 CLBs. Figure 3.8 is a simple example of the clock regions on the FPGA.

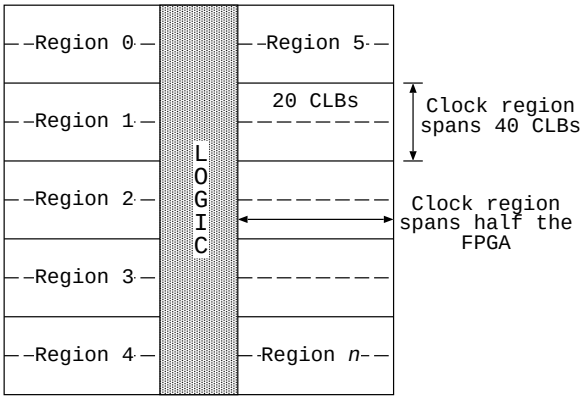

Figure 3.8: Clock regions on an FPGA span 40 CLBs vertically and half of the FPGA horizontally.

Each Virtex 6 FPGA provides five different types of clock lines, global clock buffer, BUFG, regional clock buffer, BUFR, I/O clock buffer, BUFIO, horizontal clock buffer, BUFH, and the high performance clock to address the different clocking requirements.

In each Virtex 6 FPGA, 32 global clock lines have the highest fanout and can reach the clock signal of every flip-flop. There are 12 global clock lines in any regions. Global clock lines can be driven by global clock buffers, which can also perform glitchless clock multiplexing. Global clocks are often driven from the Clock Management Tiles (CMTs), which can completely eliminate the basic clock distribution delay.

Regional clocks can drive all clock destinations in their region as well as the region above and below. A region is defined as any area corresponding to 40 I/Os and 40 CLBs. Virtex 6 FPGAs have 6 to 18 regions. There are 6 regional clock tracks in every region. Each regional clock buffer can be driven from either four clocks capable input pins, and its frequency can be optionally divided by any integer from 1 to 8.

In addition, I/O clocks are provided especially for the fast I/O logic and serializer/deserializer circuits. Virtex 6 devices have a high performance direct connection from the Mixed Mode Clock Managers (MMCMs) to the I/O directly for low-jitter and high performance interfaces.

To help design, Xilinx uses Digital Clock Managers (DCMs) for clock management. Generally speaking, a DCM takes an input clock and can generate a customizable output clock. By specifying the multiply and divider values M and D, the output clock $clkfx$ can be generated as a custom clock. Given an input clock $clkin$, the following equation is used to generate the output clock:

$$clkfx = M/D \times (clkin) \tag{3.1}$$

However, the DCM provides more than just generating different clock rates. It also can shift the phase of the input clock by 90, 128, and 270 degrees. The DCM provides a 2× input clock rate, and can shift the phase of the input clock by 180 degrees.

3.3 Computing Using FPGAs

FPGAs have historically not been frequently used for HPC applications since they are relatively small and slow. Over time, however, advancements in process technology have enabled vendors to manufacture chips containing multi-millions of transistors [46]. The architectural enhancements increased both logic cell count and speed. For instance, with an average 25% improvement in typical clock frequency for each FPGA generation, the logic computing performance has improved approximately by 92 times over the past decade while the cost of FPGAs has decreased by 90% in the same time period. These developments have made it feasible to perform massive computations on a single chip. Figure 3.9 shows the advancement of density, speed, and reduction in price over time.

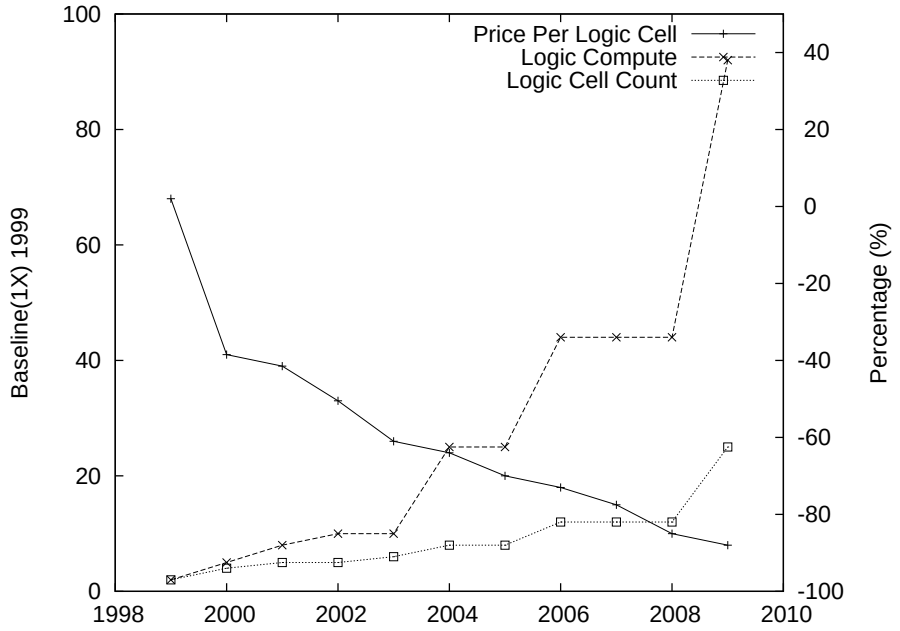

Figure 3.9: Historical advancement of FPGA technology.

3.3.1 Parallelism Offered by FPGA

FPGAs are composed of a large array of logic blocks. Depending on the type of used operators, CLBs and DSPs can perform bitwise, integer, and floating point operations. The results of the operations are stored in the registers present in CLBs, DSPs, and Block RAM. These blocks within an FPGA can be connected via flexible configurable interconnects. The output of an operator can directly forwarded to the input of the next operator. That is, the FPGA architecture can be used for the design of data flow engines.

The FPGA architecture provides the flexibility to create a massive array of application specific ALUs that enable both instruction and data level parallelism. Because data flows between operators implemented with CLBs and DSPs, there are no inefficiencies like processor cache misses. These operators can be configured so as to be connected with point-to-point dedicated interconnects, thereby a pipeline can be formed between multiple operators.

3.4 Applications in Fluid Dynamics

Performance acceleration using FPGAs has been done for various applications. This subsection introduces some examples of HPC systems using FPGAs for fluid dynamics applications. Early work on methodology for CFD acceleration through reconfigurable hardware is reported by Andres *et al.* [47]. It discussed about early development of FPGA-based hardware modules for acceleration of most time consuming algorithms in aeronautics analysis. Results comparing CPU-based and FPGA-based solutions are presented, and it shows that speedups around two orders of magnitude can be

expected from the FPGA-based implementation. Another work has proposed a heterogeneous architecture for evaluating real-time one-dimensional CFD on FPGAs [48]. The paper presents a novel framework to evaluate 1D CFD models in real time on an FPGA. The results demonstrate the resource savings and scalability of the framework. The feasibility of the approach is confirmed. Acceleration of fluid dynamics using multi-FPGA platforms was also proposed in [49]. The paper presents an implementation of the fluid registration algorithm on a multi-FPGA platform called Convey HC-1. The implementation is achieved using a high-level synthesis (HLS) tool, with additional source code level optimization approach.

In addition, a few notable examples of deploying FPGA for CFD applications are discussed as follows.

3.4.1 FLOPS-2D

FLOPS-2D is a scalable multi-FPGA system developed by JAXA [23], as shown in Figure 3.10. Each FLOPS board provides a Xilinx Virtex 4 XC4VLX100 FPGA, four high-speed serial links interfaced by PCM's PM8358 and two SO-DIMM slots each for 2GB SDRAMs. By connecting boards with serial links, FLOPS-2D can increase its size or change the interconnection topologies freely. Although the interface chip PM8358 enables to use various type of serial interconnection, 10 Gigabit standard interface is used in the implementation.

FLOPS-2D is connected with a host PC through a general purpose FPGA board. The data is continuously transferred form the host PC through the interface board, and computed with pipelines implemented on multiple FPGAs. If required, the data can be stored in the SDRAM in each board. This structure allows high degree of modularity and flexibility, although the parallel/serial data transform is needed to exchange data between FPGAs.

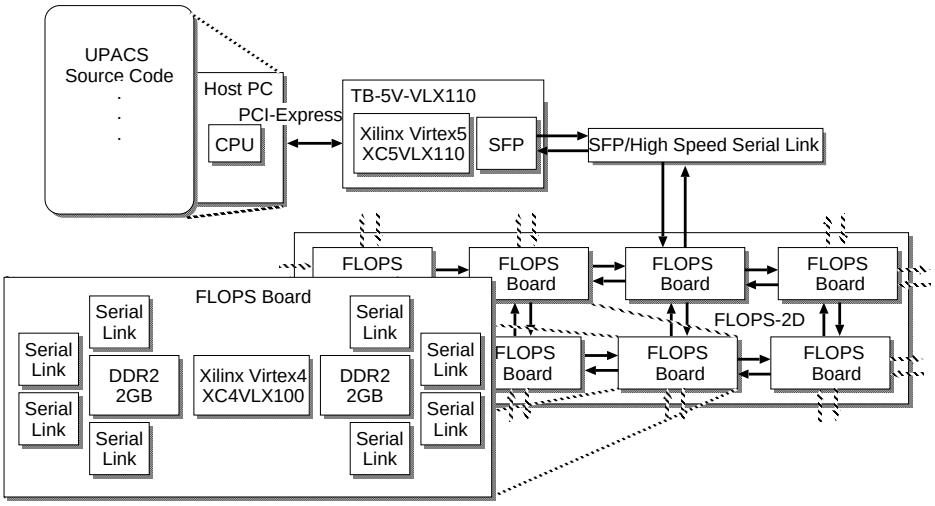

Figure 3.10: The whole FLOPS-2D composition.

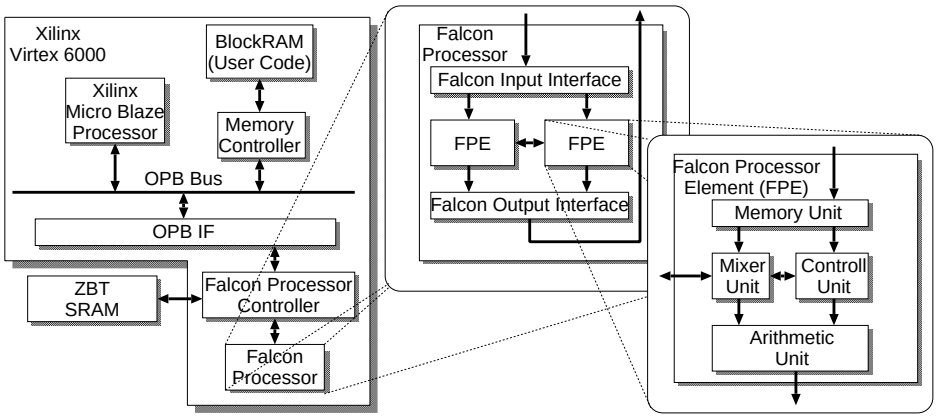

Figure 3.11: The outline of Falcon architecture.

3.4.2 Falcon

Kocsardi *et al.* used soft core processors provided by FPGA cooperated with external high-speed RAM [50]. Figure 3.11 shows the Falcon architecture where part of the CFD processing is mounted. It provides a floating point processing unit. Compared to the Intel Core 2 Duo processor operating at 2 GHz, improvement in the speed of about 21 times is realized.

3.4.3 Systolic Architecture

Another work is an FPGA-based flow solver based on the systolic architecture [51,52]. It shows that the fractional-step method with central difference schemes can be executed with a systolic algorithm, and therefore the systolic architecture is suitable for a dedicated processor.

2D systolic array of cells, each of which has a micro-programmable data-path containing a MAC (Multiplication and Accumulation) unit and a local memory to store necessary data for CFD, is designed as shown in Figure 3.12. With ALTERA Stratix II FPGA, 96(=12×8) cells are implemented running at 60 MHz. Total peak performance is 11.5 GFlops, while 7.14 and 6.41 times faster computations are achieved compared to Pentium 4 processor at 3.2 GHz and Itanium 2 at 1.4 GHz, respectively.

3.4.4 Alpha-Data

Nagy *et al.*'s approach is implementing an unstructured mesh geometry on an Alpha-Data reconfigurable development system equipped with a Xilinx Virtex 6 FPGA with 2 Gbyte on-board DRAM [53]. The DRAM has four 32 bit wide banks running on 800 MHz. Thus, it provides 12.8 Gbyte/s peak theoretical bandwidth. The cell-centered state and constant values are stored and loaded from an off-chip memory, since the number of cells is far exceeding the available on-chip memory as shown in Figure 3.13. Overlapping of input data sets is adopted to reduce the number of memory accesses and to save memory bandwidth.

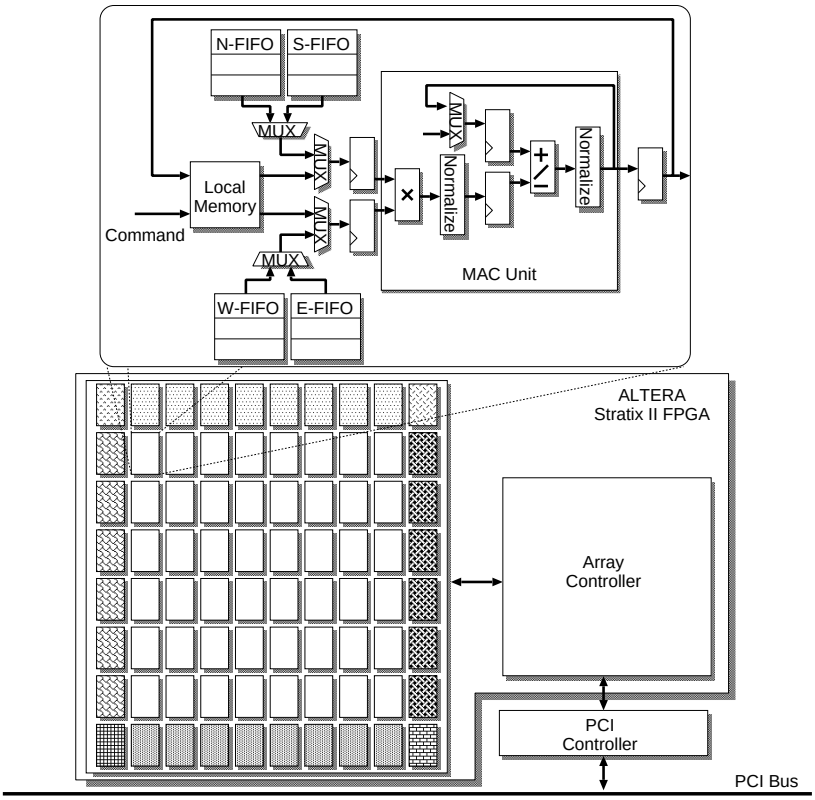

Figure 3.12: The outline of systolic architecture.

Performance of this architecture is determined using the result of the post place and route static timing analysis. It achieves 325 MHz operating frequency for the double precision computing. 23.08 GFLOPs performance is reported on a Virtex 6 FPGA. Three arithmetic units can be implemented and connected in a pipeline manner, and it achieves 69.22 GFLOPs cumulative computing performance.

3.4.5 XtremeData

Another work has addressed the problem of accelerating CFD applications by studying the hardware implementation of a cell-vertex finite volume algorithm to solve Euler equations, using the XtremeData in-socket FPGA accelerator [54]. Figure 3.14 shows the hardware process distribution in XtremeData FPGA.

The target hardware module is the XtremeData in-socket accelerator, which contains three Altera Stratix III FPGAs. The module plugs directly into the FSB (Front Side Bus) socket of any dual processor Xeon system. In the three FPGAs within the XtremeData in-socket accelerator, one serves as a bridge to the FSB, and the others are available to implement the user logic. These two application FPGAs are connected through two 64-bit dedicated buses. In addition, the XtremeData module includes two QDRII+ SRAM banks, one for each user FPGA. Each of these banks has 8 MB - 2 MB $\times$ 32 bits with two independent read/write ports whose bandwidth is 2.8 GB/s for each.

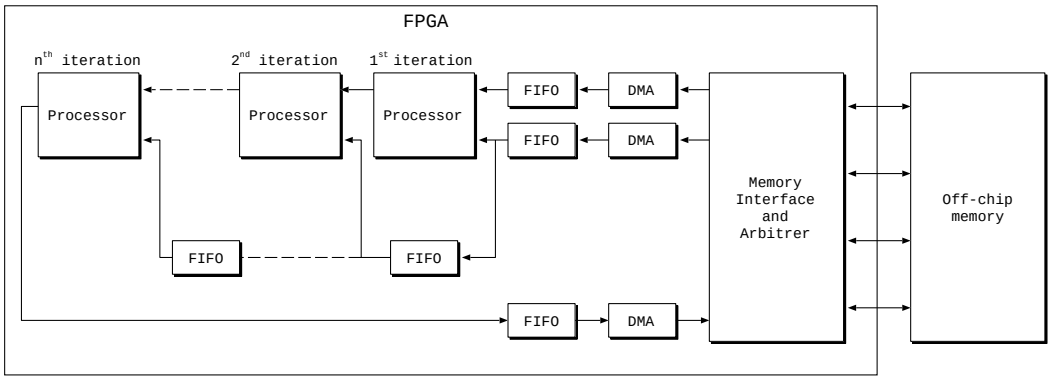

Figure 3.13: Block diagram of the FPGA in Alpha-Data architecture.

For taking advantage of high-level language synthesis tools, an Impulse C tool is used with optimized low level components. The hardware accelerated implementation achieved speedups up to 13.25 times in performance.

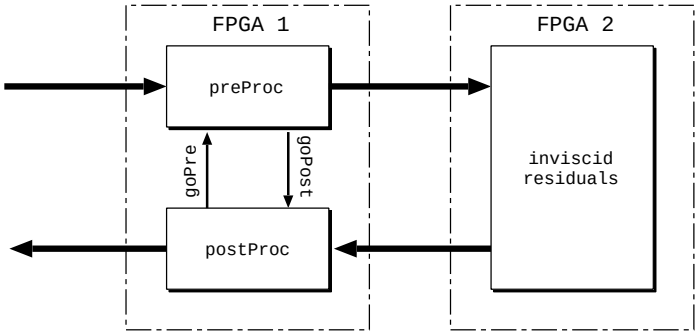

Figure 3.14: Hardware process distribution in XtremeData.

3.5 Partial Reconfiguration

One of the features of the Xilinx Virtex architecture is the ability to reconfigure a portion of the FPGA while the remainder of the design is still operational. The technique called partial reconfiguration is useful for applications that require the loading of different designs into the same area of the device or the flexibility to change portions of a design without reconfiguring the entire device. Using smaller devices improves system cost and lowers power consumption.

As systems become more complex and designers are asked to do more with less, FPGA adaptability has become a critical asset. While FPGAs have always provided the flexibility to do on-site device reprogramming, today's constraint demands even more efficient design strategies. Partial reconfiguration extends the inherent flexibility of the FPGA by allowing specific regions of the FPGA to be reprogrammed with new functionality while applications continue to run in the remainder of the device. Partial reconfiguration addresses three fundamental needs by enabling the user to:

- reduce cost and/or board space,
- change a design in the field, and
- reduce power consumption.

3.5.1 Reduce Cost

Two most prevalent user problems addressed by partial reconfiguration are:

- fitting more logic into an existing device, and
- fitting a design into a smaller, less expensive device.

Historically, designers have spent days on trying new implementation, reworking code, and re-engineering solutions to squeeze their design into the smallest possible FPGA. Partial reconfiguration enables these designers to reduce the size of their designs by dynamically time-multiplexing portions of the available hardware resources. The ability to load functions as-needed basis also reduces the amount of idle logic.

An example of this strategy is the use of partial reconfiguration within a Software Defined Radio (SDR) system [55], where the user uploads a new waveform on demand to establish communication with a new channel. Any number of waveforms can be supported by a single hardware platform, requiring only unique partial bitstreams to be available for these waveforms. Established links to other channels are not disrupted by the update to another channel due to the on the fly characteristics of partial reconfiguration.

3.5.2 Increased System Flexibility

In the past, changing a design in the field required new placement and routing of the design and the delivery of a full configuration file. The designer also had to shut the system down while making the change. In contrast, when using partial reconfiguration, the designer needs only to place and route the modified function in the context with the already verified remainder of the design, and then delivers this new partial image to a system in the field.

Moreover, the designer can dynamically insert new functions, while the system is up and running. It improves system available time. Thus, mutually exclusive functions can be plugged into the same space without having to redesign the system or move to a bigger device.

Another example of the benefits of partial reconfiguration is its use within Optical Transport Network (OTN) solutions [56]. Like with SDR, different protocols can be supported to create a more efficient hardware system. Only the protocol for a particular channel needed at any point in time is loaded in the FPGA. A deployed system can handle traffic of many different types using minimal resources. The system also can be updated with the latest protocols without requiring a full redesign.

3.5.3 Reduce Power Consumption

Power consumption has become a primary concern for today's designers. Like size and cost, it is a metric with strict limits in most systems. However, as FPGA designs grow in size and complexity, they consume more power. While synthesis and implementation tools coupled with appropriate design techniques can help to reduce power consumption, partial reconfiguration implementations can further reduce static and dynamic power [57, 58].

One way to reduce static power is to simply use a smaller device. With partial reconfiguration, designers can essentially use time slice in the FPGA and run parts of their design independently. The design then requires a much smaller device, since not every part of the design is needed 100% of the time.

Partial reconfiguration also has the potential to reduce operating power as well as static power. For example, many designs must be able to run at a very high speed. However, only small percentage of the time that maximum performance might be needed. In order to save power, designers can use partial reconfiguration to swap out a high performance design with a low power version of the same design. The designer can then switch back to the high performance design when the system requires it.

3.5.4 Additional Advantages

The ability to time multiplex hardware dynamically on a single FPGA offers a number of additional advantages.

- provides real-time flexibility in the choice of algorithms or protocols available to an application at any given moment,
- enables the use of new techniques in design security,
- improves FPGA fault tolerance, and
- reduces bitstream storage requirements.

Partial reconfiguration is a powerful solution that can dramatically extend the capabilities of FPGAs. In addition to the potential for reducing size, weight, power, and cost, partial reconfiguration enables new types of FPGA designs that provide efficiencies unattainable with the conventional design techniques.

3.6 Design Flow for Partial Reconfiguration

Implementing a partially reconfigurable FPGA design is similar to implementation of multiple non-partial reconfiguration design that shares common logic. Partitions are used to ensure that the common logic between the multiple designs is identical. Figure 3.15 illustrates this concept.

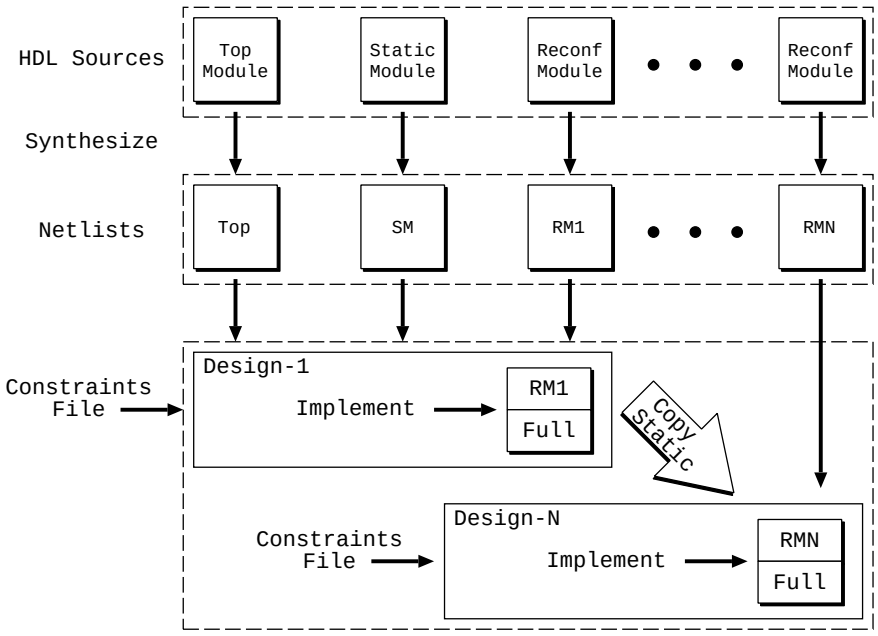

Figure 3.15: Overview of the partial reconfiguration design flow.

The top box in the first row represents HDL sources for top, static and reconfigurable modules. At this stage, the RTL simulation is also performed. Then, all modules are synthesized to produce netlist for each module. Before the design, constraints files are inserted for the target design. The appropriate netlists are implemented in each design to generate the full and partial bit files for that configuration. The static logic from the first implementation is shared among all subsequent design implementations.

At the same time, the design specification must be analyzed thoroughly, and limitations associated with partial reconfigurable designs are considered. Key considerations in partial reconfiguration design are listed as follows:

1. *Design structure*

   An appropriate design hierarchy must be provided to resolve complexities and difficulties during implementation. A clear design instance hierarchy simplifies physical constraints. Grouping logic that is packed together in the same hierarchical level is necessary.

2. *Elements inside reconfigurable module*

   Not all resources on FPGA are permitted to be targets of reconfiguration. Although most modules such as slice, block RAM, and IOB can be reconfigured, global logic and clocking resources must be placed in the static region. This is because it should remain operational during reconfiguration.

3. *Packing logic*

   Any logic packed together must be placed in the same group, whether it is static or reconfig-

urable. For example, I/O registers must remain with the I/O port. Hierarchical boundaries must be chosen appropriately, since the insertion of proxy logic may result in suboptimal results or impossible routing.

4. *Clocking*
   Virtex-6 is the first architecture which has multiple columns of Region Clock Buffer (BUFR) within the same clock region. Partial reconfiguration requires that all BUFRs used within one clock region must be contained in the same partition.

5. *Decoupling functionality*
   Since the reconfigurable logic is modified while the FPGA device is operating, the static logic connected to outputs of reconfigurable module must ignore data during the partial reconfiguration. The reconfigurable modules will not output valid data until partial reconfiguration is completed and the reconfigured logic is reset.

6. *Partition boundaries*
   Partial reconfiguration is done frame by frame. When partial bit files are generated, they are built with a discrete number of configuration frames. Partition boundaries do not have to align to reconfigurable frame boundaries, but the most efficient place and route results are achieved when this is done.

7. *Proxy logic*
   Partition pins are defined as the interface between static and reconfigurable logic. No special logic or tags are required to accommodate this definition. In most cases, an LUT is inserted at this interface point to represent this node.

8. *Black boxes*
   The partial reconfiguration design also allows black boxes to be implemented as reconfigurable modules. This is an effective way to reduce the size of full configuration bit file, and therefore it reduces the initial configuration time.

9. *Constraints*
   In order to adequately constrain the entire design, constraints are given for both the static and reconfigurable portions of the design. Top User Constraints File (UCF) is shared by all the static and reconfigurable modules.

10. *Implementation strategies*
    There are trade-offs associated with optimizing partial reconfiguration design. Reconfigurable partition becomes a barrier for optimization, and reconfigurable frames require specific layout constraints. When configurations for this design is built, the first configuration chosen for implementation is the most challenging one. It is also important that the selected physical region has required amount of resources such as DSP48Es.

3.7 Summary

FPGAs continue to be used in a myriad of programmable systems. Technology and feature advancements make FPGAs ideal for use in HPC applications. Due to massive parallelism offered by FPGA architectures, many HPC applications can be accelerated in performance when compared to stand alone CPUs.

On the other hand, enormous performance gains are due to architectural enhancements and increasing of chip density. It is directly correlating to significant applications speedups. Together with low operating power consumption, extremely high performance to power ratios can be realized on FPGA-based HPC systems. In addition to their longstanding reputation as the platform of choice for designing programmable systems, FPGAs are rapidly becoming a valuable and lasting solution to meet the challenging processing and interface demands of HPC applications.

In the applications of fluid dynamics, FPGAs have attracted attention from many researchers, and several implementations for CFD applications were proposed. Moreover, recent partial reconfiguration technology has provided extra benefits in deploying FPGAs for various applications. In addition to the potential for reducing size, power and cost, partial reconfiguration increased system flexibility, and enables new types of FPGA designs that provide efficiencies. We discussed the design flow of partial reconfiguration and the design consideration to achieve required performance.

Chapter 4

UPACS Code Implementation

This chapter presents target subroutine implementation in UPACS exploration. Two types of reconfiguration — static and dynamic, are implemented, followed by performance evaluation for both designs.

4.1 UPACS

UPACS is a CFD package to simulate compressible flow using multi-block grids. It provides researchers an easy way to run large-scale simulations. It has been developed as a common aerospace CFD software equipping with flexibility, scalability and portability since year 2000 [28]. The application is written in Fortran 90 and it supports the Message Passing Interface (MPI) interface. UPACS supports Euler, Navier-Stokes and Reynolds Averaged Navier-Stokes equations as governing equations. By choosing solvers, users can execute simulations on their parallel systems without any code tuning. Users can also select desired solutions and determine the number of processes by setting parameters. In order to run a simulation, users just prepare a parameter file and grid data files.

4.1.1 Profiling

As the early stage of this study, we profiled execution time of UPACS on SPARC64V processors at 1.3 GHz with Solaris8 operating system. To this end, we used a 40^3 grid dataset. Figure 4.1 shows the simulation flow and profiling result of UPACS and its correspondent percentage of execution time. The core routine occupies about 70% of the total execution time. It also shows that the execution time of UPACS is mainly consumed in the main loop distributed into several procedures. Note that, the percentage of the execution time is only shown from "boundary condition, pressure" to "calculate residual", and thus the sum does not reach 100%. Amdahl's law indicates that subroutine with large computation time should be selected.

On the other hand, subroutines with complicated data dependency are hard to implement. Considering these factors, two subroutines are possible for selected: MFGS (Matrix Free Gauss-Seidel) and MUSCL (Monotone Upstream-centered Schemes for Conservation Law). MFGS has the largest

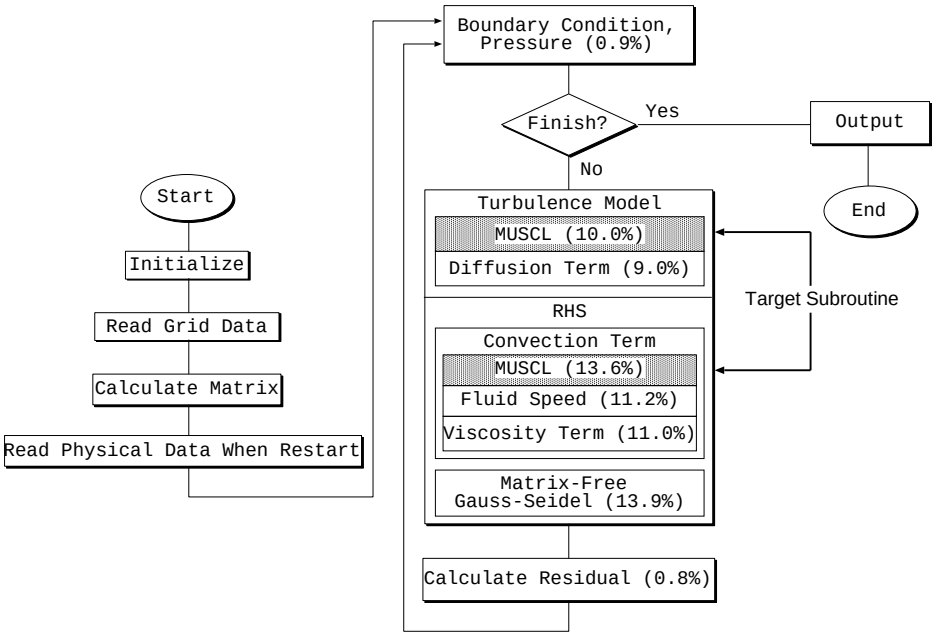

Figure 4.1: UPACS profiling result.

portion in execution time that is 13.9%, while MUSCL in convection term has consumed 13.6% of the total execution time. In addition to these two subroutines, which consumed 27.5%, MUSCL is also used in the turbulence model. MUSCL in turbulence model has consumed 10.0% of the total execution time resulting in total our coverage being almost 37.5%. However, in the preliminary study, Fujita *et al.* had mapped the data flow graphs of the MFGS subroutine directly onto FPGA [59]. Figure 4.2 shows the arithmetic pipeline for MFGS subroutine.

Therefore, in this study, we focus on MUSCL scheme subroutine, since this subroutine is used twice in core routine of UPACS from the turbulence model to calculate residual. The core routine occupies a large portion of total execution time and its ratio grows up more than 90% as grid size increases [24].

4.1.2 MUSCL Scheme

MUSCL (Monotone Upstream-centered Schemes for Conservation Laws), which is a method to improve the accuracy, was introduced by Bram van Leer in 1979 [60]. It provides highly accurate numerical solutions for a given system. In UPACS, MUSCL is used in the turbulence model (TMUSCL) and the convection term (CMUSCL) calculation. It extrapolates cell surface values from cell center values in equations (4.1) to (4.4), as shown in Figure 4.3:

$$q'_{i+1/2} = (q_{i+1} - q_i)/(\Delta_{i+1} + \Delta_i) \tag{4.1}$$

$$q'_{i-1/2} = (q_i - q_{i-1})/(\Delta_i + \Delta_{i-1}) \tag{4.2}$$

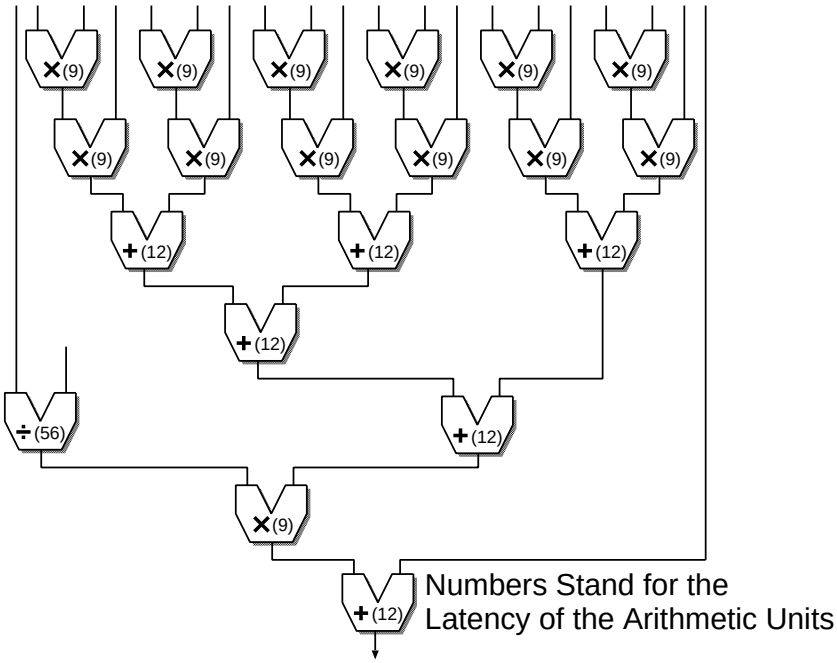

Figure 4.2: The structure of pipeline for MFGS.

$$q_{i\pm1/2} \cong q_i \pm \phi(r)\Delta_i q'_{i-1/2} \tag{4.3}$$

$$r = (q'_{i+1/2})/(q'_{i-1/2}) \tag{4.4}$$

where, q_i denotes the cell center value, Δ_i the distance between cell center and cell surface, $q_{1/2}$ the cell surface value, and $\phi(r)$ the Flux Limiter Function (FLF). The suffix i in the equations indicates the direction which can be extended to three dimensions. In addition, q_i consists of five physical values in UPACS, and there are data dependency between them. These physical values can represent density, velocity, pressure, viscosity or energy. FLFs are used to suppress oscillation of values, which often arises in the field where values change rapidly with a high order difference scheme. Six FLFs of MUSCL are shown in equations (4.5) to (4.10). In TMUSCL, four symmetry FLFs of no limiter,

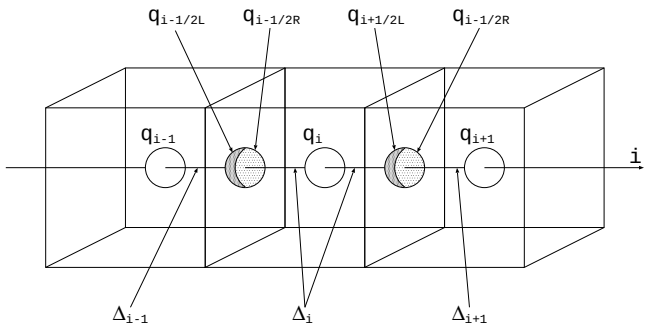

Figure 4.3: MUSCL scheme.

Table 4.1: Available flux limiter functions in each MUSCL.

Subroutine	Flux Limiter Functions
Turbulence MUSCL	no limiter van Leer van Albada minmod
2nd Convection MUSCL	no limiter van Leer van Albada minmod superbee
3rd Convection MUSCL	no limiter minmod Hemker-Koren

van Albada, van Leer and minmod limiter functions are included. On the other hand, 2nd order CMUSCL uses no limiter, van Albada, van Leer, minmod and superbee limiter functions. Finally, 3rd order CMUSCL consists of no limiter, minmod and Hemker-Koren limiter functions. Summary of available flux limiter functions for each MUSCL is shown in Table 4.1.

$$no\ limiter, \quad \phi(r) = 0.5(r+1) \tag{4.5}$$

$$van\ Leer, \quad \phi(r) = \frac{(r+|r|)}{(1+r+EPS)} \tag{4.6}$$

$$van\ Albada, \quad \phi(r) = \frac{(r^2+r)}{(1+r^2+EPS)} \tag{4.7}$$

$$minmod, \quad \phi(r) = max[0, min(1, r)] \tag{4.8}$$

$$superbee, \quad \phi(r) = [0, min(2r, 1), min(r, 2)] \tag{4.9}$$

$$Hemker-Koren, \quad \phi(r) = \frac{(r+2r^2)}{(2-r+2r^2)} \tag{4.10}$$

Here, r comes from (4.4), and EPS is a machine epsilon (1×10^{-16}**).**

4.2 Static Reconfiguration

The flexibility of FPGA raises the possibility for hardware configurations with software as needed to improve efficiency, robustness, security and capability to be programmable on the fly. A partially reconfigurable design of an FPGA consists of three major modules: the top module, static module and Reconfigurable Modules (RMs). The top module includes the static module and the RMs. The static module is a set of non-reconfigurable modules, while RMs are the dynamically reconfigurable parts of the design. The area of the device in which RMs is implemented is called Reconfigurable Partition (RP). Our first implementation uses static partial reconfiguration. This means the computation is not active during the reconfiguration process. While the partial data is sent into the FPGA, the rest of the device is stopped and brought up after the reconfiguration is completed.

As shown in previous section, MUSCL scheme is used twice in UPACS execution flow. Therefore, at the beginning, users must specify which limiter function they want to use at both parts. First, MUSCL is used in the turbulence model with four FLFs got involved. Then, MUSCL is used again in the convection term calculations part. 2nd order calculation for convection term involves 5 FLFs, and 3 FLFs are available for 3rd order calculation. However, in the CMUSCL calculation part, 2nd order CMUSCL and 3rd order CMUSCL are alternatively used. Here, partial reconfigurability of the FPGA and intractability of the bitstream is effective to meet the requirements. Figure 4.4 shows the block diagram of the system. In FPGA, the system consists of top MUSCL module, RP module for FLFs and on-chip memory using block RAM. The system is connected with the host PC that contains all FLFs bitstream files. The connection is through UART via a JTAG port. JTAG was chosen for configuration port because of quick testing and debugging. Although ICAP port is a good alternative, it requires user-designed partial reconfiguration controller such as custom state machine or embedded processor. Moreover, when the partial bit files are stored in host PC, JTAG is convenient compared to self configurations using ICAP.

Since each MUSCL function has similar structure except FLFs, we can design a single MUSCL module with all FLFs for TMUSCL, 2nd order CMUSCL and 3rd order CMUSCL. However, it becomes a large hardware, which is difficult to be implemented on a single FPGA. The straightforward way is designing three MUSCLs each of which has their own FLFs. Although this approach reduces the hardware, we must provide three independent designs. Our approach is to provide a single design whose FLF can be replaced by making the best use of partial reconfiguration. The total required hardware and power usage can be minimized, since it provides only a single FLF required in the target application. When the execution of UPACS starts and the functions required for MUSCL is decided, an appropriate FLF module is loaded by using the partial reconfiguration while the other part of FPGA is remaining unaffected.

Each FLF module has the same inputs and outputs, thus, it can be specified in the HDL description as the functional modules with the RP attribute in the description of the MUSCL top module. Multiple instances corresponding to each FLF can be defined for such a single functional module.

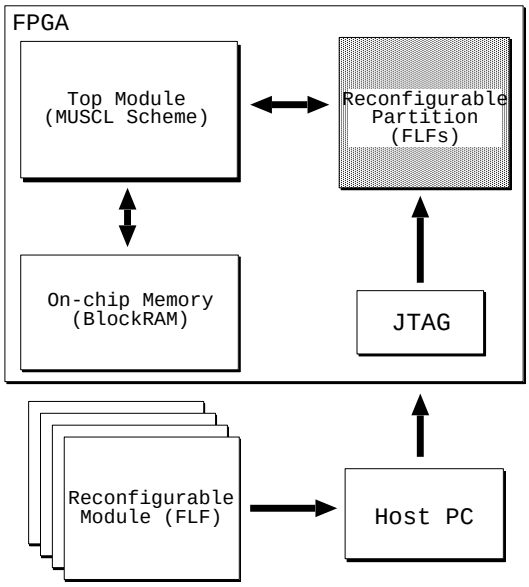

Figure 4.4: High level system overview of static reconfiguration design.

Software tools as NGDBuild, MAP and PAR detect the reconfigurable partition attribute on the instance and process it correctly [17].

4.2.1 Design and Implementation

Here, Xilinx Virtex-6 FPGA (XC6VLX240T-1FF1156), which supports a partial reconfiguration, was chosen as a target device. MUSCL scheme is implemented as a top, static module with a reconfigurable partition. Using its reconfigurable partition, FLFs are implemented as partial reconfigurable modules. The datapath can be obtained from partially simplified data-flow representation of the algorithm shown in Figure 4.5. By inserting shift registers in the datapath, the fundamental structure of the pipeline is designed. Xilinx CORE Generator is used to provide the core for floating-point adder, subtractor, multiplier, divider and shift register. Efficient memory system is primary concern over here. Therefore, memory system implementation is based on proposed work by Morishita *et al.* [61]. To solve 3D model of fluid dynamics problem, pipeline datapath is implemented three times inside an FPGA.

All modules are described using Verilog HDL and simulated with Xilinx ISim Simulator. The modules are synthesized and used resources are measured using Xilinx ISE 12.4. Floor-planning, constraint entry and Design Rule Checks (DRCs) are all accessed through the PlanAhead 12.4 software environment which supports a partial reconfiguration flow. In order to demonstrate that our system works on the real FPGA, Xilinx ML605 board is used with 200 MHz operating frequency. All modules were also implemented using IEEE 754 standard 64-bit double precision floating-point arithmetic[1]. Here, the floating-point computational module is based on the Xilinx Floating-Point Operator v5.0 incorporated into Xilinx ISE 12.4 software. The Floating-Point Operator v5.0 is an IP

[1]Further specification of IEEE standard 754 floating-point numbers is given in Appendix A.

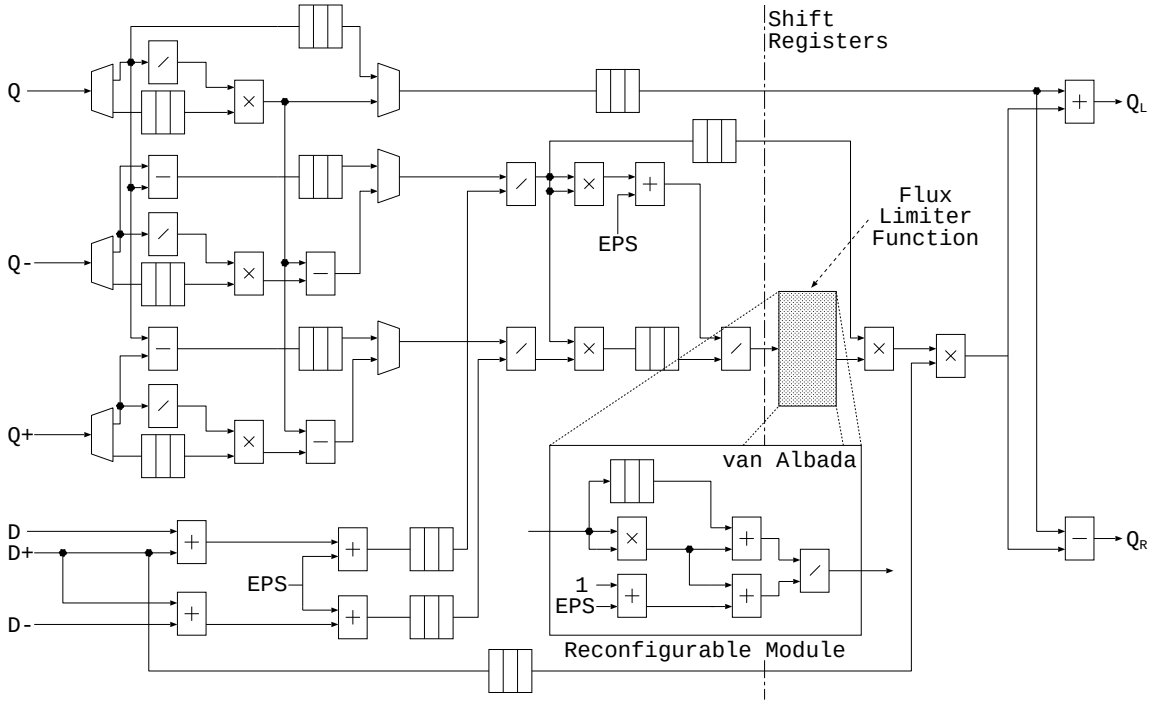

Figure 4.5: MUSCL pipeline with *van Albada* limiter function.

core for handling floating-point operations, and it is configurable by the user specifications. In order to generate high performance computation unit, the level of DSP48E usage is set to the maximum to get the desired output.

Inputs given to the pipeline are vectors each of which consists of five physical values mentioned before. At one time, only one FLF is used and employed in the FPGA. All FLFs are synthesized separately from the top module. The top MUSCL and reconfigurable FLF modules consist of many arithmetic functions. The parameters used for each computing unit are shown in Table 4.2. Adder and Subtractor are set to 14 clock cycles per operation using high-speed mode. In addition, Multiplier takes 16 clock cycles with 11 DSP48E modules. The latency of Divider is set to be 57. Although it is possible to decrease the divider pipeline latency, it will severely degrade the clock frequency.

Table 4.2: Data of used computing units.

Units	Latency	Registers	LUTs	DSP48E
Adder	14	947	797	3
Subtractor	14	947	798	3
Multiplier	16	483	362	11
Divider	57	5973	3261	0
Comparator	1	0	128	0

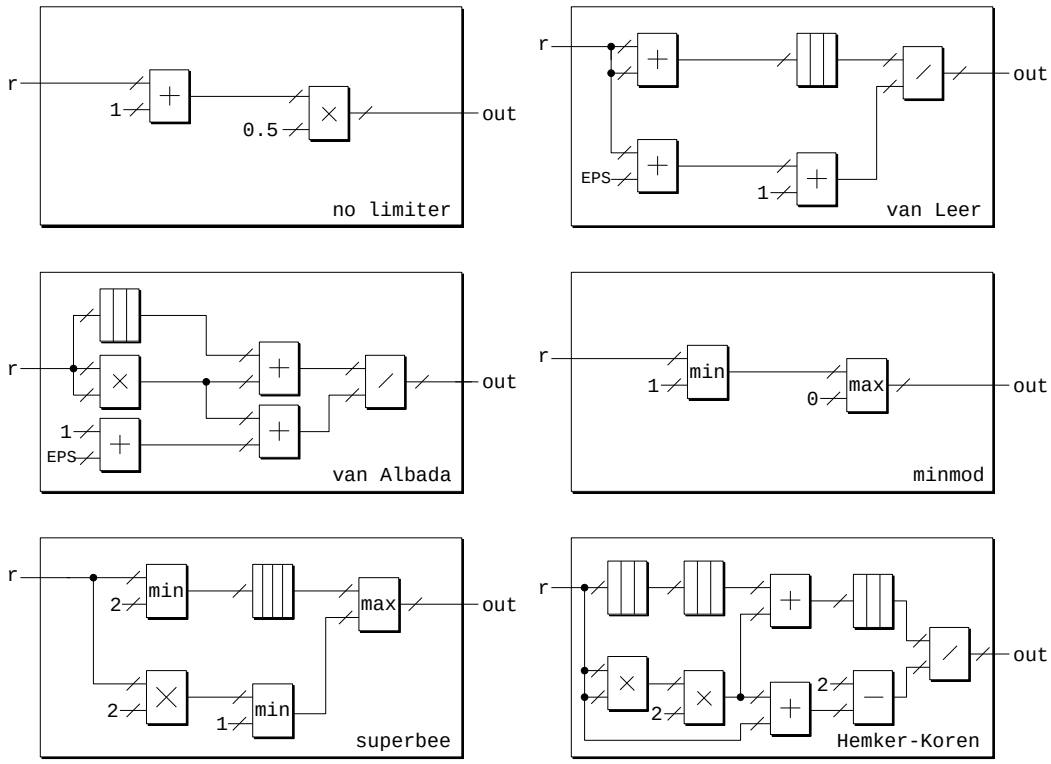

Figure 4.6: Implemented flux limiter functions.

The designs for all FLFs are shown in Figure 4.6. The one with the smallest clock cycles is minmod limiter function, which only requires 2 clock cycles. The largest latency is by Hemker-Koren limiter function, which requires 117 clock cycles to get the result. Moreover, van Albada and van Leer limiter functions require 87 and 85 clock cycles, respectively. In these FLF modules, shift registers are used to synchronize the input value. Shift registers are 64 bit width and can take various clock cycle depth depending on the situation. Machine epsilon (1×10^{-16}) and constant value 1.0 are also used in these FLFs modules. The remaining minmod and superbee limiter functions require comparator modules, which are used for minimum and maximum value comparison.

4.2.2 Evaluation

In this section, evaluation results are shown in order to demonstrate the concept of the system. The following three designs are evaluated and compared. Sample of $100 \times 100 \times 100$ grid size is used for evaluation.

- *design-1:* One static module of MUSCL scheme with all FLFs.
- *design-2:* Three static modules with the associate FLFs for TMUSCL, 2nd order CMUSCL and 3rd order CMUSCL.
- *design-3:* One top module of MUSCL scheme with partial reconfiguration FLFs. Statically reconfigurable design.

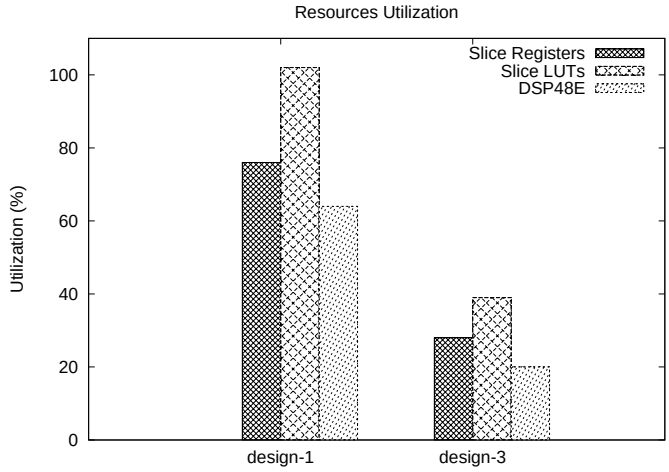

Figure 4.7: Resource usage in *design-1* and *design-3*.

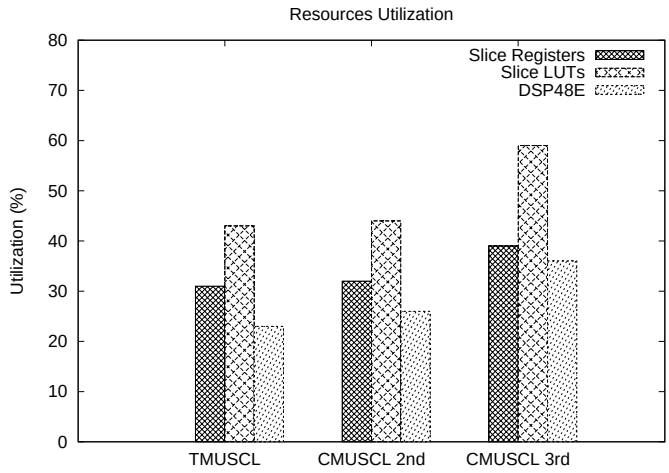

Figure 4.8: Resource usage in *design-2*.

4.2.2.1 Resources Utilization

The amount of required slice registers, slice LUTs and DSP48E is evaluated when the design is synthesized. The results for all three designs are shown in Figure 4.7 and Figure 4.8. The results show that *design-3* has the lowest resource utilization compared to *design-1* and *design-2*. The largest resource is occupied by *design-1*, which is the slice LUT usage had exceeded 100%, so it cannot be implemented on a single chip. *Design-2* has less resource required for implementation compared to *design-1*. However, *design-2* will require three different FPGAs or three times full reconfiguration on an FPGA.

In *design-2* TMUSCL has four FLFs: no limiter, van Leer, van Albada and minmod. CMUSCL 2nd has five FLFs: no limiter, van Leer, van Albada, minmod and superbee. CMUSCL 3rd has three FLFs: no limiter, minmod and Hemker-Koren. In *design-3*, all FLFs share the same reconfigurable

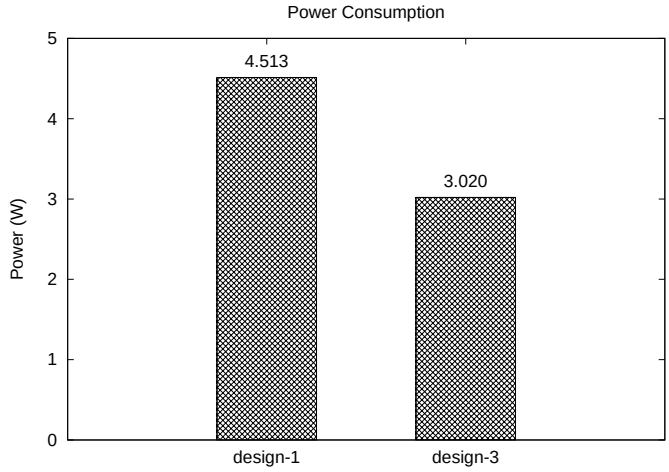

Figure 4.9: Total on-chip power for *design-1* and *design-3*.

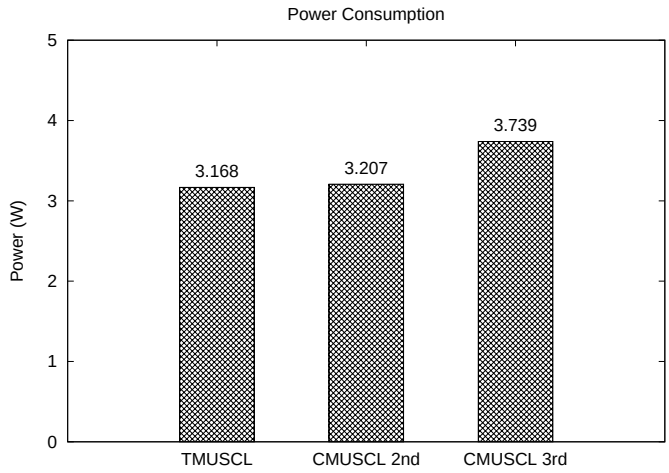

Figure 4.10: Total on-chip power for *design-2*.

partition in MUSCL. In contrast, the overhead in resource utilization is small. It comes from unused resource in reconfigurable partition. Since the partition size is fixed, each FLF is not fully used the resources available. However, in this case, the overhead is very small and can be negligible.

4.2.2.2 Power Consumption

Power consumption becomes one of the biggest concerns in FPGA design as capacity and performance of FPGAs have been increased. The results of total power consumption for all three designs are shown in Figure 4.9 and Figure 4.10. In *design-1*, Xilinx XPower Estimator(XPE) 13.3 is used since the design cannot be implemented in an FPGA. Resources usage from synthesis result of *design-1* is used as an input to XPE to estimate total power usage. In short, it shows that *design-3* has the lowest power consumption compared to the other two designs.

On the other hand, the highest power is consumed by *design-1*. In *design-1*, the total power is high because of static power by unused limiter functions module. In *design-3*, the total power is only consumed by MUSCL module and the required limiter function. Moreover, the power for *design-2* is less than that for *design-1*, but *design-3* is advantageous even when it is compared with *design-2*. This is because *design-2* also has unused limiter functions in every TMUSCL, CMUSCL 2nd and CMUSCL 3rd during operation. This increases the static power in *design-2*.

4.2.2.3 Configuration Time

The configuration time for full reconfiguration and partial reconfiguration are compared. In this case, only *design-2* and *design-3* are evaluated, since *design-1* cannot be implemented on a single chip. In the case of JTAG configuration, for Virtex-6 device, configuration time is given by:

$$configuration\ time = \frac{(2,044 + bits\ in\ bitstream)}{TCK\ frequency} \tag{4.11}$$

where *bits in bitstream* is size of the configuration bitstream in bits and *TCK frequency* is maximum configuration TCK (Test Clock) frequency and used for boundary-scan operations. In this case, for Virtex-6 device with -1 speed grade, TCK frequency is 66 MHz. 2,044 is the total number of clock cycles needed for pre-processing and post-processing while programming the bitstream to FPGA.

In full reconfiguration, each MUSCL module bitstream size is 9,017 KB. Based on the Eq. (4.11), the configuration time is equal to 1.119 sec. On the other hand, bitstream size for each partial reconfiguration bit file for the 2nd order FLFs is 255 KB. This means that the configuration time is equal to 0.031 sec. In the case of the 3rd order FLFs, partial bitstream size is 266 KB and corresponding configuration time is 0.033 sec. In short, the partial reconfiguration method accelerated the configuration speed by 34 times. In other words, execution time is not so degraded compared with *design-2* when the FLFs are switched dynamically.

In this implementation, the MUSCL will stop working when the FLF is loaded to reconfigurable partition in the FPGA. However, since the time taken to load the FLF is between 0.031 sec and 0.033 sec, this small overhead is acceptable. Even if low speed JTAG is used through iMPACT tool to reconfigure the FPGA, the time to change from one FLF to the other is less than one second, and hard to been recognized by human eyes. Usually, the partial reconfiguration is done once when TMUSCL is changed into 2nd-order CMUSCL or 3rd-order CMUSCL in a job, which requires large execution time. Thus, the overhead for partial reconfiguration will be acceptable.

Table 4.3: Total clock-cycle in TMUSCL for each respective limiter functions.

Flux Limiter Function	# Clock-cycle
no limiter	233
van Albada	290
van Leer	288
minmod	204

Table 4.4: Total clock-cycle in CMUSCL for each respective limiter functions.

Flux Limiter Function	# Clock-cycle
no limiter 2nd	233
van Albada 2nd	290
van Leer 2nd	288
minmod 2nd	204
superbee 2nd	222
no limiter 3rd	233
minmod 3rd	204
Hemker-Koren 3rd	320

4.2.2.4 Performance

MUSCL is implemented with pipelined structure and the clock cycles are measured. Total clock cycles for MUSCL with each FLF are shown in Table 4.3 and Table 4.4. The number of clock cycles is corresponding to the time for solving iteration. In TMUSCL, van Albada is the largest, and it takes 290 clock cycles. In CMUSCL, Hemker-Koren requires 320 clock cycles to get the result.

The execution time in MUSCL with partial reconfigurable FLFs is compared with the execution time by software. In software, Core 2 Duo 2.4 GHz executes MUSCL with Linux Kernel 2.6.18 operating system. The compiler used is GNU Fortran 4.1.2. The execution time to solve $100 \times 100 \times 100$ iterative calculation is measured by using *call cpu_time* in Fortran 90 language and the 3rd order Hemker-Koren is selected for comparison, since it has the largest clock cycles. In software, the execution time took 0.08399 sec. while it takes 320 clock cycles to finish one iterative calculation in the FPGA. Adding the time for I/O sending the data sequentially, it took 1,000,320 clock cycles to finish the whole simulation. 1,000,000 comes from the grid size corresponding to the total mesh points. Since the operating frequency in the FPGA is 200 MHz, the total execution time is 5.0016 x 10^{-3} sec. That is, by execution of CMUSCL in FPGA, about 17 times acceleration is expected. Since the grid size will grow large and take a lot of iterations, configuration time will not caused a bottleneck to the system. Furthermore, overhead also did not influence the operating frequency.

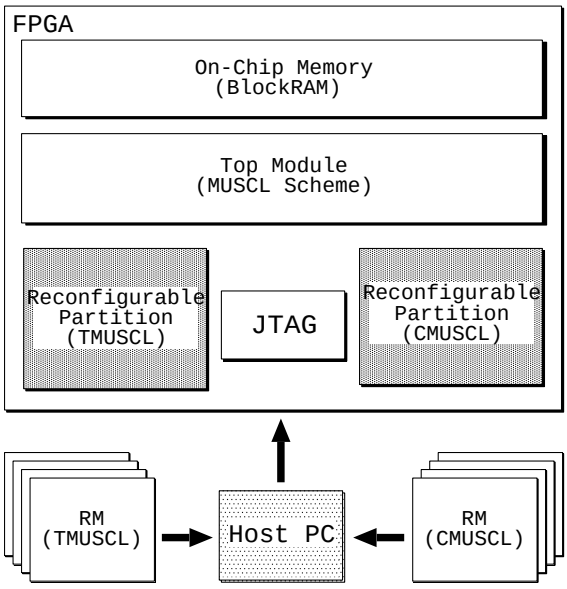

Figure 4.11: High level system overview of dynamic reconfiguration design.

4.3 Dynamic Reconfiguration

Our next implementation of the MUSCL scheme uses dynamic reconfiguration for flux limiter functions. Dynamic reconfiguration is an active partial reconfiguration, which permits to change the part of the device while the rest of an FPGA is still running.

In this design, we created a separate reconfigurable partition for TMUSCL and CMUSCL. As mentioned before, MUSCL scheme is used twice in UPACS execution flow. Therefore, at the beginning, users must specify which limiter function they want to use at both parts. First, MUSCL is used in the turbulence model with four Flux Limiter Functions (FLF) got involved. Then, MUSCL is used again in the convection term calculations part. 2nd order calculation for the convection term involves 5 FLFs, and 3 FLFs are available for 3rd order calculation. However, in CMUSCL calculation part, 2nd order CMUSCL and 3rd order CMUSCL are alternatively used. Figure 4.11 shows the block diagram of the system. In FPGA, the system consists of top MUSCL module; two reconfigurable partitions module for FLFs (TMUSCL and CMUSCL); and on-chip memory using Block RAM. The system is connected with the host PC that contains all FLFs bitstreams for both TMUSCL and CMUSCL. The connection is through UART via a JTAG port.

In this implementation, design strategy is the same as statically reconfigurable design except for the reconfigurable partitions. At start up, the top MUSCL module with blank bitstream in both TMUSCL and CMUSCL is loaded to target FPGA. Then, an appropriate FLF module for the turbulence model is loaded to the TMUSCL reconfigurable partition and start the computation of TMUSCL. During calculation of the turbulence model, users can load the partial bitstream of required FLF for the convection term calculation in CMUSCL reconfigurable partition. After the turbulence calculation finishes, the calculation for convection term can start immediately. Again,

in both reconfigurable partitions, each FLF module has the same inputs and outputs, thus it can be specified in the HDL description as the functional modules with the reconfigurable partition attribute in the description of the MUSCL top module.

4.3.1 Design and Implementation

This section examines the implementation of dynamic reconfiguration design. We use the same FPGA, Xilinx Virtex-6 (XC6VLX240T-1FF1156), which supports a partial reconfiguration as a target device. The top MUSCL and reconfigurable FLF modules consist of many arithmetic functions. The parameters used for each computing unit is the same as in static reconfiguration design and shown in Table 4.2.

4.3.1.1 Bottom-Up Synthesis

Bottom-up synthesis is synthesis of the design by modules, from bottom modules to top module. This synthesis technique requires that a separate netlist is written for each partition ensuring that each portion of the design is synthesized independently. MUSCL scheme as a top level logic is synthesized with black box for the reconfigurable partitions. In this case, TMUSCL and CMUSCL modules are defined as a black box in top module synthesis. TMUSCL and CMUSCL modules are synthesized beforehand to provide the netlist to top module. The modules are synthesized, and resources utilization is measured using Xilinx ISE 12.4. All modules are described using Verilog HDL and simulated with Xilinx ISim Simulator.

4.3.1.2 Floorplanning

In this step, it is required to perform manual floorplanning for reconfigurable partitions, which requires knowledge of the physical architecture of FPGA and understanding of how to floorplan for optimal performance and area. Therefore, we manually floorplan the TMUSCL and CMUSCL reconfigurable regions through PlanAhead 12.4 software environment. Figure 5.5 shows reconfigurable partition for TMUSCL and CMUSCL separately. After that, constraint entry and Design Rule Checks (DRCs) are performed.

In order to demonstrate that our system works on the real FPGA, Xilinx ML605 board is used with 200 MHz operating frequency. All modules are implemented using IEEE 754 standard 64-bit double precision floating-point arithmetic. Here, the floating-point computational module is based on the Xilinx Floating-Point Operator v5.0 incorporated into Xilinx ISE 12.4 software. The Floating-Point Operator v5.0 is an IP core for handling floating-point operations, and it is configurable by the user specifications. In order to generate high performance computation unit, the level of DSP48E usage is set to the maximum to get the desired output.

At the beginning, MUSCL scheme without any limiter functions for TMUSCL and CMUSCL are loaded to FPGA. This is done by inserting the blank bit file to both reconfigurable partitions.

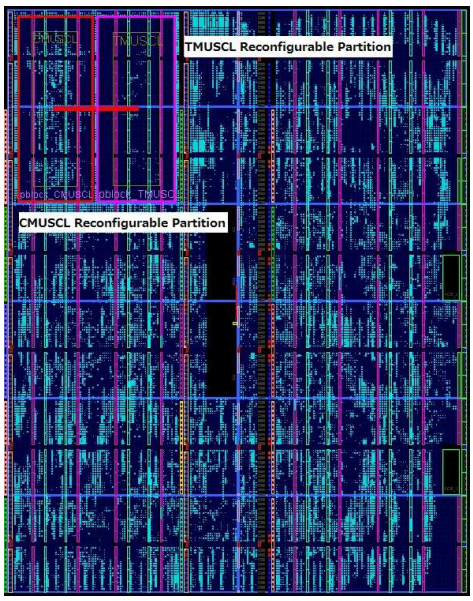

Figure 4.12: Floorplan of FLF reconfigurable partitions for dynamic reconfiguration design.

Then, users can decide which limiter functions they want to use for TMUSCL. After limiter function in TMUSCL is successfully loaded, calculation is started. While MUSCL is operating for TMUSCL calculation, users can load the desired limiter functions for CMUSCL in CMUSCL reconfigurable partition. When TMUSCL calculation is finished, it can immediately start the CMUSCL calculation since CMUSCL limiter function is readily available.

Implemented flux limiter functions are the same as static reconfiguration design as shown in Figure 4.6. The difference lies in that we separately assign TMUSCL FLFs to TMUSCL reconfigurable partition and CMUSCL FLFs to CMUSCL reconfigurable partition, respectively. Therefore, this design strategy occupies 2 times reconfigurable partition area compared to static reconfiguration design.

4.3.2 Evaluation

In this section, evaluation results are shown and discussed. The following four designs are evaluated and compared. Same sample of $100 \times 100 \times 100$ grid size is used for evaluation.

- *design-1*: One static module MUSCL scheme with all FLFs.
- *design-2*: Three static modules of MUSCL scheme with the associate FLFs for TMUSCL, 2nd order CMUSCL and 3rd order CMUSCL.
- *design-3*: One top module of MUSCL scheme with one reconfigurable partition for TMUSCL and CMUSCL. Statically reconfigurable design.
- *design-4*: One top module of MUSCL scheme with two reconfigurable partitions for TMUSCL and CMUSCL respectively. Dynamically reconfigurable design.

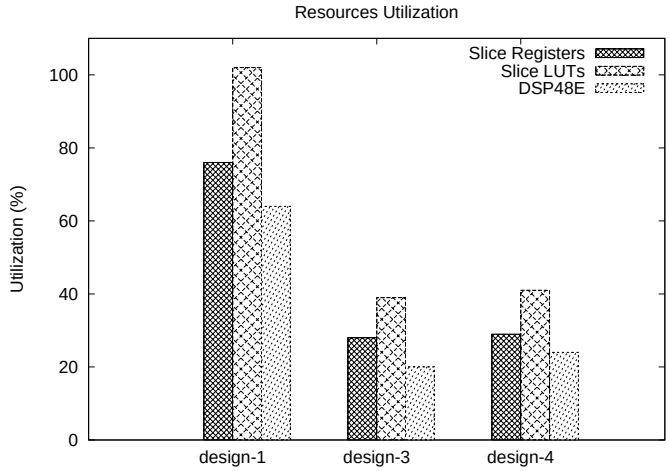

Figure 4.13: Resource usage in *design-1*, *design-3* and *design-4*.

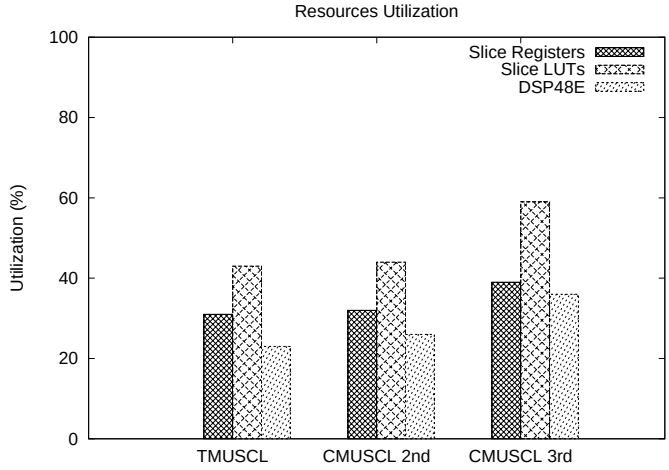

Figure 4.14: Resource usage in *design-2*.

4.3.2.1 Resources Utilization

The amount of resources usage for slice registers, slice LUTs and DSP48E is determined when the design is synthesized. The results for all four designs are shown in Figure 4.13 and Figure 4.14. The results show that *design-3* has the lowest resource utilization compared to *design-1*, *design-2* and *design-4*. This is because *design-3* only provides a single reconfigurable partition compared to two reconfigurable partitions for *design-4*. The largest resource is occupied by *design-1*. The slice LUT usage for *design-1* had exceeded 100%, therefore it is not possible to be implemented on a single FPGA. On the other hand, *design-2* has less resource required for implementation compared to *design-1*. However, *design-2* will require three different FPGAs or three times full reconfiguration on an FPGA.

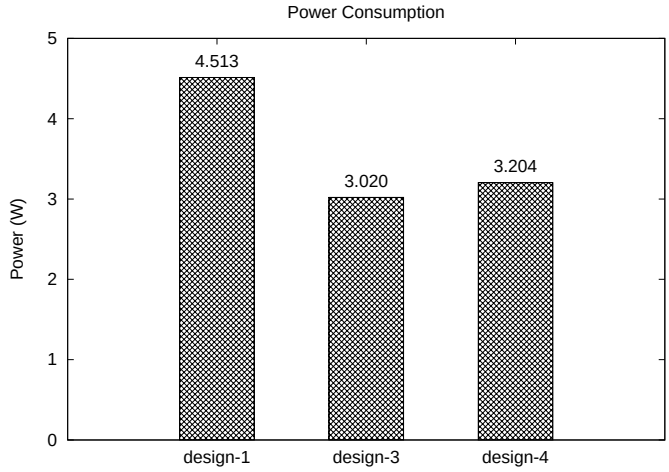

Figure 4.15: Total on-chip power for *design-1*, *design-3* and *design-4*.

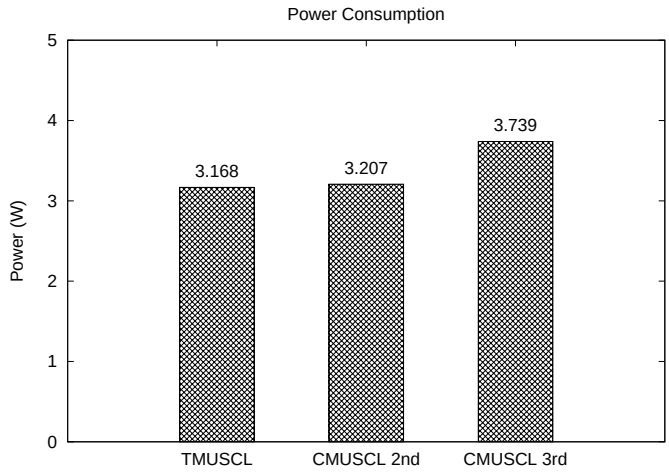

Figure 4.16: Total on-chip power for *design-2*.

4.3.2.2 Power Consumption

Power consumption is one of the crucial issues among FPGA users. Power usage becomes more important as FPGAs increase in logic capacity and performance. The results of total power consumption for all four designs are shown in Figure 4.15 and Figure 4.16. For *design-1*, Xilinx XPower Estimator (XPE) 13.3 is used since the design cannot be implemented in a real chip. It shows that *design-3* has the lowest power consumption compared to other three designs. However, the power increased in *design-4* is not significantly higher than *design-3*.

In *design-1*, the total power is increased because of static power by unused limiter functions. In *design-3* and *design-4*, the total power is only consumed by top MUSCL and the required limiter function. The power for *design-2* is less than that for *design-1*, but *design-4* is advantageous even when it is compared with *design-2*.

4.3.2.3 Configuration Time

The configuration time for full reconfiguration and partial reconfiguration are compared. In this case, only *design-2*, *design-3* and *design-4* are evaluated, since *design-1* cannot be implemented. In the case of JTAG configuration, for Virtex-6 device, configuration time is calculated using Eq. (4.11)

In full reconfiguration, each MUSCL module bitstream size is 9,017 KB. Based on the above formula, the configuration time to program into an FPGA is equal to 1.119 sec. In *design-3*, bitstream size for each partial reconfiguration bit file for the 2nd order FLFs is 255 KB. This means that the configuration time for partial bitstreams in *design-3* is equal to 0.031 sec. In the case of the 3rd order FLFs, partial bitstream size is 266 KB and corresponding configuration time is 0.033 sec. In *design-4*, partial bit stream size for TMUSCL is 554 KB and for CMUSCL is 577 KB. Therefore the corresponding configuration time for partial bitstreams of TMUSCL is 0.069 sec and partial bitstreams for CMUSCL is 0.072 sec. In short, the partial reconfiguration method in *design-4* accelerated the configuration speed by 15 times compared to full reconfiguration design. In other words, execution time is not so degraded compared with *design-2* when the FLFs are switched dynamically.

In comparison to statically reconfigurable design, *design-3*, MUSCL will stop working when the FLF is loaded to reconfigurable partition in the FPGA. Therefore, there is small overhead while configuring the reconfigurable partition to change FLF. However, in *design-4* this overhead is eliminated by introducing two reconfigurable partition for both TMUSCL and CMUSCL.

4.3.2.4 Performance

Dynamically reconfigurable MUSCL scheme is implemented in a single FPGA and the clock cycles are measured. In the same way as static reconfiguration design, grid size to solve $100 \times 100 \times 100$ iterative calculation is used. Again, 3rd order Hemker-Koren limiter functions is selected for comparison, since it has the largest clock cycles. In software, MUSCL is executed by Core 2 Duo 2.4 GHz with Linux Kernel 2.6.18 operating system. In hardware, MUSCL is executed using ML605 board with 200 MHz operating frequency. As a result, total clock cycles for MUSCL with each FLF is the same as static reconfiguration design as shown in Table 4.3 and Table 4.4. The number of clock cycles is corresponding to the time for solving an iteration. In short, by execution of MUSCL in FPGA, about 17 times more acceleration is expected.

In addition, total execution time for static reconfiguration design, *design-3* and dynamic reconfiguration design, *design-4* are compared. Considering the configuration time into account, the total execution time is given by:

$$T_{EX} = T_{FR} + T_{PR} + T_{TMUSCL} + T_{PR} + T_{CMUSCL} \tag{4.12}$$

where

- T_{EX} - total execution time
- T_{FR} - time taken for full reconfiguration
- T_{PR} - time taken for partial reconfiguration
- T_{TMUSCL} - processing time in turbulence MUSCL
- T_{CMUSCL} - processing time in convection MUSCL

In *design-3*, to change from TMUSCL limiter function to CMUSCL limiter function takes T_{PR} to do the reconfiguration. This is because TMUSCL and CMUSCL limiter functions are occupied by the same reconfigurable partition on an FPGA. Meanwhile, in *design-4*, while FPGA is processing TMUSCL, users can load the CMUSCL limiter function. Therefore, in *design-4*, T_{PR} to download CMUSCL limiter function is eliminated resulting in shorter total execution time, T_{EX}.

To conclude, clearly *design-3* and *design-4* outperformed *design-1* and *design-2* in overall performance. However, there is a small trade off between *design-3* and *design-4*. Even though *design-3* is superior in terms of resource utilization and power consumption, *design-4* has an advantage from the viewpoint of performance. This is because, in *design-3* MUSCL operation must be stopped during configuration for TMUSCL and CMUSCL, which is not needed in *design-4*.

4.4 Summary

In this chapter, we have discussed the implementation of target subroutine in UPACS software package. UPACS is a convenient CFD package that allows users to select various sets of solutions. UPACS solver together with support utilities has proven its effectiveness in a simulating flow around complex configurations using multi-block structured grid scheme. However, it is hard to be implemented even on a high capacity FPGAs because of a large hardware amount. To address this problem, exploitation of partial reconfigurability in recent FPGAs was considered. Partially reconfigurable flux limiter functions in MUSCL scheme were implemented. Two types of partial reconfiguration were explored that are static and dynamic partial reconfiguration.

In the static reconfiguration design, MUSCL scheme using partial reconfiguration platform has been implemented to reduce the required hardware resource, power consumption, and configuration time. It also aims to improve the performance. This implementation successfully reduced the resource utilization by 44% to 63%. Also, power consumption was reduced by 33%. Configuration speed was accelerated 34 times faster. Overall speed-up at least 17 times in performance compared to software execution was achieved.

In the dynamic reconfiguration design, cost effective implementation of MUSCL scheme using partial reconfiguration strategy has been explored. Two reconfigurable partitions were implemented, which are for TMUSCL and CMUSCL respectively. This design dynamically allows users to load CMUSCL limiter functions during the TMUSCL computation. This implementation successfully reduced the resource utilization by 60%. Also, power consumption was reduced by 29%. Configuration speed was accelerated 15 times faster compared to full reconfiguration design. Performance evaluation also shows that 17 times more acceleration was achieved compared to the Intel Core 2 Duo at 2.4 GHz.

Chapter 5

FaSTAR Code Implementation

In the previous chapter, we have studied a single MUSCL scheme used in UPACS and implemented on an FPGA. Partial reconfiguration is applied to flux limiter functions available in MUSCL scheme. We had successfully took both statically and dynamically reconfigurable strategies. Although the total hardware is reduced, the effect is limited since it only occupies a small portion of the target FPGA.

In this chapter, we will study the FaSTAR code implementation using partial reconfiguration on an FPGA. We try to extend the applications field of partial reconfiguration to large flux calculation scheme in advection term computation used in FaSTAR. Five flux calculation schemes are analyzed and studied. Section 5.3 discusses the design and implementation, followed by Section 5.4 for performance evaluation.

5.1 FaSTAR

FaSTAR is a CFD software package developed by JAXA to simulate compressible flow using unstructured grids. FaSTAR consists of many solvers with multiple solutions. Its source code is written in Fortran 90 with MPI. By choosing certain solvers, users can select various solutions supported by the application and run simulation in parallel with their systems without specific software tunings. Users just are requested to prepare parameter file and grid data file before the simulation. By selecting a combination of solvers, a user can simulate the target object with desired solutions. When the partial reconfiguration is applied to selectable solutions, the profiling is required at first. Then, the part in which multiple schemes are available is picked up from the subroutines which take a long computation time.

5.1.1 Profiling

As the first step in our study, we profiled the execution time of FaSTAR to find out which routines consume the highest percentage. Compiler used is Intel Fortran Compiler 10.1 on Intel Core2Duo processor at 2.66 GHz with Linux Kernel 2.6. The profiling results are shown in Figure 5.1.

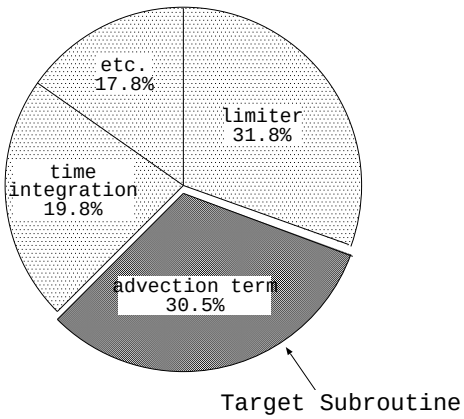

Figure 5.1: FaSTAR profiling result.

Code for single core without MPI was compiled. The result indicates that more than 60% of the total execution time is occupied by two calculations part: `limiter` and `advection term`. FaSTAR limiter part is difficult for implementation in reconfigurable hardware because of its complicated iteration. Here, we selected the advection term, since it occupies a large part of the total computation time, and has a selectable function whose hardware requirement is relatively large.

5.1.2 Target Subroutine

Advection term consists of three subroutines:

- `pre-processing`,
- `flux calculation`, and
- `surface integral of flux`.

In `pre-processing`, data such as cell-A number are prepared. Using these data, flux is obtained by applying to scheme equations in `flux calculation` part. Here, we try to apply partial reconfiguration to make reconfigurable scheme selection as a focus in this study. Then, the finite volume method for discretization of the space is processed in `surface integral of flux`. Akamine *et al.* have reported the study for this part in [62].

5.2 Flux Calculation Scheme

In flux calculation subroutines, there are five schemes available for selection: Roe's scheme, HLLE scheme, HLLEW scheme, AUSM$^+$-up scheme and SLAU scheme [63–67]. We describe how to compute the inviscid flux using Riemann solver approximation. As shown in Figure 5.2, conserved quantities, Q_{na} and Q_{nb} in cell A and cell B were used to determine flux, F_n. In this case, with the respective to cell A, the normal vector, d_s has the same right direction with flux. On the other hand,

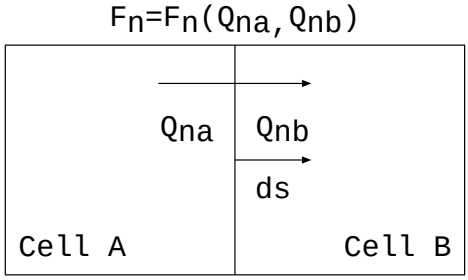

Figure 5.2: Flux in definition of cell surface boundary.

with respect to cell B, the opposite direction is positive. In all schemes, F_n is evaluated when it is viewed from the cell A which means right direction is positive. If it is viewed from cell B, it will become -F_n.

5.2.1 Roe's Scheme

The method proposed by Roe [63] is based on Euler's equation into windward of the linearization. The numerical flux function can be written as:

$$F_n = \frac{1}{2}[f(Q_{na}) + f(Q_{nb}) - |A|_{ave}(Q_{nb} - Q_{na})] \tag{5.1}$$

where

$$f(Q_{na}) = \begin{pmatrix} \rho_a u_{na} \\ \rho_a u_{na}^2 + p_a \\ \rho_a u_{na} u_{t1a} \\ \rho_a u_{na} u_{t2a} \\ \rho_a u_{na} H_a \end{pmatrix} = \rho_a u_{na} \begin{pmatrix} 1 \\ u_{na} \\ u_{t1a} \\ u_{t2a} \\ H_a \end{pmatrix} + \begin{pmatrix} 0 \\ p_a \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{5.2}$$

$$Q_{na} = \begin{pmatrix} \rho_a \\ \rho_a u_{na} \\ \rho_a u_{t1a} \\ \rho_a u_{t2a} \\ e_a \end{pmatrix} \tag{5.3}$$

Note that $f(Q_{nb})$ and Q_{nb} are calculated in the same way for $f(Q_{na})$ and Q_{na}. In addition, A is the Jacobian matrix for flux,

$$|A|_{ave} = R_{ave}|\Lambda|_{ave}R_{ave}^{-1} \tag{5.4}$$

where R is the matrix with right eigenvector, Λ is matrix with eigenvalue and R^{-1} is matrix with left eigenvector. The matrix with subscript *ave*, is composed by variables which obtained by Roe average. The Roe average can be expressed as follows:

$$\begin{aligned}
\rho_{ave} &= \sqrt{\rho_a \rho_b} \\
u_{ave} &= \frac{\sqrt{\rho_a} u_a + \sqrt{\rho_b} u_b}{\sqrt{\rho_a} + \sqrt{\rho_b}} \\
H_{ave} &= \frac{\sqrt{\rho_a} H_a + \sqrt{\rho_b} H_b}{\sqrt{\rho_a} + \sqrt{\rho_b}} \\
c_{ave} &= \sqrt{(\gamma - 1)(H_{ave} - \frac{1}{2} u_{ave}^2)}
\end{aligned} \tag{5.5}$$

where, c denotes the speed of sound, H the total enthalpy per unit mass, and γ specific weight representing the force exerted by gravity on a unit volume. Therefore, $\gamma = \rho \times g$. Then, we can write the matrices in details as follows:

$$R = \begin{bmatrix}
1 & 0 & 0 & \frac{\rho}{2c} & \frac{\rho}{2c} \\
u & 0 & 0 & \frac{\rho(u+c)}{2c} & \frac{\rho(u-c)}{2c} \\
v & 0 & -\rho & \frac{\rho v}{2c} & \frac{\rho v}{2c} \\
w & \rho & 0 & \frac{\rho w}{2c} & \frac{\rho w}{2c} \\
\frac{u^2+v^2+w^2}{2} & \rho w & -\rho v & \frac{\rho H + \rho u c}{2c} & \frac{\rho H + \rho u c}{2c}
\end{bmatrix} \tag{5.6}$$

$$|\Lambda| = \begin{bmatrix}
|u| & 0 & 0 & 0 & 0 \\
0 & |u| & 0 & 0 & 0 \\
0 & 0 & |u| & 0 & 0 \\
0 & 0 & 0 & |u + c| & 0 \\
0 & 0 & 0 & 0 & |u - c|
\end{bmatrix} \tag{5.7}$$

Note here that if *ave* subscript can be omitted, $u = u_n$, $v = u_{t1}$ and $w = u_{t2}$.

5.2.2 HLLE Scheme

Einfeldt [64] discussed an adapted version of the HLL scheme [68], called HLLE (*Harten-Lax-van Leer-Einfeldt*) scheme, which can be considered as a modification of Roe's scheme. This scheme is a stable procedure that is solved by approximation of two characterized waves, which are valid for the flow with a strong expansion. However, there is a disadvantage that the numerical viscosity is large. Total flux can be calculated using the following equations:

$$F_n = \frac{b^+ f(Q_{na}) - b^- f(Q_{nb})}{b^+ - b^-} + \frac{b^+ b^-}{b^+ - b^-}(Q_{nb} - Q_{na}) \tag{5.8}$$

where

$$\begin{aligned} b^+ =& max(u_{ave} + c_{ave}, u_{nb} + c_b, 0) \\ b^- =& min(u_{ave} - c_{ave}, u_{na} - c_a, 0) \end{aligned} \tag{5.9}$$

The average value b, is calculated using Roe average in Eq. (5.5)

5.2.3 HLLEW Scheme

Obayashi and Wada proposed a new, modified HLLE scheme that satisfies the positively conservative condition called HLLEW (*Harten-Lax-van Leer-Einfeldt-Wada*) scheme [65]. The numerical flux can be calculated using the following equations:

$$F_n = \frac{1}{2}\left[f(Q_{na}) \begin{pmatrix} 1 \\ u_{na} \\ u_{t1a} \\ u_{t2a} \\ H_a \end{pmatrix} + f(Q_{nb}) \begin{pmatrix} 1 \\ u_{nb} \\ u_{t1b} \\ u_{t2b} \\ H_b \end{pmatrix} + \begin{pmatrix} 0 \\ p_a + p_b + \delta_2 \\ \delta_3 \\ 0 \\ 0 \end{pmatrix} \right] \tag{5.10}$$

where

$$\begin{aligned} \delta_2 =& -(\lambda^+ \rho_{ave} \Delta u + \lambda^- \frac{\Delta p}{c_{ave}}) \\ \delta_3 =& -(\lambda_1 \Delta p + u_{ave} \delta_2) \\ \lambda^+ =& \frac{\lambda_2 + \lambda_3}{2} - \lambda_1 \\ \lambda^- =& \frac{\lambda_2 - \lambda_3}{2} \end{aligned} \tag{5.11}$$

Coefficients, δ_2 and δ_3 are obtained using the Roe average in Eq. (5.5)

5.2.4 AUSM$^+$-up Scheme

AUSM$^+$-up is an improved version of AUSM (Advection Upstream Splitting Method) scheme, discussed by Liou [66]. Inviscid flux function is explicitly split into two parts, mass flux and pressure flux, written as follows:

$$F_n = \dot{m}\psi + p_{ave} \tag{5.12}$$

and discretises them separately as follows:

$$\begin{aligned} \dot{m} &= \rho V \\ \psi &= (1, u, v, w, H)^T \\ p_{ave} &= (0, p_x, p_y, p_z, 0)^T \end{aligned} \tag{5.13}$$

where the first term in F_n is the convective flux, indicating the convection of ψ by the mass flux $\dot{m}$ and the second term is the pressure flux, p, containing nothing but the pressure. One advantage of this scheme is that the Jacobian matrix does not need to be calculated.

5.2.5 SLAU Scheme

Shima and Kitamura has introduced Simple Low-Dissipation Scheme of AUSM-Family (SLAU) [67]. The numerical flux of SLAU scheme is given by:

$$F_n = \frac{\dot{m} + |\dot{m}|}{2}\phi + \frac{\dot{m} - |\dot{m}|}{2}\phi + \tilde{p}N \tag{5.14}$$

$$\begin{aligned} \phi &= (1, u, v, w, h)^T \\ N &= (0, x_n, y_n, z_n, 0)^T \\ h &= (e + p)/\rho \end{aligned} \tag{5.15}$$

where x_n, y_n, z_n denote Cartesian components of a normal vector from the left to the right, u, v, w velocities in x, y, z directions, ρ, e, $\dot{m}$, and $\tilde{p}$, density, total energy per unit volume, mass flux and pressure flux, respectively.

5.3 Design and Implementation

As in UPACS implementation, Xilinx Virtex-6 FPGA (XC6VLX240T-1FF1156) was chosen as a target device, which supports partial reconfiguration. In a large software package, a subroutine itself is not always appropriate as a target of partial reconfiguration. For example, three schemes treated here include Roe average calculation and so it must be implemented as a static module. Before the design, the target is well re-structured so that the static module and partial reconfiguration modules are appropriately separated.

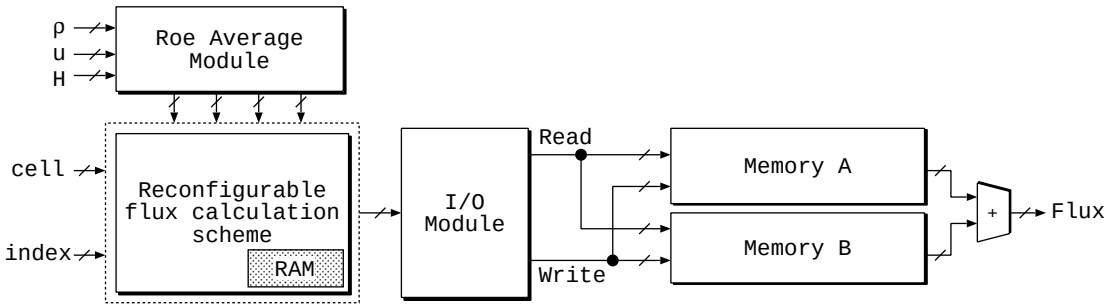

Figure 5.3: System overview.

Overview of the system is shown in Figure 5.3. At the beginning, the system will initialize and updates mesh size for each grid data and face index. Then, it will calculate Roe average value in Roe average module since this value is needed for three schemes: Roe, HLLE and HLLEW. AUSM$^+$-up and SLAU schemes are not required of Roe average values. Inputs for the Roe average module are density, ρ, velocity, u and total enthalpy per unit mass, H. The result of Roe average module is directly input to the flux calculation module.

After that, in the reconfigurable flux calculation module, users can choose which scheme they want to use. All schemes are defined as reconfigurable modules. Each scheme module has the same inputs and outputs as shown in Figure 5.4, and thus it can be specified in the HDL description as the functional modules with the reconfigurable partition attribute in the description of the top module. However, since AUSM$^+$-up and SLAU schemes do not require Roe average values, we created a virtual port for Roe average input indicated by dotted line in these two schemes. This is important because all reconfigurable modules must have identical input/output port while instantiation as a black box in top module. Multiple instances corresponding to the schemes are defined for such a single functional module. RAM is allocated in each scheme to store variable values during calculation. It is built with block RAMs, in which data is stored temporarily.

Table 5.1: Implementation environments.

Name	Tools
HDL	Verilog HDL
FPGA	Virtex-6 XC6VLX240T FF1156
Synthesis	ISE 12.4 XST
Simulation	ISim M.81d Simulator
PR Flow	PlanAhead 12.4
Programming	iMPACT M.81d
Floating-point Unit	CORE Generator

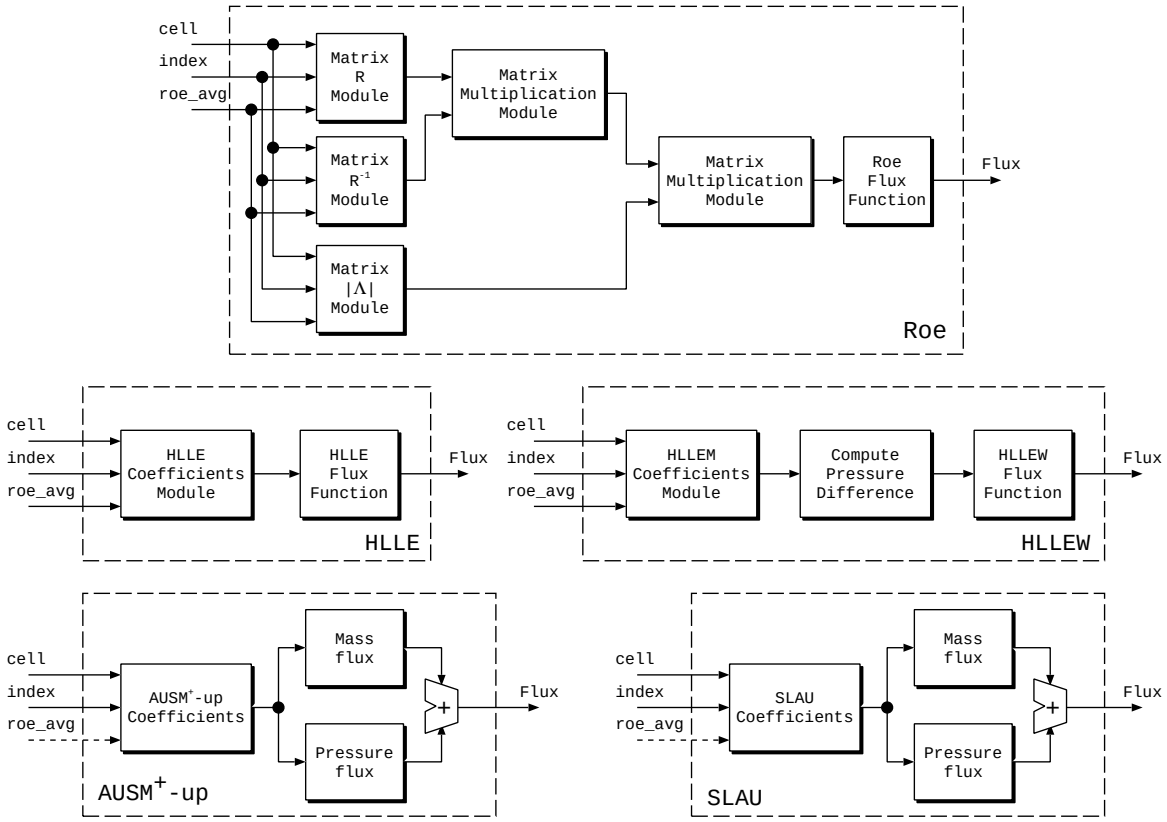

Figure 5.4: Implemented flux calculation schemes.

Programmable input/output (I/O) module is designed to control the access to memory. A result of flux calculation module is stored in memory. We implement a simple dual-port RAM for each adjacent cells A and B. Here, Block RAM used is 36 Kb block, RAMB36E, which is configured in a simple dual-port RAM mode. Read/Write data width is set to 64 bit. Read/Write process is performed in parallel with the flux calculation module. Summation of all cell flux values gives the total flux.

All modules were described using Verilog HDL and simulated with Xilinx ISim simulator. The modules were synthesized, and used resources were measured with Xilinx ISE 12.4. Floorplanning, constraint entry and Design Rule Checks (DRCs) are all accessed through the PlanAhead 12.4 software environment, which supports partial reconfiguration flow. All modules were implemented using IEEE 754 standard 64-bit double precision floating-point arithmetic[1]. Here, the floating-point computational module is based on the Xilinx Floating-Point Operator v5.0 incorporated into Xilinx ISE 12.4 software. The Floating-Point Operator v5.0 is an IP core for handling floating-point operations, and it is configurable by the user specifications. CORE Generator was used to provide the core for floating-point arithmetic units. So as to generate high performance computation unit, the level of DSP48E usage was set to the maximum to get the fastest output. In order to demonstrate

[1]Further specification of IEEE standard 754 floating-point numbers is given in Appendix A.

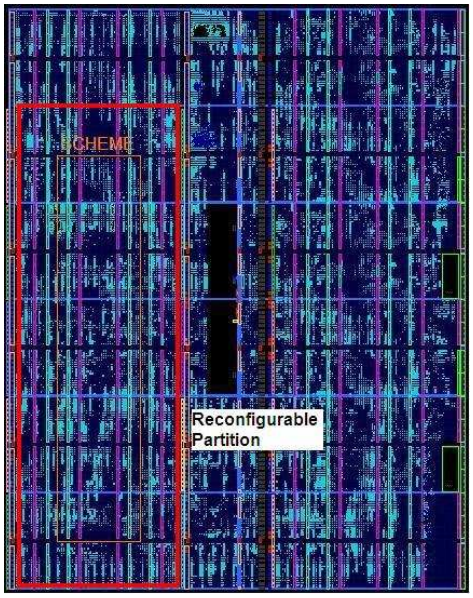

Figure 5.5: Floorplan of reconfigurable partition with HLLE scheme.

that our system works on a real FPGA, Xilinx ML605 board was used with 200 MHz operating frequency. Finally, for programming the FPGA, Xilinx iMPACT software was used. Summary for the implementation environments is shown in Table 5.1.

At one time, only one scheme is used and employed in the FPGA. The top, static and reconfigurable modules consist of many arithmetic functions. The parameters used for each computing unit are shown in Table 5.2. We used bottom-up synthesis technique to synthesize the design by modules. This synthesis technique requires that a separate netlist is written for reconfigurable partition, ensuring that each portion of the design is synthesized independently. Top and static module are synthesized with black box for the reconfigurable partition. In this case, flux calculation scheme module is defined as a black box in a top module synthesis. Roe, HLLE, HLLEW, AUSM$^+$-up and SLAU scheme modules are synthesized beforehand to provide the required netlist.

The next crucial step is to perform manual floorplanning for reconfigurable partition, which requires knowledge of the physical architecture of FPGA and understanding of how to floorplan for

Table 5.2: Data of used computing units.

Units	Latency	Registers	LUTs	DSP48E
Adder	14	947	797	3
Subtractor	14	947	798	3
Multiplier	16	483	362	11
Divider	57	5973	3261	0
Square Root	57	3283	1902	0
Comparator	1	0	128	0

optimal performance and area. Here, the challenge is how to create and pack large flux calculation schemes into a single partition. The partition boundary is defined so that the inserted proxy logic and the extra wiring cost may not degrade the total performance. Although irregular shaped partition such as T or L shapes is allowed, placement and routing in such regions sometimes degrade the performance because of the shortage of the routing resources and long wires. Therefore, we manually floorplan the flux schemes reconfigurable partition through PlanAhead 12.4 software environment. We chose a certain size rectangular shape for flux calculation scheme reconfigurable partition. Figure 5.5 shows a floorplan of the system while HLLE scheme is deployed. This is important for all reconfigurable modules to have enough resources to fit in the partition when the bitstream is loaded. Then, timing constraint entry and DRCs are performed.

5.3.1 Roe Average Module

As shown in Section 5.2, three schemes are needed of Roe average values before the flux computation is processed. Therefore, Roe average module is decided as a static module since it will be used in these three cases. In this module, managing parallelism is an important issue. The FaSTAR source code for Roe average calculation in Fortran 90 is written as follows:

```
RAT = SQRT(RRHT/RLFT)
RATI = 1.0/(RAT + 1)
RAV = RAT*RLFT
UAV = (RAT*URHT + ULFT) * RATI
VAV = (RAT*VRHT + VLFT) * RATI
WAV = (RAT*WRHT + WLFT) * RATI
HAV = (RAT*HRHT + HLFT) * RATI
QA2 = UAV*UAV + VAV*VAV + WAV*WAV
CA2 = GM1*(HAV - 0.5d0*QA2)
CAV = SQRT(CA2)
```

The code is executed sequentially from top to bottom. The advantage of sequential operations is that they efficiently use the resources, whereas parallelism can be used to reduce the time to completion and get the Roe average values at the expense of additional hardware resources.

The idea behind control parallelism is that the statements used to compute u_{ave} (UAV), v_{ave} (VAV), w_{ave} (WAV) and H_{ave} (HAV) can be performed simultaneously while still producing the correct answer. A scheduled data flow graph as shown in Figure 5.6 represents a data dependencies between operations. After square root, SQRT output to get the value of RAT, all multiplications operations are executed in parallel. At this stage, ρ_{ave} (RAV) are obtained. Then, after RATI value is obtained, all UAV, VAV, WAV and HAV can be also computed in parallel. However, CAV cannot begin until QA2 and

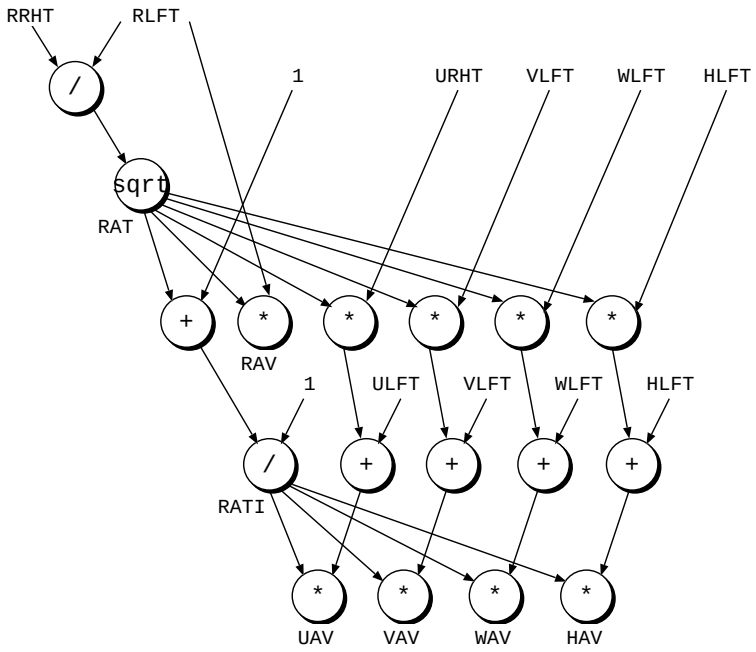

Figure 5.6: Scheduled data flow graph for Roe average module.

`CA2` finish. At the beginning to compute `RAT` will take a large number of clock cycles, since square root and divider computing units require a large number of clock cycles each. Therefore, we implemented pipeline datapath to address this as shown in Figure 5.7. Registers are inserted in dotted line to created a single stage pipeline.

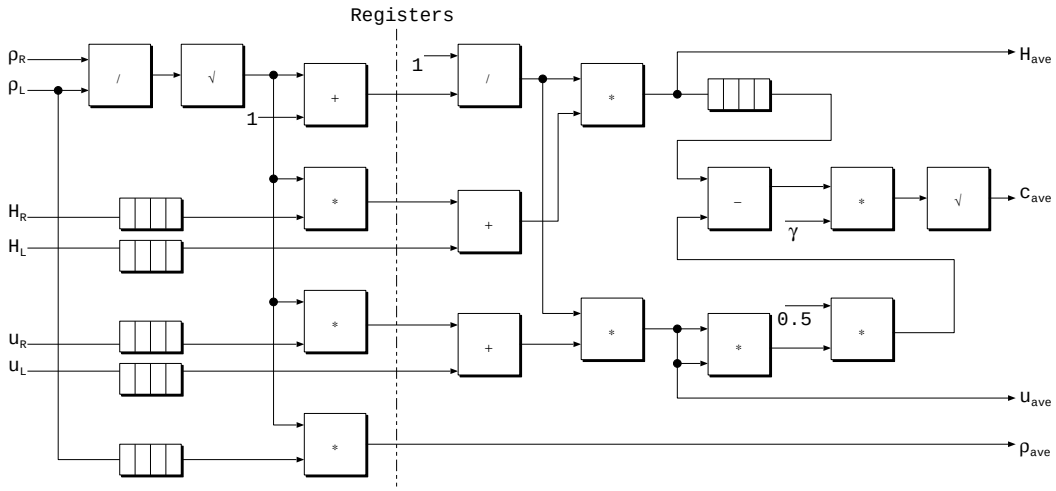

Figure 5.7: Pipeline datapath for Roe average module.

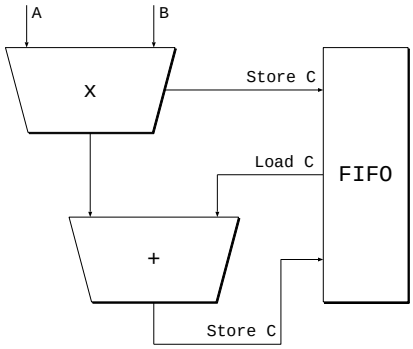

Figure 5.8: The structure of MAC organization for Roe scheme.

5.3.2 Roe Scheme Module

Roe scheme module was implemented in reconfigurable partition as a reconfigurable module. This module is designed and synthesized separately from the top module. Roe calculation scheme involves 5 steps to get the results:

1. Compute matrix R;
2. Compute matrix R^{-1};
3. Compute matrix $|\Lambda|$;
4. Compute Jacobian matrix $|A|$; and
5. Compute Roe's numerical flux.

The main arithmetic operation of this module is a 5×5 matrix multiplication as shown in Eq. (5.4). In general, the standard matrix multiplication $C = A \times B$ is defined as follows:

$$C_{i,j} = \sum_{k=0}^{N-1} A_{i,k} \times B_{k,j}, (0 \leq i \leq M, 0 \leq j \leq R) \tag{5.16}$$

where A, B and C are $M \times N$, $N \times R$, and $M \times R$ matrices, respectively. However, it requires two times matrix multiplication to obtain the Jacobian matrix, which utilizes a lot of resources.

However, computation of each matrices R, R^{-1} and $|\Lambda|$ are done in parallel. Then, matrix R is multiplied with matrix R^{-1}. Result of this matrix is multiplied with matrix $|\Lambda|$ to obtain Jacobian matrix. Finally, Jacobian matrix is used to compute the numerical flux.

We implemented a MAC (Multiplication and Accumulation) unit structure that couples the multiplication and the accumulation closely as shown in Figure 5.8. The multiplier receives the elements of A and B in a data driven manner. That means whenever both data are available, they will enter the pipeline. After the multiplication, the result is stored in FIFO and loaded address is generated. The

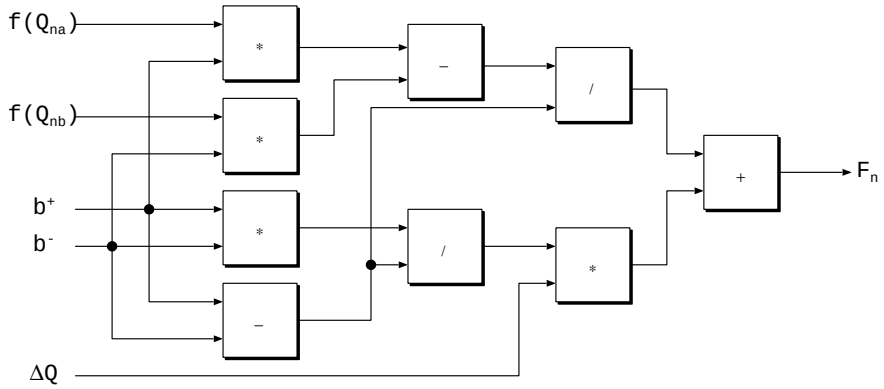

Figure 5.9: HLLE flux function circuit.

next multiplication result will be added with the prefetch data from the FIFO, and accumulated results are stored in temporary memory. This operation strategy is repeated continuously until calculation finishes.

5.3.3 HLLE Scheme Module

HLLE scheme module is rather straightforward compared to Roe scheme module. This module is also defined as a reconfigurable module in the same as Roe scheme module. Therefore, it is designed and synthesized separately from the top module to produce the required netlist. HLLE calculation scheme requires 2 steps to get the final result:

1. Compute HLLE coefficients and eigenvalues; and
2. Compute HLLE numerical flux.

After received the Roe average values, HLLE coefficients, b^+ and b^- are computed using adder, subtractor and comparator computing units. Then, HLLE flux function circuit shown in Figure 5.9 is used to compute the HLLE numerical flux. This hardware implementation divides the fraction of Eq. (5.8) into two parts. Therefore, calculation of left fraction and right fraction are done in parallel. Summation of these two values makes the total flux.

5.3.4 HLLEW Scheme Module

Apparently, HLLEW scheme is a modification of HLLE scheme. This scheme is also based on Roe scheme as well as HLLE scheme. However, to compute the HLLEW flux, there is no need to do a matrix computation as suggested for Roe scheme. HLLEW scheme is also implemented as a reconfigurable module in reconfigurable partition. Therefore, it is synthesized beforehand and separately to provide the required netlist to the top module. Computation of HLLEW flux requires the following steps:

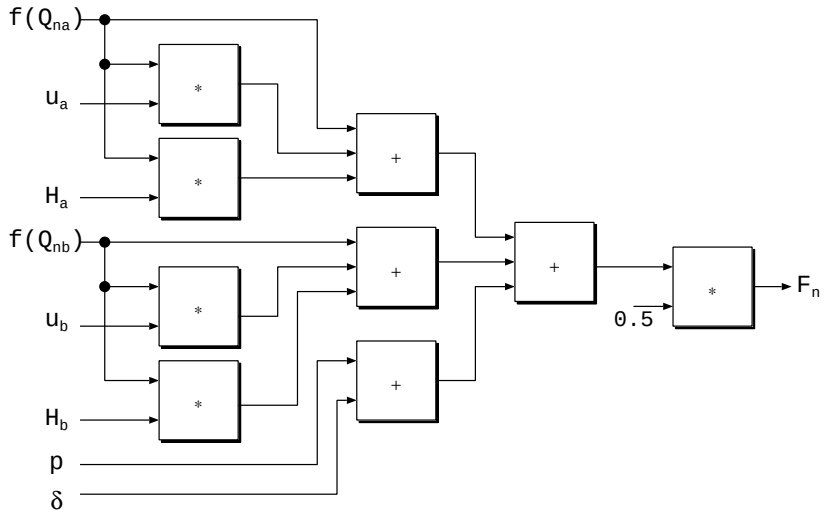

Figure 5.10: HLLEW flux function circuit.

1. Compute coefficients and eigenvalues;
2. Compute pressure difference; and
3. Compute HLLEW numerical flux.

Again, Roe average values are used here to compute the coefficients and eigenvalues. After that, pressure difference, Δp is calculated. When all values are obtained, inputs are sent to the HLLEW flux function circuit as shown in Figure 5.10 to compute the numerical flux. This circuit is explicitly designed to calculate the Q_{na} and Q_{nb} data values in parallel. Total flux is obtained after summation of result divided by two.

5.3.5 AUSM$^+$-up Scheme Module

In this module, AUSM-based flux calculation is divided to two, mass flux and pressure flux. Sum of these fluxes will give the final numerical flux. Steps involved in this module are as follows:

1. Compute AUSM$^+$-up coefficients;
2. Compute mass flux and pressure flux in parallel; and
3. Summation of mass and pressure flux.

After coefficients values are obtained, the data are sent to mass flux circuit and pressure flux circuit. Mass flux circuit is straightforward as density, ρ times velocity, V for all dimensions. Pressure flux circuit is shown in Figure 5.11.

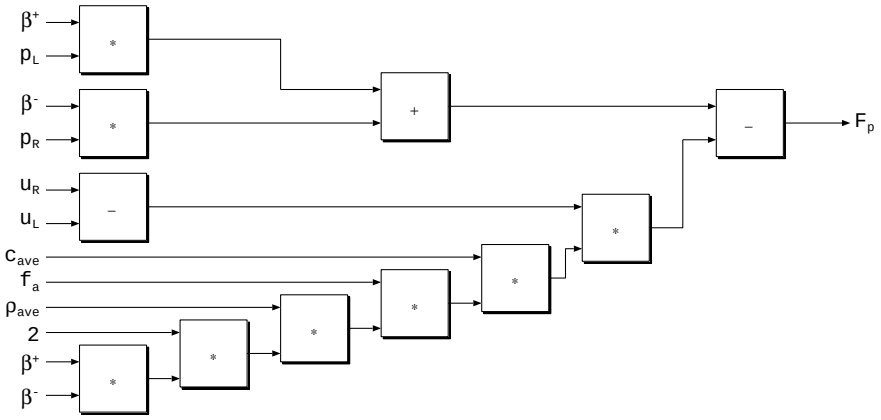

Figure 5.11: AUSM$^+$-up pressure flux circuit.

5.3.6 SLAU Scheme Module

Same as AUSM$^+$-up scheme, SLAU scheme is a AUSM-based family. Therefore, calculation of numerical flux is also split to two. Three steps of computation held in this module as follows:

1. Compute SLAU coefficients;
2. Compute mass flux and pressure flux in parallel; and
3. Summation of mass and pressure flux.

Again, implementation in FPGA allows users to compute mass flux and pressure flux in parallel. Pressure flux circuit for SLAU scheme is shown in Figure 5.12.

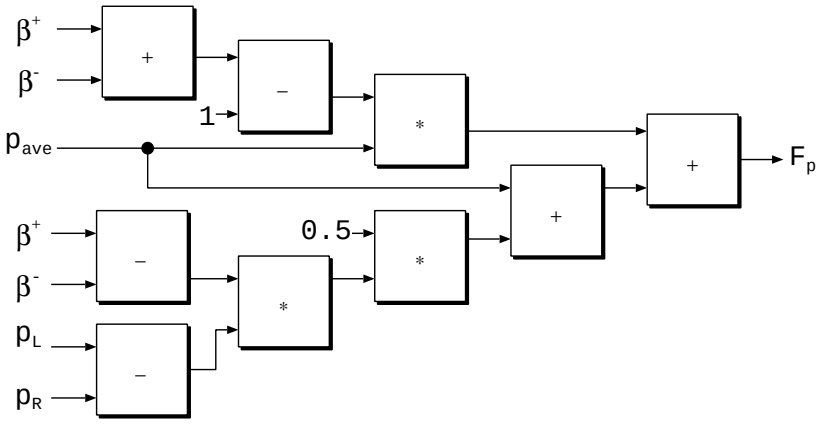

Figure 5.12: SLAU pressure flux circuit.

5.3.7 Implementation Issues

Crucial challenge in this design is to implement large-scale scientific computation using partial reconfiguration. Careful design requirements and considerations should be carried out. At the same time, the design specification must be analyzed thoroughly, and the limitations associated with partial reconfigurable designs are considered. The challenges and solutions are listed as follows:

1. *I/O in each scheme module*
   Flux calculation schemes must include the I/O circuitry, Input Buffer (IBUF) and Output Buffer (OBUF) that are required to connect internal logic to package pins. In other words, the I/O features must be completely contained within the scheme module, but the port list for the complete design remains at the top-level design description. Besides, the limitation of the I/O pins of FPGA must be considered, since flux calculation module requires many I/O.

2. *DSP blocks in each scheme module*
   It is also important that the physical region selected has adequate resources especially DSP48E for all schemes. Flux calculation scheme requires a lot of DSP blocks to perform the computation. Therefore, we properly set the last blocks occupied in both end columns of reconfigurable partition are DSP blocks, instead of slice or block RAM. Using this strategy, we can maximize DSP blocks in the partition.

3. *Interaction with CORE Generator*
   Since we used CORE Generator to generate all computing units, netlist-based cores were created to be instantiated in the design. To make sure these cores can be instantiated easily, the boundaries of flux calculation scheme partition are not modified. We also made considerations for the definition of the flux calculation scheme region to ensure the proper elements are contained within.

4. *Optimization*
   In order to optimize the design time for bit file generation, the most complicated and highly resource consuming design should be selected first. This is because full bit file is generated only once in the first configuration. In subsequent configuration, full bit file is just promoted from the first configuration.

5.4 Evaluation

In order to demonstrate the effectiveness of our design, we used a sample data of NACA 0012 airfoil. *The National Advisory Committee for Aeronautics* (NACA) develops the NACA airfoils shapes for aircraft wings. The grid dataset consisting of 11,564 grids with 22,883 faces was used in our study. We evaluated the used resources, configuration speed for full and partial reconfiguration designs, and system performance.

5.4.1 Resource Utilization

The amount of required slice registers, slice LUTs, DSP48E and BlockRAM was evaluated when the design is synthesized. The design is synthesized module by module. Consumed resources for each module is shown in Table 5.3. Result of the resource utilization is shown in Figure 5.13.

There are 3 design options for implementation consideration. Obviously, the first option is to fit in all modules in a single FPGA, shown in first column noted by "Full". It means all modules shown in Table 5.3 are implemented in one design. The main advantage of this method is that it only requires one time configuration and no reconfiguration is needed. However, it requires a large amount of hardware that is not enough in a single FPGA. Total resource utilization for registers, LUTs and DSP48E usage all are exceeds 100%.

Second strategy is implementing only a scheme in one design. It implies that for five schemes, there is five difference designs. This is shown in second, third, forth, fifth and sixth column denoted by "Roe", "HLLEW", "HLLE", "AUSM$^+$-up" and "SLAU". In this strategy, AUSM$^+$-up and SLAU are do not require Roe average module. Therefore, their resources usages are less than the other three schemes. Although there are enough resources to do this, two disadvantages arise. First, it will require two times full reconfiguration if a user wants to change from one scheme to another. Second, resource is overused since the same I/O and Roe average module are used again except the flux scheme module.

The third option, which is our proposed method, is to utilize partial reconfiguration. This is shown in most right column noted by "Partial". Top, static and reconfigurable modules are fixed in a single FPGA. Flux calculation scheme bitstreams are stored in the host PC. When users want to use any particular scheme, it is loaded to an FPGA. Resource utilization when no scheme loaded is small. In addition, consumed resources when system is in use and one scheme loaded is the same as the second option. However, another advantage of this technique is the configuration time. If users want to change from one scheme to another, it can be faster compared to the full reconfiguration.

Table 5.3: Available and consumed resources.

Module	Registers(%)	LUTs(%)	DSP48E(%)	BRAM(%)
I/O Module	0.3	1	0.4	0
Roe average	10	14	11	2
Roe	57	62	52	17
AUSM$^+$-up	33	45	19	11
SLAU	31	43	16	10
HLLEW	30	43	16	10
HLLE	27	37	11	4

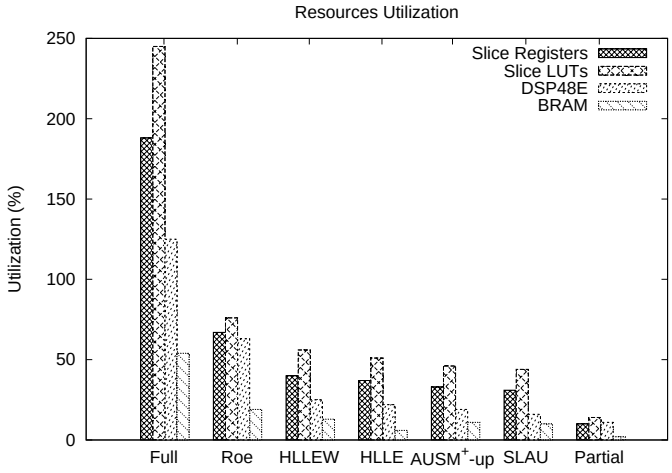

Figure 5.13: Resources utilization for all possible implementation.

We examined the amount of resources utilization for each design option. We found out that all modules cannot be implemented in single Virtex-6 XC6VLX240T-1FF1156 FPGA since resources available are not enough to accommodate all modules. For the second option design, all schemes are implemented separately. Although there are enough resources to implement this, all designs require full reconfiguration to load in an FPGA independently. In partially reconfigurable design, bitstreams of all schemes are stored in host PC. Therefore, maximum resources reduction is measured when Roe scheme is deployed since it requires the highest resources. On average, consumed resources for "Full" design is 153% while for partially reconfigurable design when Roe scheme is deployed is 57%. By normalizing "Full" design to 100%, we divided 57/1.53. As a result, resource utilization is successfully reduced by 62.75% on average.

Even though the resources are not enough to implement all modules in a single FPGA, there are resources overhead in partially reconfigurable design. This is because all schemes modules are implemented in reconfigurable partition. The partition is manually floorplaned and resources allocated are fixed to fit in all modules. Since Roe scheme occupied higher resources than any other schemes, the reconfigurable partition wasted resources when other schemes are loaded. However, unused resources in these schemes are used again when Roe scheme is selected.

5.4.2 Configuration Time

The configuration time for the full reconfiguration and partial reconfiguration was compared. The speed of configuration is directly related to the size of the partial bit file and the bandwidth of the configuration port. Since we use JTAG configuration port, for Virtex-6 device [69], configuration time is given by:

$$configuration\ time = \frac{(2,044 + bits\ in\ bitstream)}{TCK\ frequency} \tag{5.17}$$

where *bits in bitstream* is the size of the configuration bitstream in bits and *TCK frequency* is maximum configuration TCK (*Test Clock*) frequency and used for boundary-scan operations. 2,044 is the total number of clocks needed for pre-processing and post-processing for single device configuration sequence while programming the bitstream to FPGA. Although the maximum bandwidth available is 66 Mbps, we found out that while configuring the FPGA using iMPACT, used bandwidth is 16.7 Mbps and data width is 1 bit.

In a full reconfiguration, the total bitstream size is 9,017 KB. Based on the Eq. (5.17), the configuration time is equal to 4.423 sec. On the other hand, bitstream size for all schemes are 1,422 KB. This means configuration time for partial reconfiguration is 0.704 sec. In short, the partial reconfiguration method accelerated the configuration speed by 6.28 times.

There is no overhead for partial reconfiguration since the users must decide which scheme they want to use before the calculation starts. Therefore, the configuration time will not affect the computation time in FPGA. In addition, configuration time will not cause a bottleneck to the system when the grid size grows large and takes a lot of iterations. This is because all flux scheme bitstream is fixed and not affected by large input size.

5.4.3 Performance

Total clock cycles of flux computational module were measured. The total clock cycles for each Roe, AUSM$^+$-up, SLAU, HLLEW and HLLE scheme were 205600×10^3, 202200×10^3, 200800×10^3, 197400×10^3 and 191200×10^3, respectively. The execution time for flux calculation scheme in software was compared with the execution time by hardware as shown in Figure 5.14.

In software, all schemes were executed by Core 2 Duo 2.4GHz with Linux Kernel 2.6.18 operating system. All schemes are compiled by using Intel Fortran Compiler 10.1. Execution time is measured by using `call system_clock` prepared in Fortran 90 language. We found out the execution time took 5.400 sec. for Roe scheme, 4.723 sec. for AUSM$^+$-up scheme, 4.616 sec. for SLAU scheme, 4.533 sec. for HLLEW scheme and 4.399 sec. for HLLE scheme. This is shown in the first column of Figure 5.14, denoted by "Software".

In hardware, since we know the total clock cycles required from the beginning to the end, and operating frequency for the FPGA is 200 MHz. Therefore, computation time by FPGA for Roe scheme was 1.028 sec., 1.011 sec. for AUSM$^+$-up scheme, 1.004 sec. for SLAU scheme, 0.987 sec. for HLLEW scheme and 0.956 sec. for HLLE scheme. Adding the configuration time and computation time gives an execution time in hardware. Therefore, the second column of Figure 5.14 shows an execution time in FPGA if second option design of full reconfiguration (FR) is deployed, noted by "FPGA FR". Adding 4.423 sec configuration time to each scheme computation time of Roe, AUSM$^+$-up, SLAU, HLLEW and HLLE scheme gives an execution time become 5.451 sec., 5.434 sec., 5.427 sec., 5.410 sec. and 5.379 sec., respectively.

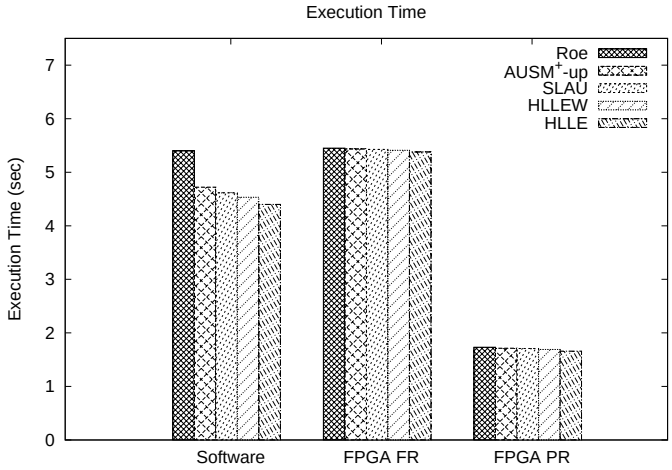

Figure 5.14: Execution time in software and FPGA.

Third column of Figure 5.14 shows an execution time in FPGA if Partial Reconfiguration (PR) is used, denoted by "FPGA PR". Adding 0.704 sec. configuration time to each scheme computation time gives an execution time of Roe, AUSM$^+$-up, SLAU, HLLEW and HLLE scheme are 1.732 sec., 1.715 sec., 1.708 sec., 1.691 sec. and 1.660 sec., respectively. Accelerated speed between hardware and software is compared when HLLE scheme in partially reconfigurable design is deployed since it is executed fastest in software. Therefore, the execution time of FPGA was 2.65 times faster compared to the software execution.

Full reconfiguration design strategy for each scheme gives almost the same performance produced by software. In fact, software execution timings for AUSM$^+$-up, SLAU, HLLE and HLLEW schemes are faster than FPGA. However, partial reconfigurable design approach at least gives a 2.65 fold speed-up compared to software execution. Therefore, taking configuration time into account, performance improvement using partial reconfiguration method is justified.

In real simulation, the computational model for acceleration consists of a host processing node, usually a high-end multicore processor supplemented with our accelerator. The advection term computation is assigned to the accelerator, while the remaining part of FaSTAR stays on the host. The host application manages the interaction with the FPGA and controls data flow between them through external interfaces such as PCI Express. For applications of interest, the advection term computation is a small fraction of the FaSTAR code and represents well over 30% of the computational time.

The kind of optimization used for FPGA depends on the resource that limits the computational speed. In a well-designed FPGA accelerator card, the on-board memory is designed to provide multiple channels of streamed data into the FPGA pins to fully support its available bandwidth. It can also buffer any intermediate results, so that the host memory provides only initial source data and stores the final results. Figure 5.15 shows a sample computational result of NACA 0012 airfoil pressure flow using FaSTAR packages.

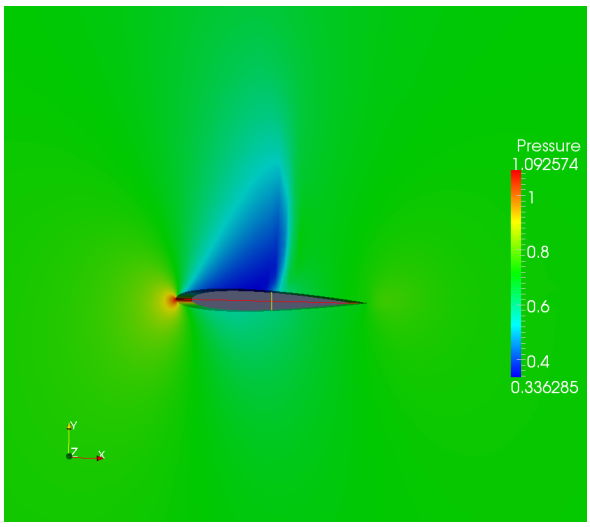

Figure 5.15: Visualizing sample computational result of NACA 0012 airfoil pressure flow.

5.5 Summary

In this chapter, we presented FaSTAR, a convenient CFD software package that allows users to select various set of solutions. However, it is hard to implement even on high-capacity FPGAs because of its large module. The efficient use of partial reconfigurability in recent FPGAs was explored. Advection term computation was chosen as a target subroutine, and flux calculation scheme was deployed as a reconfigurable module. Five flux calculation schemes were implemented: Roe, HLLE, HLLEW, AUSM$^+$-up and SLAU schemes.

The implementation using partial reconfiguration platform has successfully reduced required hardware resources, improved configuration time and its performance. Resources utilization was saved up to 60% on average. The proposed design also improved the configuration time by 6.28 times faster and accelerated the system at least 2.65 times in performance.

Chapter 6

Conclusions

6.1 Summary

In order to expand a design space for CFD applications on reconfigurable hardware, we proposed a reconfigurable fluid dynamics computation using partial reconfiguration on an FPGA. CFD applications receive benefit from this technique, since not all available solutions are used in one particular CFD simulation. Instead of configuring all solvers in an FPGA, only required stuff should be implemented to reduce the resource usage and power consumption by using the partial reconfiguration.

In this thesis, we first introduced the CFD and FPGA. The CFD procedure and their simulation process were briefly explained and discussed. Conventional systems where CFD applications are executed were also presented. Two CFD software packages, UPACS and FaSTAR were introduced and their features were discussed. Then, we discussed about FPGA and their advantages compared to ASIC and CPU. System applications using FPGA for fluid dynamics were also presented. Some performance improvement was achieved and reported by related researchers.

In Chapter 4, we implemented MUSCL scheme in UPACS with reconfigurable flux limiter functions. UPACS is a convenient CFD package that allows users to select various sets of solutions. UPACS solver together with support utilities has proven its effectiveness in simulating flows around complex configurations using multi-block structured grid scheme. However, it is hard to be implemented even on large FPGAs because of the required hardware amount. We proposed reconfigurable flux limiter functions, which are used in TMUSCL and CMUSCL. Six limiter functions were implemented: no limiter, van Leer, van Albada, minmod, superbee and Hemker-Koren.

In Chapter 5, we implemented reconfigurable flux calculation scheme in advection term computation in FaSTAR. FaSTAR is another convenient CFD software package with various solvers and automatic generation of grid data. However, implementation even on large FPGAs is difficult because of its large modules and complicated structure. Therefore, exploitation of partial reconfigurability in recent FPGAs was considered. Advection term computation was chosen as a target and flux calculation scheme was deployed as a reconfigurable module. Five flux calculation schemes were implemented: Roe, HLLE, HLLEW, AUSM$^+$-up, and SLAU schemes.

Table 6.1: Max. bandwidth for configuration ports in Virtex 6 architecture.

Configuration Mode	Max Clock Rate	Data Width	Max Bandwidth
ICAP	100 MHz	32 bit	3.2 Gbps
SelectMAP	100 MHz	32 bit	3.2 Gbps
Serial Mode	100 MHz	1 bit	100 Mbps
JTAG	66 MHz	1 bit	66 Mbps

Overall, both implementations have successfully reduced the resource utilization by 44% to 63%. Total power consumption for UPACS exploration was also reduced by 29% to 33%. Configuration speed was improved by 6 to 34 times faster compared to full reconfiguration. Approximately 2 to 17 times speed up was achieved compared to execution on Core 2 Duo at 2.4 GHz.

We make a discussion on important issues for both implementations in the next section. Finally, future directions are indicated in Section 6.3.

6.2 Discussion

In Chapter 4, we proposed a reconfigurable flux limiter functions in MUSCL scheme for TMUSCL and CMUSCL. The proposed reconfigurable module strategy successfully reduced the total resources usage. In Chapter 5, we have proposed flux calculation scheme in advection term computation. However, to take full advantage of the partial reconfiguration capability, we must analyze the design specification thoroughly, and consider the limitations associated with partial reconfiguration designs.

An appropriate hierarchical design would resolve many complexities and difficulties when implementing a partially reconfigurable design. A clear design instance hierarchy simplifies physical constraints. Grouping logic that is packed together in the same hierarchical level is necessary. They are well known design practice in FPGA designs but are hardly followed by designers. Following the design rules is not strictly required in partially reconfigurable design, but the potential negative effects of not following them are pronounced.

Although satisfactory performance was achieved for both designs, configuration time to upload the partial bit file remains as a key issue. In our observation, the speed of configuration is directly related to the size of partial bit file and the bandwidth of the configuration port. The different configuration ports in Virtex 6 architectures have the maximum bandwidth as shown in Table 6.1. Any of the following configuration ports can be used to load the partial bitstream: ICAP (Internal Configuration Access Port), SelectMAP, Serial Mode, or JTAG (Joint Test Action Group).

We used JTAG as a configuration mode because it is advantageous for quick testing and debugging. It also can be easily driven by iMPACT tools using configuration cable that supports JTAG. However, as shown in Table 6.1, JTAG has the lowest maximum clock rate and lowest maximum bandwidth with only 1 bit data width. Although stated JTAG maximum clock rate is 66 MHz, we

found that clock rate available practically was 16.7 MHz. It further degrades the configuration speed. If ICAP port can be utilized, faster configuration speed could be achieved. This port is a good choice for user configuration solutions. However, it requires the instantiation of an ICAP controller as well as logic to drive the ICAP interface. Users need to design internal partial reconfiguration controller to load partial bitstreams through the ICAP interface. Internal configuration can consist of either a custom state machine, or an embedded processor such as MicroBlaze soft processor.

Moreover, not all logic is allowed to be actively reconfigured. Although most types of logical components may be reconfigured, global logic and clocking resources must be placed in the static region to remain operational during reconfiguration. Logic that must remain in static logic includes:

- clock modifying blocks (PLL, PMCD, DCM),
- clock buffers (BUFG), and
- device features blocks (ICAP, USR_ACCESS).

Since clock buffer region exists across the entire FPGA, it is hard to design reconfigurable partition with more than 50% of available resources.

Other issue to consider for CFD implementation is the capacity of target FPGA. In this thesis, we have targeted Virtex 6 device, which was the high-end FPGAs when the study was started. Recently a high capacity Virtex 7 FPGA has been introduced. Therefore, if our target can be replaced with such newly developed FPGA, we can implement more modules and expand our design.

6.3 Future Directions

Computing systems using FPGAs will expand the future to achieve high performance for many applications at lower operating cost. The price of an FPGA chip will degrade by volume efficiency. This is because FPGAs are now embedded in large scale computing systems in some research institute, such as BEE2 [70] and commercial products like high-vision recorders and video capture cards. This means that the advantage of introducing FPGA devices is becoming much higher than producing ASIC. Moreover, the operating cost is much lower than personal computers and other hardware like Cell/BE (Broadband Engine) and GPU.

This study shows that middle-range FPGA can achieve several fold performance compared to a recent microprocessor. Although it is difficult for FPGAs to outperform the personal computers, performance improvement by parallel processing on a chip can be expand linearly according to the capacity of the FPGA. Moreover, recent FPGAs are being developed to operate with operating frequency higher than 500 MHz. In contrast, recent microprocessors will not be able to improve performance simply by increasing their number of cores, since each core shares a cache with other cores.

In fluid dynamics application, FPGA growth rates are exceeding commodity CPUs and future implementations will ultimately provide some interesting platforms for exploration. The use of high level language to target this platform will be necessary for making them usable in the scientific community. However, a single FPGA is not sufficient for large CFD software. Integration with a host processor and many FPGAs are very useful to accommodate the whole CFD code. Therefore, further study on effective communication medium such as ethernet is unavoidable. The ability to identify problems of the implementation issues might lead to realization of full CFD code execution using FPGAs.

While the limitations of a single FPGA are noticed, multi-FPGA platform with multiple reconfigurations can be a target of another area of research. They offer the potential to mega-boost the capacity of resource in FPGA as well as more modules can be reconfigured. However, the interconnection between an FPGA and how they are communicating are another issue to solve for successful implementation.

Another prospective area for CFD implementation is to deploy heterogeneous architecture between FPGA and CPU. Successful example of this kind of architecture for high performance computing is reported by Lindtjorn *et al.* [71]. Heterogeneous computing systems refer to electronic systems that use a variety of different types of computational units. A computational unit could be a General-Purpose Processor (GPP) and custom acceleration logic of an FPGA. In general, a heterogeneous computing platform consists of processors with different Instruction Set Architectures (ISAs). Again, the same issue with multi-FPGA is to be solved interconnection between those architectures.

The CFD on cluster computing using FPGA is another possible area for future research. This approach is very hopeful because the most computationally intensive part in the CFD must be performed iteratively, and that can be parallelized by using many FPGAs.

Bibliography

[1] A. Chandramowlishwaran, K. Knobe, and R. Vuduc. Performance evaluation of concurrent collections on high-performance multicore computing systems. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, 2010.

[2] Xingjun Zhang, Yanfei Ding, Yiyuan Huang, and Xiaoshe Dong. Design and implementation of a heterogeneous high-performance computing framework using dynamic and partial reconfigurable FPGAs. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 2329–2334, 2010.

[3] Y. Ogata, T. Endo, N. Maruyama, and S. Matsuoka. An efficient, model-based CPU-GPU heterogeneous FFT library. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–10, 2008.

[4] D. Gohringer and J. Becker. FPGA-based runtime adaptive multiprocessor approach for embedded high performance computing applications. In *VLSI (ISVLSI), 2010 IEEE Computer Society Annual Symposium on*, pages 477–478, 2010.

[5] M. de Kruijf and K. Sankaralingam. MapReduce for the cell broadband engine architecture. *IBM Journal of Research and Development*, 53(5):10:1–10:12, 2009.

[6] L. Pate, J. Duboue, and Ph. Picot. CFD-a tool to design jet engine internal cooling system. *34th AIAA/ASME/SAE/ASEE Joint Propulsion Conference and Exhibit*, 1998.

[7] B.H. Blessing, J. Pham, and D.D. Marshall. Using CFD as a design tool on new innovative airliner configurations. *47th AIAA Aerospace Sciences Meeting Including The New Horizons Forum and Aerospace Exposition*, 2009.

[8] K.J. Page and P.M. Chau. A FPGA ASIC communication channel systems emulator. In *ASIC Conference and Exhibit, 1993. Proceedings., Sixth Annual IEEE International*, pages 345–348, 1993.

[9] C.R.W. Reinbrecht, J.L. Da Silva, and E.E. Fabris. Applying in education an FPGA-based methodology to prototype ASIC soft cores and test ICs. In *Programmable Logic (SPL), 2012 VIII Southern Conference on*, pages 1–5, 2012.

[10] B. Sukhwani and M.C. Herbordt. FPGA acceleration of rigid-molecule docking codes. *Computers Digital Techniques, IET*, 4(3):184–195, 2010.

[11] J.D. Bakos. FPGA acceleration of gene rearrangement analysis. In *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, pages 85–94, 2007.

[12] N. Alachiotis and A. Stamatakis. FPGA acceleration of the phylogenetic parsimony kernel? In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pages 417 –422, Sept. 2011.

[13] K. Nakano and E. Takamichi. An image retrieval system using FPGAs. In *Design Automation Conference, 2003. Proceedings of the ASP-DAC 2003. Asia and South Pacific*, pages 370 – 373, Jan. 2003.

[14] A. Kaganov, P. Chow, and A. Lakhany. FPGA acceleration of monte-carlo based credit derivative pricing. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 329 –334, Sept. 2008.

[15] William D. Smith and Austars R. Schnore. Towards an RCC-based accelerator for computational fluid dynamics applications. *J. Supercomput.*, 30:239–261, December 2004.

[16] E. Andres, M. Molina, G. Botella, A. del Barrio, and J. Mendias. Aerodynamics analysis acceleration through reconfigurable hardware. In *Programmable Logic, 2008 4th Southern Conference on*, pages 105 –110, March 2008.

[17] Xilinx Inc. *Partial Reconfiguration User Guide 13.1 UG702*, March 2011.

[18] Y. Hori, A. Satoh, H. Sakane, and K. Toda. Bitstream encryption and authentication with AES-GCM in dynamically reconfigurable systems. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 23 –28, Sept. 2008.

[19] Christopher Claus, Johannes Zeppenfeld, Florian Müller, and Walter Stechele. Using partial-run-time reconfigurable hardware to accelerate video processing in driver assistance system. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '07, pages 498–503, San Jose, CA, USA, 2007. EDA Consortium.

[20] B.J. LaMeres and C. Gauer. Dynamic reconfigurable computing architecture for aerospace applications. In *Aerospace conference, 2009 IEEE*, pages 1 –6, March 2009.

[21] B. Osterloh, H. Michalik, S.A. Habinc, and B. Fiethe. Dynamic partial reconfiguration in space applications. In *Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on*, pages 336 –343, 29 2009-Aug. 1 2009.

[22] Y. Matsuo, M. Tsuchiya, M. Aoki, N. Sueyasu, T. Inari, and K. Yazawa. Early experience with aerospace CFD at JAXA on the fujitsu PRIMEPOWER HPC2500. In *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*, page 11, Nov. 2004.

[23] Hirokazu Morisita, Kenta Inakagata, Yasunori Osana, Naoyuki Fujita, and Hideharu Amano. Implementation and evaluation of an arithmetic pipeline on FLOPS-2D: multi-FPGA system. *SIGARCH Comput. Archit. News*, 38:8–13, January 2011.

[24] K. Inakagata, H. Morishita, Y. Osana, N. Fujita, and H. Amano. Modularizing flux limiter functions for a computational fluid dynamics accelerator on FPGAs. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 654 –657, Sept 2009.

[25] Takayuki Akamine, Kenta Inakagata, Yasunori Osana, Naoyuki Fujita, and Hideharu Amano. An implementation of out-of-order execution system for acceleration of computational fluid dynamics on FPGAs. *SIGARCH Comput. Archit. News*, 39(4):50–55, December 2011.

[26] Mohamad Sofian Abu Talip, Takayuki Akamine, Yasunori Osana, Naoyuki Fujita, and Hideharu Amano. Cost effective implementation of flux limiter functions using partial reconfiguration. In *Proceedings of the 8th international conference on Reconfigurable Computing: architectures, tools and applications*, ARC'12, pages 215–226, Berlin, Heidelberg, 2012. Springer-Verlag.

[27] Mohamad Sofian Abu Talip, Takayuki Akamine, Yasunori Osana, Naoyuki Fujita, and Hideharu Amano. Dynamically reconfigurable flux limiter functions in MUSCL scheme. In *Reconfigurable*

Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on, pages 1 – 7, July 2012.

[28] Hiroyuki Yamazaki, Shunji Enomoto, and Kazuomi Yamamoto. A common CFD platform UPACS. In *Proceedings of the Third International Symposium on High Performance Computing*, ISHPC '00, pages 182–190, London, UK, 2000. Springer-Verlag.

[29] Ryoji Takaki, Kazuomi Yamamoto, Takashi Yamane, Shunji Enomoto, and Junichi Mukai. The development of the UPACS CFD environment. In *High Performance Computing*, volume 2858 of *Lecture Notes in Computer Science*, pages 307–319. Springer Berlin Heidelberg, 2003.

[30] Atsushi Hashimoto, Keiichi Murakami, Takeshi Aoyama, Manabu Hishida, Lahur Paulus R., Masahide Sakashita, and Yukio Sato. Development of fast flow solver FaSTAR. In *Proc. 42nd Fluid Dynamics Conference/Aerospace Numerical Simulation Symposium*, 2010.

[31] M. Yokokawa. The K computer and its application. In *Networking and Computing (ICNC), 2012 Third International Conference on*, pages 21–22, 2012.

[32] T. Maruyama, T. Yoshida, R. Kan, I. Yamazaki, S. Yamamura, N. Takahashi, M. Hondou, and H. Okano. SPARC64 VIIIfx: A new-generation octocore processor for petascale computing. *Micro, IEEE*, 30(2):30–40, 2010.

[33] Y. Ajima, S. Sumimoto, and T. Shimizu. Tofu: A 6D mesh/torus interconnect for exascale computers. *Computer*, 42(11):36–40, 2009.

[34] T. Toyoshima. ICC: An interconnect controller for the tofu interconnect architecture. *Hot Chips 22*, 2010.

[35] S. Saini, P. Mehrotra, K. Taylor, M. Aftosmis, and R. Biswas. Performance analysis of CFD application Cart3D using MPInside and performance monitor unit data on Nehalem and Westmere based supercomputers. In *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, pages 331–338, 2011.

[36] Sun cluster architecture: a white paper. In *Cluster Computing, 1999. Proceedings. 1st IEEE Computer Society International Workshop on*, pages 331–338, 1999.

[37] Li Xiao, Xiaodong Zhang, Zhengqian Kuang, Baiming Feng, and Jichang Kang. Auto-CFD: efficiently parallelizing CFD applications on clusters. In *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on*, pages 46–53, 2003.

[38] Tetsu Narumi, Yousuke Ohno, Noriaki Okimoto, Takahiro Koishi, Atsushi Suenaga, Noriyuki Futatsugi, Ryoko Yanai, Ryutaro Himeno, Shigenori Fujikawa, Makoto Taiji, and Mitsuru Ikei. A 55 TFLOPS simulation of amyloid-forming peptides from yeast prion Sup35 with the special-purpose computer system MDGRAPE-3. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[39] Michael Creel and Mohammad Zubair. High performance implementation of an econometrics and financial application on GPUs. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1147–1153, 2012.

[40] J. Michalakes and M. Vachharajani. GPU acceleration of numerical weather prediction. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–7, 2008.

[41] P. Bailey, J. Myre, S.D.C. Walsh, D.J. Lilja, and M.O. Saar. Accelerating Lattice Boltzmann fluid flow simulations using graphics processors. In *Parallel Processing, 2009. ICPP '09. International Conference on*, pages 550–557, 2009.

[42] Chih-Wei Hsieh, Sheng-Hsiu Kuo, Fang-An Kuo, and Chau-Yi Chou. Solving parabolic problems using multithread and GPU. In *Parallel and Distributed Processing with Applications (ISPA), 2010 International Symposium on*, pages 75–80, 2010.

[43] Youquan Liu, Kai Shi, Heng Deng, and Enhua Wu. A multi-GPU based semi-Lagrangian fluid solver. In *Proceedings of the 10th International Conference on Virtual Reality Continuum and Its Applications in Industry*, VRCAI '11, pages 321–326, New York, NY, USA, 2011. ACM.

[44] K. Maeda, M. Murase, M. Doi, H. Komatsu, S. Noda, and R. Himeno. Automatic resource scheduling with latency hiding for parallel stencil applications on GPGPU clusters. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 544–556, 2012.

[45] Ben Cope, Peter Y. K. Cheung, Wayne Luk, and Lee Howes. Performance comparison of graphics processors to reconfigurable logic: A case study. *IEEE Trans. Comput.*, 59(4):433–448, April 2010.

[46] Xilinx Inc. *Virtex-6 Family Overview v2.4 DS150*, Jan 2012.

[47] E. Andres, C. Carreras, G. Caffarena, M.d.C. Molina, O. Nieto-Taladriz, and F. Palacios. A methodology for CFD acceleration through reconfigurable hardware. In *46th AIAA Aerospace Sciences Meeting and Exhibit*, pages 1–20, 2008.

[48] I. Liu, E.A. Lee, M. Viele, Guoqiang Wang, and H. Andrade. A heterogeneous architecture for evaluating real-time one-dimensional computational fluid dynamics on FPGAs. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 125–132, 2012.

[49] J. Cong, Muhuan Huang, and Yi Zou. Accelerating fluid registration algorithm on multi-FPGA platforms. In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pages 50–57, 2011.

[50] S. Kocsardi, Z. Nagy, A. Csik, and P. Szolgay. Two-dimensional compressible flow simulation on emulated digital CNN-UM. In *Cellular Neural Networks and Their Applications, 2008. CNNA 2008. 11th International Workshop on*, pages 169–174, 2008.

[51] K. Sano, T. Iizuka, and S. Yamamoto. Systolic architecture for computational fluid dynamics on FPGAs. In *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, pages 107 –116, April 2007.

[52] K. Sano, Takanori Iizuka, and S. Yamamoto. Systolic computational-memory architecture for an FPGA-based flow solver. In *Circuits and Systems, 2006. MWSCAS '06. 49th IEEE International Midwest Symposium on*, volume 1, pages 423–427, 2006.

[53] Z. Nagy, C. Nemes, A. Hiba, A. Kiss, A. Csik, and P. Szolgay. FPGA based acceleration of computational fluid flow simulation on unstructured mesh geometry. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 128–135, 2012.

[54] D. Sanchez-Roman, G. Sutter, S. Lopez-Buedo, I. Gonzalez, F.J. Gomez-Arribas, and J. Aracil. An Euler solver accelerator in FPGA for computational fluid dynamics applications. In *Programmable Logic (SPL), 2011 VII Southern Conference on*, pages 149–154, 2011.

[55] K.G. Nezami, P.W. Stephens, and S.D. Walker. Handel-C implementation of early-access partial-reconfiguration for software defined radio. In *Wireless Communications and Networking Conference, 2008. WCNC 2008. IEEE*, pages 1103–1108, 2008.

[56] J.-F.P. Labourdette. Performance impact of partial reconfiguration on multihop lightwave networks. *Networking, IEEE/ACM Transactions on*, 5(3):351–358, 1997.

[57] J. Noguera and I.O. Kennedy. Power reduction in network equipment through adaptive partial reconfiguration. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 240–245, 2007.

[58] S.U. Bhandari, S. Subbaraman, and S. Pujari. Power reduction in embedded system on FPGA using on the fly partial reconfiguration. In *Electronic System Design (ISED), 2010 International Symposium on*, pages 77–80, 2010.

[59] Naoyuki Fujita, Takashi Nakamura, Yuichi Matsuo, Katsumi Yazawa, Yasuyuki Shiromizu, Hiroshi Okubo. Feasibility Study of CFD Code Acceleration using FPGA. In *Supercomputing*, 2007.

[60] Bram van Leer. Towards the ultimate conservative difference scheme. V. A second-order sequel to Godunov's method. *Journal of Computational Physics*, 32(1):101 – 136, 1979.

[61] H. Morishita, Y. Osana, N. Fujita, and H. Amano. Exploiting memory hierarchy for a computational fluid dynamics accelerator on FPGAs. In *Field Programmable Technology, 2008. FPT 2008. International Conference on*, pages 193 –200, Dec. 2008.

[62] Takayuki Akamine, Kenta Inakagata, Yasunori Osana, Naoyuki Fujita, and Hideharu Amano. Reconfigurable out-of-order mechanism generator for unstructured grid computation in computational fluid dynamics. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 136 –142, Aug. 2012.

[63] P.L Roe. Approximate Riemann solvers, parameter vectors, and difference schemes. *Journal of Computational Physics*, 43(2):357 – 372, 1981.

[64] Bernd Einfeldt. On Godunov-type methods for gas dynamics. *SIAM J. Numer. Anal.*, 25(2):294–318, April 1988.

[65] Shigeru Obayashi and Yasuhiro Wada. Practical formulation of a positively conservative scheme. *AIAA Journal*, 32(5):1093–1095, 1994.

[66] Meng-Sing Liou. A sequel to AUSM, part II: AUSM+-up for all speeds. *Journal of Computational Physics*, 214(1):137 – 170, 2006.

[67] Eiji Shima and Keiichi Kitamura. On new simple low-dissipation scheme of AUSM-family for all speeds. *AIAA Aerospace Sciences Meeting including The New Horizons Forum and Aerospace Exposition*, 47(136):1–15, 2009.

[68] Amiram Harten, Peter D. Lax, and Bram Van Leer. On upstream differencing and Godunov-type schemes for hyperbolic conservation laws. *SIAM Review*, 25(1):pp. 35–61, 1983.

[69] Xilinx Inc. *Virtex-6 FPGA Configuration User Guide v3.2 UG360*, Nov 2010.

[70] C. Chang, J. Wawrzynek, and R.W. Brodersen. BEE2: A high-end reconfigurable computing system. *Design Test of Computers, IEEE*, 22(2):114–125, 2005.

[71] O. Lindtjorn, R. Clapp, O. Pell, O. Mencer, M. Flynn, and Haohuan Fu. Beyond traditional microprocessors for geoscience high-performance computing applications. *Micro, IEEE*, 31(2):41–49, 2011.

Publications

Related Papers

Journal Papers

[1] Mohamad Sofian Abu Talip, Takayuki Akamine, Mao Hatto, Yasunori Osana, Naoyuki Fujita and Hideharu Amano, "Adaptive Flux Calculation Scheme in Advection Term Computation Using Partial Reconfiguration", *International Journal of Networking and Computing (IJNC)*, Vol. 3, No. 2, pp. 289–306, Jul 2013.

[2] Mohamad Sofian Abu Talip, Takayuki Akamine, Yasunori Osana, Naoyuki Fujita and Hideharu Amano, "Partial Reconfiguration of Flux Limiter Functions in MUSCL Scheme Using FPGA", *IEICE Transactions on Information and Systems*, Vol. 95-D, No. 10, pp. 2369–2376, Oct 2012.

International Conference Papers

[3] Mohamad Sofian Abu Talip, Takayuki Akamine, Yasunori Osana, Naoyuki Fujita and Hideharu Amano, "Dynamically Reconfigurable Flux Limiter Functions in MUSCL Scheme", *Proc. of IEEE 7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pp. 1–7, Jul 2012.

[4] Mohamad Sofian Abu Talip, Takayuki Akamine, Yasunori Osana, Naoyuki Fujita and Hideharu Amano, "Cost Effective Implementation of Flux Limiter Functions Using Partial Reconfiguration", *Proc. of the 8th International Symposium on Applied Reconfigurable Computing (ARC 2012)*, pp. 215–226, Mar 2012.

[5] Mohamad Sofian Abu Talip and Hideharu Amano, "A Design of One-Dimensional Euler Equations for Fluid Dynamics on FPGA", *Proc. of IEEE 1st International Symposium on Access Spaces (IEEE-ISAS'11)*, pp. 170–173, Jun 2011.

Other Papers

International Conference Papers

[6] Mohamad Sofian Abu Talip, Aisha Hassan Abdalla, Abdurazzag Ali Aburas, A.H.M. Zahirul Alam and Ahmed Asif, "Knowledge-based Disk Scheduling Policy Using Fuzzy Logic", *In Proc. of International Conference on Computer and Communication Engineering (ICCCE 2010)*, pp. 1–6, May 2010.

[7] Mohamad Sofian Abu Talip, Aisha Hasan Abdalla, Ahmed Asif and Abdurazzag Ali Aburas, "Fuzzy Logic Based Algorithm for Disk Scheduling Policy", *In Proc. of International Conference of Soft Computing and Pattern Recognition (SOCPAR'2009)*, pp. 746–749, Dec 2009.

[8] Mohamad Sofian Abu Talip, Abdurazzag Ali Aburas, Aisha Hasan Abdalla, and Ahmed Asif, "Information Quality Measurement for Disk Scheduling Algorithm", *In Proc. of International Conference on Software Engineering and Computer Systems (ICSECS'09)*, pp. 159–164, Oct 2009.

Appendix A

IEEE Standard 754 Floating Point Numbers

IEEE Standard 754 floating point is the most common representation today for real numbers on computers, including Intel-based PC's, Macintoshes, and most Unix platforms. This is a brief overview of IEEE floating point and its representation.

A.1 What are floating point numbers?

There are several ways to represent real numbers on computers. Fixed point places a radix point somewhere in the middle of the digits, and is equivalent to using integers that represent portions of some unit. For example, one might represent 1/100ths of a unit; if you have four decimal digits, you could represent 10.82, or 00.01. Another approach is to use rationals, and represent every number as the ratio of two integers.

Floating-point representation - the most common solution - basically represents reals in scientific notation. Scientific notation represents numbers as a base number and an exponent. For example, 123.456 could be represented as 1.23456×102. In hexadecimal, the number 123.abc might be represented as $1.23abc \times 162$.

Floating-point solves a number of representation problems. Fixed-point has a fixed window of representation, which limits it from representing very large or very small numbers. Also, fixed-point is prone to a loss of precision when two large numbers are divided.

Floating-point, on the other hand, employs a sort of "sliding window" of precision appropriate to the scale of the number. This allows it to represent numbers from 1,000,000,000,000 to 0.0000000000000001 with ease.

A.2 Storage Layout

IEEE floating point numbers have three basic components: the sign, the exponent, and the mantissa. The mantissa is composed of the fraction and an implicit leading digit. The exponent base (2) is implicit and need not be stored.

Table A.1 shows the layout for single (32-bit) and double (64-bit) precision floating-point values. The number of bits for each field are shown (bit ranges are in square brackets):

Table A.1: Layout for single and double precision floating-point values.

	Sign	Exponent	Fraction	Bias
Single Precision	1[31]	8[30-23]	23[22-00]	127
Double Precision	1[63]	11[62-52]	52[51-00]	1023

A.2.1 The Sign Bit

The sign bit is as simple as it gets. 0 denotes a positive number; 1 denotes a negative number. Flipping the value of this bit flips the sign of the number.

A.2.2 The Exponent

The exponent field needs to represent both positive and negative exponents. To do this, a bias is added to the actual exponent in order to get the stored exponent. For IEEE single-precision floats, this value is 127. Thus, an exponent of zero means that 127 is stored in the exponent field. A stored value of 200 indicates an exponent of (200-127), or 73. For double precision, the exponent field is 11 bits, and has a bias of 1023.

A.2.3 The Mantissa

The mantissa, also known as the significand, represents the precision bits of the number. It is composed of an implicit leading bit and the fraction bits.

To find out the value of the implicit leading bit, consider that any number can be expressed in scientific notation in many different ways. For example, the number five can be represented as any of these:

- 5.00×10^{0}
- 0.05×10^{2}
- 5000×10^{-3}

In order to maximize the quantity of representable numbers, floating-point numbers are typically stored in normalized form. This basically puts the radix point after the first non-zero digit. In normalized form, five is represented as 5.0×100.

A nice little optimization is available to us in base two, since the only possible non-zero digit is 1. Thus, we can just assume a leading digit of 1, and don't need to represent it explicitly. As a result, the mantissa has effectively 24 bits of resolution, by way of 23 fraction bits.

A.3 Ranges of Floating Point Numbers

The range of positive floating point numbers can be split into normalized numbers (which preserve the full precision of the mantissa), and denormalized numbers, which use only a portion of the fractions's precision. Table A.2 shows range of floating point numbers.

Table A.2: Range of floating point numbers.

	Denormalized	Normalized
Single Precision	$\pm 2^{-149}$ to $(1-2^{23}) \times 2^{-126}$	$\pm 2^{-126}$ to $(2-2^{23}) \times 2^{127}$
Double Precision	$\pm 2^{-1074}$ to $(1-2^{52}) \times 2^{-1022}$	$\pm 2^{-1022}$ to $(2-2^{-52}) \times 2^{1023}$

A.4 Special Values

IEEE reserves exponent field values of all 0s and all 1s to denote special values in the floating-point scheme.

A.4.1 Zero

Zero is not directly representable in the straight format, due to the assumption of a leading 1. Zero is a special value denoted with an exponent field of zero and a fraction field of zero. Note that -0 and +0 are distinct values, though they both compare as equal.

A.4.2 Denormalized

If the exponent is all 0s, but the fraction is non-zero (else it would be interpreted as zero), then the value is a denormalized number, which does not have an assumed leading 1 before the binary point. Thus, this represents a number $(-1)^s \times 0.f \times 2^{-126}$, where s is the sign bit and f is the fraction. For double precision, denormalized numbers are of the form $(-1)^s \times 0.f \times 2^{-1022}$.

A.4.3 Infinity

The values + infinity and - infinity are denoted with an exponent of all 1s and a fraction of all 0s. The sign bit distinguishes between negative infinity and positive infinity. Being able to denote infinity as a specific value is useful because it allows operations to continue past overflow situations. Operations with infinite values are well defined in IEEE floating point.

A.4.4 Not a Number

The value NaN (*Not a Number*) is used to represent a value that does not represent a real number. NaN's are represented by a bit pattern with an exponent of all 1s and a non-zero fraction.