Architectural Explorations for Streaming Accelerators with Customized Memory Layouts

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Author:
Muhammad Shafiq

Advisors:
Dr. Miquel Pericàs, Prof. Nacho Navarro, Prof. Eduard Ayguadé

Department of Computer Architecture

Submitted to the Departament d'Arquitectura de Computadors in Partial Fulfillment of the Requirements for

Doctor of Philosophy (PhD)

Barcelona - April 2012

بسم الله الرحمن الرحيم

To my great father Prof. Ch. Faqir Muhammad, my sweet mother Kaniz Fatima, my lovely wife Sofia Shafiq and my charming little kids Muhammad Hamza, Hamail Shafiq and Muhammad Hanzalah

Acknowledgements

* All the praises and thanks be to Allah(swt), Lord of the worlds. The Most Gracious, The Most Merciful. The Owner of the day of the Judgment. He alone we worship and alone His aid we seek. We ask Him to guide us to the straight way. The way of those on whom He has bestowed His Grace, not the way of those who earned His anger nor of those who went astray.

* This work was not possible to complete without the help of my advisors Prof. Eduard Ayguadé, Prof. Nacho Navarro and specially Dr. Miquel Pericàs. Dr. Miquel Pericàs is the one who was actually absorbing and tolerating my strange ideas and helping me extraordinarily to bring them into a good and meaningful shape before that these are presented to our seniors and as well at various research platforms. He has been conducting regular weekly meetings throughout the years. I always found him ready to spare time for the research discussions. Prof. Nacho Navarro is the one who is guiding me in my studies since the very first time I joined UPC. He always been with a tight time schedule due to his teaching activities at UPC, his collaborations with university of Illinois and also due to the responsibility of taking care of his father suffering from the health problems. In-spite of all this, he always rushed to join the research meetings by sparing time from his schedule. Prof. Eduard Ayguadé is the great gentleman because of whom I am here. I found him so much experienced researcher that his few precise words on our research activities have always been enough to highlight the possible issues in a research direction. Moreover, he is the one who actually helped me and my family both socially and financially to survive in the costly city of Barcelona. I am really thankful to all of my advisors from the depths of my heart.

* I would like to express my gratitude to Prof. Mark Oskin, Dr. Yoav Etsion, Dr. Nikola Puzovic, Dr. Felipe Cabarcas, Alejandro Rico and Mauricio Araya-Polo. I did many sittings with all of them to push this work ahead in the right direction. I am also thankful to Dr. Isaac Gelado for his feed-back on some of my questions about the internals of GPU architecture. It is not possible to forget to say many thanks to Prof. Marisa Gil Gomez and Dr. Mauricio Alvarez Mesa. They are the really ones who actually gave me a startup in my studies at UPC. I did a project on the CABAC decoder for H.264 under their and Prof. Nacho Navarro's supervision. This learning helped me a lot to effectively pursue later studies.

* I am extremely grateful to my father and a great educator Prof. Ch. Faqir Muhammad and my mother Kaniz Fatima. They always encouraged me in my studies and I believe that all of my successes in the life are just because of their prayers and supplications for me. My wife Sofia Shafiq also deserve lot of thanks. In-fact, she exactly demonstrated the saying: *Behind Every Successful Man, There Is a Woman*. I would also like to express my gratitude to my brothers Muhammad Yasin Shami and Muhammad Saeed for their full support to me in dealing various social and administrative issues in Pakistan while I am sitting here abroad. I also wish to thank my friends and colleagues Branimir Dickov, Tassadaq Hussain, Islam Almalkawi and Ahmed Helmi for sharing their time and having nice technical and non-technical discussions on the lunch and coffee breaks.

* The acknowledgements will remain incomplete if I do not quote the organizations and the projects who financially supported this thesis. These include National Engineering and Scientific Commission Pakistan, Barcelona Supercomputing Center (BSC), Technical University of Catalonia (UPC), The Ministry of Science and Innovation of Spain (Project: Computaciòn de Altas Prestaciones V, TIN2007-60625), The European Union Framework Program-7 (Project: Scalable Computer Architecture, IST-2006-27648) and the HiPEAC Network of Excellence (High-performance and Embedded Architectures and Compilers, ICT 217068). I am extremely thankful to all these organizations for their full support during the work.

Abstract

The basic concept behind the architecture of a general purpose CPU core conforms well to a serial programming model. The integration of more cores on a single chip helped CPUs in running parts of a program in parallel. However, the utilization of huge parallelism available from many high performance applications and the corresponding data is hard to achieve from these general purpose multicores. Streaming accelerators and the corresponding programing models improve upon this situation by providing throughput oriented architectures. The basic idea behind the design of these architectures matches the everyday increasing requirements of processing huge data sets. These high-performance throughput oriented devices help in high performance processing of data by using efficient parallel computations and streaming based communications.

The throughput oriented streaming accelerators – similar to the other processors – consist of numerous types of micro-architectural components including the memory structures, compute units, control units, I/O channels and I/O controls etc. However, the throughput requirements add some special features and impose other restrictions for the performance purposes. These devices, normally, offer a large number of compute resources but restrict the applications to arrange parallel and maximally independent data sets to feed the compute resources in the form of streams.

The arrangement of data into independent sets of parallel streams is not an easy and simple task. It may need to change the structure of an algorithm as a whole or even it can require to write a new algorithm from scratch for the target application. However, all these efforts for the re-arrangement of application data access patterns may still not be very helpful to achieve the optimal performance. This is because of the possible micro-architectural

constraints of the target platform for the hardware pre-fetching mechanisms, the size and the granularity of the local storage and the flexibility in data marshaling inside the local storage. The constraints of a general purpose streaming platform on the data pre-fetching, storing and maneuvering to arrange and maintain it in the form of parallel and independent streams could be removed by employing micro-architectural level design approaches. This includes the usage of application specific customized memories in the front-end of a streaming architecture.

The focus of this thesis is to present architectural explorations for the streaming accelerators using customized memory layouts. In general the thesis covers three main aspects of such streaming accelerators in this research. These aspects can be categorized as : i) Design of Application Specific Accelerators with Customized Memory Layout ii) Template Based Design Support for Customized Memory Accelerators and iii) Design Space Explorations for Throughput Oriented Devices with Standard and Customized Memories

This thesis concludes with a conceptual proposal on a Blacksmith Streaming Architecture (*BSArc*). The Blacksmith Computing allow the hardware-level adoption of an application specific front-end with a GPU like streaming back-end. This gives an opportunity to exploit maximum possible data locality and the data level parallelism from an application while providing a throughput natured powerful back-end. We consider that the design of these specialized memory layouts for the front-end of the device are provided by the application domain experts in the form of templates. These templates are adjustable according to a device and the problem size at the device's configuration time. The physical availability of such an architecture may still take time. However, simulation framework helps in architectural explorations to give insight into the proposal and predicts potential performance benefits for such an architecture.

Contents

List of Figures **xv**

List of Tables **xix**

1 Introduction **1**

1.1 Streaming Architectures 2

1.2 Target Computing Architecture 3

1.3 Thesis Contributions 6

1.3.1 Design of Application Specific Accelerators with Customized Memory Layouts 7

1.3.2 Template Based Design Support for Customized Memory Accelerators 8

1.3.3 Deign Space Explorations for Throughput Oriented Devices with Standard & Customized Memories 9

1.4 State of the Art of Streaming Accelerators 10

1.4.1 Imagine 11

1.4.2 Raw 12

1.4.3 IBM Cell/B.E 12

1.4.4 GPU 13

1.5 Thesis Organization 14

1.6 Summary 15

I Design of Application Specific Accelerators with Customized Local Memory Layout 17

2 A Design of Streaming Architecture for Structured Grid Application **19**

2.1 High Performance Computing for Structured Grids 20

2.2 3D Stencil Computations 22

2.3 3D Memory Organizations for Stencil Computations in Reconfigurable Logic . 26
2.3.1 Front-End (Data-Engine) 27
2.3.2 Working Principle of the Data-Engine 29
2.3.3 Back-End (Compute-Engine) 31
2.3.4 Control-Engine 32
2.3.5 Evaluations 32
2.3.6 Results and Discussion 33
2.4 RTM Algorithm and its Mapping on FPGA 36
2.4.1 RTM Implementation Problems 38
2.4.1.1 Memory 38
2.4.1.2 Input/Output 39
2.4.1.3 Computation 40
2.4.2 Application Specific Design of RTM 40
2.4.2.1 ABC 42
2.4.2.2 Correlation and Illumination 42
2.4.2.3 Compression and decompression 43
2.5 Results and Discussion 43
2.6 Summary 45

3 Generalization of 3D-Memory as a Common Memory Layout 47
3.1 Application Specific Front-Ends 48
3.2 Compute Models for Multi-Kernel Applications 49
3.2.0.4 Full Reconfiguration (FRC) 50
3.2.0.5 Spatially Mapped-Local Memory (SM-LM) 50
3.2.0.6 Spatially Mapped-Shared Memory (SM-SM) 50
3.3 Front End Memory Layout For Reconfigurable Accelerators 51
3.3.1 Example FEM layouts for Scientific Kernels 53
3.3.1.1 Digital Filters 53
3.3.1.2 Sparse Matrix-Vector Multiplication (SpMVM) 54
3.3.1.3 Fast Fourier Transform (1D & 2D) 54
3.3.1.4 N-Body (Naive/Barnes-Hut), DFT (1D/2D) and Matrix Multiplication 54
3.3.1.5 Acoustic Wave Equation Solver (AWE) 55
3.4 Evaluations 55
3.5 Results and Discussion 56
3.5.1 AWE (WFC, BPC) Solver 56

3.5.2 N-Body Hermite Algorithm 57
3.6 Summary 60

II Template Based Design Support for Customized Memory Accelerators 61

4 The DATE System 63
4.1 Templates in Reconfigurable Computing 63
4.2 Background 67
4.3 The DATE System 68
4.3.1 The DATE System : From The Front-Side 68
4.3.1.1 Functional Bucket Types (FBT) 69
4.3.1.2 Read/Write Data Access Channels (DACs) 70
4.3.1.3 Abstract Methods 70
4.3.1.4 Parameter Set 70
4.3.1.5 System I/O 70
4.3.1.6 The Translations 71
4.3.1.7 DATE Input Source Code Types (Implicit and Explicit) . . . 71
4.3.2 The DATE System : At The Back-End 72
4.3.2.1 Template's Expansion to C 72
4.3.2.2 Template's Expansion to HDL 73
4.3.3 The DATE System Center: The Domain-Translator 73
4.3.3.1 Template Design 75
4.3.4 The Date System : Template Library 76
4.3.4.1 Example HDL Template (The FIR Domain) 77
4.3.4.2 Example C Template (The FIR Domain) 78
4.4 Generation of Complete System Architecture Using DATE System 79
4.4.1 DATE System for Generation of TARCAD Based Designs 80
4.5 Evaluations 82
4.5.1 A Stencil Template Scalable to Multidimensional Stencil 82
4.5.1.1 Stencil Type 84
4.5.1.2 Stencil Dimensions 84
4.5.1.3 Parallel Computations 84
4.5.2 2D-FFT Translations 85
4.5.3 Handling Applications with Multiple Kernels 86
4.5.4 Results and Discussion 87

4.6 Summary 90

5 A Throughput Oriented Template Based Streaming Accelerator 91

5.1 Customized Accelerators 92

5.2 The TARCAD Architecture 94

5.2.1 Accelerator Models for Supercomputing 94

5.2.2 The External Memory Interface 95

5.2.3 The Application Specific Data Management Block 95

5.2.4 The Algorithm Compute Back-End 97

5.2.5 The Event Managing Block 97

5.3 TARCAD Implementation 98

5.4 Application Kernels on TARCAD 99

5.4.1 Matrix-Matrix Multiplication (MxM) 99

5.4.2 Acoustic Wave Equation (AWE) 101

5.4.3 Smith Waterman (SW) 103

5.4.4 Fast Fourier Transform (FFT) 103

5.4.5 Sparse Matrix-Vector Multiplication (SpMV) 104

5.4.6 Multiple Kernels On TARCAD 104

5.5 Evaluations 105

5.6 Results and Discussion 106

5.6.1 Matrix-Matrix Multiplication (MxM) 106

5.6.2 Acoustic Wave Equation (AWE) 108

5.6.3 Smith-Waterman (SW) 108

5.6.4 Fast Fourier Transform (FFT) 108

5.6.5 Sparse Matrix-Vector Multiplication (SpMV) 109

5.7 Summary 109

III Design Space Explorations for Throughput Oriented Devices with Standard and Customized Memories 111

6 A Simulator Framework for Performance Characterization of Streaming Architectures 113

6.1 Simulators and Computer Architecture Research 114

6.2 The Simulator Framework 117

6.3 Trace Generation 117

6.3.1 S-S Translator 119

6.3.2 TTrace Tool 120
6.4 Device Map 121
6.4.1 Garbage (Built-in) Removal 122
6.4.2 WARP Instructions Formation 122
6.4.3 Coalescing Effects 123
6.4.4 Registers and Shared Memory Handling 123
6.4.5 Grouping Blocks 123
6.5 Device Simulation 124
6.6 Evaluations 126
6.6.1 Case 1: Memory Micro-Kernels 127
6.6.1.1 (a) SM Level Evaluations 127
6.6.1.2 (b) Multiple Block Evaluations 127
6.6.2 Case 2. Application Kernels 127
6.7 Results and Discussion 128
6.7.1 SArcs Limitations and the Future Work 131
6.8 Summary 132

7 Design Space Explorations for Streaming Architectures using SArcs 133
7.1 Design Space Explorations 134
7.2 Effectiveness of the Design Space Exploration Tool 137
7.3 Design Space Explorations for GPU 140
7.3.1 Application Kernels 140
7.3.2 Base Line Architecture 141
7.3.3 Simulation Platform 141
7.3.4 Evaluated Architectural Configurations 142
7.3.4.1 Global Memory Bandwidth 142
7.3.4.2 Data Channels Between Memory Hierarchy and SMs 142
7.3.4.3 Cache Memory 143
7.3.4.4 Streaming Multiprocessors 143
7.4 Results and Discussion 143
7.5 Blacksmith Computing 146
7.5.1 Target Platform Model 147
7.6 Application Specific Front-Ends 148
7.6.1 2D-FFT 149
7.6.2 Matrix Multiplication 150
7.6.3 3D-Stencil 151

7.7 Design Space Exploration Environment 152
7.7.1 Base Line Architecture 152
7.7.2 Simulation Platform 152
7.8 Results and Discussion 153
7.9 Summary 155

8 Conclusions and Future Work 157
8.1 Conclusions 157
8.2 Future Work 160
8.2.1 Unified Front-End memory for BSArc 161
8.2.2 CUDAb 163

Publications 165
I Publications 167
II Other Papers and Extended Abstracts 168
III UPC Research Reports 168

References 171

List of Figures

1.1 An example view of data flow in a streaming execution 3

1.2 H.264 video decoder (Top), 2D-Wave approach for exploiting MB parallelism (Bottom Figure Source : [1]). The arrows indicate the MBs dependencies. . . . 4

1.3 The simplified model of target computing architecture 5

1.4 Streaming processors: (a) Imagine (b) Raw 12

1.5 (a) IBM Cell/B.E (b) GPU 13

2.1 Odd-symmetric 3D stencil for n=8 ($8 \times 9 \times 8$ stencil) : (a) 3D- stencil algorithm where *p[..]* represents input volume and *C[..]* are the constants), (b) Points access pattern form 3-dimensions 23

2.2 (a) 3D-stencil core and interfaces, (b) Extended base volume, partitioned volume and axis conventions 25

2.3 (a) Generic architecture of Data-Engine : Implementing three memory levels (Y-Layer, X-Layer, Z-Layer). “Pr0/Cr0” and “Pr1/Cr1“ are the plane/Column read pointers for lower and upper halves respectively at Y-Layer and X-Layer, (b) $8 \times 9 \times 8$ specific simplified (one-side write and other side read at each level) layout of the special purpose data engine for the 3D stencil computation used in RTM mapping. 28

2.4 Architecture of Compute-Engine in the Back-End for n=8 stencil type. C(0) to C(12) are the constant coefficient multipliers of the stencil. P0 to p23 and Cp are the input data operands coming from Front-Engine. *Cp* is the central point of an odd symmetric stencil 31

2.5 Internal Bandwidths (Bytes/Second) achieved for Multi-Volume and Single-Volume design approaches 34

2.6 Ratios between Internal and external Bandwidths for Multi-Volume and Single-Volume design approaches 35

2.7 The RTM Algorithm 37

2.8 (a) A generic 3D stencil structure, (b) a 3D 7-point stencil, and (c) its memory access pattern. 39
2.9 Elapsed times for computation only experiments, 100 steps, forward and backward 43
2.10 RTM forward and backward with stack 5, and high level of compression. Hypernode is a technology proposed by IBM for providing high-performance I/O, for instance for the Cell/B.E. platform. 44

3.1 Compute Models that are evaluated in this work 49
3.2 FEM based conceptual machine architecture (a) and Front-End Memory Layouts for various kernels shown in the sub-figures b, c, d, e and f 52
3.3 AWE and N-Body Performance Evaluation for **FRC**,**SM-LM** and **SM-SM** Compute Models 58

4.1 Streaming Environment with programmable streaming memory controller . . . 66
4.2 The Architecture of the DATE System 69
4.3 An example of explicit and implicit coding styles with their mappings for the generation of an accelerator architecture 72
4.4 Internal Flow of the Domain Translator 74
4.5 Direct HDL : Template for FIR Shifter Module 76
4.6 Direct HDL : Template for FIR System Module 77
4.7 C Template : FIR Computational Module 79
4.8 C Template : FIR System Template 79
4.9 TARCAD Conceptual Diagram 80
4.10 Integration of specialized design modules for a kernel into TARCAD system . . 81
4.11 DATE System Support for TARCAD Design 82
4.12 The Basic FBT Template structure for the abstract Stencil Domain (a), Generation of FBT for 3-Dimensional Stencil (b) 83
4.13 2D-FFT Architecture 85
4.14 Implicit Type of HLL Description for RTM Implementation 86
4.15 Application kernels Implementation Time for various cases 87

5.1 A High Level View of TARCAD 95
5.2 TARCAD architectural layout 96
5.3 TARCAD Implementation: Environment of the DATE System 99
5.4 MxM : (a) Matrices elements' distribution into application specific memory layout and (b) Pseudo code for matrices data accesses by the *Event Manager* . 100

5.5 Odd symmetric 3D stencil: (a) Large input volume partitioned into sub volumes (b) Pseudo code for sub-volume accesses by the *Event Manager* 101

5.6 Smith Waterman : Left: The Systolic array of compute blocks, Right: Architectural support for inter-compute block communication. 102

5.7 Mapping an existing FFT core on TARCAD 104

5.8 Architectural Efficiency for TARCAD and GPU based Kernels. Performance Numbers are only shown for FPGA based Designs. The device used for FPGA is Virtex-6 XC6VSX475T and the corresponding GPUs are (a) MxM (GPU: Tesla C2050 [2]) (b) AWE (GPU: Tesla C1060 [3]) (c) SW (GPU: Tesla C1060 [4]), (d) FFT (GPU: Tesla C2050 [2]) (e) SpMV (GPU: GTX 280, Cache Enabled [5]) 107

6.1 The Framework of *SArcs* 117

6.2 Some example declarations & definitions in modified cuda header file (“mcuda.h”) 118

6.3 An example code insertion for the replacement of the target gpu kernel call . . 118

6.4 An Example SIMT Trace (The left side numbering (1,2,3...) is added just to describe the trace inside the text) 121

6.5 GPU Simulation Core (GSCore) 125

6.6 Evaluations Methodology (SArcs Program Executions vs Real GPU Program Executions) 126

6.7 Case 1 (a): Memory micro-kernels for SM Level evaluations (a) GPU - Tesla C2050 execution time (b) SArcs simulated execution time (c) Averaged Percentage performance error for each micro-kernel 128

6.8 Case 1 (b): Memory micro-kernels with multiple blocks evaluations (a) GPU - Tesla C2050 execution time (b) SArcs simulated execution time (c) Averaged Percentage performance error for each micro-kernel 129

6.9 Case 2: (a) Matrix Matrix multiplication : Simulated and real execution time using L1 and with-out L1 (b) 2D-Convolution : Simulated and Real execution time using Tiling in the Shared Memory (c) Vector Reduction : Simulated and Real execution time using shared memory and multiple invocations of the GPU-kernel from the CPU during the reduction process (d) Percentage of the error of the simulated performance to that of the real one on GPU for the kernels 130

7.1 Establishment of the effectiveness of the simulator (SArcs) by performance characterization against the real GPU for the base line architecture (NVIDIA's Tesla C2050) (a) Memory Micro-Kernels (real GPU Executions) (b) Memory Micro-Kernels (Simulated Executions) (c) Vector Reduction using shared Memory (d) 2D-Convolution using shared memory (e) Matrix Multiplication with/without L1 (f) 3D-Stencil Kernel using shared memory (g) 2D-Fast Fourier Transform 138
7.2 Matrix multiplication Kernel (No shared memory) 144
7.3 3D-Stencil Kernel using shared memory 144
7.4 2D-Convolution Kernel 144
7.5 Vector Reduction using shared memory and multiple Invocations of the device . 144
7.6 (a) An Analogy for the Blacksmith Computing (b) The simplified target platform model 146
7.7 GPU Simulation Core (GSCore) with configurable L2 148
7.8 2D-FFT Memory Layout 149
7.9 Matrix-Matrix Multiplication (MM) Memory Layout 150
7.10 3D-Stencil Memory Layout 151
7.11 The application kernel's execution times for the three configurations : (i) Base Line (L2 Cache disabled and No Application Specific Memory) (ii) L2 Cache: Using only L2 cache (iii) AS Mem: Using only Application Specific (AS) memory. (a) 2D-FFT (b) Matrix Multiplication (c) 3D-Stencil (d) The speedups for the test kernels using Application Specific (AS) memory with reference (Ref) to: The base line (Base) architectural configuration and L2 Cache Based Executions 154

8.1 Unified Front-End Memory for BSArc (Under Consideration) 162

List of Tables

2.1 Dimensions ($Xdim \times Zdim$ whereas Ydim=∞) for various volume decompositions and their BlockRAM consumption in the **Multi-Volume Design** Approach. Total Data-Engines in Front-End = 3 (for n=2), 2(for n=4,6,8). 24

2.2 Dimensions ($Xdim \times Zdim$ whereas Ydim=∞) for various volume decompositions and their BlockRAM consumption in the **Single-Volume Design** Approach. Total Data-Engines in Front-End : 1 25

2.3 Processor technical specs. Peak GFlops are obtained considering SIMD extensions 33

2.4 Slice counts and throughput per slice for Multi-Volume and Single-Volume Approach. The selected domain decomposition (Tables 2.1 and 2.2) is the one delivering the highest throughput as in Figure 2.5 36

2.5 Performance values for all Architectures when computing the 8x9x8 stencil. Native compilers (xlc,icc) have been used at -O3 optimization level. Internal BW refers to the bandwidth observed by the 3D stencil algorithm 36

3.1 Comparison of resources (FEM layout & Controls) required by AWE Solver and N-Body Hermite Algorithm for the three computing models. 56

4.1 DATE System Evaluations are done on Virtex-4 LX200 device using Xilinx ISE 11.1 tool suite. In-case of 2D-FFT, the table shows only results for the computational FFT (1D-FFT) part with templates for both C and HDL. The memory part uses less than 5% of Block RAMs (square frames) and less than 2% of slices to implement WR and RD toggling logic in our example case studies 88

5.1 Applications Mapped to TARCAD using Virtex-6 & ISE 12.4 106

LIST OF TABLES

1 Introduction

Stream processing is being used extensively from the smart-phones to high performance supercomputing machines. It is no more surprising to say that the streaming devices promises to be a major computing force in the coming decades. Generally, the stream processing uses streaming architectures like GPUs, Cell/BE and application specific designs on the reconfigurable devices. The basic architecture of a streaming device decouples computations from the memory accesses. This makes it possible to improve upon the both (i.e. computations and communications) architectural aspects independent of each other across a streaming interface. The compute components in the streaming architecture consume and produce unbounded data vectors. These kind of architectures can deliver performance if-and-only-if the data-management front-end of a device is able to arrange data-sets in the form of independent sets of streams.

Software based approaches are normally required to accomplish the job of data decoupling and its arrangement. However, these approaches may not perform efficiently for arbitrary application domains. This is because of a possible mismatch between an application requirement of a memory configuration and the available hardware memory structure on the target generic streaming architecture. Therefore, an application specific hardware support could be very beneficial to increase the performance for many applications by improving the management of data before that it is streamed to the compute units. This application specific management of data results either or both as an increase in the data locality for the application data and an optimized data level parallelism for the streaming back-end.

This chapter presents an overall view of the thesis work which centers upon the architectural explorations for the streaming accelerators with customized front-ends. The chapter starts by giving a general introduction on the streaming architectures. This is followed by an unveiling of our conceptual streaming model along with its high level descriptions. The chapter also present an overall view of the contributions made during this thesis work. Before summarizing the chapter, we will briefly look at some of the existing streaming accelerators and as well the organization of this thesis document.

1.1 Streaming Architectures

The extensive research of last many decades to improve upon the general purpose single core architectural features like the branch predictions, pipelining, out-of-order processing and frequency scaling have nearly touched their corresponding walls. This is because the research is almost saturated for further improvements in the first two while the out-of-order processing causes an extreme microarchitectural complexity of a processor to get small benefits for suitable applications. Increasing the device frequency while reducing the feature size is beneficial for all applications but higher power dissipation and fault tolerance issues do not allow to further improve the performance by scaling the frequency. Therefore, the negative slope for the performance opportunities from further improvements on top of the single core processors have sharply shifted the research focus to the parallel computing architectures, algorithms and techniques. This trend of exploring the parallel paradigms expected to continue deep into the future [6]. However, the last decade's research efforts have already laid-down a strong foundation for the future parallel computing.

In a comparison to the traditional parallel computing on general purpose cores, the streaming architectures have exhibited significant performance advantages in the application domains such as multi-media, graphics, digital signal processing and some scientific applications. The other application domains like the ones using data in the form of non-linear grids could also utilize the potential of stream computing. This may require the corresponding algorithms to go through some radical changes to exploit the streaming architectures. These changes make possible the decoupling of the data accesses from the computations and their separate optimizations.

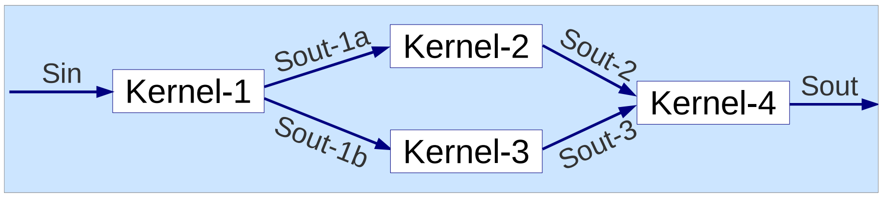


Figure 1.1: An example view of data flow in a streaming execution

An example view of the data flow through a streaming device is shown in the Figure 1.1. The figure shows that a stream (S_{in}) of data is processed by a *kernel-1*. The output data stream from this kernel is split into two parallel work loads. These work loads are processed by two different kernels (*kernels-2 and 3*). The processed outputs from these kernels are combined and further computed by *kernel-4*. Later, the results are written back as an output stream (S_{out}).

The streaming architectures lead to throughput oriented computing on devices with parallel streaming architectures. These throughput oriented architectures execute parallel workloads while attempting to maximize total throughput, even though sacrificing the serial performance of a single task [7]. The streaming devices, normally, offer a large number of compute resources but restrict the applications to arrange parallel and maximally independent data sets to feed the compute resources as streams. That's why the streaming applications use a data-driven approach. The performance for these applications on an arbitrary platform depends how well the data is managed into streams before forwarding to the compute components.

1.2 Target Computing Architecture

The efficient data management is a key to the performance for many HPC applications [8]. The programmable devices normally support efficient utilization of data by providing a fixed architecture of caches or scratch pad memories [9]. These caches or scratch pad memories are designed on the basis of few heuristics that are generic enough to provide varying degree of performance enhancement for various applications. However, the performance for certain applications on a device can still be improved by providing more customized memory layouts for those applications. In order to highlight the possibilities of this memory customization for a stream of data using

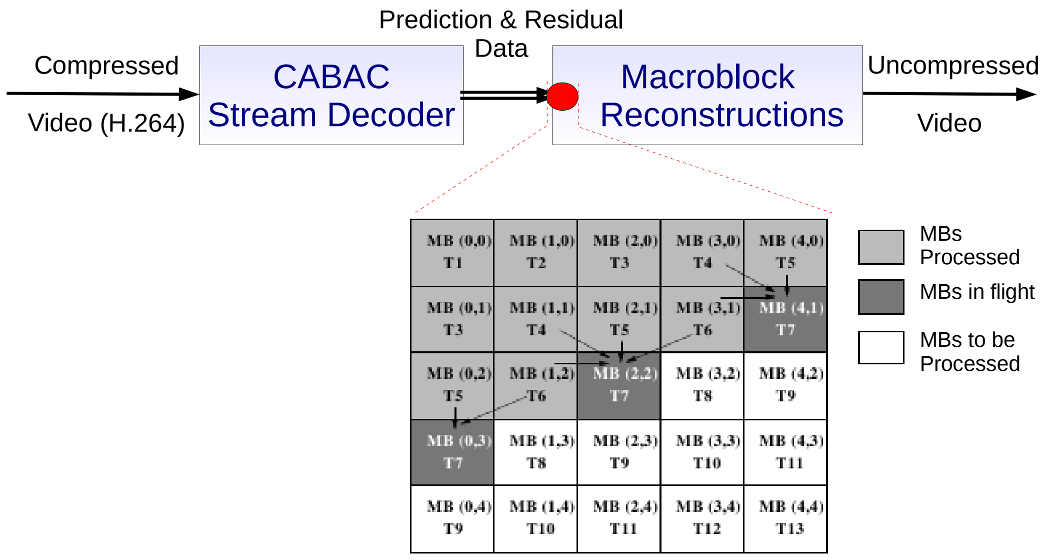


Figure 1.2: H.264 video decoder (Top), 2D-Wave approach for exploiting MB parallelism (Bottom Figure Source : [1]). The arrows indicate the MBs dependencies.

an application specific front-end, we show an example of H.264 video decoder. The top of the Figure 1.2 shows the video decoder in the form of two blocks. The CABAC entropy decoding block is extremely sequential by its nature. It provides parallelism only at frame or slice (slice can constitute a full frame or part of the frame) level of the compressed video stream. However, once a video slice is decoded, it is possible to identify the Macroblocks (MBs) boundaries. The bottom of the figure shows that the MBs with identical *Ts* can be executed in parallel [1] during the Macroblock reconstruction phase. However, this reconstruction process faces an ordered sparsity in the data because of the information required from previously decoded MBs as shown by the arrows in the Figure 1.2. The MBs reconstruction process can be accelerated with the support of a specialized memory layout. This layout would need a customized memory structure consuming a local memory of size less than the size of the memory to hold data for twice the number of MBs in the principal diagonal of the slice. This special memory structure can hold MB's data in an independently accessible form for the process of parallel decoding of the blocks by a large number of compute units in the back-end. Moreover, the functionality attached to the design of this specialized memory architecture will also keep the left, top and diagonal data dependencies from the previously decoded blocks in the required ordered for all the macroblocks decode-able

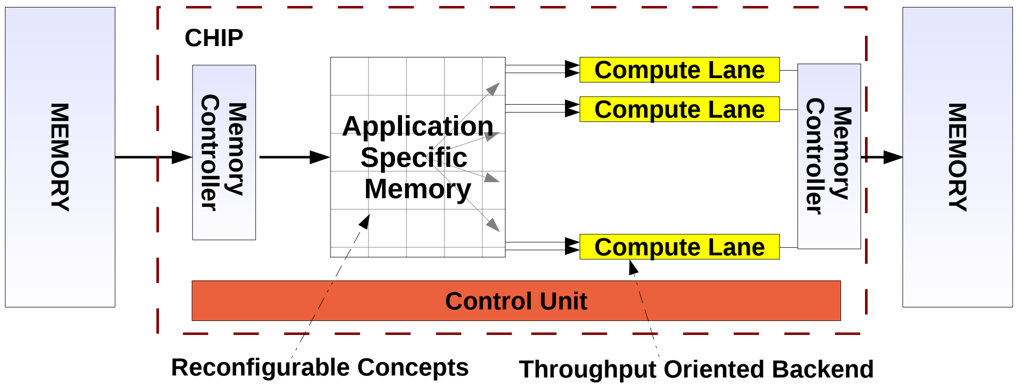


Figure 1.3: The simplified model of target computing architecture

in parallel.

A simplified form of our target architectural model is shown in Figure 1.3. It can be seen from the figure that the model can be partitioned into three representative main blocks. The global memory interface is based on a memory controller. We consider this interface to be based on a programmable pattern based memory controller (PPMC) [10] for fetching large data sets. Next, it comes the region for the application specific memory layouts. These application specific customizations of the memory layouts can be achieved using coarse grained or fine grained reconfigurable regions. However, it is also possible that a set of applications can share a common customized memory layout [11]. This memory layout reorganizes data basically for two reasons : i) Data arrangement for distribution as parallel work loads and (ii) Transformation of data from a memory default arrangement to an application required arrangement. The transformation of data arrangement inside specialized memory is important for many applications. A simple example is the memory layout for the FFT (decimation in time) where data is written sequentially to the layout while it is read in a bit-reversed order. Therefore, our model keeps separate write and read interfaces. The customized memory region of the model can reshape and unfold data-sets specific-to-an-application requirement by configuring and incorporating domain specific architectural templates developed by the domain experts [12]. This means that the programmer does not need to worry about the hardware related programing and configuration constraints while using this architectural model. By using this model, the memory load/store operations no longer need to be scheduled amongst compute operations. Moreover, now the optimal scheduling of operations does not depend upon memory latencies and therefore does not effect the scheduling of computations. The model's third block consists of the parallel compute units. The architecture – in general – expects the compute units performing logical and arithmetic operations. However, there is no constraint for incorporation of specialized compute units. Each compute unit supposed to keep a register storage and a combination of compute units can share data across a small local memory. The parallel compute units in the back-end of the model communicate with this configurable front-end part through a group of commands, controls and status registers and a large set of index based circular buffers. These index based buffers exploit a programing model supporting indexed based accesses of data. The group of commands, controls and status registers help to synchronize the front-end with the back-end. The scope of this thesis does not cover the details on the programing model. However, as described in Chapter 8, CUDA [13] programing model with extensions can support our target computing architecture.

In this thesis work, initially, on the top of this abstract target computing platform we implemented some application specific accelerator designs. Later, we suggested a template based design methodology to generate and map customized memory accelerators for this target computing architecture. Finally, we proposed a *Blacksmith Computing Architecture (BSArc)* with the underlying concept from the same target computing model.

1.3 Thesis Contributions

The main contributions of this thesis can be categorized into three parts. The first part is based on the proposals for the application specific designs of the accelerators with customized memory layouts. The second part of the work studies the template based generic design support mechanism for the customized memory accelerators. The last part of the contributions explores the design space for the throughput oriented accelerators with standard/customized memory designs.

1.3.1 Design of Application Specific Accelerators with Customized Memory Layouts

—(1)— 3D stencil computations are compute-intensive kernels often appearing in high-performance scientific and engineering applications. The key to efficiency in these memory-bound kernels is full exploitation of data reuse. We proposed a state of the art streaming accelerator for the 3D Stencil kernels. The design of the architecture makes it possible to maximize the reuse of data by handling the input data volume through a specialized 3D memory hierarchy. The 3D-memory keeps busy the back-end compute units to maximum throughput. Our proposal also shows the scalability of the accelerator for various sizes of stencils. This makes it possible to map the design to different sizes of reconfigurable devices or ASICs. This contribution was recognized by the `IEEE Conference on Field-Programmable Technology 2009`.

—(2)— Reverse Time Migration (RTM) is a real-life application with a requirement of huge computations for the seismic imaging of geologically complex subsurface areas. The economic value of the oil reserves that require RTM to be localized is in the order of 10^{13} dollars. But RTM requires vast computational power, which somewhat hindered its practical success. We ported our 3D-stencil streaming accelerator to implement the most time consuming computational kernel *acoustic wave equation (AWE) solver* for the 3D-RTM application on Altix-4700 system. Later, the performance of the application was projected on the HC-1 accelerator by mapping the hardware design of the application for Multi-FPGA implementation. This work appears in the `IEEE Journal Transactions on Parallel and Distributed Systems, January 2011`.

—(3)— The findings of our work on 3D-Stencil and RTM reveal that the usage of specialized data organization is very beneficial from performance point of view for an application. However, it can restrict the generality of the architecture. Therefore, we proposed an idea of specialized but at the same time a common memory layout for various application kernels. The benefit of such a scheme – other than the benefits of architectural specialty along with generality – also gives a possibility of the data-reuse across different application kernels. This contribution was published in the `IEEE` proceedings for the Conference on Field Programmable Logic and Applications 2010.

1.3.2 Template Based Design Support for Customized Memory Accelerators

—(1)— The last proposal on the common memory layout draw a sketch of a multi-level memory hierarchy. It also outlines the general characteristics for the flow of data through the common memory layout. This means that the proposal helps in narrowing down the design space for automated generation of an application specific memory structure. However, the fine details of the data flow for an application and the scalability of the memory design according to a target device and/or the problem size still require an automated mechanism. This lead us to propose a HLL translation tool named *DATE* (Design of Accelerators By Template Expansion System). This tool use a library based approach. It keeps templates for specialized memory structures, compute units and the interconnects to generate a design according to the user given parameters. This work is presented in HiPEAC Workshop on Reconfigurable Computing 2011. An extended version of the same work is accepted for the Elsevier's Journal of System Architecture.

—(2)— We used our last work on the *DATE* tool to propose a Throughput Oriented Template based Streaming Accelerator. In general, the throughput oriented streaming accelerators offer a large number of compute resources but restrict the applications to arrange parallel and maximally independent data sets to feed the compute resources as streams. Therefore, the design specialization for – both – the compute units and the local memory structures could improve the performance efficiency for such devices. This makes the basis for our proposal on an template based architecture design for the reconfigurable accelerators (*TARCAD).* This template accelerator accommodates the application specific compute units and the application specific memory structures with generic types of system level control and the I/O channels under our *DATE* based design generation scheme. This contribution was recognized in the IEEE Symposium On application Specific Processors 2011.

1.3.3 Deign Space Explorations for Throughput Oriented Devices with Standard & Customized Memories

—(1)— The purpose of the throughput oriented *TARCAD* (Template Architecture for Reconfigurable Accelerators) is conceptually very close to the GPU design concept. The main difference is that a *GPU* is not reconfigurable and a *TARCAD* does not have a generic programming model. We came with an idea of combining the interesting features from both architectures to propose potentially a new heterogeneous architecture. This lead us to develop a simulator for GPU kind of streaming architectures and resulted in the form of *SArcs* (Streaming Architectural Simulator). *SArcs* is a trace based simulation tool chain. Its framework uses GPU performance modeling based on run-time CPU code explorations on a streaming simulator which is a part of the designed framework. To the best of our knowledge *SArcs* is the first trace-based GPU architectural simulator which does not require a physical GPU environment or any GPU related tool-chain. This contribution is from our paper accepted in `ACM International Conference on Computing Frontiers; May 15th, 2012`.

—(2)— The *SArcs* framework, on the one hand, is very useful for the design space explorations for the future GPU devices and on the other hand, it can be used for performance evaluation of different applications on the existing GPU generations with a good accuracy. The framework exploits the fact that an application compiled for any architecture would require to transact the same amount of data with the main memory in the absence of registers or cache hierarchy. Moreover, the computations inside an application can be simulated by the target device latencies. We use *SArcs* for the design space explorations of GPU like streaming architectures and show that the configurations of the computational resources for the current Fermi GPU device can deliver higher performance with further improvement in the global memory bandwidth for the same device. This work is a part of the `research report: UPC-DAC-RR-2012-6`.

—(3)— We Proposed a *Blacksmith Streaming Architecture* (BSArc) for high performance accelerators. The *Blacksmith Computing* on *BSArc* uses a forging front-end to efficiently manage data according to the application nature. A large set of simple streaming processor in the back-end can fetch this arranged data to run computations on it. We apply this concept to a SIMT execution model and present it as a part of a modified GPU like device supporting an Application Specific Frond-End. The accuracy of the base line simulator was established against the NVIDIA's Fermi architecture (GPU Tesla C2050) using L2 cache. We evaluate the performance difference for the Blacksmith Computing based architectural approach against the standard L2 cache base configuration of the GPU like device by using our *SArcs* simulator. The performance of Blacksmith Architecture show highly promising results as compared to the newest GPU generation i.e. Fermi. This contribution is recognized by the `ACM International Conference on Computing Frontiers; May 15th, 2012.`

1.4 State of the Art of Streaming Accelerators

The topic of streaming architectures is very vast. A lot of work has been done previously on the streaming accelerators like the one GOPS streaming processor presented by Khailany et al [14]. This processor contains 16-lane data-parallel unit (DPU) with 5 ALUs per lane, two MIPS 4KE CPU cores, and I/Os. This architecture designed to support applications such as video encoding, image filtering, wireless signal processing and scientific computing. The memory interface for the processor includes two 64b DDR1/DDR2 666Mb/s memory channels for 10.7GB/s total. The processor works under a VLIW instruction set. The Stanford project of Merrimac [15] – in comparison to the general purpose cluster based scientific computers – develops a stream-based supercomputer. The focus of the project is to reduce the memory bandwidth requirement from representative applications by organizing the computation into streams and exploiting the resulting locality using a register hierarchy. The Crypto engine in Sun Ultra-SPARC T2 [16] contains a Streams Processing Unit (SPU) offering encryption/decryption and offloading of the hash-operations. This cryptography streaming accelerator can work efficiently on large chunks of data because of an integrated direct memory access (DMA) engine inside SPU. This allows the accelerator to access the L2 cache without having to go through the regular pipeline. Bove and Watlington proposed Cheops [17] which is a media processing system for video streams. Cheops framework uses individual specialized processing units – the stream processors – typically comprised of multiple parallel computing elements. Multiple stream processors acts simultaneously as one processor module. The processor module comprises of eight dual ported dynamic memory (VRAM) units communicating through a full crosspoint switch with up to eight stream processing units. In Cheops system, multiple processor modules may be placed in the backplane of the system to form a huge parallel system. The SYDAMA-II [18] system proposes an architecture based on two main parts. The low level computing part directly map the streaming data flow graphs of image processing applications to one or more stream processing elements. The second part of the architecture consists of general purpose processing to handle algorithms at higher level and as well to run the operating system.

In the following we will describe in more detail some prominent architectures researched in the past or available as commercial products.

1.4.1 Imagine

Imagine [19] is a programmable streaming processor shown in the Figure 1.4-a. This processor handles data and computations in a decoupled manner. The Imagine architecture achieves this decoupling by programming the processor at two levels : kernel level and application level. Kernel code is kept inside the *controller* of the imagine processor shown in the Figure 1.4(a). The kernel code use 48 ALUs organized as 8 SIMD clusters to run computations on the stream elements. These clusters take data from the *Stream Register File* and provide it to the ALUs under the controller's program. The application level program manipulate the streams and pass these between the kernel functions.

Each of the SIMD cluster contains 6 ALUs, large number of local register files and executes completely static VLIW instructions. The memory system, the host interface, network-interface, arithmetic clusters and the controller interact for transferring streams to and from the stream register files. The most important feature of the imagine processor is considered its multi-level high memory bandwidth to efficiently operate 48 ALUs. The maximum bandwidth achieved at the register file level is 435GB/s which is approximately 17$\times$ more than the available bandwidth with the external memory. This register level bandwidth is an ideal one and requires application level software to somehow forward the streams by increasing its reuse 17$\times$ to keep busy all the compute units.

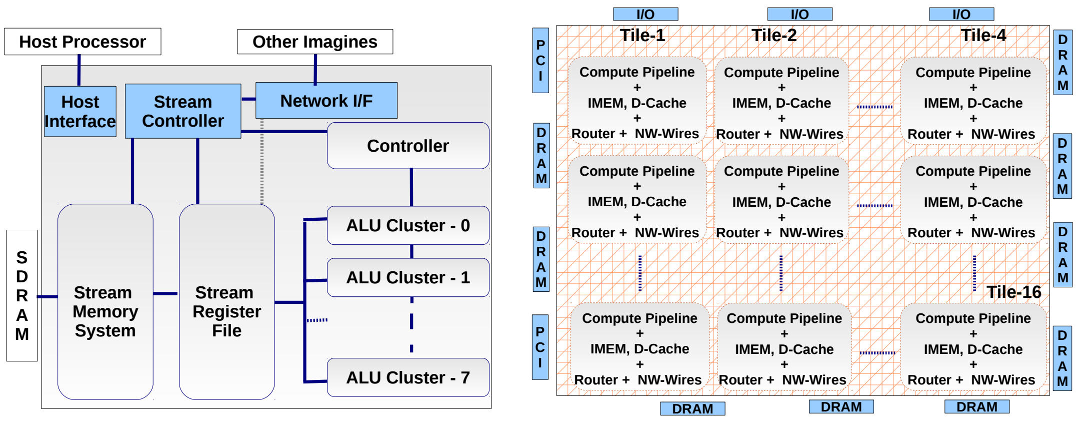


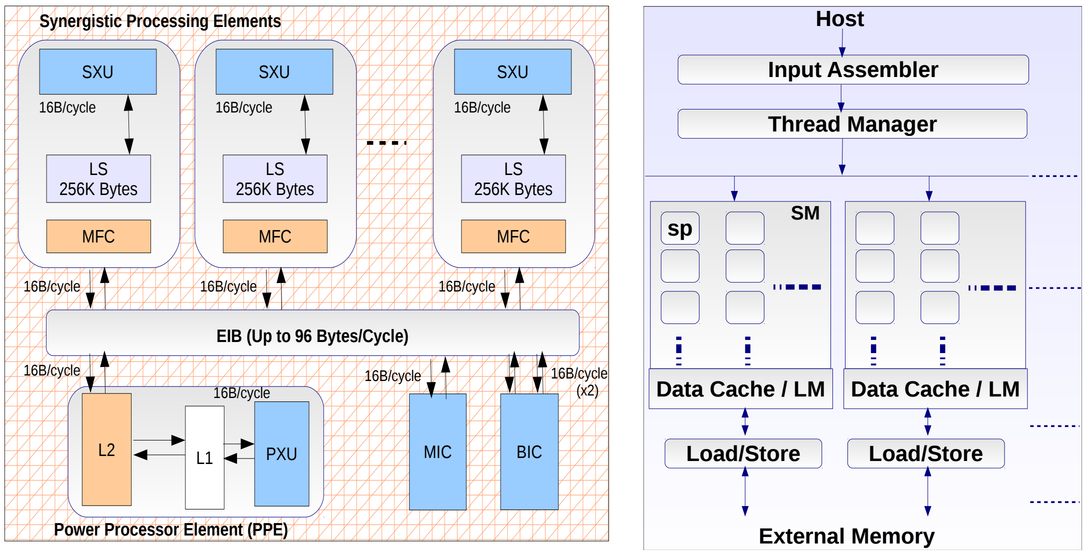
Figure 1.4: Streaming processors: (a) Imagine (b) Raw

1.4.2 Raw

In the design of *RAW* processor [20] (shown in the Figure 1.4-b), the most innovative feature is its on-chip interconnect and its interface with the processing pipeline. The tiled architecture of *RAW* processor connects its 16 processing tiles (Figure 1.4-b) using four 32-bit full duplex on-chip networks. Two of the network routs are specified at compile time (i.e. static) while the other two networks could be specified at run time. These networks are exposed to the software under RAW ISA. RAW uses Raw Stream Compiler to map pipeline parallel code onto the networks. This allows the programmer to directly program the wiring resources of the processor. This means that the programmer can transfer data streams to different combinations of the Tiles according to an application need. Each Tiles can run computations on a stream of data using its 8-stage in-order single-issue MIPS style processing pipeline, a 4 stage single precision pipelined FPU and 32 Kbyts of data cache.

1.4.3 IBM Cell/B.E

The Cell/B.E. [21] (Figure 1.5-a) is an example of a SoC with a general purpose processor and SIMD accelerators. It is a *multi-core chip* composed of a general 64-bit PowerPC processor core (PPE) and 8 SPEs (SIMD processors called Synergistic Processor Elements) that have a small scratch-pad memory called *local store* (LS). A high



(a) (b)

Figure 1.5: (a) IBM Cell/B.E (b) GPU

speed bus (EIB, Element Interconnect Bus) is shared among all components, allowing all of them to directly access main memory through the Memory Interface Controller. There are two types of storage domains within the Cell/BE architecture: the main storage domain and the local storage domain. The local storage of the SPEs exists in the local storage domain. All other kind of memories are in the main storage domain. Each SPE can only execute SIMD instructions (including data load and data store operations) from within its own associated local storage domain. Therefore, any required data transfers to, or from, storage elsewhere in a system is always be performed by issuing a memory DMA command to transfer data between the local storage domain of the individual SPE and the main storage domain. The memory unit for each SPE can typically support multiple DMA transfers at the same time and can maintain and process multiple memory requests.

1.4.4 GPU

GPUs [22] (Figure 1.5-b) adopt a streaming based compute methodology in their architectures. These devices expect from the user to efficiently arrange parallel sets of data for the computations. A single GPU device contain hundreds of simple processing cores. These use multi-threading (SIMT) to keep a high throughput and hide the memory latency by switching between thousands of threads. Generally, the architecture of a GPU device consists of dual level hierarchy. The first level is made-up of vector processors, termed as streaming multiprocessors (SMs) for NVIDIA GPUs and SIMD cores for AMD GPUs. Each of the vector processor contains an array of simple processing cores, called streaming processors (SPs). All processing cores inside one vector processor can communicate through an on-chip user managed memory, termed local memory for the AMD GPUs and shared memory for NVIDIA.

1.5 Thesis Organization

This thesis document consists of eight chapters in total. This – first – chapter gives a general introduction to the work and the last chapter (chapter 8) contains the conclusions and the future work. The second and third chapters of this document covers the application specific design of customized memory for structured grid application and its generalization. These correspond to the contributions mentioned in Section 1.3.1. The details on the template based accelerator designs related to the contributions listed in Section 1.3.2 can be found in chapters 4 and 5 respectively. The chapters 6 and 7 contain information on the streaming architectural simulator and the corresponding design space explorations regarding the contributions listed in the section 1.3.3 of the current chapter.

1.6 Summary

Streaming accelerators are becoming widely dominant for a range of application domains including the scientific, the web and the digital signal processing. These accelerators are also being considered as an interesting choice from the embedded processing to high performance computing. The basic reason of this growing popularity for these architectures is their throughput oriented nature. However, this throughput imposes a requirement on the continuous availability of data for the compute units. This requires to maximally exploit data locality and a way to arrange data in the form of independent parallel data sets. The software based approaches using general purpose caches and local memories are beneficial to some extent. However, the task of efficient data management for the throughput oriented devices could be improved by providing application specific front-end to a streaming architecture.

The current chapter has presented an overall view of this thesis document. In the next chapter, we will start by presenting a detailed study on a customized memory design for the structured grid application domain. The chapter will further show how such design can be used in accelerators for a real life oil and gas exploration application.

Part I

Design of Application Specific Accelerators with Customized Local Memory Layout

2

A Design of Streaming Architecture for Structured Grid Application

Stencil computations are extensively used in structured grid applications. These have wide spread usage in the real life. Reverse Time Migration (RTM) is one of those real life applications that uses stencil computations. In this chapter, we present two studies: i) A generic design of a streaming architecture for 3D-stencil. (ii) Implementation of RTM using application specific design of the 3D-stencil.

The first study explores the design aspects for 3D-stencil implementations that maximize the reuse of all input data on a FPGA architecture. The work focuses on the architectural design of 3D stencils with the form $n \times (n+1) \times n$, where $n = \{2, 4, 6, 8\}$. The performance of the architecture is evaluated using two design approaches, "Multi-volume" and "Single-Volume". When $n = 8$, the designs achieve a sustained throughput of 55.5 GFLOPS in the "Single-Volume" approach and 103 GFLOPS in the "Multi-Volume" design approach in a 100-200MHz multi-rate implementation on a Virtex-4 LX200 FPGA. This corresponds to a stencil data delivery of 1500 bytes/cycle and 2800 bytes/cycle respectively. The implementation is analyzed and compared to two CPU cache approaches and to the statically scheduled local stores on the IBM PowerXCell 8i. The FPGA approaches designed here achieve much higher bandwidth despite the FPGA device being the least recent of the chips considered. These numbers show how

[1] Chapter 2 is based on the publications :
(1) Exploiting Memory Customization in FPGA for 3D Stencil Computations;
Muhammad Shafiq, Miquel Pericas, Raul de la Cruz, Mauricio Araya-Polo, Nacho Navarro and Eduard Ayguade appeared in IEEE ICFPT December 2009, Sydney, Australia
(2) Assessing Accelerator based HPC Reverse Time Migration; Mauricio Araya Polo, Javier Cabezas, Mauricio Hanzich, Felix Rubio, Enric Morancho, Isaac Gelado, Muhammad Shafiq, Miquel Pericas, Jose Maria Cela, Eduard Ayguade, Mateo Valero appeared in IEEE Journal TPDS, Special Issue January 2011

a custom memory organization can provide large data throughput when implementing 3D stencil kernels.

The second study shows the mapping of RTM on the reconfigurable device. RTM is a proven most advanced seismic imaging technique for making crucial decisions on drilling investments. However, RTM requires vast computational power, which somewhat hinders its practical success. Our mapping of RTM as an application specific design uses $8 \times 9 \times 8$ specific 3D-stencil memory design. The performance of the kernel is projected for HC-1 Convey machine. We compare the performance of RTM algorithm on FPGA system against the implementations for Intel Harpertown, the IBM Cell/B.E. and NVIDIA Tesla. All streaming accelerator based implementations outperform the traditional processor (Intel Harpertown) in terms of performance (10x), but at the cost of huge development effort. GPU remains the best between the accelerator based implementations. These results show that streaming accelerators are well positioned platforms for these kind of workloads.

2.1 High Performance Computing for Structured Grids

The necessity for High Performance Computing (HPC) will keep increasing as there is always a problem that needs more computational power than currently available. However, the last years technological issues have put an end to frequency scaling, and hence to traditional single-processor architectures. Thus, processors designers and application developers have turned to multi-core architectures and accelerators in the search for performance. During this quest, one of the possible solution found for the new HPC generation hardware is to use reconfigurable logic device (e.g. Xilinx FPGAs). These devices use a design approach based on configurable hardware [23]. Inside an FPGA, the hardware logical layout is configured before doing the computation, usually by generating a custom computation unit and replicating it as many times as possible. This allows FPGAs to achieve higher performance even while running at frequencies far below ISA processors or accelerators. However, this performance does not come for free: the development cost increases. As these architectures are all different from traditional homogeneous processors, they have their own particularities. Considerable effort must be invested to adapt the algorithm to the architectural features.

Reverse Time Migration (RTM) [24] is the structured grid application that we consider as a case study for the application specific design in this chapter. RTM implements an algorithm based on the calculation of a wave equation through a volume representing the earth subsurface. RTM's major strength is the capability of showing the bottom of salt bodies at several kilometers ($\sim$6 km) beneath the earth surface. In order to understand the economical impact of RTM we just have to review the USA Mineral Management Service (MMS) reports[25]. The oil reserves of the Mexican Gulf under the salt layer are approximately 5×10^{10} barrels. Moreover the reserves in both Atlantic coasts, Africa and South America, are also under a similar salt structure. A conservative estimation of the value of all these reserves is in the order of 10^{13} dollars. RTM is the key application to localize these reserves. RTM is the method that produces the best subsurface images, however its computational cost (at least one order of magnitude higher than others) hinders its adoption in daily industry work.

In the complete algorithm of RTM, the most time consuming and data intensive kernel implements the stencil computations. Stencils use nearest neighbor computations. These algorithms are frequently found in scientific, engineering and digital signal processing applications. Due to their importance, these applications have been studied in great detail. Single-dimension nearest neighbor computations are best approached using streaming techniques. The input data is temporarily stored in a FIFO buffer and the output is computed using the data available in the FIFO. 1D stencils are basically single-dimension FIR filters. When stencils operate on more than one dimension, the streaming approach is not directly applicable. In this case a combination of domain decomposition and streaming is a better way to process the input data. Many applications involve multidimensional nearest neighbor calculations: 2D stencils are common in image processing applications and 3D stencil computations appear, among others, in seismic imaging and in computational electrodynamics (FDTD). As the number of dimensions increases, not only the number of input points per output point increases but also memory accesses become more sparse. For this reason stencil computations easily get memory bound and hardware caches are less efficient. The key to alleviating these problems is to maximize the reuse of input points occurring when computing adjacent points. For example, a 3D stencil computation operating on $\{x, y, z\}$ input points will use each point up to $x \times y \times z$ times. Keeping these points in fast and specialized local memories can considerably reduce the required external bandwidth.

Attempts to implement 2D and 3D stencils in hardware have been presented in literature. Durbano et al. [26] were among the first to propose a FPGA implementation of 3D stencils as part of a complete accelerator system for FDTD (Finite Difference, Time Domain) simulation. A more detailed description of a 3D stencil FPGA implementation was presented by He et al. [27]. In this work the authors propose to exploit data reuse by streaming the planes through multiple FIFOs. The main limitation of these works is that they focus on so-called single-point stencils (i.e. $2 \times 3 \times 2$ stencil). Despite having been extensively studied, most real-life applications require higher order stencils.

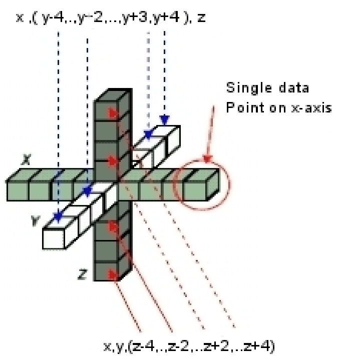
In the first part of the chapter, we study a generic implementation for symmetric stencils of type $n \times (n + 1) \times n$ where $n = \{2, 4, 6, 8, ..\}$. The contribution of our work is twofold. First, we evaluate the impact on performance for various mappings of a decomposed volume to the Virtex-4 FPGA's fine-grained distributed and block memory system [28] and, second, we evaluate the performance in terms of internal data bandwidth achieved by our proposed 3D memory architecture in comparison to various conventional memory organizations, including the Itanium2 cache subsystem [29], the PPC970MP's subsystem [30] and the CellBE's scratchpad memories (the *local stores*) [31] implemented by *Raul de la Cruz* and *Mauricio Araya-Polo* [32]. Exploiting data layout customization in FPGA we find that a distributed three-level data cache implementation can considerably increase the amount of data processed per cycle.

2.2 3D Stencil Computations

Stencils are used in numerous scientific applications like, computational fluid dynamics, geometric modeling, electromagnetic, diffusion and image processing. These applications are often implemented using iterative finite-difference techniques that sweep over a 2D or 3D grid, while performing computations called *stencil* on the nearest neighbors of the current point in the grid. In a stencil operation, each point in a multi-dimensional grid is updated with weighted contributions from a subset of its neighbors in both time and space. In a 3D stencil, each point's computation needs to access data from the three axis of a volume as shown in the Figures 2.1(a) & (b) for 3D stencil

```
int iter_j = Size_Z-axis;
int iter_k = Size_X-axis  *  Size_Z-axis;

for (k=4;k<Size_Y-axis -4;k++)        //  Y-axis
  for (j=4;j<Size_X-axis -4;j++)      //  X-axis
    for (i=4;i<Size_Z-axis -4;i++)    //  Z-axis
    {

    iter   = k * iter_k + j * iter_j + i ;

    // STENCIL ON PLANES  (Y-AXIS)
    Cpoint  =
            C[0] * (p[i+j*iter_j+(k-4)*iter_k] + p[i+j*iter_j+(k-3)*iter_k]) +
            C[1] * (p[i+j*iter_j+(k-2)*iter_k] + p[i+j*iter_j+(k-1)*iter_k]) +
            C[2] * (p[i+j*iter_j+(k+1)*iter_k] + p[i+j*iter_j+(k+2)*iter_k]) +
            C[3] * (p[i+j*iter_j+(k+3)*iter_k] + p[i+j*iter_j+(k+4)*iter_k]);

    // STENCIL ON COLUMNS (X-AXIS)
    Cpoint = Cpoint +
            C[4] * (p[i+(j-4)*iter_j+k*iter_k] + p[i+(j-3)*iter_j+k*iter_k]) +
            C[5] * (p[i+(j-2)*iter_j+k*iter_k] + p[i+(j-1)*iter_j+k*iter_k]) +
            C[6] * (p[i+(j+1)*iter_j+k*iter_k] + p[i+(j+2)*iter_j+k*iter_k]) +
            C[7] * (p[i+(j+3)*iter_j+k*iter_k] + p[i+(j+4)*iter_j+k*iter_k]);

    //  STENCIL ON  POINTS  (Z-AXIS)
    Cpoint =  Cpoint +
            C[8] * (p[(i-4)+j*iter_j+k*iter_k] + p[(i-3)+j*iter_j+k*iter_k]) +
            C[9] * (p[(i-2)+j*iter_j+k*iter_k] + p[(i-1)+j*iter_j+k*iter_k]) +
            C[10]* (p[(i+1)+j*iter_j+k*iter_k] + p[(i+2)+j*iter_j+k*iter_k]) +
            C[11]* (p[(i+3)+j*iter_j+k*iter_k] + p[(i+4)+j*iter_j+k*iter_k]);

    Volume_out [iter] =   Cpoint + C[12] * p[iter] ;
    }
```

(a)


x ,(y-4,..,y-2,...,y+3,y+4), z
Single data
Point on x-axis
X
Y
Z
x,y,(z-4,..,z-2,..z+2,..z+4)

(b)

Figure 2.1: Odd-symmetric 3D stencil for n=8 ($8 \times 9 \times 8$ stencil) : (a) 3D- stencil algorithm where *p[..]* represents input volume and *C[..]* are the constants), (b) Points access pattern form 3-dimensions

algorithm and 3D accesses respectively. The three `for` loops in the algorithm correspond to the accesses from the three dimensions of the input volume `p[]`. The constant weights could be identified as `C[]` in the Figure 2.1(a). The algorithm and the stencil access pattern show that 3D stencil computation increases the complexity not only by increasing 3 times the number of computations but also due to the sparse data access pattern arising from a volume linearly laid out in memory.

This work focuses on two design approaches for 3D-stencil computation cores to compute stencils with dimensions $\{n \times (n+1) \times n\}$ where n: 2, 4, 6 and 8. Our first approach is based on a “Multi-Volume” design which intends to use maximum possible number of modules for the same stencil although by compromising the base volume size. The base volume is the one without extension for the boundary points in contrast to the extended volume dimensions as shown in Figure 2.2. Various base

Stencil	A:$2 \times 3 \times 2$	B:$4 \times 5 \times 4$	C:$6 \times 7 \times 6$	D:$8 \times 9 \times 8$
1	32×256	32×256	16×256	16×256
2	16×512	16×512	8×512	NA
3	8×1024	8×1024	NA	NA
4	4×2048	NA	NA	NA
BRAM BLOCKS	276	276	256	320

Table 2.1: Dimensions ($Xdim \times Zdim$ whereas Ydim=∞) for various volume decompositions and their BlockRAM consumption in the **Multi-Volume Design** Approach. Total Data-Engines in Front-End = 3 (for n=2), 2(for n=4,6,8).

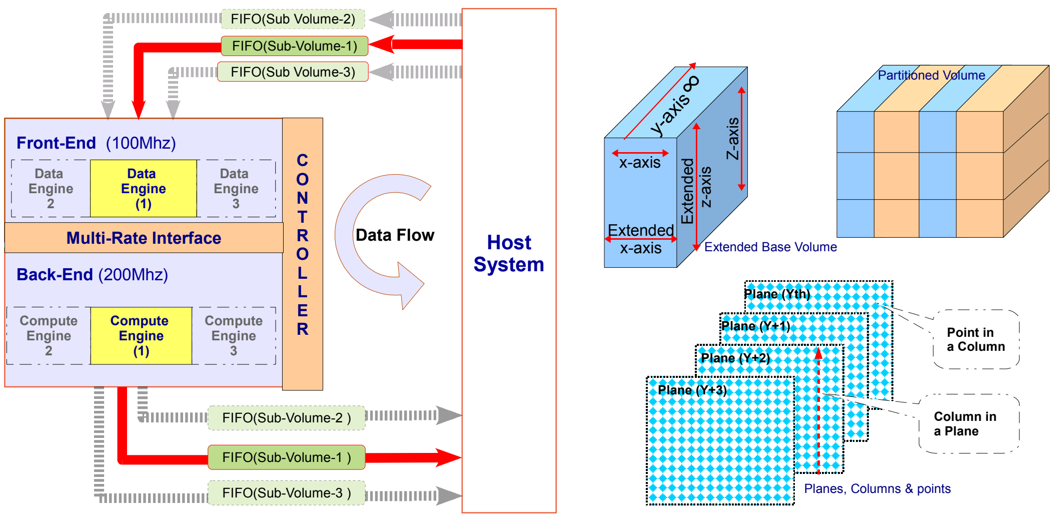
volume sizes, as shown in Table 2.1, have been used for performance evaluations. The base volume sizes used in the “Single-Volume” approach are shown in Table 2.2. Out of the 336 18Kb BRAM blocks present in the Virtex4-LX200, 320 blocks were reserved for storing volume data. The rest of BRAM blocks were to meet the internal requirement (e.g. FIFOs) of the design.

In both cases the base volume sizes (*Xdim, Zdim and Ydim*) as shown in the corresponding tables represent the $\{x, z\}$ dimensions while the third dimension $\{y\}$ is streamed into the core for any number of planes ($n + 1 \to \infty$). In order to evaluate performance overheads for different decompositions in these cases, we have used a main input volume with dimensions $4096 \times 2048 \times 8192$. This volume has been selected to keep it evenly decomposable for most of the cases in Tables 2.1 and 2.2. On the other hand, in order to compare the performance with other implementations (Itanium2, PowerPC970 and Cell/B.E.), we have used stencil for n=8 and the input volume is taken from an implementation of a reverse-time migration kernel [33] which makes intensive use of the same stencil ($8 \times 9 \times 8$) to solve the partial differential wave equation.

The input volume normally needs to be extended at its boundaries to compute the stencil on all points lying on the boundaries of input volume. In order to accommodate this boundary condition for the input volume, our core architectures accepts base volumes, extended by $n/2$ points in each dimension (extended base volume). Figure 2.2(b) shows input volume decomposition. Our proposed 3D-Stencil architecture can handle any size of large volume decomposed into the “extended base volumes”. In a decomposed volume, consecutive sub-volumes are required to be overlapped for $n/2$ points in two dimensions. This decomposition of large volume into sub-volumes

Stencil	E:2 × 3 × 2	F:4 × 5 × 4	G:6 × 7 × 6	H:8 × 9 × 8
1	160 × 256	96 × 256	64 × 256	48 × 256
2	40 × 1024	32 × 768	32 × 512	24 × 512
3	20 × 2048	16 × 1536	16 × 1024	16 × 768
4	10 × 4096	6 × 4096	8 × 2048	12 × 1024
BRAM BLOCKS	320	306	296	300

Table 2.2: Dimensions ($Xdim \times Zdim$ whereas Ydim=∞) for various volume decompositions and their BlockRAM consumption in the **Single-Volume Design** Approach. Total Data-Engines in Front-End : 1



(a) (b)

Figure 2.2: (a) 3D-stencil core and interfaces, (b) Extended base volume, partitioned volume and axis conventions

of extended base volumes can be processed sequentially with same processing unit or it can be distributed over a set of processing elements for parallel computation. For a clearer picture, the convention used to understand the axis of the volume is shown in Figure 2.2(b) which mentions: **Y-axis** (planes in the volume), **X-axis** (columns in a Plane), and **Z-axis** (points in a column).

It is apparent from the Figures 2.1(b) that the 3D stencil computations need $3 \times n$ operands (**n** operands from each axis) before it can fully compute one point of the output volume. In addition to these $3 \times n$ operands, an operand corresponding to the central point is also required. Since our 3D-stencil core is designed to compute single precision floating point data or 32 bit integer operands, the core needs $(3 \times n + 1) \times 4$ bytes for each computed point. Besides this large data requirement, it needs $3 \times n/2 + 1$ multiplications and $3 \times n$ addition operations for computing one output point.

Since stencil computations sweep on consecutive neighboring points in all directions, with a specific arrangement an extensive data reuse is possible. In our case, an exclusively accessible arrangement of data for minimum $n + 1$ consecutive planes corresponding to Y-axis makes it feasible to get all operands needed from Y-axis and as well for other two axis. Thus, an architecture with specialized data layout can be designed which makes available all required operands from X-axis and Z-axis by utilizing previously fetched data from Y-axis to compute a single point. Such architecture can sustain the availability of all input operands to compute a point just by fetching only one new operand from Y-axis rather than fetching $\{3 \times n + 1\}$ operands from three axis. This means that the architecture for specialized data layout would give a $(3 \times n + 1)$-fold increase in data bandwidth at the input of Back-End compared to the input bandwidth of the data Front End. For example, a fetch cycle with M new operands can ideally increase the data bandwidth up to $((3 \times n + 1) \times M)$ times at the input of the Back-End for the stencil computation. However, the practically achievable external–to–core bandwidth normally remains one of the major factors that limit the scaling of the core architecture.

2.3 3D Memory Organizations for Stencil Computations in Reconfigurable Logic

In this section we present a generalized data engine for algorithms based on 3D stencils. The 3D-Stencil core designs are based on three main modules: the Front-End, the Back-End and the Controller. The architecture of every module can be scaled according to the available external–to–core data bandwidth and on-chip resources of FPGA device. Our configuration of the core is based on Virtex-4 FPGA device XC4VLX200-10. The FPGA is present in an SGI RASC RC100 board part of an SGI Altix 4700 system. In this configuration the FPGA can achieve a maximum external data bandwidth of 3.2GBytes/Sec/Direction when using streaming DMA. A simplified view of the core environment is shown in Figure 2.2(a). Next we will elaborate on the structure and working principles of each module in the 3D-Stencil core.

2.3.1 Front-End (Data-Engine)

The Role of Front-End in 3D stencil core can be seen as a specialized data cache backing an arithmetic logic unit. The Front-End can consist of multiple Data Engines ("Multi-volume Design" approach) or of a single Data Engine ("Single-Volume" approach). Therefore, the Data Engine is a basic building block of the Front-End. It consists of multiple sub-modules. These sub-modules include an external memory interface (Streaming DMA in our case), three levels of internal memory hierarchy, circular write control and circular read control for each level of memory. Along with management of three memory levels, the architecture offers independent read and write ports at each level. This capability is achieved by using dual ported block RAMs. In other words, the Data-Engine, besides streaming interface, consists of a specialized 3D memory layout and 3D write and read control corresponding to the three dimensions of the input volume. The Data-Engine's internal structure – consisting of three memory layers (Y-layer, X-Layer and Z-Layer) – is shown in Figures 2.3(a) and (b). The architecture shown in the Figure 2.3(b) is a subset of the architecture shown in Figure 2.3(a). This subset implements a $8 \times 9 \times 8$ specific simplified memory layout for 3D-stencil. The three memory layers (Y-layer, X-Layer and Z-Layer) implements `n+1` memory structure where *n* belongs to $n \times (n+1) \times n$, for $n = 8$. This specific example (Figure 2.3-b) uses one-side write and other side read at each level of dual ported memory blocks. The $8 \times 9 \times 8$ specific special purpose data engine for the 3D stencil computation is used in our RTM mapping as discussed in the Section 2.4.2.

To exactly understand the functionality of the generic design of Data-Engine (Figures 2.3(a)), it is important to correctly understand the pattern of data required to compute output points. In Figure 2.3(a), we can see exactly next to the "STREAM Read Controller", the first layer of memory, representing *Planes* corresponding to the Y-axis of the volume (therefore named *Y-layer*). This first layer in the Data-Engine's memory hierarchy consists of $256bit \times Xdim$ sized $n+2$ dual ported block RAMs. Here the value "Xdim" corresponds to the dimensions (Zdim, Xdim, Ydim) given in Tables 2.1 and 2.2. The architecture keeps one extra plane (`n+2` structure) thus the total number

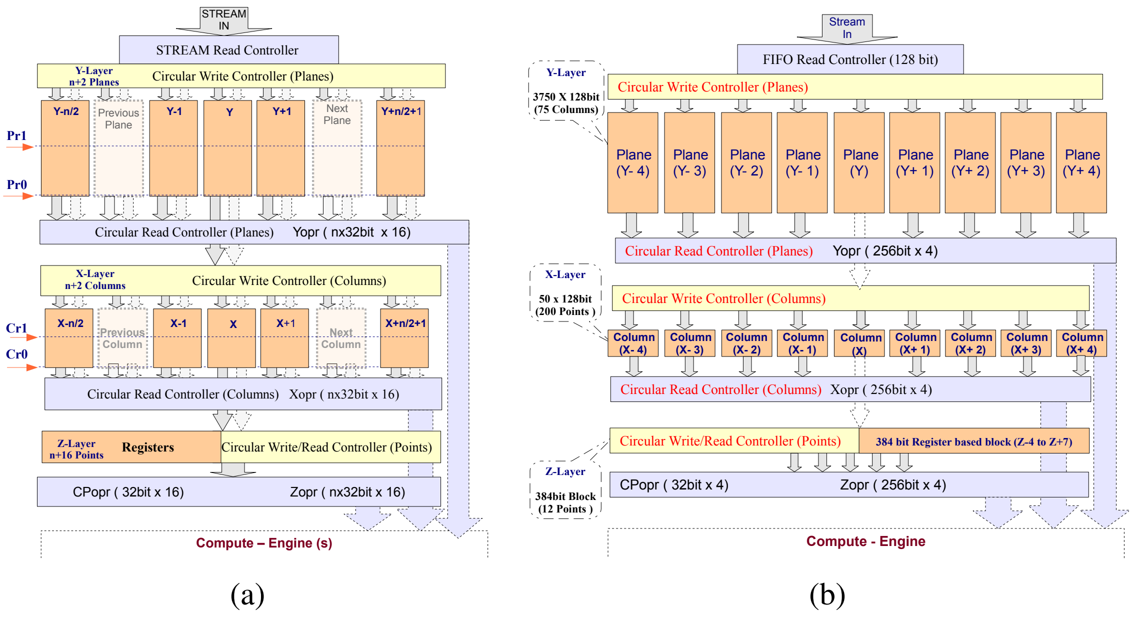


Figure 2.3: (a) Generic architecture of Data-Engine : Implementing three memory levels (Y-Layer, X-Layer, Z-Layer). “Pr0/Cr0” and “Pr1/Cr1“ are the plane/Column read pointers for lower and upper halves respectively at Y-Layer and X-Layer, (b) $8 \times 9 \times 8$ specific simplified (one-side write and other side read at each level) layout of the special purpose data engine for the 3D stencil computation used in RTM mapping.

of planes managed is *n+2*. This additional *(n+2)th* plane is used for exclusively writing the data using both of its ports and at the same time it is possible to read all the other *n+1* planes from their two ports. This means that at any time one plane would be working in a dual write address mode for both of its ports and all other planes would be in a dual read address mode at the same time. However, the $8 \times 9 \times 8$ specific architecture shown in the Figure 2.3(b) uses only `n+1` planes. Therefore, in this simple case all planes (also columns and points) are read from one side of dual ported memory blocks in the corresponding layer and the other side of the memory blocks are fixed for writing. Our architecture logically splits each plane in two halves as shown in Figure 2.3(a). The two address pointers *Pr0* (plane read pointer for lower half) and *Pr1* (plane read pointer for upper half). This technique effectively doubles the throughput of the Data-Engine at the cost of maintaining one extra plane inside FPGA using few more BRAMs.

All planes in Y-Layer are sequentially writable at the time in turn when a *plane's* status becomes $n + 2^{th}$. Other *n+1* planes are possible to read in parallel. This means that one write to Y-layer is of $256bit \times 2$ where factor 2 corresponds to dual port write. A read from this layer is possible for $256bit \times 2 \times (n+1)$ where factor 2 is the dual port read. The read side inherits a minimum latency of one clock cycle. The second layer of memory is labled as *Column* and corresponds to X-axis of input volume (named *X-Layer*) This layer has exactly the same features as that of Y-layer except that its size is $256bit \times Zdim \times n + 2$ where Zdim corresponds to the dimensions given in Tables 2.1 and 2.2. Both X-layer and Y-layer memories are created by using internal Block RAMs of the FPGA. The third memory layer corresponds to Z-axis (Z-layer) and it is based on FPGA registers. Its total size is *1536bits*.

2.3.2 Working Principle of the Data-Engine

At host interface, input and output volume(s) are streamed into FIFOs of the 3D-Stencil core as shown in the Figure 2.2(a). The number of streaming channels used in the design varies according to the "Multi-Volume or Single-Volume" approach used. The SGI Altix 4700 provides four input and four output DMA streaming channels which are enough to support both design approaches. In order to synchronize the operation of the Data-Engine(s) with stalls in the input/output stream, each FIFO maintains an upper bound and lower bound to activate and stall all the data management sequences of the Engine. The Compute-Engines in Back-End, however, always continues working if any data is available in its pipelines.

As soon as a FIFO crosses the upper-bound limit, the corresponding Data-Engine starts working. The Stream Interface Controller prefetches the data of extended base volume from the external memory into the Y-layer. As soon as last plane (*Y+ n/2+1*) in Y-layer starts to fill, an overlapped prefetch operation is started jointly by the circular read controller (Planes) and circular write controller (Column) to fill the X-layer from the Yth plane. The prefetch sequence ends by the circular write controller (Points) after filling the Z-layer by fetching data from the Xth column. The prefetch operation is instantly followed by simultaneous reading of X, Y, Z-**opr**ands as shown in Figure 2.3(a) and by forwarding these operands to the Back-End through a multirate interface (Figure 2.2).

The prefetch operation is required at the start of every extended base volume. After prefetch phase, computations are overlapped with data fetched from the FIFO's to Y-layer, Yth plane to X-layer and Xth column to Z-layer. If FIFO(s) touches a lower bound, Data-Engine is stalled until upper bound is not reached. The writing and reading for each layer continues in a round circle at its both ends. The circular write is quite simple. For example, when finishing writing to *Y+n/2+1* plane the two writing pointers for lower and upper half of the plane are taken to the *Y-n/2* plane. On write completion to *Y-n/2* the pointers are taken to *Y-n/2+1* and so on. Same separate processes of writing are valid for the other two layers. On the reading side, these layers are accessed such that all planes, all columns and all points corresponding to the three layers are read simultaneously in the same clock cycle. As soon as any layer is read to its top, all pointers to its units (plane or column) are shifted by one. For example in the case of the X-layer, the two read pointers for the *X+n/2* column will become *X+n/2-1* and the pointers for the *X-n/2* will become *X+n/2+1* and so on. It is same for the Y-layer but a little different in Z-layer, where data is shifted rather than the pointers.

The prefetching phase ends while starting a write to the last plane (*Y+n/2+1*). After this prefetch phase, writing to a plane of layer is automatically followed by the read cycles because write operations, which now are overlapping computations, are still continued to fill up to the last plane. This phase difference is important for correct data read otherwise an over-write of data is possible after an arbitrary time even due to a minor mismatch in read and write rate. This difference in write and read rate is possible because of some regular stalls on the read side. These stalls occur at plane and column boundaries. Data from central (*Yth*) plane is forwarded to X-layer but as soon as this plane is finished sending data to X-Layer, the plane pointer is shifted by one, ie. *Y+1* plane is now Yth plane therefore it is necessary that all data present in X-layer must be fetched from the new plane. The same case is true for the Xth column to the Z-layer. Therefore a latency of $n \times Zdim/(256 \times 2)$ Cycles occurs after each shift of read pointers in the Y-layer and a latency of *n/2* cycles occurs at each shift of read pointers to the X-layer.

Data-Engine utilizes between 75% to 95% of the Block RAMs and from 14% 63% slices on Virtex-4 Lx200 device that depends upon the design approach used, stencil size and selected decomposition dimensions.

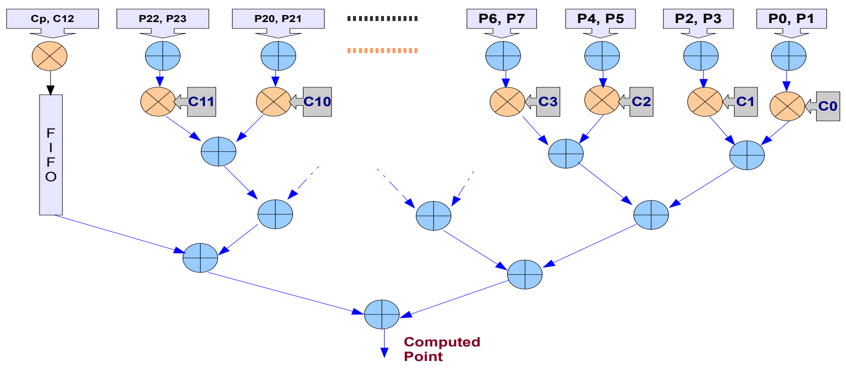


Figure 2.4: Architecture of Compute-Engine in the Back-End for n=8 stencil type. C(0) to C(12) are the constant coefficient multipliers of the stencil. P0 to p23 and Cp are the input data operands coming from Front-Engine. *Cp* is the central point of an odd symmetric stencil

2.3.3 Back-End (Compute-Engine)

This module of the 3D-stencil core also scales based on the design approach (Multi-volume or Single-Volume) used for the Front-End. Therefore the Back-End is normally based on multiple instantiations of a Compute-Engine. Each Compute-Engine outputs values at the rate of 1 result/cycle

The Compute-Engine works at 200MHz (Figure 2.2) which is twice the operating frequency of Data-Engine. As shown in Figure 2.3(a), the Data-Engine can arrange operands (Yopr, Xopr, Zopr, CP) for computing 16 points in parallel. These operands are forwarded to the Compute-Engines in the Back-End. A multirate data interface takes care of transacting the data (operands and results) between the Front-End and Back-End in a correct way. The computation requirement of 16 points per cycle, posed by the Data-Engine needs a Back-End with 8 Compute-Engines working in parallel at twice the rate of the Front-End.

Each of the Compute-Engine in Back-End takes $3 \times n + 1$ variable operands mentioned as “Pn and Cp” corresponding to “Yopr, Xopr, Zopr and CPopr”. It also takes a number of constants coefficients mentioned as “Cn”. These constants are fed through some of I/O registers directly writeable by a Host, outside of the FPGA. Each Compute-Engine implements a binary tree for computing output as shown in Figure 2.4. A small FIFO is also implemented to accommodate the latency for the odd operand before it is added in the binary tree. A summary of sustained performance for the FPGA implementation of the 3D memory organization with $8 \times 9 \times 8$ 3D stencil is shown in table 2.5.

2.3.4 Control-Engine

This module is responsible for synchronizing the data flow in the whole architecture. It takes care of synchronizing the Host Interface, the Data-Engine and the Compute-Engines. As mentioned in subsection 2.3.1, the 3D data layout is forced to stall at certain positions during the execution, therefore the control engine is responsible for managing the effects of these stalls. The Control-Engine, in fact, integrates all stalls to a single major stall at plane boundaries so that Stream Interface Controller can transfer data in larger chunks to the input FIFO(s).

2.3.5 Evaluations

In our evaluations for FPGA based 3D-stencil kernel, we compare with the IBM PowerXCell 8i and two homogeneous Processors: Itanium2 and PowerPC970.

The two main problems for the implementation of stencils on homogeneous processors are the access pattern and the low computation/access ratio. Only in the one direction of the 3D-stencil, the points are consecutive in memory. Therefore, the accesses to memory for the other directions are very expensive in terms of L2/L3 cache misses. This forces us to be careful with the way the data is placed and accessed for these systems with regular caches. One of the main approaches when trying to diminish the memory access cost is the idea of *blocking* [34; 35]. The goal of this technique is to fill the cache levels in order to maximize the locality of the data being accessed, hence diminishing the necessity of accessing slower memory levels for getting the data. In practical terms, the blocking technique divides the dataset in blocks that fit the memory hierarchies. The evaluated processors, their cache hierarchies and other specifications can be obtained from table 2.3.

The Cell/B.E. is an example of a SoC with a general purpose processor (PowerPC) and SIMD accelerators. It is a *multi-core chip* composed of a general 64-bit PowerPC Architecture processor core (PPE) and 8 SPEs (SIMD processors) that have a small scratch-pad memory called *local store* (LS). A high speed bus (EIB, Element

	PowerPC970MP	Itanium2 *Montecito*	PowerXCell 8i	Virtex4-LX200
Num. cores	2	2	8	1
Frequency (GHz)	2.3	1.6	3.2	<500MHz
Peak (Single Precision GFlop/s)	36.8	25.6	204.8	NA
L2-D p/core (KB)	1024	256	512 (PPE) 256 (SPE)	756 KB (Block RAM)
L3-D (MB)	NA	8	NA	NA
Max.Power (Watts)	130	80	157	115
Year of Introduction	2005	2006	2008	2005

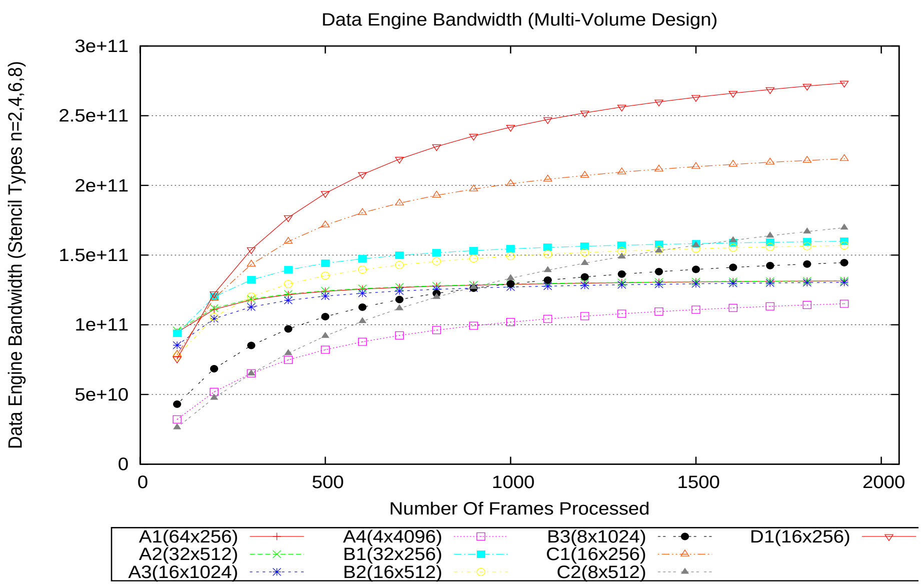
Table 2.3: Processor technical specs. Peak GFlops are obtained considering SIMD extensions

Interconnect Bus) is shared among all components, allowing all of them to directly access main memory through the Memory Interface Controller (MIC). Due to the size of the 3D data to be processed it is necessary to split the data for parallel processing. This splitting (or blocking) has to respect LS size and optimize the bandwidth. In the Cell/B.E. based implementations, the data space is divided and scattered among the SPEs. The 3D space is blocked in X direction, then each sub-block given to one SPE to be processed. Y direction is again traversed by a streaming of ZxX planes. In this architecture the memory management is programmer duty. In order to achieve efficient scheduling of data transfers to/from the main memory and the LS, we use *double-buffering* technique, as explained in [36], thus we almost completely overlap computation time and memory transfer time.

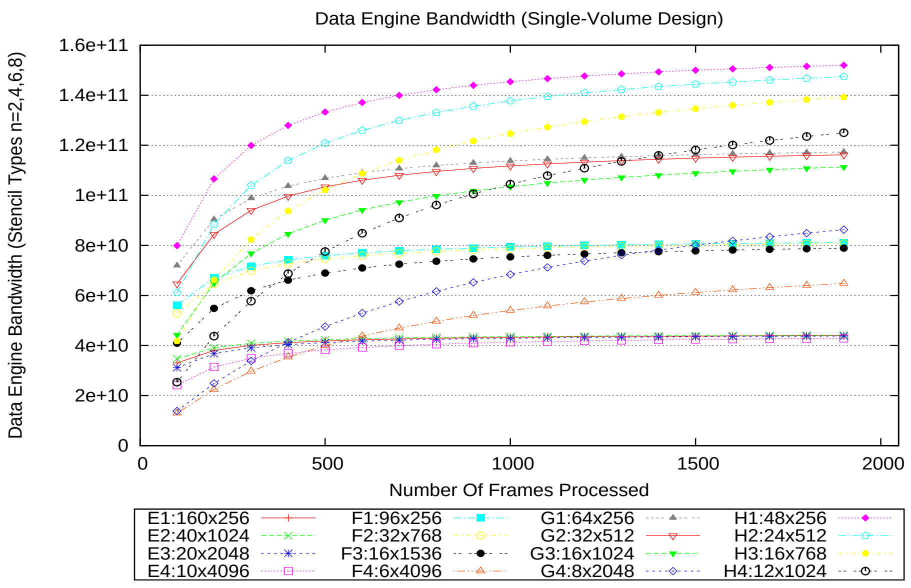
The estimates for the maximum power required at chip level for the target architectures are also shown in the table 2.3. These power estimates are taken from the power specifications for the corresponding chip (IBM PowerXCell 8i, PowerPC970 and Itanium2) boards integrated into IBM QS-22, IBM JS-21 and Altix-4700 machines respectively. The FPGA power ratings are taken from the specifications of the boards for RC100 [37] and the host in the Altix-4700 system.

2.3.6 Results and Discussion

Figure 2.5 shows the internal data throughput of the different stencil implementations as a function of the number of frames (i.e. z, x planes). As can be seen from this figure, a small number of frames has a large impact on the performance. This is mostly because of the higher data latencies as compared to the stencil execution time in these

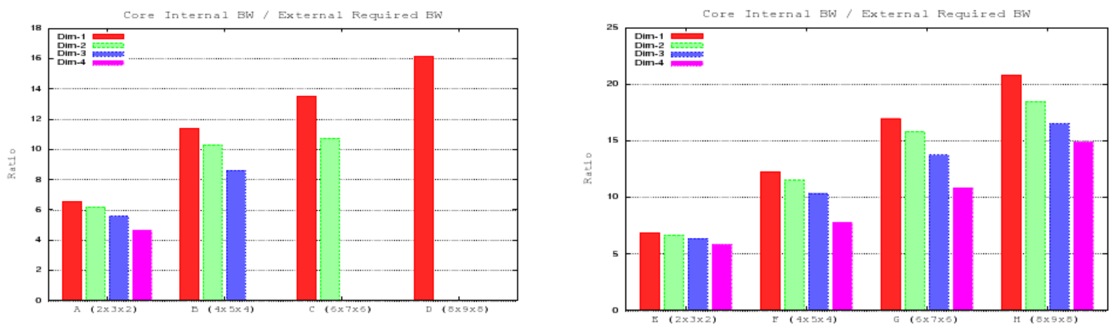


(a) 3D-Stencil Core Internal Data Bandwidth – (Bytes/Second) – (Multi-Volume Design)



(b) 3D-Stencil Core Internal Data Bandwidth – (Bytes/Second) – (Single-Volume Design)

Figure 2.5: Internal Bandwidths (Bytes/Second) achieved for Multi-Volume and Single-Volume design approaches



(a) Ratio: Internal BW / External BW (Multi-Volume Design)

(b) Ratio: Internal BW / External BW (Single-Volume Design)

Figure 2.6: Ratios between Internal and external Bandwidths for Multi-Volume and Single-Volume design approaches

cases. Among the “Multi-Volume Design” implementations, D1 achieves the highest throughput at 280 GBytes/s. Among the ”Single-Volume” designs the best configuration is H4, which obtains 150 GBytes / second. The Figure 2.6 shows the ratios between internal and external bandwidths. In this figure, higher bars indicate better usage of external bandwidth (i.e., less overhead).

Table 2.4 presents an evaluation of the efficiency (throughput per slice) of the different stencil approaches. We compare the Multi-Volume approach with the Single-Volume design. The numbers are for the best performing volume decomposition. As expected Single-Volume designs are somewhat more efficient than the Multi-Volume designs (6%-14%). This difference increases with higher order stencils.

Throughput data for the different processors has been collected in Table 2.5. The table makes it evident that a big gap exists between dynamic cache hierarchies and statically scheduled accelerators. There are about two orders of magnitude difference between conventional processors and the considered accelerators. For the case of the $8 \times 9 \times 8$ stencil, the impressive internal bandwidth of the FPGA (2783 bytes per cycle) allows it even to outperform PowerXCell 8i processor despite the fact that processor is clocked more then 30 times faster (3.2GHz vs 100MHz). Moreover, the customized implementation of 3D-Stencil achieves the best green ratio (GFlops/watts) as compared to all other best implementations based on blocking technique. Also interesting is the fact that the Virtex4 LX200 is actually the oldest of all hardware analyzed. We

	Multi Volume (3, 2, 2, 2)				Single-Volume (1, 1, 1, 1)			
	A:$2 \times 3 \times 2$	B:$4 \times 5 \times 4$	C:$6 \times 7 \times 6$	D:$8 \times 9 \times 8$	E:$2 \times 3 \times 2$	F:$4 \times 5 \times 4$	G:$6 \times 7 \times 6$	H:$8 \times 9 \times 8$
V4LX200 Slices (%)	40569 (46%)	39666 (45%)	52310 (59%)	64922 (73%)	12632 (14%)	18942 (21%)	25264 (28%)	31570 (35%)
Throughput per Slice $MB/slice$	3.2410	4.031	4.1889	4.2119	3.4934	4.2835	4.6444	4.8142

Table 2.4: Slice counts and throughput per slice for Multi-Volume and Single-Volume Approach. The selected domain decomposition (Tables 2.1 and 2.2) is the one delivering the highest throughput as in Figure 2.5

expect to see considerable gain when using more recent Virtex-6 or Virtex-7 hardware.

	PowerPC970MP	Itanium2 Montecito	PowerXCell 8i	Virtex4-LX200 (D1 implementation)
GFLOPS	0.81 (naive) 1.14 (blocking)	0.5 (naive) 0.69 (blocking)	59.4 (blocking)	103 (blocking)
Output Points / Second	30.8×10^6	18.6×10^6	1605×10^6	2783×10^6
Operation Frequency	2.3 GHZ	1.6 GHz	3.2 GHz	100 MHz (Data) 200 MHz (Compute)
Stencil Data Throughput	3.08 GB/s	1.86 GB/s	160.5 GB/s	278 GB/s
Normalized Data Throughput	1.34 bytes/cycle	1.16 bytes/cycle	50.2 bytes/cycle	2783 bytes/cycle
Green ratio [GFlops/watts] (Blocking)	0.0087	0.0086	0.378	0.90

Table 2.5: Performance values for all Architectures when computing the 8x9x8 stencil. Native compilers (xlc,icc) have been used at -O3 optimization level. Internal BW refers to the bandwidth observed by the 3D stencil algorithm

It is however important to note that, despite the efficient data reuse which reduces the external bandwidth, at such high rates, external bandwidth will also need to be very fast. For the V4LX200 implementation, this means that 22.24 GBytes/s (11.12 GB/s in each direction) are required to operate without stalls. High performance hardware needs to be developed in order to provide such bandwidths. For example, our development system (SGI Altix 4700) provides only 3.2GB/s per direction, which is only about one fourth of the required bandwidth.

2.4 RTM Algorithm and its Mapping on FPGA

Seismic imaging tries to generate images of the terrain in order to see the geological structures. The raw data for the seismic imaging is collected by producing acoustic shots. Due to the fact that these acoustic shots (medium perturbation) are introduced in different moments, we can process them independently. The most external loop of RTM sweeps all shots. This embarrassingly parallel loop can be distributed in a cluster or a grid of computers. The number of shots ranges from 10^5 to 10^7, depending on the size of the area to be analyzed. For each shot, we need to prepare the data of the velocity model, and the proper set of seismic traces associated with the shot.

In this chapter we are only interested in the RTM algorithm needed to process one shot, what we will call the RTM kernel. Figure 2.7 shows the pseudo-code of this algorithm. RTM is based on solving the wave equation two times. Firstly, using as left hand side the input shot (forward propagation), and secondly using as right hand side the receiver's traces (backward propagation) as shown in the Figure 2.7. Then, the two computed wave fields are correlated at each point to obtain the image.

```
Forward propagation
input: velocity model, shots
output: forward wavefield
 1: for all time steps do
 2:     for all main grid do
 3:         compute wavefield
 4:     end for
 5:     for all source location do
 6:         add source wavelet
 7:     end for
 8:     for all ABC area do
 9:         apply ABC
10:     end for
11:     for all main grid do
12:         store wavefield
13:     end for
14: end for
```

```
Backward propagation
input: velocity model, receivers' traces,
    forward wavefield
output: image
 1: for all time steps do
 2:     for all main grid do
 3:         compute wavefield
 4:     end for
 5:     for all receivers location do
 6:         add receivers data
 7:     end for
 8:     for all ABC area do
 9:         apply ABC
10:     end for
11:     for all main grid do
12:         load forward wavefield,
            correlate wavefields
13:     end for
14: end for
```

Figure 2.7: The RTM Algorithm

The statements in Figure 2.7 stands for the following:

- Line 3: Computes the Laplacian operator and the time integration. Spatial discretization uses the Finite Difference method [38], and time integration uses an explicit method. Typically, for stability conditions 10^3 points per each space dimension and 10^4 time-steps are needed. Also, this is the most computational intensive step.

- Line 6: Is the source wave introduction (shot or receivers).
- Line 9: Computes the absorbing boundary conditions (ABC).
- Line 12: Does the cross-correlation between both wave fields (backward only) and the needed I/O.

2.4.1 RTM Implementation Problems

RTM implementations have well known hotspots, on top of that, when the RTM implementation has as target platform a heterogeneous architecture, the list of those hotspots increased in particularities but not in diversity. We can divide the hotspots into three groups: memory, Input/Output and computation. In the next items, we will describe these groups:

2.4.1.1 Memory

RTM is the contemporary best migration algorithm for subsalt imaging. RTM memory consumption is related to the frequency at which the migration should be done. Higher frequencies (e.g.: over 20-30Hz) may imply the usage of several GiB ($>$ 10 GiB) of memory for migrating one single shot. The total amount of required memory could be greater than the amount available in a single computational node, forcing a domain decomposition technique to process one shot.

A 3D Finite Differences stencil has a memory access pattern [39] that can be observed in Figure 2.8 (c), the stencil is represented by the cross-shaped object (Figure 2.8(a)(b)). As can be seen from Figure 2.8 (c), only one direction (Z in that case) has the data consecutively stored in memory, then accesses to memory for other directions is very expensive, in terms of cache misses. The stencil memory access pattern is a main concern when designing the RTM kernel code [40], because it is strongly dependent on the memory hierarchy structure of the target architecture. Besides, due to the reduced size of the L1, L2 or L3 caches, blocking techniques must be applied

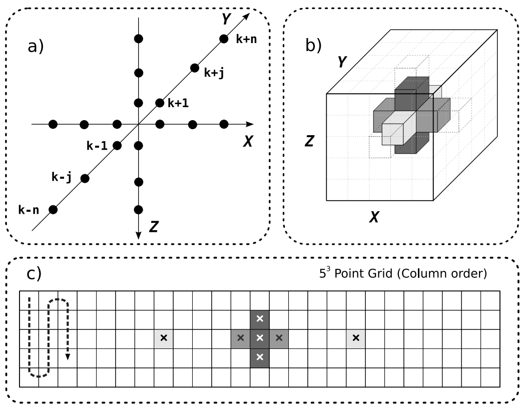


Figure 2.8: (a) A generic 3D stencil structure, (b) a 3D 7-point stencil, and (c) its memory access pattern.

to efficiently distribute the data among them [35], at least for classical multi-core platforms. Moreover, modern HPC environments (e.g. Cell/B.E or SGI Altix) have a Non Uniform Memory Access (NUMA) time, depending on the physical location of memory data. Thus, a time penalization may be paid if data is not properly distributed among memory banks.

2.4.1.2 Input/Output

We divided the I/O problem into three categories: data size ($>$ 1 TiB), storage limitations and concurrency. On one hand, looking for high accuracy the spatial discretization may produce a huge computational domain. On the other hand, the time discretization may imply large number of time-steps.

RTM implementations store the whole computational domain regarding the number of time-steps (line 12 in Fig 2.7), which may overwhelm the storage capacity ($>$ 300 GiB). In order to avoid that RTM becomes an I/O bounded application, it is mandatory to overlap computation and I/O using asynchronous libraries. Additionally, some data compression techniques can be used to reduce the amount of data transferred. Finally, the correlation can be performed every `n` steps at the expense of image quality (we call this rate *stack*).

As a distributed file system is generally used for sharing the global velocity model and seismic traces, negative behavior could be observed as the number of shots concurrently accessing the shared data increases. Therefore, using global file systems impose new constraints: the required available storage network bandwidth and the maximum number of concurrent petitions that can be served.

2.4.1.3 Computation

In order to efficiently exploit the vectorial functional units present in modern processors, we have to overcome two main problems: the low computation vs memory access (c/ma) ratio and the vectorization of the stencil computation. In order to use the pool of vector registers completely, unrolling techniques are needed.

The low c/ma ratio means that many neighbor points are accessed to compute just one central point, and even worse, many of the accessed points are not reused when the next central point is computed. This effect is called low data locality ratio. For instance, the generic stencil structure in Figure 2.8 (a) defines a $(3 \times (n \times 2)) + 1$ stencil. If $n = 4$ then 25 points are required to compute just one central point, then the c/ma ratio is 0.04, which is far from the ideal $c/ma = 1$ ratio. To tackle this problem strategies that increase data reuse must be deployed.

2.4.2 Application Specific Design of RTM

To maximize performance and minimize off-chip accesses we concentrate on maximizing data reuse in the 3D stencil. Four streams are used for the input volumes in the forward phase (current volume, previous volume, illumination and velocity volume) and one output stream is used for the output volume, one for the illumination and another for the compressed output (only when disk writes need to be performed). In the backward phase the illumination stream is replaced by the correlation stream.

A special purpose cache focusing on data reuse has been designed based on the FPGA internal Block-RAM (BRAM). In the ideal case, every point of the previous volume loaded onto the FPGAs Block-RAM would be used exactly 25 times before it is removed from the FPGA, as there are 25 stencil computations that make use of every point. In practice, however, the reuse ratio is slightly lower because no output is generated for ghost points. However, one benefit of our modeled platform is its global shared memory which allows to proceed computation without the need of communicating the ghost points between time-steps. The sub-volumes are sized such that 9 contiguous planes can be kept simultaneously in the BRAMs (Figure 2.3-b). These planes form the smallest volume that allows to compute a plane of the output sub-volume. To complete the remaining planes of the output sub-volume two techniques are used. First, internally, planes are streamed from the sub-volumes in Y-direction. Second, externally, domain decomposition is used to partition the volume into sub-volumes in the Z and X axis. This completes the computation of the whole dataset. Because the stencil requires access to volume points from the neighboring sub-volumes, the real sub-volume that is streamed already includes these ghost points.

The stencil data is laid out internally in the FPGA BRAM in a 3-level memory hierarchy (Figure 2.3-b) from which all necessary input points can be read in a single cycle. For the Virtex4-LX200 device present in the SGI Altix 4700, the dimensions of the extended sub-volume (i.e., including ghost points) are 200 points in the Z-dimension and 75 in the X-dimension. No output points are being computed for these ghost points. Therefore the reuse degree is slightly smaller, 21.44 for the sub-volumes used in this mapping. We assume the same dimensions for the Virtex-5 chip even though this chip has more on-chip memory and might thus enable somewhat larger sub-volumes with less overhead. Planes are streamed sequentially in the Y-direction, thus there is no limit on the number of planes in this direction. Thanks to data reuse and an aggressive data cache, this design can internally generate a huge supply of data. Unfortunately this supply cannot be matched by the compute units. This happens because synthesizing floating point (FP) units on FPGA chips is costly in terms of area. In general, implementing standard floating point on FPGA should be avoided due to the complexity of the IEEE754 standard, which requires, among others, continuous normalization after each operation and handling rounding modes, NaNs, etc. For FPGA it is much more efficient to use fixed point units, which can better map to the available DSP units. For RTM an interesting option to reduce area is to avoid rounding, and normalization between each partial FP operation and do it only once before the data is stored back to main memory such as in [41]. On the other hand, the data front-end can easily scale to much higher bandwidth [42].

In this basic implementation the compute units are standard data-flow versions of the stencil and time integration. In one Virtex4-LX200, two compute units are implemented running at twice the frequency of the data front-end. This allows the basic design to generate 4 output points per cycle. However, factoring the plane and column switching overheads in results in a steady state performance of 3.14 points/cycle (1.57 results/cycle per compute unit).

Using Xilinx ISE 11.1 we conclude that even without implementing the single normalization option, three compute units can be implemented in each of the 4 Virtex5 LX330 devices present in the modeled FPGA platform (CONVEY HC-1). Each compute unit consists of 27 adders and 16 multipliers. We expect the data cache to run at 150MHz and compute unit at 300 MHz. This configuration will deliver a steady state performance of 18.84 points/cycle at 150 MHz. Thus, the FPGA model requires 36 GiB/s of input bandwidth (3 Volumes $\times$ 18.84 $\times$ 67/63 sub-volume overhead $\times$ 4 bytes/point x 150MHz) and 11.3 GiB/s of output bandwidth. This is less than the 80 GiB/s that the coprocessor memory can provide. Given that memory access patterns are completely deterministic an intelligent memory scheduler should not have problems exploiting this bandwidth by minimizing memory bank access conflicts.

We complete the estimation by also analyzing the performance that can be obtained if we also accelerate the remaining parts of the code: the absorbing boundary conditions, the illumination, correlation and the compression/decompression.

2.4.2.1 ABC

Regarding the boundary conditions, they can be implemented using the same logic as the 3D stencil and time integration, but streaming planes from the volume ghost points. This way we reuse the slices of the stencil and only implement little additional logic. This will not deliver the best performance and will not be very efficient, but since the processing of ghost points is small compared to the stencil (less than 10% additional points for the volumes considered here) we do not consider it critical to accelerate this even further.

2.4.2.2 Correlation and Illumination

These operations should also be accelerated. These two embarrassingly parallel operations are very simple computationally, but they require reading and writing a whole volume. They can be computed just after completing the stencil and time integration. Given that reading and writing a volume to/from coprocessor memory proceeds at 11.3 GiB/s, we need 22.6 GiB/s to accommodate this operation without performance penalty. Overall, the computation requires 70 GiB/s, still below the 80 GiB/s maximum bandwidth.

2.4.2.3 Compression and decompression

These steps are necessary to reduce the I/O requirements. We integrate these computationally simple operations into the stencil processing unit, both to compress a volume during forward and store it, and to decompress it during the backward phase. This requires 11 GiB/s more data bandwidth because a new volume is generated. Fortunately these operations can be performed when no illumination and correlation are being computed.

2.5 Results and Discussion

We have carried out experiments to verify first the numerical soundness, and second the performance of the implementation. The experimental results show the appealing

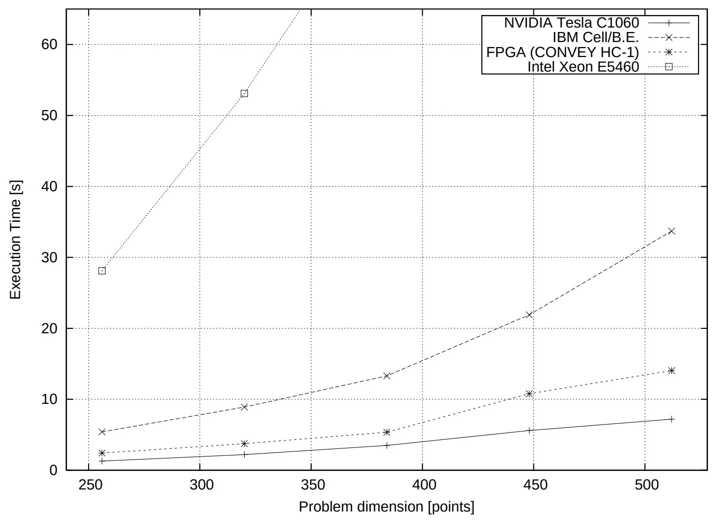


Figure 2.9: Elapsed times for computation only experiments, 100 steps, forward and backward

performance of the GPUs, Cell/B.E and FPGA with respect to the traditional multi-core architecture. The results are averages over repeated runs, to eliminate spurious effects (e.g. bus traffic, or unpredictable operating system events).

Figure 2.9 shows that all the accelerators outperform the homogeneous multi-core from 6 (Cell/B.E.) to 24 times (Tesla C1060). The Tesla C1060 outperforms all other accelerators because: is more recent than the Cell/B.E., its hardware characteristics and mainly its architecture is well suited for the algorithm mapping.

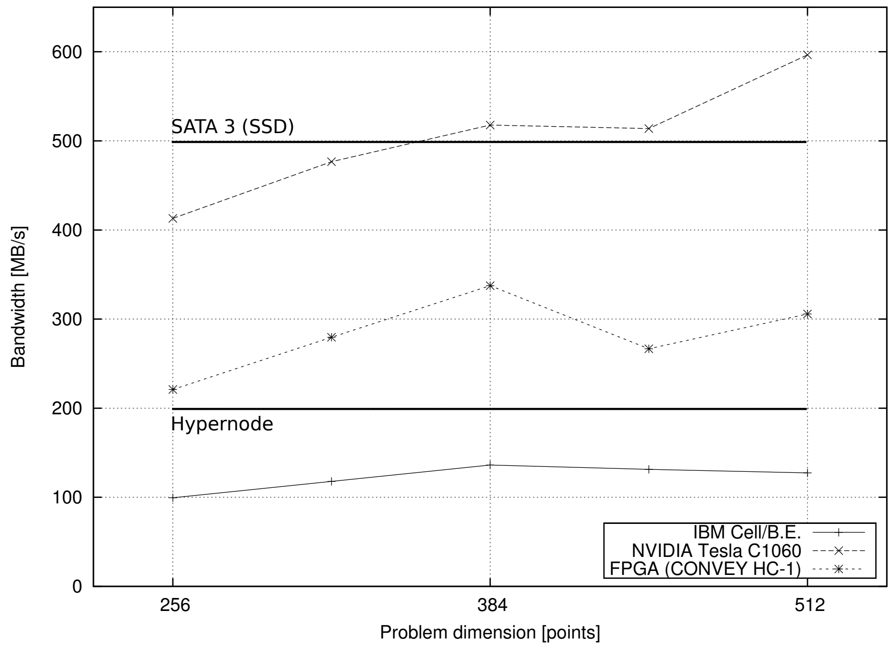


Figure 2.10: RTM forward and backward with stack 5, and high level of compression. Hypernode is a technology proposed by IBM for providing high-performance I/O, for instance for the Cell/B.E. platform.

It is observed during our work on RTM mappings to different accelerators that the I/O technologies attached to the tested architectures become an important bottleneck. This is because the accelerators deliver ready to be stored data at a rate that the I/O is unable to handle. In order to avoid this problem, we take advantage of two main strategies: increase the stack rate or apply data compression. Figure 2.10 depicts the I/O requirements for some RTM test cases, where the stack has been set to 5 steps, compression is in place and the dimension problem ranges from 256 to 512 cubic points. As can be observed, under the mentioned conditions a Hypernode (similar to a

SATA 2 10000 RPM disk), can not handle the work for every accelerator, further for GPU and FPGA cases the need for better I/O technologies is a must. If the compression level have to be reduced, even for Cell/B.E. case there will be a severe I/O bottleneck.

2.6 Summary

In this chapter we have presented a generalized implementation of 3D-Stencil and its specific mapping for RTM. The performance analysis of 3D stencils was presented for various memory organizations: CPU cache hierarchies, ScratchPad Memories (the *Local Stores* in the CellBE) and a distributed 3D memory scheme implemented on a FPGA. The key to efficiency in stencil computations is to maximize data reuse fetching input data only once. The presented FPGA implementation not only shows how this can be achieved, it also demonstrates how this approach provides tremendous internal bandwidth to the compute units. On a Virtex4-Lx200, the normalized bandwidth (i.e., bytes per cycle) is, even compared to the accumulation of the 8 CellBE SPEs, 56 times larger when operating on the $8 \times 9 \times 8$ stencil.

The performance analysis for the RTM shows that GPUs, Cell/B.E. and FPGAs outperform traditional multi-cores by one order of magnitude. However, in general, a great development effort is required – for this performance achievement – mainly because the programming environments are still immature. In particular, the RTM porting to FPGA is the one that requires most effort. All operations need to be described in HDL. IP cores provided by Xilinx CoreGen were used to increase productivity. However, for the future, high-level productivity tools will be critical to allow developers harness the potential of FPGA technology.

This chapter presented case studies specific to the implementations of 3D-Stencils in structured grid domain. The next chapter (Chapter 3) show how a 3D memory hierarchy can be very useful for mapping different application kernels as a sub-set of such a multi-level memory layout.

3

Generalization of 3D-Memory as a Common Memory Layout

Reconfigurable devices like FPGAs are mostly utilized for customized application designs with heavily pipelined and aggressively parallel computations. However, little focus is normally given to the FPGA memory organizations to efficiently use the data fetched into the FPGA. This chapter presents a Front End Memory (FEM) layout based on BRAMs and Distributed RAM for FPGA-based accelerators. The presented memory layout serves as a template for various data organizations which is in fact a step towards the standardization of a methodology for FPGA based memory management inside an accelerator. We present example application kernels implemented as specializations of the template memory layout. Further, the presented layout can be used for Spatially Mapped-Shared Memory multi-kernel applications targeting FPGAs. This fact is evaluated by mapping two applications, an Acoustic Wave Equation code and an N-Body method, to three multi-kernel execution models on a Virtex-4 Lx200 device. The results show that the shared memory model for Acoustic Wave Equation code outperforms the local and runtime reconfigured models by 1.3–$1.5\times$, respectively. For the N-Body method the shared model is slightly more efficient with a small number of bodies, but for larger systems the runtime reconfigured model shows a $3\times$ speedup over the other two models.

[1] Chapter 3 is based on the publication:
FEM : A Step Towards a Common Memory Layout for FPGA Based Accelerators; Muhammad Shafiq, Miquel Pericas, Nacho Navarro, Eduard Ayguade appeared in *20th IEEE International Conference on Field Programmable Logic and Applications, Milano, ITALY, September 2010*

3.1 Application Specific Front-Ends

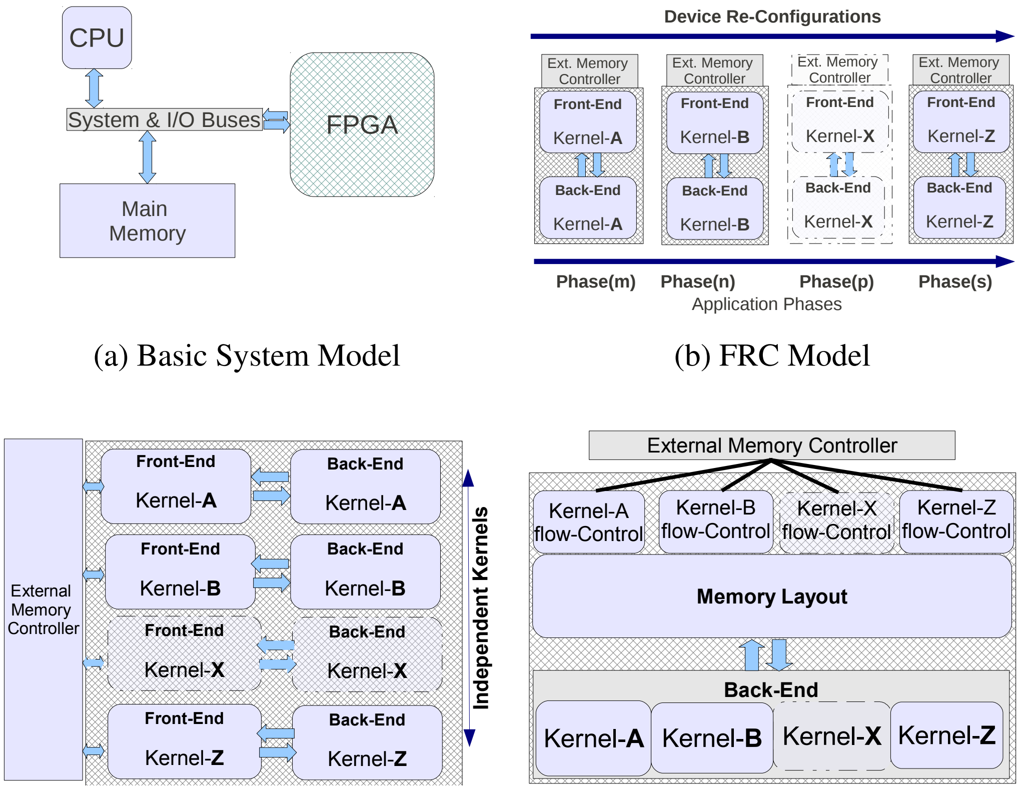
Application specific hardware designs are considered as potential candidate for accelerating applications by introducing specialized data paths and specialized computations as required by the application. One way to implement customized application architectures is by using fine grained reconfigurable Field Programmable Gate Arrays (FPGAs) technology. These devices normally operate at an order of magnitude lower frequency than that of fixed logic devices. However, performance gains are possible due to parallelism and potential elimination of overheads. Since external data bandwidth is often limited [43], it is necessary to build efficient memory management strategies by using FPGA local memory. However, FPGA based designs like [44; 45] and many others give only little attention to the efficient data management strategies for on-chip data-reuse, loop-unrolling and data-movement. One principal reason is that HDL developers do not have any application level standard view of the memory layout that they can conceive in their designs for their applications. Therefore, with some exceptions like [46] or [32], most of the application specific implementations remain more focused on computations while on-chip memory is only used for lookup-data or to stream data through simple FIFOs. This is why if we look at various FPGA based implementations of web applications [45; 46], sequence alignment algorithms [47; 48], signal processing kernels [32; 44; 49; 50; 51; 52] and many others, we will observe almost no harmony between the memory layouts used for each implementation.

This work is a step towards the harmonization of front-ends of various FPGA based application specific architectures for an efficient arrangement of data before it is forwarded to the compute back-ends. The main contributions of this work are:

- We present a *template memory layout* that implements a Front End Memory (FEM) on FPGA for various applications.
- We show how the template memory layout can be specialized for various example kernels.
- We evaluate the template memory layout using two applications and three ways (Section 3.2) to map multi-kernel applications to reconfigurable hardware layout.

3.2 Compute Models for Multi-Kernel Applications

Although much research has focused on individually accelerating compute-intensive kernels, real HPC applications actually consist of many kernels [53]. Accelerating these applications on hybrid CPU-FPGA machine will need to focus on the integration of the accelerated kernels with the rest of the system in order to overcome the natural limits expressed by Amdahl's law. However, if porting of the full compute-intensive section of the application to the FPGA subsystem is possible, then the remaining Host-FPGA overheads and host computations can be mostly neglected. However, this proves challenging because it requires to integrate multiple different kernels into a single design and to efficiently manage data.



(a) Basic System Model

(b) FRC Model

(c) SM-LM Model

(d) SM-SM Model

Figure 3.1: Compute Models that are evaluated in this work

One of the main complexities of mapping multiple kernels to an FPGA device is how to share data across kernels. The following list describes different ways to map more than one kernel on an FPGA so that kernels can share data. In this chapter we focus on a simplified machine architecture that consists of a host with main memory, and an FPGA that can receive data streams from the main memory. The data arrangement and stream generation needs to be fixed by the application. We consider that an application is implemented on an arbitrary FPGA having enough slices to accommodate the target application. Figure 3.1(a) shows the machine model considered in this design.

3.2.0.4 Full Reconfiguration (FRC)

Each kernel maps to the complete FPGA (Figure 3.1(b)) and can make use of all BRAMs for its storage purposes. When a kernel finishes and a new kernel needs to start, the FPGA is reconfigured and a new bitstream is loaded. This model incurs the overheads of reconfiguration and the need to checkpoint/restore the data across reconfigurations. The benefit of this model is that kernels can store a larger working set in the FPGA.

3.2.0.5 Spatially Mapped-Local Memory (SM-LM)

In this model (Figure 3.1(c)), all kernels are mapped at the same time on the FPGA. Data Storage is partitioned among the kernels so that each one has exclusive access to its working set. This model overcomes the overheads of reconfiguration, and is simple to implement. However, it can only store a smaller copy of local data and it may require to move data from one local store to the next one before the following kernel can be executed.

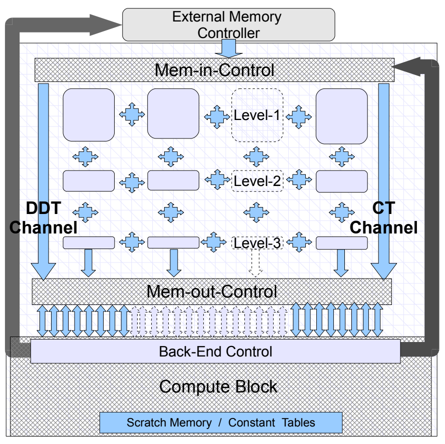
3.2.0.6 Spatially Mapped-Shared Memory (SM-SM)

This model is similar to the previous one, but instead of keeping local copies of data, a shared memory model is implemented that fronts all back-ends as shown in Figure 3.1(d). This model removes the working set constraints and data movements imposed by the previous model. However, it has slightly increased complexity in the design of the shared memory which can result in area overheads and slower execution frequency.

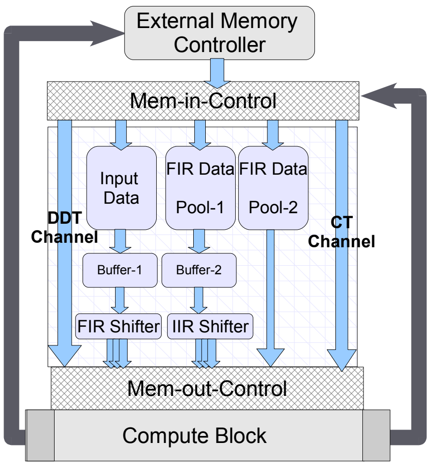
3.3 Front End Memory Layout For Reconfigurable Accelerators

The memory layout-focused FPGA-based compute model shown in Figure 3.2(a) is the generic block diagram of the FPGA based computing architecture pursued in our proposal. This architectural model includes a front-end memory layout and a back-end compute-block along with the major data-paths. The front-end memory layout deals with the memory management issues for an accelerator while the back-end performs computations. The front-end and the back-ends work in a tightly coupled configuration, however, the flow of data inside the front-end layout can be changed as required by the back-ends. This is shown for various examples in Section 3.3.1. In case an application is using the FEM layout for implementation of different data-flows for different application kernels then in order to select a data-flow control for a specific kernel, we need to send the identification of the control-flow to the FPGA compute model. This identification includes, but is not limited to, the **type** of required kernel and the **size** of the data, that will be streamed from host to FPGA during the phase of execution for selected kernel. This startup information can vary in identification parameters from selection of one kernel to another one.

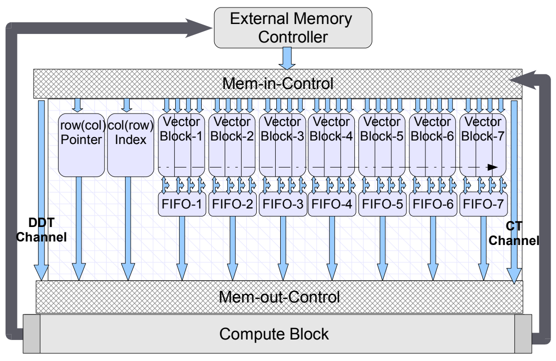
The FEM layout or a subset of the layout can be used by various application kernels. The layouts provide a front-end for dynamic data organization inside the FPGA. This front-end layout is based on three levels. Level-1 is a set S_n of n memory blocks having depth D_{l1} and width W_{l1}. Each memory block in this level can be accessed to a finer granularity G_{bits} on the horizontal front. For example in the evaluation we use S_9, $D_{l1} = 4096$, $W_{l1} = 128$ and G_{32} or G_{64}. The evaluation considers 32-bits or 64-bits as the basic data types for kernels. The second level (Level-2) has the same number of blocks as in Level-1 but with different depth ($D_{l2} = 64$) and same width ($W_{l2} = 128$) as that of Level-1. Moreover, at this level access granularity remains constant and equal to that of width W_{l2}. The third level is based on a register set of size $128 \times k$ with capability to shift-right on-demand for $32 \times m$ bits where k and m are arbitrary numbers chosen according to the implementation of the kernel. The third level is implemented using distributed RAM while Levels 1 and 2 are implemented using BlockRAMs.



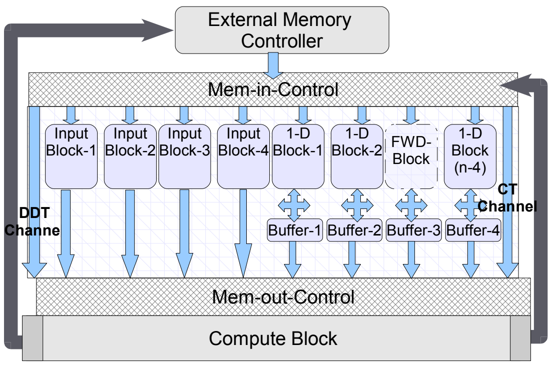
(a) FEM based Compute Machine



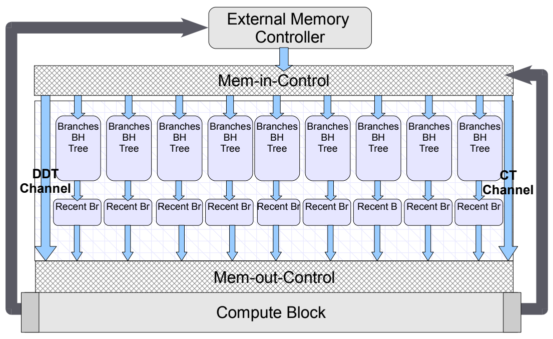
(b) FIR/IIR



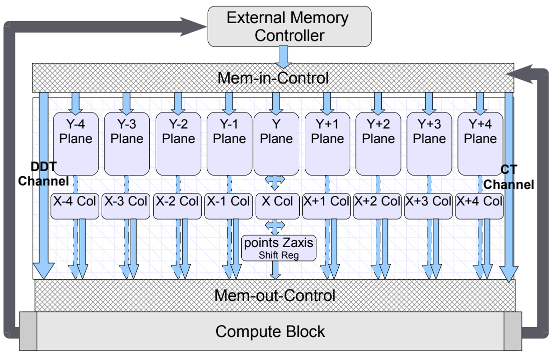
(c) SPMvM (SpMVMs/SpMVMl)



(d) FFT (1D/2D)



(e) NBody(Naive/BH),DFT(1D/2D),MM



(f) AWE (WFC, BPC)

Figure 3.2: FEM based conceptual machine architecture (a) and Front-End Memory Layouts for various kernels shown in the sub-figures b, c, d, e and f

The writing to the three-level memory blocks is controlled by the Mem-in-Control (Memory Input Control) block and the data read from this layout is controlled by the Mem-out-Control (Memory Output Control) block. These mem-in/out-control blocks are specialized units to support various data flows in multiple directions and up to three dimensions corresponding to three levels according to the selection of the kernels at compile time. These specialized memory controls can also implement conditional execution of some states for run-time selection of different application kernels whose data sets are part of the memory layout or a sub-set layout. These control blocks can also use a Direct Data Transfer (DDT) Channel, shown in Figure-3.2(a), to directly forward the stream data by bypassing the layout if needed by the kernel. The Constant Transfer (CT) channel can be used same as DDT channel by the mem-in/out-control to transfer constant data to the back-ends. The computed results in the back-end are directly forwarded by the back-end control to the external memory controller and/or sent to Mem-in-Control for saving in blocks of Level-1. All data flows are implemented according to the need of an application kernel. However, FEM supports data flow only from top to bottom i.e. from Level-1 to Level-2 and Level-2 to Level-3. Any level can bypass its data directly to compute-block. However, skipping is not allowed across levels which means Level-1 can not forward data to Level-3 by skipping Level-2. Mem-in-Control allows Level-1 to be simultaneously written by both data coming from the external memory controller as well as data being fed back from the compute blocks. Multiple blocks can be written with the same data in parallel and all blocks at all levels can be read in parallel. This in-fact gives FEM architecture an opportunity to increase the internal bandwidth of the data and do parallel loop-unrolled computations.

3.3.1 Example FEM layouts for Scientific Kernels

In this section we show how several different kernels can be implemented using the FEM layout that has just been introduced.

3.3.1.1 Digital Filters

Figure-3.2(b) corresponds to an infinite impulse response (IIR) filter. A finite impulse response (FIR) filter can be selected as a sub-set layout by ignoring the feedback path (pool-1, pool-2 etc.). The input data samples pass by the *Input Data Block*, *Buffer-1* and are arranged in *fir-shifter* before being forwarded to the back-end. The *fir-shifter* works such that multiple sets of time shifted samples can be forwarded to back-end for parallel computations of multiple points. For IIR, *Pool-1* and *Pool-2* contains the feedback data (FIR computed data) transferred from compute-block. The FIFO buffer (Pool-2) absorbs the multipliers' and adders' latencies inside Compute Block during computations on the *pool-1* data before it is added to the the *pool-2* data.

3.3.1.2 Sparse Matrix-Vector Multiplication (SpMVM)

FEM can accommodate two types of memory layouts for SpMVM ($SpMVM_s$ & $SpMVM_l$). The difference between these two types is the size of the vector that is kept inside the FPGA's memory. In case of $SpMVM_s$ the maximum vector length could be up to $D_{l1} \times W_{l1}/G_{bits}$ with each entry of size G_{bits}. This vector would only need to be sent once to the FPGA. Inside the FPGA seven copies of the vector are maintained as shown in Figure-3.2(c). Each copy is accessed as independent W_{l1}/G_{bits} channels making the data flow deterministic for parallel multiplications. On the other hand, $SpMVM_l$ stores only one copy of the vector with maximum size of $D_{l1} \times W_{l1}/G_{bits} \times (n-2)$. This version, however, can have non-deterministic latencies because requests for a data set present in the same *vector block* need first to be arbitrated before being fetched and arranged into FIFOs at Level-2.

3.3.1.3 Fast Fourier Transform (1D & 2D)

The FEM layout for fast fourier transforms (FFT) is presented in Figure-3.2(d). For 2D FFT, the size of 2D-Frame must be lesser than $D_{l1} \times W_{l1}/G_{bits} \times (n-5)$ of G_{bits} sized data elements. FEM memory layout follows the data organization concept presented in [52] for 1D Radix-4 FFT (decimation in frequency). It further extends the same idea for handling 2D Radix-4 FFT by enabling feedback of 1D FFT data to Level-1 and incorporating Level-2 as a buffer for selected set of data from Level-1.

3.3.1.4 N-Body (Naive/Barnes-Hut), DFT (1D/2D) and Matrix Multiplication

The architectural layout shown in Figure-3.2(e) works as a memory structure for the following kernels: *N-Body naive/ Barnes-Hut*, *discrete fourier transform (1D/2D)* and *matrix-vector* or *matrix-matrix multiplication*. This layout fits for the applications kernels with property of large repeated interactions between various sets of data. In these applications, in most cases, one set of data remains constant over a long period of time before it is replaced by another set of data. In this layout a chunk of data is pre-fetched from external memory and arranged into FPGA FEM blocks such that for every cycle, the set of different blocks should be able to feed data as parallel operands to the compute block in the back-end.

3.3.1.5 Acoustic Wave Equation Solver (AWE)

The AWE solver has two main kernels: Wave Front Computation (WFC) and Boundary Point Computation (BPC). Figure-3.2(f) shows FEM layout for the WFC kernel for which we use a 4-point 3D even symmetric stencil with time integration. For the BPC kernel we use a one point 3D stencil with time integration which can be implemented as a subset of the WFC layout. The FEM layout for *AWE* follows the memory organization concept given in [32]. However, there are some differences. In this work the solver uses only 9 blocks at Level-1 and Level-2 (shown as planes and columns) while keeping a 12 point shift register at Level-3. Writing and reading is done exclusively for each block in the same circular read/write control for Mem-in/out-Control. This FEM layout produces enough internal data bandwidth to accommodate two compute modules in the back-end. The compute modules also contain a time-integration part which only adds a latency corresponding to a new floating point module and directly uses two more volumes in its computations forwarded through the DDT channel.

3.4 Evaluations

Two example applications, namely the 3D- Acoustic Wave Equation Solver and an N-Body Hermite algorithm, have been mapped on the three execution models of the FPGA based computing system. These models include Fully Reconfigurable Compute (FRC) Model, Spatially Mapped-Local Memory (SM-LM) compute model and Spatially Mapped-Shared Memory (SM-SM) model. The details on these models have been discussed in Section-3.2. The evaluated multi-kernel applications have been implemented in *Verilog* HDL using ISE 9.2i and tested on a Virtex-4 Lx200 [28] device,

Table 3.1: Comparison of resources (FEM layout & Controls) required by AWE Solver and N-Body Hermite Algorithm for the three computing models.

	AWE Solver		**N-Body (Naive)**	
	Total Slices	Total BRAMS	Approx. Slices	Approx. BRAMS
FRC	17818(max)	307(max)	7518 (max)	312(max)
SM-LM	25734	322	13518	312
SM-SM	17818	307	10518	320

attached to an Altix-4700 [37] machine. Table 3.1 shows the resource usage in terms of slices and BlockRAMs for both the applications.

3.5 Results and Discussion

In following, we discuss the results of AWE solver implementation which is followed by a detailed discussion on the N-Body related results.

3.5.1 AWE (WFC, BPC) Solver

The acoustic wave solver has two main kernels: *Wave Field Computation (WFC)* and *Boundary points Computations (BPC)*. These kernels have similar structure and share the following main properties:

- WFC implements a four point even symmetric 3D-Stencil while BPC is a one point even symmetric stencil.
- WFC and BPC both involve time integration using two previous volumes.
- In the case of volume decomposition into sub-volumes, BPC kernel is required only for the sub-volumes including the boundary points of the main volume.

The WFC kernel is implemented as shown in Figure-3.2(f). However, the other kernel (BPC) is implemented as small subsets of the same layout by enabling the process for only three blocks at Level-1 and Level-2 because it implements only a single point stencil and computes only the wave front side of the boundary points. AWE uses in total three volumes of data, of which two are directly consumed by the back-end through the DDT channel, and only one (the current) volume is managed by the FEM memory layout to exploit data reuse.

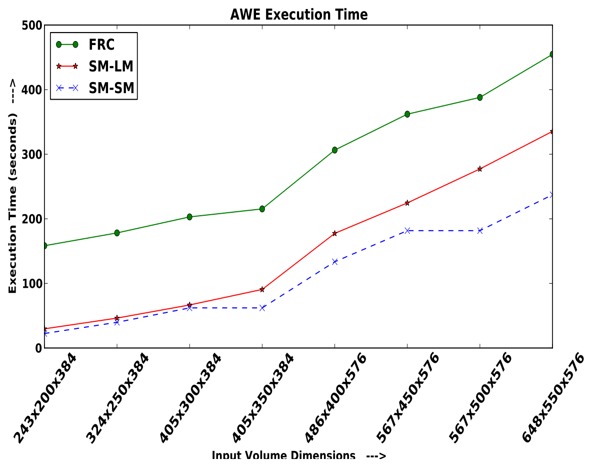
The FEM layout for AWE, working at 100MHz, supports computation of 3.14 points/cycle [32]. In SM-SM model, two kernels are selected conditionally while SM-LM model implements them in parallel. The results shown in Figure-3.3(a,b) correspond to different sizes of input volumes, computed for 500 time steps. The results show that the Full Reconfiguration (FRC) execution model performs the worst compared to the other two models. This is because a fixed time for re-configuration of the device (128ms@50MHz [28]) is needed for every time-step as many times as there are different kernels. Further use of data fetched for WFC kernel for the boundary blocks is not possible in FRC scheme. However, The SM-SM takes significantly less execution time ($0.66\times$) than the SM-LM model even though the SM-LM model allows the BPC kernel to execute in parallel to WFC. This happens because the shared model can use bigger sub-volumes with lesser overheads due to the shared memory between sub-set layouts of the two kernels which is not the case in the local memory organization. The device resource usages are also better for the SM-SM model than the SM-LM because of less replication. The FEM property of common layout makes it feasible for SM-SM model to utilize the same layout for the three kernels. Only a conditional selection is needed at Mem-out-Control for selecting one or multiple kernels to forward the data at an arbitrary time during the execution.

3.5.2 N-Body Hermite Algorithm

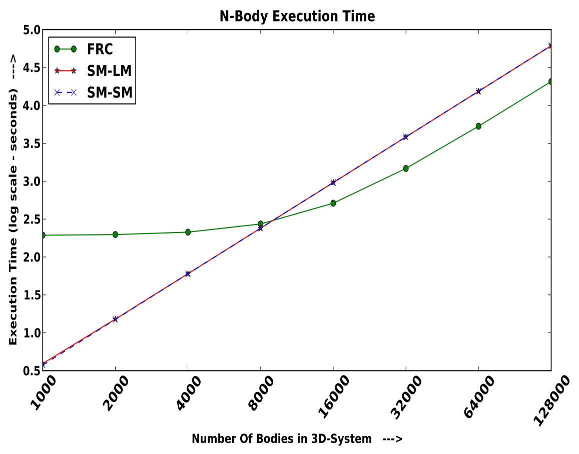
The FEM layout for N-Body naive method presented here, uses a 3-Dimensional Hermite Scheme. The 3D-Hermite algorithm computes movement of bodies using the newtonian gravitational force. This kernel is compute intensive and offers the possibility of high data reuse. The algorithm is based on the following three main computational kernels, executed for N bodies over an arbitrary number of time steps:

- Prediction of Bodies Movement (PBM)
- Computation of Newtonian Forces (CNF)
- Correction of Bodies Movement (CBM)

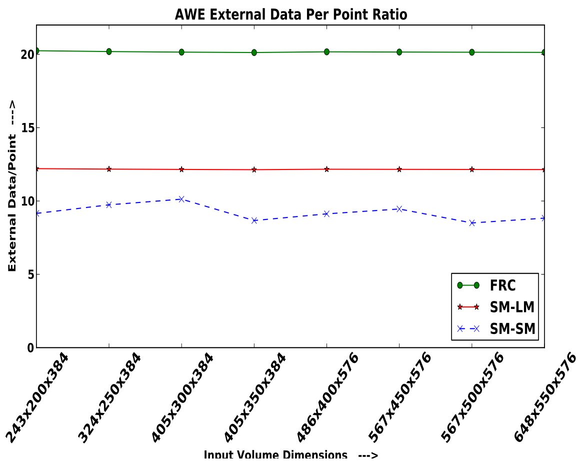
The basic description of the N-Body system is represented by a set of three parameters for each body. These include mass of body, 3D initial velocity and 3D initial



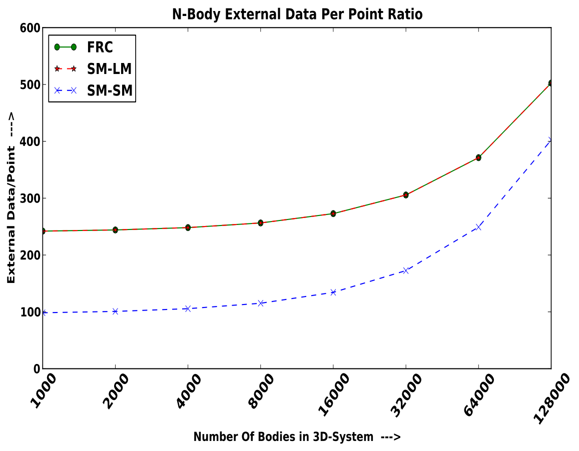
(a) AWE Execution Time



(c) NBody Execution Time



(b) AWE External Data/Point Ratio



(d) N-Body External Data/Point Ratio

Figure 3.3: AWE and N-Body Performance Evaluation for **FRC**,**SM-LM** and **SM-SM** Compute Models

position. During execution of the algorithm, the CNF kernel generates two more parameters corresponding to 3D-acceleration and 3D-jerk and the CBM kernel produces two additional parameters corresponding to the updated velocity and position for each particle. In addition to this, the system also needs to maintain two parameters corresponding to the *old Jerk* and the *old acceleration* for a body to be used in CBM. This means that an active N-Body system needs to maintain eight parameters which in turn correspond to 25 double precision data elements for each of the bodies in the system. The FEM layout arranges bodies data in the horizontal order in sets of memory blocks at Level-1 such that all data corresponding to at least two bodies is accessible in the same cycle. A system with a large number of bodies (not fitable inside BRAM) is processed by a decomposition of the system into subgroups.

The computational complexity of PBM and CBM is *O(N)* while for CNF, it is *O(N^2)*. Moreover, the computations in PBM, CNF and CBM require fetching 12, 7 and 18 data elements (with each element requiring 8 bytes), respectively, for each body. PBM and CBM perform computations as the data arrives. This means that these kernels do not need large storage using BRAMs. However, CNF uses BRAMs to store the maximum possible number of bodies (i.e. mass and predicted position and velocity parameters). We consider the accelerator to be working at 100MHz with external data bandwidth of 1GB/s/direction for 500 time steps using various system sizes as shown in the Figure 3.3(c,d).

In the FRC and SM-LM models, the three kernels execute sequentially. Therefore, the data per point ratio for these models is the same, as is apparent from Figure 3.3(d). However, in the case of the SM-SM model, the sharing of memory by kernels makes it possible to use the four parameters for predicted values of velocity and position and new values for acceleration and jerk from inside the accelerator. This makes the external data per point ratio better than for FRC and SM-LM. The execution time for the SM-SM model (Figure 3.3(c)) also performs slightly better than the other two models for systems with a small number of bodies. Here FRC loses efficiency due to overhead of the reconfiguration time and some latency. The SM-LM model remains inefficient in this case (small N-Body system) due to the latency produced by PBM computations on the first group of bodies before these can be forwarded to the CNF computational kernel. However, for larger N-Body systems, these reconfiguration and latency factors are negligible for FRC and SM-LM compared to the overall computation time of the system. Moreover, the FRC model shows better execution time ($3\times$) as compared to other models due to the availability of full chip resources that makes it possible for each kernel to use more compute units.

3.6 Summary

This work is a step towards standardization of a common memory layout for FPGA based accelerators. In this work we have presented the FEM layout for FPGA based accelerators and shown with various examples that the idea works for a range of application kernels. Further, the concept of a FEM based common memory layout enables the conditional selection of multiple kernels, using the same or a subset of the layout. This configuration has the potential to result in a shared memory computational model which we have then compared with other execution models for two applications. The results reveal that the shared memory model gets better performance in solving the Acoustic Wave Equation while full reconfiguration model improves the execution time for the computationally intensive N-Body algorithm for systems with more than 8K bodies. However, for both applications the requirement of external data per point ratio remains best for the shared memory model.

This chapter has presented a motivational study showing that various application kernels can be designed by using a similar memory structure. The usability of such a memory layout can be limited because of the fact that different application kernels can require different data-flow paths. Therefore, all data-flows may not be possible to model in a common way on top of a common memory layout. This motivates to develop systems that can map different kernels on top of a common memory layout in a generic way. This makes the basis of our next chapter which proposes a source to source translation tool for template based design expansions targeting reconfigurable devices.

Part II

Template Based Design Support for Customized Memory Accelerators

4

The DATE System

Past research has addressed the issue of using FPGAs as accelerators for HPC systems. Such research has identified that writing low level code for the generation of an efficient, portable and scalable architecture is challenging. We propose to increase the level of abstraction in order to help developers of reconfigurable accelerators deal with these three key issues. Our approach implements domain specific abstractions for FPGA based accelerators using techniques from generic programming. In this chapter we explain the main concepts behind our system to Design Accelerators by Template Expansions (DATE). The DATE system can be effectively used for expanding individual kernels of an application and also for the generation of interfaces between various kernels to implement a complete system architecture. We present evaluations for six kernels as examples of individual kernel generation using the proposed system. Our evaluations are mainly intended to provide a proof-of-concept. We also show the usage of the DATE system for integration of various kernels to build a complete system based on a Template Architecture for Reconfigurable Accelerator Designs (TARCAD).

4.1 Templates in Reconfigurable Computing

Previous research like the ones presented by Shafiq et al. [32], Lin et al. [46] and Chao et al. [52] has shown how FPGAs can achieve high performance on certain kernels by

[1] Chapter 4 is based on the following works:
(1) A Template System for the Efficient Compilation of Domain Abstractions onto Reconfigurable Computers; Muhammad Shafiq, Miquel Pericas and Eduard Ayguade appeared in HiPEAC WRC January 2011, Heraklion, Greece
(2) A Template System for the Efficient Compilation of Domain Abstractions onto Reconfigurable Computers; Muhammad Shafiq, Miquel Pericas, Nacho Navarro and Eduard Ayguade, Accepted for Journal of System Architecture. [Pending Publication]

customizing the hardware to the application. However, applications are getting more and more complex, with multiple kernels and complex data arrangements. The efficient management of the memory, compute modules and their interfaces is a task that is difficult for performance. This is because different applications exhibits different data access patterns, forcing the architecture designers to keep a generic interface between the memory management unit and the compute units. This results in a compromise on the performance because of the generic way of data transactions. The performance can be improved if data is marshaled according to application need before writing to the local memory and then a generic interface between (local) memory and compute units can deliver better performance by accessing aligned data. However, it is not easy to achieve this concept even by using fully configurable devices. Many studies like the ones by Henry [54] and Araya-Polo et al. [3] highlights that accelerating applications of various kernels is not an easy task on reconfigurable accelerators. It requires significant effort of the application programmer to make an efficient implementation of each kernel and as well handle an efficient flow of data between these kernels. In our view, the implementation of individual kernels can be done in a better way by the domain experts and application programmers may only concentrate on the flow of data between these ready-made kernels.

Achieving notable speedups for HPC applications by using reconfigurable devices is not the only requirement. Portability and scalability of the architectures are also of great concern. Contemporary methods for the development of customized architectures using HDLs (Hardware Description Languages) or using HLS (High-level synthesis) tools allow portability and scalability of a kernel implementation to an arbitrary extent largely dependent upon the design of the tool. However, this work proposes to increase the level of abstraction on top of a HDL or HLS tool for ultimate generation of an RTL for a reconfigurable device. Conceptually, this gives an opportunity to translate the domain specific code for any selected target HDL tool like Verilog [55], VHDL [56] or for an HLS tool like ROCCC [57], GAUT [58], Autopilot [59] etc. This makes our proposal – presented in this work – an interesting choice for better potability and scalability by choosing from any of the supported HLS tools for mapping the domain abstractions into a code compilable by that selected tool.

In the existing systems with high level abstrations, the most widely used are the C++ Templates [60] for general purpose computing. The methodology of sofware abstraction has also enabled domain specific libraries to be developed for dense/sparse algebra, spectral analysis, structured grids for solving PDEs, and also parallelization and domain decomposition [61; 62]. Similarly, Catanzaro et al. [63] from UC Berkeley presents case studies for source to source transformations of high level abstractions done in productivity languages Ruby and Python to performance oriented languages C++ and CUDA targeting multicore x86 and multicore GPUs. A recent keynote by Truchard [64] and the work done by Bhatt et al. [65] propose LabView and MATLAB packages respectively for describing problems in high level abstractions. Kulkarni et al. [66] and Rubow et al. [67] present CLIFF and CHIMPP frameworks respectively, both of which are a mapping of a network domain specific language "CLICK" to FPGA platform. A qualitative study done by Vajda et al. [68] proposes language oriented software engineering (LOSE) to create domain specific high level development environments usable by domain experts. On the accelerators side CUDA Templates [69], VHDL generics [56] and the Xilinx CoreGen tool [70] are some examples of the template systems and its variant forms. However, both CoreGen and VHDL Generics suffer from the fact that substitution is performed at a level too close to the implementation (namely at netlist level). In these conditions high level optimizations such as loop unrolling, code hoisting or dependency analysis are not possible. Our intention in the template based design approach is to support all problem domains in a generic way by using both substitution and code expansion on top of a high level programming language, allowing developers to specify domain abstractions and at the same time generate high performance implementations.

In this work we explore the possibility of using generic programming as a way to generate high performance FPGA implementations for individual kernels and to generate the interfaces between various kernels to be integrated into an efficient system. The generation of kernels and the integration at the system level, both use high level domain abstractions. Templates are used to implement domain specific constructs. The proposed template system is used for two types of source to source translations: i) C to C, ii) HDL to HDL. In the first case, the system offerers translations of high level domain abstractions in the source code to a C version specific for a C to HDL compiler. These C to HDL compilers (eg. ROCCC, GAUT etc) then further apply optimizations like the loop unrolling, code hoisting along with data dependency analysis before generating the host code. The domain experts – while writing the domain specific templates– can accommodate specific optimization directives related to the tool by using control directives of C (e.g #ifdef, #def etc). This allows the tool directives to be automatically inserted in the final output. In the case of HDL to HDL translations, loop unrolling, code hoisting and dependency analysis like processing is not required. However, the template system does a very fine job by offloading the dirty work of scaling the data-paths, scaling of control structures and as well scaling the functionality by using a template architecture for the system. In our current work, the system level integrations use only HDLs and HDL templates.

This chapter explains the core idea of the system using the example of a simple FIR filter. We also explain how the DATE system can be very helpful for combining various modules in an envelop of a Template Architecture for Reconfigurable Accelerators Designs (TARCAD). Moreover, we evaluate the DATE system for six kernels from three individual complex domain *classes*: Multidimensional Stencil, Multidimensional FFT and Digital Filters. Further, it is also recommended to refer to our TARCAD work [8] which is based on the template expansion system and presents further case studies from other application domains. Our evaluation are based on expansions of the templates using DATE system in combination with the HLS and RTL tools. Different tools use very different internal designs and their outputs can not be compared meaningfully (Sarkar et al. [71]). Therefore, our evaluations are mainly intended to give a proof-of-concept.

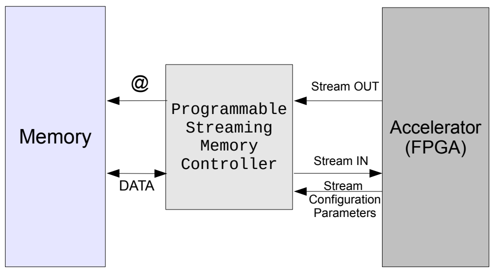

Memory
@
DATA
Programmable Streaming Memory Controller
Stream OUT
Stream IN
Stream Configuration Parameters
Accelerator (FPGA)

Figure 4.1: Streaming Environment with programmable streaming memory controller

4.2 Background

Our proposal on the template system for reconfigurable accelerators to support complexity and minimize the glue code is based on modern programming techniques. As a compute model that supports acceleration of full applications we will assume a variation of the streaming dataflow model. The layout of the proposed computing model is shown in Figure 4.1. In a traditional streaming dataflow model, data is read from memory in chunks of sequential data. These chunks are fed into the accelerator, which processes them and generates another data stream corresponding to the result. Within the accelerator, a pipeline of tasks processes the code, possibly making use of some local storage for tasks such as buffering or data reuse. Thus, the complexity of (FPGA) accelerator code is only limited by how many operations one can fit. Streaming itself is often limited in the form of memory accesses that it can support. Therefore, in our model we consider an external streaming memory controller to have more intelligence and to offer data access patterns that are not only linear, but which can be programmed with techniques with loop nests such as those described by Ketterlin et al. in [72] and Hussain et al. in [73]. The result is similar to the decoupled access execute (DAE) architectures proposed by Smith et al. [74] and provides a much more general execution model. Of course, many problems exist that cannot be expressed as a streaming-accelerator problem. Problems such as database processing, tree sorting, etc, have tight memory read-write-read cycles, little computation and very unpredictable control flow which do not fit in this model. However, this is no way a constraint for our proposed template system. It is just a matter of choice for a system. A non-streaming model can be designed by selecting templates of the modules which can work efficiently for randomly addressed accesses.

Most FPGA-based application accelerators have focused on implementing simple kernels in FPGA and executing the remaining parts of the application in a host. The approach works effectively if applications consist largely of a single (possibly parameterized) kernel or a set of kernels that can be spatially mapped on hardware provided that the rate of synchronizations between host and accelerator is sparse enough. Many applications do, however, not correspond to this simple model, and this is a trend that is changing even further as mentioned by Dongarra et al. [75]. As applications evolve, their complexity increases as new components are integrated into the code base and platform specific optimizations are introduced. The glue code necessary to handle multiple components requires the developer to write control code that is difficult to scale, manipulate and not always portable. Therefore, the focus of this work is to provide a base platform to handle these every day increasing design complexities in an automated way but not at the cost of performance loss for the resultant reconfigurable accelerators.

4.3 The DATE System

The DATE (Design Of Accelerators by Template Expansion) system is shown in Figure 4.2. The basic idea of the DATE system is to support the translation of template based HLL (High Level Language) programs to a notation acceptable by an arbitrary set of HLS (High Level Synthesis) tools or RTL (Register Transfer Level) synthesis tools. These tools then help either individually or in a combination to generate functionally equivalent hardware for a reconfigurable device. The DATE system is developed keeping in mind the contemporary and future needs of reconfigurable accelerator designs for HPC applications. The system generates output either in C (always compatible to an HLS tool) or HDL (Verilog or VHDL) forms. The dual type of output makes the DATE system a potential tool aligned with the contemporary needs. The translation of HLL descriptions to C makes it possible to utilize the contemporary and future outcomes from the large number of development efforts being made for a generic C to HDL/Netlist (i.e HLS) tool. Second, it also gives an opportunity to the system to generate Direct HDL from HLL domain abstractions for specialized architectures not efficiently conceivable by a generic HLS tool. Inside the DATE system, our main focus is on the Domain-Translator which is developed as a prototype by using Python and its extensions. It is important to explain the working principle of the whole system before understanding the focused part.

4.3.1 The DATE System : From The Front-Side

The DATE System accepts a HLL code that uses abstract constructs and methods to implement domain specific computing. The HLL coding style for the input of the

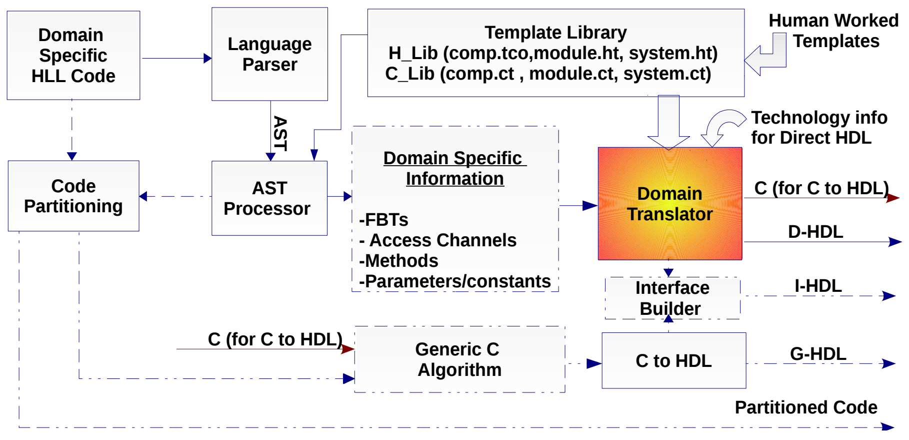


Figure 4.2: The Architecture of the DATE System

system follows closely the syntax and semantics of C++ language. The implementation of the front-side (language parser and AST processor) of the DATE system is not an objective of this work. We are using command line inputs to model the data-set coming from the front-side of the DATE System. This data-set is passed-on to the Domain-Translator (Section 4.3.3). However, for an extended automated tool set, the input source code will be passed through a parser at the front-side to get an abstract syntax tree (AST). An AST-Processor will process this AST in combination with the original source code and the directory of the domain specific templates available from the Template-Library of the DATE system to retrieve the domain specific information related to FBTs, DACs, Abstract Methods and other parameters. These inputs (currently as command line inputs) to the DATE system are described in the forthcoming sections.

4.3.1.1 Functional Bucket Types (FBT)

FBTs are domain specific abstract data types which need some predefined periodic operations on their data before it is forwarded to the computational parts. For example, in the case of a simple FIR filter, every time step the data samples of type Data_Type are shifted for one sample inside a filter window so that the most recent TAP number of samples can be used in computations. It can be noticed in this example that data needs to go through some operations (periodic shifting in an order) before that any computations are applied on it. This motivates to decouple data and its movements from computations. Therefore, the concept behind FBT is to represent a data set in containers called *Buckets* along with cyclic operations (called *function*) on these containers before forwarding for any computations. These containers are created from BRAMS and/or Registers and cyclic operations are performed in a state machine.

4.3.1.2 Read/Write Data Access Channels (DACs)

DACs make it possible to access data from FBTs with an arbitrary granularity and interleaving, both defined through parameters.

4.3.1.3 Abstract Methods

The DATE system expects that the input HLL code would be using generic programming constructs like class templates, function templates, etc. Therefore, the abstract methods declared in HLL code and available in the Template Library of the DATE system are forwarded to the Domain-Translator (more details in Section 4.3.3). Any procedural methods used inside the HLL code would need to pass through the HLS tool directly as shown by the block “Generic C Algorithm” in Figure 4.2.

4.3.1.4 Parameter Set

The translation process also requires the related parameters, arguments and type specifiers from the HLL source to expand and generate the code for FBT’s, DACs and Abstract Methods.

4.3.1.5 System I/O

In the case of the system I/O, currently the DATE system considers that the global memory access by the reconfigurable accelerator is based on a programmable streaming memory controller (Figure 4.1) based on the proposal of Hussain et al. [10]. The controller fetches complete data patterns and forward them to the accelerator as data stream.

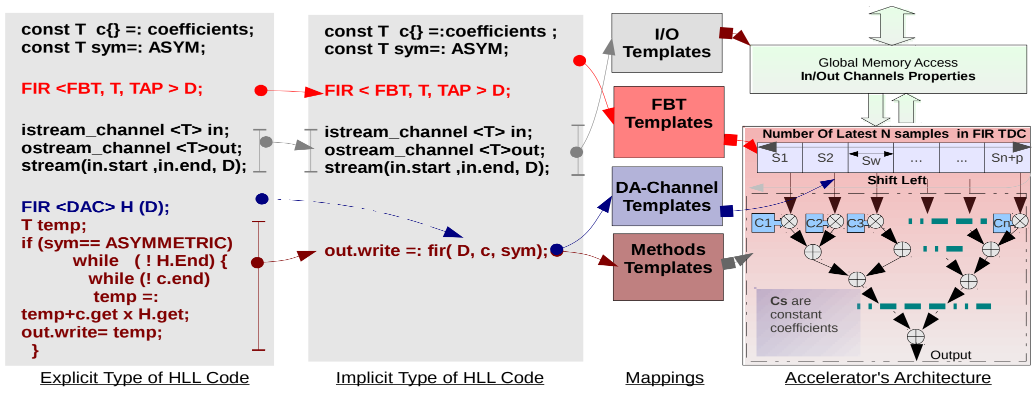
4.3.1.6 The Translations

The Buckets in FBTs are translated to dual ported memory modules which are accessible independently on one side by the system I/O for streamed data and from the other side by the Data Access Channels (DACs). The DACs are translated to the internal data and address buses. The abstract methods access data from the FBTs in a pattern through the DACs. This data access functionality of the DACs is translated to the behavior of the bus controller inside the hardware of an abstract method. The abstract methods are also kept as function templates in the template library of the DATE system and these are expanded according to the type of the domain to which the methods belong to. The example shown in Figure 4.3 helps to clarify the domain abstracted HLL mappings done during the translation process. This example presents a simple domain of a FIR (Finite Impulse Response) filter. The FBTs are parameterized for the size of *buckets* (here one bucket stores only one sample of data) i.e. *int16*, *int32*, *long* or *an arbitrary structure* and the number of buckets (*TAPs*) inside the filter. Moreover, the shift operations –required for an FIR domain– are also part of the FBT template.

4.3.1.7 DATE Input Source Code Types (Implicit and Explicit)

The FBT's data is accessed by using the DACs. However, the DATE system differentiates between the different input HLL codes or parts of a single HLL source code. This differentiation is done on the basis of the behavior needed for the DACs and the usage of the data fetched from FBTs using these DACs. If the behavior of DACs and the usage of data is inherited by an abstract method corresponding to a domain then the code is taken as an implicit implementation as shown in Figure 4.3 (Implicit Type of HLL Code). Otherwise, if the behavior of DACs and the usage of data is done by using control programming constructs then the DATE system considers it as an explicit implementation as shown in Figure 4.3 (Explicit Type of HLL Code).

For implicit codes, along with the FBTs and DACs templates, the abstract methods are also maintained as templates in the Template-Library of the DATE System. The FBT's templates and method's templates are expanded separately and connected to each other based on the DAC's templates. The implicit expansions of templates by the DATE System are actually the original goal of the DATE system for flexible and efficient mapping of domain abstractions onto reconfigurable computers. However, the



```
const T  c{} =: coefficients;
const T sym=: ASYM;

FIR <FBT, T, TAP > D;

istream_channel <T> in;
ostream_channel <T>out;
stream(in.start ,in.end, D);

FIR <DAC> H (D);
T temp;
if (sym== ASYMMETRIC)
        while   ( ! H.End) {
          while (! c.end)
           temp =:
temp+c.get x H.get;
out.write= temp;
  }
```

```
const T  c{} =:coefficients ;
const T sym=: ASYM;

FIR < FBT, T, TAP > D;

istream_channel <T> in;
ostream_channel <T>out;
stream(in.start ,in.end, D);

out.write =: fir( D, c, sym);
```

I/O Templates
FBT Templates
DA-Channel Templates
Methods Templates
Global Memory Access
In/Out Channels Properties
Number Of Latest N samples in FIR TDC
S1
S2
Sw
...
...
Sn+p
Shift Left
C1
C2
C3
Cn
Cs are constant coefficients
Output
Explicit Type of HLL Code
Implicit Type of HLL Code
Mappings
Accelerator's Architecture

Figure 4.3: An example of explicit and implicit coding styles with their mappings for the generation of an accelerator architecture

DATE system is also being extended to handle the explicitly styled HLL codes. In the case of the explicit implementations, expansions for FBTs are generated by the DATE system but the algorithmic part having control coding constructs is forwarded to a C-to-HDL compiler. The interface between the two parts (i.e. the FBTs and the algorithmic HDL generated by a C-to-HDL compiler) can be written manually or integrated in an automated way under TARCAD system (Section-4.4).

4.3.2 The DATE System : At The Back-End

The Back-End side of the DATE system (right side of Figure 4.2) generates outputs either in C or HDL forms. This is purely dependent on the availability of a template type (HDL-Template or C-Template) for a domain inside the Template-Library. However, it can also be a matter of choice.

4.3.2.1 Template's Expansion to C

The DATE system can expand HLL templates into C-codes. The generated C codes are specifically compatible to an HLS tool. Template expansion targeting HLS uses C-based templates corresponding to the abstract classes/methods declared in the input source code. The ultimate goal of the generated C-code is to be later translated to some kind of hardware description format like an HDL or a netlist, etc. As far as we know, till this date all publicly available tools for C-to-HDL or C-to-Netlist compilation use a subset of the C language and add extensions for more comprehensive types. These practical facts also need to be considered by the DATE system while expanding a template into a C-code so that the generated code can be tested using available C to HDL tools. Currently, we are keeping DATE output in a C format compatible to ROCCC [57] which is an open source C-to-HDL compiler. The C-code generated by DATE is also used for compilation by the GAUT [58] tool. We use GAUT for evaluation purposes, therefore the coding style adjustments needed by GAUT are done manually. However, an extension in the DATE system to cover broader range of data types along with adjustments in function interfaces can make it work for GAUT and other HLS tools.

4.3.2.2 Template's Expansion to HDL

The DATE system also outputs codes in HDL format. In case of D-HDL (Direct HDL), HDL templates are expanded by the DATE System's from its template library (more information in 4.3.4). However, G-HDL (HDL generated by C-to-HDL compiler) is produced first by templates expansions to C by the DATE system and then using a C-to-HDL tool. The I-HDL (Interface HDL) is used optionally, it is generated at *Interface Builder* by selection of a template interface out of a predefined set of interfaces. I-HDL provides an interface between the D-HDL with the G-HDL. Currently the DATE System generates Verilog based HDL implementations. ROCCC and GAUT both generate VHDL modules therefore for a multi-module application we obtain mixes of Verilog and VHDL designs. This does not make much difference because contemporary synthesis tools can work well for these kinds of designs.

4.3.3 The DATE System Center: The Domain-Translator

The Domain-Translator takes as input the domain specific types for the FBTs, DACs, Methods and related parameters (data types, arguments, constants, dimensions etc.) as shown in Figure 4.2. In the case of implicit types of codes (Section 4.3.1), the DACs are the parts of the template definitions of abstract methods and FBTs. However, the DATE system extensions (more information in Section 4.4) to integrate multiple kernels in one system uses separate templates for the DACs. A domain specific template

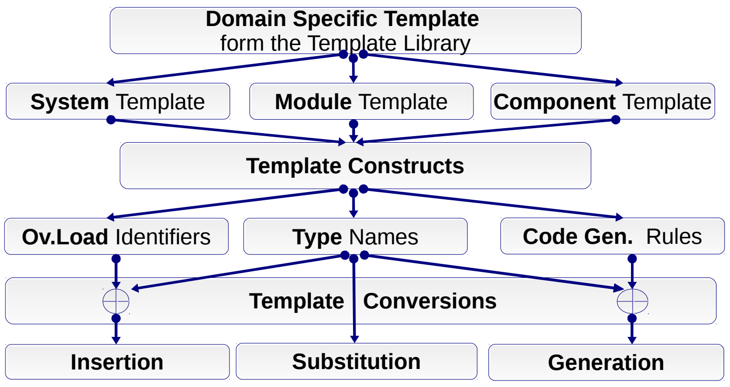


Figure 4.4: Internal Flow of the Domain Translator

in the Template-Library can contain three types of Sub-Template definitions as shown in Figure 4.4 and discussed below.

- System-Template: Defines the top-level functionality and template connectivity between the system and the module.
- Module-Template: Defines a template with a data access pattern for the method and operations on the data.
- Component-Template: Keeps templates for the components (adders / multipliers /square roots/special functions etc.) used in the template module.

These Sub-Template definitions are maintained in either or both languages (C or HDL) for which the ultimate translation is required. Each one of these Sub-Template types can contain further three types of Template-Constructs.

- Overload Identifiers : Makes it possible for the Domain-Translator to choose and insert an appropriate Component-Template for a Module-Template. The Module-Templates of an application can also works as Component-Templates for another application.
- Type-Names: These define the data types being utilized in a Sub-Templates.
- Code Gen Rules: Allows the identification of the part of a code and its generation with induction of variables where required.

The Domain-Translator uses Type-Names from the Template-Constructs for processing the other two constructs (i.e. Overload-Identifiers and Code Gen Rules) in its final Template-Conversion stage as shown in Figure 4.4. The Overload-Identifiers are used to insert a code specific to the overloaded operator or the overloaded function to make the template meaningful and functionally correct in its ultimate expansion. This inserted code can even be a sort of a Sub-Template type or a simple function or an operator. The Code Gen Rules generates code based on definitions of the rules inside the Domain-Translator design. The Type-Names are only substituted according to the type parameters provided by the application programmer at compile time. The Domain-Translator treats both the C-based sub-templates and HDL-based sub-templates in the same way. However the sub-templates are themselves should be in the same language for which output is required.

4.3.3.1 Template Design

The DATE system isolates the domain experts from the application programmers by raising the level of program abstractions. These high level abstractions are translated by the DATE system to the HLS or HDL specific code facilitating the application programmers to get efficient and fast implementations of the RTL codes for arbitrary kernels. Otherwise, the programmer would need to write the domain specific code manually by possibly consuming more development time as can be seen from the Figure 4.15. Moreover, the ultimate performance will also depend upon the application programmer's expertise for the domain under implementation. Therefore, we assume that a template for a kernel would be designed by the kernel's domain expert rather than an application level programmer.

In general, the template designer should be aware of all or most of the the possible expansions a kernel would need in the future. This makes it possible for the designer to list the input parameters those could be used as a set of external parameters to the *DATE* system for specific code generation. Moreover, the domain expert while developing the template should also be able to identify those locations of the code which could be expanded either based on the *Overload Identifiers* or the *Code Generation Rules*. The template designer will use all this information along with the set of *Type-Names*, *Overload Identifiers* and the *Code Generation Rules* provided by the *DATE* system to implement a template for a kernel. The *DATE* system supports user defined *Type-Names* and *Overload Identifiers*. However, *Code Generation Rules* are only used which are defined by the *DATE* system. The implementation of *System-Template* for a kernel is compulsory. However, writing the *Module-Templates* and the *Component-Templates* are optional. This depends upon the choice of the designer to keep the template based kernel designs modular or just as system level implementations.

4.3.4 The Date System : Template Library

We explain the internals of the Template Library with a simple example of a digital filter from the FIR (Finite Impulse Response) domain as shown in Figure 4.3. Generally a FIR filter takes N of the most recent input samples, multiplies them by N coefficients and sums the result to form one output Y_n. In this example the FIR's FBT is parameterized for the size (i.e. Data_Type like *int32* or *long* etc.) of the Buckets and the total number of Buckets (i.e. TAPs) inside the filter. We will consider that one Bucket stores only one sample of data. Moreover, the FBT also contains periodic shift operations as a property of the FIR domain.

```
template < typename Data_Type, typename TAP >
module FilterContainer_Data_Type (reset,clk,FilterEnable,CurrentInput,
DelayedValues,VlidDValues);
input    reset;
input    clk;
input    FilterEnable;
input    [Data_Type-1:0] CurrentInput;
output   [TAP*Data_Type-1:0] DelayedValues;
output   VlidDValues;
reg      DelayedValues;
reg      VlidDValues;
always @ (posedge clk or posedge reset)
  begin
        if (reset)
                begin
                VlidDValues          <=1'd0;
                DelayedValues        <=0;
                end
        else begin
        if(FilterEnable) begin
DelayedValues <= (DelayedValues << Data_Type)|{TAP*Data_Type-Data_Type,CurrentInput};
        end
        VlidDValues          <= FilterEnable;
  end
end
endmodule
```

Figure 4.5: Direct HDL : Template for FIR Shifter Module

4.3.4.1 Example HDL Template (The FIR Domain)

Figure 4.5 shows a template for a FIR Shifter-Module maintained in the Template-Library for generating a Direct HDL code using the DATE system. This Shifter-Module only works for the shifting of data samples. This simple template uses the Type-Names for adapting itself at compile time according to the declaration of the HLL Filter class. As described in Section 4.3.3, Type-Names are only substituted by the DATE system. Therefore, for this code expansion only the parameters passed to the Domain-Translator corresponding to Data_Type and TAP are substituted at the appropriate places.

The HDL template for the System-Module of the FIR computational part is shown in Figure 4.6. This module also uses *Type_Names* and substitutes three parameters corresponding to Data_Type, TAP and CONST_VALUES (the constant filter coefficients). The system module also uses *Code Gen Rules* to generate terms with multiplication

```
template < typename Data_Type , typename TAP, typename CONST_VALUES >
module FilterMethod_Data_Type
(reset ,clk ,FilterEnable ,CurrentInput ,FilterOutPut , OutputVlid );
input  reset;
input  clk;
input  FilterEnable;
input  [Data_Type:0] CurrentInput;
output [Data_Type-1:0] FilterOutPut;
output OutputVlid;
wire  [TAP*Data_Type-1:0] DelayedValues;
wire  VlidDValue;
reg   FilterOutPut;
reg   OutputVlid;
wire [Data_Type:0] coef[TAP:0];
template <rule DECLARE TAP>
assign coef[${Declare_Index}]= {${Declare_Value}} ;
template<rule DECLARE TAP> wire [Data_Type-1:0] iter${Declare_Index} ;
template<rule DECLARE TAP>
assign  iter${Declare_Index} = DelayedValues[Data_Type * (${Declare_Index}+1)-1 :
Data_Type * ${Declare_Index}];
FilterContainer_Data_Type  fc_Data_Type(reset ,clk ,FilterEnable ,CurrentInput ,
DelayedValues ,VlidDValue);
always @ (posedge clk or posedge reset)
  begin
if (reset)begin
OutputVlid <=1'd0;
FilterOutPut <=0;
end
else begin
if(VlidDValue)begin
template<rule ADDA TAP>FilterOutPut<=coef[${Declare_Index}]* iter${Declare_Index}
end
OutputVlid<=VlidDValue;
end
end
endmodule
```

Figure 4.6: Direct HDL : Template for FIR System Module

and addition according to the number of TAPs. In the next step, the multiplication and addition signs in the code are taken as *Overload Identifiers* and activate the insertion of related Component-Templates in the form of instantiation of modules for multipliers and adders. These *Component Templates* further use *Type_Names* to modify the component parameters. For example, in the case of Xilinx tools [76] the command file (.xco) for the CoreGen [70] is updated with the widths of input and output operands according to the *Data_Type*. The DATE system then uses *coregen* shell command to generate the new multipliers and adders for the FIR filter. Further details on the *CodeGen Rules* are given in the next section.

4.3.4.2 Example C Template (The FIR Domain)

The Sub-Templates for Module and System for the FIR filter domain to generate a C-code for ROCCC are shown in Figures 4.7 and 4.8. Both of the templates use Type-Names and the CodeGen Rules but Overload Identifiers are not needed in this case. The substitution of the Type-Names is done exactly the same way as described for Direct HDL generation. The module template uses two rules at different places of the code as shown in Figure 4.7. The rule <*rule DECLARE TAP*> generates TAP number of variables of type ROCCC_int_Data_Type. The rule <*rule ADDA TAP*> is used to generate code based on the code following the rule declaration until it encounters the end of line . This rule means that the Domain-Translator should "ADD right side of the **equal** operator for TAP times and assign to the left side".

In the System-Template, along with similar Template Constructs and other code generation rules, <*rule InsFCallArg TAP FIR 1*> is declared to insert TAP number of arguments in the *FIR* function call starting from the first place in the argument list. The identifier *Declare_Index*, used in different rules is considered as an internal variable of the Domain-Translator. This internal variable is used to substitute any incremental values in a generated code. In the *FIR* case, the internal variable starts from zero value for the times the code is repeated in its generation. The Domain-Translator maintains various types of internal variables to support the generation of variable names inside the expanded code at compile time. All identifiers in a template starting from a "**$**" sign represent some kind of internal variables of the Domain-Translator.

```
template < typename Data_Type , typename TAP ,
           typename CONST_VALUES >
typedef int ROCCC_intData_Type;
typedef struct
{
template < rule DECLARE TAP > ROCCC_intData_Type A${Declare_Index}_in ; // Inputs
ROCCC_intData_Type result_out ; // Outputs
} FIR_t ;
FIR_t FIR(FIR_t f)
{
const ROCCC_intData_Type T[TAP] = {CONST_VALUES} ;
template < rule ADDA TAP >f.result_out=f.A${Declare_Index}_in * T[${Declare_Index}];
  return f ;
}
```

Figure 4.7: C Template : FIR Computational Module

```
template < typename Data_Type ,
           typename TAP >
typedef int ROCCC_intData_Type;
#include "roccc-library.h"
void firSystem()
{
  ROCCC_intData_Type A[10] ;
  ROCCC_intData_Type B[10] ;
  int i ;
  ROCCC_intData_Type myTmp ;
  for(i = 0 ; i < 100 ; ++i)
  {
template <rule InsFCallArg TAP FIR 1 > FIR(A[i+${Declare_Index}],myTmp) ;
     B[i] = myTmp ;
  }
}
```

Figure 4.8: C Template : FIR System Template

4.4 Generation of Complete System Architecture Using DATE System

An extension of the DATE system is used to map and integrate multiple kernels on top of a template architecture TARCAD [8]. This extension of the DATE system uses a TARCAD-Template-Library to connect various blocks of TARCAD. The motivation behind the TARCAD based design framework is to harmonize the implementation of data-flow architectures for various FPGA-based applications written in HDLs (e.g. Verilog, VHDL) and High Level Languages (HLL). The architectures generated by HLL to HDL/Netlist tools (e.g. such as ROCCC [77] or GAUT [58]) also follow a simplified and standardized compilation target, but they have been designed specifically as compiler targets, which reduces their applicability to HDL designers. The TARCAD proposal is based on an architectural template that allows to efficiently exploit FPGAs supported by a simple programming methodology. TARCAD not only

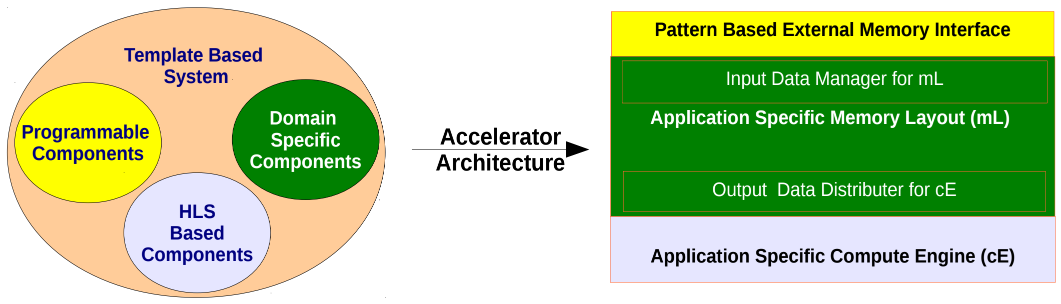

Template Based System
Programmable Components
Domain Specific Components
HLS Based Components
Accelerator Architecture
Pattern Based External Memory Interface
Input Data Manager for mL
Application Specific Memory Layout (mL)
Output Data Distributer for cE
Application Specific Compute Engine (cE)

Figure 4.9: TARCAD Conceptual Diagram

enables HDL designers to work on a highly customizable architecture, it also defines a set of interfaces that make it attractive as a target for a HLL-to-HDL compilation infrastructure.

As it appears from its name, TARCAD (Template Architecture for Reconfigurable Accelerator Designs) is a proposal based on a template architecture which consists of a number of existing modules. The user is given liberty to make a design of his desire for local memory and compute units. The user also writes a program for TARCAD's central control to guide the application execution. The generation of a new architecture based on TARCAD needs to precisely adjust the interfaces between the existing modules of the TARCAD and the user modules. The adjustment of these interfaces is not just the actual task but our template system also scales the internal data paths and controls of the existing modules accordingly. In this regard, the DATE system can be very useful for mapping applications correctly on such a generic architectural layout (TARCAD).

4.4.1 DATE System for Generation of TARCAD Based Designs

A conceptual diagram of TARCAD is shown in Figure 4.9. The left side of the figure shows the basic concepts used in the TARCAD proposal. The right side of the figure shows that the TARCAD layout can be partitioned into minimum three representative main blocks: *The External Memory Interface* , *The Application Specific Data Management Block* and *The Algorithm Compute Back-End*. A fourth important part of TARCAD – not shown in this figure – is attributed as *Event Managing Block* which acts as a supervisor for the whole TARCAD based system. These main blocks also have their constituent sub-blocks to help in the generation of a complete working system. More details on TARCAD architectures can be found in a work done by by Shafiq et al. [8].

The motivation behind the TARCAD layout is to support efficient mapping of applications on TARCAD's partitioned layout. TARCAD has its own modules (blocks) and it also accepts modules (blocks) from the user. Therefore, specific mappings for a designs require to physically interface different blocks and sub-block as shown from a high level view in Figure 4.10. This interface management is important to correctly plug-in different design modules to realize a TARCAD based design. The template interfaces of the TARCAD can be scaled to correctly map flow of data between various modules. These interface changes for a reconfigurable device can be made only at compile time. Therefore, we are propounding the implementation of the TARCAD using a template expansion method based on the DATE system.

The proposed application of DATE system for TARCAD architecture generation is based on two steps as shown in Figure 4.11. In the first step, the DATE system is used to generate RTL or C for a HLS tool from a domain description. In case of a C-code specific to a HLS, it is passed through the HLS tool along with any other hand written C based modules. Once all modules are converted to physical RTLs, these are again fed to the DATE system as a second step to map on the TARCAD layout. In order to map the RTLs on TARCAD, the user may need to provide a set of external parameters along with the RTLs to the DATE system. These parameters also include the identification of various modules corresponding to various TARCAD blocks. The DATE system uses a TARCAD based template library while building the interfaces between the input RTLs and the existing modules of TARCAD. The availability of the DATE system allows

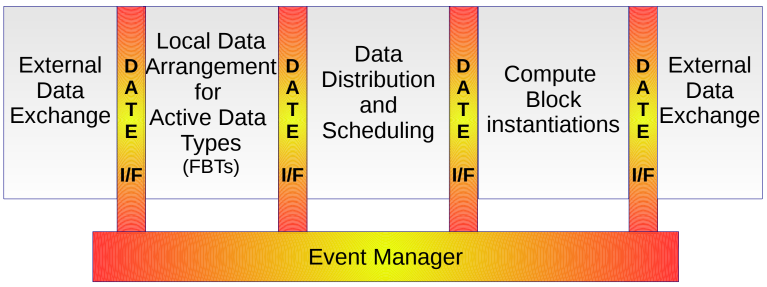

External Data Exchange
DATE I/F
Local Data Arrangement for Active Data Types (FBTs)
DATE I/F
Data Distribution and Scheduling
DATE I/F
Compute Block instantiations
DATE I/F
External Data Exchange
Event Manager

Figure 4.10: Integration of specialized design modules for a kernel into TARCAD system

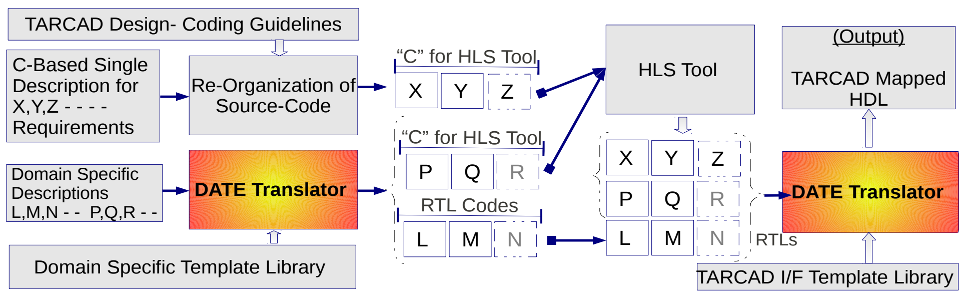

TARCAD Design- Coding Guidelines
C-Based Single Description for X,Y,Z - - - - Requirements
Re-Organization of Source-Code
"C" for HLS Tool
X
Y
Z
"C" for HLS Tool
P
Q
R
RTL Codes
L
M
N
Domain Specific Descriptions L,M,N - - P,Q,R - -
DATE Translator
Domain Specific Template Library
HLS Tool
(Output)
TARCAD Mapped HDL
DATE Translator
RTLs
TARCAD I/F Template Library

Figure 4.11: DATE System Support for TARCAD Design

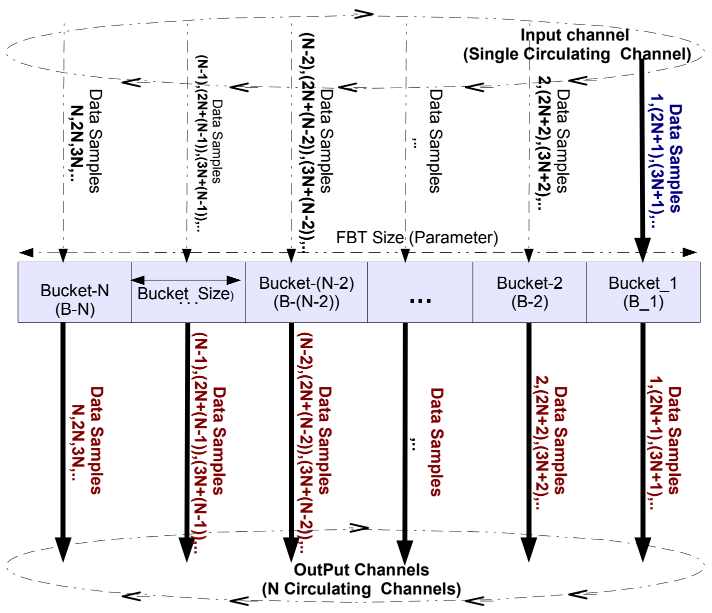
to independently design specialized architectures for various parts of the kernel in a data-flow envelope supported by the TARCAD architectural layout.

4.5 Evaluations

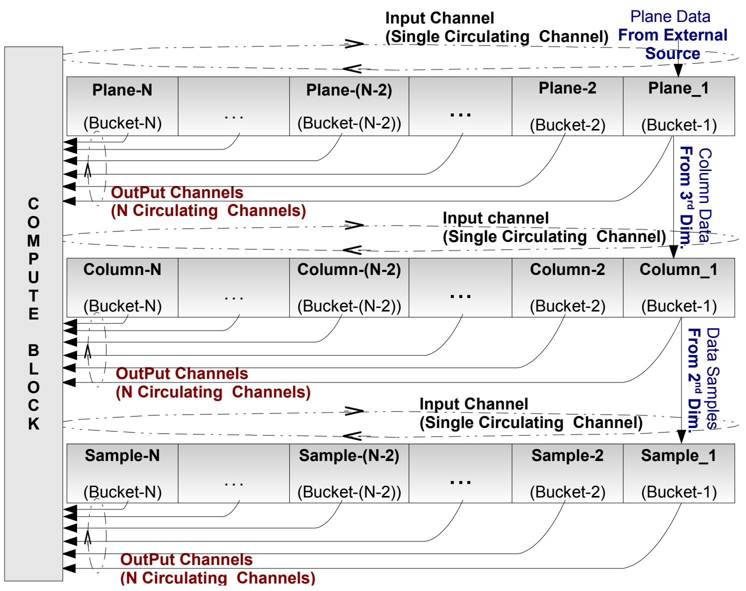
In our evaluation we use six example kernels from three domain abstractions: FIR filter, Multidimensional Stencil and Multidimensional FFT. The FIR domain abstractions are already explained in Section 4.3 to present the basic methodology of the DATE system design. In the following we discuss the remaining of the two domains.

4.5.1 A Stencil Template Scalable to Multidimensional Stencil

A stencil is a kind of filter that can be extended to multiple dimensions. The scalability of our stencil template from the basic template declarations for the architecture shown in Figure 4.12(a) to the evolution of the architecture presented in Figure 4.12(b) emphasizes the potential of the template expansion system of DATE. We only discuss the FBT part of the basic stencil template which makes it possible to maximize the reuse of data by its efficient handling of data. The computational template module of the stencil is just a representative of a reduction tree consisting of simple multiplications and additions. The FBT template for the abstract stencil domain follows the memory organization concept given by Shafiq et al. [32] with the difference that the writing and reading at different levels (corresponding to different dimensions) is done exclusively but in parallel on both sides of the dual ported BRAM-Blocks (here Buckets). This means that in our case, one side of the dual ported memory is fixed for writing


Input channel
(Single Circulating Channel)
Data Samples N,2N,3N,..
Data Samples (N-1),(2N+(N-1)),(3N+(N-1)),..
Data Samples (N-2),(2N+(N-2)),(3N+(N-2)),..
Data Samples ..
Data Samples 2,(2N+2),(3N+2),..
Data Samples 1,(2N+1),(3N+1),..
FBT Size (Parameter)
Bucket-N (B-N)
Bucket Size
Bucket-(N-2) (B-(N-2))
...
Bucket-2 (B-2)
Bucket_1 (B_1)
Data Samples N,2N,3N,..
Data Samples (N-1),(2N+(N-1)),(3N+(N-1)),..
Data Samples (N-2),(2N+(N-2)),(3N+(N-2)),..
Data Samples ..
Data Samples 2,(2N+2),(3N+2),..
Data Samples 1,(2N+1),(3N+1),..
OutPut Channels
(N Circulating Channels)

(a)



(b)

Figure 4.12: The Basic FBT Template structure for the abstract Stencil Domain (a), Generation of FBT for 3-Dimensional Stencil (b)

and the other side for reading. As compared to the design presented by Shafiq et al. [32] for a fixed $8 \times 9 \times 8$ points (odd-symmetric), 32bit, 3D-Stencil architecture, our FBT template generates a flexible implementation of a stencil that can be 1D, 2D or 3D of various sizes stencils and handling different Data_Types. The declaration of this stencil class in HLL is parameterized with “P” point stencil, having “Dim“ number of dimensions that uses data samples of type “T”. The FBT template for the basic stencil is constructed by keeping in mind some basic requirements described in following sections.

4.5.1.1 Stencil Type

Stencil Type is the total number of points from a dimension used in computation of a single output point. The stencil type can be even or odd symmetric. Therefore, the parameter Stencil Type also contains “e“ or “o“ as identifier along with the number of points. For example, in case of $8 \times 9 \times 8$ stencil. The stencil type will be ”8 o“. The stencil type defines the number of Buckets in the FBT. Figure 4.12(a) shows N-Buckets in the stencil FBT.

4.5.1.2 Stencil Dimensions

The FBT for each dimension can consist of one or more Buckets of data depending on the stencil type. The size of these Buckets plays an important role in forming a multidimensional stencil. For example, in a 1D stencil, each Bucket might only hold a single sample. However, in a 2D stencil each Bucket for the second dimension can hold a full column (i.e. all data samples from 2nd dimension). The size of the Buckets for a dimension is described by the sample’s Data_Type and the number of data samples in the dimension. The sizes of DACs (bus connections with Buckets) should also be expanded in compliance (i.e. size of address and data buses) with the sizes of the *Buckets*.

4.5.1.3 Parallel Computations

The requirement of parallel computations defines the number of samples accessible in one cycle from a Bucket. In this case, a Bucket can have multiple consecutive samples accessible in the same cycle. This also makes the DACs (buses) to be defined

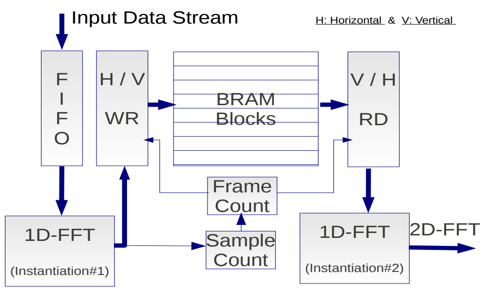


Figure 4.13: 2D-FFT Architecture

as of the same size as of the Buckets widths so that all samples maintained for parallel computations should be accessible in the same cycle.

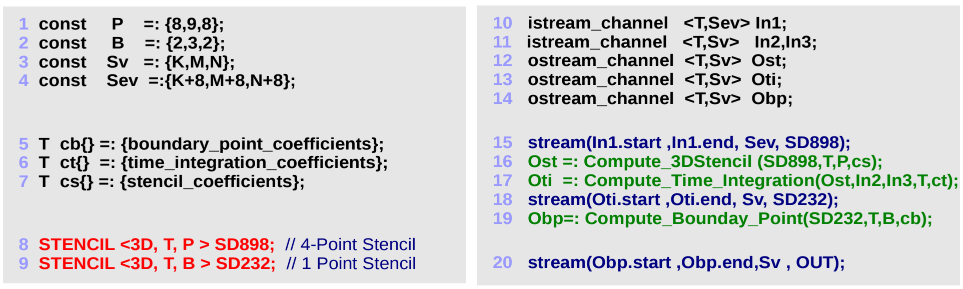
4.5.2 2D-FFT Translations

FFT (Fast Fourier Transform) adopts Divide-and-Conquer approach in DFT (Discrete Fourier Transform) algorithm to make the computation more efficient. In our evaluations of the DATE system for the two dimensional FFT (2D-FFT), we use an architecture based on two main parts, the data management part and the 1D-FFT computational part instantiated twice as shown in Figure 4.13. The data management part maintains internal 2D-Frames for transposed accesses by the second 1D-FFT module. The internal 2D-Frames are managed by toggling the writing (WR) and reading (RD) sides for the horizontal and vertical order of the BRAMs on the alternative frames. This specialized memory layout is hard to generate by HLS tool. Therefore, this memory management part of the 2D-FFT architecture is only kept in the DATE library as a template in the HDL. This template can be used by the DATE system to increase the size of individual memory block and the number of independent memory blocks according to the X and Y dimensional parameters passed to the system. The read and write widths of the memory blocks are decided based on the width of the data samples (*Data_Type*).

In case of the computational part (1D-FFT), C and HDL templates are built as domain abstraction that can expand to Radix-2 implementations. The FFT (1D-FFT) template takes two parameters: the *Data_Type* and the *Points* (points to be computed for FFT). Currently the FFT templates (C & HDL) can be expanded to 64 point computations for various data types. The *Code Gen Rules* operates on the *Butterfly* compute templates to expand the code for the FFT according to the parameter for the number of points in the HLL code. The *Butterfly* module takes two numbers each of which has a real and an imaginary part. *Twiddle* factor (real and imaginary) is an other standard FFT parameter used in FFT computation which is declared as an array of constants and selected for each stage of the *Butterfly* unit by the *Overload Identifiers*. The template design uses the basic *Butterfly* unit instantiations by passing relevant over loaded *Twiddle* factors and sample values in different instantiations. A template based bit reversal module with ability of multiple bit reversals for parallel access of data is expanded at the system level module.

4.5.3 Handling Applications with Multiple Kernels

DATE system can handle multiple kernels an application working at the same time. Currently, the DATE system only support implicit type of HLL descriptions for these kernels. Figure 4.14 shows an example implementation of Reverse Time Migration (RTM) technique. More details on RTM can be found in a work done by Araya-Polo et al. [3]. This application consists of three main kernels, *The Stencil Computation*, *The Time Integration* and *The Boundary Point Computation*. The first four lines in Figure 4.14 show initializations of stencil size for the stencil kernel, stencil size for the boundary point computational kernel, input volume size and the extended volume size respectively. The next four lines of code shows initialization of constant coefficients needed for the three kernels. Lines 8 and 9 declare the stencil's FBT (specialized stencil memory template) for feeding data to stencil and boundary point computational kernels respectively. Lines 10 to 14 declare data streams and initialize them with the



```
1  const    P    =: {8,9,8};
2  const    B    =: {2,3,2};
3  const   Sv   =: {K,M,N};
4  const   Sev  =:{K+8,M+8,N+8};


5  T  cb{} =: {boundary_point_coefficients};
6  T  ct{}  =: {time_integration_coefficients};
7  T  cs{} =: {stencil_coefficients};


8  STENCIL <3D, T, P > SD898;  // 4-Point Stencil
9  STENCIL <3D, T, B > SD232;  // 1 Point Stencil

10  istream_channel   <T,Sev> In1;
11  istream_channel   <T,Sv>   In2,In3;
12  ostream_channel  <T,Sv>  Ost;
13  ostream_channel  <T,Sv>  Oti;
14  ostream_channel  <T,Sv>  Obp;

15  stream(In1.start ,In1.end, Sev, SD898);
16  Ost =: Compute_3DStencil (SD898,T,P,cs);
17  Oti  =: Compute_Time_Integration(Ost,In2,In3,T,ct);
18  stream(Oti.start ,Oti.end, Sv, SD232);
19  Obp=: Compute_Bounday_Point(SD232,T,B,cb);

20  stream(Obp.start ,Obp.end,Sv , OUT);
```

Figure 4.14: Implicit Type of HLL Description for RTM Implementation

sizes and data type T. The streams *In2* and *In3* are directly fed to the *Time Integration* kernel. The *stream* functions in lines 15,18 and 20 connect data streams to a memory or an I/O and allow moving data into them. Lines 16,17 and 19 incorporate the template expansions for the three kernels of the application.

4.5.4 Results and Discussion

The evaluation of the DATE System is done by using three abstract domain classes: FIR , FFT and Multidimensional (MD) Stencil. FIR and FFT abstractions are maintained in the Template Library for both kinds of templates (i.e. C-templates and HDL-templates). From the abstract class declarations for these example classes inside the HLL source code, the DATE system generates a C code and Direct HDL code using respective templates. The C codes generated for the abstract classes FIR and FFT are compatible with the ROCCC C to HDL tool. However to use the GAUT tool in our evaluation we do manual adjustments in the generated C codes to be able to correctly compiled by the GAUT tool. In the case of the MD Stencil, we need a special FBT architecture therefore the stencil domain keeps only the HDL template and generates the Direct HDL code.

The real benefit of the proposed DATE System is that it helps isolate the application programmers from the domain experts. This makes things easier for the application programmers to port their designs to FPGAs. In our view, measuring this *"ease"* can not be done realistically because of the possible huge variance in expertise between the application programmers. However, the Figure 4.15 shows *"ease"*

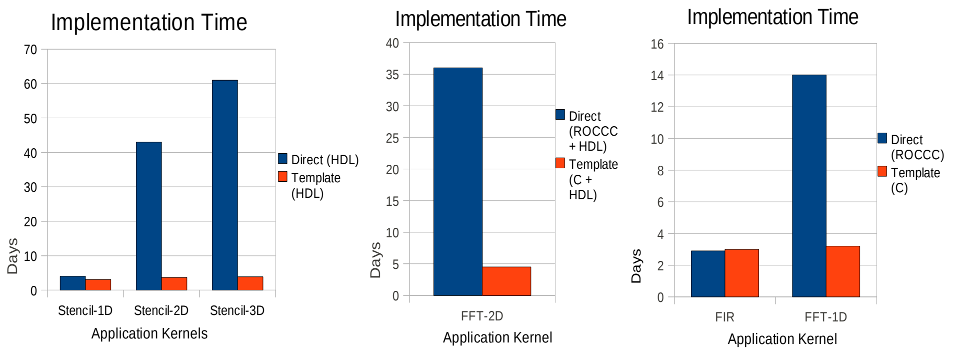


Figure 4.15: Application kernels Implementation Time for various cases

in terms of relevant development time in our case. The figure shows Normal and Template based implementations of the kernels. The Normal case in the Figure 4.15 represents implementations either using ROCCC or HDL while the Template based implementations use either C or HDL templates . Only in the case of 2D-FFT (for both Normal and Template based implementations), ROCCC is used to generate 1D-FFT HDL. This is true for the Normal C case and the C code generated from the C Template. The data management part is written in HDL (Normal HDL and HDL Template). Two instantiations of the HDL for 1D-FFT are used to integrate along with the special memory organization to generate a 2D-FFT architecture. The *implementation time* for the Normal implementations of different types of stencils are taken from our work on 3D-Stencil [32] while other timings are observed during the development of current work. The results shows that the template based design time is almost constant for all implementations. It is evident from the Figure 4.15 that the template system support can effectively increase the productivity of the application programmer to a great extent . The only exceptions may occur where the kernels are pretty small but still in such cases a novice to a C to HDL tool or the direct HDL programming can have potential benefits from using templates and template system.

Table 4.1 shows different template parameters (Bits for Data Types, TAPS, Points)

Table 4.1: DATE System Evaluations are done on Virtex-4 LX200 device using Xilinx ISE 11.1 tool suite. In-case of 2D-FFT, the table shows only results for the computational FFT (1D-FFT) part with templates for both C and HDL. The memory part uses less than 5% of Block RAMs (square frames) and less than 2% of slices to implement WR and RD toggling logic in our example case studies

				FIR			FFT			Stencil(MD)	
Bits	Taps, Points	Resource	ROCCC	GAUT	Direct	ROCCC	GAUT	Direct	Direct(1D)	Direct(2D)	Direct(3D)
16	6-TAPs (FIR) 8-Point(FFT) 2-Point(Stencil)	Slices	675	283	28	1383	5489	705	767	1180	1632
		DSP48	3	5	11	18	30	37	-	-	-
		BRAM	0	0	0	0	0	0	0	5	155
		Freq (MHz)	134	118	270	156	99	127	330	242	156
	12-TAPs (FIR) 16-Point(FFT) 4 -Point(Stencil)	Slices	958	674	67	7600	13868	4356	980	2275	3876
		DSP48	5	5	23	40	84	83	-	-	-
		BRAM	0	0	0	0	0	0	0	9	280
		Freq (MHz)	135	109	213	140	83	102	322	216	121
32	6 -TAPs (FIR) 8-Point(FFT) 2-Point(Stencil)	Slices	957	594	30	2628	12425	1740	1187	2420	4563
		DSP48	12	9	11	72	72	74	-	-	-
		BRAM	0	0	0	0	0	0	0	5	155
		Freq.	128	73	265	144	66	110	356	230	134
	12-TAPs (FIR) 16-Point(FFT) 4-Point(Stencil)	Slices	1529	1266	60	52056	43081	22500	2350	4751	7127
		DSP48	18	15	23	87	72	92	-	-	-
		BRAM	0	0	0	0	0	0	0	9	280
		Freq (MHz)	128	71	205	127	56	76	290	207	101

used as the test cases in DATE system evaluations for the correct expansions of the example domains. The HDL codes generated by ROCCC, GAUT and the DATE system are compiled by Xilinx ISE-11.1 tool suit for a Virtex-4 LX200 as the target device. Results for the usage of slices, DSP48 modules, BRAM Blocks and estimated frequency (after synthesis) are shown in Table 4.1. Our goal in this work is not to compare the performance of different C to HDL compilers or the manual implementations. This largely because most of the existing C to HDL compilers are based on certain heuristics – for source to source translations – adopted by each developer at its own in the absence of any standard. The optimizations performed during HDL generation process varies a lot depending on the style in which the original code is written. A minor variation in the C-code can drastically change the performance (execution time, power consumption or area usage etc) of the resultant HDL. Therefore, the idea of comparing the performance or speedup from HDLs generated by different C to HDL compilers for the same C-code is simply not valid and the results would be misleading. This fact is also highlighted by Sarkar et al. in a recent study [71] on various HLS (High Level synthesis) tools.

Our work presents a case study and shows the potential of the DATE system for dealing with codes including HLL domain abstractions. However, from the data in Table 4.1, an interesting observation is that the ROCCC tool generates architectures with lesser variations in the operational frequencies. This means it generates balanced pipelined architectural designs. HDLs generated from HDL based templates shows that the *Overload Identifiers* use larger number of of DSP48 modules as compared to the other tools. This can be a better approach to run systems at higher frequencies. However, a balance in resources would be necessary when compiling multi-kernel domain abstractions. BRAMs are not used by any kernel except the Direct-HDLs for the special front-end memory architecture of MD Stencil and 2D-FFT. The front-end memory templates presented here are only for the generation of FBTs as specialized front-end architectures. Therefore the DSP48 modules are not expected to be generated in these case.

4.6 Summary

Domain abstractions are an efficient way of handling complex applications. In addition, these enable high performance by keeping the developers from handling low level system details. In this work we present a template system (DATE) which uses domain abstractions for reconfigurable accelerator designs. This work is a step towards making the accelerator designs highly customized and more efficient. Our approach has the potential to support the scalability of the architectural designs by just varying few input parameters. This also enables the portability of accelerator architectures to various sizes of small and large FPGA devices. Further, the standard output generated by the DATE system makes it platform independent. We have evaluated the system for six kernels from three example abstract domains (FIR, FFT and a special Stencil Architecture) and their expansions. The results are promising and motivate further research in supporting application complexity and performance using the current approach.

This chapter briefly discuss how DATE system can be very helpful to generate a complete system based on TARCAD, a standard architecture template for custom reconfigurable accelerators. The next chapter of this document extends the study on TARCAD for the complete template based accelerator designs.

5

A Throughput Oriented Template Based Streaming Accelerator

In the race towards computational efficiency, accelerators are achieving prominence. Among the different types, accelerators built using reconfigurable fabric, such as FPGAs, have a tremendous potential due to the ability to customize the hardware to the application. However, the lack of a standard design methodology hinders the adoption of such devices and makes the portability and reusability across designs difficult. In addition, generation of highly customized circuits does not integrate nicely with high level synthesis tools.

In this work, we introduce TARCAD, a template architecture to design reconfigurable accelerators. TARCAD enables high customization in the data management and compute engines while retaining a programming model based on generic programming principles. The template provides generality and scalable performance over a range of FPGAs. We describe the template architecture in detail and show how to implement five important scientific kernels: MxM, Acoustic Wave Equation, FFT, SpMV and Smith Waterman. TARCAD is compared with other High Level Synthesis models and is evaluated against GPUs, a well-known architecture that is far less customizable and, therefore, also easier to target from a simple and portable programming model. We

[1] Chapter 5 is based on the publications:
(1) TARCAD: A Template Architecture for Reconfigurable Accelerator Designs; Muhammad Shafiq, Miquel Pericas, Nacho Navarro, Eduard Ayguade; appeared in IEEE Symposium On application Specific Processors, San Diego, CA, 2011
(2) PPMC : A Programmable Pattern based Memory Controller Tassadaq Hussain, Muhammad Shafiq, Miquel Pericas, Nacho Navarro, Eduard Ayguade appeared in The 8th IEEE/ACM International Symposium on Applied Reconfigurable Computing, March 2012, Hong Kong

analyze the TARCAD template and compare its efficiency on a large Xilinx Virtex-6 device to that of several recent GPU studies.

5.1 Customized Accelerators

The integration levels of current FPGA devices have advanced to the point where all functions of a complex application kernel can be mapped in a single chip. However, these high density FPGAs appear just like a sea of logic slices and embedded hard IP resources such as general purpose processors, multi-ported SRAMs and DSP slices. The final performance depends on how well the FPGA application designer maps an application to the device. This practice is problematic for several reasons. First, it is a low-level approach that requires a great deal of effort for mapping the complete application. Second, reusability of modules across projects is significantly reduced. And, last but not least, it is difficult to compare hardware implementations that adhere to different high-level organizations and interfaces. This emphasizes the need to abstract out these particular hardware structures in a standard architectural design framework.

The tremendous potential of reconfigurable devices to exploit both the customized data layout inside the local memory and the intrinsic parallelism of an algorithm has attracted many application designers to design accelerators on top of these devices. As a result, a plethora of application kernels from the HPC domain have been ported to these devices. However, most designs are tied to a specific environment due to the lack of a standard design methodology. In fact, this is a long standing challenge and the future reconfigurable devices will not become mainstream accelerators if they are unable to solve the implementation of applications in a well defined, simple and efficient way. The architectures generated by HLS (High Level Synthesis) tools (e.g. ROCCC [77] or GAUT [58]) also follow a simplified and standardized compilation target, but they have been designed specifically as compiler targets, which limits their applicability to HDL designers. In addition, these models are too constrained to support complex memory organizations or unorthodox compute engines which are often required to best exploit FPGAs.

This work is a step towards the harmonization of data-flow architectures for various FPGA-based applications written in HDLs (e.g. Verilog, VHDL) and HLLs (High Level Languages). We propose an architectural template named TARCAD that allows to efficiently exploit FPGAs using a simple programming methodology. The methodology uses a retargetable template expansion system based on metaprogramming techniques called DATE [12]. TARCAD not only enables HDL designers to work on a highly customizable architecture, it also defines a set of interfaces that make it attractive as a target for an HLS compilation infrastructure.

TARCAD defines both a high-level model for the computation flow as well as a strategy for organizing resources, managing the parallelism in the implementation, and facilitating optimization and design scaling. Following DeHon's taxonomy, these two correspond to the fields of compute models and system architectures [78]. Similarly to TARCAD, Coarse-Grained Reconfigurable Architectures (CGRA) also define stricter compute models and system architectures. PipeRench [79], MUCCRA [80] or ADRES [81] are examples of CGRA architectures. A related architecture is the so-called Massively Parallel Processor Arrays (MPPA), which are similar to CGRAs, but include complete, although very simple, processors instead of the functional units featured within CGRAs. PACT-XPP [82] is an example of a MPPA-style architecture. Defining a compute model and a system architecture are not only specific to chip design. Several efforts have concentrated on defining environments in which to accommodate FPGA chips. Kelm et al. [83] used a model based on local input/output buffers on the accelerator with DMA support to access external memory. Brandon et al [84] proposes a platform-independent approach by managing virtual address space inside their accelerator. Several commercially available machines like the SGI Altix-4700 [37] or the Convey HC-1 [85] propose system level models to accelerate application kernels using FPGAs. These models combine a CPU with one or multiple FPGAs running over a system bus. Another option is to integrate CPU and FPGA directly in a single chip. Several research projects have covered this possibility. In the Chimaera architecture [86], the accelerator targets special instructions that tell the microprocessor to execute the accelerator function. The accelerator in Molen processor [87] uses some exchange registers which get their data from processor register file.

This chapter discusses the generic architectural layout of the TARCAD template for reconfigurable accelerators. The proposed architecture is based on the decoupling of the computations from the data management of the application kernels. This makes it possible to independently design specialized architectures for both parts of the kernel in a data-flow envelope supported by our architectural layout. Computation scales depending on the size of the FPGA or the achievable bandwidth from the specialized memory configuration that feeds the compute part. TARCAD can also be a basis to develop a Reconfigurable GPU-like architecture under a streaming programming model. This new GPU can be highly efficient in its memory accesses by using a fully configurable front-end for custom memory layouts. However, the study of this architecture is left as future work. The current work evaluates the architectural efficiency of an FPGA device for several applications using TARCAD and compares it with GPUs. This is an interesting comparison because both platforms require applications with data level parallelism and control divergence independent kernels.

5.2 The TARCAD Architecture

5.2.1 Accelerator Models for Supercomputing

The TARCAD proposal targets both HDL accelerator designers by providing them with a standard accelerator design framework, as well as HLS tool developers by giving them a standard layout to map applications on. Furthermore, TARCAD can act as a top model to design new ASICs or the aforementioned Reconfigurable GPUs. HLS tools (e.g. ROCCC [77] and GAUT [58]) define an architectural framework into which they map the algorithmic descriptions. The basic compute model for ROCCC requires streaming data inputs from an external host. This data is stored in *smart buffers* before being consumed by the compute units and then again before being sent back to main memory. The GAUT architecture, on the other hand, provides an external interface to access data based on data pointers. The memory model of GAUT is simple and can keep large chunks of data using BRAM as buffer memory. GPUs are another architecture that is nowadays highly popular. GPUs use their thread indexes to access data from up to five dimensions. A large number of execution threads help hiding external memory data access latencies by allowing threads to execute based on data availability.

A simple high level view of TARCAD is shown in Figure 5.1. The microarchitectural details of the TARCAD layout are presented in Figure 5.2. It is evident from the figures that the TARCAD layout can be partitioned into four representative main blocks and their constituent sub-blocks. A detailed description for these main blocks

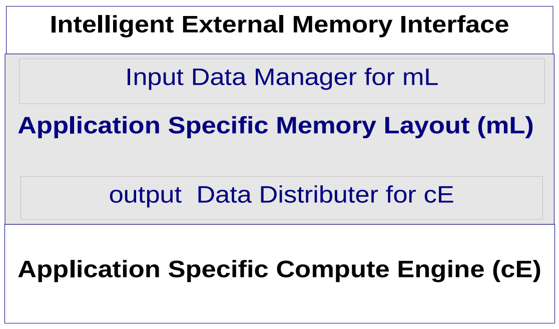

Intelligent External Memory Interface
Input Data Manager for mL
Application Specific Memory Layout (mL)
output Data Distributer for cE
Application Specific Compute Engine (cE)

Figure 5.1: A High Level View of TARCAD

(*External Memory Interface*, *Application Specific Data Management Block*, *Algorithm Compute Back-End* and the *Event Managing Block*) follows.

5.2.2 The External Memory Interface

In general, the nature of accelerators is to work on large contiguous data sets or streams of data. However, data accesses within a data set or across multiple data sets from an algorithm are not always straight forward. Therefore, accelerators can be made more efficient by providing some external support to manage the data accesses in a more regular way. TARCAD supports a Programmable Memory Controller (PMC) as an external interface to the main memory. This controller is inspired from work done by Hussain et al. [73]. It helps to transfer pattern-organized blocks of data between the accelerator and the global memory. Among different options, PMC improves the accelerator kernel performance by providing programmable strided accesses. This makes it possible for PMC to directly handle 1D, 2D and 3D tiling of large data sets rather than doing the same in software at the host processor.

5.2.3 The Application Specific Data Management Block

TARCAD's application specific management block helps arranging data for efficient usage inside the computations. This block consists of four sub-blocks identified in Figure 5.2 as *Data-Set (DS) Manager*, *Configurable Memory Input Control*, *Algorithm Specific Memory Layout* and the *Programmable Data Distributer*. Out of these sub-blocks, the *Algorithm Specific Memory Layout (mL)* plays a central role in designing an efficient accelerator by providing rearrangement and reuse of data for the compute blocks. The memory layouts can be common for various applications as shown by Shafiq et al. [11]. TARCAD can also adopt a similar common memory layout but in this chapter we only consider that a memory layout for an application is customized using the block RAMs (BRAMs) of the device.

The data writing pattern to a customized memory layout can be very different from the reading pattern from the same layout. Let us consider a simple example of MxM in which PMC can access matrices in row-major order from the external memory. In the case of data for the multiplicand matrix, the internal memory layout could either be written in the same row-major order followed by reading in the transposed (column-major) order from the memory layout or vice versa. Therefore, TARCAD keeps separate write and read interfaces (*CFG MEM-IN-CONTROL* and the *Programmable Data Distributer*) to the *memory layout* block as shown in Figure 5.2. The preset FSM based memory input control expects various streams of independent data sets through the streaming FIFO channels (*DS-ix*). Each of the *DS-ix* can have multiple sub-channels to consume the peak external bandwidth. However, all sub-channels in a *DS-ix* represent the same data set.

The *Data-Set Manager* provides a command data interface between the reconfigurable device and the external-to-device PMC unit. This *Data-Set Manager* helps to

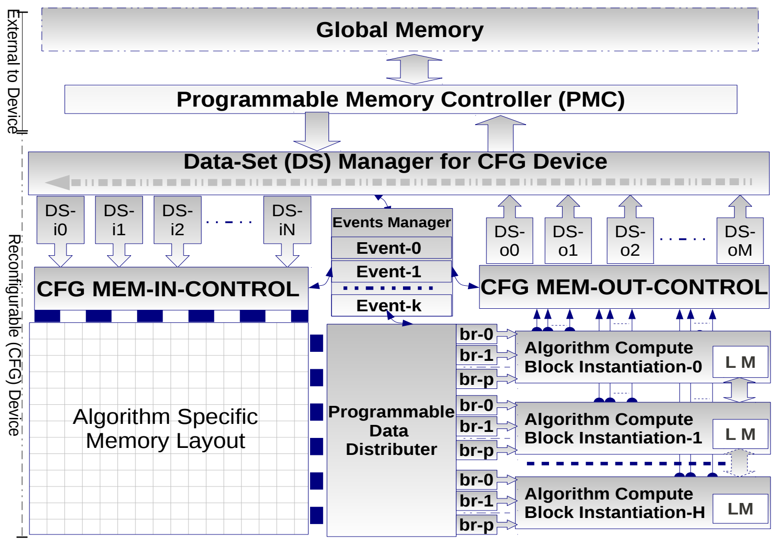

External to Device
Global Memory
Programmable Memory Controller (PMC)
Data-Set (DS) Manager for CFG Device
Reconfigurable (CFG) Device
DS-i0
DS-i1
DS-i2
DS-iN
Events Manager
Event-0
Event-1
Event-k
DS-o0
DS-o1
DS-o2
DS-oM
CFG MEM-IN-CONTROL
CFG MEM-OUT-CONTROL
Algorithm Specific Memory Layout
Programmable Data Distributer
br-0
br-1
br-p
Algorithm Compute Block Instantiation-0
Algorithm Compute Block Instantiation-1
Algorithm Compute Block Instantiation-H
L M
LM

Figure 5.2: TARCAD architectural layout

fill the *DS-ix* streaming FIFOs. On the reading side of the memory layout, the *Programmable Data Distributer* is used. This data distributed is also a FSM. However, it is programmable in the sense of distributing different sets of data to the different instantiations of the same compute block (see Section 5.2.4).

5.2.4 The Algorithm Compute Back-End

The compute Back-End consists of the *Branch-Handlers*, *Compute Block Instantiations* and *Configurable Memory Output Control.* The compute block is the main part of this Back-End and it can have multiple instantiations for an algorithm. Each instantiation of the compute block interfaces with the programmable data distributer through its *Branch-Handler*. These *Branch-Handlers* are similar to FIFO buffers. They support data prefetch and help to reduce the time penalty in case of branch divergence in the compute block.

The TARCAD architecture expects a compute block as a combination of arithmetic compute units with minimal complexity in the flow of data inside the compute block. All compute blocks either keep a small set of their computational results in the local memory (LM) shareable with other instantiations or forward the results to *configurable memory output control (CFG MEM-OUT-CONTROL). CFG MEM-OUT-CONTROL* collects data from the compute blocks for specific set of output data set (*DS-Ox*). The results collected at *CFG MEM-OUT-CONTROL* are either routed back to the global memory by the *Data-Set Manager* or written back to the *CFG MEM-IN-CONTROL.*

5.2.5 The Event Managing Block

The role of *Event Manager* is to guide and monitor the kernel mapped on TARCAD. The *Event Manager* can be a FSM or a simple processor with multiple interrupt inputs. Here we consider the Event Manager to be a FSM. In general each event in the *Event Manager* guides and monitors any single phase of kernel execution. The event manager is initialized by the user before the execution of a kernel. It holds information like the set of events (signals from various blocks) for each phase, input/output memory pointers and the data sizes for different data sets used in the execution of each phase of a kernel. The *Event Manager* monitors the execution of the kernel and takes actions at the appropriate event. The actions are in the form of exchanging information (setting/getting state data by the event manager) with all the other state machine based blocks. The *Event Manager* keeps a set of counters shared in all phases while a set of registers for each phase initialized by the user.

5.3 TARCAD Implementation

The motive behind the TARCAD layout is to support efficient mapping of application specific accelerators onto the reconfigurable devices. Therefore, these specific mappings of various designs require to physically change or scale the data paths, FSMs, the special memory layouts and the compute blocks. These changes for a reconfigurable device can be made only at compile time. Therefore, we are propounding the implementation of TARCAD using a template expansion method. This is a template metaprogramming process that generates a specific HDL of the accelerator based on the TARCAD layout. The template expansion is provided by our prototype translator called *Design of Accelerators by Template Expansion (DATE)* system [12]. This is an in-house research tool to support template based expansions for high level domain abstractions. The simple block diagram in Figure 5.3 shows the flow of the *DATE* system. The main inputs from the user to the DATE system are annotated HDL based template code for the compute block and the data flow definitions for the memory layout. The annotations used in coding the HDL are similar to those used in the DATE templates [12]. A set of parameters is also passed to the *DATE* translator to adjust and generate other HDL design modules by using the TARCAD templates for various blocks maintained inside the TARCAD template library. For example, some important parameters related to the *Event Manager* are the total number of phases through which a kernel will execute, the total repetitions of a phase, the maximum number of events connected to that phase, the total number of data pointers used in the phase and the equations for memory block accesses for each of the pointers in the phase. However, the actual list of data pointers, the monitoring and activation events and the event's target blocks are initialized using special commands directly by the *Data Set manager* at the execution startup or during the runtime.

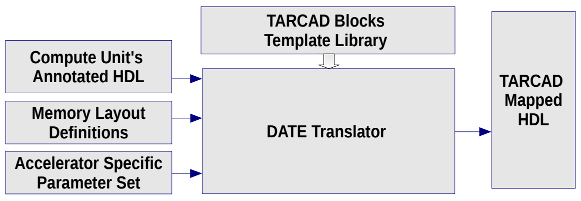

TARCAD Blocks Template Library
Compute Unit's Annotated HDL
Memory Layout Definitions
Accelerator Specific Parameter Set
DATE Translator
TARCAD Mapped HDL

Figure 5.3: TARCAD Implementation: Environment of the DATE System

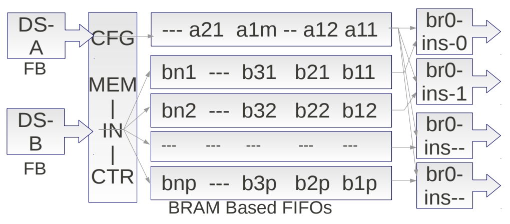
5.4 Application Kernels on TARCAD

The TARCAD layout can be mapped for all kinds of application kernels. The following section presents some example application kernels mapped on TARCAD.

5.4.1 Matrix-Matrix Multiplication (MxM)

Matrix-Matrix multiplication offers numerous design possibilities. Here we use a memory layout and compute block which are efficient for large sized matrices. The matrices are accessed in the same "row major order" from the external memory.

As shown in the Figure 5.4 (a), matrices A and B are fetched in the order of one row and multiple columns. The process of fetching matrices' data and writing the results back is managed by the *Event Manager* with the help of the *Data Set Manager* and *CFG MEM-IN/OUT-Controls*. A small piece of pseudo code which represents the *Event Manager* FSM for the data fetch requests is shown in Figure 5.4 (b). In order to make it clear, the FSM actions are non-blocking (i.e simultaneous but based on conditions) and the purpose of the sequential pseudo code is just to give the basic idea of the mechanism. The structure of this FSM already exists as a template in the *DATE Translator library* (Figure 5.3). However, an arbitrary number of registers to keep kernel specific information are created from the parameterized information at translation time. For example in Figure 5.4 (b), ISa and ISb are registers created for the initial source pointers to access matrices from external memory. FSa and FSb are the tuple registers for the fetch source pointers (the current pointers). FSaz and FSbz represent the registers for the fetch sizes of data. The source size registers are mentioned as SSra and SSmb. The external parameters to the *DATE System* also include simple equations to generate data accesses in big chunks, like "$FSa = ISa + i \times SSra$" where "i" is an internal incremental variable. The parameterized inputs also create two



(a)

```
1  ISa    = A_pointer
2  ISb    = B_pointer
3  SSra   = A_row_size
4  SSmb = B_matrix_size
5  loop(EVre) :
6  if (EVrr) : i=0 ; i++
7  FSa   = ISa + i x SSra
8  FSaz = SSra
9  FSb   = ISb
10 FSbz = SSmb
11 end_if
12 end  loop
```

(b)

Figure 5.4: MxM : (a) Matrices elements' distribution into application specific memory layout and (b) Pseudo code for matrices data accesses by the *Event Manager*

events, the "row request event" (EVrr) and the "rows end event" (EVre) coming from the *CFG MEM-IN-Control* and *CFG MEM-OUT-Control* respectively. These events are monitored by the *Event Manager*.

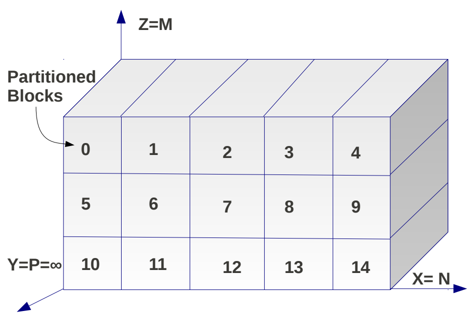
At runtime, the FSM of the *Event Manager* corresponding to the pseudo code shown in Figure 5.4 (b) initializes the registers ISa, ISb, SSra and SSrb. This is done by using special initialization commands from an external host. These commands are decoded by the *DATA Set Manager* and forwarded to the *Event Manager*. The *DATA Set Manager* can also hold multiple requests from the *Event Manager* and forward these requests consecutively to the programmable memory controller (PMC). As in lines 5 and 6 of the pseudo code, the *Event Manager* monitors the event signals EVrr and EVre and sends the tuples of data for the external memory fetch pointers and their sizes to the *Data Set Manager* along with necessary control signals. This starts the fetching of data by the PMC from both matrices A and B from external memory. The physical data transactions are directly handled by the *Data Set Manager* and the *CFG MEM-IN/OUT-Controls*. The FSMs at *CFG MEM-IN/OUT-Controls* are also built based on their own parameterized information and take care of the generation of the events EVrr and EVre at the appropriate execution time.

During the run, one row of Matrix A is fetched from the external memory into a single circular buffer and used element by element in each cycle while the fetched row from Matrix B is scattered around the multiple circular buffers equal to the number of compute block instantiations in the back-end. Therefore, the dot product of an element from the row of Matrix A is done with multiple columns of Matrix B. Each instantiation of the compute block accumulates the results for the element wise dot product of a row (Matrix A) and a column (Matrix B).

5.4.2 Acoustic Wave Equation (AWE)

A common method to solve the Acoustic Wave Equation (AWE) numerically consists of applying a stencil operator followed by a time integration step. A detailed description of the AWE solver and its implementations is provided by Araya et al. [3]. In our TARCAD based mapping of the AWE solver, the two volumes of previous data sets for the time integration part are forwarded to the compute block by using simple FIFO channels in TARCAD's memory layout. Our implementation of the stencil operations follows the memory layout of an $8 \times 9 \times 8$ odd symmetric 3D stencil as shown by Shafiq et al. [32].

In our TARCAD based mapping of the AWE kernel, we consider real volumes of data that are normally larger than the internal memory layout of the accelerator. Therefore, a large input volume is partitioned into its sub-volumes as shown in Figure 5.5 (a). A sub-volume block also needs to copy the so-called "ghost points" (input points that belong to the neighboring sub-volume). For example, Block 7 shown in Figure 5.5 (a) needs to be fetched as an extended block that includes ghost points from the neighboring Blocks 2, 6, 12 and 8. However, these ghost points are only required for the one volume being used in stencil computations.



```
ISa    = V1_pointer
ISb    = V2_pointer
ISc    = V3_Pointer
BnV1=BnV2=BnV3= 0
loop(EVbe) :
if (EVrb) :
FSa     = ISa
FBv1   = BnV1++
FSb     = ISb
FBv2   = BnV2++
FSc     = ISc
FBv3   = BnV3++
end_if
end_loop
```

(a) (b)

Figure 5.5: Odd symmetric 3D stencil: (a) Large input volume partitioned into sub volumes (b) Pseudo code for sub-volume accesses by the *Event Manager*

The TARCAD layout supports offloading the management of block-based data accesses to the programmable memory controller (PMC). In the AWE case, for simplicity, TARCAD accesses the same pattern of the extended sub-volumes for all three input volumes. The *CFG MEM-IN-CONTROL* discards the ghost points accessed for the two volumes used in time integration. The PMC is programmed by the host to access the three volumes of data –block by block– on the request of the *Event Manager*. The example pseudo code for the FSM of the *Event Manager* is shown in Figure 5.5 (b).

In the first three lines of the pseudo code, the FSM initializes the initial source pointers (ISx) for the three input volumes. In the next line, a reset to zero of block counts (BnVx) for the sub volumes is done. Similar to the MxM kernel case, the *Event Manager* of AWE monitors two events. One event, "Block Ends" (EVbe), is sourced from the *CFG MEM-OUT-CONTROL* and ends the execution of the kernel while the other event "Block Request" (EVbr) comes from the *CFG MEM-IN-CONTROL* and initiates a new request of the block. Inside the control structure, the FSM updates three tuples of parameters corresponding to the three input volumes. Each tuple consists of the base pointer of the volume (FSx) and the block number (FBvx). These tuples of data are used by the *Data-Set Manager* to access external data through the programmable memory controller. The flow of data between the *Data-Set Manager* and *CFG MEM-IN-CONTROL* is synchronized with handshake signals between the two interfaces.

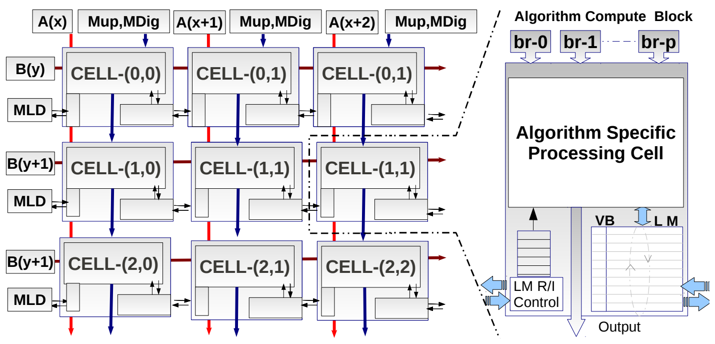


Figure 5.6: Smith Waterman : Left: The Systolic array of compute blocks, Right: Architectural support for inter-compute block communication.

5.4.3 Smith Waterman (SW)

The implementation of Smith Waterman algorithm results in a systolic array of processing cells. This kind of data flow is also well suited to map on the compute blocks of the TARCAD architecture. The left part of Figure 5.6 shows the TARCAD-based systolic array of processing cells that results from joining a number of compute blocks to run the SW kernel. Each of the compute blocks consists of an algorithm specific processing cell. This processing cell, in our case, consists of the Smith Waterman compute architecture proposed by Hasan et al. [88]. The input data for a compute block constitutes only a single branch set that consists of A_x, B_y (the two sequences) and M_{up}, M_{Diag} (the top and diagonal elements) from the similarity matrix. M_{LD} represents the current data passed through the LM to the next compute block as left side's Matrix M data. This data word is also passed in stair case flow to be used as a diagonal data element.

The generic layout of the compute block in TARCAD is shown in Figure 5.6 (Right). Each compute block keeps a dual ported local memory (LM) for low-latency communication of data with other compute blocks. Each word of this local memory is also accompanied by a valid bit which describes the validity of the data written to it. This valid bit is invalidated by the receiving compute block. In case the receiving blocks are more than one then only one of them can drive the invalidation port of the source compute block and others work synchronous to it. Inside a compute block the LM is written as a circular buffer, therefore, the invalidation of the valid bit does not create any read/write hazards for few (equal to number of words in LM) consecutive cycles for the LM data between the source and destination. The width and depth of the LM is parameterized and it can be decided at translation time. Moreover, each compute block also has a local memory read and invalid control (LM R/I Ctrl) for reading and invalidating a word of the source block's LM. The read word is placed into a FIFO which is readable by the compute block's algorithm specific processing cell.

5.4.4 Fast Fourier Transform (FFT)

The TARCAD layout is flexible and can also integrate with third party cores. For the FFT case, we show in Figure 5.7 how TARCAD interfaces with an FFT core generated by Xilinx CoreGen [70]. TARCAD interfaces and controls the single or multiple

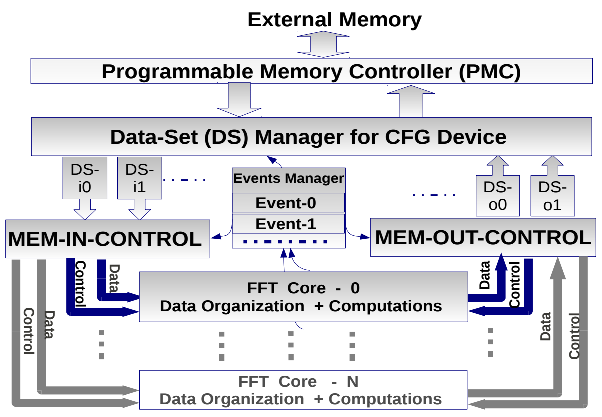

External Memory
Programmable Memory Controller (PMC)
Data-Set (DS) Manager for CFG Device
DS-i0
DS-i1
Events Manager
Event-0
Event-1
DS-o0
DS-o1
MEM-IN-CONTROL
MEM-OUT-CONTROL
Control
Data
FFT Core - 0
Data Organization + Computations
FFT Core - N
Data Organization + Computations

Figure 5.7: Mapping an existing FFT core on TARCAD

input/output streams of data corresponding to one or more instantiations of the FFT cores.

5.4.5 Sparse Matrix-Vector Multiplication (SpMV)

In our TARCAD based mapping of the SpMV kernel, we use an efficient architecture that is based on a row interleaved input data flow described by Dickov et al. [89]. TARCAD's FSM in *CFG MEM-IN-CONTROL* uses a standard generic Sparse Matrix format and converts it internally to the row-interleaved format before feeding to the compute block. However, this methodology needs to know in advance (at translation phase), the maximum possible number of non- zero elements in any row of the matrix. This information helps the translator to correctly estimate the maximum number of rows possible to decode and maintain inside the SpMV memory layout.

5.4.6 Multiple Kernels On TARCAD

TARCAD can handle multiple algorithms working at the same time. In general, each algorithm should be maintained with separate data paths, memory organization and the compute units. Only data requests to the global memory (through the *Data-Set Manager*) are shared. However, design schemes like a spatially mapped, shared memory layout as shown by Shafiq et al. [11] can help to use shared data for certain kernels with different types of compute block instantiations.

5.5 Evaluations

To evaluate TARCAD, we simulate the mappings of various application kernels as presented in section-5.4 by using a Xilinx Virtex-6 XC6VSX475T device. The HDL designs were placed and routed using the Xilinx ISE 12.4 environment. The Virtex-6 device used in our evaluations has a very large number (more than 2K) of DSP48E1 modules. Therefore, we did maximum possible instantiations of the compute blocks for the kernels and used the device's maximum operational frequency after place-and-route for all the back-end instantiations. The external memory support for TARCAD is dependent on the board design. Xilinx Virtex-6 FPGAs can achieve a maximum external data bandwidth of 51GB/s [90]. However, in our simulated evaluations for TARCAD we assume an aggressive external memory interface with multiple memory controllers, providing an aggregate peak bandwidth between 100GB/sec to 144GB/s. This simulated external memory interface performance is similar to what can be achieved today by GPUs.

In our evaluations, the efficiency of the application kernels mapped on the TARCAD layout are compared with state of the art implementations of the same kernels on various GPU devices. The choice of the GPU based implementation is based on two points. One is that the GPU implementation is selected out of the available ones for the best possible GPU device and second, we are able to reproduce the same input test data for the TARCAD based implementations. The architectural efficiencies shown in Figure 5.8 are defined differently for the kernels using floating point computations and cell updates. These efficiencies are computed using Equations 5.1 and 5.2.

$$EFF_{flops} = FLOPS_{total}/FLOPS_{max} \tag{5.1}$$

$$EFF_{cups} = CUPS_{total}/Freq_{opr} \tag{5.2}$$

In Equations 5.1 and 5.2 *total* refers to the achieved FLOPS or CUPS for an application while *max* and *opr* represent the maximum FLOPS for a device and the operational frequency of the device, respectively.

Table 5.1: Applications Mapped to TARCAD using Virtex-6 & ISE 12.4

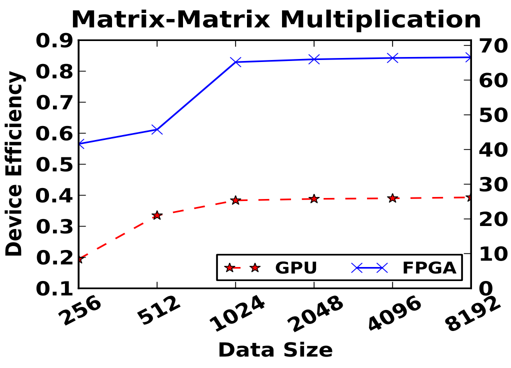
Applications	Compute Blocks	Freq (MHz)	DSP48E1	Slices	BRAMs (36Kb)
MxM	403	105	2015	49757	432
AWE Solver	22	118	2008	45484	677
SW	4922	146	2012	63989	85
FFT	4-48	125	472-2016	48K-59K	0-1060
SpMV	134	115	2010	33684	516

5.6 Results and Discussion

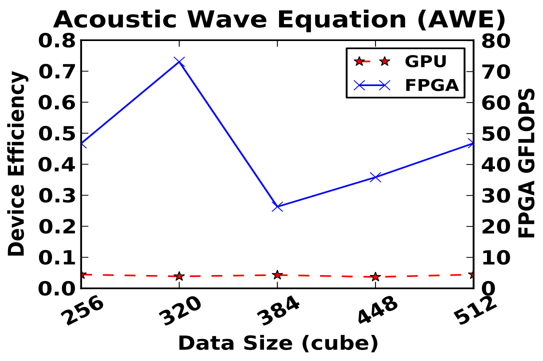
The overall performance (Figure 5.8 (a-e : Right Y-axis)) for various kernels mapped on TARCAD remained lower than 100 GFlops (or GCUPS for SW). This is considerably lower than that for the reference performances on GPUs. It is important to note that the right-axis of the plots in Figure 5.8 only corresponds to the FPGA performance and the corresponding GPU performance can be seen in reference implementations for MxM [2], AWE [3], SW [4], FFT [2] and SpMV [5]. The lower performance for TARCAD based implementations of these kernels is an expected phenomena as the current reconfigurable technology operates at an order of magnitude lower operational frequency (see Table-5.1) for the mapped designs. However, if we look at the efficiency of the TARCAD mapped applications, these are quite promising due to the customized arrangement of data and compute blocks. In the following we will discuss efficiency of each kernel. In support to the discussion, the total number of compute units instantiated along with their operational frequencies and the usage of chip resource are given in Table 5.1. The numbers for FFT correspond to the implementations for 128 points to 65536 points and frequency is chosen for the lowest value.

5.6.1 Matrix-Matrix Multiplication (MxM)

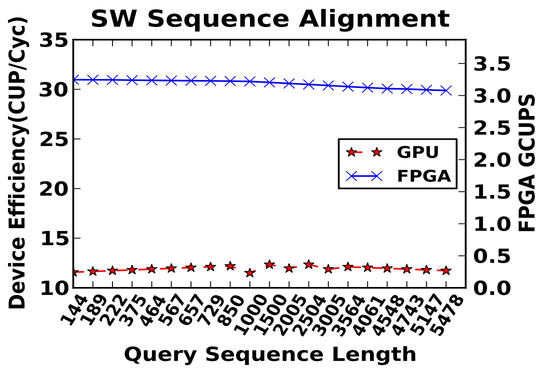
In the case of MxM, we can observe from Figure 5.8 (a) that the efficiency of the TARCAD-based implementation is on average 4 times higher than that for GPU. However, for smaller size of matrices the efficiency is relatively lower because of two factors: The first case occurs when the number of columns in Matrix B is less than 403 (total compute block instantiations). The second case occurs when the number



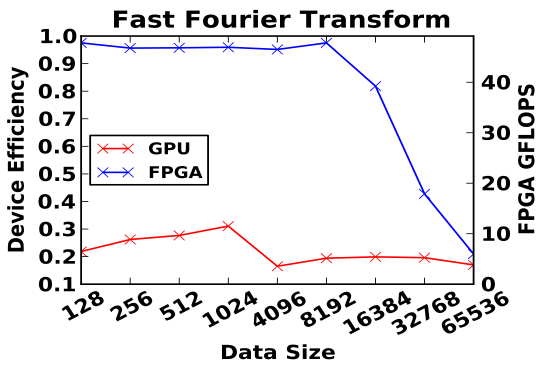
(a)



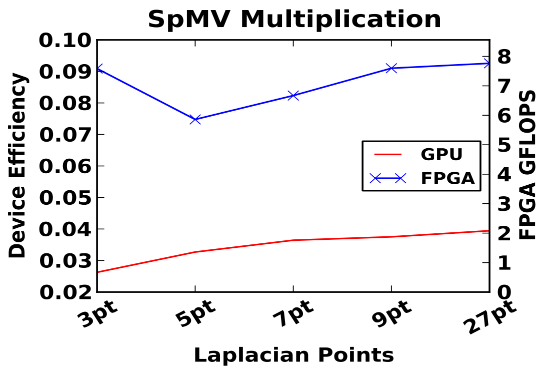
(b)



(c)



(d)



(e)

Figure 5.8: Architectural Efficiency for TARCAD and GPU based Kernels. Performance Numbers are only shown for FPGA based Designs. The device used for FPGA is Virtex-6 XC6VSX475T and the corresponding GPUs are (a) MxM (GPU: Tesla C2050 [2]) (b) AWE (GPU: Tesla C1060 [3]) (c) SW (GPU: Tesla C1060 [4]), (d) FFT (GPU: Tesla C2050 [2]) (e) SpMV (GPU: GTX 280, Cache Enabled [5])

of columns are not multiples of 403. Both cases result in unoptimized usage of the available compute units on TARCAD.

5.6.2 Acoustic Wave Equation (AWE)

The TARCAD mapped memory layout for the AWE kernel can handle sub-volumes of size $320 \times 320 \times \infty$ in the Z, X and Y axes respectively. The results for AWE (Figure 5.8 (b)) show that TARCAD-based AWE kernel efficiency reaches 14 times that of the GPU based implementation. However, then it drops to 5× for 384-point 3D volumes. This is because 384 is not the multiple of the basic size ($320 \times 320 \times \infty$) for AWE managed specialized memory layout and suffers huge data and computational overhead. However, this penalty starts reducing with an increase in the size of the actual input volumes.

5.6.3 Smith-Waterman (SW)

The Smith Waterman's implementation on TARCAD is approximately 3 times (Figure 5.8 (c)) more efficient than the referenced GPU-based efficiency. In fact, this edge in architectural efficiency of TARCAD is only a result of the customized mapping for the computing cells and the systolic array. The front-end data management only takes care to buffer new sequences for comparison or for feeding back the results from the cells on the boundary of the systolic array through *CFG MEM-OUT-CTRL*, *Data Set Manager* and *CFG MEM-IN-CTRL* path.

5.6.4 Fast Fourier Transform (FFT)

The memory requirement for the floating point streaming based implementation of Xilinx's FFT core increases rapidly for larger number of points. In the case of the TARCAD-based mapping, the instantiations of the FFT kernel for 16384 or larger points are limited by the total available BRAM of the device. This limitation is accordingly apparent from the plot shown in Figure 5.8 (d). However, for the lower number of points (equal or lower than 8192), the instantiations of FFT compute blocks are dictated by the total number of DSP48E modules available on the device.

5.6.5 Sparse Matrix-Vector Multiplication (SpMV)

In the SpMV mapping on TARCAD, we modified the original design of Dickov et al. [89] to a special yet generic compute block for handling any kind of laplacian data. This design handles a three point front-end which accumulates three dot products at a time from a row. However, the inefficiencies (Figure 5.8 (e)) for this laplacian specific compute block appear when the non-zero diagonals in the laplacian matrix are not a multiple of 3.

5.7 Summary

In this chapter we have presented our developments towards a unified accelerator design for FPGAs that improves FPGA design productivity and portability without constraining customization. The evaluation on several scientific kernels shows that the TARCAD template makes efficient use of resources and achieves good performance. In this work we have focused on showing how efficient architectural mapping can be achieved for HDL-based designs. Our TARCAD design also targets adoption by High Level Synthesis tools as a main goal in order to provide interoperability and high customization to such tools.

Although we have shown that TARCAD is more efficient than GPUs, final performance is often worse due to the slower operational frequencies of FPGAs. Designing a reconfigurable GPU based on the TARCAD architecture is an interesting idea to improve the final performance. However, the challenge is how to evaluate such a hybrid GPU. This motivates the development of simulator for GPU like streaming architecture. The next chapter of this thesis document present a complete simulation framework developed for GPU like streaming devices.

Part III

Design Space Explorations for Throughput Oriented Devices with Standard and Customized Memories

6

A Simulator Framework for Performance Characterization of Streaming Architectures

Streaming architectures like GPUs and reconfigurable devices with application specific designs are offering an interesting solution for high performance parallel computing. However, a lack in the availability of easy to experiment simulation tools for these streaming devices has severely restricted the researchers in computer architecture from vast level of explorations in this direction.

In this work, we present a framework for a trace driven simulator (*SArcs: **S**treaming **Arc**hitectural **S**imulator*) targeting GPU like devices. Our proposed framework functions as a standalone system. It uses GPU performance modeling based on runtime CPU code explorations. Therefore, it does not require its users to have any kind of GPU environment. By using our simulation framework, researchers can perform new architectural explorations or just go for a performance estimations for their applications by configuring the simulator specific to a target device. To the best of our knowledge *SArcs* is the first trace-based GPU architectural simulator which does not require a physical GPU environment or any GPU related tool-chain.

We evaluate *SArcs* for the timing correctness against a real GPU device (Tesla C2050) based on the NVIDIA Fermi generation. We evaluate our simulator by us-

[1] Chapter 6 is from the following publication:
BSArc: Blacksmith Streaming Architecture for HPC Accelerators ; Muhammad Shafiq, Miquel Pericas, Nacho Navarro, Eduard Ayguade; Accepted in ACM International Conference on Computing Frontiers, Cagliary Italy; May 15th, 2012.

ing multiple micro-kernels and application kernels. The results show that the simulated performance for the evaluated kernels closely follow the trend of real executions. The averaged error as compared to real GPU executions remains around 20%. This error mainly comes from compiling codes for different Instruction Set Architectures (ISA). However, the usage of CPU ISA projections over GPU ISA offer a platform-independent simulator to research GPU-like architectures.

6.1 Simulators and Computer Architecture Research

Advancements in computer architecture research have drastically changed the world by offering a range of devices, from abundant types of hand-held computing gadgets to multi-Petaflop supercomputers. Many high performance computing centers are now moving to heterogeneous solutions consisting of general purpose CPUs along with streaming accelerators like GPUs and reconfigurable devices. This is evident from the fact that in late 2011, at least, three out of the top five [91] supercomputers in the world belong to this class of heterogeneous systems. It shows the enormous performance potential of such systems, which have GPUs working as accelerators for the CPUs. GPUs and CPUs in a machine can run in parallel but execute different types of codes. In general, the CPUs run the main program, sending compute intensive tasks to the GPU in the form of kernel functions. Multiple kernel functions may be declared in the program but in general only one kernel is executed on one GPU device at a time.

GPU computing has become an effective choice for the fine-grained data-parallel programs with limited communications. However, these are not so good for programs with irregular data accesses and a lot of communication [92]. This is because the original architecture of GPU was designed for graphics processing. In general, these graphical applications perform computations that are embarrassingly parallel. Later, the GPU architecture was improved [93] to be able to perform general purpose computing like a general purpose processor under CUDA [13] and OpenCL [94] like programming models. However, the performance from these devices is still largely dependent on the arrangement of data, whether coalescing is possible and if data sets are independent [95]. Therefore, on the one hand, all this imposes a need for writing new algorithms focusing on exposing parallelism in the data to get performance from these devices and, on the other hand, further improvements are necessary in the GPU architecture to make it less sensitive to the nature of applications. These requirements can be met rapidly with the availability of an easily usable simulation infrastructure for GPU-like streaming devices.

If we look at the history of simulation environments like SimpleScalar [96], Simics [97], PTLsim [98], M5 [99], TaskSim & Cyclesim [100] etc, available for research on general purpose processor architectures, it becomes evident that streaming architectures like GPUs are lacking of similar level of support from the simulation infrastructures. No-doubt, there exist some good efforts in the development of GPU simulation environments. These efforts mostly adopt the analytical methods but efforts also have been made to develop GPU simulation tools. In analytical methods, two interesting contributions are from Hong et al. Initially they proposed a GPU performance model [101] and later extended it as integrated performance and power model for GPUs [102]. *CuMAPz* is a CUDA program analysis tool proposed by Y. Kim and A. Shrivastava [103]. The *CuMAPz* approach is compile-time analysis . Therefore, It can not Handle any information that can only be determined during run-time, such as dynamically allocated shared memory, indirect array accesses, etc. In 2009, A. Bakhoda et al. proposed a detailed GPU simulator [104] for analyzing the CUDA Workloads. A GPU adaptive performance modeling tool [105] presented by Baghsorkhi et al. *GROPHECY* [106] takes as input a modified CPU code called *Code Skeleton* from the user to tune it for a GPU based implementation. *GpuOcelot* [107] is another interesting compilation framework for heterogeneous systems. *GpuOcelot* provides various back-end targets for CUDA programs and analysis modules for the PTX instruction set. In addition to the current standalone framework of *SArcs*, we are planning to use *GpuOcelot* as a front-end of *SArcs* to enable a provision to also generate traces directly from the PTX code. MacSim [108] is a *GpuOcelot*-based trace driven simulation tool chain for heterogeneous architectures. The idea behind MacSim is to convert the program trace to RISC style *uops* and then simulate these *uops*. *SArcs*, on the other hand, controls the trace generation process. The generated trace is either from a CPU code or a PTX based GPU code, thus *SArcs* can directly map and simulate the real trace for a GPU generation. It gives an opportunity to researchers in computer architecture to be able to explore various possibilities to improve current GPU designs. The SArcs framework is equally beneficial for application programmers. They can use it for performance estimations of their applications by configuring the simulator to model a specific GPU device.

Our proposal on *SArcs* contributes in computer architecture research by providing an automated framework for simulations of streaming architectures like GPUs. *SArcs* can be used either as a standalone system – completely independent of a streaming environment – or it can be connected to other existing simulation related tools. However, this work only intends to present *SArcs* as an independent simulation infrastructure for GPUs which does not require to have a physical GPU or any GPU related software tool-chain. *SArcs* is a trace driven simulation framework and exploits the fact that an application compiled for any architecture would require to transact the same amount of data with the main memory in the absence of registers or cache hierarchy. Moreover, the computations inside an application can be simulated by the target device latencies. The instruction level dependencies in GPU like architectural philosophy pose least impact on the performance because of zero-overhead switching between the stalled and large number of available threads. *SArcs* creates an architectural correlation with the target device by passing the source code through a source to source translator followed by a thread aware trace generation. This trace is used by a device mapping process which transforms the trace into a SIMT trace specific for a GPU architecture. The SIMT trace is passed through a cycle accurate simulator to get the performance and related statistics.

The modules of *SArcs* are written in C or C++. These are enveloped inside a python script to run in an automated way which starts by grabbing the application source file and finalizes showing performance results. To the best of our knowledge *SArcs* is the first trace based GPU architectural simulator which can also be used independent of the requirements of having any kind of GPU environment. We compare our simulator for the performance characterizations against a real GPU device (Tesla C2050). In our evaluations we use a set of five micro-kernels, to minutely explore different aspects of the simulator in comparison to the real performance of the GPU device. Further, we evaluate three different application kernels from Matrix-Matrix Multiplication, Vector Reduction and 2D-Convolution. The results show that the averaged error for *SArcs* simulated performance is around 20% of the real executions on GPU. It shows the potential of the *SArcs* framework, which offers a platform-independent simulator to research GPU-like architectures.

6.2 The Simulator Framework

The basic goal of *SArcs* is to provide a simulation platform for streaming architectures that could be used for applications performance analysis or to experiment around the architectural innovations. These objectives are achieved by working through different stages of the *SArcs* framework. These stages – as shown in Figure 6.1 – consist of the *Trace Generation*, the *Device Mapping*, the *Device Simulation* and the *Results Analysis*. The Figure 6.1 also shows that these stages are executed in different steps. All steps – from reading a CUDA source file (step-1) to the analysis of simulation results (step-6) – are automated under python and its extensions like SciPy, NumPy etc. The steps 3 to 5 can be repeated for the number of device kernels in an application and/or as many times a device kernel requires to run with different inputs. The details on different stages of the *SArcs* framework are given in the next sections.

6.3 Trace Generation

SArcs supports CUDA programming model. The users of *SArcs* are only required to write a plain CUDA program (The *main* and the device *kernel(s)*) for an application. The users can use CUDA specific API's inside the device kernel. However, it is not allowed to call any application specific API's for the standalone version of *SArcs*. The CUDA source files for an application are processed by a source to source translator (*S-S Translator*) before compilation with the *g++* compiler in *step-2* as shown in the Figure 6.1. After compilation, the generated binary of the application is forwarded to a thread aware tracing tool (*TTrace tool*) to generate the traces. The details on *S-S Translator* and *TTrace tool* are given below:

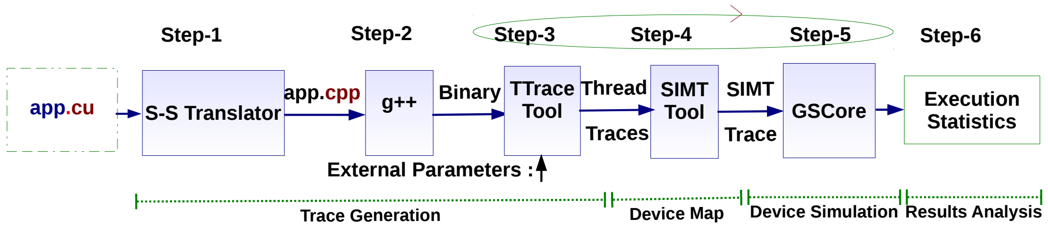


Figure 6.1: The Framework of *SArcs*

```
#define __global__   extern "C" __attribute__((noinline))
#define __device__   inline
#define cudaMemcpyHostToDevice 1
#define cudaMemcpyDeviceToHost 2

typedef struct {
        unsigned long x;
        unsigned long y;
}struct_blockIdx;

typedef struct {
        unsigned long x;
        unsigned long y;
}struct_blockDim;

struct_blockIdx        blockIdx;
struct_blockDim        blockDim;

class dim3 {
public:

unsigned long x;
unsigned long y;

dim3(long dimx): x(dimx), y(1)
        {}
dim3(long dimx, long dimy): x(dimx), y(dimy)
        {}
};

void cudaMalloc (void ** memptr, unsigned long memsz)
{
   *memptr = (char *)malloc(memsz);
}
 - -  - - - - - -  - - - - - -
```

Figure 6.2: Some example declarations & definitions in modified cuda header file ("mcuda.h")

```
/* kernel_name dimGrid,dimBlock >>> (a_d,b_d,c_d,iter); */
      blockDim.x = dimBlock.x;
      blockDim.y = dimBlock.y;
printf("GDim.y , GDim.x , BDim.x , BDim.y , BId.y , BId.x , TId.y , TId.x\n");
printf(":>REF:>%p %p %p %p %p %p %p %p<:REF<:\n", &dimGrid.x,&dimGrid.y,&blockDim.x,&blockDim.y,
        &blockIdx.y,&blockIdx.x,&threadIdx.y, &threadIdx.x);

printf("BId.y , BId.x , TId.y , TId.x , GDim.y , GDim.x , BDim.x , BDim.y \n");
printf(":>PAR:>%ld %ld %ld %ld %ld %ld %ld %ld<:PAR<:\n",dimGrid.x,  dimGrid.y, blockDim.x, blockDim.y,
        blockIdx.y, blockIdx.x, threadIdx.y, threadIdx.x );

   for (blockIdx.y=0; blockIdx.y< dimGrid.y; blockIdx.y++)
                for(blockIdx.x=0; blockIdx.x< dimGrid.x; blockIdx.x++)
                        for(threadIdx.y=0; threadIdx.y< dimBlock.y; threadIdx.y++)
                                for(threadIdx.x=0; threadIdx.x< dimBlock.x; threadIdx.x++) {

                                  kernel_name (a_d,b_d,c_d,iter);

                                }
```

Figure 6.3: An example code insertion for the replacement of the target gpu kernel call

6.3.1 S-S Translator

S-S Translator is a source to source translator. It takes in a CUDA program and applies appropriate modifications and additions for two main reasons: (i) Program should be compilable by a GNU *g++* compiler (ii) The added code inside the source forces to output necessary runtime information to support the next stages of the simulator. At first, to make the CUDA code compilable with the GNU compiler, we provide the simulator framework with a modified cuda header file (*mcuda.h*). Inside the CUDA source code, the *S-S Translator* replaces normal *cuda.h* with *mcuda.h.* Some example declarations & definitions in the modified cuda header file are shown in Figure 6.2.

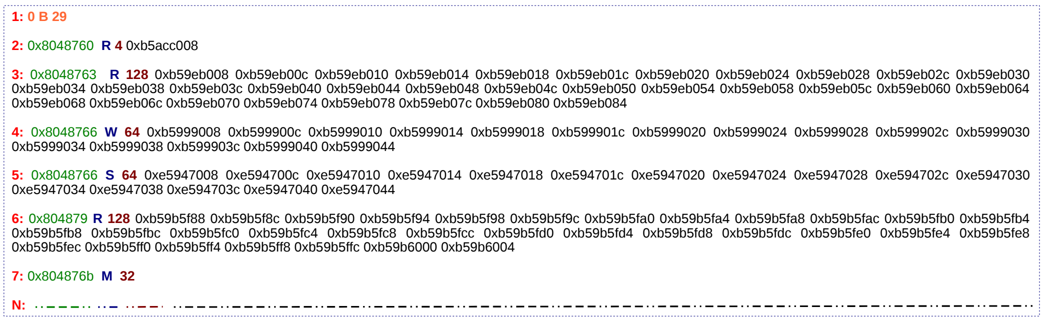
As it can be seen in line 1 of Figure 6.2, *SArcs* uses CUDA identifier *__global__* to avoid name mangling of the corresponding function names by the *g++* compiler. This helps the trace tool to recognize the calls to these functions by their names during the execution of the program. However, *SArcs* forces CUDA identifier *__device__* to make its related functions as *inline*. It is important to remember that global and device – both types – of functions specifically execute on a GPU device and not on the host. Further, the device functions can only be called from inside the global function. The lines 6 & 11 in the Figure-6.2 shows declarations of the structures representing the CUDA internal variables *blockIdx* and *blockDim* which are later instantiated as global variables in lines 16 & 17. The *dim3* structure of CUDA is replaced by the declaration of a *dim3* class between the lines 19 & 29 in the modified header file. Further, CUDA APIs like the *cudaMalloc(..)* is replaced with our own cudaMalloc which uses normal *malloc(..)* as shown in lines 31 to 34 of Figure 6.2. The same way SArcs framework redefines *_synchronise()* CUDA API and the CUDA internal variable *threadIdx* structure. The calls to *_synchronise()* function are marked inside the trace at the time of trace generation. However, all kinds of synchronizations between the threads are taken care by the GPU Simulation Core (*GSCore*) at the *Device Simulation* stage (Section 6.5). The shared memory identifier (*__shared__*) is also redefined so that shared memory should be treated as accesses from the stack. Currently, *SArcs* redefines all important structures of CUDA and as well most common CUDA APIs inside the modified header file.

The *S-S Translator* also inserts additional code at predefined places in the CUDA source file(s) as shown in Figure 6.3. This code insertion helps the simulator in two ways: (i) To get a detailed trace of target application kernel that needs to be run on the GPU device. (ii) To extract certain information from the code at run-time. The code between lines 2 and 19 – as shown in the Figure 6.3 – is an example replacement done by the *S-S Translator* for the code in line 1. Line 1 shows a commented CUDA call to a global function (*kernel_name*) that originally has to run on the GPU device. However, the *S-S Translator* commented this call and inserts a code with some assignment statements, *printf* instructions and nested loops. In this example piece of code (Figure 6.3), the lines 2 and 3 copies values of Block Dimensions to the global variables. Next, the lines 4 to 10 show code inserted to extract some runtime information specific to a code and also specific to a run. The examples of this runtime information are the pointer addresses assigned to the global variables *dimGrid*, *blockDim*, *blockIdx* and *threadIdx*. This information is used during the later steps of the simulation process. The nested loops in the inserted code from lines 12 to 19 calls the target function (*kernel_name*) at the thread granularity (the most inner loop). These nested loops make it possible to generate a complete trace for all the threads (originally CUDA Threads) in a Block (originally CUDA Block) and for all the Blocks in a Grid. It is important to remember that these nested loops work according to the dimensions of a block and the grid dimensions. These dimensions are defined by the user before calling a gpu target function in a CUDA program.

6.3.2 TTrace Tool

The modified source code from the *S-S Translator* is compiled with the *g++* compiler at the step-2 (Figure 6.1) of SArcs framework. The binary of the program is executed with the thread aware trace (*TTrace*) tool. *TTrace* tool uses dynamic instrumentation of the programs in the PIN [109] environment. The target kernel function name (originally the GPU device kernel) can either be given as an external argument or – by default – it is identified by the *S-S Translator* and forwarded to *TTrace tool*. The name of the kernel function allows the tool to only instrument this function.

The main parameters traced by this binary instrumentation tool include the *Instruction Pointers*, *Instruction Ops*, *Memory Addresses*, *Memory Access Sizes* and any calls to the sub-functions from the kernel function e.g the calls to the thread synchronization APIs. In a CPU ISA, the instruction set can be very large. Therefore *TTrace Tool* only



```
1: 0 B 29

2: 0x8048760  R 4 0xb5acc008

3: 0x8048763   R 128 0xb59eb008 0xb59eb00c 0xb59eb010 0xb59eb014 0xb59eb018 0xb59eb01c 0xb59eb020 0xb59eb024 0xb59eb028 0xb59eb02c 0xb59eb030
0xb59eb034 0xb59eb038 0xb59eb03c 0xb59eb040 0xb59eb044 0xb59eb048 0xb59eb04c 0xb59eb050 0xb59eb054 0xb59eb058 0xb59eb05c 0xb59eb060 0xb59eb064
0xb59eb068 0xb59eb06c 0xb59eb070 0xb59eb074 0xb59eb078 0xb59eb07c 0xb59eb080 0xb59eb084

4: 0x8048766 W 64 0xb5999008 0xb599900c 0xb5999010 0xb5999014 0xb5999018 0xb599901c 0xb5999020 0xb5999024 0xb5999028 0xb599902c 0xb5999030
0xb5999034 0xb5999038 0xb599903c 0xb5999040 0xb5999044

5: 0x8048766 S 64 0xe5947008 0xe594700c 0xe5947010 0xe5947014 0xe5947018 0xe594701c 0xe5947020 0xe5947024 0xe5947028 0xe594702c 0xe5947030
0xe5947034 0xe5947038 0xe594703c 0xe5947040 0xe5947044

6: 0x804879 R 128 0xb59b5f88 0xb59b5f8c 0xb59b5f90 0xb59b5f94 0xb59b5f98 0xb59b5f9c 0xb59b5fa0 0xb59b5fa4 0xb59b5fa8 0xb59b5fac 0xb59b5fb0 0xb59b5fb4
0xb59b5fb8 0xb59b5fbc 0xb59b5fc0 0xb59b5fc4 0xb59b5fc8 0xb59b5fcc 0xb59b5fd0 0xb59b5fd4 0xb59b5fd8 0xb59b5fdc 0xb59b5fe0 0xb59b5fe4 0xb59b5fe8
0xb59b5fec 0xb59b5ff0 0xb59b5ff4 0xb59b5ff8 0xb59b5ffc 0xb59b6000 0xb59b6004

7: 0x804876b  M  32

N:
```

Figure 6.4: An Example SIMT Trace (The left side numbering (1,2,3...) is added just to describe the trace inside the text)

identifies common operations and rest of the operations are accommodated under the same identification. The operations for addition, multiplication, devision and memory accesses are identified separately. Further, separate identifications are also given to the heap based memory accesses and the stack based memory accesses.

The *TTrace tool* arranges the instruction level trace information in separate thread groups. *SArcs* framework helps in this thread level grouping of instruction level trace by the insertion of nested loops with the *S-S Translator* as described in the previous section. The execution of the modified program's binary also spit out different types of information related to the program execution e.g. Address pointers for various variables, the size and the base address of the shared memory array, Block and Grid Dimensions etc. This information is saved into a temporary file to be used in the processing of the next stages.

6.4 Device Map

The *Device Mapping* stage provides an isolation between the user control over the program and the micro-architectural level handling of the program execution by a GPU generation. For example, In the trace generation stage, the user has a control over the CUDA program to adjust the Block and Grid dimensions while the number of threads in a WARP is a micro-architectural feature of a GPU device handled at the *Device Mapping* stage. This stage of *SArcs* framework uses a *SIMT tool* to map a user program trace (the output of *TTrace tool*) for a specific GPU device. The output of the the *SIMT* *tool* is a SIMT trace which is fed to a *GPU Core Simulator* in the next stage. The *SIMT tool* passes the user program trace through multiple processing phases. Some important phases are described below:

6.4.1 Garbage (Built-in) Removal

A real GPU uses some built-in variables represented in CUDA as *dimGrid*, *blockDim*, *blockIdx* and *threadIdx* etc. These variables act as parts of the GPU micro-architecture. However, in our trace generation methodology, these variables acts as global variables with their accesses from the main memory. *SArcs* removes all accesses to these variables from the trace by identifying their address pointers obtained at the execution of program with *TTrace tool*.

6.4.2 WARP Instructions Formation

The user program trace (the output of *TTrace tool*) only groups the instructions traces at thread level granularity. The *SIMT tool* arranges these trace instructions as WARP Instructions and group these WARP Instructions at the Block granularity. In a real GPU a WARP consists of *N* number of consecutive threads. As we mentioned earlier the user program trace consists of a set of trace instructions for each thread. A WARP Instruction is formed by taking one trace instruction from each of the consecutive N threads. The next WARP Instruction is formed by taking the next one trace instruction from the instruction trace set for each of the consecutive N-Threads and by combining them all. The WARP Instruction formation process makes it sure that each trace instruction added in a WARP Instruction should correspond to the same trace instruction pointer. If the instruction pointer changes for an expected trace instruction of a thread for a WARP, it is taken as *control divergence* inside the WARP. The formation of the current WARP Instruction completes at this point and a new WARP Instruction formation is started until the end of the N^{th} thread or it encounters another control divergence. This Instruction formation process also allows the convergence of diverged thread WARPs. The WARP Instruction formation process completes with the creation of sets of WARP Instructions for all the WARPs inside a Block and for all the Blocks inside a Grid.

6.4.3 Coalescing Effects

The sets of WARP Instructions created in the previous step are further processed by the *SIMT tool* to add the coalescing or un-coalesced effects for the memory access instructions. The *SIMT tool* runs an analysis on the data access pointers for the WARP instructions. A WARP Instruction is split into multiple WARP Instructions if the memory accesses are not coalesced inside the original WARP Instruction. The new WARP Instructions contains accesses which are coalesced. If the selected target is a GPU Fermi device then the *SIMT tool* allows memory WARP Instructions to be formed for fetching up-to a maximum 128 Bytes in one transaction otherwise (target GPU is not a Fermi device), the coalesced memory access instructions are further split the WARP Instruction to new WARP Instructions such that the maximum allowable coalesced access from a WARP Instruction should not exceed 64 Bytes.

6.4.4 Registers and Shared Memory Handling

In a GPU kernel, the local variables are mapped to the SM (Streaming Multiprocessor) registers. Therefore, the scope of accesses to these local variables inside a GPU remains inside a block allocated to a SM. *SArcs* categorize all stack based accesses inside a kernel either as registered accesses or the shared memory accesses. The shared memory accesses are isolated from the registered accesses based on the base pointer of the shared array and its allocation size. Currently *SArcs* does not handle corner cases like dynamic allocation of shared memory. The shared memory is also organized as WARP Instructions with identification 's' as shown in line 5 of Figure 6.4. The device map tool runs an analysis on the shared memory accesses and arranges them as one or more than one WARP Instructions based on the access pattern and GPU specifications.

6.4.5 Grouping Blocks

We call the new formatted trace generated by the *SIMT tool* as *SIMT Trace*. The *SIMT Trace* is arranged in Blocks. In order to help the *GPU Simulation Core* (Section 6.5) to efficiently access the SIMT Trace, *SIMT tool* arranges these blocks in multiple files (called SIMT trace files) which are kept equal to the number of SMs in the target GPU device. This means that if there are *M* number of SMs then the first SIMT file will contain 1^{st}, $M + 1^{th}$, $2 * M + 1^{th}$ and so on SIMT trace Blocks. However, as we will see in the explanations of *GPU Simulation Core* that this arrangement does not create any binding on the choice of SIMT trace Blocks for any SM during the simulation process.

The Figure 6.4 shows some example entries of a SIMT Trace. The left most mark-1 identifies the grid number and block number for a SIMT Trace. The mark-2 to 6 show memory access WARP instructions each of which includes – from left – the instruction pointer, Operation i.e memory read (R), memory write (W) or shared memory (S), size of data (in Bytes) to be transact and the memory addresses. The last mark-7 in the example SIMT trace represent Multiply(M) operation scheduled for all streaming processors. In-case, two consecutive WARP Instructions are Multiply and ADD, the *SIMT Tool* fuses them as one Multiply-ADD WARP Instruction.

6.5 Device Simulation

The *Device Simulation* stage models the dynamic effects for various micro-architectural components of a target GPU device. This stage uses *GSCore* (GPU Simulation Core), a cycle accurate simulator specifically developed in-house for simulating the GPU like streaming devices. The functional layout of *GSCore* is shown in the Figure 6.5. This simulator accepts SIMT Trace files generated by the *SIMT tool*. These SIMT trace files contains Blocks of WARP Instructions as shown at the top of the Figure 6.5. These Blocks corresponds to the Blocks defined in a Grid for the target application kernel. However, now these Blocks do not contain threads but traces arranged in the form of WARP Instructions. The *GSCore* implements a *Block Scheduler* which is responsible for delivering these Blocks to the SMs – initially – in a round-robin fashion and later based on requests from a SM. SMs are represented as *WIL Schedulers* next to the *GSCore's Block Scheduler* in the Figure 6.5. The *WIL Scheduler* is named upon its real function which is to schedule the WARPs Instructions & Latency (WIL).

The *WIL Scheduler*, schedules WARPs Instructions from one or more Blocks based on the latencies corresponding to the operations these WARPs have to do. The latency values for different operations are loaded by the *GSCore* corresponding to a target device from a *GPU Constants File*. This constant parameters file is provided with the *SArcs* frame work. The GPU Constants file keep architectural and micro-architectural

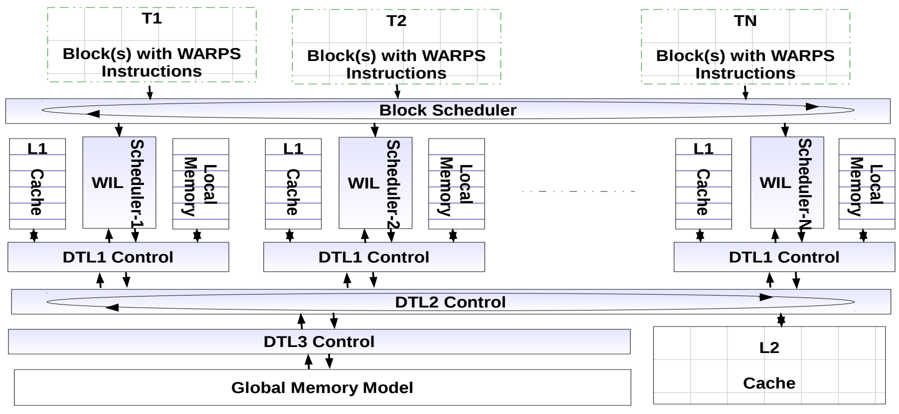


Figure 6.5: GPU Simulation Core (GSCore)

parameters for various GPU devices. The latencies due to the instruction level dependencies are normally hidden or unknown in trace driven simulators. However, In case of *GSCore*, the final performance as compared to a real GPU shows almost no effect for these dependencies. This is because of the inherent nature of the real GPU architecture which switches with almost zero-overhead between the WARPs to avoid performance loss due to these dependencies.

The WARP Instructions corresponding to memory transactions are forwarded to the Data transaction Level-1 (DTL-1) control. The memory WARP Instructions are scheduled as first-come first-serve basis or in a round-robin way if multiple requests are available in the same cycle from different WILs (SMs). These memory WARP Instructions goes through the *GScor's* modeled memory hierarchy corresponding to a real GPU. This includes implementation of configurable L1 Cache and Local Scratch Pad memory for each of the WIL Scheduler (i.e for each SM in a real GPU), L-2 Cache and the Global Memory. All levels of *GScor* works in a synchronous way and simulate latencies from going one level to another one. In-case, a memory WARP Instruction is not fulfilled at (DTL-1), it is passed to the DTL-2 – for L2 cache test —. and if required it is forwarded to the DTL-3 level which models a Global memory access. All WARP Instructions which are memory writes are forwarded to the Global memory.

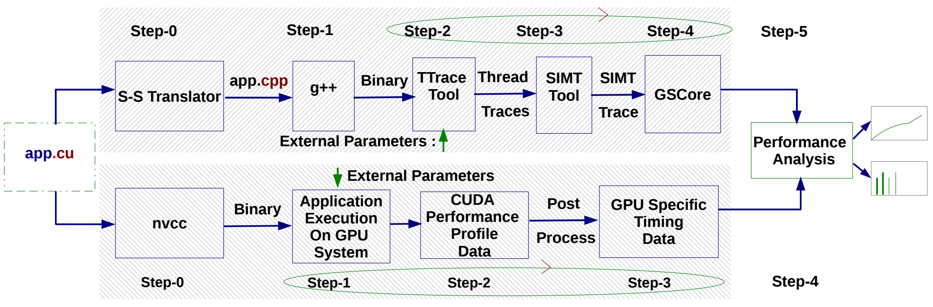


Figure 6.6: Evaluations Methodology (SArcs Program Executions vs Real GPU Program Executions)

6.6 Evaluations

The Figure 6.6 shows the methodology used in *SArcs* evaluations. In our evaluations of *SArcs*, we target NVIDIA's GPU Tesla C2050 from the Fermi generation. This device has 14 Streaming Multiprocessors (SMs) each contains 32 scalar processors. The device is capable of performing 32 fp32 or int32 operations per clock cycle. Moreover, it has 4 Special Function Units (SFUs) to execute transcendental instructions such as sin, cosine, reciprocal, and square root. On the memory hierarchy side the device supports 48 KB / 16 KB Shared memory, 16KB / 48 KB L1 data cache and 768Kbytes of L2 memory.

The *SArcs* can be compiled for any host machine. The only constraint is that the PIN environment used in *TTrace tool* should have support for that CPU. In our evaluations, we use IBM "x3850 M2" machine. It has 48GBytes of main memory and 4 chips of Intel Xeon E7450, each one with 6 Cores running at 2.40GHz. This machine only helps us to run multiple instances of the simulation in parallel, otherwise a single core machine can be used for running single instance of the simulator. Further, in our case, the host machine uses x86_64-suse-linux and gcc compiler version 4.3.4. The target application kernels are compiled for optimization level 3 (switch -O3). On the GPU side, we use *nvcc* compiler with cuda compilation tool release 4.0, V0.2.1221. We compiled the the CUDA codes using optimization level 3. Further, we use compilation switch *-Xptxas* along with *-dlcm=ca or -dlcm=cg* to enable and disable L1 cache accesses where ever needed.

In our evaluations, we study two different cases. The first case is used for fine detailed analysis of the simulator targeting memory while the second case shows performance for three application kernels. Some details for the case studies are as follows:

6.6.1 Case 1: Memory Micro-Kernels

The memory micro-kernels are based on five different types of memory accesses during single execution of a thread. We categorize these single thread accesses in the ratio between consecutive reads (R) and writes (W). These ratios are R:W = 0:4 , 1:3, 2:2, 3:1 and 4:0. In order to avoid *nvcc* compiler from optimizing out the R:W=4:0 case, we use an external flag passed from command prompt to implement the kernel for only a conditional write. This flag always remain *false*.

The memory micro-kernels are used for two types of evaluations, the SM level evaluations and multiple block evaluations as described in the next sections.

6.6.1.1 (a) SM Level Evaluations

The purpose of SM level evaluations is to test the simulator behavior at the individual SM level. In this case we always keep thread blocks less than or equal to the maximum number of SMs in the GPU device. In these evaluations *SArcs* assumes that the real GPU scheduler will schedule each thread block to a different SM to maximize the parallelism inside the device.

6.6.1.2 (b) Multiple Block Evaluations

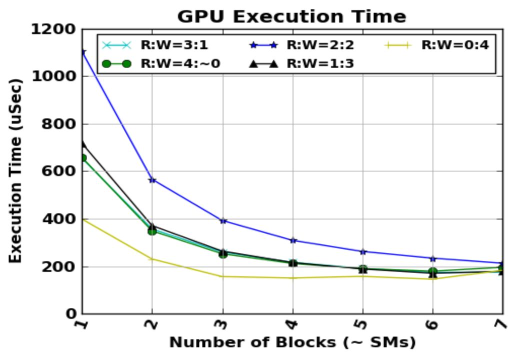
The multiple block evaluations for memory micro-kernels always configure the CUDA code to run number of thread blocks larger than the total number of SMs.

6.6.2 Case 2. Application Kernels

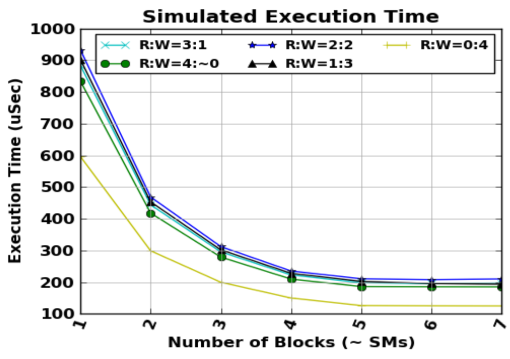
In the application kernels we use Matrix Matrix multiplication (MM), 2D-Convolution (CV) and the Vector Reduction (RD). The MM kernel is evaluated for both compiled with L1 and with-out L1. The other two kernels (CV and RD) uses configurations for the shared memory usage in their implementations. For all of the three kernels, L2 cache is always kept enabled in *SArcs* and as well for the GPU. This is mainly because we did not find a proper way to shut-off L2 in the GPU device. The vector reduction kernel uses multiple invocations of the the GPU device in reducing the whole vector to a single value. This also validates the *SArcs* capability of handling multiple kernel invocations by an application. However, only one *__global__ kernel* can be invoked by an application at one time for the GPU simulated execution.

6.7 Results and Discussion

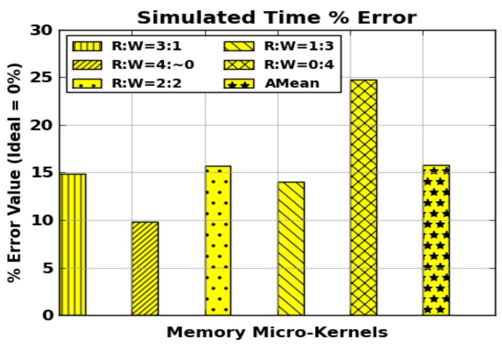
The results for the *SArcs* evaluations are shown in Figures 6.7 to 6.9. It can be observed that in all cases the *SArcs* simulated results closely follow the trends for the real GPU



(a)

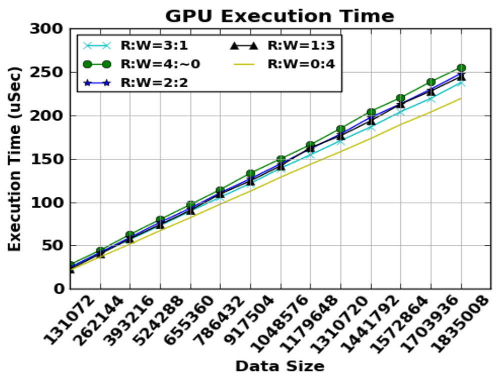


(b)

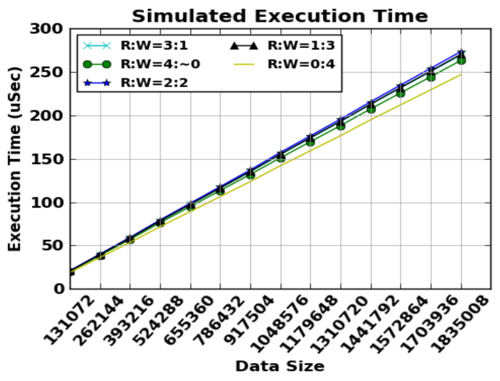


(c)

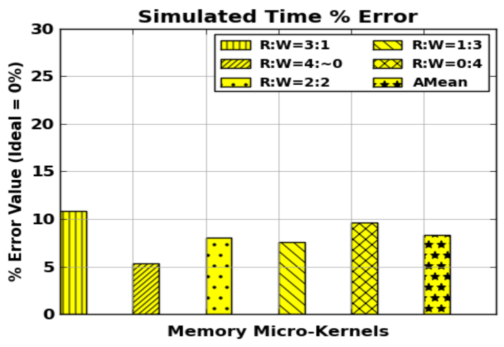
Figure 6.7: Case 1 (a): Memory micro-kernels for SM Level evaluations (a) GPU - Tesla C2050 execution time (b) SArcs simulated execution time (c) Averaged Percentage performance error for each micro-kernel



(a)



(b)



(c)

Figure 6.8: Case 1 (b): Memory micro-kernels with multiple blocks evaluations (a) GPU - Tesla C2050 execution time (b) SArcs simulated execution time (c) Averaged Percentage performance error for each micro-kernel

based executions. In our evaluations – to remain fair – we are not taking any specific part of execution for error computation between the simulated results and the real execution times. In all cases we computed point to point error for all evaluated points in a case and then averaged over the total number of points in that case. The results show that the averaged error for *SArcs* simulated performance remains 20% of the real executions on GPU. It is important to remember that the current version of *SArcs* is using CPU code projections for the GPU and one source of this error comes from compiling codes for different Instruction Set Architectures (ISA). The difference in compilation platforms appears in the form of different size of the compiled code which ultimately appears as a difference in the execution times. Other source of error include the choice of selection from the huge set of CPU instructions at the simulation phase of trace gen-

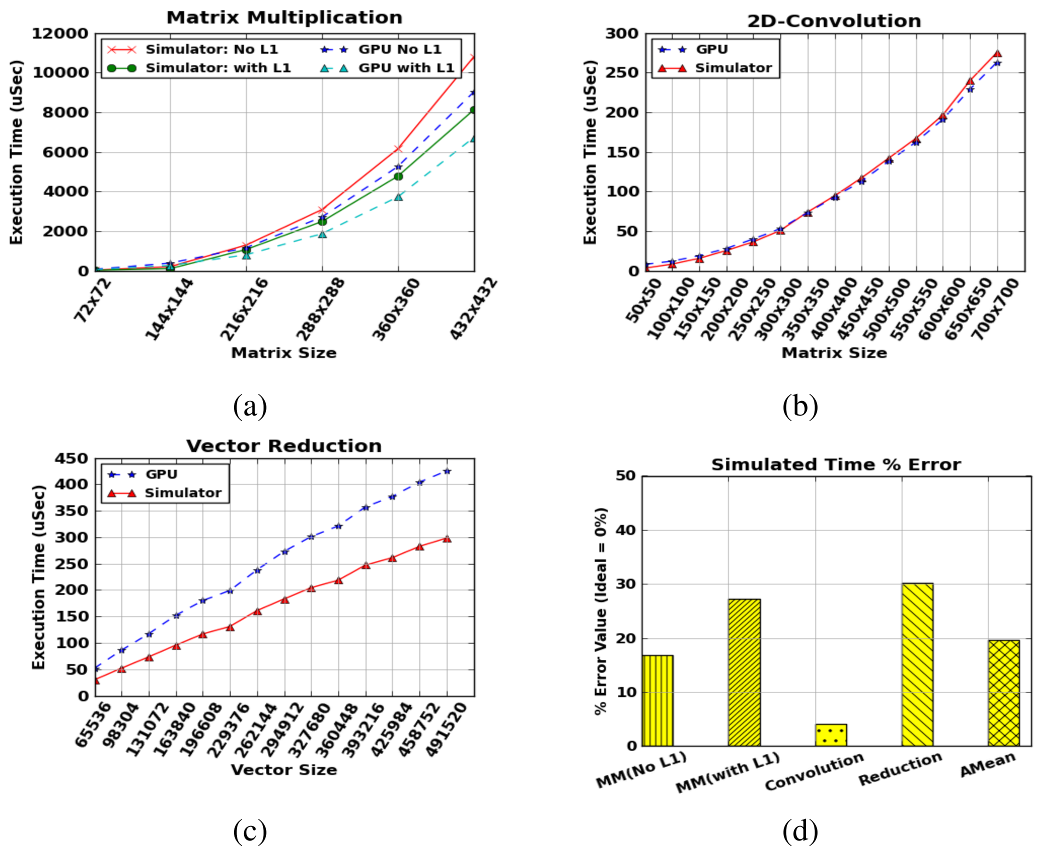


Figure 6.9: Case 2: (a) Matrix Matrix multiplication : Simulated and real execution time using L1 and with-out L1 (b) 2D-Convolution : Simulated and Real execution time using Tiling in the Shared Memory (c) Vector Reduction : Simulated and Real execution time using shared memory and multiple invocations of the GPU-kernel from the CPU during the reduction process (d) Percentage of the error of the simulated performance to that of the real one on GPU for the kernels

eration. Moreover, a lack of the precise information regarding the micro-architectural details of the target GPU device also contributes in the error between the simulated and real performance. However, the overall behavior of simulator appeared in our results shows that the usage of CPU ISA projections over GPU ISA has a potential to provide researchers a platform-independent simulator to research GPU-like architectures.

6.7.1 SArcs Limitations and the Future Work

- The trace generation process takes very long time as compared to that for the GPU Simulation Core. Moreover, the traces can reach to sizes in 20's of GBytes. Generally, these issues are common for all trace based system. A lot of work has already been done on the sampling techniques to reduce the trace sizes and tracing time. In *SArcs* framework the traces of a program are required to be generated only once in the step-3 (Figure 6.1). The later simulation steps are pretty fast and these can be decoupled from the trace generation process for rapid architecture level investigations.

- The CUDA application programmer needs to take care for the usage of target device resources. *SArcs* team is enhancing the *S-S Translator* to provide the user with a prediction of the expected resource usage by the target CUDA program.

- The users can use CUDA specific API's inside the device kernel. However, it is not allowed to call any application specific API's for the current standalone version of *SArcs*. We have a future plan to also use a modified GpuOcelot [107] environment at the *SArcs* front-end to increase the coverage to library based CUDA codes. This will enable SArcs to be used either as a standalone framework or by involving some components from the GPU environment.

- The error in the *SArcs* simulated results can vary depending upon the compiler and the instruction set of the host for which the target kernel is compiled. However, generally the simulated performance should follow the real execution trends.

6.8 Summary

GPUs introduced just a decade back are now an effective part of many HPC platforms. However, GPUs still lacks for the availability of simulation infrastructures as compared to the simulation environments available for the general purpose processors. In this work, we show that the architectural model of GPU devices can be effectively transformed to a simulator infrastructure under our proposed *SArcs* framework. *SArcs* framework provides an automated interface to simulate application performance on a target GPU. Moreover, the design of *SArcs* framework uses software components which are well known to a vast majority of researchers in computer architecture. Therefore, we consider *SArcs* as a potential step towards extending the research for GPU like streaming architectures. We show detailed methodology of using CPU code projections to simulate for a target GPU device. The overall behavior of the simulator appeared in the results shows that the usage of CPU ISA projections over GPU ISA has a potential to provide researchers a platform-independent simulator to research GPU-like architectures. The results of the *SArcs* framework motivates for further research and explorations in this direction. The next chapter of this document uses the *SArcs* tool chain for the design space explorations of throughput orientd streaming accelerators.

7

Design Space Explorations for Streaming Architectures using SArcs

In the recent years streaming accelerators like GPUs have been pop-up as an effective step towards parallel computing. The wish-list for these devices span from having a support for thousands of small cores to a nature very close to the general purpose computing. This makes the design space very vast for the future accelerators containing thousands of parallel streaming cores. This complicates to exercise a right choice of the architectural configuration for the next generation devices. However, design space exploration tools specifically developed for the massively parallel architectures can ease this task. This chapter presents two studies related to the design space explorations for the streaming architectures i) Design space explorations for the GPU like Streaming Architectures ii) Design space explrations for Blacksmith Streaming Architecture

The main objective of the first study is the design space explorations of a GPU like streaming architecture using the trace driven simulator *SArcs* (**S**treaming **Arc**hitectural **S**imulator). Our design space explorations for different architectural aspects of a GPU like device are with reference to a base line established for NVIDIA's Fermi architecture (GPU Tesla C2050). The explored aspects include the performance effects by the variations in the configurations of *Streaming Multiprocessors*, *Global Memory*

[1] Chapter 7 is based on the followings:
(1) Design Space Explorations for Streaming Accelerators using Streaming Architectural Simulator; Muhammad Shafiq, Miquel Pericas, Nacho Navarro and Eduard Ayguade; UPC research report: UPC-DAC-RR-2012-6.
(2) BSArc: Blacksmith Streaming Architecture for HPC Accelerators ; Muhammad Shafiq, Miquel Pericas, Nacho Navarro, Eduard Ayguade; Accepted in ACM International Conference on Computing Frontiers, Cagliary Italy; May 15th, 2012.

Bandwidth, *Channels between SMs down to Memory Hierarchy* and *Cache Hierarchy*. The explorations are performed using application kernels from Vector Reduction, 2D-Convolution, Matrix-Matrix Multiplication and 3D-Stencil. Results show that the configurations of the computational resources for the current Fermi GPU device can deliver higher performance with further improvement in the global memory bandwidth for the same device.

The second study presents a conceptual computing architecture named *BSArc* (Blacksmith Streaming Architecture). *BSArc* introduces a forging front-end to efficiently distribute data to a large set of simple streaming processors in the back-end. We apply this concept to a SIMT execution model and present design space explorations in the context of a GPU-like streaming architecture with a reconfigurable application specific front-end. These design space explorations are carried out on the streaming architectural simulator that models BSArc. We evaluate the performance advantages for the BSArc design against a standard L2 cache within a GPU-like device. In our evaluations we use three application kernels: 2D-FFT, Matrix-Matrix Multiplication and 3D-Stencil. The results show that employing an application specific arrangement of data on these kernels achieves an average speedup of 2.3× compared to a standard cache in a GPU-like streaming device.

7.1 Design Space Explorations

In computer architecture research, design space explorations are a key step for proposing new architectures or modifications in an existing architectural configuration. During the last decade, computer architecture research has witnessed a shift from a single core to mulicore processors and expectedly the future of computer architecture research will be revolving around the parallel architectures [6]. This has made the design space explorations a great challenge for the computer architects. The designs of new high performance computing (HPC) systems which are sharply converging towards the idea of exploiting massively data-level parallelism on large number of compute cores – like in a GPU – has further complicated this challenge. The one way to overcome these challenges is the development of new architectural exploration tools by taking into account the new research trends in computer architecture. Our proposed simulation infrastructure *SArcs* (Chapter 6) for GPU like stream devices is a step toward meeting such challenges.

GPUs introduced just a decade back are now considered an effective part of many HPC platforms [91]. GPUs are throughput-oriented devices. A single GPU device can contain hundreds of small processing cores. These use multi-threading to keep a high throughput and hide memory latency by switching between thousands of threads. In general, the architecture of a GPU consists of dual level hierarchy. The first level is made of vector processors, termed as streaming multiprocessors (SMs) for NVIDIA GPUs and SIMD cores for AMD GPUs. Each of the vector processor contains an array of simple processing cores, called streaming processors (SPs). All processing cores inside one vector processor can communicate through an on-chip user-managed memory, termed local memory for AMD GPUs and shared memory for NVIDIA. On a single HPC platform, GPUs and CPUs can run in parallel but execute different types of codes. Generally, the CPUs run the main program, sending compute intensive tasks to the GPU in the form of kernel functions. Multiple kernel functions may be declared in the program but as a common practice only one kernel is executed on one GPU device at a time. Therefore, most of the HPC platforms uses configurations of single CPU with multiple GPUs to run kernels independently and in parallel. However, the performance driving factor remains the basic architecture of the device being used in all the GPUs of the platform.

GPUs are still considered at an early stage of an era of their architectural growth and innovations. As compared to the enormous amount of efforts devoted to application development for GPUs, only a little has been done on the GPU performance characterization and the architectural explorations. Only a few years back, GPUs were only an effective choice for the fine-grained data-parallel programs with limited communications. However, these were not so good for programs with irregular data accesses and a lot of communication [92; 95]. This is because the original architecture of GPU was designed for graphics processing. In general, these graphical applications perform computations that are embarrassingly parallel. Later, the GPU architecture was improved [93] to be able to run general purpose programs under CUDA [13] and OpenCL [94] like programming models. The general purpose programs with arbitrary data-sets may or may not perform well on the GPU like streaming devices. This motivates the newer generation of the GPUs like the NVIDIA's Fermi architecture to incorporate both the level-1 and the level-2 caches in their memory hierarchy. However, further architectural improvements in these devices can make them most interesting choice for the efficient parallel computing.

The design choices for GPU like streaming architectures are so large and diverse that these architectures are still finding, on one hand, a balance between the available bandwidth and the on-chip computational resources and on the other hand, a balance between generality and specialty of the underlying architecture. This imposes a need to rapidly explore design spaces for the new GPU like proposals. We – in this work – present: i) Design space explorations for the GPU like Streaming Architectures ii) Design space explrations for Blacksmith Streaming Architecture containing configurable front-end and GPU like back-end. These explorations are done using a locally developed environment of a trace driven simulator called *SArcs* (**S**treaming **Arc**hitectural **S**imulator). A brief introduction to the simulator is given next. However, the details on the simulator design could be found in the chapter 6.

SArcs simulation framework uses GPU performance modeling based on runtime CPU code explorations on a streaming simulator. This platform independent simulation infrastructure, on the one hand, is very useful for the design space explorations for the future GPU devices and on the other hand, it can be used for performance evaluation of different applications on the existing GPU generation with good accuracy. The modules of *SArcs* are written in C and C++. These are enveloped inside a python script to run in an automated way which starts by grabbing the application source file and finalizes showing performance results. Some performance characterization results of the *SArcs* are shown in Figure 7.1 and explained in the next section (section 7.2). To the best of our knowledge *SArcs* tool is the first trace based GPU architectural simulator which can also be used independent of the requirements of having any kind of GPU environment.

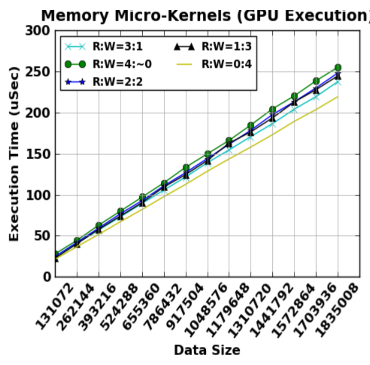
In this chapter, the first type of evaluations for the GPU like devices explore different architectural aspects against a base line established for NVIDIA's Fermi architecture (GPU Tesla C2050). The explored aspects include the performance effects by the variations in the configurations of *Streaming Multiprocessors*, *Global Memory Bandwidth*, *Channels between SMs down to Memory Hierarchy* and *Cache Hierarchy*.

The explorations are performed using application kernels from Vector Reduction, 2D-Convolution, Matrix-Matrix Multiplication and 3D-Stencil computations. The results show that the configurations of the computational resources for the current Fermi GPU device can deliver higher performance with further improvement in the global memory bandwidth for the same device.

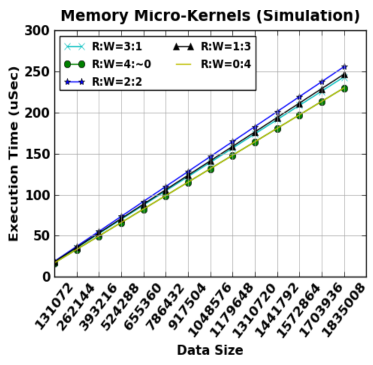
In the second type of evaluations, we present design space explorations for the streaming architectures with application specific configurable frond-end. These explorations are based on a concept of *Blacksmith Computing* performed on a Blacksmith Streaming Architecture (*BSArc*). The *Blacksmith Computing* uses a forging front-end to efficiently manage data according to the application nature. A large set of simple streaming processor in the back-end can fetch this arranged data to run computations on it. This computing concept is generic and adoptable for different target platforms. However, in this work we apply this concept to a SIMT execution model and present it as a part of a modified GPU like device. Our design space explorations for the *BSArc* suppose a configurable front-end in a GPU like device. The accuracy of the base line simulator is established against the NVIDIA's Fermi architecture (GPU Tesla C2050). We evaluate the performance difference for the Blacksmith Compute model based design approach against the standard L2 cache in the modified GPU like device. In our evaluations we use three application kernels from 2D-FFT, Matrix-Matrix Multiplication and 3D-Stencil. The results show that employing an application specific arrangement of data can achieve an average speedup of $2.3\times$ as compared to the usage of standard cache based design in a GPU like streaming architecture.

7.2 Effectiveness of the Design Space Exploration Tool

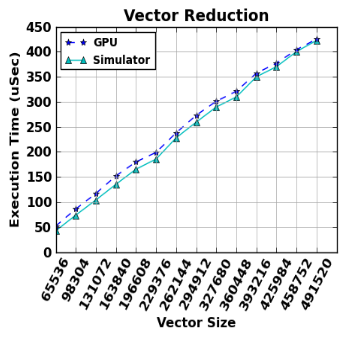
The simulator effectiveness is an important factor to be established before that one proceed for design space exploration for a target architecture using that simulator. The proposal on *SArcs* contributes in computer architecture research by providing an automated framework for simulations of streaming architectures like GPUs. *SArcs* can be used either as a standalone system – completely independent of a streaming environment – or it can be connected to other existing simulation related tools due to its modular nature. *SArcs* as an independent simulation infrastructure for GPUs does not require to have a physical GPU or any GPU related software tool-chain.



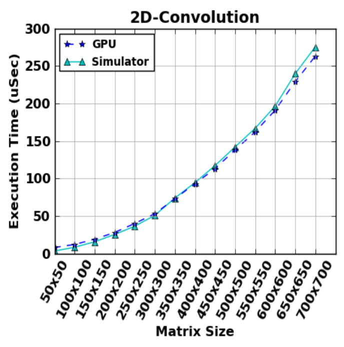
(a)



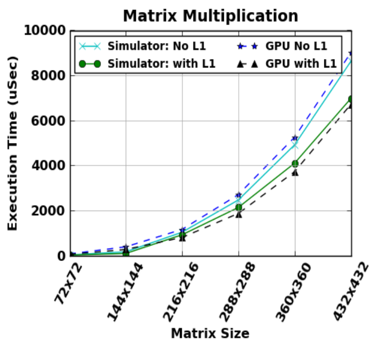
(b)



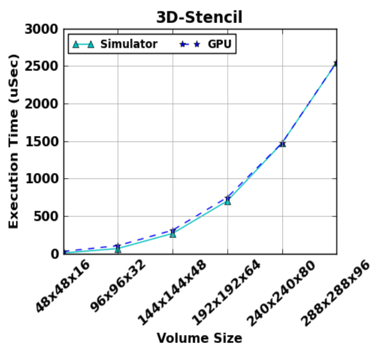
(c)



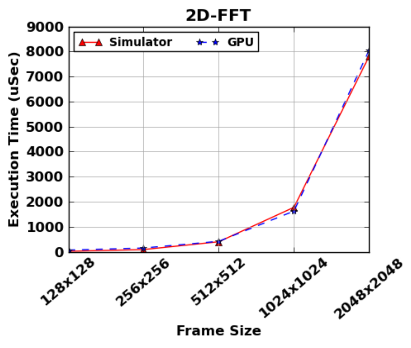
(d)



(e)



(f)



(g)

Figure 7.1: Establishment of the effectiveness of the simulator (SArcs) by performance characterization against the real GPU for the base line architecture (NVIDIA's Tesla C2050) (a) Memory Micro-Kernels (real GPU Executions) (b) Memory Micro-Kernels (Simulated Executions) (c) Vector Reduction using shared Memory (d) 2D-Convolution using shared memory (e) Matrix Multiplication with/without L1 (f) 3D-Stencil Kernel using shared memory (g) 2D-Fast Fourier Transform

SArcs is a trace driven simulation framework and exploits the fact that an application compiled for any architecture would require to transact the same amount of data with the main memory in the absence of registers or cache hierarchy. Moreover, the computations inside an application can be simulated by the target device latencies. The instruction level dependencies in GPU like architectural philosophy pose least impact on the performance because of zero-overhead switching between the stalled and large number of available threads. However, there could be cases where these dependencies can took longer time but the current version of *SArcs* is not accommodating these corner cases. *SArcs* creates an architectural correlation with the target device by passing the source code through a source to source translator followed by a thread aware trace generation. This trace is used by a device mapping process which transforms the trace into a SIMT trace specific for a GPU architecture. The SIMT trace is passed through a cycle accurate simulator to get the performance and related statistics.

The simulation results of *SArcs* and the reference results of real GPU (NVIDIA's Tesla C2050) based executions for the performance characterization of different application kernels are shown in the Figure 7.1 (a) to (g). The Memory Micro-Kernels shown in the Figure 7.1 (a) & (b) are used for the fine detailed analysis of the simulator targeting the evaluations of the simulator memory behavior. These memory micro-kernels are based on five different types of memory accesses during single execution of the kernel. We categorize these single kernel accesses in the ratio between consecutive reads (R) and writes (W). These ratios are R:W = 0:4 , 1:3, 2:2, 3:1 and 4:0. In order to avoid *nvcc* compiler from optimizing out the R:W=4:0 case, we use an external flag passed from command prompt to implement the kernel for only a conditional write. This flag always remain *false*. The descriptions of application kernels (Figure 7.1 (c) to (g)) are given in the sections 7.3.1 and 7.6. It can be observed that in all cases, the *SArcs* simulated results follow the real GPU based executions. The results for matrix-multiplication (MM) kernel also present the real and simulated behavior of L1 cache. Other kernels use shared memory to exploit data locality thus makes only a little use of L1 cache.

The simulation framework apply a large set of architectural optimizations as described in the chapter 6. The original results of the corresponding test cases show that the *SArcs* averaged error remains around 20% of the real executions on GPU. It is important to remember that the current version of SArcs is using CPU code projections for the GPU and one source of this error comes from compiling codes for different Instruction Set Architectures (ISA). The difference in compilation platforms appears in the form of different size of the compiled code which ultimately appears as a difference in the execution times. Other source of error include the choice of instructions for the trace from the huge set of CPU instructions at the trace generation phase. Moreover, a lack of the precise information regarding the micro-architectural details of the target GPU device also contributes in the error between the simulated and the real performance. However, the results for the design space explorations could be extrapolated with more accuracy. This is because our simulation results deviates from the real ones with a constant factor for each kernel. In our design space explorations, we adjust the baseline results – as in the Figure 7.1 (a) to (g) – with single constant factor for every kernel to make the results matching the real executions with an error less than 5%. We use the same constant factor for each kernel results during the design space exploration process.

7.3 Design Space Explorations for GPU

In our explorations for GPU like streaming architectures, we use four application kernels covering one dimensional (1D), 2D and 3D types of data accesses. A brief description of application kernels, the base line GPU configuration and the test platform is given in the following:

7.3.1 Application Kernels

In our tests for the various architectural configurations of GPU like device, we use Vector Reduction (VR), 2D-Convolution (CV), Matrix Matrix multiplication (MM), and 3D-Stencil (ST) kernels. The implementations for the two kernels (RD and ST) uses configurations for the shared memory usage. However, the MM and CV kernels do not use shared memory and the performance benefits for these applications only comes from the reuse of data in the standard L1 and L2 caches. The vector reduction kernel uses shared memory along with multiple invocations of the the GPU device during the reduction process of the whole vector to a single value. The convolution kernel uses a constant filter of size 5×5 to be convolve with various sizes of 2D image data sets. The 3D-Stencil kernel implements an odd symmetric stencil of size $8 \times 9 \times 8$. The choice of a kernel implementation is to have diversity in data access patterns and computations from the other kernel.

7.3.2 Base Line Architecture

In our design space explorations, *SArcs* simulation infrastructure uses a base line architecture for NVIDIA's GPU of Tesla C2050. This device belongs to Fermi generation [22] of GPUs which is the most recent architecture from NVIDIA. This device has 14 Streaming Multiprocessors (SMs) each contains 32 streaming (scalar) processors. The device is capable of performing 32 fp32 or int32 operations per clock cycle. Moreover, it has 4 Special Function Units (SFUs) to execute transcendental instructions such as sin, cosine, reciprocal, and square root. On the memory hierarchy side the device supports 48 KB / 16 KB Shared memory, 16KB / 48 KB L1 data cache and 768Kbytes of L2 memory.

7.3.3 Simulation Platform

The *SArcs* can be compiled for any host machine. The only constraint is that the PIN environment used in *TTrace tool* should have support for that CPU. In our evaluations, we use IBM "x3850 M2" machine. It has 48GBytes of main memory and 4 chips of Intel Xeon E7450, each one with 6 Cores running at 2.40GHz. This machine only helps us to run multiple instances of the simulation in parallel, otherwise a single core machine can be used for running single instance of the simulator. Further, in our case, the host machine uses x86_64-suse-linux and gcc compiler version 4.3.4. The target application kernels are compiled for optimization level 3 (switch -O3). On the GPU side, we use *nvcc* compiler with cuda compilation tool release 4.0, V0.2.1221. We compiled the CUDA codes using optimization level 3. Further, we use compilation switch *-Xptxas* along with *-dlcm=ca or -dlcm=cg* to enable and disable L1 cache accesses where ever needed.

7.3.4 Evaluated Architectural Configurations

Normally, the design space for a processor can be huge one based on the different combinations of the architectural configurations. Therefore, in a realistic way and to give a proof of concept along with some insight for the possible improvements in the current GPU generation, we choose four main architectural components of a GPU device for the experimentations and the explorations. The selection of various test configurations for each component are just based on our intuition and a user of our design space exploration tool can modify these according to one's own requirements.

7.3.4.1 Global Memory Bandwidth

On our base line architecture for the Fermi device, the global memory accesses are processed per warp bases. The maximum bandwidth achievable on the base line configuration is 144 GBytes/second. The memory controllers of the GPU device operates at a bit higher frequency as compared to the SMs operational frequency. This makes it possible that the throughput of the Global memory – in an ideal case – can reach to 128 Bytes/cycle (with respect to the the SM's frequency). The *DTL3* (Data Transaction Level 3) shown in the GPU Simulation Core (Figure 6.5) is responsible for the bandwidth scaling. In our evaluations, we test the global memory configurations in the ranges from $\times 1$ to $\times 10$ where the first-one is the base line bandwidth and the later-one is the 10 times of the base bandwidth.

7.3.4.2 Data Channels Between Memory Hierarchy and SMs

The Streaming Multiprocessors at the back-end of a GPU device do data transactions with the front-end memory hierarchy through multiple data channels. The *DTL2-Control* shown in the Figure 6.5 of *GSCore* handles these channels for the data transactions between the SMs and the memory hierarchy. In the base architecture, there are six channels. In our evaluations we increase and decrease the number of these channels to see their possible effect on the applications performance.

7.3.4.3 Cache Memory

Our base line device uses both L1/L2 cache hierarchy to cache the local and the global memory accesses. However, It is possible that both or anyone of these caches can be turned-on or turned-off at any time. Both caches are fully configurable for any cache size. However, the cache-line size is fixed. The cache line size for L1 cache is 128 bytes and it is 32 Bytes for the L2 cache. Moreover, these caches can be configured for two types of replacement policies: LRU and FIFO.

7.3.4.4 Streaming Multiprocessors

Streaming Multiprocessors (SMs) work as the vector processing units. This is the same as we model SMs in our simulation framework. The SM model in the GPU Simulation Core (GSCore) of our *SArcs* framework consists of *WARP Instruction and Latency (WIL) Scheduler*, Local memory, L1 cache and the *Data Transaction Level-1* control. Our simulator implements the L1 cache and Local memory separately. However, both of these in their functionality exactly behaves like a real NVIDIA's GPU. In order to be concise, we did not go for testing of all the internals of the SM rather than we simply vary the number of SMs in a GPU device to see how these changes effect the execution of the WARP instructions and eventually effect the overall performance of an application.

7.4 Results and Discussion

The results for the evaluated architectural configurations of a GPU like streaming device are shown in Figures 7.2 to 7.5. Here, before that we proceed to discuss the results, we define two terms being used in the discussion. These are the *SM WARP Instructions* and *Global WARP Instructions*. The general descriptions of the *WARP Instructions* formation are given in section-6.4. The *SM WARP Instructions* are the *WARP Instructions* which complete their execution phase inside an SM and the *Global WARP Instructions* consume cycles inside an SM and as well these are forwarded to the downside memory hierarchy. We are not calling the *Global WARP Instructions* as *Memory WARP Instructions* because if local memory is used inside an SM or there are read hits in the L1 cache then it is quite possible that a number of *Global WARP*

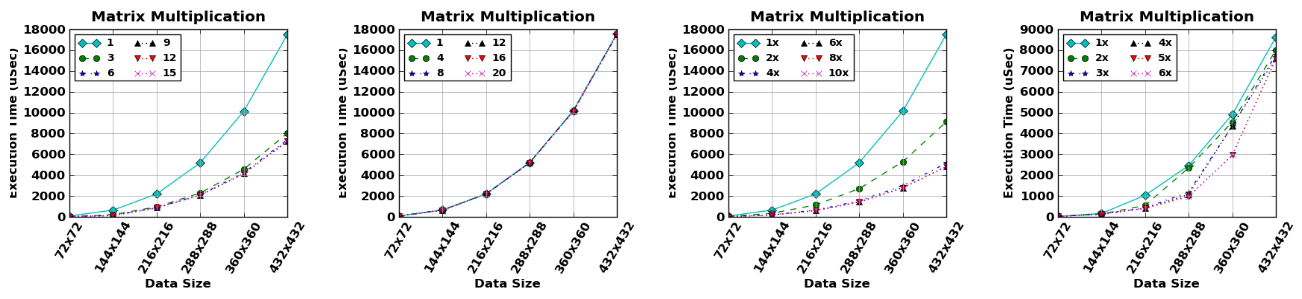


Figure 7.2: Matrix multiplication Kernel (No shared memory)

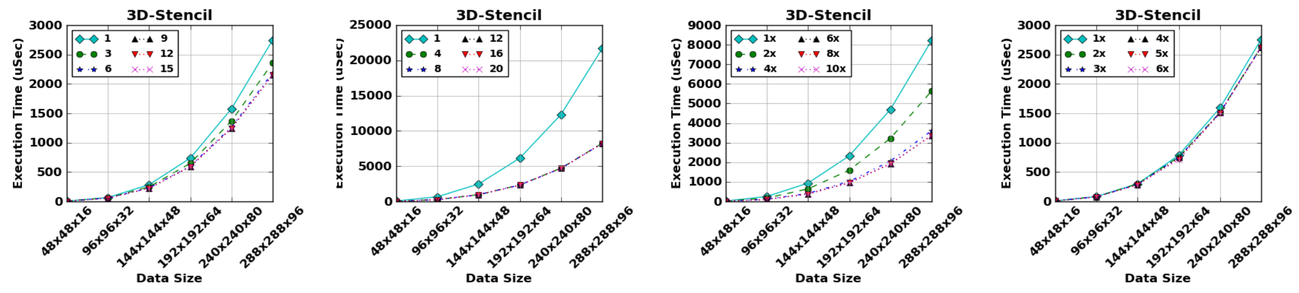


Figure 7.3: 3D-Stencil Kernel using shared memory

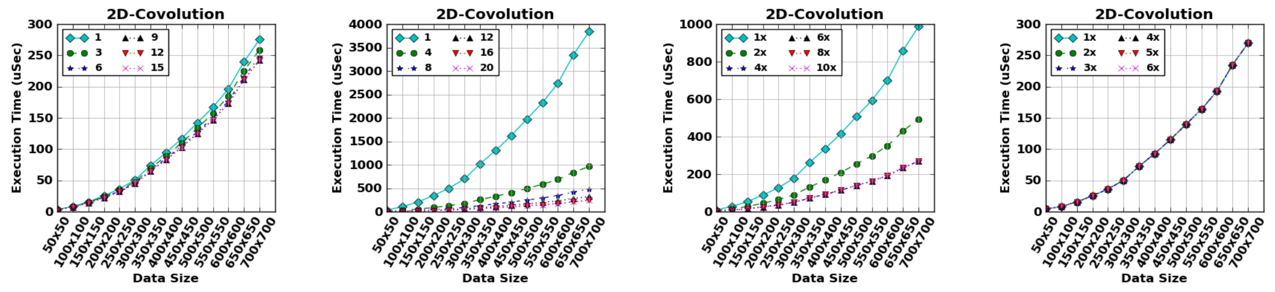
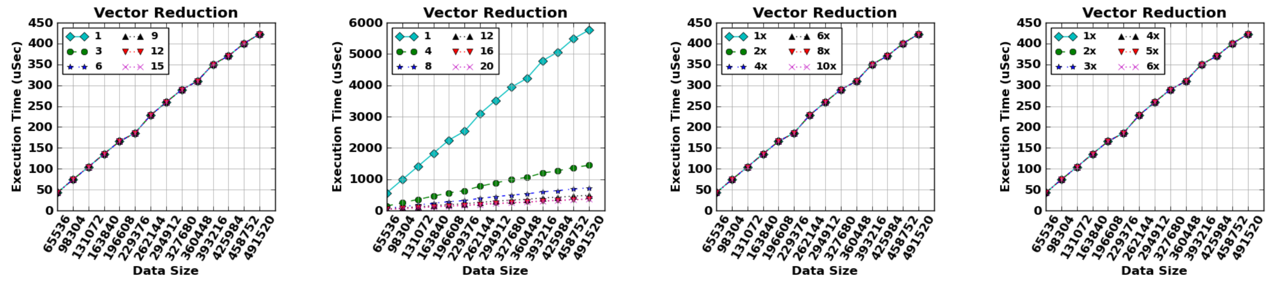


Figure 7.4: 2D-Convolution Kernel



(a) Channel Config. (b) Number of SM (c) Memory BW (d) L2 Cache

Figure 7.5: Vector Reduction using shared memory and multiple Invocations of the device

Instructions becomes *SM WARP Instructions*. All writes to the global memory are always categorized as part of *Global WARP Instructions*.

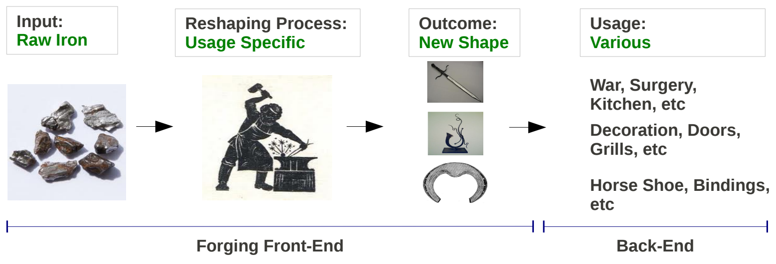
The effects of various channel configurations on the application kernels are shown in Figures 7.2(a), 7.3(a), 7.4(a) and the 7.5 (a). The usage of multiple channels from SMs on the top of a GPU are beneficial in two ways: (i) To keep busy the memory subsystem by forwarding data requests from various SMs (ii) To increase the Bandwidth of the system at L2 cache level. The results show that vector reduction kernel (Figure 7.5(a)) does not show any significant performance effect due channel variations. The basic reason for this behavior is that the reduction kernel uses local memory for the reduction process. In this case the reduction result for two values is reused with the next one and this process of reuse remain inside the shared memory. Ultimately only a single value is written back to the main memory for a single call to the device. Therefore the overall data required to transact with the global memory for this kernel is also very small. This means that the application kernel dominates with the *SM WARP Instructions* and does not show any effect with the channel variations. The same reason is true for the behavior of the reduction kernel for the corresponding results of the Memory Bandwidth and L2 cache shown respectively in the Figures 7.5(c) and (d). However, the reduction kernel shows performance improvements for the increase in the number of SMs as shown in Figure 7.5(b). This makes sense because the kernel is dominated by the *SM WARP Instructions* and increasing the number of SMs increase the parallelism in the execution. However, this performance due to parallelism with more number of SMs is saturated for 16 SMs because of the fixed channel configuration (6 in the base case) and the ultimate limit of the global memory bandwidth. On the other extreme, it can be seen that the matrix multiplication kernel does not show any effect for the Number of SMs as shown in the Figure 7.2(b). The MM kernel does not use local memory therefore this kernel dominates with the *Global WARP Instructions*. In this case the requests generated by a single SM saturates the memory sub-system (L2 and L1 are disabled in the test). Therefor, increasing the number of SMs does not show any significant variation in the results for the kernel.

The effects of various Global memory bandwidth configurations on the test kernels are shown in Figures 7.2(c), 7.3(c), 7.4(c) and the 7.5 (c). All the kernels except the reduction kernel respond to the increase in the memory bandwidth. The reason about the behavior of reduction kernel is already explained in the last paragraph. The effect of the bandwidth is saturated because of the limited number of channels used to transfer memory requests. The Figures 7.2(d), 7.3(d), 7.4(d) and the 7.5(d) shows the effects of L2 cache configurations. The 2D-convolution kernel only show negligible effect of L2 cache same as the reduction kernel. But here, the reason for this behavior of convolution kernel is that it uses only a small filter matrix (5×5) which gives only a little reuse as compared to the data set size.

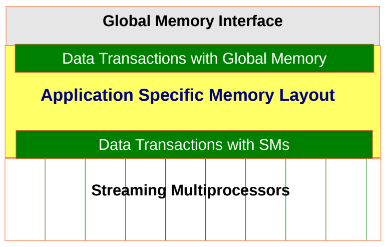
The rest of the results follow almost the same or the similar reasoning for their performance behavior as explained in the above two paragraphs. During our evaluations, we also tested L1 cache and the replacement policies. However, only the usage of L1 cache gives some performance benefits and in some cases shows even a little degradation in the results.

7.5 Blacksmith Computing

The basic concept of the blacksmith computing can be understood more easily from the working of a Blacksmith as shown in Figure 7.6(a). In this figure, one can see that the Blacksmith takes raw iron and hammer it to give a required shape depending upon the end-purpose of the produced item. Similarly, in blacksmith computing the raw input data (unprocessed data) is laid out inside a specialized front-end memory so that the algorithm running at the streaming multiprocessors in the back-end of the compute device could use this arranged data in an efficient way.



(a)


Global Memory Interface
Data Transactions with Global Memory
Application Specific Memory Layout
Data Transactions with SMs
Streaming Multiprocessors

(b)

Figure 7.6: (a) An Analogy for the Blacksmith Computing (b) The simplified target platform model

7.5.1 Target Platform Model

The mapping of application designs on a GPU device for performance is not an easy task. Every application can require a different set of optimizations and fine tuning to achieve an acceptable level of performance. Furthermore, the stringent hardware restrictions do not allow the programmer to fetch data efficiently using different pattern based approaches. This painful exercise of experimentation and restricted ways of fetching data could be get rid off by facilitating a configurable front-end while using the similar simple configurations of the SMs (streaming multiprocessors) in the back-end. This configurable front-end is adjustable to layout data according to the nature of the application running on a target device.

A simplified target platform model for the Blacksmith computing is shown in Figure 7.6(b). This model follows the basic concept given in a proposal on a template based architecture for reconfigurable accelerators [8]. We embed the idea in a GPU like SIMT architecture which results in a heterogeneous device that could be high level programmable using a CUDA [13] like programing model while at the same time partially configurable. This device essentially results as a modified GPU with a configurable forging front-end. However, the computing cores in the back-end of the target platform model are kept similar to the existing GPU architecture with *WARP* as the fundamental unit of dispatch within a single SM. The new data front-end can reshape and unfold data-sets specific-to-an-application requirement by configuring and incorporating domain specific architectural templates developed by the domain experts. The memory layouts for the forging front-end could even be common for various application kernels [11]. This means that the programmer does not need to worry about the hardware constraints and as well the difficult task of software tuning for the modified GPU device.

In order to perform design space explorations for the proposed compute model we develop a trace driven GPU simulator as explained in the Chapter 6. This simulation framework uses CPU code projections for the GPU performance modeling on a streaming simulator. We use this simulator to evaluate GPU with a configurable L2 cache in the device's front-end as shown in the Figure 7.7. This L2 cache can be configured either as a standard cache or it can be modeled as an application specific memory. This platform independent simulation infrastructure, on the one hand, is very useful for the

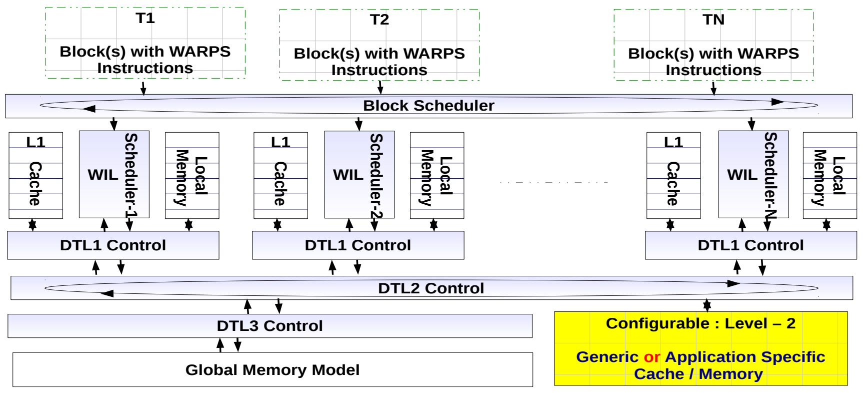


Figure 7.7: GPU Simulation Core (GSCore) with configurable L2

design space explorations for the future GPU devices and on the other hand, it can be used for performance evaluation of different applications on the existing GPU generation with a high accuracy. The modules of *SArcs* are written in C and C++. These are enveloped inside a python script to run in an automated way which starts by grabbing the application source file and finalizes showing performance results.

7.6 Application Specific Front-Ends

In order to explore the potential benefits of Blacksmith computing, we use three example application kernels from 2D-FFT, Matrix-Matrix Multiplication and 3D-Stencil. These kernels use either 2D or 3D data sets. In general, the efficient handling of data in 2D and 3D create a complex problem as compared to dealing single dimensional vectors. Moreover, each of these kernels use data in an arrangement very different from the other one. We show specialized memory layouts selected for each kernel in the Figures 7.8, 7.9 and 7.10. However, one can choose some other layout according to ones own requirements.

It is very important to mention that many data dependent application kernels may not get any benefit from the specialized memory layouts. In these cases, we consider that the best application specific memory layout will be like a standard L2 cache to utilize randomly available data locality. Furthermore, there also exist some strictly sequential algorithms. We consider these algorithms as not architected for the throughput

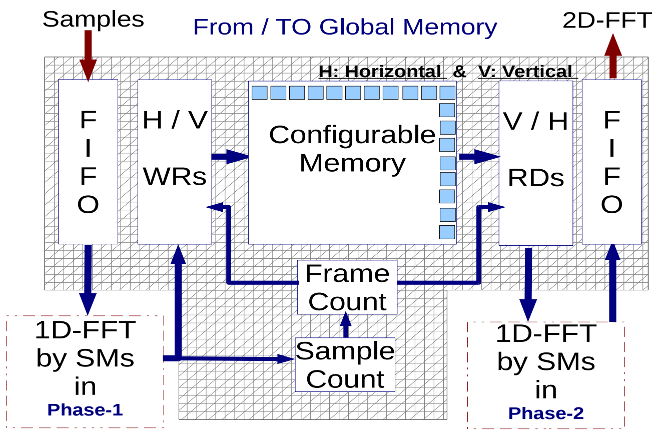


Figure 7.8: 2D-FFT Memory Layout

oriented streaming architectures.

7.6.1 2D-FFT

The shaded area in the Figure 7.8 shows the specialized front-end memory design for 2D-FFT. The complete design is based on two main parts: the data management part (shaded region) and the 1D-FFT computational part using streaming multiprocessors (SMs). The data is processed for 2D-FFT in two phases shown as phase-1 and phase-2 in the Figure 7.8 . Both phases run 1D-FFT on the orthogonal dimensions of the frame. These phases are executed in the same call to the device. However, their execution occur in a sequential order. The data management part maintains internal 2D-Frames for transposed accesses by the 1D-FFT executed in the phase-2. The internal 2D-Frames are managed by toggling the writing (WR) and reading (RD) sides for the horizontal and vertical order of the configurable memory on the alternative frames. We show – as an example – the horizontal and vertical memory blocks which are dual ported to help their accesses in two different orders. The size of individual memory block and the number of independent memory blocks is generated according to the X (*Points*) and Y dimensional parameters for the input frames. It is important that the the size of the data frames needing 2D-FFT should fit inside the specialized memory design.

During the phase-1, frame data is processed for 1D-FFT and written to the dual ported memory blocks in *H* or *V* order while during the second execution phase for another 1D-FFT, this memory is read in the reverse order that is *V* or *H*. This way the specialized memory design helps a faster 1D-FFT for the orthogonal dimension by providing all data available in a fully ordered way at the level-2 of the memory hierarchy. The hardware support for two dimensional accesses of memory also simplifies the program and the programmer's job.

7.6.2 Matrix Multiplication

The data accesses in Matrix-Matrix multiplication requires – in general – two basic optimizations: transposed access to one of the matrices and retaining a vector data from a matrix (row vector) for longer period of time to be computed with all the columns of the other matrix. We opt for the similar specialized memory design as proposed in the work for the template based systems [8]. This memory design is efficient for large sized matrices processed by streaming processors similar as in our case. The specialized memory design for our modified GPU is shown in the Figure-7.9. In this implementation, the matrices are accessed in the same "row major order" from the external memory. The matrices A and B are fetched in the order of one row and multiple columns. During the run, one row of matrix-A is fetched from the external memory into a single circular buffer. It is used element by element while the fetched row from matrix-B is scattered around the multiple circular buffers proportional to the compute capability in the SMs of the GPU back-end. Therefore, the product of an element from the row of Matrix-A is done with multiple columns of Matrix-B. Each SM accumulates the results for the element wise product of allocated rows (Matrix-A) and the columns (Matrix-B).

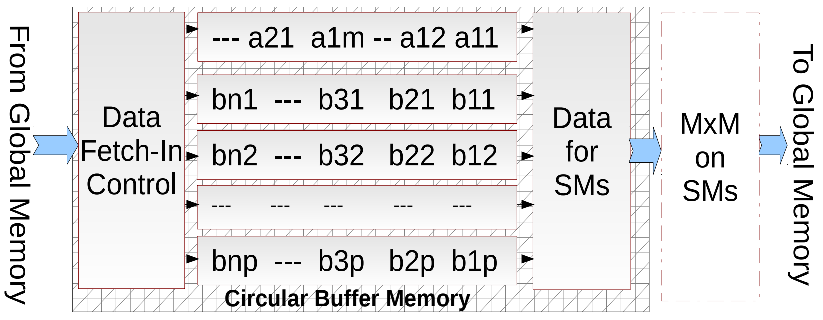


Figure 7.9: Matrix-Matrix Multiplication (MM) Memory Layout

7.6.3 3D-Stencil

A 3D-Stencil kernel operates on near neighboring points in three dimensions of a volume. We adopt a specialized memory architecture for the $8 \times 9 \times 8$ 3D-stencil from a work done by Araya et al. on RTM [3]. However, we modify the design according to the modified GPU needs as shown in Figure 7.10. The original specialized memory design consists of a specialized 3D memory layout and 3D write and read control corresponding to the three dimensions of the input volume. In our design we use only two dimensions with farthest points (Y-dim and X-dim receptively) while the consecutive data from the Z-dimension is processed inside the registers of the SMs.

The application specific memory layout for 3D-Stencil (Figure 7.10) show the first layer of memory labeled as **Plane** and corresponding to the Y-axis of the volume (therefore named *Y-layer*). This layer in the memory hierarchy consists of nine dual ported memory blocks. All nine planes in the layer are sequentially writable but possible to read in parallel. The second layer of memory is labeled as **Column** and corresponds to X-axis of input volume (named *X-Layer*). This layer has exactly the same features as that of Y-layer except that its size is equal to a column in a plane. The third memory layer corresponds to Z-axis (Z-layer) – as we mentioned earlier – is being managed inside the SMs. All these memory layers and their controls function in a way that SMs can access data from all the three dimensions as near to perfect parallel streams.

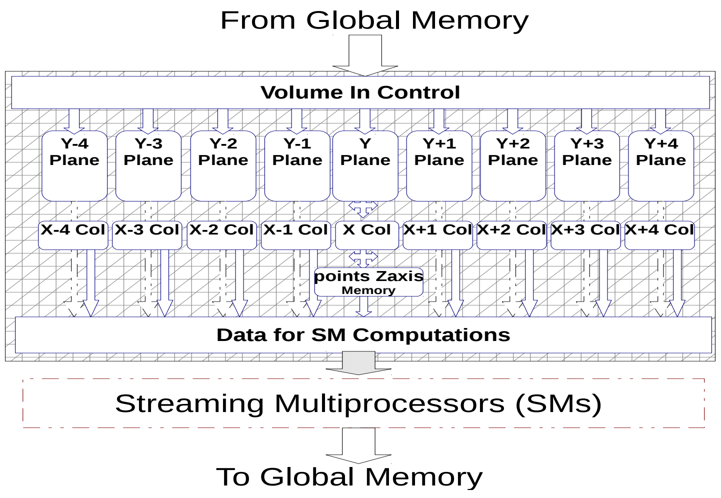


Figure 7.10: 3D-Stencil Memory Layout

7.7 Design Space Exploration Environment

In our explorations for the Blacksmith Compute model, we use three application kernels covering 2D and 3D types of data accesses. A brief description of application kernels and their related application specific memory layouts are given in 7.6. In the following we will introduce the base line GPU configuration and the test platform used in our design space explorations.

7.7.1 Base Line Architecture

In our design space explorations, *SArcs* simulation infrastructure uses a base line architecture for NVIDIA's GPU of Tesla C2050. This device belongs to Fermi generation [22] of GPUs which is the most recent architecture from NVIDIA. This device has 14 Streaming Multiprocessors (SMs) each contains 32 streaming (scalar) processors. The device is capable of performing 32 fp32 or int32 operations per clock cycle. Moreover, it has 4 Special Function Units (SFUs) to execute transcendental instructions such as sin, cosine, reciprocal, and square root. On the memory hierarchy side the device supports 48 KB / 16 KB Shared memory, 16KB / 48 KB L1 data cache and 768Kbytes of L2 memory. The L2 cache module is replaceable with application specific memory models. The size of L2 cache is configurable to keep it compatible with the memory sizes used in the application specific memory layouts.

7.7.2 Simulation Platform

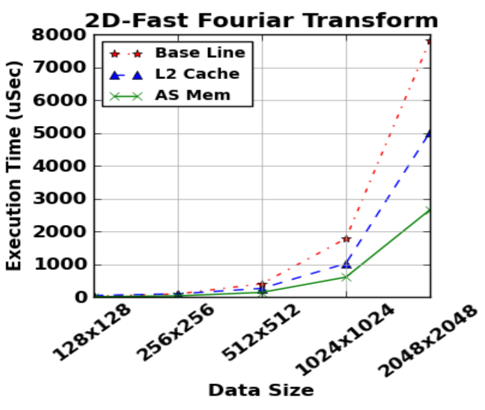
The *SArcs* can be compiled for any host machine. The only constraint is that the PIN environment used in *TTrace tool* should have support for that CPU. In our evaluations, we use Intel Xeon E7450 processor embed in IBM "x3850 M2" machine. The host machine uses x86_64-suse-linux and gcc compiler version 4.3.4. The target application kernels are compiled for optimization level 3 (switch -O3). On the GPU side, we use *nvcc* compiler with cuda compilation tool release 4.0, V0.2.1221. We compiled the the CUDA codes using optimization level 3. Further, we use compilation switch *-Xptxas* along with *-dlcm=ca or -dlcm=cg* to enable and disable L1 cache accesses where ever needed.

7.8 Results and Discussion

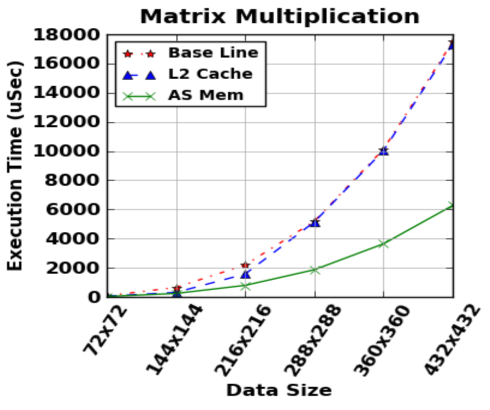
In our architectural explorations, we used three application kernels: 2D-FFT, Matrix-Matrix (MM) Multiplication and 3D-Stencil. The program configurations and optimizations for all these kernels use only registers inside an SM as the local memory resource. In all cases, we keep the size of the memory used for the L2-cache configurations equal to the size of memory used in application specific memory layouts. As compared to the original GPU configuration, the 3D-stencil uses same size of memory and MM needs only half of that for the largest data set. Due to the nature of the FFT algorithm, we use around 32MB of memory in simulation to retain a complete frame of complex FFT data for the largest execution (2048×2048 points). However, we consider it as a corner case. This is because, in general, contemporary algorithms for signal processing almost never require more than 64×64 point FFTs. This further indicates that the problem domains that could be decomposed into sub-domains are better suited for the proposed architecture. However, this constraint applies generally to all microprocessor architectures because of the upper limit on the size of processor's local storages and the cache memories.

The results for the evaluations of *BSArc* are shown in Figures 7.11 (a) to (d). All results in the figure include the execution time of an application kernel for the three configuration cases: (i) The base case: L2 cache off and no application specific memory (ii) L2 case: using only L2 cache (iii) Using only application specific (AS) memory. It can be observed that in all cases (in the case of MM only for small matrix sizes) the usage of L2 cache improves the performance for an application kernel as compared to the base line executions but *BSArc* based executions take a significant edge on the cache based performances. The basic reason for this performance impact is the increase in the locality and the parallelism of data according to the requirement of the application. However, this increase in the performance is not free as it comes at the cost of an increased architectural complexity. In this work we consider that the design of these specialized memory layouts is provided by the application domain experts in the form of templates. These templates are adjustable according to a device and the problem size at the device's configuration time.

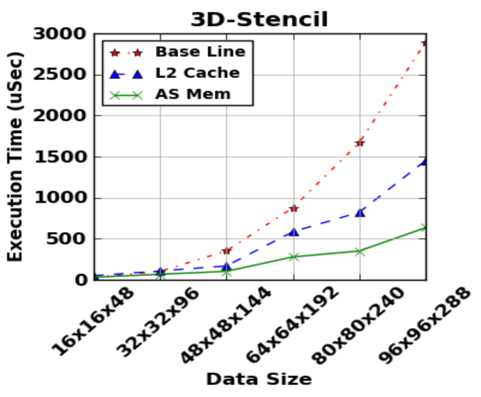
The Figure 7.11(d) shows the speedups achieved by using *BSArc*. These speedups for the test kernels are achieved by using the Application Specific (AS) memory front-



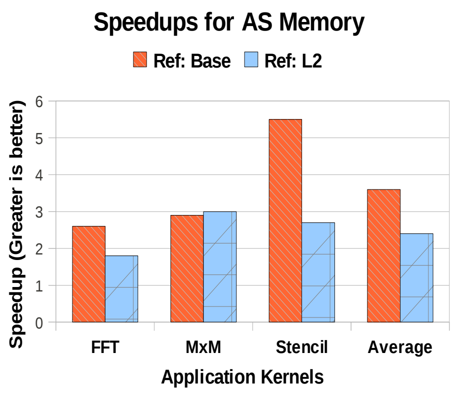
(a)



(b)



(c)



(d)

Figure 7.11: The application kernel's execution times for the three configurations : (i) Base Line (L2 Cache disabled and No Application Specific Memory) (ii) L2 Cache: Using only L2 cache (iii) AS Mem: Using only Application Specific (AS) memory. (a) 2D-FFT (b) Matrix Multiplication (c) 3D-Stencil (d) The speedups for the test kernels using Application Specific (AS) memory with reference (Ref) to: The base line (Base) architectural configuration and L2 Cache Based Executions

end with reference to the the base line execution and the L2 cache based executions. These results show that employing an application specific arrangement of data for these kernels achieves an average speedup of $3.6\times$ with reference to the base case. However, the impact of cache improves the performance of kernels therefore the relative speedup for the BSArc based configuration achieves $2.3\times$ compared to a GPU-like streaming device equipped with a standard cache.

7.9 Summary

The design and development of new computing architectures is not possible without well-focused design space explorations. This chapter present example explorations for the design of future GPU devices. Results show that the configurations of the computational resources for the current Fermi GPU device would still be enough for the newer designs. The current generation of GPUs can deliver higher performance with further improvements in the design of GPU's global memory for higher bandwidth and efficiency.

This chapter also present design space explorations for a conceptual Blacksmith Computing Architecture (BSArc). Blacksmith Computing using a Blacksmith Streaming Architecture (*BSArc*) gives an opportunity to exploit maximum possible data locality and the data level parallelism for an application. The results show significance of the efficient data management strategies for high performance computing. The generic methods like the standard cache hierarchies for improving the data locality may not achieve the potential performance benefits for an application. Therefore, the performance oriented devices might need to converge for a solution with more specialized memory front-ends.

The physical availability of *BSArch* like accelerators may still take time. However, development of precise architectural exploration tools like *SArcs* can be very useful for giving an insight and the design space explorations for new architectural proposals. Moreover, the specialized front-end designs might be able to support all kinds of applications. Further, these front-ends must communicate with the back-end across a standard interface. The changes in the front-end of a GPU like device would also require to extend the related programing models. These issues are further discussed as a part of our future work in the next – last – chapter of the thesis document.

8

Conclusions and Future Work

This chapter presents detailed conclusions of the research pursued during this thesis work. Moreover, it also through some light on future research and potential future targets.

8.1 Conclusions

—*— The current trend in high performance computing (HPC) systems focuses on parallel computing using either general purpose multi-core processors or multi-core streaming accelerators. However, the performance of these multi-cores is often constrained by the limited external bandwidth or by badly matching data access patterns. The latter reduces the size of useful data during memory transactions. A change in the application algorithm can improve the memory accesses but a hardware support mechanism for an application specific data arrangement in the memory hierarchy can significantly boost the performance for many application domains.

—*— The key to efficiency for many applications is to maximize the data-reuse fetching input data only once. This is also true for the stencil computations from the structured grid problem domain. We presented an application specific implementation of the stencil algorithm which not only shows how such a design can be achieved, it also demonstrates how this approach provides tremendous internal bandwidth to the compute units. We expect that the performance for the problems from the unstructured grids could also be boosted to a great extent by devising new memory ideas – for example Traversal Caches [110] – in these domains. Moreover, general purpose data caches can take a large portion of the power of a chip. This consumption of power can also be reduced by using more application specific memory layouts.

—*— The real life applications like RTM can get huge benefit from the streaming accelerators. In general terms GPUs, Cell/B.E. and the FPGAs – for the corresponding accelerator based implementations of RTM – outperform traditional multi-cores by one order of magnitude. However, to achieve this, a great development effort is required for the accelerators specially the porting of the design on a configurable device. This is because all operations need to be described in HDL. IP cores provided by Xilinx CoreGen were used to increase productivity. However, for the future, high-level productivity tools will be critical to allow developers harness the potential of FPGA technology. Moreover, application specific designs for specialized applications could also out-perform programmable accelerators if ported as high frequency ASIC devices.

—*— The complete generalization or the domain based generalization of an architecture for the application specific memories is an interesting topic. The proposals like *FEM* presented in this thesis show viability of such design ideas of a common memory layout. The idea of an application specific common memory layout also enables the conditional selection of multiple kernels, using the same or a subset of the layout. This configuration has the potential to result in a shared memory computational model promising a possibility of a greater data reuse across the kernels.

—*— Little focus has been given in the past on mapping domain specific abstraction onto the reconfigurable devices. Our presented DATE system is a step towards filling of this gap. The study on the DATE system show that the domain abstractions are an efficient way of handling complex applications. These enable high performance by keeping the developers from handling low level system details. *DATE* system like approaches have the potential to support the scalability of the architectural designs by just varying few input parameters. This also enables the portability of accelerator architectures to various sizes of small and large reconfigurable devices. Further, the standard output generated by the such systems makes it platform independent. *DATE* like systems use a library based approach to maintain the templates. This gives an opportunity to various related research groups to use the library for their own research tools and also to participate in writing the template designs to rapidly populate a common library from various application domains.

—*— The developments towards a unified reconfigurable accelerator design (like TARCAD) that improves application design productivity and portability without constraining customization is very important. These unified accelerators use heterogeneous kind of programing that includes the low level coding (HDL) techniques, high level synthesis tools and as well micro-codes in order to provide interoperability and high customization for an application. Although, as we show in this thesis, TARCAD is more efficient than GPUs, final performance is often worse due to the slower operational frequencies of reconfigurable devices. Designing a reconfigurable GPU based on the TARCAD architecture is an interesting idea to improve the final performance as well.

—*— The new architectural explorations are not possible without accurate design space exploration tools. Therefore, the development of architectural exploration tools like *SArcs* are very useful for giving an insight and the design space explorations for new architectural proposals. These tools could also be very helpful for simulating the conceptual architectures which are not possible to fabricate in the near future due to the fabrication constraints. GPUs – being newer architectures – lacks for the availability of simulation infrastructures as compared to the simulation environments available for the general purpose processors. The simulation frameworks like *SArcs* are required to extend the research for GPU like throughput oriented streaming architectures. The findings presented in this thesis show that the idea of using CPU ISA projections over GPU ISA has a potential to provide researchers a platform-independent simulator to research GPU-like architectures.

—*— The simulation infrastructures play very important role in advancing computer architecture research for proposing state of the art new architectures. Our architectural explorations for GPU like device using the *SArcs* framework reveals that the configurations of the computational resources for the current Fermi GPU device would still be enough in the near future for the newer designs. The current generation of GPUs can deliver higher performance with further improvements in the design of GPU's global memory for higher bandwidth and efficiency.

—*— GPU like throughput oriented streaming architectures can be improved for their performance, efficiency and lesser pressure on the requirements of external bandwidth by using a GPU front-end to accommodate more efficient data organizations as compared to the standard cache hierarchy. This observation generates an idea of Blacksmith Computing. The concept of Blacksmith Computing using a Blacksmith Streaming Architecture (*BSArc*) gives an opportunity to exploit maximum possible data locality and the data level parallelism for an application. The related results emphasize the significance of adopting the efficient data management strategies for high performance computing. The generic methods like the standard cache hierarchies for improving the data locality may not achieve the potential performance benefits for an application. Therefore, the performance oriented devices might need to converge for a solution with more specialized memory front-ends.

—*— During the development of this thesis work, we researched for the different architectural aspects of the streaming accelerators with customized front-ends. The results are promising and motivates for further research and explorations in this direction.

8.2 Future Work

In the previous chapter (Chapter 7) of this thesis, we propose a streaming architecture which introduces a forging front-end to efficiently manage data. This front-end connects to a large set of simple streaming cores in the back-end by using a streaming programing model. The forging front-end is a configurable part. This data front-end reshapes and unfold data sets specific-to-an-application requirement by incorporating domain specific architectural templates. The computing cores in the back-end are the multiple sets of simple fabricated cores similar to the streaming processors (SP) in a GPU. We enclosed this proposal (chapter 7) with the name Blacksmith Streaming Architecture (*BSArc*) for highly efficient data accesses and throughput oriented computations.

The future research opportunity related to the *BSArc* is to find-out ways of designing a generic front-end memory with application specific support. This would further need a supporting programing model. Therefore, the future research might be focused on two domains : (1) Design of a unified front-end memory for the Blacksmith Streaming Architecture (BSArc) and (2) *CUDA**b*** programing model which will be an extended CUDA programing model for supporting Blacksmith architecture containing unified front-end memory.

8.2.1 Unified Front-End memory for BSArc

The main benefit of the Unified Front-End memory for the future *BSArc* is that the same memory can be selected or configured in a coarse grained way for three different configurations. The first configuration will support regular data applications in the shape of application specific memory, the second configuration – being the standard cache – can help applications with irregular data accesses and the last configuration allows a user to play with application data by using the memory as scratch-pad. These three configurations are given below:

(i) Application Specific Streaming Cache

(ii) Standard Cache

(iii) Scratch-Pad Memory

A top level view for one of the possible architectural proposal for the future research of a unified front-end memory is shown in Figure 8.1. This architectural proposal is only presented here for the motivational purpose and as well to highlight the future lines of research. Some details on the figure are given by expanding the circled components and a brief description is given in the following.

The basic concept behind this multi-memory level architecture for the application specific streaming cache configuration is to provide arranged data sets (in the form of streams) accessible by the 3D-indexing from the *CUDAb* programing model. The size of memory in each level is in orders of magnitude smaller from one to another with largest size allocated to the first level and the smallest size allocated to the last (lowest) level. *BSArc* with unified memory scheme uses *CUDAb* program running on SMs in the back-end while a firmware micro-coded program or a specialized configuration works for handling data in the unified memory. This micro-code or the configured

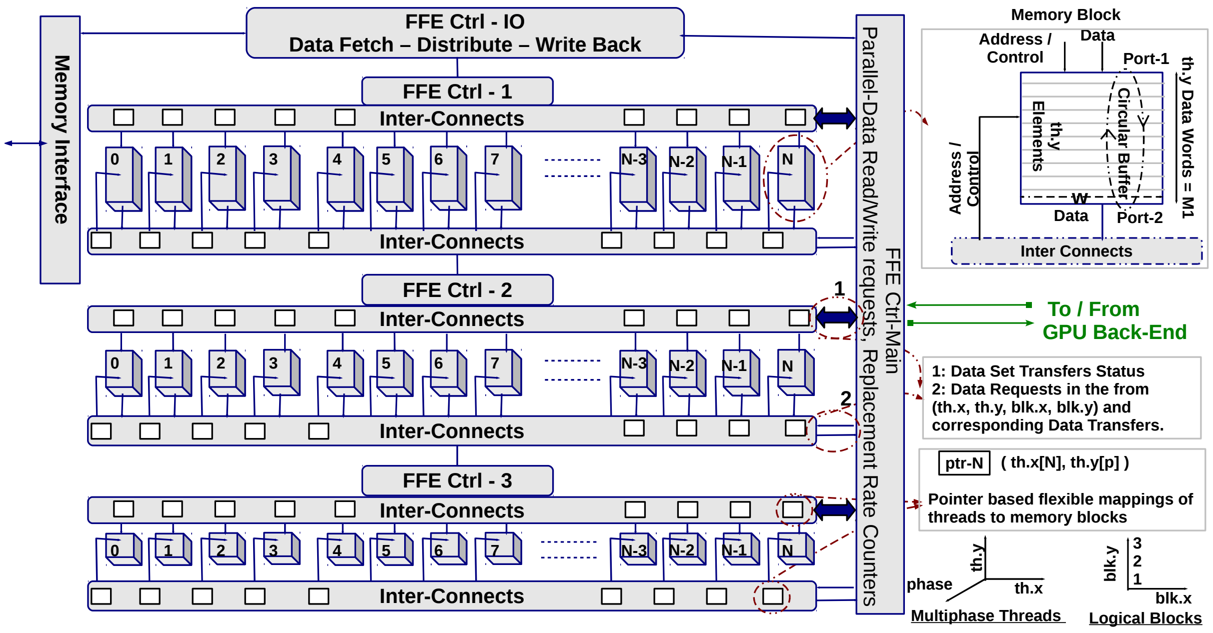

Memory Interface
FFE Ctrl - IO
Data Fetch – Distribute – Write Back
FFE Ctrl - 1
FFE Ctrl - 2
FFE Ctrl - 3
Inter-Connects
0 1 2 3 4 5 6 7 N-3 N-2 N-1 N
FFE Ctrl-Main
Parallel-Data Read/Write requests, Replacement Rate Counters
Memory Block
Address / Control
Data
Port-1
Port-2
th.y Elements
Circular Buffer
W
th.y Data Words = M1
Inter Connects
To / From GPU Back-End
1: Data Set Transfers Status
2: Data Requests in the from (th.x, th.y, blk.x, blk.y) and corresponding Data Transfers.
ptr-N (th.x[N], th.y[p])
Pointer based flexible mappings of threads to memory blocks
phase
th.y
th.x
Multiphase Threads
blk.y
blk.x
Logical Blocks

Figure 8.1: Unified Front-End Memory for BSArc (Under Consideration)

hardware is shown as `FFE Ctrl` (Firmware Front-End Control). In the case of application specific streaming cache configuration, the pre-arrangement of data in the unified front-end memory makes it unnecessary for these applications to hide memory latencies by switching between the large number of threads in the current CUDA based GPU device. Therefore, in order to keep busy all the streaming processors (SPs) inside SMs with the application specific streaming cache configuration, one needs only the number of threads equal to the number of SPs in a GPU. In the specialized memory configuration, each level of the memory keeps the number of memory blocks equal to number of threads. This means that a thread can access data from the three levels. Moreover, the threads can perform parallel data accesses from the memory by accessing different blocks.

In the standard cache configuration, the top level – with largest memory size – keeps the real data as cache lines by combining memory blocks. The lower two levels are configured to keep *tags* and other meta data. Moreover, the tags and data, both are maintained in the form of multiple memory banks. In this standard cache configuration for the modified GPU (*BSArc*), an application will use the usual CUDA programing concepts by using large number of threads to hide the memory latencies.

The Unified Front-End memory for BSArc when configured as scratch-pad memory, *CUDAb* programs will directly manage this with a software prefetching mechanism. This is similar to the usage of current local memory inside an SM but with a difference that new scratch-pad memory is now shared by all SMs as a common resource.

8.2.2 CUDAb

CUDA programing model with extensions to support Blacksmith Architecture is named as *CUDAb* where *b* stands for the *Blacksmith*. This programing model will provide a strong software support for the *BSArc* design that uses a unified memory front end (section 8.2.1). The standard cache configuration and the scratch pad memory configuration will allow *CUDAb* to work just like *CUDA* model. However, in the case of specialized memory configuration *CUDAb* use special concepts. In this concept, the block and thread indexes are considered as members of different data objects maintained inside the specialized memory configuration. A data object will be represented as a combination of one or more than one memory blocks. An object can have multiple dimensions which could be in the range of 1 dimension to 3 dimensions for the three memory levels of the current proposal of *BSArc*. It is also possible that different levels of unified memory can act together as object(s). The data movement between these objects and the global memory and the synchronization issues are handled by the FFE controls of the unified memory. These memory operations on the data of an object work like object's methods in the object oriented terms.

Publications

I Publications

1. Exploiting Memory Customization in FPGA for 3D Stencil Computations; Muhammad Shafiq, Miquel Pericas, Raul de la Cruz, Mauricio Araya-Polo, Nacho Navarro and Eduard Ayguade;

   IEEE International Conference on Field-Programmable Technology (FPT'09), Sydney, Australia, 9-11 December 2009.

2. FEM : A Step Towards a Common Memory Layout for FPGA Based Accelerators; Muhammad Shafiq, Miquel Pericas, Nacho Navarro, Eduard Ayguade;

   IEEE International Conference on Field Programmable Logic and Applications, Milano, ITALY, 31 August 02 September 2010.

3. Assessing Accelerator-based HPC Reverse Time Migration; Mauricio Araya-Polo, Javier Cabezas, Mauricio Hanzich, Felix Rubio, En- ric Morancho, Isaac Gelado, Muhammad Shafiq, Miquel Pericas, Jose Maria Cela, Eduard Ayguade, Mateo Valero;

   IEEE Journal Transactions on Parallel and Distributed Systems, January 2011.

4. A Template System for the Efficient Compilation of Domain Abstractions onto Reconfigurable Computers; Muhammad Shafiq, Miquel Pericas, Nacho Navarro and Eduard Ayguade;

   HiPEAC, WRC 2011, January 23, 2011 Heraklion, Greece.

5. TARCAD: A Template Architecture for Reconfigurable Accelerator Designs; Muhammad Shafiq, Miquel Pericas, Nacho Navarro, Eduard Ayguade;

   IEEE Symposium On application Specific Processors. San Diego, CA, June 5-10 2011

6. A Template System for the Efficient Compilation of Domain Abstractions onto Reconfigurable Computers; Muhammad Shafiq, Miquel Pericas, Nacho Navarro and Eduard Ayguade;

   Accepted for the Elsevier Journal of System Architecture 2012 [Pending publication].

7. BSArc: Blacksmith Streaming Architecture for HPC Accelerators ; Muhammad Shafiq, Miquel Pericas, Nacho Navarro, Eduard Ayguade;

   Accepted in ACM International Conference on Computing Frontiers, Cagliary Italy; May 15th, 2012.

II Other Papers and Extended Abstracts

1. PPMC : A Programmable Pattern based Memory Controller; Tassadaq Hussain, Muhammad Shafiq, Miquel Pericas, Nacho Navarro, Eduard Ayguade;

   IEEE/ACM International Symposium on Applied Reconfigurable Computing, March 2012, Hong Kong.

2. HLL Containers as a Way of Efficient Data Representation for Translation to FPGA; Based Accelerators, Muhammad Shafiq, Miquel Pericas, Nacho Navarro, Eduard Ayguade;

   Proceedings of ACACES 2010 Extended Abstracts: Advanced Computer Architecture and Compilation for Embedded Systems, Academia Press, Ghent, ISBN 978-90-382-1631-7, Terrassa, Spain, July 2010.

3. A Streaming Based High Performance FPGA Core for 3D Reverse Time Migration; Muhammad Shafiq, Miquel Pericas, Nacho Navarro, Eduard Ayguade;

   Proceedings of ACACES 2009 Extended Abstracts: Advanced Computer Architecture and Compilation for Embedded Systems, Academia Press, Ghent, ISBN 978-90-382-1467-2, Terrassa, Spain, July 2009.

III UPC Research Reports

1. Design Space Explorations for Streaming Accelerators using Streaming Architectural Simulator; Muhammad Shafiq, Miquel Pericas, Nacho Navarro and Eduard Ayguade;

   UPC research report: UPC-DAC-RR-2012-6, February 2012.

2. Performance Evaluation and Modeling of Smith-Waterman Algorithm on HPC Plateform; Muhammad Shafiq, Jorda Polo, Branimir Dickov, Tassadaq Hussain, Daniel Jimenez, Eduard Ayguade;

   UPC research report: UPC-DAC-RR-2010-8, April 2010.

3. A Hybrid Processor with Homogeneous Architecture for Heterogeneous Solutions; Muhammad Shafiq, Nacho Navarro, Eduard Ayguade;

   UPC research report: UPC-DAC-RR-2009-28, April 2009

4. H.264/AVC Decoder Parallelization in Context Of CABAC Entropy Decoder; Muhammad Shafiq, Mauricio Alvarez, Marisa Gil, Nacho Navarro; UPC research report: UPC-DAC-RR-2008-38, July 2008

References

[1] C. Meenderinck, A. Azevedo, B. Juurlink, M. Alvarez, and A. Ramirez, “Parallel Scalability of Video Decoders,” *Journal of Signal Processing Systems*, November 2009. xv, 4

[2] NVIDIA, “Tesla C2050 Performance Benchmarks,” Tech. Rep., 2010. [Online]. Available: www.siliconmechanics.com/files/C2050Benchmarks.pdf xvii, 106, 107

[3] M. Araya-Polo, J. Cabezas, M. Hanzich, M. Pericàs, F. Rubio, I. Gelado, M. Shafiq, E. Morancho, N. Navarro, E. Ayguadé, J. M. Cela, and M. Valero, “Assessing Accelerator-Based HPC Reverse Time Migration,” *IEEE TPDS*, 2011. xvii, 64, 86, 101, 106, 107, 151

[4] NVIDIA, “CUDASW++ on Tesla GPUs,” 2010. [Online]. Available: http://www.nvidia.com/object/swplusplus_on_tesla.html xvii, 106, 107

[5] N. Bell and M. Garland, “Efficient sparse matrix-vector multiplication on cuda,” *NVIDIA Technical Report NVR-2008-004*, Dec. 2008. xvii, 106, 107

[6] B. E. W. Page, “The landscape of parallel computing research: A view from berkeley,” March 2010. [Online]. Available: http://view.eecs.berkeley.edu/w/index.php?title=Main_Page&redirect=no 2, 134

[7] M. Garland and D. B. Kirk, “Understanding throughput-oriented architectures,” *Commun. ACM*, vol. 53, pp. 58–66, November 2010. 3

[8] M. Shafiq, M. Pericas, N. Navarro, and E. Ayguade, “TARCAD: A Template Architecture for Reconfigurable Accelerator Designs,” *IEEE Symposium On application Specific Processors. San Diego, CA*, June 2011. 3, 66, 79, 81, 147, 150

[9] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick, “Impact of modern memory subsystems on cache optimizations for stencil computations,” in *MSP ’05: Proceedings*

of the 2005 workshop on Memory system performance. New York, NY, USA: ACM, 2005, pp. 36–43. 3

[10] T. Hussain, M. Shafiq, M. Pericas, and E. Ayguade, "PPMC : A Programmable Pattern based Memory Controller," *IEEE/ACM International Symposium on Applied Reconfigurable Computing, Hong Kong*, March 2012. 5, 70

[11] M. Shafiq, M. Pericàs, N. Navarro, and E. Ayguadé, "FEM: A Step Towards a Common Memory Layout for FPGA Based Accelerators," *20th Intl. Conf. on FPL and Apps.*, Aug. 2010. 5, 96, 104, 147

[12] M. Shafiq, M. Pericàs, N. Navarro and E. Ayguadé, "A Template System for the Effcient Compilation of Domain Abstractions onto Reconfigurable Computers," *HiPEAC WRC, Heraklion Crete*, Jan 2011. 5, 93, 98

[13] "CUDA Programming Model." [Online]. Available: http://developer.nvidia.com/category/zone/cuda-zone 6, 114, 135, 147

[14] B. Khailany, T. Williams, J. Lin, E. Long, M. Rygh, D. Tovey, and W. J. Dally, "A Programmable 512 GOPS Stream Processor for Signal, Image, and Video Processing," *ISSCC*, 2007. 10

[15] Stanford, "Merrimac - Stanford Streaming Supercomputer Project." [Online]. Available: http://merrimac.stanford.edu/ 10

[16] FUJITSU, "High Throughput UltraSPARC T2/T2 Plus Processors." [Online]. Available: http://www.fujitsu.com/global/services/computing/server/sparcenterprise/technology/performance/processor3.html 10

[17] V.Michael Bove and John A. Watlington, "Cheops: A Reconfigurable Data-Flow System for Video Processing," 1995. [Online]. Available: http://web.media.mit.edu/~wad/cheops_CSVT/cheops.html 10

[18] D. Stokar, A. Gunzinger, W. GuggenbÃ¼hl, E. Hiltebrand, S. Mathis, P. Schaeren, B. Schneuwly, and M. Zeltner, "Sydama ii: A heterogeneous multiprocessor system for real time image processing." *CONPAR'90*, 1990. 11

[19] U. J. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, , and B. Khailany, "The Imagine Stream Processor," *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 2002. 11

[20] M. B. Taylor et al., "Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams," *International Symposium on Computer Architecture*, 2004. 12

[21] IBM, "Cell Broadband Engine Architecture (Version-1.02)," October 2007. 12

[22] NVIDIA, "Whitepaper : NVIDIA's Next Generation CUDA Compute Architecture," 2009. 13, 141, 152

[23] S. Hauck and A. DeHon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007. 20

[24] E. Baysal, D. D. Kosloff, and J. W. C. Sherwood, "Reverse time migration," *Geophysics*, vol. 48, no. 11, pp. 1514–1524, 1983. 21

[25] R. Baud, R. Peterson, G. Richardson, L. French, J. Regg, T. Montgomery, T. Williams, C. Doyle, and M. Dorner, "Deepwater gulf of mexico 2002: America's expanding frontier," *OCS Report*, vol. MMS 2002-021, pp. 1–133, 2002. 21

[26] J. P. Durbano, F. E. Ortiz, J. R. Humphrey, P. F. Curt, and D. W. Prather, "Fpga-based acceleration of the 3d finite-difference time-domain method," in *FCCM '04: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines.* Washington, DC, USA: IEEE Computer Society, 2004, pp. 156–163. 22

[27] C. He, W. Zhao, and M. Lu, "Time domain numerical simulation for transient waves on reconfigurable coprocessor platform," in *FCCM '05: Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines.* Washington, DC, USA: IEEE Computer Society, 2005, pp. 127–136. 22

[28] Xilinx, "Virtex-4 Family Overview (Product Specification)," *DS112-v3.0*, 2007. 22, 55, 57

[29] C. McNairy and R. Bhatia, "Montecito: A dual-core, dual-thread itanium processor," *IEEE Micro*, vol. 25, no. 2, pp. 10–20, 2005. 22

[30] E. Cohen, N. Rohrer, P. Sandon, M. Canada, C. Lichtenau, M. Ringler, P. Kartschoke, R. Floyd, J. Heaslip, M. Ross, T. Pflueger, R. Hilgendorf, P. McCormick, G. Salem, J. Connor, S. Geissler, and D. Thygesen, "A 64b cpu pair: Dual- and single-processor chips," in *Solid-State Circuits Conference, 2006. ISSCC 2006. Digest of Technical Papers. IEEE International*, Feb. 2006, pp. 333–342. 22

[31] H. P. Hofstee, "Power efficient processor architecture and the cell processor," in *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 258–262. 22

[32] M. Shafiq, M. Pericàs, R. de la Cruz, M. Araya-Polo, N. Navarro, and E. Ayguade, "Exploiting Memory Customization in FPGA for 3D Stencil Computations," *IEEE FPT*, Dec. 2009. 22, 48, 55, 57, 63, 82, 84, 88, 101

[33] F. Ortigosa, M. A. Polo, F. Rubio, M. Hanzich, R. de la Cruz, and J. M. Cela, "Evaluation of 3D RTM on HPC Platforms," *SEG Technical Program Expanded Abstracts*, vol. 27, pp. 2879–2883, 2008. 24

[34] G. Rivera and C.-W. Tseng, "Tiling optimizations for 3d scientific computations," in *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*. Washington, DC, USA: IEEE Computer Society, 2000, p. 32. 32

[35] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," *SIGPLAN Not.*, vol. 26, no. 6, pp. 30–44, 1991. 32, 39

[36] "3d seismic imaging through reverse-time migration on homogeneous and heterogeneous multi-core processors," *Scientific Programming*, vol. 17 (1-2), pp. 185–198, 2009. 33

[37] SGI, "Reconfigurable Application-Specific Computing User Guide," Tech. Rep., 2008. 33, 56, 93

[38] A. Ray, G. Kondayya, and S. V. G. Menon, "Developing a finite difference time domain parallel code for nuclear electromagnetic field simulation," *IEEE Transaction on Antennas and Propagation*, vol. 54, pp. 1192–1199, April 2006. 37

[39] S. Operto, J. Virieux, P. Amestoy, L. Giraud, and J. Y. L'Excellent, "3D frequency-domain finite-difference modeling of acoustic wave propagation using a massively parallel direct solver: a feasibility study," *SEG Technical Program Expanded Abstracts*, pp. 2265–2269, 2006. 38

[40] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick, "Impact of modern memory subsystems on cache optimizations for stencil computations," in *MSP '05: Proceedings of the 2005 workshop on Memory system performance*. New York, NY, USA: ACM Press, 2005, pp. 36–43. 38

[41] C. He, G. Qin, M. Lu, and W. Zhao, “An efficient implementation of high-accuracy finite difference computing engine on fpgas,” in *ASAP ’06: Proceedings of the IEEE 17th International Conference on Application-specific Systems, Architectures and Processors*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 95–98. 41

[42] M. Shafiq, M. Pericas, R. de la Cruz, M. Araya, N. Navarro, and E. Ayguade, “Exploiting Memory Customization in FPGA for 3D Stencil Computations,” in *FPT’09: Proceedings of the 2009 International Conference on Field-Programmable Technology*, 2009. 41

[43] J. Kelm, I. Gelado, K. Hwang, D. Burke, S.-Z. Ueng, N. Navarro, S. Lumetta, and W. Hwu, “Operating System Interfaces : Bridging the Gap Between CPU and FPGA Accelerators,” in *Intl. Symp. on FPGAs*, Feb. 2007. 48

[44] Y. Liang, Q. Meng, Z. Wang, and X. Guo, “Design of bit-stream neuron based on direct sigma-delat signal process,” *WCSP, International Conference on*, 2009. 48

[45] Y. Liu, K. Benkrid, A. Benkrid, and S. Kasap, “An fpga-based web server for high performance biological sequence alignment,” *NASA/ESA Conference on Adaptive Hardware and Systems*, 2009. 48

[46] W. Lin, Y. Tang1, B. Liu, D. Pao, and X. Wang, “Compact DFA Structure for Multiple Regular Expressions Matching,” *IEEE ICC*, 2009. 48, 63

[47] S. Kasap and K. Benkrid, “High performance phylogenetic analysis with maximum parsimony on reconfigurable hardware,” *IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS*, 2010. 48

[48] A. Jain, P. Gambhir, P. Jindal, M. Balakrishnan, and K. Paul, “Fpga accelerator for protein structure prediction algorithm,” *Programmable Logic, SPL. 5th Southern Conference on*, 2009. 48

[49] A. Y. JAMMOUSSI, S. F. GHRIBI, and D. S. MASMOUDI, “Implementation of face recognition system in virtex ii pro platform,” *International Conference on Signals, Circuits and Systems*, 2009. 48

[50] S. Reddy.P and R. Reddy.G, “Performance comparison of autocorrelation and cordic algorithm implemented on fpga for ofdm based wlan,” *International Conference on Communication Software and Networks*, 2009. 48

[51] O. Cheng, W. Abdulla, and Z. Salcic, "Hardware-software co-design of automatic speech recognition system for embedded real-time applications," *Accepted and To Be Published in IEEE Journal*, 2009. 48

[52] C. Chao, Z. Qin, X. Yingke, and H. Chengde, "Design of a high performance fft processor based on fpga," in *ASP-DAC '05: Proceedings of the 2005 Asia and South Pacific Design Automation Conference*. New York, NY, USA: ACM, 2005, pp. 920–923. 48, 54, 63

[53] J. Dongarra and et al., "International Exascale Software Project Roadmap (Draft 1/27/10 5:08 PM)," Nov. 2009. [Online]. Available: http://www.exascale.org/mediawiki/images/a/a1/Iesp-roadmap-draft-0.93-complete.pdf 49

[54] John Henry, "Operating System Interfaces to Reconfigurable Systems," *Master Thesis ; Department of ECE ; University Of ILLinois at Urbana-Champaign*, 2006. 64

[55] "IEEE Std. Verilog HDL." [Online]. Available: http://www.verilog.com/IEEEVerilog.html 64

[56] "VHDL AS. Group." [Online]. Available: http://www.vhdl.org/vhdl-200x/ 64, 65

[57] Z. Guo, W. Najjar, and B. Buyukkurt, "Efficient Hardware Code Generation for FPGAs," *ACM Trans. Archit. Code Optim.*, vol. 5, no. 1, pp. 1–26, 2008. 64, 73

[58] P. Coussy and D. Helle, "GAUT - High-Level Synthesis tool From C to RTL." [Online]. Available: http://www-labsticc.univ-ubs.fr/www-gaut/ 64, 73, 79, 92, 94

[59] X. Inc., "AutoESL." [Online]. Available: http://www.autoesl.com/ 64

[60] "C++ STL." [Online]. Available: http://www.cppreference.com/wiki/stl/start 64

[61] "BLITZ++: Object-Oriented Scientific Computing." [Online]. Available: http://www.oonumerics.org/blitz/ 65

[62] M. Blatt and P. Bastian, "The Iterative Solver Template Library," in *Applied Parallel Computing. State of the Art in Scientific Computing*. Springer Berlin / Heidelberg, 2007. 65

[63] B. Catanzaro et al., "SEJITS: Getting Productivity and Performance With Selective Embedded JIT Specialization," in *Tech Report No. UCB/EECS-2010-23*, UC Berkeley Parallel Computing Lab and L. Berkeley National Lab, March 1, 2010. 65

[64] J. Truchard, “Bringing FPGA Design to Application Domain Experts,” FPT 2010 Keynote, Tsinghua University, Beijing, 8-10 December 2010. 65

[65] T. M. Bhatt and D. McCain, “Matlab as a Development Environment for FPGA Design,” DAC 2005, Anaheim, California, USA, 13-17 June 2005. 65

[66] C. Kulkarni, G. Brebner, and G. Schelle, “Mapping a Domain Specific Language to a Platform FPGA,” DAC 2004, San Diego, CA, USA, June 2004. 65

[67] E. Rubow, R. McGeer, J. Mogul, and A. Vahdat, “Chimpp: A Click-based Programming and Simulation Environment for Reconfigurable Networking Hardware ,” ANCS’10, La Jolla, CA, USA, 25-26 October 2010. 65

[68] A. Vajda and J. Eker, “Return to the Language Forrest:the Case for DSL Oriented Software Engineering,” FoSER 2010, New Mexico, USA, November 7-8, 2010. 65

[69] “THRUST, a C++ Template Library for CUDA.” [Online]. Available: http://code.google.com/p/thrust/wiki/QuickStartGuide 65

[70] Xilinx, *ISE Design Suite CORE Generator IP Updates*. [Online]. Available: http://www.xilinx.com/ipcenter/coregen/updates.htm 65, 78, 103

[71] S. Sarkar, S. Dabral, P. K. Tiwari, and R. S. Mitra, “Lessons and experiences with high-level synthesis,” *IEEE Design and Test of Computers*, vol. 26, pp. 34–45, 2009. 66, 89

[72] A. Ketterlin and P. Clauss, “Prediction and trace compression of data access addresses through nested loop recognition,” in *CGO ’08: Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. New York, NY, USA: ACM, 2008, pp. 94–103. 67

[73] T. Hussain, M. Pericàs, and E. Ayguadé, “Reconfigurable Memory Controller with Programmable Pattern Support,” HiPEAC WRC, Heraklion Crete, Jan. 2011. 67, 95

[74] J. E. Smith, “Decoupled Access/Execute Computer Architectures,” in *ISCA ’82: Proceedings of the 9th annual symposium on Computer Architecture*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1982, pp. 112–119. 67

[75] J. Dongarra and et al., “International Exascale Software Project Roadmap (Draft),” Nov. 2009. [Online]. Available: http://www.exascale.org/mediawiki/images/a/a1/Iesp-roadmap-draft-0.93-complete.pdf 67

[76] “Xilinx Inc.” [Online]. Available: http://www.xilinx.com/ 78

[77] B. Buyukkurt, J. Cortes, J. Villarreal, and W. A. Najjar, “Impact of high-level transformations within the ROCCC framework,” *ACM Trans. Archit. Code Optim.*, Dec. 2010. 79, 92, 94

[78] S. Hauck and A. DeHon, “Reconfigurable computing: the theory and practice of FPGA-based computation,” November 2007. 93

[79] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer, “Piperench: a co/processor for streaming multimedia acceleration,” in *Proc. of the 26th annual intl. symp. on Computer arch.*, 1999. 93

[80] Y. Saito, T. Sano, M. Kato, V. Tunbunheng, Y. Yasuda, M. Kimura, and H. Amano, “Muccra-3: a low power dynamically reconfigurable processor array,” in *Proc. of 2010 Asia and South Pacific Design Automation Conf.*, 2010. 93

[81] J. Bormans, “ADRES Architecture - Reconfigurable Array Processor,” *Chip Design Magazine*, November 2006. 93

[82] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt, “Pact xpp – a self-reconfigurable data processing architecture,” *J. Supercomput.*, vol. 26, pp. 167–184, September 2003. 93

[83] J. Kelm, I. Gelado, K. Hwang, D. Burke, S.-Z. Ueng, N. Navarro, S. Lumetta, and W. mei Hwu, “Operating System Interfaces: Bridging the Gap between CPU and FPGA Accelerators,” *Intl. Symp. on FPGAs*, Feb. 2007. 93

[84] A. Brandon, I. Sourdis, and G. N. Gaydadjiev, “General Purpose Computing with Reconfigurable Acceleration,” *Intl. conf. on FPL and Applications*, 2010. 93

[85] C. C. Corporation, “The Convey HC-1: The Worldâs First Hybrid-Core Computer,” *HC1- Data Sheet*, 2008. 93

[86] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao, “The Chimaera reconfigurable functional unit,” *IEEE Trans. on VLSI Systems*, 2004. 93

[87] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, “The MOLEN Polymorphic Processor,” *IEEE Transactions on Computers*, vol. 53, pp. 1363–1375, 2004. 93

[88] L. Hasan, Y. M. Khawaja, and A. Bais, “A Systolic Array Architecture for the Smith-Waterman Algorithm with High Performance Cell Design,” *Proc. of IADIS Eu. Conf. on Data Mining*, 2008. 103

[89] B. Dickov, M. Pericàs, N. Navarro, and E. Ayguade, “Row-interleaved streaming data flow implementation of Sparse Matrix Vector Multiplication in FPGA,” in *4th Workshop on Reconfigurable Computing, WRC-2010*, 2010. 104, 109

[90] P. Sundararajan, “High Performance Computing Using FPGAs,” *WP (Xilinx): WP375 (v1.0) September 10, 2010.* 105

[91] “Top 500 Supercomputer Sites,” June 2011. [Online]. Available: http://top500.org/lists/2011/11 114, 135

[92] G. Caragea, F. Keceli, A. Tzannes, and U. Vishkin, “General-Purpose vs. GPU: Comparison of Many-Cores on Irregular Workloads,” *HotPar, Berkeley, CA*, June 2010. [Online]. Available: http://www.usenix.org/event/hotpar10/final_posters/Caragea.pdf 114, 135

[93] D. B. Kirk and W. mei W. Hwu, “Programming Massively Parallel Processors: A Hands-on Approach (Chapter-2),” *Published by Elsevier Inc*, 2010. 114, 135

[94] “Open Computing Language (OpenCL).” [Online]. Available: http://developer.nvidia.com/opencl 114, 135

[95] S. Asano, T. Maruyama, and Y. Yamaguchi, “Performance Comparison of FPGA, GPU and CPU in Image processing,” *IEEE FPL*, September 2009. 114, 135

[96] “SimpleScalar: .” [Online]. Available: http://pages.cs.wisc.edu/~mscalar/simplescalar.html 115

[97] “simics: .” [Online]. Available: https://www.simics.net/ 115

[98] “PTLsim: .” [Online]. Available: http://www.ptlsim.org/ 115

[99] “M5: .” [Online]. Available: http://www.m5sim.org/Main_Page 115

[100] “TaskSim and Cyclesim: .” [Online]. Available: http://pcsostres.ac.upc.edu/cyclesim/doku.php/tasksim:start 115

[101] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," *SIGARCH Comput. Archit. News*, June 2009. 115

[102] Sunpyo Hong and Hyesoon Kim, "An integrated GPU power and performance model," *ACM ISCA 10*, June 2010. 115

[103] Y. Kim and A. Shrivastava, "CuMAPz: A tool to analyze memory access patterns in CUDA," *ACM/IEEE DAC 2011*, June 2011. 115

[104] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," *IEEE ISPASS 09*, April 2009. 115

[105] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W. mei W. Hwu, "An Adaptive Performance Modeling Tool for GPU Architectures," *ACM PPoPP10*, January 2010. 115

[106] J. Meng, V. A. Morozov, K. Kumaran, V. Vishwanath, and T. D. Uram, "GROPHECY: GPU Performance Projection from CPU Code Skeletons," *ACM/IEEE SC11*, November 2011. 115

[107] "GpuOcelot: A dynamic compilation framework for PTX." [Online]. Available: http://code.google.com/p/gpuocelot/ 115, 131

[108] H. Kim, "GPU Architecture Research with MacSim ," 2010. [Online]. Available: http://comparch.gatech.edu/hparch/nvidia_kickoff_2010_kim.pdf 115

[109] "Pin - A Dynamic Binary Instrumentation Tool." [Online]. Available: http://www.pintool.org/ 120

[110] G. Stitt, G. Chaudhari, and J. Coole, "Traversal caches: a first step towards fpga acceleration of pointer-based data structures," in *CODES+ISSS '08: Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*. New York, NY, USA: ACM, 2008, pp. 61–66. 158

Declaration

I herewith declare that I have produced this work without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken over directly or indirectly from other sources have been identified as such. This work has not previously been presented in identical or similar form to any other Spanish or foreign examination board.

The thesis work was conducted from `November 2007` to `April 2012` under the supervision of Dr. Miquel Pericàs, Prof. Nacho Navarro and Prof. Eduard Ayguadé.

Muhammad Shafiq,
Barcelona, April 2012.