UNIVERSITY OF CALIFORNIA, SAN DIEGO

GUSTO: General architecture design Utility and Synthesis Tool for Optimization

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Ali Umut İrtürk

Committee in charge:

Ryan Kastner, Chair
Jung Uk Cho
Bhaskar Rao
Timothy Sherwood
Steven Swanson
Dean Tullsen

2009

The dissertation of Ali Umut İrtürk is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____

_____
Chair

University of California, San Diego

2009

DEDICATION

To my family (İale, Ömer, Bülent, Topsi)

EPIGRAPH

Let nothing perturb you, nothing frighten you. All things pass.
God does not change. Patience achieves everything.
—Mother Teresa

TABLE OF CONTENTS

Signature Page . . . iii

Dedication . . . iv

Epigraph . . . v

Table of Contents . . . vi

List of Figures . . . x

List of Tables . . . xviii

Acknowledgements . . . xix

Vita and Publications . . . xxi

Abstract of the Dissertation . . . xxiii

Chapter 1 Introduction . . . 1
1.1 Motivation . . . 1
1.2 Research Overview . . . 3
1.3 Organization of Dissertation . . . 6

Chapter 2 Parallel Platforms for Matrix Computation Algorithms . . . 9
2.1 Graphic Processing Units (GPUs) . . . 11
2.2 Massively Parallel Processor Arrays (MPPAs) . . . 17
2.2.1 Ambric Family Overview . . . 20
2.2.2 Ambric AM2045 Architecture . . . 22
2.3 Field Programmable Gate Arrays (FPGAs) . . . 23
2.3.1 Xilinx Virtex-4 Family Overview . . . 26
2.3.2 Xilinx Virtex-4 Architecture . . . 26
2.4 Learning from Existing Parallel Platforms . . . 27
2.4.1 Comparison of Parallel Platforms . . . 28
2.4.2 Roadmap for the Future Many-Core Platform . . . 35

Chapter 3 Overview of Design Tools . . . 38
3.1 System Generator for DSP . . . 39
3.2 AccelDSP Synthesis Tool . . . 46
3.3 Simulink HDL Coder . . . 51
3.4 C-based High Level Design Tools . . . 55
3.5 A Case Study for Filter Designs using Domain Specific High Level Design Tools . . . 62

3.5.1 Filter Design HDL Coder Toolbox 63
3.5.2 Xquasher 67
3.5.3 Comparison: Filter HDL Toolbox versus Xquasher 71
3.6 Roadmap for the Future Single/Many-Core Platform Generator Tool 78

Chapter 4 Matrix Computations: Matrix Multiplication, Matrix Decomposition and Matrix Inversion 83
4.1 Building Blocks of Matrix Computations 84
4.2 Matrix Decomposition and its Methods 91
4.2.1 QR Decomposition 91
4.2.2 LU Decomposition 93
4.2.3 Cholesky Decomposition 94
4.3 Matrix Inversion and its Methods 96
4.3.1 Matrix Inversion of Triangular Matrices 97
4.3.2 QR Decomposition Based Matrix Inversion 98
4.3.3 LU Decomposition Based Matrix Inversion 98
4.3.4 Cholesky Decomposition Based Matrix Inversion . 99
4.3.5 Matrix Inversion using Analytic Method 100

Chapter 5 GUSTO: General architecture design Utility and Synthesis Tool for Optimization 101
5.1 Flow of GUSTO 103
5.1.1 Error Analysis 107
5.2 Matrix Decomposition Architectures 110
5.2.1 Inflection Point Analysis 111
5.2.2 Architectural Design Alternatives for Matrix Decomposition Algorithms 113
5.3 Adaptive Weight Calculation Core using QRD-RLS Algorithm 115
5.3.1 Comparison 116
5.4 Matrix Inversion Architectures 117
5.4.1 Inflection Point Analysis 118
5.4.2 Architectural Design Alternatives for Matrix Inversion Architectures 124
5.4.3 Comparison 126
5.5 Conclusion 127

Chapter 6 GUSTO's Single Processing Core Architecture 129
6.1 Related Work 130
6.2 Automatic Generation and Optimization of Matrix Computation Architectures 133
6.2.1 Flow of Operation 134

6.2.2 Designing the General Purpose Processing Core . 135
6.2.3 Designing the Application Specific processing core 137
6.3 Designing a Multi-Core Architecture 145
6.3.1 Partitioning 147
6.3.2 Generation of the Connectivity Between Cores . . 149
6.4 Conclusion 150

Chapter 7 Hardware Implementation Trade-offs of Matrix Computation Architectures using Hierarchical Datapaths 151
7.1 Hierarchical Datapaths Implementation and Heterogeneous Architecture Generation using GUSTO 154
7.1.1 Hardware Implementation Trade-offs of Matrix Computation Architectures using Hierarchical Datapaths 155
7.1.2 Flow of GUSTO for Multi-Core Designs 157
7.2 Architectural Implementation Results of Different Matrix Computation Algorithms 164
7.2.1 Matrix Multiplication 166
7.2.2 Matrix Inversion 179
7.3 Conclusion 186

Chapter 8 FPGA Acceleration of Mean Variance Framework for Optimal Asset Allocation 190
8.1 The Mean Variance Framework for Optimal Asset Allocation 194
8.1.1 Computation of the Required Inputs 195
8.1.2 Mean Variance Framework Step 1: Computation of the Efficient Frontier 198
8.1.3 Mean Variance Framework Step 2: Computing the Optimal Allocation 199
8.2 Implementation of the Mean Variance Framework 201
8.2.1 Implementation Motivation 201
8.2.2 Hardware/Software Interface 203
8.2.3 Generation of Required Inputs - Phase 5 205
8.2.4 Hardware Architecture for Mean Variance Framework Step 1 205
8.2.5 Hardware Architecture for Mean Variance Framework Step 2 208
8.3 Results 210
8.4 Conclusions 212

Chapter 9 Future Research Directions 214

Appendix A Matrix Computations 215
A.1 Matrix Decomposition Methods 215
A.1.1 QR Decomposition 215
A.1.2 LU Decomposition 226
A.1.3 Cholesky Decomposition 229

Bibliography 233

LIST OF FIGURES

Figure 1.1: Design Flow of GUSTO. 4
Figure 1.2: Flow of GUSTO's trimming feature. 5
Figure 1.3: Hardware implementation of matrix multiplication architectures with different design methods using GUSTO. 6

Figure 2.1: GPU Architecture. 15
Figure 2.2: The design flow using CUDA for NVIDIA GPUs. 16
Figure 2.3: Multi-core CPU architecture. 18
Figure 2.4: Structural object programming model for Ambric Architecture. 19
Figure 2.5: Design Flow of Ambric MPPA and its steps: Structure, Code, Reuse, Verify, Realize and Test. 20
Figure 2.6: Ambric Architecture. 21
Figure 2.7: An FPGA architecture and its resources: I/O cells, logic blocks (CLBs) and interconnects. 24
Figure 2.8: Xilinx ISE design flow and its steps: design entry, design synthesis, design implementation and Xilinx device programming. Design verification occurs at different steps during the design flow. 25

Figure 3.1: Two simple design examples using System Generator for DSP are presented: multiply & accumulate and FIR filter. 40
Figure 3.2: Example blocks from System Generators' library. 42
Figure 3.3: Design and implementation of the Matching Pursuits algorithm for channel estimation using System Generator. Even a small change in the architecture affects blocks inside the design and requires a large amount of manual synchronization efforts. . . . 45
Figure 3.4: The design flow for AccelDSP. 48
Figure 3.5: Comparison of AccelDSP/AccelWare and Hand-Code/Coregen implementations for various signal processing algorithms: FFT, 10×10 matrix multiplication, FIR filter, CORDIC and *Constant False Alarm Rate (CFAR)* [156]. Results are presented in terms of area and required calculation time. 50
Figure 3.6: C-based high level design tools are divided into 5 different families including Open Standard: System-C; tools producing generic HDL that can target multiple platforms: Catapult-C, Impulse-C and Mitrion-C; tools producing generic HDL that are optimized for manufacturer's hardware: DIME-C, Handel-C; tools that target a specific platform and/or configuration: Carte, SA-C, Streams-C; tool targeting RISC/FPGA hybrid architectures: Napa-C. 56
Figure 3.7: Design Flow of Filter Design HDL Coder Toolbox. 64

Figure 3.8: An example of page and its lines, dots and BMs. Dots are shown with ellipses and a circle inside each dot demonstrates the BM part. 68
Figure 3.9: Xquasher Algorithm. 70
Figure 3.10: Comparison of Xquasher, Spiral and Filter Design HDL Coder Toolbox architectural results for 20 tap FIR filter designs. Different architectures designed using Design HDL Coder Toolbox include fully parallel, fully serial, partly serial, cascade serial and distributed arithmetic. Xquasher performs different number of term extraction: 2, 3, 4, 6, 8, 10 and infinite terms to find most suitable extraction method that provides best results in terms of area and throughput. 73
Figure 3.11: Comparison of Xquasher and Filter Design HDL Coder Toolbox architectural results for 30 tap FIR filter designs. Different architectures designed using Design HDL Coder Toolbox include fully parallel, fully serial, partly serial, cascade serial and distributed arithmetic. Xquasher performs different number of term extraction: 2, 3, 4, 6, 8, 10 and infinite terms to find most suitable extraction method that provides best results in terms of area and throughput. Spiral cannot generate filter that use more than 20 taps, therefore its results are not included. 74
Figure 3.12: Comparison of Xquasher and Filter Design HDL Coder Toolbox architectural results for 20 tap FIR designs. Different architectures designed using Design HDL Coder Toolbox include fully parallel, fully serial, partly serial, cascade serial and distributed arithmetic. Xquasher performs different number of term extraction: 2, 3, 4, 6, 8, 10 and infinite terms to find most suitable extraction method that provides best results in terms of area and throughput. Spiral cannot generate filters that use more than 40 taps, therefore its results are not included. 76
Figure 3.13: Comparison of Xquasher, Spiral and Filter Design HDL Coder Toolbox architectural results for 20 tap FIR designs that includes different coefficient multiplier optimization methods for Filter Design HDL Coder Toolbox. Different architectures designed using Design HDL Coder Toolbox include fully parallel (with Multipliers, CSD and Factored CSD), fully serial, partly serial, cascade serial and distributed arithmetic. Xquasher performs different number of term extraction: 2, 3, 4, 6, 8, 10 and infinite terms to find most suitable extraction method that provides best results in terms of area and throughput. 77

Figure 3.14: Comparison of Xquasher, Spiral and Filter Design HDL Coder Toolbox architectural results for 20 tap FIR filter designs that includes an option for use of pipeline registers for Filter Design HDL Coder Toolbox. Different architectures designed using Design HDL Coder Toolbox include fully parallel (with and without pipeline registers), fully serial, partly serial (with and without pipeline registers), cascade serial and distributed arithmetic (with and without pipeline registers). Xquasher performs different number of term extraction: 2, 3, 4, 6, 8, 10 and infinite terms to find most suitable extraction method that provides best results in terms of area and throughput. 78

Figure 4.1: The solution steps of the matrix inversion using QR decomposition. 98
Figure 4.2: The solution steps of the matrix inversion using LU decomposition. 99
Figure 4.3: The solution steps of the matrix inversion using Cholesky decomposition. 99
Figure 4.4: Matrix Inversion with analytic approach. The calculation of the first element of cofactor matrix, C_{11}, for a 4×4 matrix is shown. 100

Figure 5.1: Design Flow of GUSTO. 104
Figure 5.2: General purpose architecture and its datapath. 105
Figure 5.3: An Example for the GUSTO's trimming feature. 105
Figure 5.4: An Example for the GUSTO's trimming feature. 106
Figure 5.5: Performing error analysis using GUSTO. 108
Figure 5.6: An error analysis example, mean error, provided by GUSTO for QR decomposition based 4×4 matrix inversion. The user can select the required number of bit widths for the application where the increasing number of bits results in high accuracy. . . 109
Figure 5.7: Total number of operations for decomposition methods in log domain. 112
Figure 5.8: The comparison between different decomposition methods using sequential execution. 112
Figure 5.9: The comparison between different decomposition methods using parallel execution. 113
Figure 5.10: The comparison of the general purpose processing element and application specific processing element architectures in terms of slices (area) and throughput (performance). 114

Figure 5.11: Area and throughput tradeoffs for different bit width of data: 19, 26 and 32 bits. A user can determine the bitwidth of the data by the error analysis part of GUSTO. High precision can be obtained by using more number of bits as bitwidth which comes at the price of larger area and lower throughput. 115
Figure 5.12: Adaptive Weight Calculation (AWC) using QRD-RLS method consists of two different parts to calculate the weights, QR decomposition and back-substitution. 117
Figure 5.13: Three different designs, *Implementation A, B, and C*, with varying levels of parallelism (using cofactor calculation cores in parallel) to form cofactor matrices. 119
Figure 5.14: The total number of operations for both the QR decomposition based matrix inversion and the analytic method in log domain. 120
Figure 5.15: The inflection point determination between the QR decomposition based matrix inversion and the analytic method using sequential execution. 120
Figure 5.16: The inflection point determination between QR decomposition based matrix inversion and analytic method using parallel execution. 121
Figure 5.17: The total number of operations for different decomposition based matrix inversion methods in log domain. 122
Figure 5.18: The comparison between different decomposition based matrix inversion methods using sequential execution. 123
Figure 5.19: The comparison between different decomposition based matrix inversion methods using parallel execution. 123
Figure 5.20: Design space exploration for QR decomposition based matrix inversion architectures using different resource allocation options.124
Figure 5.21: Design space exploration for decomposition based matrix inversion architectures using different bit widths. 125
Figure 5.22: Design space exploration for decomposition based matrix inversion architectures using different matrix sizes. 125

Figure 6.1: Design Flow of GUSTO. 135
Figure 6.2: (a) The instruction scheduler generates scheduled instructions i.e., assigning operations to the functional resources, performing scheduling and binding. (b) Each functional resource receives scheduled instructions and waits for the required operands to begin execution. 136
Figure 6.3: Detailed Architecture for the Instruction Scheduler. 137

Figure 6.4: (a) Increasing the number of functional resources results in an increase in the area (Slices) and the critical path where we assume that the memory has 20 entries. (b) Increasing the size of the memory entries from 20 to 100 results in an increase in the area (Slices) and the critical path where we assume that there are 8 functional resources. 139
Figure 6.5: A comparison between unoptimized (Unopt.) and optimized (Opt.) instruction scheduler architectures in terms of area (Slices) and critical path (ns) with the increasing number of functional resources. 140
Figure 6.6: Detailed Architecture for a Functional Resource. 141
Figure 6.7: (a) shows that controllers in a functional resource consumes most of the silicon (61%-78%) in order to be able to track the operands. (b) presents the optimization results for functional resources using trimming optimizations. 142
Figure 6.8: Detailed Architecture for the Memory Controller. 144
Figure 6.9: Shows the area results for the memory controller and required clock cycles to execute the given matrix computation algorithm using different number of functional resources. 145
Figure 6.10: (a) Shows the percentage distribution of arithmetic resources and controllers for a single core design. 146
Figure 6.11: There are several different ways that one can partition the given algorithm into different cores. We show three different possibilities: (a), (b), (c), that are examples of designing one core, two cores and three cores for a given matrix computation algorithm respectively. 148
Figure 6.12: (a) GUSTO the optimizes general purpose processing elements to generate application specific processing cores that use a shared memory. (b) GUSTO partitions the data into individual processing cores. The data is classified as local variables, that used only for that core, and shared variables that are written only once and used by the next processing core. 150

Figure 7.1: Design space exploration using different resource allocation options for matrix inversion using QR decomposition. 156
Figure 7.2: Presents hierarchical datapath design for multi-core architecture using different number/type of application specific processing elements which execute a specific part of the given algorithm. . 158
Figure 7.3: Flow of GUSTO showing various parameterization options and output results for multi-core architecture design. 159
Figure 7.4: GUSTO analyzes the instructions that are generated in the instruction generation step, and creates a file to be read via graphviz. 160

Figure 7.5: There are several different ways that one can partition the given algorithm into different cores. We show three different possibilities: (a), (b), (c), that are examples of designing one core, two core and three core for a given matrix computation algorithm respectively. 161
Figure 7.6: The general purpose processing element which uses dynamic scheduling and dynamic memory assignments. This architecture has full connectivity between functional units and controllers. 162
Figure 7.7: The application specific processing element which uses static scheduling and static memory assignments and has only the required connectivity between functional units and controllers for specific algorithm/s. 163
Figure 7.8: A 4×4 matrix multiplication example is shown in (a). Example calculations for the resulting matrix C entries, C_{11} and C_{44} are also shown in (b) and (c). (d) shows a fully parallel architecture for the matrix multiplication. 165
Figure 7.9: Implementations 1-3 are the application specific architectures that are generated by GUSTO with different number of functional units. 167
Figure 7.10: Hardware implementation of matrix multiplication architectures with one PE for entire computation using GUSTO. Implementation 1-3 employs different number of resources for the computation of the matrix multiplication algorithm. 167
Figure 7.11: Implementations 4-9 are the application specific architectures that are generated by GUSTO with different number of PEs. . 168
Figure 7.12: Hardware implementation of matrix multiplication architectures with different design methods, implementation 1-9, using GUSTO. 171
Figure 7.13: Implementations 10-12 are application specific heterogeneous architectures that are generated by GUSTO with different types of PEs. 171
Figure 7.14: Hardware implementation of matrix multiplication architectures with different design methods, implementation 1-12, using GUSTO. 173
Figure 7.15: We present some of the important points of matrix multiplication hardware implementation results: 1) finding the optimum hardware to minimize the hardware while maximizing the throughput, 2) improvement in both area and throughput with hierarchical datapath compared to single core design. 174
Figure 7.16: An analysis to see the effects in area, required clock cycles to execute and the throughput of the architectures by generation of memory controllers with the increasing complexity: A_1, A_2, A_4, A_8 and A_{16} type of processing elements for 4×4 matrix multiplication. 177

Figure 7.17: Hardware implementation of 4×4 matrix multiplication core with 18 bits of precision using System Generator for DSP design tool from Xilinx. (a), (b), (c) and (d) are the address generation logic, input data memory, multiply-accumulate units and destination data memory respectively. 178
Figure 7.18: The data dependencies and the data flow of QR decomposition algorithm. 180
Figure 7.19: Heterogeneous matrix inversion architecture for matrix inversion using QR decomposition. 181
Figure 7.20: Heterogeneous matrix inversion architecture for matrix inversion using LU decomposition. 181
Figure 7.21: Heterogeneous matrix inversion architecture for matrix inversion using Cholesky decomposition. 182
Figure 7.22: Total number of operations in log domain for decomposition based matrix inversion (light) and decompositions only (dark). Note that the dark bars overlap the light bars. 183
Figure 7.23: Hardware implementation of matrix inversion architectures with different design methods (using QR, LU and Cholesky decompositions) using GUSTO. 184
Figure 7.24: GPU architectures employs large number of ALUs by removing the scheduling logic to exploit instruction level parallelism and caches that removes memory latency. Therefore GPUs are simply very powerful number crunching machines. Thus, the future's high performance parallel computation platform should have an ALU dominant architecture to employ more resources for computation by removing as much control logic as possible. We show that GUSTO generated architectures for matrix decomposition, multiplication and inversion are all ALU oriented indeed by consuming 65% - 87% of their silicon into ALUs. . . 187

Figure 8.1: Increasing the number of assets in a portfolio significantly improves the efficient frontier, the efficient allocations of different assets for different risks. Adding new assets to a portfolio shifts the frontier to the upper left which gives better return opportunities with less risk compared to the lower number of assets portfolios. 192
Figure 8.2: The required steps for optimal asset allocation are shown in (a), (b) and (c). After the required inputs to the mean variance are generated in (a), computation of the efficient frontier and determination of the highest utility portfolio are shown in (b) and (c) respectively. This figure also presents the inputs and outputs provided to the user. 194

Figure 8.3: The procedure to generate required inputs is described. The numbers 1-5 refers to these computation steps which are explained in subsections in more detail. 195
Figure 8.4: To determine the computation time of different variables, we compare number of securities, N_s, versus number of portfolios, N_p, and number of portfolios, N_p, versus number of scenarios, N_m, respectively. By looking at the slopes of these lines in the figures it can be easily seen that N_s dominates computation time (has a steeper slope) over N_p (a), N_p dominates computation time over N_m (b). 202
Figure 8.5: Identification of the bottlenecks in the computation of the optimal asset allocation. We run two different test while holding all but one variable constant. We determined that generation of the required input does not consume significant amount of time. On the other hand, step 1 and 2 of the mean variance framework consumes significant amount of time. 204
Figure 8.6: Parameterizable serial hardware architecture for the generation of the required inputs - phase 5. 206
Figure 8.7: Parameterizable fully parallel hardware architecture for the generation of the required inputs - phase 5. As can be seen from the parallel architecture, phase 5 has very high potential for the parallel implementation, therefore a good candidate for decreasing the computational time of the optimal asset allocation. 207
Figure 8.8: Parallel optimum allocation calculator IP Cores. 208
Figure 8.9: Parallel parameterizable hardware architecture for Satisfaction Function Calculator IP Core to be used in the mean variance framework step 2. 209
Figure 8.10: Parallel parameterizable hardware architecture for the mean variance framework step 2. The Monte-Carlo block, Utility Calculation Block, and Satisfaction Function Calculator IP core can be easily parallelized a maximum of N_m, N_m and N_p times respectively. 210
Figure 8.11: Possible speed-ups for "generation of the required inputs - phase 5" 212
Figure 8.12: Possible speed-ups for "Mean Variance Framework Step 2". . . 213

Figure A.1: Visualizing orthonormalization method. 222
Figure A.2: Visualizing reflection method. 223

LIST OF TABLES

Table 1.1: Comparisons between our results and previously published papers. NR denotes not reported. 7

Table 2.1: Specifications of the NVIDIA Tesla S1070 Computing System [105]. 17
Table 2.2: Specifications of the SR and SRD processors. 22
Table 2.3: Specifications of the Xilinx *Virtex-4 XC4VSX35* device. 28
Table 2.4: Breaking down the run-times of GPU and CELL BE in terms of computation and memory accesses [102] 31
Table 2.5: Summary of target platforms [87] 32
Table 2.6: Comparison of random number generation algorithms while implementing on different platforms (top of the Table) and Comparison of different platforms (bottom of the Table) [87] 32
Table 2.7: Comparison of explicit finite difference option pricing implementation using three different platforms: FPGA, GPU and CPU [113]. 34
Table 2.8: Comparison of different platforms, CPU, FPGA and GPU for implementation of polygon based lithography imaging simulations and 2D-FFT based image convolution [114] 34
Table 2.9: A comparison between FPGAs, GPUs and MPPAs. 37

Table 3.1: Our Specifications for Filter Design HDL Coder Toolbox Generated Filters. 66

Table 5.1: Comparisons between our results and previously published papers. NR denotes not reported. 118
Table 5.2: Comparisons between our results and previously published articles for Matrix Inversion using Analytic Method. NR denotes not reported. 127
Table 5.3: Comparisons between our results and previously published articles for Decomposition based Matrix Inversion Architectures. NR denotes not reported. 127

Table 7.1: Comparisons between our results with the architectures employing heterogeneous cores using hierarchical datapaths and previously published articles for Decomposition based Matrix Inversion Architectures. NR denotes not reported. 188
Table 7.2: Comparisons between our results and previously published articles for Matrix Multiplication Architectures. NR denotes not reported. 189

Table 8.1: Different Investor Objectives: Specific and Generalized Forms . . 197
Table 8.2: Different Utility Functions for Satisfaction Indices 200
Table 8.3: Control Inputs for Different Investor Objectives 205

ACKNOWLEDGEMENTS

It is a pleasure to thank everybody who made this thesis possible.

It is difficult to overstate my gratitude to my advisor, **world famous Professor Ryan Kastner**. I couldn't ask for a better advisor; he is hands down the best. I would like to thank Arash Arfaee, **my Main man (mMm)**, for helping me get through the difficult times, and for all the emotional support, comradery, entertainment, and caring he provided.

I also thank to my committee members Professor Timothy Sherwood, Professor Steven Swanson, Professor Bhaskar Rao, Professor Dean Tullsen and Doctor Jung Uk Cho for guiding me through the writing of this thesis, and for all the corrections and revisions made to text that is about to be read. I would like to thank my mentors, Bora Turan, Gökhan Sarpkaya and Byeong K. Lee for helping me to find not only a sense of direction, but of purpose too.

I am indebted to my many student colleagues for providing a stimulating and fun environment in Santa Barbara and San Diego which to learn and grow. I am especially grateful to Nikolay Pavlovich Laptev, Fırat Kart, Onur Güzey, Onur Sakarya, Bridget Benson, Ying Li, Deborah Goshorn, Shahnam Mirzaei, Pouria Bastani, Nick Callegari, Mustafa Arslan, Arda Atalı, Aydın Buluç, Ahmet Bulut, Burak Över, Önder Güven and Erkan Kiriş, my great housemate Shibin Parameswaran, brother Shijin Parameswaran and Shikha Mishra.

I am very thankful for the many friends I have had at Bull & Bear: Erin, James, Jamie, Charles, Ethan, John, Dave; Porters Pub: Stefan, Chris, Kevin, David, Martin; Jack's and West. They definitely made my life easier in graduate school (I might be able graduate faster if I haven't met them though).

I wish to thank my brothers Cihan Akman and Doğan Akman, my cousin Serkan İrtürk and my best friends Mustafa Ilıkkan, Özen Deniz and Serhat Sarışın for being with me all these years.

Lastly, and most importantly, I wish to thank my family, Jale Sayıl, Ömer Sayıl, Bülent İrtürk, Topsi and Turkish Navy. They bore me, raised me, supported me, taught me, and loved me.

I dedicate this thesis to my family, Jale and Ömer Sayıl, for teaching me

everything I know about life. They have always supported my dreams and aspirations. I'd like to thank them for all they are, and all they have done for me.

The text of Chapter 3.5.2 is in part a reprint of the material as it appears in the proceedings of the Design Automation Conference. The dissertation author was a co-primary researcher and author (with Arash Arfaee) and the other co-authors listed on this publication [179] directed and supervised the research which forms the basis for Chapter 3.5.2.

The text of Chapter 5 is in part a reprint of the material as it appears in the proceedings of the Transactions on Embedded Computing Systems. The dissertation author was the primary researcher and author and the co-authors listed on this publication [52–54] directed and supervised the research which forms the basis for Chapter 5.

The text of Chapter 7 is in part a reprint of the material as it appears in the proceedings of the International Conference on Wireless Communications and Networking. The dissertation author was the primary researcher and author and the co-authors listed on this publication [55] directed and supervised the research which forms the basis for Chapter 7.

The text of Chapter 8 is in part a reprint of the material as it appears in the proceedings of the Workshop on High Performance Computational Finance. The dissertation author was the primary researcher and author and the co-authors listed on this publication [180] directed and supervised the research which forms the basis for Chapter 8.

VITA

2004	B. S. in Electrical and Electronics Engineering *cum laude*, Turkish Naval Academy, Istanbul
2004	B. A. in Naval Profession, Turkish Naval Academy, Istanbul
2007	M. S. in Electrical and Computer Engineering, University of California, Santa Barbara
2007	M. A. in Economics, University of California, Santa Barbara
2009	Ph. D. in Computer Science and Engineering, University of California, San Diego

PUBLICATIONS

Ali Irturk, Jason Oberg, Jeffrey Su and Ryan Kastner, "Simulate and Eliminate: A Methodology to Design Application Specific Multi-Core Architectures for Matrix Computations", under review, *16th International Symposium on High-Performance Computer Architecture (HPCA 16)*.

Bridget Benson, Ying Li, Ali Irturk, Junguk Cho, Ryan Kastner, Xing Zhang, "Towards a Low-energy, Reconfigurable Software Defined Acoustic Modem", under review, ACM Transactions on Embedded Computing Systems.

Shahnam Mirzaei, Yan Meng, Arash Arfaee, Ali Irturk, Timothy Sherwood, Ryan Kastner, "An Optimized Algorithm for Leakage Power Reduction of Embedded Memories on FPGAs Through Location Assignments", under review, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

Arash Arfaee, Ali Irturk, Nikolay Laptev, Ryan Kastner, Farzan Fallah, "Xquasher: A Tool for Efficient Computation of Multiple Linear Expressions", *In Proceedings of the Design Automation Conference (DAC 2009), July 2009*.

Ali Irturk, Bridget Benson, Shahnam Mirzaei and Ryan Kastner, "GUSTO: An Automatic Generation and Optimization Tool for Matrix Inversion Architectures", *In Proceedings of the ACM Transactions on Embedded Computing Systems*.

Bridget Benson, Ali Irturk, Junguk Cho, Ryan Kastner, "Energy Benefits of Reconfigurable Hardware for use in Underwater Sensor Nets", *In Proceedings of the 16th Reconfigurable Architectures Workshop (RAW 2009)*, May 2009.

Ali Irturk, Bridget Benson, Nikolay Laptev and Ryan Kastner, "Architectural Optimization of Decomposition Algorithms for Wireless Communication Systems", *In Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC 2009)*, April 2009.

Ali Irturk, Bridget Benson, Nikolay Laptev and Ryan Kastner, "FPGA Acceleration of Mean Variance Framework for Optimum Asset Allocation", *In Proceedings of the Workshop on High Performance Computational Finance at SC08 International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2008.

Ali Irturk, Bridget Benson and Ryan Kastner, "Automatic Generation of Decomposition based Matrix Inversion Architectures", *In Proceedings of the IEEE International Conference on Field-Programmable Technology (ICFPT)*, December 2008.

Bridget Benson, Ali Irturk, Junguk Cho, and Ryan Kastner, "Survey of Hardware Platforms for an Energy Efficient Implementation of Matching Pursuits Algorithm for Shallow Water Networks", *In Proceedings of the Third ACM International Workshop on UnderWater Networks (WUWNet), in conjunction with ACM MobiCom 2008*, September 2008.

Shahnam Mirzaei, Ali Irturk, Ryan Kastner, Brad T. Weals and Richard E. Cagley, "Design Space Exploration of a Cooperative MIMO Receiver for Reconfigurable Architectures", *In Proceedings of the IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, July 2008.

Ali Irturk, Bridget Benson, Shahnam Mirzaei and Ryan Kastner, "An FPGA Design Space Exploration Tool for Matrix Inversion Architectures", *In Proceedings of the IEEE Symposium on Application Specific Processors (SASP)*, June 2008.

Ali Irturk, Bridget Benson, and Ryan Kastner, "An Optimization Methodology for Matrix Computation Architectures", *UCSD Technical Report, CS2009-0936*, 2009.

Ali Irturk, Shahnam Mirzaei and Ryan Kastner, "FPGA Implementation of Adaptive Weight Calculation Core Using QRD-RLS Algorithm", *UCSD Technical Report, CS2009-0937*, 2009.

Ali Irturk, Shahnam Mirzaei and Ryan Kastner, "An Efficient FPGA Implementation of Scalable Matrix Inversion Core using QR Decomposition", *UCSD Technical Report, CS2009-0938*, 2009.

Ali Irturk, "Implementation of QR Decomposition Algorithms using FPGAs", *M.S. Thesis*, Department of Electrical and Computer Engineering, University of California, Santa Barbara, June 2007. Advisor: Ryan Kastner.

ABSTRACT OF THE DISSERTATION

GUSTO: General architecture design Utility and Synthesis Tool for Optimization

by

Ali Umut İrtürk
Doctor of Philosophy in Computer Science
University of California San Diego, 2009
Ryan Kastner, Chair

Matrix computations lie at the heart of many scientific computational algorithms including signal processing, computer vision and financial computations. Since matrix computation algorithms are expensive computational tasks, hardware implementations of these algorithms requires substantial time and effort. There is an increasing demand for a domain specific tool for matrix computation algorithms which provides fast and highly efficient hardware production.

This thesis presents GUSTO, a novel hardware design tool that provides a push-button transition from high level specification for matrix computation algorithms to hardware description language. GUSTO employs a novel top-to-bottom design methodology to generate correct-by-construction and cycle-accurate application specific architectures. The top-to-bottom design methodology provides simplicity (through the use of a simple tool chain and programming model), flexibility (through the use of different languages, e.g. C/MATLAB, as a high level specification and different parameterization options), scalability (through the ability to handle complex algorithms) and performance (through the use of our novel trimming optimization using a simulate & eliminate method providing results that are similar to these in commercial tools).

Although matrix computations are inherently parallel, the algorithms and commercial software tools to exploit parallel processing are still in their infancy.

Therefore, GUSTO also provides the ability to divide the given matrix computation algorithms into smaller processing elements providing architectures that are small in area and highly optimized for throughput. These processing elements are then instantiated with hierarchical datapaths in a multi-core fashion.

The different design methods and parameterization options that are provided by GUSTO enable the user to study area and performance tradeoffs over a large number of different architectures and find the optimum architecture for the desired objective. GUSTO provides the ability to prototype hardware systems in minutes rather than days or weeks.

Chapter 1

Introduction

1.1 Motivation

Matrix computations lie at the heart of many scientific computational tasks. For example, wireless communication systems use matrix inversion in equalization algorithms to remove the effect of the channel on the signal [4–6], mean variance framework in financial computation uses matrix inversion to solve a constrained maximization problem to provide optimum asset allocations for an investor [21], and the optimal flow computation algorithm in computer vision uses matrix inversion for motion estimation [22]. There is an increasing demand for a domain specific tool for matrix computation applications which provides fast and highly efficient hardware production.

This thesis presents **GUSTO** (**G**eneral architecture design **U**tility and **S**ynthesis **T**ool for **O**ptimization), a novel hardware design tool that provides a **push-button transition** from high level specification for matrix computation algorithms to hardware description language. GUSTO employs a novel **top-to-bottom** design methodology to generate **correct-by-construction** and **cycle-accurate** application specific architectures. The top-to-bottom design methodology provides **simplicity** (through the use of a simple tool chain and programming model), **flexibility** (through the use of different languages, e.g. C/MATLAB, as a high level specification and different parameterization options), **scalability** (through the ability to handle complex algorithms) and **performance** (through

the use of our novel trimming optimization using a simulate & eliminate method providing results that are similar to these in commercial tools). GUSTO also provides the ability to divide the given matrix computation algorithms into smaller processing elements results in architectures that are small in area and highly optimized for throughput, then instantiates these PEs with hierarchical datapaths in a multicore fashion. The different design methods and parameterization options that are provided by GUSTO enable the user to study area and performance tradeoffs over a large number of different architectures and find the optimum architecture for the desired objective. GUSTO provides the ability to prototype hardware systems in just minutes instead of days or weeks with these capabilities.

The anticipated benefits of this dissertation are:

1) Rapid development of single-core FPGA elements: *GUSTO is a design tool which allows rapid development of complex matrix computation algorithms with different parameterization options. GUSTO is useful for a wide variety of designs, providing higher performance computing and faster time to market;*

2) Hierarchy Datapath Implementation for multi-core FPGA elements: *GUSTO is capable of dividing the given algorithms into small highly parallelizable PEs, generate hardware and combine these small PEs with hierarchical datapaths in a multi-core architecture fashion. Resulting multi-core architecture solutions are expected to be smaller, cheaper, and lower power than can be achieved using existing EDA (Electronic Design Automation) tools;*

3) An FPGA Engine for MATLAB: *MATLAB programs can have excellent performance for matrix-heavy computations. Therefore, many scientific computational algorithms such as signal processing, computer vision and financial computations are typically written and tested using MATLAB. GUSTO lets software engineers to implement hardware out of MATLAB code without knowledge in hardware design;*

4) Domain specific: *MATLAB is the de facto standard language for many matrix computation algorithms, and there are a number of tools that translate such algorithms to a hardware description language; however, the majority of these tools*

attempt to be everything to everyone, and often fail to do anything for anyone. GUSTO takes a more focused approach, specifically targeting matrix computation algorithms;

5) Built-in Libraries: *GUSTO provides a path to built-in libraries, including previously implemented matrix computation algorithms, to be used while designing larger applications;*

6) End Platform Independency: *GUSTO can target different platforms such as GPUs (Graphics Processing Units) and CMPs (chip multiprocessors) with the appropriate changes to the back end of the tool.*

1.2 Research Overview

GUSTO receives the algorithm from the user and allows him/her to select the type and number of arithmetic resources, the data representation (integer and fractional bit width), and automatically generates optimized application specific PEs (Figure 1.1). Application specific architectures that are generated by GUSTO employ the optimal number of resources which maximizes the throughput while minimizing area. GUSTO also incorporates hierarchical datapaths and heterogeneous architecture generation options. By using these features, a user can divide the given algorithms into small highly parallelizable parts, generate hardware using GUSTO and combine these small PEs with hierarchical datapaths to perform multi-core processing.

In the architecture generation step, GUSTO creates a general purpose PE which exploits instruction level parallelism. GUSTO then simulates the general purpose PE to collect scheduling information and perform resource trimming to create an optimized application specific PE while ensuring the correctness of the solution is maintained. These optimizations are divided into two sections:

1) Static architecture generation: GUSTO generates a general purpose PE and its datapath by using resource constrained list scheduling after the required inputs are given. Simulating this architecture helps GUSTO to reveal the assign-

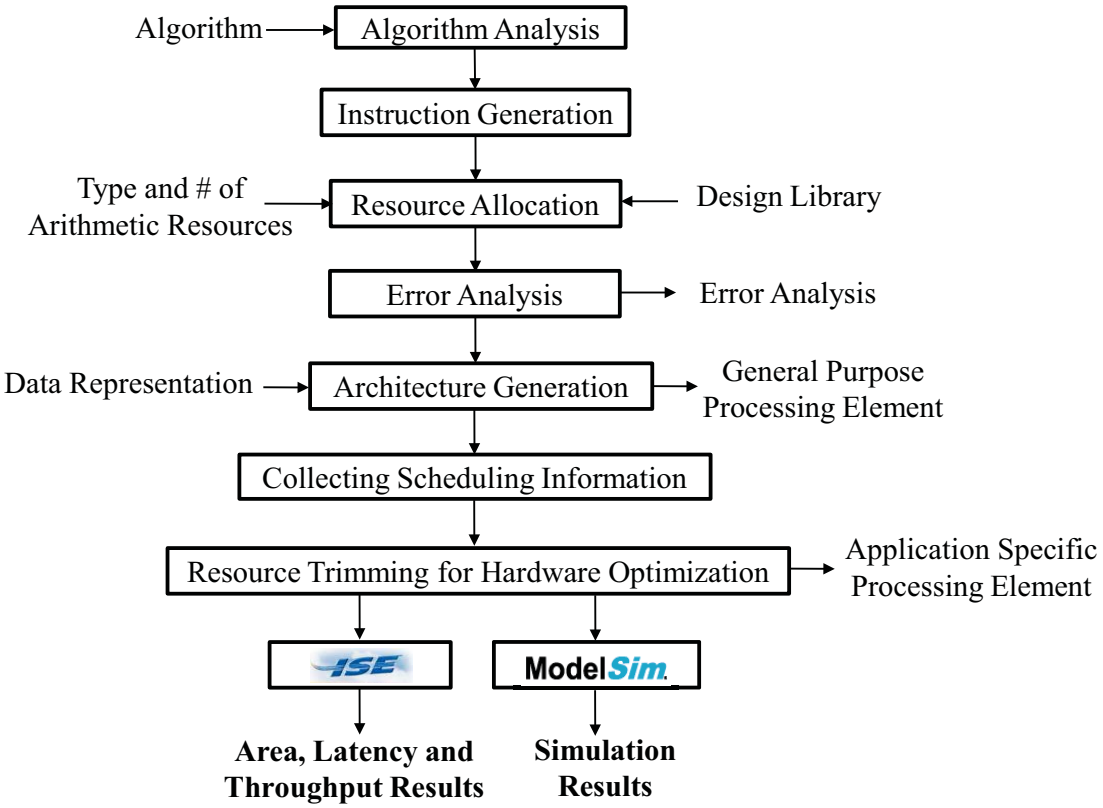


Figure 1.1: **Design Flow of GUSTO.**

ments done to the arithmetic units and the memory elements during the scheduling process. Gathering this information and using it to cancel the scheduling process and dynamic memory assignments results in an optimized architecture with significant area and timing savings.

2) Trimming for optimization: GUSTO performs trimming of the unused resources from the general purpose PE while ensuring that correctness of the solution is maintained. GUSTO simulates the architecture to define the usage of arithmetic units, multiplexers, register entries and input/output ports and trims away the unused components with their interconnects. A trimming example is shown in Figure 1.2(a,b,c and d).

GUSTO provides different design methods and parameterization options which enables the user to study area and performance tradeoffs over a large num-

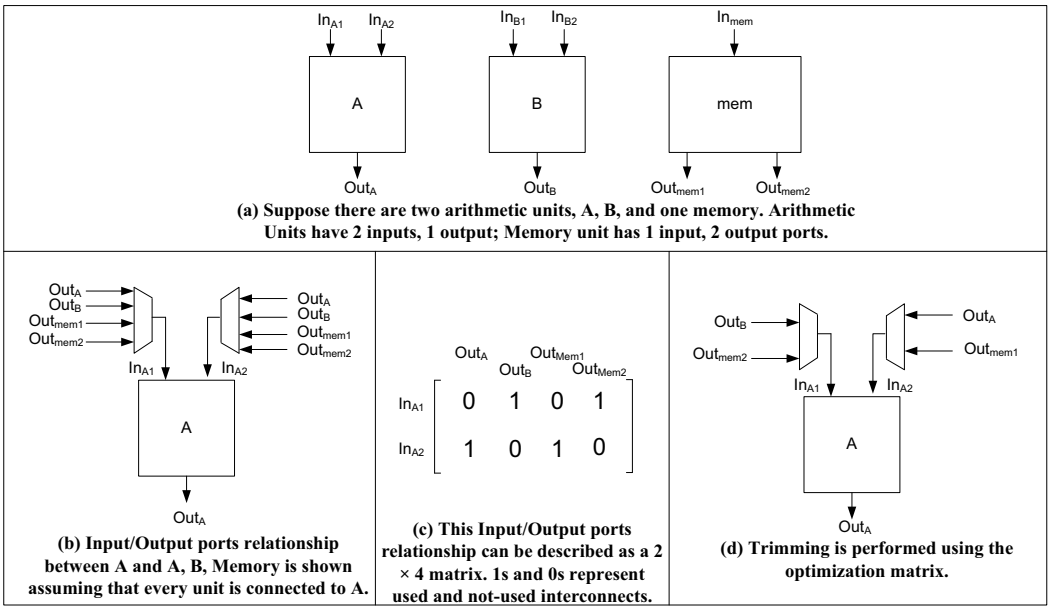


Figure 1.2: **Flow of GUSTO's trimming feature.**

ber of different architectures and pick the most efficient one in terms of the desired objective. Figure 1.3 shows the tradeoff between computational throughput and area for various matrix multiplication architectures. There are three different design methods:

1) Using one PE for entire matrix multiplication: Implementations 1-3 are the outputs of GUSTO with different number of functional units;

2) Designing a homogeneous architecture by dividing the given computation into identical PEs: Implementations 4-9 are the outputs of GUSTO with different number of PEs;

3) Designing a heterogeneous achitecture with different types of PEs using hierarchical datapaths: Implementations 10-12 are heterogeneous architectures that are the outputs of GUSTO with different types of PEs using hierarchical datapaths.

The ability to divide the given algorithm into smaller processing elements results in architectures that are small in area and highly optimized for throughput. These different design methods and parameterization options enables the user to study area and performance tradeoffs over a large number of different architec-

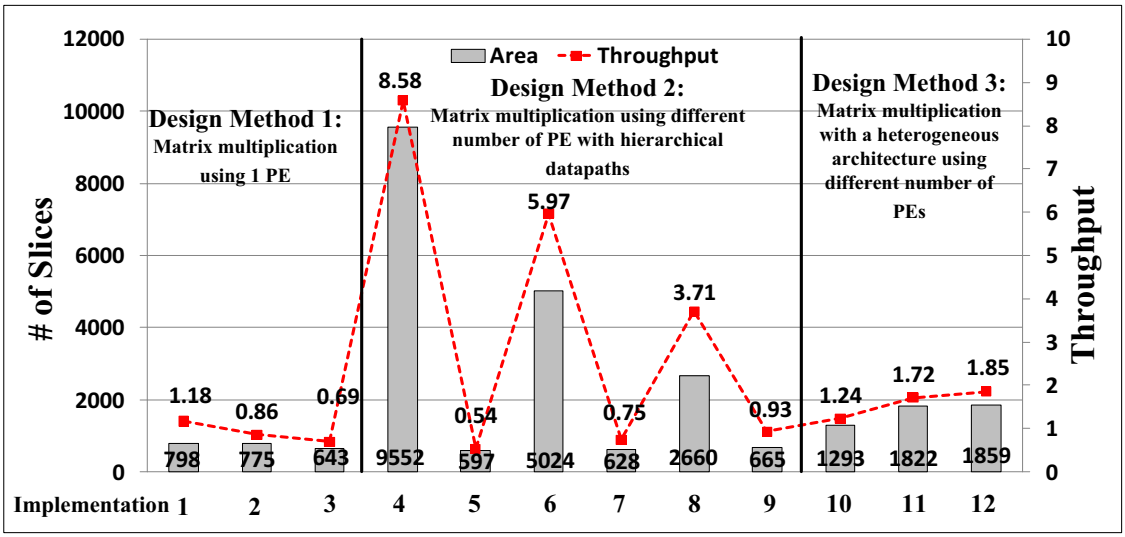


Figure 1.3: **Hardware implementation of matrix multiplication architectures with different design methods using GUSTO.**

tures. This will result in more detailed design space exploration and more efficient hardware implementations that enable us to exploit both instruction and task level parallelism.

The efficiency of hardware implementation using GUSTO is validated with various matrix computation algorithms and applications that mainly employs matrix computations. For example, we compare our results for QR decomposition based matrix inversion algorithms to other published work in Table 1.1. It can be seen that even though the GUSTO algorithms are defined in a convenient high level manner suitable to rapid design, the resulting hardware is as fast, efficient, and compact as state of the art hardware designed without the benefit of the GUSTO high level design tool.

1.3 Organization of Dissertation

This dissertation is organized in the following manner:

CHAPTER 2 introduces existing parallel hardware platforms that are being used for implementation of matrix computation algorithms: Graphic Processing Units (GPUs), Massively Parallel Processor Arrays (MPPAs) and Field Programmable

Table 1.1: Comparisons between our results and previously published papers. NR denotes not reported.

	[163]	**GUSTO**	[160]	**GUSTO**	[161]	**GUSTO**
Matrix Dimensions	3×3	3×3	4×4	4×4	4×4	4×4
Bit width	16	**16**	12	**12**	20	**20**
Data type	fixed	**fixed**	fixed	**fixed**	floating	**fixed**
Device type (Virtex)	IV	**IV**	II	**II**	IV	**IV**
Slices	3076	**1496**	4400	**2214**	9117	**3584**
DSP48s	1	**6**	NR	**8**	22	**12**
BRAMs	NR	**1**	NR	**1**	NR	**1**
Throughput $(10^6 \times s^{-1})$	0.58	**0.32**	0.28	**0.33**	0.12	**0.26**

Gate Arrays (FPGAs), discusses their inherent advantages and disadvantages, presents a literature survey to compare existing parallel platforms in implementation of different applications and provides a roadmap for the multi-core hardware implementation that combines most of the advantages provided by the existing platforms.

CHAPTER 3 introduces existing design tools that are implemented in academia and industry, discusses their inherent advantages and disadvantages and presents a roadmap for a tool specifically targeting matrix computation architectures.

CHAPTER 4 introduces the definitions of matrix comptations and presents different matrix computation algorithms that are employed throughout this dissertation: matrix multiplication, matrix decompositions (QR, LU and Cholesky), decomposition based matrix inversion and analytic method for matrix inversion.

CHAPTER 5 focuses on the basics of GUSTO and its design flow: algorithm analysis, instruction generation, resource allocation, error analysis, architecture generation, optimizations performed namely static architecture generation and trimming for optimization. This chapter also presents GUSTO generated architectures for matrix multiplication, matrix decomposition (QR, LU and Cholesky) and matrix inversion (QR, LU, Cholesky, Analytic), a case study: *Implementation of Adaptive Weight Calculation Core using QRD-RLS Algorithm* and compares these results with the previously published works. This chapter focuses on instruction level

parallelism.

CHAPTER 6 details the general purpose processor architecture that is generated by GUSTO, discusses design decisions and challenges and optimizations performed to generate an optimized application specific processor architecture.

CHAPTER 7 focuses on the problem of scalibity of architectures with the complexity of algorithms and introduces hierarchical datapaths and heterogeneous architecture generation methodology which provides a more detailed design space exploration and more efficient hardware implementations. This chapter focuses on task level parallelism.

CHAPTER 8 presents an application, *Mean Variance Framework* that is being used to determine the best allocation for a given investor in financial markets, its design and analyses using GUSTO.

CHAPTER 9 gives some ideas for future research directions to improve our tool, GUSTO.

Chapter 2

Parallel Platforms for Matrix Computation Algorithms

Matrix computations is a topic of great interest in numerical linear algebra. Since many of these matrix computation algorithms are computationally expensive and memory demanding tasks, it is an area where there is much interest in parallel architectures. There has been an extensive research for new architectures that provide various types of computational cores on a single die. Examples to these platforms are Chip Multiprocessors (CMPs), Graphical Processor Units (GPUs), Massively Parallel Processor Arrays (MPPAs) etc. where they have different types of architectural organizations, processor types, memory management etc. Each of these architectures has their inherent (dis)advantages, yet all rely heavily on the ability to expose vast amounts of parallelism.

Since the application market that uses matrix computations is quickly spreading which reduces the time available for design of individual applications [109], there is an increasing demand for design tools that can accelerate the mapping process, as well as the optimizations of these algorithms, to these particular highly parallel platforms. Examples of these tools include NVIDIA Compute Unified Device Architecture (CUDA) for GPUs, aDesigner for MPPA. Using a high level design tool often comes with its own price, and can be seen as a tradeoff between time/efficiency and quality of results. How to design a powerful tool for a particular architecture is still an important area of research.

If one wants to design and implement a platform that provides large amount of parallelism, the choice of a computing platform plays an important role. Designers need to decide between a hardware or software implementation to exploit inherent parallelism. The hardware implementation consists of designing an Application Specific Integrated Circuit (ASIC), which offers exceptional performance, but long time to market and high costs for all but the largest production chips. The software route tends towards the use of Digital Signal Processors (DSPs) due to the ease of development and fast time to market. However, they lack the performance for high throughput applications. Field Programmable Gate Arrays (FPGAs) are prevalent for computationally intensive applications since they can provide large amount of parallelism. FPGAs play a middle role between ASICs and DSPs, as they have the programmability of software with performance approaching that of a custom hardware implementation. The reconfigurable hardware, as in the case of FPGA, was proven efficient to be allowing optimal distribution of computations [110].

We foresee that reconfigurable architectures, particularly FPGAs, are the best platforms to test and design future extremely parallel multi-core architectures due to their flexible architecture that can morph itself to tackle the precise problem at hand, their ability to handle non-regular memory accesses required by high-level algorithms, low non-recurring engineering costs (NREs) and their potential path to application specific integrated circuits (ASICs).

Therefore, the main goal of this chapter is to learn from (dis)advantages of existing parallel architectures, and propose a roadmap for an architecture that combines most of these advantages provided by the existing platforms. This chapter provides the following information:

- Architecture, (dis)advantages of the platform, design flow/tools, and an example family and/or device with the specifications for Graphic Processing Units (GPUs), Massively Parallel Processor Arrays (MPPAs) and Field Programmable Gate Arrays (FPGAs);

- A literature survey to compare existing parallel platforms in implementation of different computationally intensive applications;

- A roadmap for the multi-core hardware implementation using FPGAs that can combine most of the advantages provided by the existing platforms.

2.1 Graphic Processing Units (GPUs)

This section introduces Graphics Processing Units (GPUs) and advances in GPUs to delve into the general purpose computation using GPUs (GPGPU). GPUs traditionally designed to perform computation of the screen images to be displayed on computer monitors by transforming the input data to pixels on screen in form of geometry which is known as *rendering*. Since these devices need to render a large number of screen pixels, $\sim$ 30 times per second, they have employed huge amount of processing power. Therefore they became attractive for high performace computation.

The rendering process, computation of screen pixels from input data, in the traditional GPUs performs a classical rendering pipeline using OpenGL (OpenGL is an Application Programming Interface (API) for GPU programming) which includes several stages: Vertex, Primitive Assembly, Rasterization, Fragment and Buffer Operations. We introduce these general steps briefly and more detailed explanation can be found in [102].

Vertex stage computes all the values that belongs to a vertex, node in a geometry. The transformation of vertices to view coordinates and attributes (color, texture etc.) is also processed in this stage.

Primitive assembly stage assembles all the vertices and creates primitives, triangles etc.

Rasterization stage performs sampling on the scene at regular intervals to create fragments, meta-pixels which are not written to screen yet.

Fragment stage adds textures and performs other calculations like per-fragment to provide more realistic rendering.

Buffer operations stage perfoms tests on the fragment to determine if it should be written to the framebuffer, discarded or blended.

The programmable stages of these traditional GPUs are *Vertex processor* in

the Vertex stage and *Fragment processor* in the Fragment stage. Fragment processor is the most powerful and more in quantity compared to the Vertex processors since most of the operations are executed in this type of processors. For example, ATI Radeon X1900 XTX includes 8 vertex and 48 fragment processors [103].

Even though GPUs are such a powerful computational devices, they don't provide the programming flexibility. There exists several different interfaces to the GPU to abstract away the obscure details that a user needs to program the GPU [95,96]. These interfaces still require the user to have some knowledge about the GPU programming and GPU hardware for efficient programming. The traditional interfaces, graphics APIs: *OpenGL* [97] and *DirectX* [98], provide an ease for programming while rendering graphics. However it is hard to program advanced rendering techniques and non-graphical algorithms since these APIs only include some certain functions and it is inefficient and time-consuming to use the embedded functions for non-graphical algorithms. These APIs provide the following advantages to the vendors and users:

- Simplification of the the high-level programming model since users do not need to write codes directly to the metal, know some of the complex details of the GPU architecture and learn the new generation GPU architectures to be able to program;

- Hardware abstraction from the users lets vendors to change the GPU architecture as often as desired.

There also exist some high level interfaces to the GPUs which are a lot easier to program: PeakStream [99], RapidMind [100] and CUDA [101]. These high level interfaces use C/C++ languages and still requires some insight into the GPU programming model to be able to write some special code for parallel processing.

Therefore most of the these traditional GPUs have the following disadvantages:

- Most importantly, computer graphics vendors do not disclose the underlying architecture of GPUs which prevents users to have more control on GPUs.

Therefore considerable amount of time is spent on understanding this underlying architecture to be able to map computationally intensive calculations onto GPU hardware efficiently;

- Most of the GPUs provide single precision floating point which is not IEEE 754 conformant either and result in erronous computations of certain arithmetic operations;

- Traditional interfaces, graphics APIs: *OpenGL* [97] and *DirectX* [98], are hard to program. High level interfaces for the GPUs: PeakStream [99](acquired by Google in 2007), RapidMind [100] and CUDA [101] use only C/C++ languages, require some insight into the GPU programming model and still require the developers to analyze the application to be able to divide the data into smaller chunks to be distributed into different processors [111];

- There exist some missing hardware implementation in GPUs since these devices are designed for graphic processing [102]. For example, for-loops implementation on processors are truncated to be maximum 25 [108];

- GPUs include a little logic for branch prediction, cache control and instruction pipelining since these logics are not required to compute screen images. Branching is very costly on GPUs since if one thread takes a branch, all of other threads, which didn't take the branch, need to wait for that single thread to complete;

- GPUs lack highly efficient caches and use relatively modest cache size (compared to CPUs). Therefore GPUs rely on their fast access to the RAM and access to the RAM more often compared to CPUs;

- GPUs are power-hungry platforms that require special cooling.

The recent advances in GPUs lead to the various changes in the architectural side of GPUs to correct some of these disadvantages and make them more appropriate for general purpose computing. Some of these architectural changes are:

- Combination of the two processors, vertex and fragment, that results in a unified processor architecture: NVIDIA GeForce 8800 GTX has 128 unified processors [104]. These processors are called stream processors (SPs);
- Modification of overall organization of SPs in the latest generation GPUs like Tesla S1070 Computing System. In this new architecture, 8 SPs are combined to form a Stream Multi-processor (SM) where each SM includes a dedicated L1 cache for better data accesses [105];
- GPU vendors also start to provide GPUs with double precision floating point. However employing double precision floating point is still very costly compared to single precision or emulation of double precision floating point [106, 107], and it comes around at the price of half of the computational power compared to single precision. It is also important to note that employing double precision does not neccesarrily mean that each processor will be equipped with a floating point unit. One of the latest GPUs, the NVIDIA Tesla S1070 Computing Systems, includes one double precision floating point unit in each each Stream Multi-processor(SM).

However, GPUs still have the following disadvantages:

- Being closed architecture;
- High cost and inefficient double precision floatint point units;
- Expensive branching;
- Previous analysis requirement to be able to distribute the given application among processors;
- High level programming language dependency to specific tools;
- High power consumption.

We show an GPU architecure in Figure 2.1. GPU architecture employs large number of ALUs, therefore removes scheduling logic needed for exploiting instruction level parallelism and caches that remove memory latency. Therefore

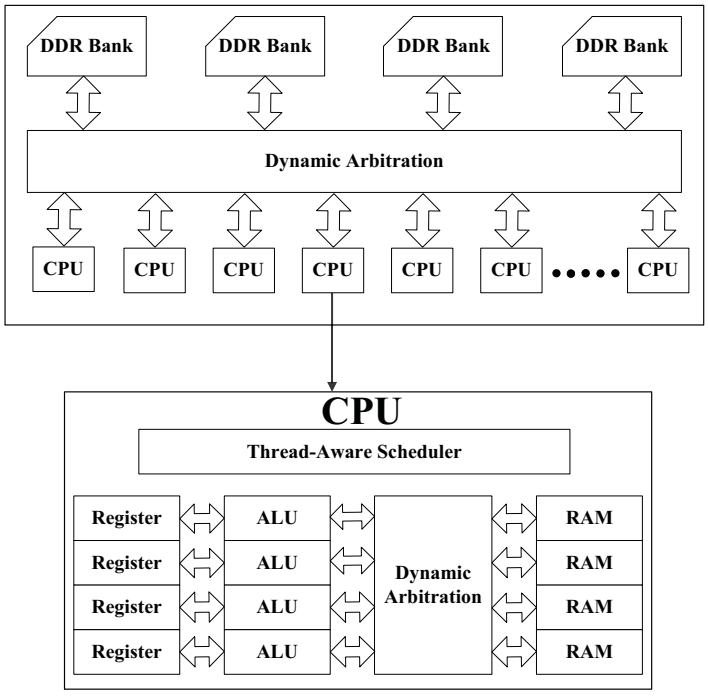


Figure 2.1: **GPU Architecture [87].**

GPUs employ thread level parallellism to decrease latency that is introduced by removing scheduling logic. Each CPU can execute up to 1024 threads at once where threads execute in batches of 32 threads. These 32 threads are called warps and they provide SIMD style parallelism which also provides the ability to independently enable and disable each thread within a wrap. While using GPUs, it is important to ensure that:

- Highest number of threads are active within a wrap to achieve a higher level of parallelism;
- All threads take the same branch of conditional statements;
- The same number of loops are executed in threads.

By far the most popular high level design tool for GPU programmming is *NVIDIA Compute Unified Device Architecture (CUDA), 2006,* that is a low-level API that accesses GPU hardware without going through the graphics API. The design flow using CUDA for NVIDIA GPUs is shown in Figure 2.2 and explained below.

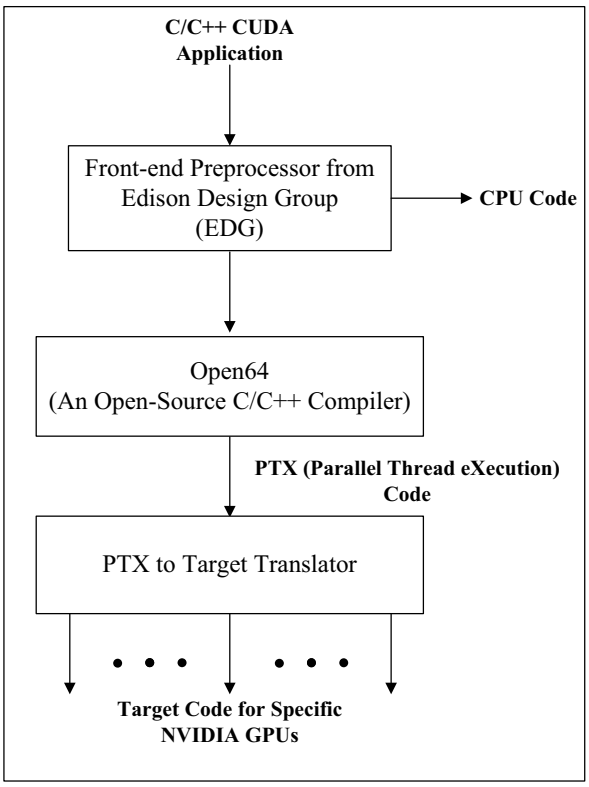


Figure 2.2: **The design flow using CUDA for NVIDIA GPUs.**

A developer needs to analyze the application, algorithm and data, decides on the optimal number of threads and blocks which fully utilize the GPU hardware and expresses the solution in C or C++ using CUDA-speak language (CUDA extensions and API calls). It is important to note that this requires a manual effort such that 5,000 concurrent threads are seen as the optimal distribution for GeForce 8 to keep the GPU occupied [111].

Codes written using CUDA target two different processor architectures: GPU and CPU. **EDG** step, which employs a front-end preprocessor from Edison Design Group, performs the seperation of the souce code for each target architecture and creates different source files for these architectures.

Open64 step, an open-source C/C++ compiler which is originally designed for Intel's Itanium architecture, generates Parallel Thread eXecution (PTX) codes which can be seen as assembly language. PTX codes are device-driver files that

are not specific to any particular NVIDIA GPU.

PTX to Target Translator step translates PTX codes to run on the particular device.

NVIDIA became the largest independent GPU vendor after ATI is acquired by AMD in 2006. We show the specifications of one of the newest generation of GPUs from NVIDIA in Table 2.1. This device is specifically designed for high performance computing. Other product brands from NVIDIA are Quadro series: professional graphics workstations, and GeForce series targeting traditional consumer graphics market.

Table 2.1: Specifications of the NVIDIA Tesla S1070 Computing System [105].

Number of Tesla GPUs	4
Number of Streaming Processor Cores	960 (240 per processor)
Frequency of Processor Cores	1.296 to 1.44 GHz
Single Precision Floating Point Performance (peak)	3.73 to 4.14 TFlops
Double Precision Floating Point Performance (peak)	311 to 345 GFlops
Floating Point Precision	IEEE 754 single & double
Total Dedicated Memory	16GB
Memory Interface	512-bit
Memory Bandwidth	408GB/sec
Max. Power Consumption	800W
System Interface	PCIe $\times$16 or $\times$8
Programming Environment	CUDA

2.2 Massively Parallel Processor Arrays (MPPAs)

Massively Parallel Processor Arrays (MPPAs) employ hundreds of extremely simple in-order-RISC CPUs to introduce massive parallelism. These simple CPUs employ small local memories and are instantiated in a regular grid using 2D communication channels between them. MPPAs are different than *multi-core* or *many-core* conventional processors (CMPs), shown in Figure 2.3, since these conventional

processors operate the same way as single-processor CPUs by employing only a few processors with a shared-memory architecture [87] and large control units devoted to management and scheduling tasks.

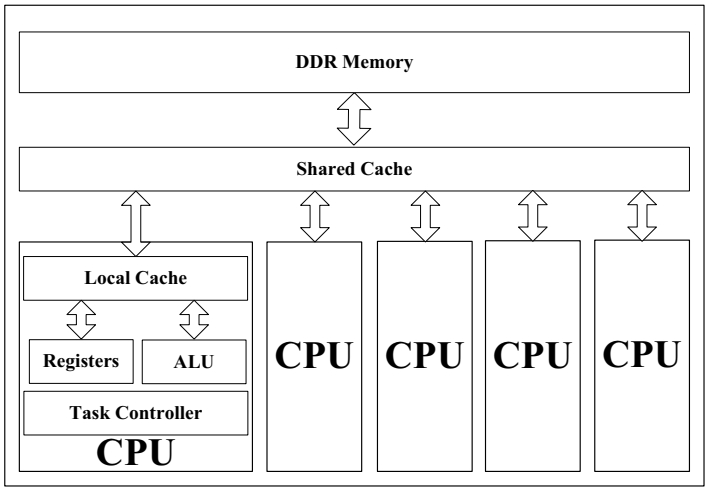


Figure 2.3: **Multi-core CPU architecture.**

There are several different MPPA architectures: *Ambric* from Ambric, Inc., *picoArray* from picoChip and *SEAforth* from IntellaSys. We consider Ambric AM2000 family [88] as an example to these type of architectures and we explore MPPAs in more detail using Ambric architectures since these are the newest and most advanced type of architectures. These extremely simple CPUs, that are employed in MPPAs, provide high efficiency in terms of peak performance per mm^2 and power efficiency. However MPPAs introduce significant problems while

- partitioning the given applications onto hundreds of processors;
- efficiently organizing communications over a network that favours local communication over global communication;
- mapping the two different type of processors that are employed in Ambric chip architectures.

Ambric family MPPAs are programmed using a software development tool suite, *aDesigner*, which uses a Structured Object Programming Model (SOPM). This

integrated development environment (IDE) is based on Eclipse [89] open development platform.

aDesigner enables software engineers to target Ambric MPPAs using a set of objects and structures. Objects are written in a subset of standard Java or assembly code and target CPUs inside the Ambric architecture. Software engineers are also able to load objects from the predefined libraries. Structured objects, combination of objects, is specified with a text-based structural language or using a graphical block diagrams [90–92]. A structural object programming model is shown in Figure 2.4.

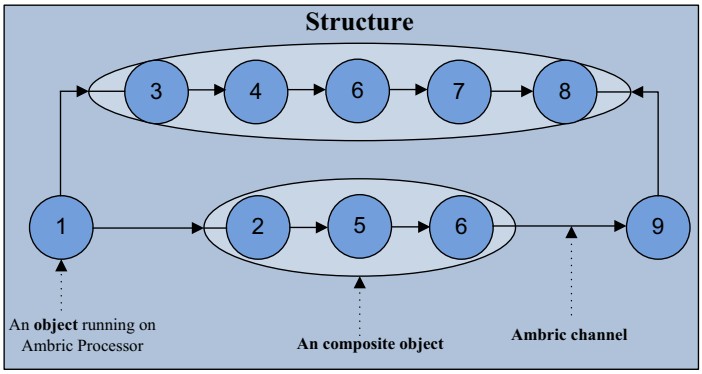


Figure 2.4: **Structural object programming model for Ambric Architecture.**

Defined objects execute independently on their own parallel architecture, no sharing, multi-threading or virtualizing is exploited. Structural objects have no effect on each other and there is no implicitly shared memory between structural objects. Data and control token communication between objects and/or structural objects is accomplished using simple FIFO style parallel structure of hardware channels which are word-wide, unidirectional, point-to-point from each other. These hardware channels are self synchronizing at run time.

Ambric MPPA design flow includes the following steps [93]: Structure, Code, Reuse, Verify, Realize and Test which are shown in Figure 2.5.
Structure step requires the design of the structure of objects for the desired application as well as the required control/data-flow between objects/structures.
In **Code** step, user codes the required structures and objects using Java or assem-

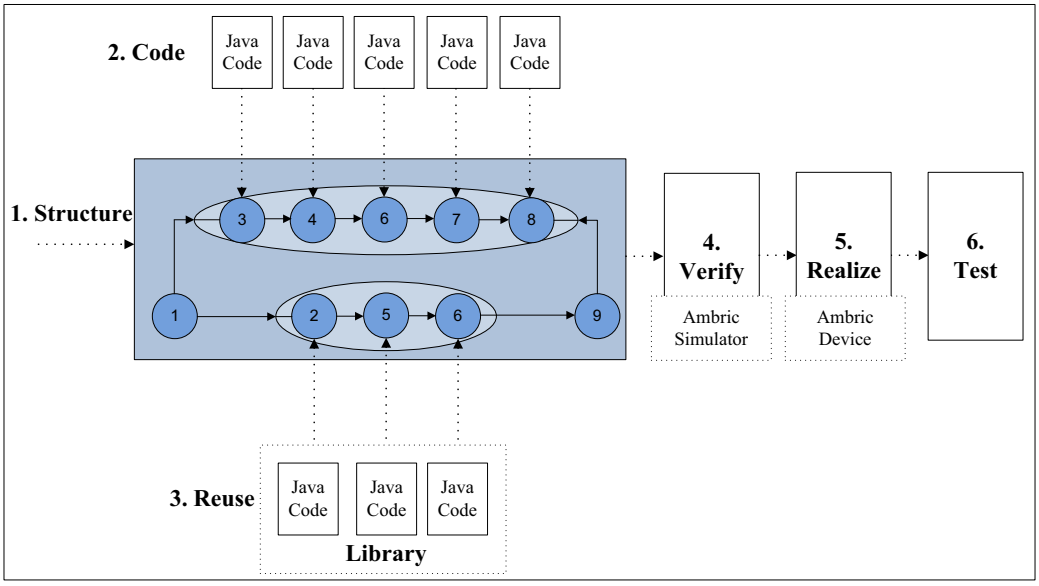


Figure 2.5: **Design Flow of Ambric MPPA and its steps: Structure, Code, Reuse, Verify, Realize and Test.**

bly code.

Reuse step shows that if any of the required structures or objects are predefined, therefore they are in the library, they can be instantiated from the library and no coding is required.

Verify step verifies the implementation using Ambric simulator.

Realize step maps and routes the design to the target Ambric device using mapping & routing tool.

Test step verifies that the designed architecture works properly.

Next subsections provide more insight about Ambric family as well as AM2045 architecture.

2.2.1 Ambric Family Overview

Ambric chip architecture, presented in Figure 2.6, employs Compute Units (CUs) which are a cluster of 4 processors. These processors have two different types: SRD and SR [94].

- *SRD* processor is a 32-bit streaming processor with a directly accessible local

memory. SRD processor is the main processor type in Ambric architecture and includes DSP extensions for math-intensive applications with larger registers and an integer multiply-accumulate unit.

- *SR* processor is another 32-bit streaming processor which is simpler compared to SRD processors. The main purpose of these type processors are managing the generation of complex address streams, controlling the channel traffic and performing other utiliy tasks to sustain high throughput for SRD processors.

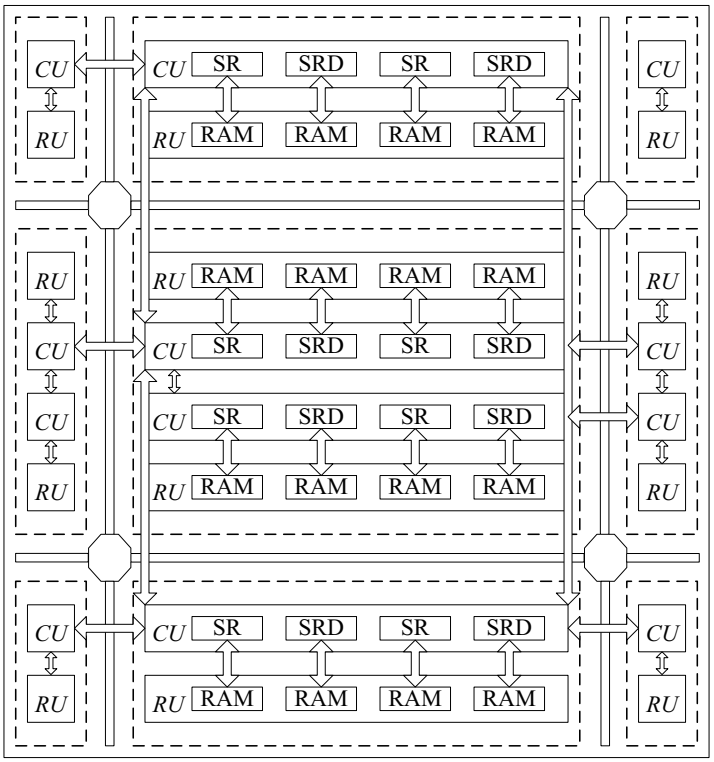


Figure 2.6: **Ambric Architecture.**

Ambric chip architecture also emplys the RAM Units (RUs) which are the main on-chip memories. RUs are responsible for data and address streaming over communication channels. CPUs connect each other through self-synchronizing unidirectional FIFOs where source and destination can operate at different clock rates. CPUs in a *bric*, the combination of CUs and RUs, connect to RAMs through a dynamically arbitrating interconnect where CPUs stream data between each other,

read and write to the RUs and create a chip-wide 2D channel network. Using this kind of interconnect lets CPUs to access external resources such as PCI Express, general purpose I/O and DDR RAMs as well as transfer data to and from CPUs and RAMs to elsewhere in the device.

The combination of 2 CUs and 2 RUs is called a *bric* in Ambric architecture which can be seen as a measure of compute-capacity. Each bric consists of 8 CPUs and 21 KBytes of SRAM. The Ambric archicture is assembled by stacking up these brics.

2.2.2 Ambric AM2045 Architecture

Ambric AM2045 employs 336 extremely simple custom RISC (32-bit SR and SRD) processors with 45 brics in an array. Processors run at 350 Mhz and access 7.1 Mbits of distributed SRAM. The properties of SR and SRD CPUs for Ambric AM2045 are shown in Table 2.2. Processors execute with a throughput and latency of 1 clock cycle due to the standard register usage as long as the instructions are scheduled in a way that no stall occurs. Other possible reasons for stalling are conditional jumps and usage of multiply-accumulator. Ambric AM2045 architecture supports 792 gigabit per second bisection interconnect bandwidth, 26 Gbps of off-chip DDR2 memory, PCI Express (8 Gbps each way) and upto 13 Gbps of parallel general purpose I/0.

Table 2.2: Specifications of the SR and SRD processors.

	SR	SRD
Instruction width	16	32
Number of Registers	8	20
Local Memory (bytes)	256	1024
ALUs	1	3
Shifter	1-bit	multi-bit
Multiply-accumulate	no	yes

2.3 Field Programmable Gate Arrays (FPGAs)

This section concentrates on Field-Programmable Gate Arrays (FPGAs) and their underlying architecture. We also provide information about the specific device, *Virtex-4* from Xilinx, that is used in our hardware implementations.

FPGAs were invented by Xilinx in 1984. They are structured like the gate arrays form of Application Specific Integrated Circuits (ASICs). The architecture of the FPGAs consists of a gate-array-like architecture, with configurable logic blocks, configurable I/O blocks, and programmable interconnects. Additional logic resources may include ALUs, memory elements, and decoders. The three basic types of programmable elements for an FPGA are static RAM, anti-fuses, and flash EPROM. Segments of metal interconnect can be linked in an arbitrary manner by programmable switches to form the desired signal nets between the cells. FPGAs can be used in virtually any digital logic system and provide the benefits of high integration levels without the risks of expenses of semicustom and custom IC development. An FPGA architecture is presented in figure 2.7.

FPGAs give us the advantage of custom functionality like the Application Specific Integrated Circuits while avoiding the high development costs and the inability to make design modifications after production. The FPGAs also add design flexibility and adaptability with optimal device utilization while conserving both board space and system power. The gate arrays offer greater device speed and greater device density. Taking these into consideration, FPGAs are especially suited for rapid prototyping of circuits, and for production purposes where the total volume is low. On the other hand, FPGAs have the following disadvantages:

- Increased development time and efforts. There are several different high level design tools to ease this burden and we will introduce these design tools in the next chapter. However, architectures that are generated using high level design tools often lead to low-quality results: under-utilized resources and/or lower throughput, compared to the hand-tuned designs;

- Higher interconnect delays in complex architectures that result in larger area and lower clock frequencies compared to ASICs.

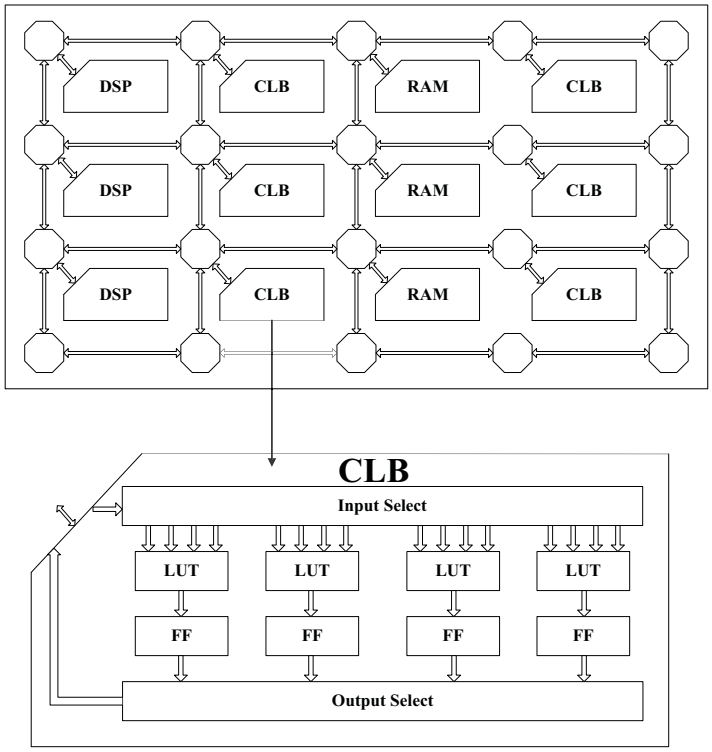


Figure 2.7: **An FPGA architecture and its resources: I/O cells, logic blocks (CLBs) and interconnects.**

FPGA design flow, shown in Figure 2.8, includes the necessary steps [86]: design entry, design synthesis, design implementation and device programming; as well as design verification, required to design an FPGA using Verilog HDL.

Design Synthesis process checks code syntax and analyzes the hierarchy of the given design in design entry step and provides RTL Schematic and technology mapping. This step ensures that given design is optimized for the design architecture that is selected. Created netlist, connectivity of the design, is saved as an NGC file (for Xilinx Synthesis Technology (XST)) or an EDIF file (for LeonardoSpectrum, Precision, or Synplify/Synplify Pro) to be used with one of the following synthesis technology tools:

- Xilinx Synthesis Technology (XST);
- LeonardoSpectrum from Mentor Graphics Inc.;
- Precision from Mentor Graphics Inc.;

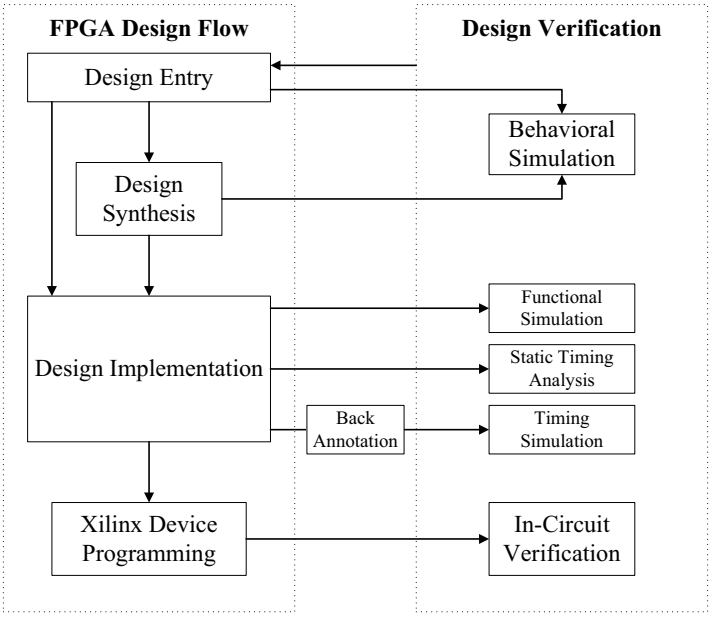


Figure 2.8: **Xilinx ISE design flow and its steps: design entry, design synthesis, design implementation and Xilinx device programming. Design verification occurs at different steps during the design flow.**

- Synplify and Synplify Pro from Synplicity Inc.

Design Implementation follows design synthesis and includes the following steps: translate, map, place and route and programming file generation. *Translate* merges the netlists (generated in design synthesis step) and constraints; and creates a Xilinx design file; *Map* fits the design into the available resources on the user defined target device; *Place and Route* places and routes the design to the timing constraints; and *Programming file generation* creates a bitstream file to be downloaded to the user defined device.

Functional Verification verifies the functionality of the design at different points in the design flow with behavioral simulation (before synthesis), functional simulation (after translate) and/or in-circuit verification (after device programming).

Timing Verification verifies the timing of the design at different points in the design flow with static timing (after Map and/or Place & Route) and timing simulation (after Map and/or Place & Route).

Unlike the two previous architectures, FPGAs do not have any fixed instruction-

set architecture. Instead they provide a fine-grain grid of bit-wise functional units, which can be composed to create any desired circuit or processor. Most of the FPGA area is actually dedicated to the routing infrastructure, which allows functional units to be connected together at run-time. Modern FPGAs also contain a number of dedicated functional units, such as DSP blocks containing multipliers, and RAM blocks.

2.3.1 Xilinx Virtex-4 Family Overview

Virtex-4 family from Xilinx combines advanced silicon modular block architecture with a wide variety of flexible features [84]. This enhances programmable logic design capabilites and makes it a powerful alternative to ASIC technology. *Virtex-4* devices are produced on a state-of-the-art 90-nm copper process using 300-mm wafer technology. There are three different platform families: LX, FX and SX, for *Virtex-4* devices. *LX family* is for high-performance logic applications solution, *SX family* is for high performance solution for digital signal processing applications and *FX family* is for high-performance, full featured solution for embedded platform application. *Virtex-4* hard-IP core blocks includes the PowerPC processors, tri-mode Ethernet MACs, dedicated DSP slices, 622 Mb/s to 6.5 Gb/s serial transceivers, high-speed clock management circuitry and source-synchronous interface blocks.

2.3.2 Xilinx Virtex-4 Architecture

Virtex-4 devices are user-programmable gate arrays. They consist of various configurable elements and embedded cores which are optimized for high-density and high-performance system designs. Functionality of *Virtex-4* devices are I/O blocks, configurable logic blocks (CLBs), block ram modules (BRAMs), embedded XtremeDSP slices and digital clock manager (DCM) blocks.

I/O blocks provide the interface between internal configurable logic and package pins. They are enhanced for source-synchronous applications.

IOB registers are either edge-triggered D-type flip-flops or level-sensitive latches.

CLBs are the basic logic elements for these devices. They provide combinatorial logic, synchronous logic and distributed memory. Each CLB resource is made up of four slices. And each slice is equivalent and contains two function generators (F&G), two storage elements, arithmetic logic gates, large multiplexers and fast carry look-ahead chain. The function generators are configurable as 4-input look-up tables (LUTs). Storage elements are either edge-trigged D-type flip-flops or level sensitive latches.

Block ram modules provide 18Kbit true dual-port ram blocks which are programmable from $16K \times 1$ to 512×36. And these modules are cascadable to form larger memory blocks. Block ram modules contain optional programmable FIFO logic for increased utilization. Each port in the block ram is totally synchronized and independent; and offers three *read-during-write* modes.

Embedded XtremeDSP slices are cascadable and contain 18×18-bit dedicated 2's complement signed multiplier, adder logic and 48-bit accumulator. Each multiplier or accumulator can be used independently.

Virtex-4 uses DCM and global-clock multiplexer buffers for global clocking. **DCM block** provides self-calibrating, fully digital solutions for clock distribution delay compensation, coarse/fine-grained clock phase shifting and clock multiplication/division. There are up to twenty DCM blocks available.

The general routing matrix (GRM) provides an array of routing switches between components. Programmable elements are tied to a switch matrix which allows multiple connections to the general routing matrix. All components in devices use the same interconnect scheme and the same access to the global routing matrix. The device utilization for *Virtex-4 XC4VSX35* is shown in Table 2.5.

2.4 Learning from Existing Parallel Platforms

This section concentrates on two different questions that are natural interest to many researchers:

- What are the advantages and disadvantages of existing parallel platforms: ASICs, DSPs, CMPs, GPUs, MPPAs and FPGAs for a given application

Table 2.3: Specifications of the Xilinx *Virtex-4 XC4VSX35* device.

Configurable Logic Blocks (CLBs)	Array ($Row \times Col$)	96×40
	Logic Cells	34,560
	Slices	15,360
	Max. Distributed RAM (Kb)	240
XtremeDSP Slices		192
Block RAM	18 Kb Blocks	192
	Max. Block RAM (Kb)	3,456
DCMs		8
PMCDs		4
Total I/O Banks		1
Max. User I/O		448

since applications typically exhibit vastly different performance characteristics depending on the platform? This subsection provides a literature survey that compares these existing parallel platforms in implementation of different computationally intensive applications;

- What is the roadmap to design and implementation of a multi/many-core platform using FPGAs that combines the advantages of these existing parallel platforms: CMPs, GPUs, MPPAs and FPGAs?

2.4.1 Comparison of Parallel Platforms

Applications typically exhibit vastly different performance characteristics depending on the platform. This is an inherent problem attributable to architectural design, middleware support and programming style of the target platform. For the best application-to-platform mapping, factors such as programmability, performance, programming cost and sources of overhead in the design flows must be all taken into consideration.

In general, FPGAs provide the best expectation of performance, flexibility and low overhead, while GPUs and MPPAs tend to be easier to program and require fewer hardware resources. FPGAs are highly customizable, while MPPAs and GPUs provide massive parallel execution resources and high memory band-

width. These devices are specific purpose processors and can be used to assist purpose processor in performing complex and intensive computations of applications as accelerators. Three extreme endpoints in the spectrum of possible platforms: FPGAs, MPPAs and GPUs can often achieve better performance than CPUs on certain workloads [75]. They can process tasks loaded off the CPU and send the results back upon completion or FPGAs can be stand alone. The vast computing resources and friendly programming environments of the FPGAs, MPPAs and GPUs make them good fits to accelerate intensive computation.

The dataflow of an application is exploited in FPGAs through parallelism and pipelining. Since CPUs have limited potential for parallelism, FPGAs have one to two orders of magnitude greater throughput rate than CPUs [76–79]. GPUs utilize a graphics pipeline designed for efficient independent processing of pixel data [80]. Multiple pipelines are used to exploit system level parallelism in the GPUs. An an example, GPUs outperform the CPUs by one to two orders of magnitude [81–83] for image processing applications with low numbers of memory accesses, and which are well matched to the instruction set.

NVIDIA's CUDA and RapidMind, and Ambric's aDesigner are high level design language APIs and development environments for programming GPUs and MPPAs respectively. Domain specific parallel libraries can be used as building blocks to ease parallel programming on these platforms. On the other hand, FPGA applications are mostly programmed using hardware description languages such as VHDL and Verilog HDL. Recently there has been a growing trend to use high level languages based on MATLAB or variations of C such as SystemC and Handel-C which aim to raise FPGA programming from gate-level to a high-level. But there are still many limitations for these languages such as control on hardware generation, parallelism limitations, data types and floating point operations.

If a design is well matched to the instruction set of the GPUs or MPPAs, we would expect their implementation to be advantageous. In case of memory access, an application requiring random memory accesses would not be suited to the GPUs with their limited memory on chip and caches or MPPAs with their small local memories. Low memory usage in a regular manner is well suited to the

GPUs and MPPAs. FPGA designs can be implemented in such a way to make efficient use of on-chip memory. This overcomes many of the limitations of the GPUs and MPPAs and makes the FPGAs a more attractive solution.

Here, we present a literature survey that provides different comparisons of architectural platforms while implementing computationally intensive algorithms. Brodtkorb et. al [102] compared three different hardware platforms: CMP (Intel Core 2 Duo 2.4 GHz processor with 4MB cache, 4 GB system memory), GPU (NVIDIA GeForce 8800 GTX with 768 MB graphics memory) and Cell BE (PlayStation 3 with 3.2GHz processor and 256 MB system memory) through implementation of four different algorithms: *solving the heat equation with a finite difference method*, *inpainting missing pixels using the heat equation*, *computing the Mandelbrot set* and *MJPEG movie compression*. These four examples represent different characteristics for computational and memory access properties and can be examples for a wide variety of highly parallel real-world problems. Solving the heat equation and inpainting computation require streaming process with regular and irregular memory accesses; computation of Mandelbrot set and MJPEG movie compression require large number of computations with uniformly and non-uniformly distribution respectively:

- *Solving the heat equation* provides a comparison for how effective each platform is at streaming data;
- *Inpainting missing pixels* provides a comparison for how effective each platform is at streaming data, executing conditionals and computing on a subset of the data;
- *Computing the Mandelbrot set* provides a comparison for how effective each platform is with dynamic workloads and performing floating point operations;
- *MJPEG movie compression* provides a comparison for how effective each platform is with a typical data flow, where the computational intensive part of the code run on an accelerator.

Authors quantify the run-times of **computation** and **memory accesses** compared to the **total run-times** for GPU and CELL BE which is shown in Table 2.4.

Table 2.4: Breaking down the run-times of GPU and CELL BE in terms of computation and memory accesses [102]

		Heat	Inpainting	Mandelbrot	MJPEG
GPU	Memory	75%	55%	0%	80%
	Computation	25%	45%	100%	20%
CELL BE	Memory	10%	10%	0%	5%
	Computation	90%	90%	100%	95%

- *Solving the heat equation:* GPU suffers from the read-backs as can be seen from the frequency of memory reads, therefore GPU barely provides better results from CMP. Cell BE provides better results for this computation;
- *Inpainting computation:* Since GPU keeps a subset of the data in graphics memory, the memory access frequency decreases and brings its performance closer to Cell BE implementation and more better than CMP implementation;
- *Computing the Mandelbrot set:* GPU performs better than Cell BE and CMP due to the computationally intensive calculations;
- *MJPEG movie compression:* GPU again performs better than CELL BE and CMP.

Another computationally intensive computation is the random number generation that is required in most of the financial computations. Random number generation is crucial for embrassingly parallel Monte-Carlo simulations. To understand how these parallel architectures can be used for Monte-Carlo simulations, one needs to understand how to efficiently implement random number generators. Thomas et. al [87] compared different platforms: CPUs (Pentium 4 Core2), GPUs (NVidia GTX 200), FPGAs (Xilinx Virtex-5) and MPPAs (Ambric AM2000) for random number generation. The summary of target platforms is shown in Table 2.5.

Table 2.5: Summary of target platforms [87]

Platform	Device Model	Process (nm)	Die Size (mm^2)	Transistors (Millions)	Max. Power (Watts)	Clock Freq. (GHz)	Parallelism (Threads)
CPU	Intel P4 Core2 QX9650	45	214	820	170	3.00	4
GPU	NVIDIA GTX 280	65	574	1400	178	1.30	960
MPPA	Ambric AM2045	130	?	?	14	0.35	336
FPGA	Xilinx XC5VLX330	65	600	?	30	0.22	N/A

The authors determined the most suitable random number generation algorithm for each platform. Furthermore, they compared the results for different platforms in terms of absolute performance of each platform and estimated power efficiency. These results are shown in Table 2.6. Estimated power efficiency is calculated using the peak power consumption of each device while ignoring the supporting infrastructure: RAM, hard disks etc.

Table 2.6: Comparison of random number generation algorithms while implementing on different platforms (top of the Table) and Comparison of different platforms (bottom of the Table) [87]

	Performance (GSample/s)				**Efficiency** (MSample/joule)			
	CPU	GPU	MPPA	FPGA	CPU	GPU	MPPA	FPGA
Uniform	4.26	16.88	8.40	259.07	15.20	140.69	600.00	8653.73
Gaussian	0.89	12.90	0.86	12.10	3.17	107.52	61.48	403.20
Exponential	0.75	11.92	1.29	26.88	2.69	99.36	91.87	896.00
Geo. Mean	1.42	13.75	2.10	43.84	5.07	114.55	150.21	1461.20
	Relative Mean Performance				**Relative Mean Efficiency**			
	CPU	GPU	MPPA	FPGA	CPU	GPU	MPPA	FPGA
CPU	1.00	9.69	1.48	**30.91**	1.00	9.26	18.00	**175.14**
GPU	0.10	1.00	0.15	**3.19**	0.11	1.00	1.95	**18.92**
MPPA	0.67	6.54	1.00	**20.85**	0.06	0.51	1.00	**9.73**
FPGA	0.03	0.31	0.05	1.00	0.006	0.05	0.10	1.00

As can be seen from the Table 2.6, FPGAs provide the highest performance ratio, 3×, 21× and 31× relative mean performance improvement compared to GPU, MPPA and CPU respectively (it is imporant to note that Xilinx Virtex-5 is more expensive than NVidia GTX 200). GPU provides 9.7× and 6.5× improvement compared to CPU and MPPA respectively. MPPA provides 1.48× perfor-

mance improvement compared to CMP. These values follow the same trend for relative mean efficiency where FPGA results are more efficient in terms of power consumption.

Explicit finite difference option pricing is another area where high performance computing is required due to the computationally intensive calculations [112]. Luk et. al compared implementation of explicit finite difference pricing model using FPGAs (Virtex 4 xc4vlx160) with single and double precision, GPUs (Geforce 8600GT with 256 MB on-board RAM and Tesla C1060 with 4 GB on-board RAM) and CPU (Intel Pentium4 with 4GB of RAM) [113]. The area, performance and power consumption comparison is provided in Table 2.7 for European option pricing problem based on $6K \times 30K$ grids. Area values are provided for FPGA implementation, both single and double precision architectures employ reasonably low area in terms of CLBs; the utilization values for BRAMs and DSPs are around $12\% - 23\%$ and $12\% - 50\%$ respectively. The performance results show that implementing 1 core in an FPGA provides 1.6× and 1.2× acceleration compared to CPU for single and double precision respectively. On the other hand if one replicates more cores on an FPGA such as 8 cores for single and 3 cores for double precision, single precision provides higher acceleration, 12.2×, than GeForce 8600 GT, 9.6×, compared to CPU implementation. On the other hand, the latest generation of GPUs, Tesla C1060, provides 43.9× and 26.6× acceleration with significantly higher power consumption, 187 Watt, compared to FPGA implementation, 5.8 Watts for single and 9.1 Watts for double precision. As can be seen from efficiency values, even though Tesla series GPUs provide higher acceleration compared to FPGAs, their power efficiency are significantly lower.

Yet another application requiring high performance computing is lithography simulations that simulate the imaging process or the whole lithography process. These computations are crucial for printing circuit patterns onto wafers and require high computation time. Cong et. al [114] presented a hardware implementation for accelerating polygon based lithography imaging simulations on FPGA platform (Xtremedata's XD1000 development system [117]) and compared their results with CPU (AMD Opteron 248 2.2 GHz with 4 GB DDR memory) and GPU acceleration

Table 2.7: Comparison of explicit finite difference option pricing implementation using three different platforms: FPGA, GPU and CPU [113].

	FPGA		GPU			CPU
	Virtex 4 xc4vlx160		GeForce 8600GT	Tesla C1060		Intel Pentium 4
Number Format	single	double	single	single	double	double
Slices	5228 (7%)	8460 (12%)	-	-	-	-
FFs	4253 (3%)	6271 (4%)	-	-	-	-
LUTs	5780 (4%)	9891 (7%)	-	-	-	-
BRAMs	37 (12%)	69 (23%)	-	-	-	-
DSPs	12 (12%)	48 (50%)	-	-	-	-
Clock Rate	106MHz	82.9MHz	1.35GHz	1.3GHz		3.6GHz
Processing Speed (M values/sec)	106	82.9	673	3057	1851	69.7
Replication (cores/chip)	8	3	32	240		1
Acceleration (1 core)	1.6×	1.2×	-	-	-	1×
Acceleration (replicated cores)	12.2×	3.6×	9.6×	43.9×	26.6×	1×
Max. Power (W)	5.8	9.1	43	187		115
Efficiency (M values/Joule)	146	27.7	15.6	16.3	9.9	0.6

(NVidia 8800 GTS) for different layout densities, N. Since there is no previous work comparing these platfroms, authors also presented 2D-FFT based image convolution (another method for imaging process) results for an FPGA (Virtex-4) and GPU (NVidia G80). As can be seen from Table 2.8, 2D-FFT based method provides better performance rates for GPU implementation, however this trend is shifted to FPGAs for polygon based simulations.

Table 2.8: Comparison of different platforms, CPU, FPGA and GPU for implementation of polygon based lithography imaging simulations and 2D-FFT based image convolution [114]

N	Polygon Software	Polygon FPGA	Polygon GPU	2-D FFT Software	2D-FFT [115] FPGA	2D-FFT [116] GPU
10	8.0	44.4	32.4	0.2	1.1	3.4
50	1.6	24.5	10.8	0.2	1.1	3.4
100	0.8	12.4	6.7	0.2	1.1	3.4

We conclude this subsection with Table 2.9 that provides advantages and disadvantages of FPGAs, GPUs and MPPAs.

2.4.2 Roadmap for the Future Many-Core Platform

There has been extensive research for the design of a platform that can process the same amount of data in a smaller amount of time or handle a larger amount of data in the same amount of time. Due to the decrease in the clock frequency of processors, the new trend is to design a larger number of cores in platforms such as GPUs, MPPAs and CMPs with varying type and composition. We believe this trend will continue and the future's high performance parallel computation platforms will have many/multi CPUs in their chip architectures.

However, the problem is to determine what type/s of CPUs to employ and what kind of connectivity is required between these CPUs. As an example, GPUs and MPPAs employ 960 cores (NVIDIA Tesla S1070) and 336 cores (Ambric AM2045) in their architectures respectively. This large number of cores provides massive parallelism, but it is also very hard to partition the given algorithm onto hundreds of processors. Thus, low utilization results in inefficient performance with a waste of resources. This is a very important fact considering that a GeForce 8600GT GPU consumes **43** Watts power whereas FPGA power consumption is between **5.8** and **9.1** Watts with a higher acceleration for the given application (option pricing algorithm implementation in financial computation [112]). A natural question that comes to mind is why allow so many processors that cannot be employed? Wouldn't be more efficient if we were capable of employing the required number of processors in our platforms rather than too many, to improve on area, performance and power?

These parallel platforms also provide full connectivity between their CPUs. For example MPPA CPUs are instantiated in a regular grid using 2D communication channels between them. Wouldn't it be more efficient if there was a way to provide required connectivity between CPUs instead of full connectivity? Therefore one could save and use more silicon for memory, more functional CPUs etc.

The type of the processor is another important topic that needs special consideration. As an example, MPPA architecture employs two types of processors: SRD for main computation and SR for managing the data/address provided for SRD. Therefore, the user needs to map the given algorithm onto these two dif-

ferent types of processors which introduces increased complexity for mapping and possibly decreased efficiency in performance. GPUs employed vertex and fragment processors previously, and these are combined for a more efficient solution. We believe that combining these processors into one will lead to more efficient hardware platforms and less design complexity.

GPU architectures employs large number of ALUs by removing the scheduling logic to exploit instruction level parallelism and caches that remove memory latency. Therefore GPUs are simply very powerful number crunching machines. Thus, the future's high performance parallel computation platform should have an ALU dominant architecture to employ more resources for computation by removing as much control logic as possible.

GPU architectures employ a small amount of caches (to be able to put more ALUs inside) and MPPA architectures employ local small memories which results in large memory access latencies. The future's high performance parallel computation platform should have many/multi core processors employed with their own local memories like MPPAs, however memory should not be distributed onto CPUs uniformly.

We foresee a parallel platform where a user can define the number as well as the types of the processors where each processor employs required connectivity internally depending on the assigned part of the algorithm. It should be possible to employ a different amount of memory resources for each processor and the overall platform should only have the required connectivity between CPUs. These CPUs should be simple processors like MPPAs but also ALU oriented like GPUs. FPGAs are perfect platforms for these type of parallel processing architectures since they provide reconfigurability and a path to ASIC design. We therefore believe that it is crucial to have a design tool which can create these type of platforms automatically for a given algorithm and a set of user decisions.

Table 2.9: A comparison between FPGAs, GPUs and MPPAs.

Device	**Advantages**	**Disadvantages**
FPGAs	- Good fit for applications that involve a lot of detailed low-level hardware control operations and have a lot of memory accesses; - High density arrays of uncommitted logic and very high flexibility; - Approximation of a custom chip, i.e. ASIC; - Low power consumption.	- Poor fit for applications that require a lot of complexity in the logic and data flow; - Not easy hardware description language (Verilog and VHDL); - Increased development time and efforts.
GPUs	- Good fit for applications that have no inter-dependences in the data flow and consist of the computations can be done in parallel; - High memory bandwidth and a large number of programmable cores with thousands of hardware thread contests; - Flexible and easy to program using high level languages/tools and APIs; - Relative short design time and efforts.	- Poor fit for applications that have a lot of memory accesses and have limited parallelism; - Constrained design and implementation based on supplied high level languages/tools and APIs; - High power consumption; - High cost and inefficient double precision floating point units; - Expensive branching.
MPPAs	- Good fit for applications that have no inter-dependences in the data flow and consist of the computations can be done in parallel; - Low power consumption; - Easy to program using aDesigner design tool and Relative short design time and efforts.	- Hard to efficiently organize communications over a network that favours local communication over local communication; - Hard to partition given applications onto hundreds of simple processors; - Hard to map the two different types of processors employed in the chip architecture.

Chapter 3

Overview of Design Tools

Field-Programmable Gate Arrays (FPGAs) consist of configurable logic blocks (CLBs) that can be reprogrammed to perform different functions in a matter of seconds. This is the major advantage of FPGAs over Application Specific Integrated Circuits (ASICs). With the increasing on die resources (configurable logic block, memory, embedded multipliers, etc.), FPGAs become attractive to the scientific computing community. However, reconfigurability of FPGAs is both a blessing and a curse since it provides great flexibility in terms of the design of an application with increasing programming complexity. Therefore there is a desperate need for high level fast prototyping systems that can aid designers.

The major goal in the development of the fast prototyping system for real-time matrix computations are better design tools and an easy-to-use, highly integrated prototyping system. There is a desperate need for domain specific design tools. A major step in the right direction is a tool that eases the mapping from algorithm to hardware. Matrix computation algorithms are typically written and tested using MATLAB code or Simulink model based environment, and there are a number of tools that translate such algorithms to a hardware description language such as System Generator for DSP and AccelDSP from Xilinx, Simulink HDL Coder from Mathworks and Synplify DSP from Synplicity. Synplify DSP is not considered in this Thesis since it provides very similar functionality to System Generator for DSP but allows the user to target devices from several different vendors that would be very useful for a user who wants to compare devices from

different vendors.

Therefore, the main goal of this chapter is to learn from (dis)advantages of existing high level design tools, and propose a roadmap for a tool that combines most of advantages provided by these existing design tools. This chapter provides the following information:

- Overview, design flow, (dis)advantages of existing high level design tools: model based tools (System Generator and Simulink HDL Coder), MATLAB code based (AccelDSP) and C/C++ code based (System-C, Catapult-C, Impulse-C, Mitrion-C, DIME-C, Handel-C, Carte, SA-C, Streams-C and Napa-C);
- A case study for comparison of Filter Design HDL Coder Toolbox with a tool designed by us, Xquasher, for filter architecture implementations;
- A roadmap for a design tool that can combine the advantages provided by the existing high level design tools.

3.1 System Generator for DSP

System Generator [120] is a high level design tool specifically designed for Digital Signal Processing (DSP) applications. System Generator is designed by Xilinx to be used in model-based design environment, Simulink from Mathworks [119], for modelling and implementing systems in FPGAs. Simulink provides a powerful component based computing model including several different blocks to be connected together by the user for designing and simulating functional systems. System Generator provides similar blocks which are used and connected the same way Simulink blocks does but target FPGA architectures to design discrete time systems which can be synchronous to a single or more clocks. The simulation results of the designed systems are bit and cycle accurate which means simulation (through Simulink) and hardware (through System Generator) results are exactly match together. We present two simple design examples, multiply & accumulate and FIR filter, in Figure 3.1 that are designed using System Generator for DSP.

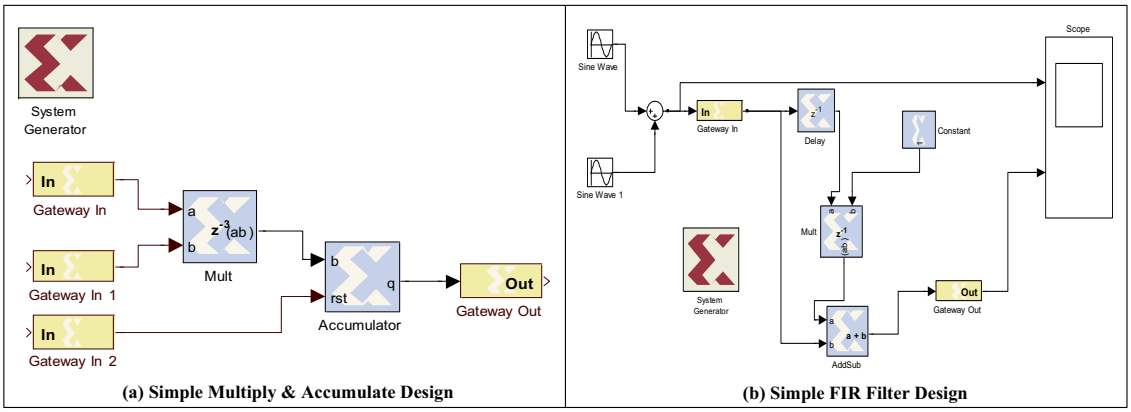


Figure 3.1: **Two simple design examples using System Generator for DSP are presented: multiply & accumulate and FIR filter.**

System Generator can be used in several different ways:

- *Algorithm exploration* to get an idea for the design problems with a little translation into hardware;
- *Implemeting part of a larger design* through a HDL wrapper that represents the entire design and uses the design part generated by System Generator as a component;
- *Implementing a complete design* with automatically generated **HDL files** describing the design, **a clock wrapper** producing the clock; and **clock enable signals** and **testbench** to test and debug the design using Simulink simulations.

There are several different compilation types that System Generator provides:

- *Netlists*, HDL Netlist (used more often) and NGC Netlist, which are collection of HDL nad EDIF files;
- *Bitstream* that is ready to run on an FPGA platform where a user can specify a wide variety of different platforms;
- *EDK Export Tool* to export the design to the Xilinx Embedded Development Kit (EDK);

- *Timing analysis* which is the timing report of the design.

System Generator for DSP is also capable of using Synplify, Synplify Pro and Xilinx XST to synthesize the design and producing Verilog HDL or VHDL as netlist of the design. The Xilinx blockset includes the following building blocks to design digital systems in Simulink environment:

- *Basic Element Blocks:* Addressable shift registers, inverters, muxes, counters, delays etc.;
- *Communication Blocks:* Convolution encoder, Reed-Solomon decoder, Viterbi decoder etc.;
- *Control Logic Blocks:* Constant, EDK processor, logical, Picoblaze microcontroller etc.;
- *Data Type Blocks:* parallel to serial, serial to parallel, convert, shift etc.;
- *DSP Blocks:* DSP48s, FFT, DAFIR, FIR, linear feedback shift register etc.;
- *Math Blocks:* Accumulator, addsub, mult, negate, sinecosine etc.;
- *Memory Blocks:* Addressable shift register, single port RAM, dual port RAM etc.;
- *Shared Memory Blocks:* Shared memory, from FIFO, to FIFO etc.;
- *Tool Blocks:* ChipScope, Clock probe, FDATool, Modelsim, WaveScope, Resource estimator etc.

System Generator for DSP also provides composite blocks that are specifically needed in particular applications such as communication, digital signal processing, imaging and math. We present some example blocks in Figure 3.2. A block in Systems Generators' library operates on Boolean or arbitrary precision fixed-point values which is different than Simulink providing floating-point values. Therefore Simulink and System Generator blocks are connected to each other with a gateway block (gateway in/out) which converts floating point precision to the given fixed

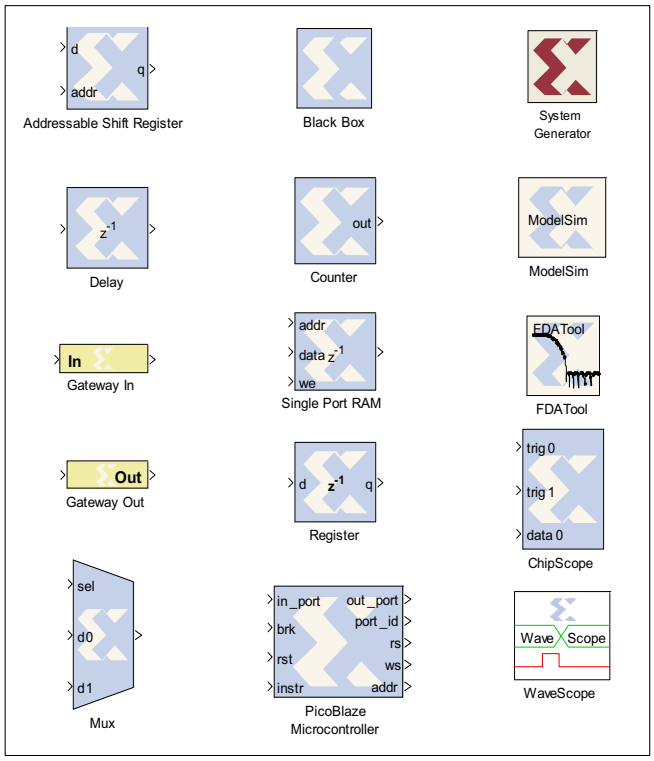


Figure 3.2: **Example blocks from System Generators' library.**

point precision or vice versa using sampling. It is important to note that most of the Xilinx blocks are polymorphic, capable of determining the appropriate output type. A user can define full precision or a user defined precision in the blocks of the design as well as how the quantization overflow will be handled and it is possible to design multi-rate systems.

Advantages of System Generator are:

- Users do not need to be experienced in Xilinx FPGAs or Register Transfer Level (RTL) design methodologies to be able to use System Generator since System Generator enables a "push a button" transition from specification to implementation and FPGA implementation steps, including synthesis and place & route are automatically performed, [118] reports a 10:1 improvement productivity improvement (time) while designing SDR waveforms compared to traditional RTL design approaches;

- Xilinx blocks are optimized architectures since these are generated by Xilinx IP core generators (coregen);
- System Generator for DSP is used in Simulink which is a user friendly visual environment with Xilinx specific blockset;
- System resource estimator block is a specific block that is provided to quickly estimate the area of the design in terms of slices, lookup tables, flip-flops, block RAMs, DSP48s, I/O blocks before place&route and provides a general idea about the design to the user without going through the real hardware implementation steps;
- System Generator for DSP provides an integration platform that a user can combine RTL through *black box block*, Simulink, MATLAB *through MCode block*, and C/C++ through *MicroBlaze embedded processor* components of a system, simulate and implement the design;
- Hardware co-simulation that is provided by System Generator is considerably faster than traditional HDL simulators [121];
- Since System Generator for DSP works in Simulink environment, it is possible to compare the results of the System Generator design which realizes into fixed point architecture with floating point results and determine the required bit width for the FPGA implementations;
- *MCode blocks* to interpret a MATLAB (.m) code providing an efficient solution to generate control logic and finite state machines.

Disadvantages of System Generator are:

- There is no implicit synchronization mechanism provided in System Generator, therefore it is the users' responsibility to satisfy synchronization explicitly. Even though System generator provides several blocks that can be employed for synchronization such as the blocks that have the capability to provide a valid signal when the sample is valid (such as FFT, FIR, Viterbi),

manual synchronization requires a huge effort. For example, System Generator creates a seperate clock wrapper to control a block to make the HDL flexible. And its users' responsibility to generate clock and clock enables when this block is added to a larger design.

We present relatively a more complex design example in Figure 3.3, implementation of the Matching Pursuits algorithm for channel estimation, that is designed and implemented by us using System Generator for DSP. Even a small change in the architecture affects blocks inside the design and requires a large amount of manual synchronization efforts. Architectural changes requiring a large amount of time includes but not limited to bitwidth changes to achieve different precision levels, adding additional resources for higher throughput values and removing resources for hardware simplification;

- System Generator designs often result in inefficient performance values since designed IP cores are treated as black boxes by current synthesis tools and therefore synthesis tools do not perform optimization that cross module boundaries. On the other hand, RTL designs provide more independency to the synthesis tools to perform better optimizations;

- Not all the blocks from Simulink are available in Xilinx blockset. Therefore, a user needs to design his/her own blocks (with the same Simulink block functionality) using existing basic Xilinx blocks;

- Determination of bit-width and the place of binary point are required for every block output as well as constant blocks in System Generator implementations since the resulting architecture is realized in fixed-point arithmetic;

- Since synchronization requires manual effort, the control units that controls the design gets more complicated in more complicated architectures like architectures that use resource sharing;

- High level abstraction that is provided with the *MCode blocks* come at a cost of inefficiency in hardware realization in most of the cases (depending on the complexity of the given code). *MCode blocks* supports a limited subset

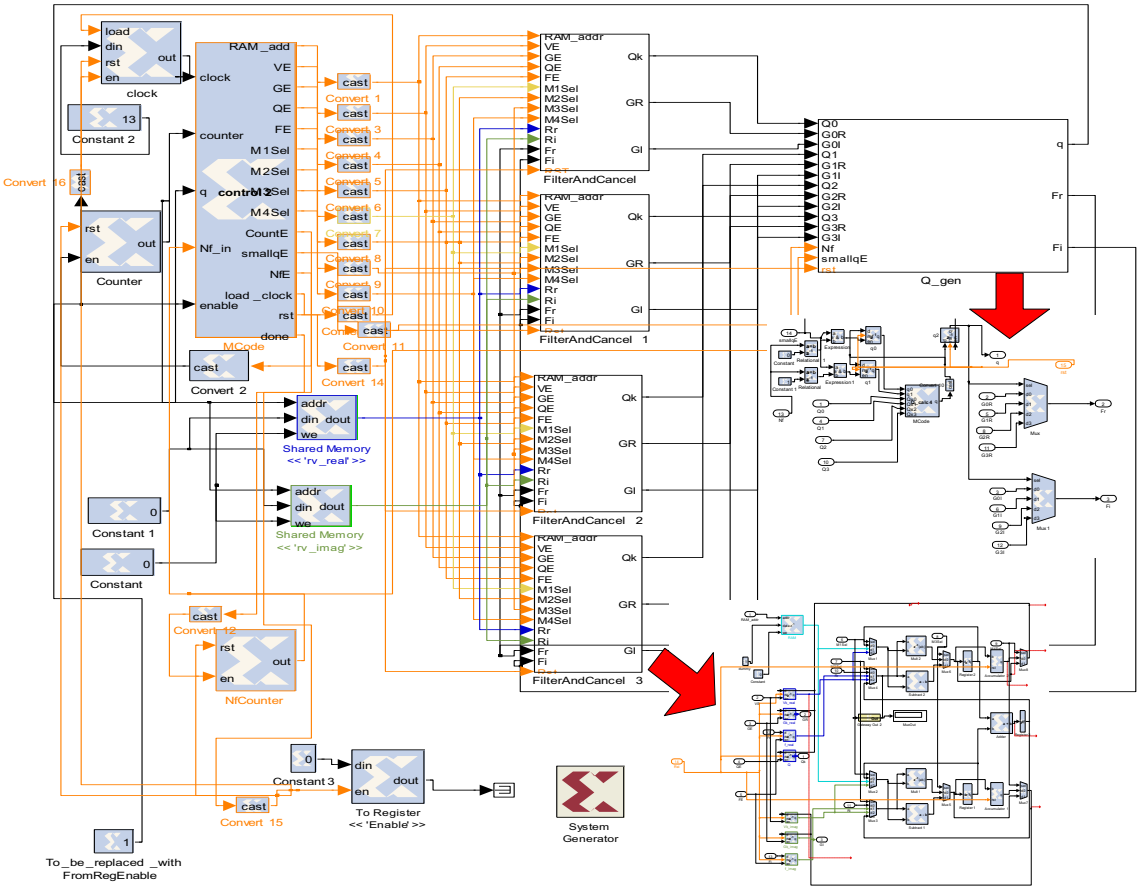


Figure 3.3: **Design and implementation of the Matching Pursuits algorithm for channel estimation using System Generator. Even a small change in the architecture affects blocks inside the design and requires a large amount of manual synchronization efforts.**

of the MATLAB language: assignment statements, simple and compound if/else/elseif end statements, switch statements, arithmetic expressions that involve only addition and subtraction, addition, subtraction, multiplication and division by a power of two where all inputs and outputs should be fixed-point value.

We see System Generator **as the best tool provided** for MATLAB code environment because of its "push a button" transition from specification to implementation, its provided libraries to design optimized architectures faster and its ability to compare fixed point and floating point results for any part of the design. However a user needs to satisfy synchronization explicitly in their designs through

control units which gets more and more complicated in larger designs. System Generator provides a solution for this problem, MCode blocks, that are suitable for this particular usage. However these blocks come at a cost of inefficiency in hardware realization and do not support a wide variety of MATLAB language constructs. Because of these reasons, Xilinx designed another tool, AccelDSP Synthesis Tool, specifically for hardware generation from a given MATLAB (.m) code which we introduce this design tool in the next section.

3.2 AccelDSP Synthesis Tool

AccelDSP Synthesis Tool [122] is another high level tool specifically designed for Digital Signal Processing (DSP) applications. AccelChip (founded in 2000) is acquired by Xilinx in 2006 and became a part of XtremeDSP solutions with its new name: AccelDSP. The purpose of AccelDSP is to transform a MATLAB floating-point design into a hardware implementation that targets FPGAs. AccelDSP provides an easy-to-use Graphical User Interface (GUI) to design MATLAB code and to control design tools (various industry-standard HDL simulators and logic synthesizers) in an integrated environment. AccelDSP automatically generates bit-true and cycle-accurate HDL codes which are ready to synthesize, implement and map onto an FPGA hardware.

AccelDSPs' design flow is shown in Figure 3.4 and includes the following steps:

- *Analysis of the given input MATLAB floating-point design.* This step performs the compatibility verification of the given MATLAB code to the AccelDSP coding sytle guidelines;

  AccelDSP generates architectures that work with streaming data. The streaming model to simulate the infinite stream of data entering and leaving the design is defined in MATLAB using a *script-file*. The script-file is a top level design that calls a *function-file*, **the main program**, in a streaming loop. AccelDSP synthesizes the main program and its subfunctions into the hardware.

- *Verification of the Floating-Point Design.* This step creates reference/golden results using floating-point arithmetic. And a user can verify the correctness of the code and compare floating-point results with the fixed-point results generated in the next steps;

- *Transformation of floating-point design into fixed-point design.* This step includes the analysis performed by the AccelDSP to determine the types and shapes of variables and generation of a fixed-point model of the design;
  Since C++ simulations run faster, AccelDSP uses C++ to generate the fixed point design by default (there is also an option to choose MATLAB for this simulation).

- *Verification of the fixed-point design through MATLAB simulations.* This step provides information to the user to compare fixed-point results with floating-point results and ensure the correctness of the design;
  If the required precision is not enough or more than desired, a user can redefine the bitwidths for variables and rerun this analysis.

- *Generation of RTL HDL code for the design.* This step generates HDL code (Verilog or VHDL) as well as a TestBench for verification of the design;

- *Verification of the RTL model.* This step verifies the RTL model on a HDL Simulator (Modelsim or Riviera) using the automatically generated Test-Bench unit and compares the simulation results with the MATLAB fixed-point simulation results;

- *Synthesize the design.* This step performs synthesis using a RTL synthesis tool (XST from Xilinx or Synplify Pro from Synplicity);

- *Implement the design.* This step implements the design using ISE (from Xilinx) implementation tools;

- *Verify the gate level design.* This step ensures that the implemented design is bit-true with the original fixed-point MATLAB design.

AccelDSP design tool provides the following synthesis flow:

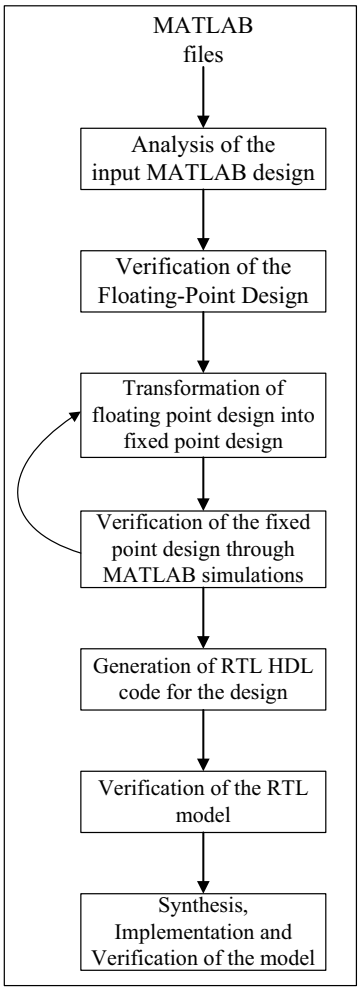


Figure 3.4: **The design flow for AccelDSP.**

- *ISE Synthesis* implements the project using ISE software and verifies the design using HDL gate-level simulations;
- *System Generator* includes converting the design into an System Generator block that can be used in larger System Generator designs;
- *HW Co-Sim* implements the project using ISE software like ISE Synthesis flow, but simulates in a hardware like platform. Available platforms are Virtex-5 on an ML506 Platform, Spartan 3-A DSP 1800A Starter Platform and Virtex-4 on an ML402 Platform. This synthesis flow provides two main benefits: lower simulation time and a proff that the design works properly in

the target platform or not.

The advantages of AccelDSP design tool are;

- AccelDSP is particularly beneficial for user who prefers to use MATLAB (.m) code to realize hardware;
- AccelDSP designed streaming applications in mind. Whether generated designs will be a part of a larger design or not, AccelDSP generates a handshaking protocol for the design which controls the flow of data in and out of the design;
- AccelDSP is capable of generating a *System Generator Block* that can be used in a larger design;
- AccelDSP provides built-in reference design library, *AccelWare* [125], that includes a library of synthesizable MATLAB equivalent functions like *sine*;
- AccelDSP provides a detailed analysis for floating-point to fixed-point conversion. Therefore a user can decide if the precision enough; and if not, AccelDSP lets user to manually adjust the model to reduce quantization error and increase the fidelity of the output;
- AccelDSP is capable of providing a high level design space exploration to determine design trade-offs by giving capabilities: unrolling a loop and inserting pipeline stages, to the user.

The disadvantages of AccelDSP design tool are;

- AccelDSP generated designs result in inefficient architectures in terms of area and timing compared to hand-coded results;
  Nissbrandt et al. [156] compared AccelDSP/AccelWare and Hand-Code/ Coregen implementations for various signal processing algorithms including FFT, 10×10 matrix multiplication, FIR filter, CORDIC and *Constant False Alarm Rate (CFAR)*. We presented their results in Figure 3.5 in terms of area and required calculation time. The authors concluded that AccelDSP should

	FFT		Matrix Multiplication		FIR Filter		CORDIC		CFAR	
Slices	2564	1575	3797	194	301	304	454	287	4141	2099
Registers	4915	3465	4404	720	1065	606	838	708	3654	2490
LUTs	7479	2689	5576	12	62	464	981	727	4850	2680
DSP48s	40	40	100	140	16	16	0	0	0	0
BRAMs	44	3	0	0	0	0	0	0	0	0
Max. Freq.	151.1	373.5	50.5	242.8	196.7	213	259.9	358.2	67	50
Calculation Time	47.45	8.53	445	11.18	344	422	34.2	25.2	577	125

Figure 3.5: **Comparison of AccelDSP/AccelWare and Hand-Code/Coregen implementations for various signal processing algorithms: FFT,** 10×10 **matrix multiplication, FIR filter, CORDIC and *Constant False Alarm Rate (CFAR)* [156]. Results are presented in terms of area and required calculation time.**

be used as a method for platform decision since the drawbacks of AccelDSP generated designs: performance, reliability of the description, readability and the problems of maintaning the source, are more important than the reduced design time.

- AccelDSP design tool requires the input MATLAB file to be written in a very specific format. This includes generation of a script file with streaming loop that requires the designer to think in a very different way than usual and this code modification can take considerably large amount of time;

- Even though AccelWare library provides some reference designs, many built-

in functions, operators and shapes are not supported [124];

- Users do not have much control on the architectures that are generated such as how many arithmetic resources are used, how they are resource shared and binded etc.;

- The inputs to the main program should be a scalar, a row vector or a column vector. Other argument types are not supported like matrices;

- AccelDSP allows maximum of 53 bits as bitwidth, and automatically quantizes a number to the fixed-point. However if a number cannot be exactly represented with a specific bitwidth, AccelDSP employs largest number of bitwidth, 53, for that particular variable. This can drastically effect the quality of the design, therefore it is users' responsibility to recognize it from the provided reports and trim back to a reasonable number of bits.

We recommend to use AccelDSP Synthesis Tool for generation of small and less complex building blocks where used algorithms employ AccelDSP supported built-in MATLAB functions (or easy to rewrite) and are easy to change to streaming behaviour. These small blocks can be employed in a larger system using System Generator for DSP.

3.3 Simulink HDL Coder

Simulink HDL Coder [126] is another high level design tool which generates HDL code from *Simulink* [119] models and *Stateflow* [127] finite state machines. Simulink HDL coder also provides interfaces to combine manually-written HDL codes, HDL Cosimulation blocks and RAM blocks in its environment.

A user uses Simulink HDL Coder via the following steps:

- A user creates a Simulink design to simulate a desired application;

- If the simulation requirements are met, user runs the Simulink HDL Coder compatibility checker for the designs' suitability for HDL code generation;

- User cosimulates the design within Simulink using *EDA Simulink Link MQ*, *EDA Simulator Link IN* or *EDA Simulator Link DS* softwares;
- If simulink design is compatible to generate HDL code, user runs Simulink HDL coder to generate HDL files for the design as well as a testbench in either Verilog or VHDL which are bit-true and cycle accurate;
- User uses testbench and HDL simulation tools to test the generated design;
- If the user is satisfied with the design, he/she can export HDL codes to synthesis and layout tools for real hardware implementation. Simulink HDL coder also generates required scripts for the Synplify family of synthesis tools and ModelSim from Mentor Graphics.

The supported blocks in Simulink HDL Coder design tool are;

- Simulink built-in blocks;
- Stateflow chart;
- Signal Processing Blockset tools;
- Embedded MATLAB Function block;
- HDL-specific block implementation library including FFT, RAMs, bitwise operators etc.
- User-selectable optimized block implementations provided for commonly used blocks;
- EDA Simulator Link MQ HDL Cosimulation block;
- EDA Simulator Link IN HDL Cosimulation block;
- EDA Simulator Link DS HDL Cosimulation block.

Simulink HDL Coder design tool generates the required interfaces for the following environments;

- Black box subsystem implementation;
- Cosimulation using Mentor Graphics Modelsim HDL simulator via EDA Simulator Link MQ;
- Cosimulation using Cadence Incisive HDL Simulator via EDA Simulator Link IN;
- Cosimulation using Synopsis Discovery VCS HDL simulator via EDA Simulator Link DS.

Code Generation Control Files give more control to the user over the overall or some part of the design by specifying some of the properties of certain blocks in the beginning, saving them in persistent form and reusing them if desired in the future designs. A code generation control file is simply MATLAB (*.m*) file and currently supports selection and action statements. A user selects a group of blocks within a model, block type and location need to be defined, with selection commands. An example is choosing all of the delay blocks. Then user applies transformations to the the selected model components with action statements. Some examples are choosing different implementation methods for the blocks which might include optimization for speed and/or area and specifying the stages for generation of output pipeline stages. A code generation control file is attached to the design and executed when code generation process is invoked. If no specifications are provided, a default code generation file is created.

Simulink HDL coder provides a detailed code generation report which eases the tracebility of hundred lines of codes generated automatically. Simulink HDL coder provides two types of linkage between the model and generated HDL code. *Code-to-model* and *Model-to-code* are hyperlinks let user to view the blocks or subsystems from which the code was generated and generated code for any block in the model respectively. Model-to-code tracing is supported for subsytems, simulink blocks, embedded MATLAB blocks and stateflow charts. Simulink HDL coder also lets user to add text annotations to generated code by directly entering text on the block diagram as Simulink annotations and placing DocBlock and entering text comments at the desired level of the model.

Simulink HDL coder is supported by *Stateflow HDL Code Generator* from MathWorks. Stateflow can be embedded into Simulink HDL coder designs to describe complex system behaviour via Finite State Machines (FSM) theory, state-transition diagrams and flow diagram notions. Stateflow is described using a chart: Classic (default), Mealy and Moore, to model a FSM or a complex control algorithm which also supports hierarchy: states containing other states, parallelism: simultaneously active multiple states and truth tables. There are some limitations to Stateflow HDL code generation such as multi-dimensional arrays are not supported in charts, MATLAB functions other than *min* and *max*; MATLAB workspace data and C math functions are not supported in the imported code. The coder also does not support recursion through graphical functions.

Simulink HDL Coder also provides *Embedded MATLAB Function Block* which automatically generates HDL code from a MATLAB (*.m*) file as well as a list of design patters which are collection of different examples including counters, shift registers, adders, comparators etc. Embedded MATLAB Function Block also employs fixed-point arithmetic via *fi* function that is in *Fixed-Point Toolbox*. This block can be used for generation of control logic and simple finite state machines. As an example, a user uses M-function equivalent of operators such as $A + B$ and A^B are defined as *plus(A,B)* and *mpower(A,B)*.

The advantages of Simulink HDL Coder design tool are:

- Simulink HDL coder lets users to realize hardware directly from Simulink designs and stateflow charts;
- Simulink HDL coder provides Code Generation Control Files options that a user can specify the properties of certain blocks including different implementation methods and pipeline stages, save and reuse these control files;
- Simulink HDL coder provides a detailed code generation report from code-to-model and model-to-code and lets user to add text annotations in different ways;
- Simulink HDL coder lets user to insert distributed pipeline using Embedded MATLAB Function Blocks and Stateflow charts. Inserting pipeline lets user

to gain higher clock rates using pipeline registers with the price of increasing latency;

- Simulink HDL Coder employs Embedded MATLAB Function Block to automatically generate HDL code from a MATLAB ($.m$) file. This Block also provides a set of examples which are ready to use called design patterns.

The disadvantages of Simulink HDL Coder design tool are;

- Not all the Simulink built-in blocks are supported. Code Generation Control Files are under development, therefore provides access to some certain blocks with certain capabilities.

- Embedded MATLAB Function Block has its own limitations and do not support all the operations such as nested functions and use of multiple values in the left side of an expression. HDL compatibility checker that provided is only capable of processing a basic compatibility check on this block.

3.4 C-based High Level Design Tools

Even though MATLAB is the de facto standard language for many computer vision, signal processing, and scientific computing researchers, there are also a large number of C-based high level design tools. These design tools are used for automatic hardware generation providing a faster path to hardware with the cost of relatively inefficient use of hardware resources. C-based high level design tools express parallelism through variations in C (pseudo-C) or compiler or both. Ideally, it is best to use pure *ANSI-C* without any variation in C and exploit parallelism through compiler that ports C code into hardware; therefore a user does not need to learn a new language. On the other hand, if parallelism is expressed through variations in C code (pseudo-C code), it is important to note that there is no standard language and each tool uses its own language constructs requiring a user to learn that specific language to be able to use the high level design tool.

Most of the C-based high level design tools have the following characteristics:

- Proprietary ANSI C-based language which do not support all ANSI C features;
- Usage of extra pragmas for corresponding compilers to generate hardware for the given C code;
- Additional libraries of functions/macros for extensions that are required with the given code;
- Requirement of specific programming style to achieve maximum optimization in terms of area, throughput, power consumption etc.;
- Capability of generating both hardware units and I/O interfaces for external resources.

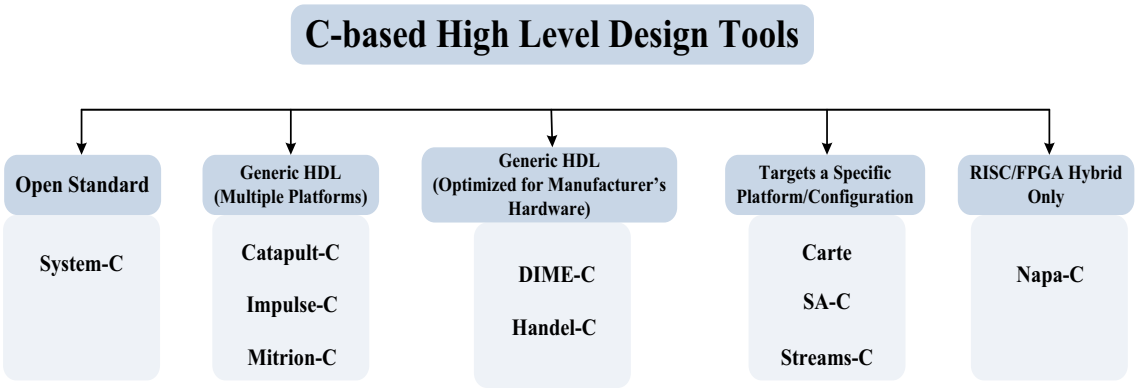


Figure 3.6: **C-based high level design tools are divided into 5 different families including Open Standard: System-C; tools producing generic HDL that can target multiple platforms: Catapult-C, Impulse-C and Mitrion-C; tools producing generic HDL that are optimized for manufacturer's hardware: DIME-C, Handel-C; tools that target a specific platform and/or configuration: Carte, SA-C, Streams-C; tool targeting RISC/FPGA hybrid architectures: Napa-C [136].**

C-based design tools can be divided into 5 different families as shown in Figure 3.6 with the corresponding design tools [136] :

- *Open Standard:* System-C [145], [146];

- *Tools producing generic HDL that can target multiple platforms:* Catapult-C [153], [154], Impulse-C [151] and Mitrion-C [138];
- *Tools producing generic HDL that are optimized for manufacturer's hardware:* DIME-C [141], Handel-C [140];
- *Tools that target a specific platform and/or configuration:* Carte [142], SA-C [147], Streams-C [149];
- *Tool targeting RISC/FPGA hybrid architectures:* Napa-C [155].

We further investigate these high level design tools in more detail in the following paragraphs:
Mitrion-C [138]:

- is a pseudo-C FPGA programming language that is developed by Mitrionics [137] where a user can specify hundreds/thousands of interdependent processes while Mitrion-C hides the synchronization between these various processes from the user;
- uses the *Mitrion Software Development Kit* which compiles Mitrion-C programs and configures a *Mitrion Virtual Processor* optimized for the chosen FPGA platform (without any support for *place&route process*);
- generates generic HDL files which can target multiple platforms;
- provides an abstraction for the user by automatically connecting required memory units and external devices that also comes with a price of restricted clock speed (the main weakness of this tool);
- some central concepts of the Mitrion-C language are rather different from ANSI-C programming such as no support for pointers; Mitrion-C concentrates on parallelism and data-dependencies while traditional languages are sequential and center on order-of-execution;
- most statements are assignments in Mitrion-C where all clauses are expressions except declaration and watch statements. Therefore, *if, for* and *while*

statements of other C-languages are represented as expressions where they all return a result value;

- types specify the kind of value to be stored (e.g. boolean or integer) as well as the precision of the data, i.e. the bit-width of that data. User should define the exact number of bits required for the data explicitly and there is only a single base kind for each type;

- type options include *scalar-type:* integer, unsigned, or positive integer, boolean variable, bits, floating point value or a type used as a value; *collection-type:* lists, vectors and streams; and *explicit type declaration for a tuple* where the elements in the tuple can be of different types;

- supports floating point data types with arbitrary width.

Handel-C [140]:

- is a pseudo-C FPGA programming language that is developed by Celoxica [139]. A user can write sequential programs (using *seq command*) as well as parallel programs to gain maximum benefit in performance from the target hardware with parallel constructs (using *par* command) where parallel processes can communicate using channels: a point-to-point link between two processes;

- is a mature language therefore provides a larger set of libraries compared to some other C-based high level design languages like Mitrion-C;

- requires more programming effort since Handel-C does not provide a virtual processor like Mitrion-C that automatically generates interfaces with external resources and memory; therefore a user is required to define interfaces using available constructs;

- provides many options to perform low level optimizations to the generated architecture;

- supports ANSI-C types except *float, double and long double*; a user can still use floating point through a set of macros where he/she should specify the minimum width required to minimize hardware usage for these types;
- no support of automatic type conversion;
- each statement should take one clock cycle therefore a statement can only contain a single assignment or an increment/decrement (such as *for* statements' initialization and iteration steps are written as statements rather than expressions); therefore complex statements in ANSI-C requires to be rewritten as multiple single statements;
- generates generic HDL files which can target multiple platforms but optimized for manufacturer's hardware.

Impulse-C [151]:

- is based on standard ANSI-C and developed by Impulse Accelerated Technologies [150];
- targets multiple platfroms by generic HDL generation through multi-process partitioning;
- supports standard C development tools including standard C debugging tools;
- is accompanied with a software-to-hardware compiler, *CoDeveloper*, to optimize C code for parallelism and generate hardware/software interfaces;
- supports parallelism in both the application level and the individual process level by providing loop unrolling, pipelining and instruction scheduling that achieves parallelism at the C statement level;
- programming model consists of communicating processes which support dataflow and message-based communications and data streams are implemented through buffered communication channels;

- enables easy partitioning of the given application between the embedded processor and FPGA device;
- implements each independent, potentially concurrent, processes as seperate state machines;
- requires different C coding techniques to increase the performance of the implementation and/or reduce the size of the given C code;
- supports fixed-point arithmetic (three fixed-point bit widths: 8, 16, and 32 bits) in the form of macros and datatypes which let a user to express fixed-point operations in ANSI-C language and to perform computations either as software (using embedded CPU) or as hardware modules (using FPGAs).

Catapult-C [153] [154]:

- is based on pure, untimed, industry-standard ANSI C++ and developed by Mentor Graphics [152];
- uses object oriented programming model properties: levels of IP generation and reuse and templates to generate correct-by-construction RTL of a given algorithm without the use of any extensions or pragmas;
- employs *algorithmic C data types* which supports arbitrary-length (scaling to any size while uniformly preserving the semantic) bit-accurate integer and fixed-point data types allowing users to easily model bit-accurate behavior in the architecture implementation;
- provides built-in quantization and overflow modes to the user;
- enables application of various high-level constraints into the design including loop unrolling and pipelining, loop merging, RAM, ROM, or FIFO array mapping, resource allocation and sharing, memory resources merging and memory bit-width re-sizing;
- automatically synthesizes interfaces to the external hardware including streaming, single/dual-port RAM, handshaking, FIFO etc.;

- provides an intuitive graphical user interface such as displaying design bottlenecks and inefficiencies in the hierarchical Gantt charts and micro-architecture what if analyses;
- allows technology independence such as support for ASIC and FPGA implementations;
- creates cycle-accurate RTL netlists in either VHDL, Verilog, or System C as well as simulation and synthesis scripts for Design Compiler, Precision Synthesis and ModelSim;
- provides technology specific libraries [154] to infer specific hardware instances from a given C code. However these code modifications consumes a significant amount of development time [64];
- applicable to a small code size where the biggest line of code is 480 and no other study reported on the C code size [65].

DIME-C and Carte exploit parallelism through compiler and receives a subset of ANSI-C as input. DIME-C is based on a subset of ANSI-C, developed by Nallatech [141] and provides a GUI, *DIMEtalk*, enabling a user to create a network of hardware components visually. DIME-C does not require a user to learn the syntax and semantics of a new language; and can be compiled and debugged using standard C compiler. However a user needs to know the elements of the supported subset of ANSI-C. DIME-C generates VHDL codes with poor cycle accuracy. On the other hand Carte is specifically targets a specific platform. Carte is based on a subset of ANSI-C or Fortran, developed by SRC Computers [142] and integrates the computational power of MAP processors [143] and microprocessors where it automatically controls the flow between the MAP processors and microprocessor in a way that the most appropriate resource is applied to the code at the optimal time [144].

Both SA-C and Streams-C targets a specific platform and/or configuration. SA-C is a pseudo-C FPGA programming language that is developed at the Computer Science Department at Colorado State University [147] specifically for image

and signal processing applications. SA-C provides additional features to C compensating and exploiting features of FPGAs. SA-C is a single assignment language and its additonal features include true multidimensional array usage to be able to use pointers that index large arrays of data, variable bit-precision data types as well as fixed-point data types and allowing variable names to be reused. Streams-C is developed at Los Alamos National Laboratory which is a stream-oriented sequential process modeling following the Communicating Sequential Processes (CSP) parallel programming model [148] where different objects are defined as processes, streams and signals [149].

3.5 A Case Study for Filter Designs using Domain Specific High Level Design Tools

We believe that the majority of these tools, that are introduced in the previous sections, attempt to be everything to everyone. We present particular disadvantages of these tools, and show that they often fail to provide results that are close to the hand-coded designs. Therefore, it is important to take a more focused approach by targeting specific algorithms. This chapter presents different tools for designing filters that typically require deeply pipelined, streaming architectures: Filter Design HDL Coder Toolbox from Mathworks and Xquasher that is designed by us, compares these filter design tools and shows that these domain specific tools that are designed specifically for filter generation can provide results close to the optimal (the results of a hand-coded design). The next subsections introduce these tools and compare them via FIR filter design using different design methods and number of taps.

We also compared these results with another domain specific design tool, Finite/Infinite Impulse Response Filter Generator from Spiral team [135] at Carnegie Mellon University. This tool generates a Transposed Direct Form implementation of a Finite Impulse Response (FIR) filter (or an Infinite Impulse Response (IIR) filter) from a given standard difference equation. The multiplier blocks inside the design are generated using the Spiral Multiplier Block Generator. Spiral results

are seen as the optimal results for an FIR filter design, however the specified filter tap cannot be larger than 20 which is a major drawback for Spirals' design tool.

In the following two subsections, we introduce the design tools: Filter Design HDL Coder Toolbox and Xquasher.

3.5.1 Filter Design HDL Coder Toolbox

Filter Design HDL Coder Toolbox is specifically designed for automatic hardware generation (Verilog or VHDL) of FIR (antisymmetric, transposed, symmetric etc.) and IIR (SOS IIR direct form I, II etc.) filters using GUI or MATLAB command-line interface [133]. Filter Design HDL Coder Toolbox is also capable of multirate filter structures and cascade filters (multirate and discrete-time). Since the tool generates Simulink HDL Cosimulation block(s), a user can cosimulate the generated filter design via EDA Simulator Link MQ, EDA Simulator Link IN and EDA Simulator Link DS. It is important to note that if the target language is Verilog, generation of a cosimulation model is not supported and a user can only use single-rate filters. Filter Design HDL Coder Toolbox also generates script files to support third-party design tools (Mentor Graphics ModelSim SE/PE HDL simulator and The Synplify family of synthesis tools) to compile, simulate and/or synthesize generated HDL code.

The design flow for Filter Design HDL Coder Toolbox is shown in Figure 3.7 and introduced below:

- Design of the filter;
- Quantification of the filter before HDL code generation to test the effects of different setting with a goal of optimizing the quantized filter's performance and accuracy;
- Customization of the HDL properties: Filter Design HDL Coder Toolbox offers various ways to optimize the generated HDL files such as optimizing coefficient multipliers, optimizing the final summation method (FIR filters), choosing serial or distributed architectures (FIR filters), and use of pipeline registers. We summarized these options below:

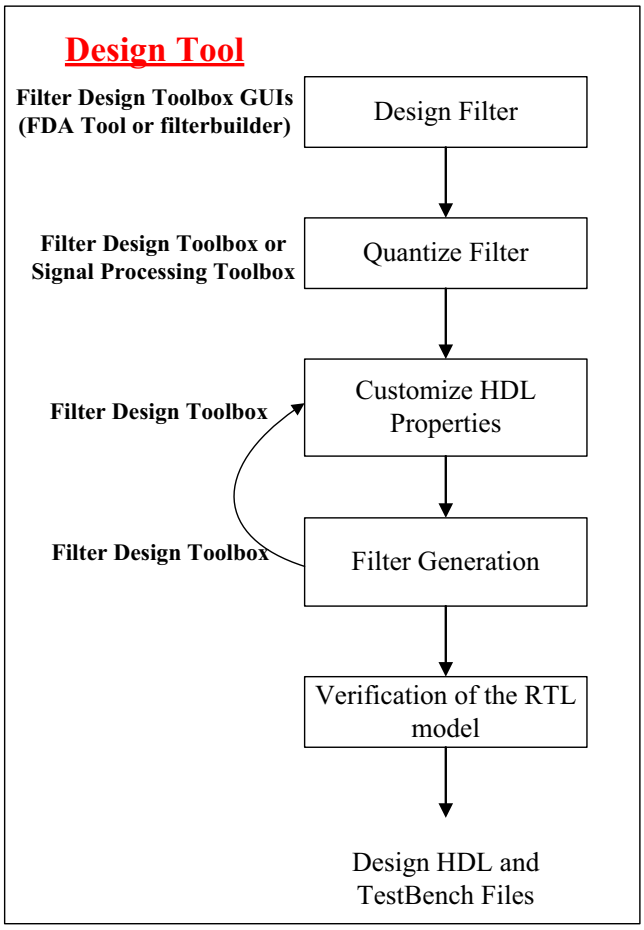


Figure 3.7: **Design Flow of Filter Design HDL Coder Toolbox.**

– **Optimizing coefficient multipliers**: User can replace multiplier operations with additions of partial products produced by canonical signed digit (CSD) or factored CSD techniques which results in a decrease in the area in terms of slices and increase in the clock speed by employing dedicated multipliers (DSP48s). (This option is not available for multirate filters.)

– **Final summation method (FIR filters):** Final summation technique is linear adder summation by default. However, a user can use tree or pipeline final summation instead. Tree summation architecture performs pair-wise addition on successive products and executes in

parallel. Pipeline summation architecture employs a stage of pipeline registers after processing each level of the tree.

- **Choosing serial or distributed architectures (FIR filters):**
  - *Fully Parallel:* This is the default option and results in the largest area since it employs a dedicated multiplier and adder for each filter tap and all taps execute in parallel.
  - *Fully Serial:* This architecture uses a multiply/accumulate operation once for each tap and results in the smallest architecture at the cost of some speed loss and higher power consumption.
  - *Partly Serial:* Partly serial option provides a user defined architecture that lies between fully parallel and fully serial. User defines the number of partitions and the number of the taps for each partition where taps within each partition execute serially and the partitions execute in parallel with respect to one another. As the last step, the outputs of the partitions are summed at the final output.
  - *Cascade Serial:* This option applies a small modification to the partly serial implementation option that the accumulated output of each partition is cascaded to the accumulator of the previous partition which is called accumulator reuse. Since there is no need for a final adder, the resulting area will be low, however requires an additional clock cycle to complete the final summation to the output.
  - *Distributed Arithmetic:* This option lets user to implement sum-of-products computations without the use of multipliers which provides high computational efficiency by distributing multiply and accumulate operations across shifters, lookup tables (LUTs) and adders.
- **Use of pipeline registers:** This option optimizes the maximum clock frequency by the usage of pipeline registers at the cost of an increase in latency and area.

We would like to note that Filter Design HDL Coder Toolbox provides a clean HDL coding style, lets user to specify the architecture as serial (fully, partly or cascade) or distributed arithmetic, provides scripts for third-party simulation and synthesis tools and creates a MATLAB (*.m*) file that captures all non-default settings for HDL and testbench generation. However the support for complex coefficients and inputs are limited to some filter types and structures.

To compare Filter Design HDL Coder Toolbox with our tool (Xquasher), we implemented various FIR filters using different design options provided by Filter Design HDL Coder Toolbox such as architectures that are fully parallel, fully serial, partly serial, cascade serial and distributed arithmetic and that uses pipeline registers with different taps: 20, 30 and 40. The detailed specifications of FIR filters that are specified in *design filter step* are presented in Table 3.1. We introduce our tool, Xquasher, in the following subsection.

Table 3.1: Our Specifications for Filter Design HDL Coder Toolbox Generated Filters.

Option	**Value**
Filter Response	Lowpass
Impulse Response Response	FIR
Filter Type	Single Rate
Design Method	Equiripple
Structure	Direct-form FIR
Arithmetic	Fixed-Point (Signed, wordlength = 16, fraction length = 15)
FIR Adder Style	Linear
Coefficient Source	internal
Wpass	1
Wstop	1
Stopband Shape	Flat
Stopband Decay	0
Fpass	.45
Fstop	.55

3.5.2 Xquasher

Digital signal processing applications often require the computation of linear systems. These computations can be considerably expensive and require optimizations for lower power consumption, higher throughput, and faster response time. Unfortunately, system designers do not have the necessary tools to take advantage of the wide flexibility in ways to evaluate these expressions. Therefore, we address the problem of efficiently computing a set of linear systems through a tool, Xquasher, that is developed by us to enable elimination of large common subexpressions (CSs) from expressions with an arbitrary number of terms [179]. Xquasher provides a methodology for efficient computation of both single and multiple linear expressions. We also introduce the concept of power set encoding which provides an effective optimization method and helps us to achieve significant improvement over previously published work.

Xquasher can effectively find and eliminate arbitrary number of common subexpression terms. Our tool provides enormous area reductions at the cost of increasing the critical path; 22% reduction in area with the cost of 40% increase in delay. However, we can easily add registers to our design which yields 15% decrease in area and a minor increase in delay (3%) compared to registered version of the non-optimized design that uses registers. By using registers at the end of each adder tree to save the results by replacing all expressions with registers instead of wires, we compromised partial area saving to gain less delay. The reason for the increase in the delay is that extensive CSE will significantly reduce the area but also increase the critical path. Since we have a significant area reduction, and linear systems are typically throughput driven, we can add register to the adder tree. This will slightly increase the area, but effectively eliminates the huge increase in delay that we occur from CSE.

Xquasher takes a matrix of coefficients as input and then uses our methodology to generate a new set of expressions. Xquasher also generates an HDL code based on these expressions. First we illustrate the cost of the computation with a three-dimensional space in Figure 3.8 and define some basic terms: *dot*, *line*, *page* and *space* that are used in our methodology for ease of understanding.

- **Canonical Signed digits (CSD)** is radix-2 signed digit representation with digit set of -1,0,1 where there is no adjacent non-zero digit in the representation [132].
- **Bit-Magnitude (BM)** is an integer with the magnitude of $\pm 2^{\mathbb{R}}$. It can be described as a CSD number with exactly one non-zero digit.
- **Dot** is an appearance of an input variable in our expression with a relative magnitude of BM. In hardware, this corresponds to rewiring an input to specific point in the circuit with possible some shifts to the right.
- **Line** is summation of multiple copies of one variable multiplied by a vector of BMs.
- **Page** is formed by a summation of a set of lines.
- **Space** can be created by union of a set of pages.

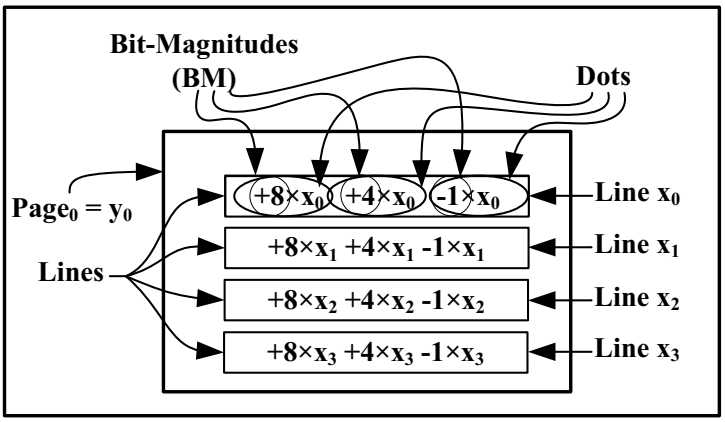


Figure 3.8: **An example of page and its lines, dots and BMs. Dots are shown with ellipses and a circle inside each dot demonstrates the BM part.**

The best hand coded optimization for many common linear systems requires the elimination of common subexpressions with large set of dots. Previous works are not able to handle such elimination due to the exponential increase in the cost of considering such common subexpressions. Although our methodology is not able

to eliminate all possible large common subexpressions because of the increase in the cost, it is still capable of covering large common subexpressions when common subexpressions' dots are focused inside one or two lines. In this section, we first introduce Power Set Encoding (PSE). This encoding let us work with arbitrarily large common subexpressions. Then we show how it is employed in our tool.

Power Set Encoding

To convert an integer to PSE, we follow this procedure:

- CSD Encode an integer, I, to a CSD minimum hamming weight format.
  For example: I = 98 $\Longrightarrow I_{CSD} = 010\bar{1}00010$
- Rewrite the CSD encoded number as a series of integer addition where each number has exactly one of the non-zero digits of the original number.
  For example:
  $010\bar{1}00010 = 010000000 + 000\bar{1}00000 + 000000010 = 128 - 32 + 2$
- Generate power set from step 2's result set.
  For example:
  $2^{\{128,-32+2\}} = \{\{\emptyset\}, \{2\}, \{-32\}, \{128\}, \{2, -32\}, \{2, 128\}, \{-32, 128\}, \{-2, 32, 128\}\}$
- Ignore the null set and generate a new set from the result of step 3 where each element is the sum of numbers inside the related set.
  For example: {2, -32, 128, -30, 130, 96, 98}
- Rewrite each number from the result of 4th step as $2^k \times p$, where p is an odd integer, and store these numbers in the following format:
  (p, k, number of non-zero digit in CSD format)
  For example:
  {[1, 1, 1], [-1, 5, 1], [1, 7, 1], [-15, 1, 2], [65, 1, 2], [3, 5, 2], [49, 1, 3]}

The reason we separate 2^k factors from all numbers is that this part can be implemented with shifting the result to the right. If p parts of two numbers in PSE

representation are equal, both can be calculated with the same circuit. Value of a PSE is not unique. We can select any $2^k \times p$ in the result set as the value. We call each of value of a PSE number a Power Set Dot (PSD). By using PSE to encode a constant multiplication, unlike previous works [128–131] that work with a set of single dots (where common subexpressions are exactly two or three dots), we perform CSE where common subexpressions are made of a set of sets of dots (PSD).

Xquasher Algorithm

Here, we present the Xquasher algorithm in Figure 3.9 and describe it below. After encoding all input constants to PSE (1), Xquasher searches for a CS that has highest number of relevant dots in all appearance of that CS (3-8). Xquasher limits the CS to two PSDs (4-5) and performs CSE for the chosen CS (9), where it updates the related pages to the CS and appends a new page, representing the eliminated CS (10). These steps continue in a loop until there is no more CSE left to be eliminated (2).

1 *convert all constants to PSE*
2 *While there is a CS*
3 *for any page in the space*
4 *for any PSD1 in any line1 in any page*
5 *for any PSD2 in any line2 after line1 in any page*
6 *find number of occurance of CS = (PSD1, PSD2) in the space*
7 *if* $\left(\begin{matrix}\text{number of acuurance}) \times \\ \text{number of non-zero digit in PSD1} \\ +\text{number of non-zero digit in PSD2}\end{matrix}\right)$ *is max in the space*
8 $best_{CF} = (PSD1, PSD2)$
9 *CSE* $(best_{CF})$
10 *Update PSE valuses*

Figure 3.9: **Xquasher Algorithm.**

In the following subsection, we compare our tool, Xquasher, which is specifically designed for filter design with Filter HDL Toolbox from Mathworks to show that domain-specific design tools provide better results than design tools that are general.

3.5.3 Comparison: Filter HDL Toolbox versus Xquasher

We compare our results from Xquasher with the Filter Design HDL Coder Toolbox from Mathworks and Finite/Infinite Impulse Response Filter Generator from Spiral team in terms of area and throughput. Area and performance results are presented in terms of slices, LUTs, FFs and DSP48s; and throughput respectively. Throughput is calculated by dividing the maximum clock frequency (MHz) by the number of clock cycles to perform FIR filtering. All of the designs are written in Verilog and synthesized using Xilinx ISE 9.2. Resource utilization and design frequency are post place and route values obtained using a Virtex 4 SX35 FPGA. It is important to note that Finite/Infinite Impulse Response Filter Generator [135] which provides optimum results for FIR filter designs cannot provide filter designs that are larger than 20 taps.

While comparing these these tool, we use different parameterizations options in our analyses that include:

- Different number of filter taps: 20, 30 and 40 (applied both Xquasher and Filter Design HDL Coder Toolbox, Spiral cannot generate filters with taps larger than 20);

- Different architectural design options: fully parallel, fully serial, partly serial, cascade serial and distributed arithmetic (applied to Filter Design HDL Coder Toolbox since Xquasher and Spiral designs use canonical signed digit (CSD) for optimizing coefficient multipliers and generates fully parallel architectures which use distributed arithmetic);

- Different optimization options are chosen for Filter Design HDL Coder Toolbox implementations such as canonical signed digit (CSD) and factored CSD techniques to optimize coefficient multipliers and use of pipeline registers to optimize the maximum clock frequency of the design;

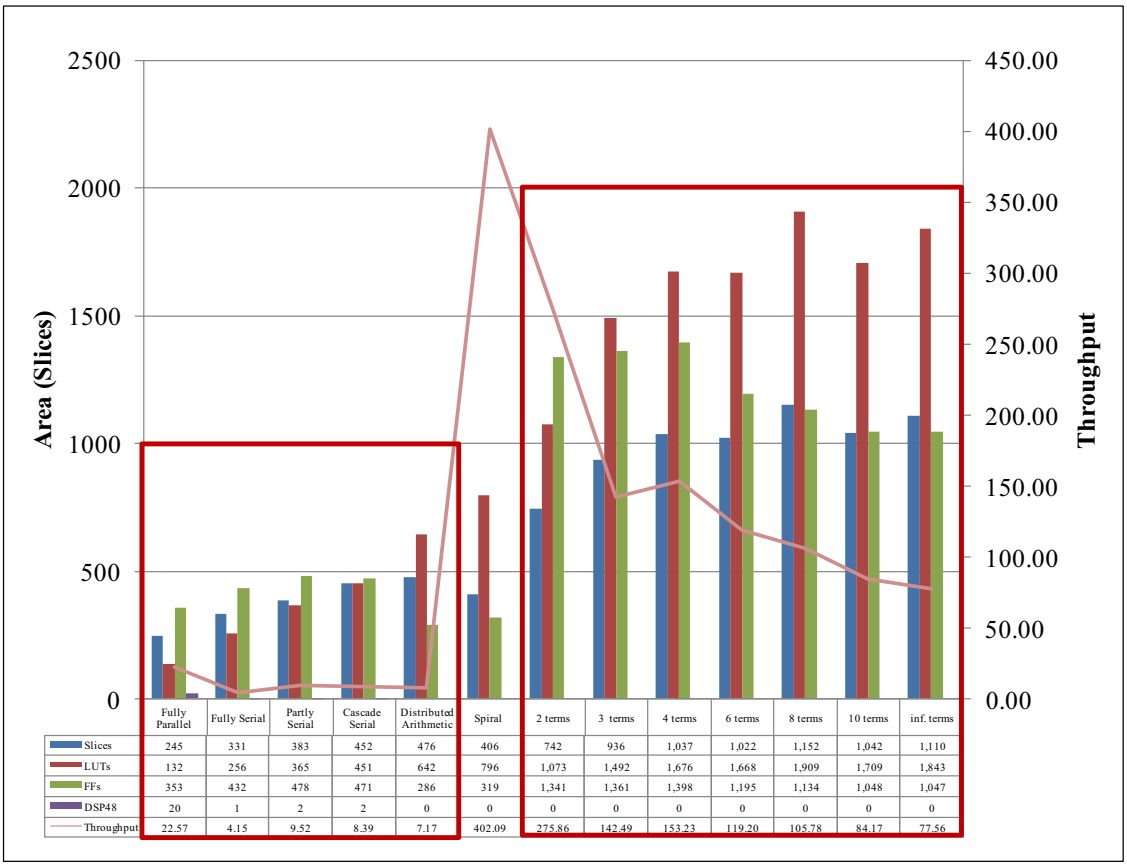
- Different number of term extraction (applied to Xquasher to search for the design space to find most suitable number of terms for extraction).

Figure 3.10 compares Xquasher, Spiral and Filter Design HDL Coder Toolbox architectural results for 20 tap FIR filter designs. We present different architectural designs provided by Filter Design HDL Coder Toolbox: fully parallel, fully serial, partly serial, cascade serial and distributed arithmetic. We also perform different number of term extraction using Xquasher: 2, 3, 4, 6, 8, 10 and infinite to find best results in terms of area and throughput. As can be seen from Figure 3.10, combining the results of these three domain specific design tools provide a very detailed design space for a user.

- *Filter Design HDL Coder Toolbox results*: Fully parallel architecture results in the smallest area in terms of slices and largest, 20, in terms of the number of embedded multipliers usage compared to the other architectural implementations. Employing highly optimized embedded multipliers in its architecture provides the highest throughput results among the Filter Design HDL Coder Toolbox designs. Fully serial architecture results in the smallest architecture in terms of slices and DSP48s with the price of the lowest throughput. Partly serial and cascade serial options bridges the gap between fully parallel and fully serial architectures. Distributed arithmetic architecture uses sum-of-products computations without the use of multipliers which results in no DSP48 usage, however this architecture has the highest area among the architectures that are generated using Filter Design HDL Coder Toolbox.

- *Spiral:* Architectures that are generated using Finite/Infinite Impulse Response Filter Generator from Spiral are fully parallel and employ distributed arithmetic instead of multipliers. Their results are the best in terms of throughput compared to the other design tools.

- *Xquasher:* Xquasher creates architectures that are fully parallel and uses distributed arithmetic like Spiral generated architectures. Xquasher performs extraction with variable number of terms to search for design methods that are the best in terms of both area and throughput. As can be seen from Figure 3.10, 2 terms extraction provides the best results compared to the other term extraction options.

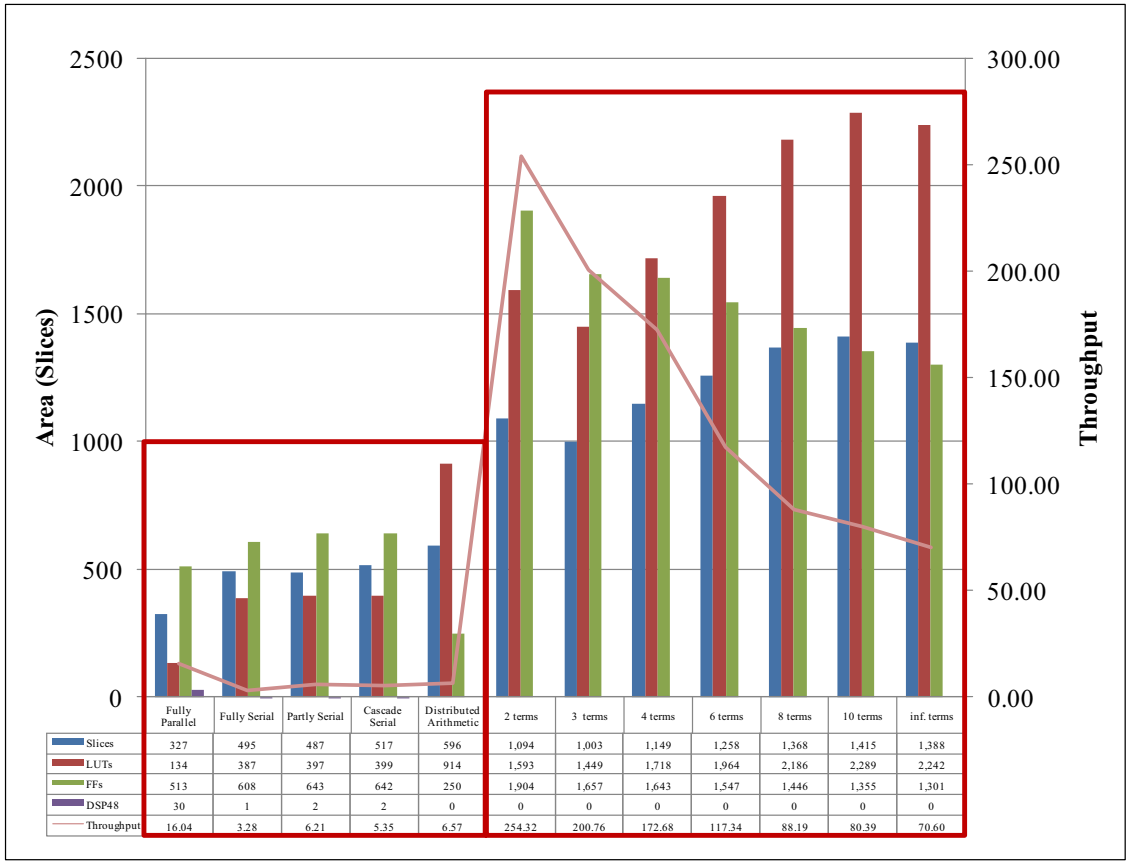


	Fully Parallel	Fully Serial	Partly Serial	Cascade Serial	Distributed Arithmetic	Spiral	2 terms	3 terms	4 terms	6 terms	8 terms	10 terms	inf. terms
Slices	245	331	383	452	476	406	742	936	1,037	1,022	1,152	1,042	1,110
LUTs	132	256	365	451	642	796	1,073	1,492	1,676	1,668	1,909	1,709	1,843
FFs	353	432	478	471	286	319	1,341	1,361	1,398	1,195	1,134	1,048	1,047
DSP48	20	1	2	2	0	0	0	0	0	0	0	0	0
Throughput	22.57	4.15	9.52	8.39	7.17	402.09	275.86	142.49	153.23	119.20	105.78	84.17	77.56

Figure 3.10: **Comparison of Xquasher, Spiral and Filter Design HDL Coder Toolbox architectural results for 20 tap FIR filter designs. Different architectures designed using Design HDL Coder Toolbox include fully parallel, fully serial, partly serial, cascade serial and distributed arithmetic. Xquasher performs different number of term extraction: 2, 3, 4, 6, 8, 10 and infinite terms to find most suitable extraction method that provides best results in terms of area and throughput.**

Figure 3.11 compares Xquasher and Filter Design HDL Coder Toolbox architectural results for 30 tap FIR designs. We present different architectural designs provided by Filter Design HDL Coder Toolbox: fully parallel, fully serial, partly serial, cascade serial and distributed arithmetic. We also perform different number of term extraction using Xquasher: 2, 3, 4, 6, 8, 10 and infinite to find best results in terms of area and throughput. Figure 3.11 does not include Spiral results since this tool cannot generate architectures for filters that have more than 20 taps. The area results of 30 tap FIR filter designs increase while the throughput

results decrease compared to 20 tap FIR filter designs due to its increasing computational complexity. It is important to see the increase in the number of DSP48 with the increasing filter taps. Furthermore, Xquasher provides an architecture using 2 terms extraction that provides the highest throughput. On the other hand, the smallest architecture in terms of area is achieved using 3 terms extraction.



	Fully Parallel	Fully Serial	Partly Serial	Cascade Serial	Distributed Arithmetic	2 terms	3 terms	4 terms	6 terms	8 terms	10 terms	inf. terms
Slices	327	495	487	517	596	1,094	1,003	1,149	1,258	1,368	1,415	1,388
LUTs	134	387	397	399	914	1,593	1,449	1,718	1,964	2,186	2,289	2,242
FFs	513	608	643	642	250	1,904	1,657	1,643	1,547	1,446	1,355	1,301
DSP48	30	1	2	2	0	0	0	0	0	0	0	0
Throughput	16.04	3.28	6.21	5.35	6.57	254.32	200.76	172.68	117.34	88.19	80.39	70.60

Figure 3.11: **Comparison of Xquasher, Spiral and Filter Design HDL Coder Toolbox architectural results for 30 tap FIR filter designs. Different architectures designed using Design HDL Coder Toolbox include fully parallel, fully serial, partly serial, cascade serial and distributed arithmetic. Xquasher performs different number of term extraction: 2, 3, 4, 6, 8, 10 and infinite terms to find most suitable extraction method that provides best results in terms of area and throughput. Spiral cannot generate filters that use more than 20 taps, therefore its results are not included.**

Figure 3.12 compares Xquasher and Filter Design HDL Coder Toolbox architectural results for 40 tap FIR filter designs. We present different architectural

designs provided by Filter Design HDL Coder Toolbox: fully parallel, fully serial, partly serial, cascade serial and distributed arithmetic. We also perform different number of term extraction using Xquasher: 2, 3, 4, 6, 8, 10 and infinite to find the best results in terms of area and throughput. Figure 3.12 does not include Spiral results since this tool cannot generate architectures for filters that have more than 20 taps. The area results of 40 tap FIR designs increase while the throughput results decrease compared to 20 and 30 tap FIR filter designs. It is important to see the increase in number of DSP48 with the increasing filter taps: 20 and 40 DSP48s for 20 tap and 40 tap FIR filters respectively. Furthermore, the best architecture that is provided by Xquasher uses 2 terms extraction that results in the smallest area with the highest throughput.

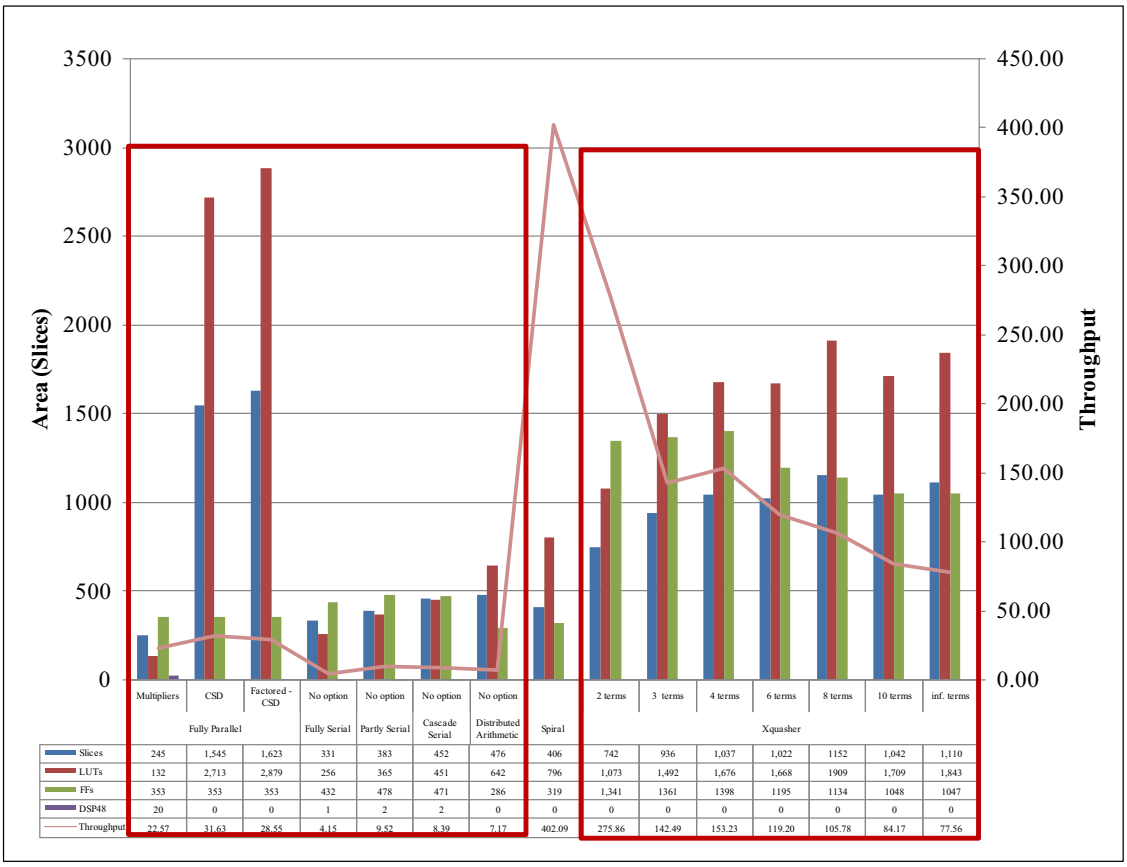
Figure 3.13 compares Xquasher, Spiral and Filter Design HDL Coder Toolbox architectural results for 20 tap FIR designs where fully parallel designs from Filter Design HDL Coder Toolbox employ different methods to optimize coefficient multipliers. Canonical signed digit and factored CSD both replace embedded multipliers with additions of partial products that result in an increase in throughput with the price of an increase in the area as well. We also present different architectural designs provided by Filter Design HDL Coder Toolbox: fully parallel, fully serial, partly serial, cascade serial and distributed arithmetic where only fully parallel architecture has an option to optimize coefficient multipliers. We also present different number of term extraction using Xquasher: 2, 3, 4, 6, 8, 10 and infinite to find best results in terms of area and throughput. We see that replacing multipliers to increase the throughput of the design results in a larger area compared to both Spiral and Xquasher area results. Optimizing the coefficient multipliers increases the throughput of Filter Design HDL Coder Toolbox results, however these results are far worse than both Xquasher and Spirals' throughput results.

Figure 3.14 compares Xquasher, Spiral and Filter Design HDL Coder Toolbox architectural results for 20 tap FIR designs where fully parallel, partly serial and distributed arithmetic designs from Filter Design HDL Coder Toolbox have an option to use of pipeline registers. Use of pipeline registers option optimizes the maximum clock frequency by the usage of pipeline registers at the cost of an

	Fully Parallel	Fully Serial	Partly Serial	Cascade Serial	Distributed Arithmetic	2 terms	3 terms	4 terms	6 terms	8 terms	10 terms
Slices	407	613	713	755	738	1,332	1,489	1,517	1,770	1,839	1,739
LUTs	138	482	699	725	1,153	1,848	2,200	2,292	2,797	2,932	2,731
FFs	674	761	805	801	298	2,389	2,404	2,247	2,112	1,946	1,833
DSP48	40	1	2	2	0	0	0	0	0	0	0
Throughput	12.19	2.08	4.42	3.54	5.47	268.10	184.37	125.06	122.80	93.60	70.35

Figure 3.12: **Comparison of Xquasher and Filter Design HDL Coder Toolbox architectural results for 20 tap FIR designs. Different architectures designed using Design HDL Coder Toolbox include fully parallel, fully serial, partly serial, cascade serial and distributed arithmetic. Xquasher performs different number of term extraction: 2, 3, 4, 6, 8, 10 and infinite terms to find most suitable extraction method that provides best results in terms of area and throughput. Spiral cannot generate filters that use more than 40 taps, therefore its results are not included.**

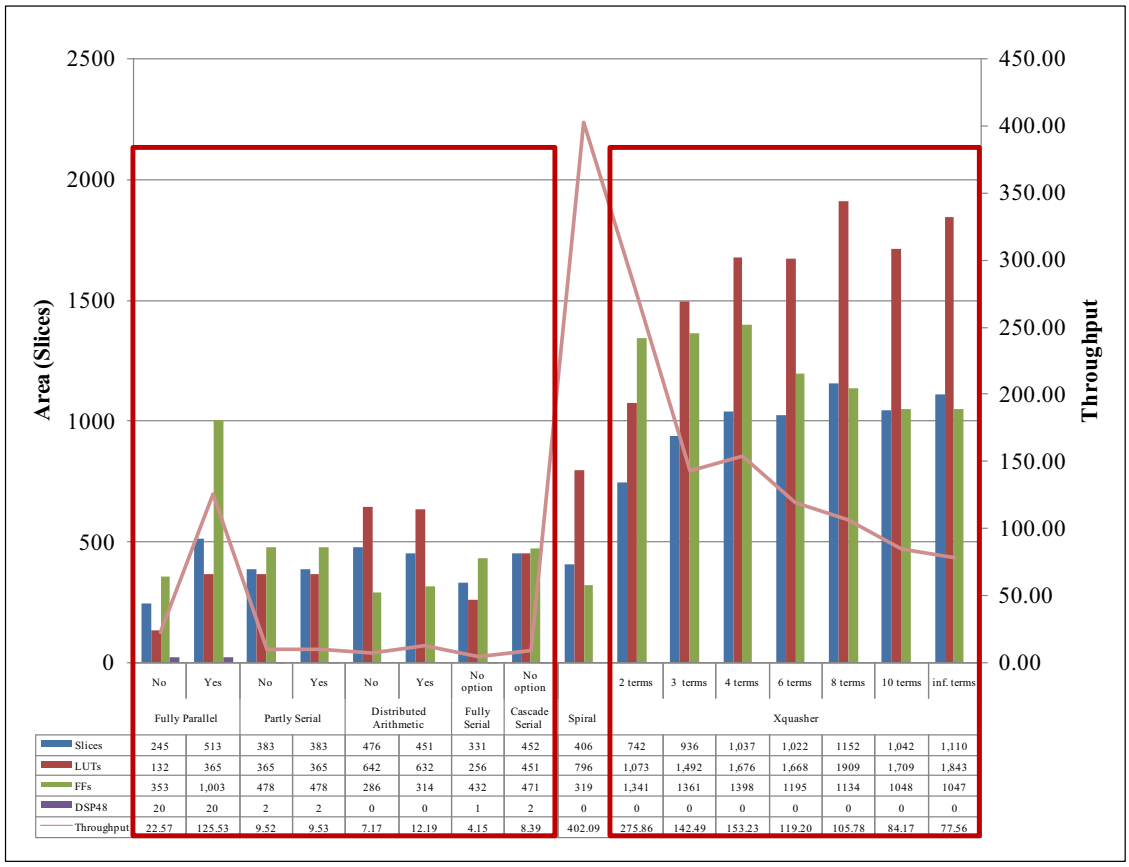
increase in latency and area. There is a tradeoff when employing pipeline registers since there is a positive effect on maximum clock frequency, on the other hand a negative effect on latency. Filter Design HDL Coder Toolbox provides this option for fully parallel, fully serial and distributed arithmetic architectures. We also present different number of term extraction using Xquasher: 2, 3, 4, 6, 8, 10 and infinite to find best results in terms of area and throughput. We see that employing pipeline registers increases throughput of the designs in fully parallel and



	Multipliers	CSD	Factored - CSD	No option	No option	No option	No option		2 terms	3 terms	4 terms	6 terms	8 terms	10 terms	inf. terms
	Fully Parallel			Fully Serial	Partly Serial	Cascade Serial	Distributed Arithmetic	Spiral	Xquasher						
Slices	245	1,545	1,623	331	383	452	476	406	742	936	1,037	1,022	1152	1,042	1,110
LUTs	132	2,713	2,879	256	365	451	642	796	1,073	1,492	1,676	1,668	1909	1,709	1,843
FFs	353	353	353	432	478	471	286	319	1,341	1361	1398	1195	1134	1048	1047
DSP48	20	0	0	1	2	2	0	0	0	0	0	0	0	0	0
Throughput	22.57	31.63	28.55	4.15	9.52	8.39	7.17	402.09	275.86	142.49	153.23	119.20	105.78	84.17	77.56

Figure 3.13: **Comparison of Xquasher, Spiral and Filter Design HDL Coder Toolbox architectural results for 20 tap FIR designs that includes different coefficient multiplier optimization methods for Filter Design HDL Coder Toolbox. Different architectures designed using Design HDL Coder Toolbox include fully parallel (with Multipliers, CSD and Factored CSD), fully serial, partly serial, cascade serial and distributed arithmetic. Xquasher performs different number of term extraction: 2, 3, 4, 6, 8, 10 and infinite terms to find most suitable extraction method that provides best results in terms of area and throughput.**

distributed arithmetic architectures with an area which is still smaller than both Spiral and Xquasher area results. However throughput results of Filter Design HDL Coder Toolbox architectures with pipeline registers are not even close to the throughput values provided by Xquasher and Spiral tools.



	Fully Parallel		Partly Serial		Distributed Arithmetic		Fully Serial	Cascade Serial	Spiral	Xquasher						
	No	Yes	No	Yes	No	Yes	No option	No option		2 terms	3 terms	4 terms	6 terms	8 terms	10 terms	inf. terms
Slices	245	513	383	383	476	451	331	452	406	742	936	1,037	1,022	1152	1,042	1,110
LUTs	132	365	365	365	642	632	256	451	796	1,073	1,492	1,676	1,668	1909	1,709	1,843
FFs	353	1,003	478	478	286	314	432	471	319	1,341	1361	1398	1195	1134	1048	1047
DSP48	20	20	2	2	0	0	1	2	0	0	0	0	0	0	0	0
Throughput	22.57	125.53	9.52	9.53	7.17	12.19	4.15	8.39	402.09	275.86	142.49	153.23	119.20	105.78	84.17	77.56

Figure 3.14: **Comparison of Xquasher, Spiral and Filter Design HDL Coder Toolbox architectural results for 20 tap FIR filter designs that includes an option for use of pipeline registers for Filter Design HDL Coder Toolbox. Different architectures designed using Design HDL Coder Toolbox include fully parallel (with and without pipeline registers), fully serial, partly serial (with and without pipeline registers), cascade serial and distributed arithmetic (with and without pipeline registers). Xquasher performs different number of term extraction: 2, 3, 4, 6, 8, 10 and infinite terms to find most suitable extraction method that provides best results in terms of area and throughput.**

3.6 Roadmap for the Future Single/Many-Core Platform Generator Tool

In this chapter, we introduced several high level design tools either based on MATLAB or C programming languages. MATLAB based design tools include **model based tools:** System Generator for DSP, Synplify DSP, and Simulink HDL

Coder and **MATLAB (.m) code based tool:** AccelDSP. C-code based design tools include System-C, Catapult-C, Impulse-C, Mitrion-C, DIME-C, Handel-C, Carte, SA-C, Streams-C and Napa-C.

Each tool has its own inherent advantages and disadvantages. Such as System Generator for DSP from Xilinx, Simulink HDL Coder from Mathworks and Synplify DSP from Synplicity that are designed to translate model based designs from Simulink environment to HDL code to target various different platforms. With these design tools, a user does not need to be experienced in FPGAs or Register Transfer Level (RTL) design methodologies since they provide a user friendly visual environment with specific library units. **Significant problems with these tools are the generation of control units (synchronization between resources) and the lack of built-in block support that are ready to use**.

The solution to the synchronization problem is to use AccelDSP from Xilinx which provides a MATLAB code, *.m*, to HDL conversion automatically. Therefore a user can design controller units and/or undefined blocks using AccelDSP easily. **Even though, this tool generates poor results while generating architectures, it can still be used as a complement to the model-based design tools.**

Recommendation: For MATLAB based design tools, the best solution is to combine **System Generator for DSP** in model based design environment with **AccelDSP** supporting for control units as well as blocks that are not supported by the built-in library and **Stateflow** for finite state machine optimization that can also be used as control logic. If a user plans to target an FPGA different than Xilinx series, **System Generator for DSP** should be replaced with **Synplify DSP**.

There are also a large number of C-based design tools that are introduced through this chapter in detail. C-based high level design tools are divided into 5 different families including Open Standard: System-C; tools producing generic HDL that can target multiple platforms: Catapult-C, Impulse-C and Mitrion-C; tools producing generic HDL that are optimized for manufacturer's hardware:

DIME-C, Handel-C; tools that target a specific platform and/or configuration: Carte, SA-C, Streams-C; tool targeting RISC/FPGA hybrid architectures: Napa-C [136].

Again, all of these tools come with different advantages and disadvantages. C-based high level design tools express parallelism through variations in C (pseudo-C) or compiler or both. Ideally, it is best to use pure *ANSI-C* without any variation in C and exploit parallelism through compiler that ports C code into hardware; therefore a user does not need to learn a new language. On the other hand, if parallelism is expressed through variations in C code (pseudo-C code), it is important to note that there is no standard language and each tool uses its own language constructs requiring a user to learn that specific language to be able to use the high level design tool. A user is required to know and perform specific programming styles to achieve maximum optimization in terms of area, throughput, power consumption etc. Most of these tools such as Behaviour Synthesizer Cyber [62], Catapult-C [153] and Cynthesizer [167] perform simultaneous architecture generation for the data path and the controller. To the best of our knowledge, these design tools are applicable to a small code size.

All of these C-based design tool design the datapath on the fly; and their architecture generation and its results depend on the controller generation where most of them employ finite state machines (FSM) as controllers. Since FSM complexity increases dramatically with the increasing application complexity, [62,167] provide different optimization procedures such as FSM partitioning and hierarchical FSM generation.

Recommendation: Among these different C-based design tools, **Catapult-C** is the most powerful tool for C/C++ code to HDL conversion so far. The other established design tools are **Impulse-C** and **Handel-C**.

However, the majority of these tools attempt to be everything to everyone, and often fail to do anything for anyone. In subsection 3.5, we present a case study comparing two different domain-specific design tools: Filter Design HDL Coder Toolbox and Xquasher, and show that their results in terms of area, latency etc. are better than general purpose design tools. Thus, we believe that it is important

to take a more focused approach, therefore we should design a tool specifically targeting matrix computation algorithms rather than a tool for everything.

We conclude this chapter by proposing the following properties that our design tool should have:

- **Domain-specific** design tool specifically targeting a set of algorithms like matrix computations;
- **Abstracting intricate details** of hardware generation from the user by providing a user friendly environment. Therefore a user does not need to be experienced in FPGA design and RTL design methodologies by enabling a "push a button" transition from specification to implementation;
- **Time and cycle accurate design methodology** that leads to a **correct-by-construction** architecture exploiting large amount of parallelism if desired;
- **Capability of targeting multiple platforms**, also providing an ability for further optimization to a specific platform;
- **Capability of specifying the resources in generated architecture** such as number of resources, number of bits used and optimization level (a general purpose or application specific architecture);
- **Automatic design space exploration, what if analyses**, for being able to eliminate results that are not pareto-optimum in terms of desired objective;
- **Capability of distributing control units** in hierarchical finite state machines as well as logic units;
- **Automatic generation of synchronization between individual hardware units**, control units;
- **Built-in libraries** to provide synthesizable and optimized MATLAB equivalant functions without requiring any modification from the user side;

- **Support for various data representation methods** such as fixed-point arithmetic, floating point arithmetic etc.;
- **Error analysis and its visualization** is essential to achieve optimum results for architectures that employ fixed point arithmetic by determination of bit-width and the place of the binary point providing similar accuracy to floating point implementation;
- **Visual data flow graph and its visualization** is essential, therefore a user can evaluate the quality of the MATLAB *.m* code provided to the tool as input. This visualization should also help user to achieve maximum optimization in terms of desired objective such as low area usage and high throughput by showing instruction scheduling, loop unrooling/merging, resource sharing and binding options;
- **Generation of a design block** to be used in model based environment along with System Generator for DSP, Simulink HDL Coder and Synplify DSP built-in blocks. Since most of the generated single/multi-core architectures can be a part of a larger design in the future, it is important to provide a block for the user.

The text of Chapter 3.5.2 is in part a reprint of the material as it appears in the proceedings of the Design Automation Conference. The dissertation author was a co-primary researcher and author (with Arash Arfaee) and the other co-authors listed on this publication [179] directed and supervised the research which forms the basis for Chapter 3.5.2.

Chapter 4

Matrix Computations: Matrix Multiplication, Matrix Decomposition and Matrix Inversion

Matrix computations lie at the heart of many scientific computational algorithms such as signal processing, computer vision and financial computations. In order to determine the effectiveness of our tool, we consider some of the most important matrix computation tasks: matrix multiplication, decomposition and inversion and implement various architectures, single core and multi-core, using our tool, GUSTO, and present their results in the following chapters. These computations are building blocks for larger matrix computation algorithms. Therefore, we believe that effective implementation of these computations provide us a clue for a better design tool and a way to evaluate GUSTO. This chapter provides an overview for these computations including their characteristics, different solution methods and algorithms. It is important to note that there are several different ways to implement these algorithms which may result in different architectural designs due to their different data dependencies, memory accesses and parallelism levels.

The following sections are organized as follows: Section 4.1 introduces the building blocks of matrix computations: addition, scalar-matrix multiplication and matrix-matrix multiplication and provides algorithms for matrix-vector multiplication and matrix-matrix multiplication. Section 4.1 also details the matrix notations with different examples. Section 4.2 looks into different decomposition methods: QR, LU and Cholesky, presents their algorithms, resulting matrices and solution methods. Section 4.3 focuses on different matrix inversion methods: analytic and decomposition based approaches. Decomposition based matrix inversion algorithms employ QR, LU or Cholesky decompositions. We also provide a detailed presentation for matrix decomposition algorithms in the Appendix at the end of this thesis.

4.1 Building Blocks of Matrix Computations

Matrix multiplication is the foremost important computation since it has been used frequently in matrix computation algorithms. The mathematical solution to the matrix multiplication problem is relatively easy, however it is rich in computational point of view which affects the efficiency of results. The different ways to compute matrix multiplication stems from the fact that it is possible to exploit the given matrix structures. Such that if the given matrix is symmetric, it can be stored in half the space as a general matrix or if it is a matrix-vector multiplication, there are ways to decrease the computation time due to the fact that vector matrix can be seen as a matrix with some zero entries. Before introducing the algorithms for matrix multiplication algorithm, we introduce some concepts and definitions that makes this as well as the following sections easier to understand.

Matrix computations can be seen as the combination of many linear algebraic operations in hierarchy. Some examples are [13]:

- Dot products are scalar operations: addition and multiplication;
- Matrix - vector multiplication is series of dot products;

- Matrix - matrix multiplication is a collection of matrix - vector products.

The vector space of all m-by-n real matrices $\mathbb{R}^{m\times n}$ is shown as:

$$A \in \mathbb{R}^{m\times n} \Leftrightarrow A = (A_{ij}) = \begin{bmatrix} A_{11} & \dots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{m1} & \dots & A_{mn} \end{bmatrix} a_{ij} \in \mathbb{R}$$

A matrix is a rectangular array of numbers. If the array has m rows and n columns then it is said to be an $m \times n$ matrix. The element in the *ith* row and *jth* column of the matrix A is denoted by A_{ij}. The building blocks for the matrix computations are:

- Addition($\mathbb{R}^{m\times n} \times \mathbb{R}^{m\times n} \rightarrow \mathbb{R}^{m\times n}$)

$$C = A + B \Rightarrow c_{ij} = a_{ij} + b_{ij} \tag{4.1}$$

- Scalar-matrix multiplication ($\mathbb{R} \times \mathbb{R}^{m\times n} \rightarrow \mathbb{R}^{m\times n}$)

$$C = \alpha A \Rightarrow c_{ij} = \alpha a_{ij} \tag{4.2}$$

- Matrix-matrix multiplication ($\mathbb{R}^{m\times r} \times \mathbb{R}^{r\times n} \rightarrow \mathbb{R}^{m\times n}$)

$$C = AB \Rightarrow c_{ij} = \sum_{k=1}^{r} a_{ik} b_{kj} \tag{4.3}$$

We present Matrix-Matrix multiplication with two different parts: Matrix-Vector Multiplication and Matrix-Matrix Multiplication.

Matrix-Vector Multiplication: Two different Algorithms, 1 and 2, for Matrix-Vector multiplications, Row and Column versions respectively, are shown below. Both algorithms uses A, x and z as input matrix, input vector and resulting matrix respectively. **(2)** in both algorithms determines the matrix dimensions, m and n as number of rows and columns respectively. Algorithm 1 accesses A row by row, and multiplies each row of the input matrix, A, with the vector elements and acculumates the result number of column

Algorithm 1 Matrix-Vector Multiplication: Row version

Require: $A \in \mathbb{R}^{m \times n}$ and $x \in \mathbb{R}^n$

```
1: Function z = mat_vec_row(A, x)
2: m = rows(A); n = cols(A)
3: z(1 : m) = 0
4: for i = 1 : m do
5:    for j = 1 : n do
6:       z(i) = z(i) + A(i, j) x(j)
7:    end for
8: end for
9: end mat_vec_row
```

times, n **(6)**. Algorithm 2 accesses A column by column, and multiplies each element in a column of the input matrix, A, with the same vector elements and updates the resulting matrix row elements. Acculumation of these multiplications result in the matrix multiplication **(6)**.

Algorithm 2 Matrix-Vector Multiplication: Column version

Require: $A \in \mathbb{R}^{m \times n}$ and $x \in \mathbb{R}^n$

```
1: Function z = mat_vec_col(A, x)
2: m = rows(A); n = cols(A)
3: z(1 : m) = 0
4: for j = 1 : n do
5:    for i = 1 : m do
6:       z(i) = z(i) + x(j) A(i, j)
7:    end for
8: end for
9: end mat_vec_col
```

Matrix-Matrix Multiplication: The most widely used matrix-matrix multiplication algorithms are dot product matrix multiply, gaxpy matrix multiply

and outer product matrix multiply which are presented in detail in [13]. We consider dot product matrix multiply as our algorithm. Assume that one wants to compute $C = AB$ where $A \in \mathbb{R}^{m \times r}$ and $B \in \mathbb{R}^{r \times n}$. This computation is shown with as example below for a 2×2 matrices:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1 \cdot 5 + 2 \cdot 7 & 1 \cdot 6 + 2 \cdot 8 \\ 3 \cdot 5 + 4 \cdot 7 & 3 \cdot 6 + 4 \cdot 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

The algorithm for dot product matrix multiply is shown in Algorithm 3. The resulting matrix C, is assumed to be an array of dot products that its entries are computed one at a time in left to right, top to bottom order **(7)**.

Algorithm 3 Matrix-Matrix Multiplication: Dot Product

Require: $A \in \mathbb{R}^{m \times r}$ and $x \in \mathbb{R}^{r \times n}$

```
Function C = mat_mat_mul(A, B)
m = rows(A); r = cols(A); n = cols(B)
C(1 : m, 1 : n) = 0
for i = 1 : m do
    for j = 1 : n do
        for k = 1 : r do
            C(i, j) = C(i, j) + A(i, j) B(k, j)
        end for
    end for
end for
end mat_mat_mul
```

Here, we continue with some basic definitions for matrix notations.

Definition 1 - Square matrix is a matrix which has the same number of rows as columns.

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}$$

Definition 2 - Column vector is a matrix with one column and row vector is a matrix with one row.

$$\text{Column Vector: } \begin{bmatrix} A_1 \\ A_2 \\ A_3 \end{bmatrix}$$

$$\text{Row Vector: } \begin{bmatrix} A_1 & A_2 & A_3 \end{bmatrix}$$

Definition 3 - Diagonal of the square matrix is the top-left to bottom-right diagonal. A square matrix is diagonal matrix if all the elements off the diagonal are zero which can be shown as $A_{ij} = 0$ if $i \neq j$.

$$\text{Diagonal Matrix: } \begin{bmatrix} A_{11} & 0 & 0 \\ 0 & A_{22} & 0 \\ 0 & 0 & A_{33} \end{bmatrix}$$

Definition 4 - A square matrix is upper triangular if all elements below the diagonal are zero. A square matrix is lower triangular if all elements above the diagonal are zero.

$$\text{Upper Triangular Matrix: } \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ 0 & A_{22} & A_{23} \\ 0 & 0 & A_{33} \end{bmatrix}$$

$$\text{Lower Triangular Matrix: } \begin{bmatrix} A_{11} & 0 & 0 \\ A_{21} & A_{22} & 0 \\ A_{31} & A_{32} & A_{33} \end{bmatrix}$$

Definition 5 - Transpose of a matrix A is another matrix which is created by writing the rows of A as the columns of transpose matrix or writing the columns of A as the rows of transpose matrix. Transpose of matrix A is written as A^T.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, A^T = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

Definition 6 - A square matrix is symmetric if it equals to its own transpose, $A = A^T$ and its elements are symmetric about the diagonal where $A_{ij} = A_{ji}$ for all i and j.

$$\begin{bmatrix} 1 & 4 & 7 \\ 4 & 3 & 2 \\ 7 & 2 & 5 \end{bmatrix}$$

Definition 7 - An identity matrix is the diagonal square matrix with 1s down its diagonal. Identity matrix is written as I.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Definition 8 - A zero matrix is the matrix which has all its elements zero.

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Definition 9 - Complex conjugate of a complex number is written by changing the sign of the imaginary part. If the complex number is y, the complex conjugate of the complex number can be written as $\bar{y}$.

$$y = a + ib, \bar{y} = a - ib \tag{4.4}$$

Definition 10 - Complex conjugate transpose of a complex matrix is created by taking the transpose of the matrix and then taking the complex conjugate of each entry. If the matrix is represented by A, complex conjugate transpose of matrix A is written as A^*.

$$A = \begin{bmatrix} 1+2i & 3i \\ 4+5i & 6 \end{bmatrix}, A^* = \begin{bmatrix} 1-2i & 4-5i \\ -3i & 6 \end{bmatrix}$$

Definition 11 - A hermitian matrix is a complex matrix and it equals to its own complex conjugate transpose, $A = A^H$.

Definition 12 Matrix inverse of square matrix A is A^{-1} such that $A \times A^{-1} = I$ where I is the identity matrix.

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} A_{11}^{-1} & A_{12}^{-1} \\ A_{21}^{-1} & A_{22}^{-1} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Definition 13 - An orthogonal matrix is a real matrix and the inverse of the matrix equals to its transpose, that is $Q \times Q^T = Q^T \times Q = I$.

Definition 14 - A complex matrix U is a unitary matrix if the inverse of U equals the complex conjugate transpose of U, $U^{-1} = U^H$, that is $U \times U^H = U^H \times U = I$.

Definition 15 - Determinant of a matrix is a function which depends on the matrix size and it is calculated by combining all the elements of the matrix. It is only defined for square matrices and denoted as *det (A)* or $|A|$. For a 2×2 matrix, it can be shown as

$$\left| \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \right| = (A_{11} \times A_{22}) - (A_{12} \times A_{21})$$

Definition 16 - A matrix is singular if its determinant is 0 and as a result it doesn't have a matrix inverse. Nonsingular matrix is the matrix which is not singular.

Definition 17 - In a matrix, if a row or column is a multiple of one another, they are not independent and its determinant is zero. Maximum number of independent rows or equivalently columns specifies the rank of a matrix.

4.2 Matrix Decomposition and its Methods

- **QR:** Given $A \in \mathbb{R}^{m \times n}$ with *rank(A) = n*, QR factorization exists as $A = Q \times R$ where $Q \in \mathbb{R}^{m \times n}$ has orthonormal columns and $R \in \mathbb{R}^{n \times n}$ is upper triangular.

- **LU:** Given $A \in \mathbb{R}^{n \times n}$ with *det(A(1:k, 1:k)) $\neq$ 0* for *k = 1 : n-1*, LU decomposition exists as $A = L \times U$. If LU decomposition exists and the given matrix, A, is nonsingular, then the decomposition is unique and det(A) = $u_{11}...u_{nn}$.

- **Cholesky:** Given a symmetric positive definite matrix, $A \in \mathbb{R}^{n \times n}$, Cholesky decomposition exists as $A = G \times G^T$ where $G \in \mathbb{R}^{n \times n}$ is a unique lower triangular matrix with positive diagonal entries.

where a matrix $A \in \mathbb{R}^{n \times n}$ is *positive definite* if $x^T Ax > 0$ for $x \in \mathbb{R}^n$ and x $\neq$ 0 and if it is *symmetric positive definite matrix* then $A^T = A$. A positive definite matrix is always nonsingular and its determinant is always positive.

Below we describe three known decomposition methods to perform matrix inversion: QR, LU, and Cholesky decomposition methods [13]. Note that Cholesky and LU decompositions work only with positive definite and nonsingular diagonally dominant square matrices, respectively. QR decomposition, on the other hand, is more general and can be applied to any matrix. For square matrices, n denotes the size of the matrix such that $n = 4$ for 4×4 matrices. For rectangular matrices, m and n denote the number of rows and columns in the matrix respectively such that $m = 3$, $n = 4$ for 3×4 matrices.

4.2.1 QR Decomposition

QR decomposition is an elementary operation, which decomposes a matrix into an orthogonal and a triangular matrix. QR decomposition of a matrix A is shown as $A = Q \times R$, where Q is an orthogonal matrix, $Q^T \times Q = Q \times Q^T = I$, $Q^{-1} = Q^T$, and R is an upper triangular matrix (as shown below for 4×4 matrices).

$$Q = \begin{bmatrix} Q_{11} & Q_{12} & Q_{13} & Q_{14} \\ Q_{21} & Q_{22} & Q_{23} & Q_{24} \\ Q_{31} & Q_{32} & Q_{33} & Q_{34} \\ Q_{41} & Q_{42} & Q_{43} & Q_{44} \end{bmatrix}; R = \begin{bmatrix} R_{11} & R_{12} & R_{13} & R_{14} \\ 0 & R_{22} & R_{23} & R_{24} \\ 0 & 0 & R_{33} & R_{34} \\ 0 & 0 & 0 & R_{44} \end{bmatrix}$$

There are three different QR decomposition methods: Gram-Schmidt orthogonalization (Classical or Modified), Givens Rotations (GR) and Householder reflections. Applying slight modifications to the Classical Gram-Schmidt (CGS) algorithm gives the Modified Gram-Schmidt (MGS) algorithm [13].

QRD-MGS is numerically more accurate and stable than QRD-CGS and it is numerically equivalent to the Givens Rotations solution [14–16] (the solution that has been the focus of previously published hardware implementations because of its stability and accuracy). Also, if the input matrix, A, is well-conditioned and non-singular, the resulting matrices, Q and R, satisfy their required matrix characteristics and QRD-MGS is accurate to floating-point machine precision [16]. We therefore present the QRD-MGS algorithm in Algorithm 4 and describe it below. A, Q, R and X are the input, orthogonal, upper triangular and intermediate matrices, respectively. The intermediate matrix is the updated input matrix throughout the solution steps. Matrices with only one index as A_i or X_j represent the columns of the matrix and matrices with two indices like R_{ij} represent the entry at the intersection of ith row with jth column of the matrix where $1 \leq i, j \leq n$.

In Algorithm 4 we show that we start every decomposition by transferring the input, 4×4, matrix columns, A_i, into the memory elements **(3)**. Diagonal entries of the R matrix are the Euclidean norm of the intermediate matrix columns which is shown as **(6)**. The Q matrix columns are calculated by the division of the intermediate matrix columns by the Euclidean norm of the intermediate matrix column, which is the diagonal element of R **(7)**. Non-diagonal entries of the R matrix are computed by projecting the Q matrix columns onto the intermediate matrix columns one by one **(9)** such that after the solution of Q_2, it is projected onto X_3 and X_4 to compute R_{23} and R_{24}. Lastly, the intermediate matrix columns

Algorithm 4 QRD-MGS Algorithm

1: *Function* $(Q, R) = QRD\_MGS(A)$
2: **for** $i = 1 : n$ **do**
3: $\quad X_i = A_i$
4: **end for**
5: **for** $i = 1 : n$ **do**
6: $\quad R_{ii} = \|X_i\|$
7: $\quad Q_i = \frac{X_i}{R_{ii}}$
8: $\quad$ **for** $j = i + 1 : n$ **do**
9: $\quad\quad R_{ij} =< Q_i, X_j >$
10: $\quad\quad X_j = X_j - R_{ij}Q_i$
11: $\quad$ **end for**
12: **end for**
13: *end QRD_MGS*

are updated by **(10)**.

4.2.2 LU Decomposition

If A is a square matrix and its leading principal submatrices are all non-singular, matrix A can be decomposed into unique lower triangular and upper triangular matrices. LU decomposition of a matrix A is shown as $A = L \times U$, where L and U are the lower and upper triangular matrices respectively (as shown below for 4×4 matrices).

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ L_{21} & 1 & 0 & 0 \\ L_{31} & L_{32} & 0 & 0 \\ L_{41} & L_{42} & L_{43} & 0 \end{bmatrix}; U = \begin{bmatrix} U_{11} & U_{12} & U_{13} & U_{14} \\ 0 & U_{22} & U_{23} & U_{24} \\ 0 & 0 & U_{33} & U_{34} \\ 0 & 0 & 0 & U_{44} \end{bmatrix}$$

The LU algorithm is shown in Algorithm 5. It writes lower and upper triangular matrices onto the A matrix entries. Then it updates the values of the A

Algorithm 5 LU Algorithm

1: *Function* $(L, U) = LU(A)$
2: **for** $j = 1 : n$ **do**
3: **for** $k = 1 : j - 1$ **do**
4: **for** $i = k + 1 : j - 1$ **do**
5: $A_{ij} = A_{ij} - A_{ik} \times A_{kj}$
6: **end for**
7: **end for**
8: **for** $k = 1 : j - 1$ **do**
9: **for** $i = j : n$ **do**
10: $A_{ij} = A_{ij} - A_{ik} \times A_{kj}$
11: **end for**
12: **end for**
13: **for** $k = j + 1 : n$ **do**
14: $A_{kj} = \frac{A_{kj}}{A_{jj}}$
15: **end for**
16: **end for**
17: *end LU*

matrix column by column (**(5)** and **(10)**). The final values are computed by the division of each column entry by the diagonal entry of that column **(14)**.

4.2.3 Cholesky Decomposition

Cholesky decomposition is another elementary operation, which decomposes a symmetric positive definite matrix into a unique lower triangular matrix with positive diagonal entries. Cholesky decomposition of a matrix A is shown as $A = G \times G^T$, where G is a unique lower triangular matrix, Cholesky triangle, and G^T is the transpose of this lower triangular matrix (as shown below for 4×4 matrices).

$$G = \begin{bmatrix} G_{11} & 0 & 0 & 0 \\ G_{21} & G_{22} & 0 & 0 \\ G_{31} & G_{32} & G_{33} & 0 \\ G_{41} & G_{42} & G_{43} & G_{44} \end{bmatrix}; G^T = \begin{bmatrix} G_{11} & G_{21} & G_{31} & G_{41} \\ 0 & G_{22} & G_{32} & G_{42} \\ 0 & 0 & G_{33} & G_{43} \\ 0 & 0 & 0 & G_{44} \end{bmatrix}$$

Algorithm 6 Cholesky Algorithm

1: *Function* $(G, G^T) = Cho(A)$
2: **for** $k = 1 : n$ **do**
3: $G_{kk} = \sqrt{A_{kk}}$
4: **for** $i = k + 1 : n$ **do**
5: $G_{ik} = \frac{A_{ik}}{A_{kk}}$
6: **end for**
7: **for** $j = k + 1 : n$ **do**
8: **for** $t = j : n$ **do**
9: $A_{tj} = A_{tj} - G_{tk}G_{jk}$
10: **end for**
11: **end for**
12: **end for**
13: *end Cho*

Algorithm 6 shows the Cholesky decomposition algorithm. We start decomposition by transferring the input matrix, A, into the memory elements. The diagonal entries of lower triangular matrix, G, are the square root of the diagonal entries of the given matrix **(3)**. We calculate the entries below the diagonal entries by dividing the corresponding element of the given matrix by the belonging column diagonal element **(5)**. The algorithm works column by column and after the computation of the first column of the diagonal matrix with the given matrix entries, the elements in the next columns are updated **(9)**. For example after the computation of G_{11} by **(3)**, G_{21}, G_{31}, G_{41} by **(5)**, second column: A_{22}, A_{32}, A_{42}, third column: A_{33}, A_{43}, and fourth column: A_{44} are updated by **(9)**.

4.3 Matrix Inversion and its Methods

Matrix inversion algorithms lie at the heart of most scientific computational tasks. Matrix inversion is frequently used to solve linear systems of equations in many fields such as wireless communication. For example, in wireless communication, MIMO-OFDM systems use matrix inversion in equalization algorithms to remove the effect of the channel on the signal [4–6], minimum mean square error algorithms for pre-coding in spatial multiplexing [7] and detection-estimation algorithms in space-time coding [8]. These systems often use a small number of antennas (2 to 8) which results in small matrices to be decomposed and/or inverted. For example the 802.11n standard [11] specifies a maximum of 4 antennas on the transmit/receive sides and the 802.16 [12] standard specifies a maximum of 16 antennas at a base station and 2 antennas at a remote station.

Explicit matrix inversion of a full matrix is a computationally intensive method and for example it requires the solution of 9 equations (i.e. $A_{11}x_{11} + A_{12}x_{21}A_{13}x_{31} = 1$, $A_{11}x_{12} + A_{12}x_{22} + A_{13}x_{32} = 0$ and $A_{11}x_{13} + A_{12}x_{23} + A_{13}x_{33} = 0$) to determine the inverse for a 3×3 matrix which is shown below.

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \times \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

If the inversion is encountered, one should consider converting this problem into an easy decomposition problem which will result in analytic simplicity and computational convenience. Decomposition methods are generally viewed as the preferred methods for matrix inversion because they scale well for large matrix dimensions while the complexity of the analytic method increases dramatically as the matrix dimensions grow. However, for small matrices, the analytic method, which can exploit a significant amount of parallelism, outperforms the decomposition methods. Also note that Cholesky and LU decompositions work only with positive definite and nonsingular diagonally dominant square matrices, respectively. QR decomposition, on the other hand, is more general and can be applied to any matrix. We further explain these different matrix inversion methods, their characteristics and

algorithms, the resulting matrices, and the solution steps for matrix inversion in the next subsections.

4.3.1 Matrix Inversion of Triangular Matrices

Triangular matrix inversion is used in all of the decomposition based (QR, LU and Cholesky) matrix inversion architectures described above and we use this subsection to describe why this inversion is relatively simple and therefore not a dominant calculation in any of these methods. Primarily, triangular matrix inversion requires fewer calculations compared to full matrix inversion because of its zero entries. The algorithm for triangular matrix inversion is shown in Algorithm 7 and described below.

Algorithm 7 Matrix Inversion of Upper Triangular Matrices

1: *Function* $(R^{-1}) = MI(R)$
2: $R^{-1} = 0$
3: **for** $j = 1 : n$ **do**
4: **for** $i = 1 : j - 1$ **do**
5: **for** $k = 1 : j - 1$ **do**
6: $R_{ij}^{-1} = R_{ij}^{-1} + R_{ik}^{-1} R_{kj}$
7: **end for**
8: **end for**
9: **for** $k = 1 : j - 1$ **do**
10: $R_{kj}^{-1} = -\frac{R_{kj}^{-1}}{R_{jj}}$
11: **end for**
12: $R_{jj}^{-1} = \frac{1}{R_{jj}}$
13: **end for**
14: *end MI*

Upper triangular matrix inversion is performed column by column. Calculating the diagonal entries of the R^{-1} matrix consists of simply dividing 1 by the diagonal entry of the R matrix **(12)** and the rest of the column entries introduce multiplication and addition iteratively **(6)** which is then divided by the diagonal

R matrix entry **(10)**.

4.3.2 QR Decomposition Based Matrix Inversion

The solution for the inversion of matrix A, A^{-1}, using QR decomposition is shown as follows:

$$A^{-1} = R^{-1} \times Q^T \tag{4.5}$$

This solution consists of three different parts: QR decomposition, matrix inversion for the upper triangular matrix and matrix multiplication (Figure 4.1). QR decomposition is the dominant calculation where the next two parts are relatively simple due to the upper triangular structure of R (as described in section A above).

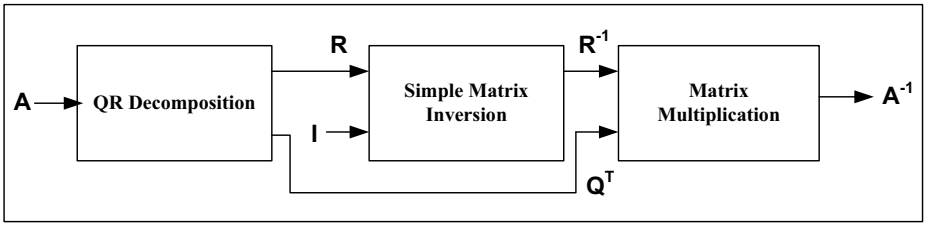


Figure 4.1: **The solution steps of the matrix inversion using QR decomposition.**

4.3.3 LU Decomposition Based Matrix Inversion

The solution for the inversion of a matrix A, A^{-1}, using LU decomposition is shown as follows:

$$A^{-1} = U^{-1} \times L^{-1} \tag{4.6}$$

This solution consists of four different parts: LU decomposition of the given matrix, matrix inversion for the lower triangular matrix, matrix inversion of the upper triangular matrix and matrix multiplication (Figure 4.2). LU decomposition

is the dominant calculation where the next three parts are relatively simple due to the triangular structure of the matrices.

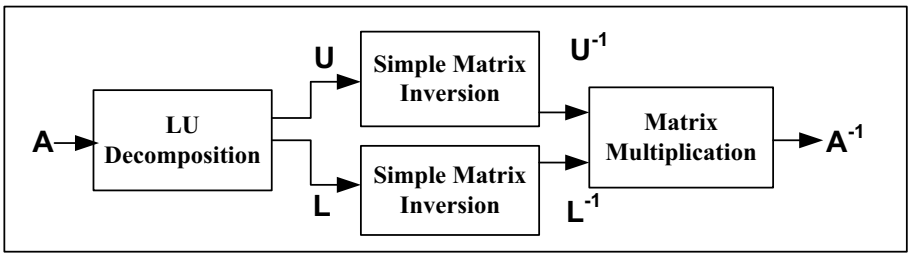


Figure 4.2: **The solution steps of the matrix inversion using LU decomposition.**

4.3.4 Cholesky Decomposition Based Matrix Inversion

The solution for the inversion of a matrix, A^{-1}, using Cholesky decomposition is shown as follows:

$$A^{-1} = G^{-1} \times G^{T^{-1}} \tag{4.7}$$

This solution consists of four different parts: Cholesky decomposition, matrix inversion for the transpose of the lower triangular matrix, matrix inversion of the lower triangular matrix and matrix multiplication (Figure 4.3). Cholesky decomposition is the dominant calculation where the next three parts are relatively simple due to the triangular structure of the matrices.

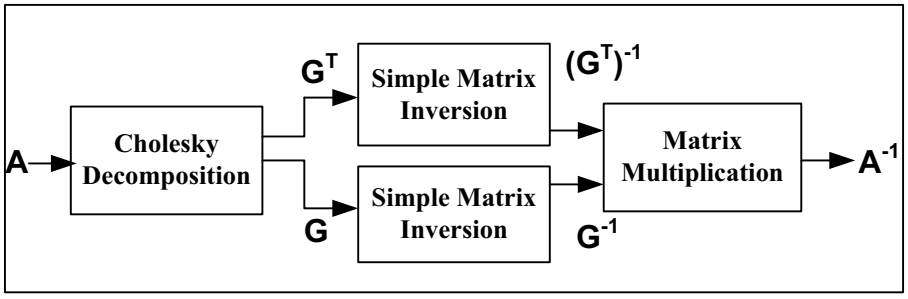


Figure 4.3: **The solution steps of the matrix inversion using Cholesky decomposition.**

4.3.5 Matrix Inversion using Analytic Method

Another method for inverting an input matrix A is the analytic method which uses the adjoint matrix, *Adj(A)*, and determinant, *det A*. This calculation is given by

$$A^{-1} = \frac{1}{detA} \times Adj(A) \tag{4.8}$$

The adjoint matrix is the transpose of the cofactor matrix where the cofactor matrix is formed by using determinants of the input matrix with signs depending on its position. It is formed in three stages. First, we find the transpose of the input matrix, A, by interchanging the rows with the columns. Next, the matrix of minors is formed by covering up the elements in its row and column and finding the determinant of the remaining matrix. Finally, the cofactor of any element is found by placing a sign in front of the matrix of minors by calculating *(-1)(i+j)*. These calculations are shown in Figure 4.4 for the first entry in the cofactor matrix, C_{11}.

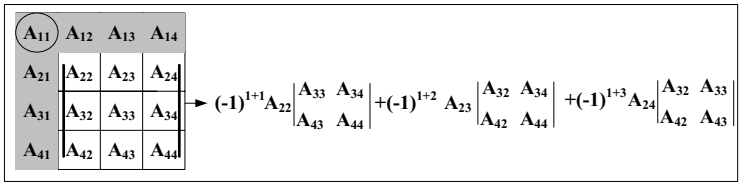


Figure 4.4: **Matrix Inversion with analytic approach. The calculation of the first element of cofactor matrix, C_{11}, for a 4×4 matrix is shown.**

The calculation of the first entry in the cofactor matrix, C_{11} is repeated 16 times for a 4×4 matrix to form the 4×4 cofactor matrix which has 16 entries. After the calculation of the adjoint matrix, the determinant is calculated using a row or a column. The last stage is the division between the adjoint matrix and the determinant which gives the inverted matrix.

Chapter 5

GUSTO: General architecture design Utility and Synthesis Tool for Optimization

Since matrix computation algorithms are expensive computational tasks, we designed an easy to use tool, GUSTO ("**G**eneral architecture design **U**tility and **S**ynthesis **T**ool for **O**ptimization") [52–55] for automatic generation and optimization of application specific processing elements (PEs) for matrix computation algorithms. GUSTO is the first tool of its kind to provide automatic generation and optimization of a variety of general purpose processing elements (PEs) with different parameterization options. It also optimizes the general purpose processing element to improve its area results and design quality with an outcome of an optimized application specific processing element. GUSTO receives the algorithm from the user and allows her to select the type and number of arithmetic resources, the data representation (integer and fractional bit width), and the different modes of operation for general purpose or application specific processing elements. The ability to choose different parameters allows the designer to explore different design methods and pick the most efficient one in terms of the desired objective.

GUSTO is capable of producing two different processor architectures in its design flow. Through it steps, GUSTO first creates a general purpose processing element and its datapath for given set of inputs. Furthermore, it opti-

mizes/customizes this general purpose processing element to improve its area results and design quality by trimming/removing the unused resources and creating an optimized application specific processing element while ensuring that correctness of the solution is maintained. GUSTO also creates required HDL files which are ready to simulate, synthesize and map.

The major contributions of this chapter are:

1) Design of an easy-to-use matrix computation core generator for efficient design space exploration for single core designs with reconfigurable bit widths, resource allocation, processor architecture types and methods which can generate and/or optimize the design;

2) Determination of inflection points, in terms of matrix dimensions and bit widths, between decomposition methods and inversion methods including analytic method;

3) A study of the area, timing and throughput tradeoffs using different design space decisions for different matrix computations such as matrix decomposition using QR, LU and Cholesky, matrix inversion using decomposition methods (QR, LU, Cholesky) and analytic method as well as a case study: Adaptive Weight Calculation (AWC) Core design.

The rest of this chapter is organized as follows: Section 5.1 introduces the flow of GUSTO: algorithm analysis, instruction generation, resource allocation, error analysis, processor architecture generation and optimization. Section 5.2 validates our methodology and effectiveness of our tool through the hardware development of matrix decomposition architectures. Section 5.3 presents a case study: Implementation of Adaptive Weight Calculation Core using QRD-RLS algorithm. Section 5.4 presents architectural result for different matrix inversion algorithms. Section 5.2, 5.3 and 5.4 also introduce FPGA resources, discuss design decisions and challenges, present implementation results in terms of area and performance, and compare our results with other published FPGA implementations. We conclude in Section 5.5.

5.1 Flow of GUSTO

There are several different architectural design alternatives for a matrix computation algorithm. For example, one can use QR, LU or Cholesky for matrix decomposition. Thus, it is important to study tradeoffs between these alternatives and find the most suitable solution for desired results such as most time efficient or most area efficient design. Performing design space exploration is a time consuming process where there is an increasing demand for higher productivity. High level design tools offer great convenience by easing this burden and giving us the opportunity to test different alternatives in a reasonable amount of time. Therefore, designing a high level tool for fast prototyping is essential.

GUSTO, "**G**eneral architecture design **U**tility and **S**ynthesis **T**ool for **O**ptimization," is such a high level design tool that is the first of its kind to provide design space exploration across different matrix computation architectures. As shown in Figure 5.1, GUSTO allows the user to select the matrix computation method (As an example, a user can select QR, LU, Cholesky decompositions or analytic method for matrix inversion), the type and number of arithmetic resources, the data representation (the integer and fractional bit width), and the type of processor element architecture.

After the required inputs are given, GUSTO generates a general purpose processing element and its datapath by using resource constrained list scheduling. The general purpose processing element is used for area and timing analysis for a general non-optimized solution. The advantage of generating a general purpose processing element is that it can be used to explore other algorithms, so long as these algorithms require the same resource library and memory resources. However, general purpose processing elements generally do not lead to high-performance results. Therefore optimizing/customizing these architectures to improve their area results is another essential step to enhance design quality.

GUSTO creates a general purpose processing element architecture which can be seen in Figure 5.2. The created architecture works at the instruction level where the instructions define the required calculations for the given matrix computation algorithm. For better performance results, instruction level paral-

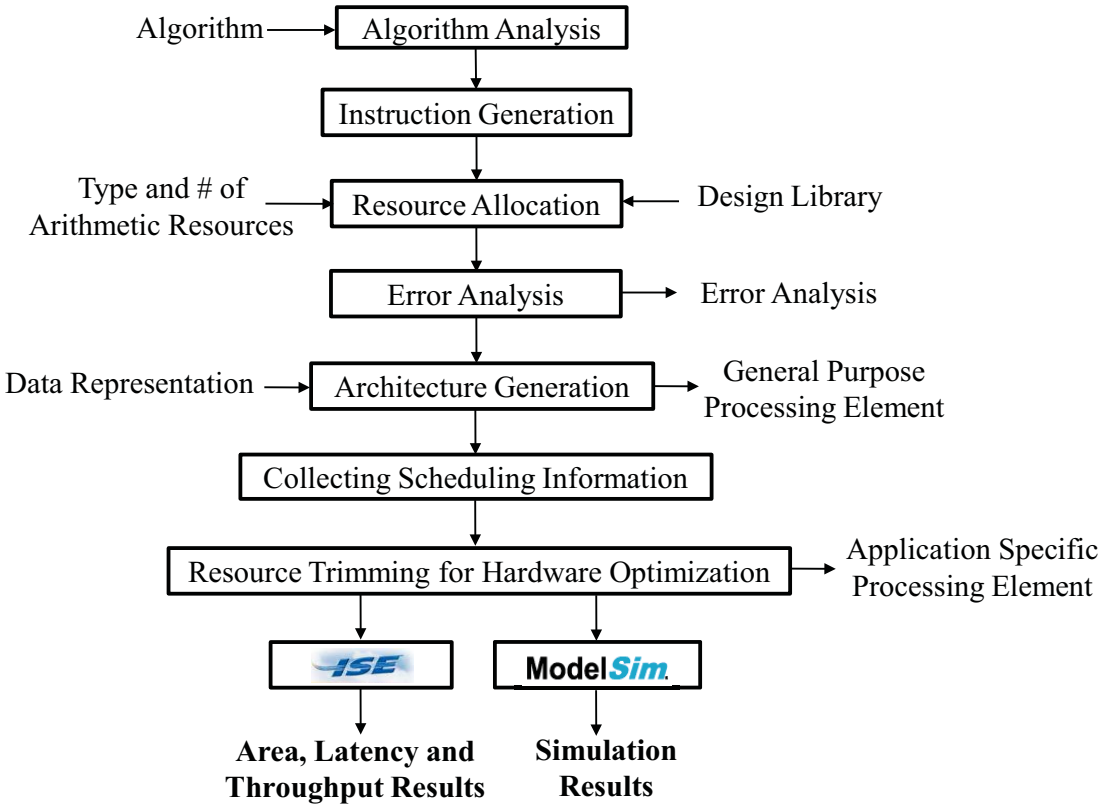


Figure 5.1: **Design Flow of GUSTO.**

lelism is exploited. The dependencies between the instructions limit the amount of parallelism that exists within a group of computations. Our proposed design consists of controller units and arithmetic units. The arithmetic units are capable of computing different matrix computation algorithms by employing adders, subtractors, multipliers, dividers, square root units, etc. that are needed where their type and number are defined by the user. In this architecture, controller units track the operands to determine whether they are available and assign a free arithmetic unit for the desired calculation. Every arithmetic unit fetches and buffers an operand as soon as the operand is ready. As the next step, GUSTO performs trimming/removing the unused resources from the general purpose processing element and creates an optimized application specific processing element while ensuring that correctness of the solution is maintained. GUSTO simulates

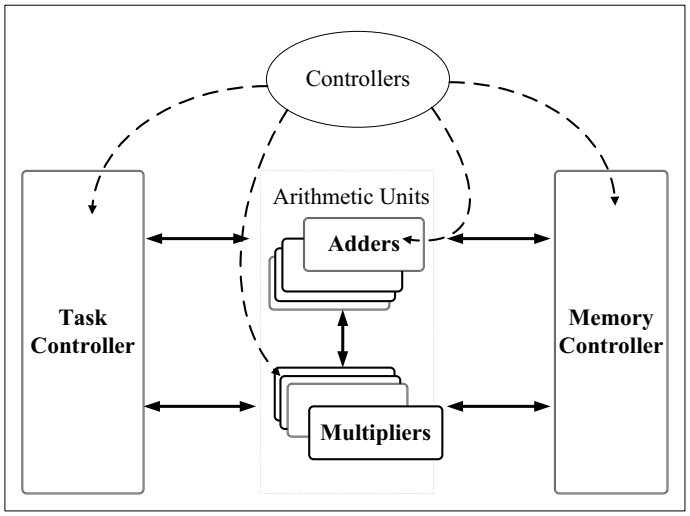


Figure 5.2: **General purpose architecture and its datapath.**

the architecture to define the usage of arithmetic units, multiplexers, register entries and input/output ports and trims away the unused components with their interconnects.

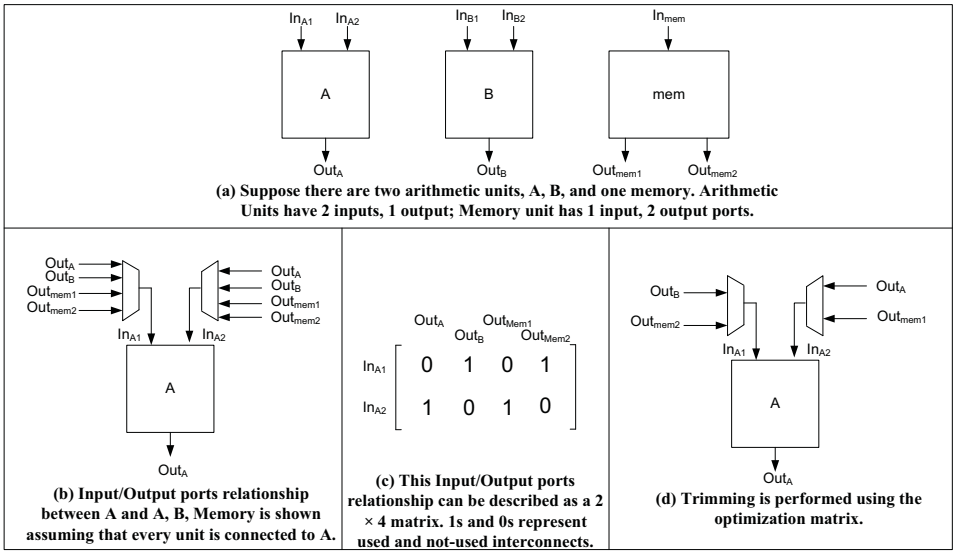


Figure 5.3: **An Example for the GUSTO's trimming feature.**

We present 2 different trimming examples:

1) There are 2 arithmetic units with 2 inputs/1 output each and one memory with 1 input/2 outputs in Figure 5.3(a). Input / output port relationships between arithmetic unit A and the other units are shown in a block diagram in (b). Although Out_A, Out_B, Out_{mem1}, and Out_{mem2} are all inputs to In_{A1} and In_{A2}, not all the inputs may be used during computation. We can represent whether an input/output port is used or not during simulation in a matrix such as the one shown in (c). As the simulation runs, the matrix is filled with 1s and 0s representing the used and unused ports respectively. GUSTO uses these matrices to remove the unused resources (d). In this example, two inputs, Out_A, Out_{mem1} to In_{A1} and another two inputs, Out_B, Out_{mem2} to In_{A2} are removed.

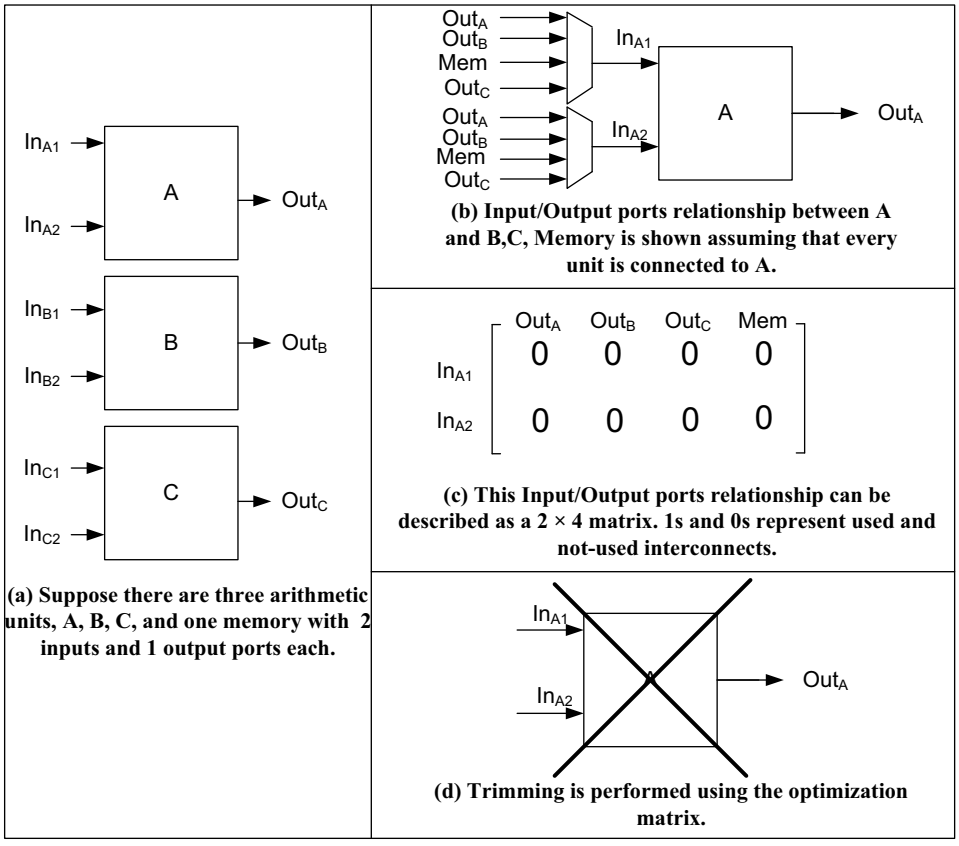


Figure 5.4: **An Example for the GUSTO's trimming feature.**

2) There are 3 arithmetic units and one memory with 2 inputs and 1 output each in Figure 5.4(a). Input / output port relationships between arithmetic unit A and the other units are shown in a block diagram in (b). Although Out_A, Out_B,

Mem, and Out_C are all inputs to In_{A1} and In_{A2}, not all the inputs may be used during simulation. We again represent whether an input/output port is used or not during simulation in a matrix such as the one shown in (c). As the simulation runs, the matrix is filled with 1s and 0s representing the used and unused ports respectively. In this example, none of the possible inputs are used and arithmetic units is removed with its interconnects(d).

GUSTO defines **an optimization matrix** for each arithmetic unit employed in the processing element that shows the connections with respect to other arithmetic units. These optimization matrices are used to trim away unused resources therefore leads to a processing element which has only the required connectivity internally for the given algorithm. In order to achieve highly efficient silicon implementations of matrix computation algorithms, GUSTO employs fixed point arithmetic in generated architectures. GUSTO performs error analysis to find a fixed point representation which provides results with the accuracy similar to that of a floating point implementation which we introduce in the following subsection.

5.1.1 Error Analysis

There are two different types of approximations for real numbers: fixed-point and floating-point arithmetic systems. Floating-point arithmetic represents a large range of numbers with some constant relative accuracy. Fixed-point arithmetic represents a reduced range of numbers with a constant absolute accuracy. Usage of floating point arithmetic is expensive in terms for hardware and leads to inefficient designs especially for FPGA implementation. On the other hand, fixed point arithmetic results in efficient hardware designs with the possibility of introducing calculation error.

We use two's complement fixed point arithmetic in our implementations as it results in faster and smaller functional units. The data lines used in our implementations for fixed point arithmetic consist of an integer part, a fractional part and a sign bit. Fixed-point arithmetic reduces accuracy and consequently introduces two types of errors: round-off and truncation errors. Round-off error occurs when the result requires more bits than the reserved bit width after a computation.

Truncation error occurs due to the limited number of bits to represent numbers. These issues must be handled carefully to prevent incorrect or low accuracy results. Thus, error analysis is a crucial step to determine how many bits are required to satisfy accuracy requirements.

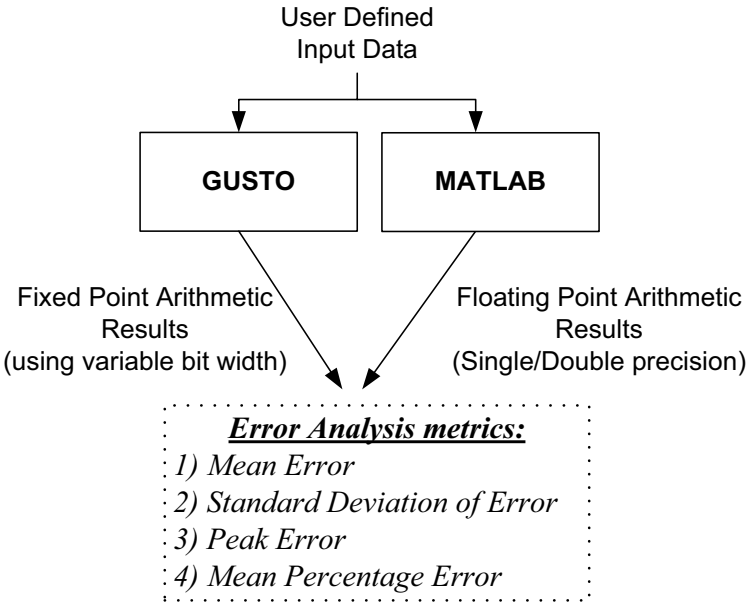


Figure 5.5: **Performing error analysis using GUSTO.**

GUSTO performs error analysis after the resource allocation step (shown in Figure 5.1) to find an appropriate fixed point representation which provides results with the accuracy similar to that of a floating point implementation. GUSTO takes the sample input data which is generated by the user. The matrix computation is performed using single or double precision floating point arithmetic and these are referred as the actual results. The same calculations are performed using different bit widths of fixed point representations to determine the error, the difference between the actual and the computed result. GUSTO provides four different metrics to the user to determine if the accuracy is enough for the application: *mean error*, *standard deviation of error*, *peak error*, and *mean percentage error* as shown in Figure 5.5.

The first metric, mean error, is computed by finding the error for all resulting matrix entries and then dividing the sum of these errors by the total number of entries. This calculation can be seen as:

$$\frac{\sum_{i=1}^{m} |y_i - \hat{y}_i|}{m} \tag{5.1}$$

where y, $\hat{y}$ and m are the actual results, the computed results and the number of entries which are used in the computation (16 for a 4×4 matrix), respectively. Mean error is an important metric for error analysis however it does not include the information about outlier errors. This is the case where a small number of entries have very high error but the majority of entries have very small error. To calculate the dispersion from the mean error, the standard deviation of error and the peak error, are introduced in our tool. Mean error sometimes leads to misleading conclusions if the range of the input data is small. Therefore the fourth metric, mean percentage error, makes more sense if the relative error is considered. This metric is defined as

$$\frac{\sum_{i=1}^{m} |\frac{y_i - \hat{y}_i}{y_i}|}{m} \tag{5.2}$$

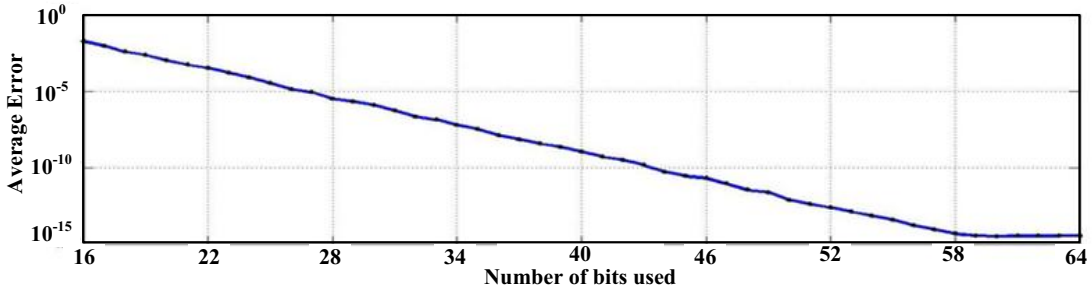


Figure 5.6: **An error analysis example, mean error, provided by GUSTO for QR decomposition based 4×4 matrix inversion. The user can select the required number of bit widths for the application where the increasing number of bits results in high accuracy.**

As an example, we perform an error analysis for QR decomposition based

matrix inversion. We generate uniformly distributed pseudorandom numbers, [0, 1], for a 4×4 matrix. The mean error results, provided by GUSTO, are shown in Figure 5.6 in log domain where mean error decreases with the increase in the number of bits used as bit width. Therefore, the user can determine how many bits are required for the desired accuracy. It is important to note that the tool also provides standard deviation of error, peak error and mean percentage error.

The following section presents the effectiveness of our tool, GUSTO, by showing its different design space exploration examples for matrix decomposition architectures. We present area results in terms of slices and performance results in terms of throughput. Throughput is calculated by dividing the maximum clock frequency (MHz) by the number of clock cycles to perform particular matrix computation. All designs are generated in Verilog and synthesized using Xilinx ISE 9.2. Resource utilization and design frequency are post place and route values obtained using a Virtex 4 SX35 FPGA.

All functional units are implemented using the Xilinx Coregen toolset. The addition and subtraction units are implemented with SLICES, the multiplications use XtremeDSP blocks, the divider core uses a circuit for fixed-point division based on radix-2 non-restoring division and the square root unit uses a CORDIC core. We use Block RAMs available on Xilinx FPGAs as memory storage space for instructions. The Block RAM modules provide flexible 18Kbit dual-port RAM, that are cascadable to form larger memory blocks. Embedded XtremeDSP SLICES with 18×18 bit dedicated multipliers and a 48-bit accumulator provide flexible resources to implement multipliers to achieve high performance. Furthermore, the Xilinx Coregen toolset implements these cores very efficiently since it uses special mapping and place & route algorithms allowing for high performance design.

5.2 Matrix Decomposition Architectures

Matrix decompositions are essential computations for simplifying and reducing the computational complexity of various algorithms used in wireless communication. For example, decomposition methods are used for simplifying matrix

inversion which are used in MIMO-OFDM systems' minimum mean square error algorithms for pre-coding in spatial multiplexing [7], equalization algorithms to remove the effect of the channel on the signal [4] and detection-estimation algorithms in space-time coding [8].

We divided this section into two parts: inflection point analysis and architectural design alternatives analysis. Inflection point analysis presents timing results of decomposition methods in terms of clock cycles for different executions (sequential and parallel), matrix sizes and bit-widths. Inflection point analysis answers at what matrix size does an inflection point occur and how does varying bit width and degree of parallelism change the inflection point for a specific decomposition method? Architectural design alternatives analysis presents area and performance results of different decomposition architectures using different parameterizations. Area and performance results are presented in terms of slices and throughput respectively. This analysis also shows how GUSTO finds the optimal hardware by moving from general purpose to application specific processing element for different decomposition methods. Throughput is calculated by dividing the maximum clock frequency (MHz) by the number of clock cycles to perform matrix decomposition. Our designs are generated in Verilog and synthesized using Xilinx ISE 9.2. Resource utilization and design frequency are post place and route values obtained using a Virtex 4 SX35 FPGA.

5.2.1 Inflection Point Analysis

We present three different analyses for comparison of decomposition methods in Figure 5.7, 5.8 and 5.9. The total number of operations for different decomposition methods is shown in Figure 5.7 in log domain. We compare sequential and parallel execution of different decomposition methods for different bit widths (16, 32 and 64 bits) and matrix sizes in Figure 5.8 and 5.9 respectively. QR, LU and Cholesky decomposition methods are shown with square, spade and triangle respectively. 16, 32 and 64 bits of bit widths are shown with smaller dashed, dashed and solid lines respectively. We show inflection points between these decomposition methods by balloons.

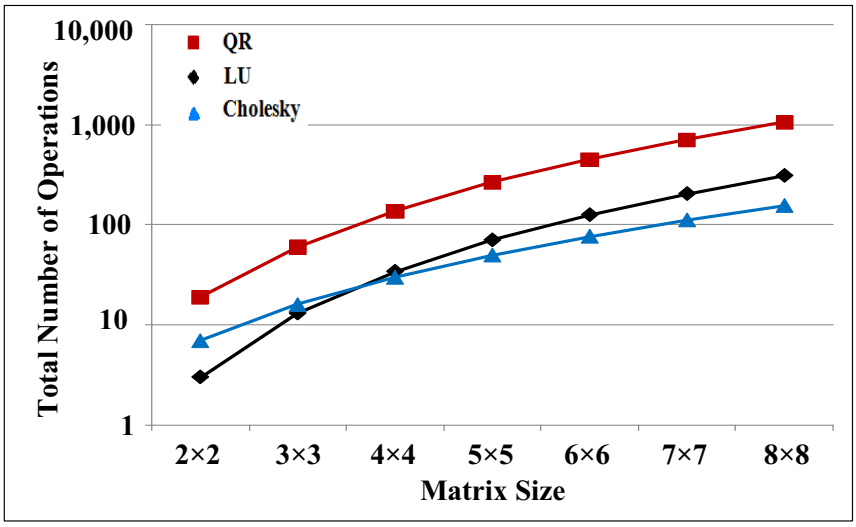


Figure 5.7: **Total number of operations for decomposition methods in log domain.**

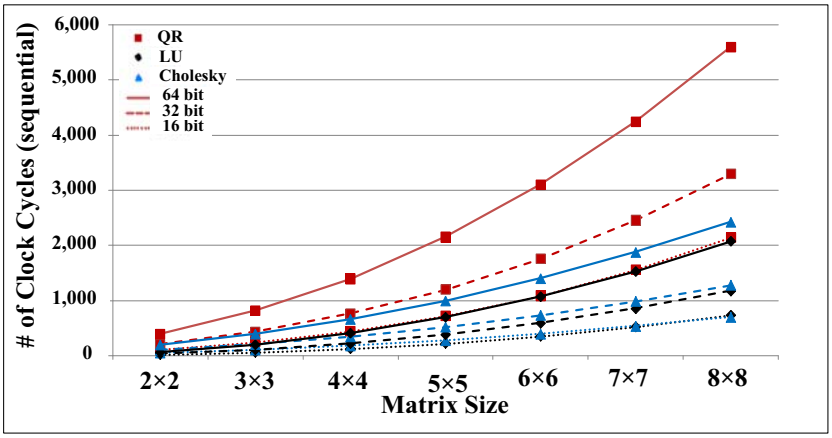


Figure 5.8: **The comparison between different decomposition methods using sequential execution.**

The total number of operations, Figure 5.7, shows that QR decomposition requires significantly higher number of operations compared to LU and Cholesky decompositions. LU and Cholesky decompositions require a close number of operations where LU decomposition requires more operations than Cholesky decomposition after 4×4 matrices. The sequential execution of decomposition methods, Figure 5.8, show that QR decomposition requires more clock cycles than the other decomposition methods for all bit widths. Execution of 16 bit QR decomposi-

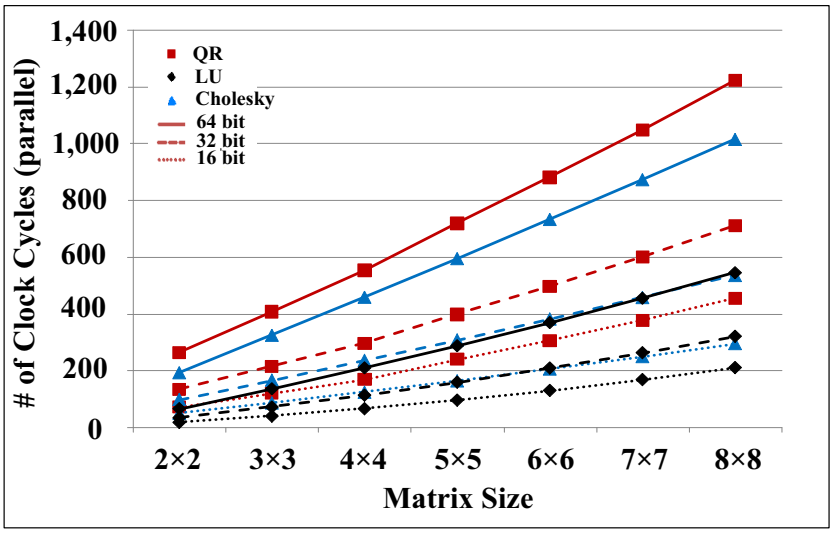


Figure 5.9: **The comparison between different decomposition methods using parallel execution.**

tion requires the same number of clock cycles with the 64 bit LU decomposition. Cholesky decomposition requires more clock cycles than LU decomposition because of its square root operations; however the difference between LU and Cholesky becomes smaller for smaller number of bit widths.

The parallel execution of decomposition methods, Figure 5.9, shows that LU decomposition performs better than other methods. 64 bit implementation of LU decomposition performs almost the same as the 32 bit Cholesky decomposition and also the 32 bit LU decomposition performs almost the same as the 16 bit implementation of Cholesky decomposition.

5.2.2 Architectural Design Alternatives for Matrix Decomposition Algorithms

We present two different analyses for comparison of decomposition methods in terms of architectural design alternatives in Figure 5.10 and 5.11 for 4×4 matrices. The general purpose and application specific processing element architectural results are compared in Figure 5.10 in terms of slices (area) and throughput (performance). We compare area and throughput results of different bit width

implementations of decomposition methods in Figure 5.11.

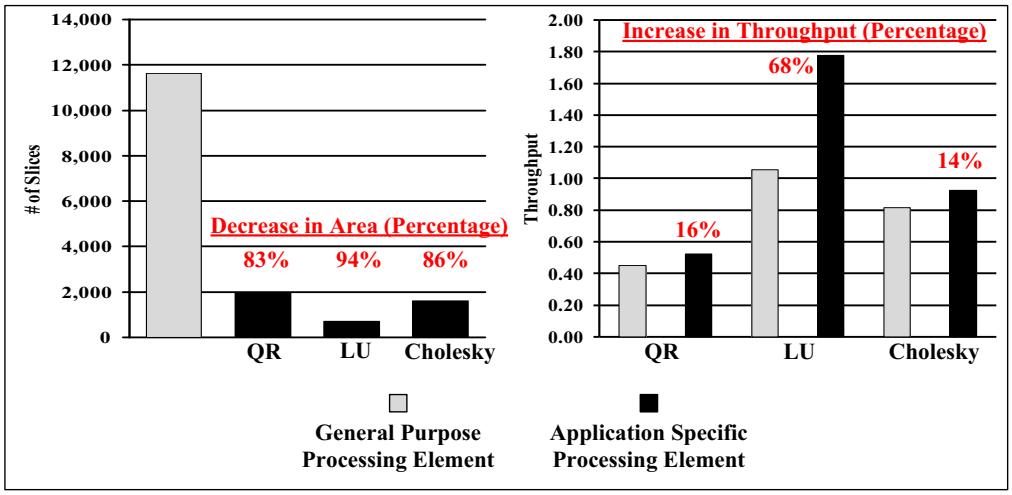


Figure 5.10: **The comparison of the general purpose processing element and application specific processing element architectures in terms of slices (area) and throughput (performance).**

The general purpose architecture is able to perform different decomposition methods with a selection input. However, it is possible to provide better area and throughput results by optimizing the general purpose architecture and creating an application specific architecture for a specific decomposition method. As can be seen in Figure 5.10, by creating an application specific architecture which uses the optimal number of resources, we can decrease the area by 83%, 94% and 86% for QR, LU and Cholesky decompositions respectively. Optimizing the architecture also provides higher clock frequencies and leads 16%, 68% and 14% increase in throughput for QR, LU and Cholesky decompositions respectively.

We also present area and throughput results for different bitwidths of data: 19, 26 and 32 bits in Figure 5.11. The user can determine the bitwidth of the data by the error analysis part of our tool. High precision can be obtained by using a larger number of bits but this comes at the price of larger area and lower throughput. As can be seen in Figure 5.11, LU decomposition provides the smallest area and highest throughput for all bit widths. Also, it is important to see that there is an inflection point between QR and Cholesky decompositions at 25 bits in terms of area where Cholesky decomposition requires more area for bit widths

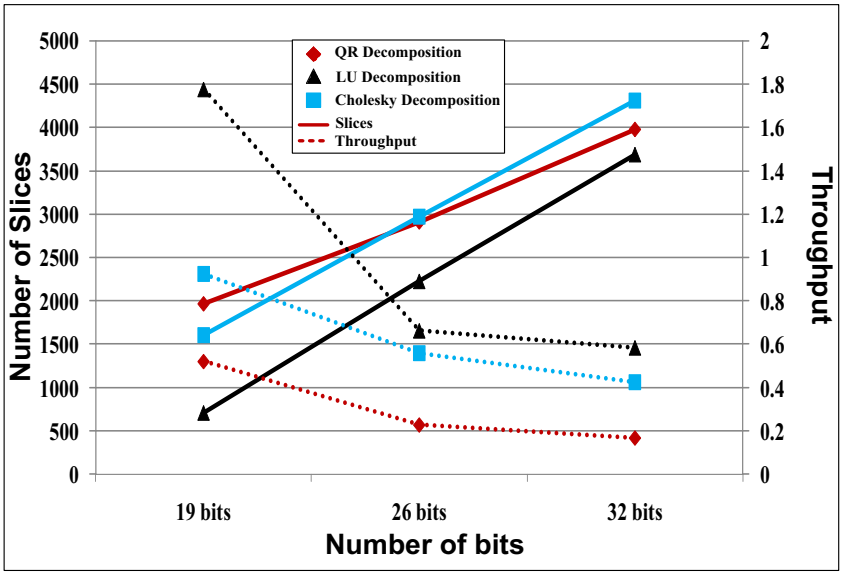


Figure 5.11: **Area and throughput tradeoffs for different bit width of data: 19, 26 and 32 bits. A user can determine the bitwidth of the data by the error analysis part of GUSTO. High precision can be obtained by using more number of bits as bitwidth which comes at the price of larger area and lower throughput.**

larger than 25 bits. On the other hand, Cholesky decomposition provides higher throughput compared to QR decomposition for all bit widths due to the fact that Cholesky decomposition requires less clock cycles.

In the next section, we present a case study: Implementation of Adaptive Weight Calculation Core, using our matrix computation core generator tool, GUSTO.

5.3 Adaptive Weight Calculation Core using QRD-RLS Algorithm

Adaptive weight calculation (AWC) is required in many wireless communication applications including adaptive beamforming, equalization, predistortion and multiple-input multiple-output (MIMO) systems [9]. These applications in-

volve solving systems of equations in many cases which can be seen as:

$$a_{11}x_1 + a_{12}x_2 \cdots + a_{1m}x_m = b_1 + e_1 \tag{5.3}$$

$$a_{21}x_1 + a_{22}x_2 \cdots + a_{2m}x_m = b_2 + e_2 \tag{5.4}$$

$$\vdots \tag{5.5}$$

$$a_{n1}x_1 + a_{n2}x_2 \cdots + a_{nm}x_m = b_n + e_n \tag{5.6}$$

where A is the observations matrix which is assumed to be noisy, b is a known training sequence and x is the vector to be computed by using least squares method. This is described more compactly in matrix notation as $Ax = b + e$. If there is the same number of equations as there are unknowns, i.e. $n = m$, this system of equations has a unique solution, $x = A^{-1}b$. However, high sampling rate communication applications are often over-determined, i.e. $n > m$. Introducing the least squares approach helps to solve the problem by minimizing the residuals: $\min \sum_n e_n^2$.

In general, the least squares approach, e.g. Least Mean Squares (LMS), Normalized LMS (NLMS) and Recursive Least Squares (RLS), is used to find an approximate solution to these kinds of system of equations. Among them, RLS is most commonly used due to its good numerical properties and fast convergence rate [10]. However, it requires matrix inversion which is not efficient in terms of precision and hardware implementation. Applying QR decomposition to perform adaptive weight calculation based on RLS is a better method and leads to more accurate results and efficient architectures. Therefore, we use our tool, GUSTO, to implement an adaptive weight calculation (AWC) core that employs QR decomposition in its solution steps. The resulting upper triangular matrix, R, which is the solution of QR decomposition, is used to find coefficients of the system by back-substitution after converting b into another column matrix, c, such that $Rx = c$ (Figure 5.12).

5.3.1 Comparison

We design two architectures with different bitwidthds: 14 and 20 bits and compare our results with different applications which use decomposition methods

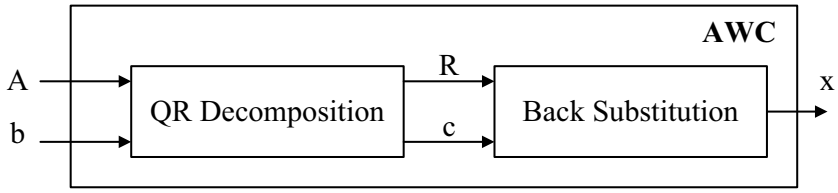


Figure 5.12: **Adaptive Weight Calculation (AWC) using QRD-RLS method consists of two different parts to calculate the weights, QR decomposition and back-substitution.**

in their solution methods in Table 5.1. These related works are hard to compare with each other since two of them are matrix inversion architectures [18,19], one is a beamformer design [20], and ours is an adaptive weight calculation design. We also use fixed point arithmetic and fully used FPGA resources like DSP48 multipliers instead of Look-up Tables (LUTs). Therefore our intention is not to directly compare these different designs, but to give an idea about our area and throughput results compared to other implementations that use decomposition methods. Edman et al. proposed a linear array architecture for inversion of complex valued matrices [18]. Karkooti et al. presented an implementation of matrix inversion using the QRD-RLS algorithm along with square GR and folded systolic arrays [19]. Dick et al. considered the architecture, design flow and the verification process of a real-time beamformer [20] which is most similar to our work since area and timing results are presented for QR decomposition and back substitution architectures (clock frequency is assumed as 250 MHz). The advantage of our work compared to the related work is we give the ability to the designer to study the tradeoffs between architectures with different design parameters to find an optimal design.

The following section presents the effectiveness of our tool, GUSTO, by showing its different design space exploration examples for matrix inversion architectures.

5.4 Matrix Inversion Architectures

Matrix inversion algorithms lie at the heart of most scientific computational tasks. Matrix inversion is frequently used to solve linear systems of equations in

Table 5.1: Comparisons between our results and previously published papers. NR denotes not reported.

	[160]	**GUSTO**	[162]	**GUSTO**
Application	Inversion	**Matrix Inv.**	Beamformer	**Beamformer**
Matrix Dimensions	4×4	4×4	4×4	4×4
Bit width	12	**12**	**18**	**18**
Data type	fixed	**fixed**	**fixed**	**fixed**
Device type (Virtex)	II	**II**	**IV**	**IV**
Slices	4400	**2214**	**3530**	**2313**
DSP48s	NR	**8**	**13**	**8**
BRAMs	NR	**1**	**6**	**1**
Throughput $(10^6 \times s^{-1})$	0.28	**0.33**	**0.19**	**0.14**

many fields such as wireless communication. These systems often use a small number of antennas (2 to 8) which results in small matrices to be decomposed and/or inverted. For example the 802.11n standard [11] specifies a maximum of 4 antennas on the transmit/receive sides and the 802.16 [12] standard specifies a maximum of 16 antennas at a base station and 2 antennas at a remote station. In this section, we present different design space exploration examples using different inputs of GUSTO and compare our results with previously published FPGA implementations.

For the analytic method, we present three different designs, *Implementation A, B, and C*, with varying levels of parallelism (using cofactor calculation cores in parallel). *Cofactor calculation core* is a processing core that is designed by GUSTO to execute instructions to form cofactor matrices which can be seen in Figure 5.13. *Implementation A* uses one cofactor calculation core, *Implementation B* uses two cofactor calculation cores and *Implementation C* uses 4 cofactor calculation cores.

Design space exploration can again be divided into two parts: inflection point analysis and architectural design alternatives analysis.

5.4.1 Inflection Point Analysis

In this subsection, we first compare QR decomposition and analytic method because they are both applicable to any matrix. Then, we compare different de-

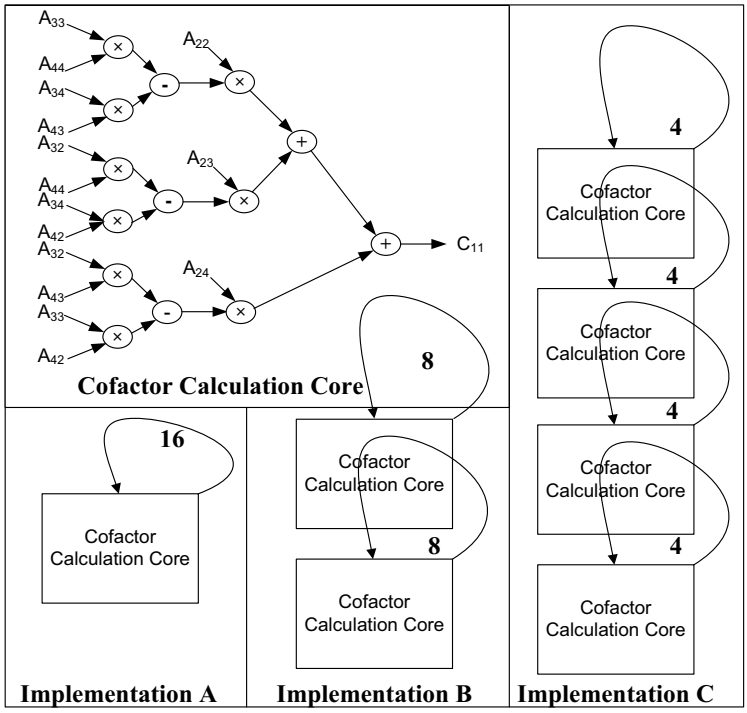


Figure 5.13: **Three different designs, *Implementation A, B, and C*, with varying levels of parallelism (using cofactor calculation cores in parallel) to form cofactor matrices.**

composition methods (QR, LU, and Cholesky) to benefit from different matrix characteristics.

Comparison of QR decomposition based matrix inversion and Analytic Method:

The total number of operations used in these methods is shown in Figure 5.14 in log domain. It is important to notice that the total number of operations increases by an order of magnitude for each increase in matrix dimension for the analytic method making the analytic solution unreasonable for large matrix dimensions. Since the analytic approach does not scale well, there will be an inflection point where the QR decomposition approach will provide better results. **At what matrix size does this inflection point occur and how does varying bit width and degree of parallelism change the inflection point?** The comparisons for sequential and parallel executions of QR and analytic methods

are shown in Figure 5.15 and 5.16 with different bit widths: 16, 32 and 64. We used implementation A for the parallel implementation of analytic method. Solid and dashed lines represent QR decomposition method and analytic method results respectively. The balloons denote the inflection points between the two methods for the different bit widths.

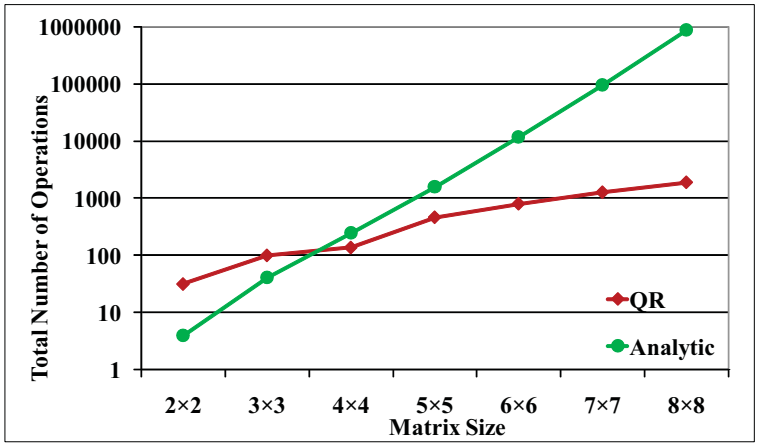


Figure 5.14: **The total number of operations for both the QR decomposition based matrix inversion and the analytic method in log domain.**

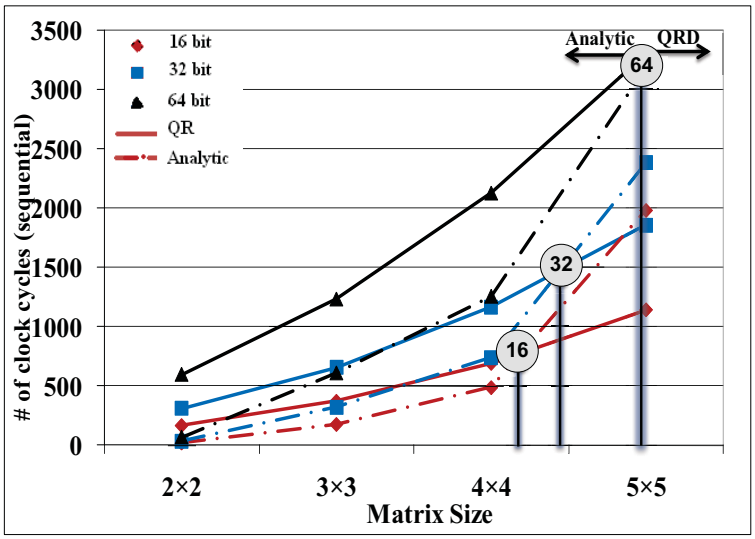


Figure 5.15: **The inflection point determination between the QR decomposition based matrix inversion and the analytic method using sequential execution.**

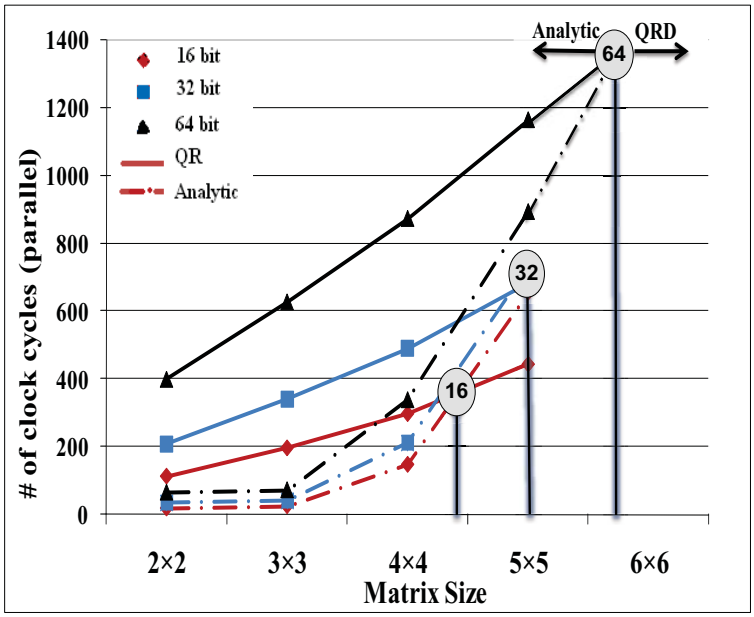


Figure 5.16: **The inflection point determination between QR decomposition based matrix inversion and analytic method using parallel execution.**

The sequential execution results (Figure 5.15) show that the analytic method offers a practical solution for matrix dimensions $\leq 4 \times 4$. It also gives the same performance as the QR decomposition method for 5×5 matrices using 64 bits. The analytic method result increases dramatically for 6×6 matrices (not shown) where it needs 12,251 clock cycles (for 16 bits) as opposed to 1,880 clock cycles for QR decomposition suggesting the analytic method is unsuitable for matrix dimensions $\geq 6 \times 6$.

The parallel execution results are shown in Figure 5.16. Analytic method offers a practical solution for matrix dimensions $\leq 4 \times 4$ and it is preferred for 5×5 matrix dimension for 32 and 64 bits. The increase in the clock cycle is again dramatic for matrix dimensions $\geq 6 \times 6$ for the analytic method demanding to use the QR decomposition method for these larger matrix dimensions.

Comparison of different decomposition based matrix inversion methods:

The total number of operations used in different decomposition methods based matrix inversion architectures is shown in Figure 5.17 in log domain. It is

important to notice that there is an inflection point between LU and Cholesky decompositions at 4×4 matrices with a significant difference from QR decomposition. The comparisons for sequential and parallel executions of QR, LU and Cholesky decomposition based matrix inversion architectures are shown in (Figure 5.18 and 5.19) respectively with different bit widths: 16, 32 and 64. Square, spade and triangle represent QR, LU and Cholesky methods respectively. Solid, dashed and smaller dashed lines represent 64, 32 and 16 bits of bit widths respectively. The balloons denote the inflection points between these methods for the different bit widths where an inflection point occurs.

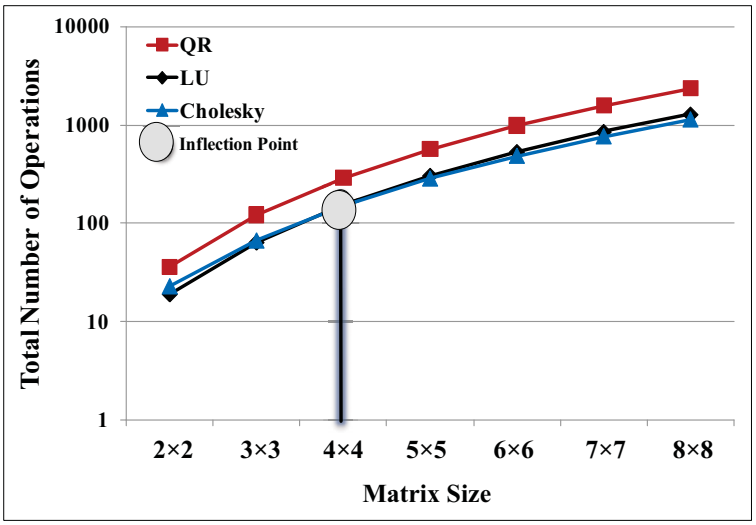


Figure 5.17: **The total number of operations for different decomposition based matrix inversion methods in log domain.**

The sequential execution results of decomposition methods based matrix inversion architectures (Figure 5.18) show that QR takes more clock cycles than Cholesky and LU where Cholesky takes more cycles than LU. As the bit widths get smaller, the difference between QR and the others doesn't change significantly, however it becomes smaller between Cholesky and LU decomposition based inversions. There is an inflection point between LU and Cholesky decompositions at 7×7 matrices for 16 bits.

The parallel execution results of decomposition methods based matrix inversion (Figure 5.19) show that QR decomposition based matrix inversion archi-

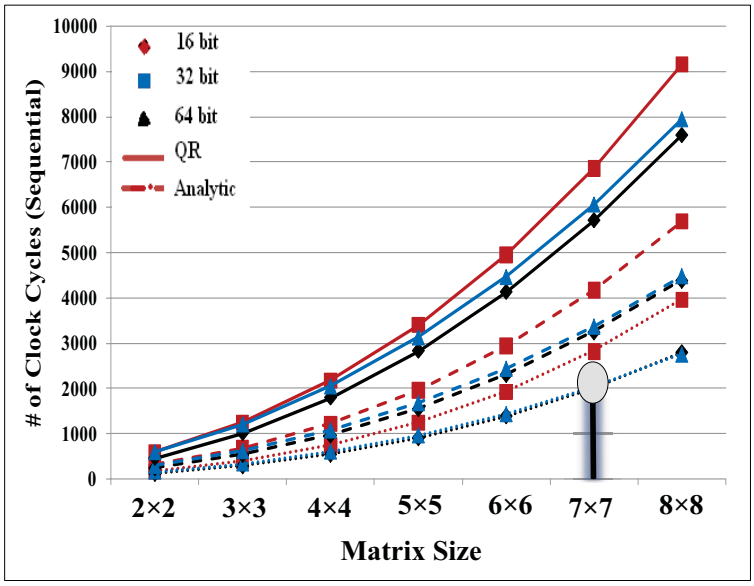


Figure 5.18: **The comparison between different decomposition based matrix inversion methods using sequential execution.**

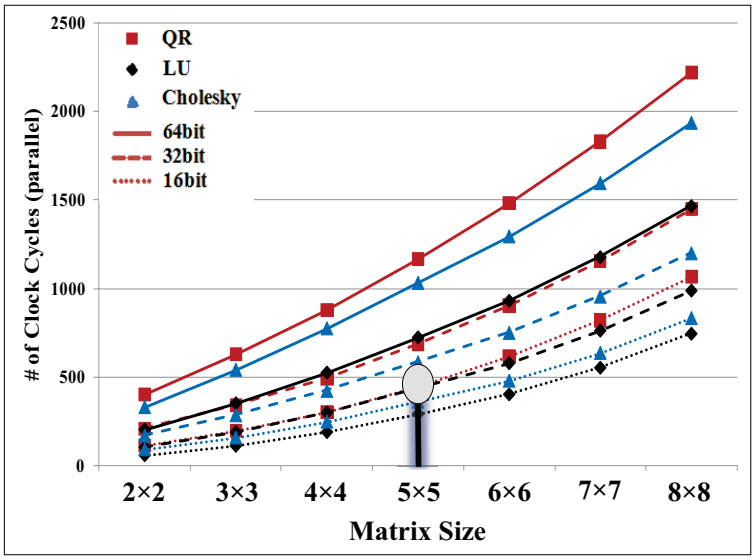


Figure 5.19: **The comparison between different decomposition based matrix inversion methods using parallel execution.**

tectures have the highest number of clock cycles for all bit widths where Cholesky and LU decomposition based matrix inversion architectures have a similar number of clock cycles for small bit widths. However, LU decomposition uses increasingly

fewer clock cycles than Cholesky decomposition with increasing bit widths and matrix dimensions. LU decomposition with 16 bits performs almost the same as QR decomposition with 32 bits. Also, 64 bits LU decomposition performs almost the same as 32 bits QR decomposition in terms of total number of clock cycles.

5.4.2 Architectural Design Alternatives for Matrix Inversion Architectures

These analyses are shown for QR, LU and Cholesky decomposition based matrix inversion architectures and analytic method based matrix inversion architectures for 4×4 matrices. We present both general purpose and application specific processing element results in Figure 5.20 to show the improvement in our results with the optimization feature, and present only application specific processing element results in Figure 5.21 and 5.22.

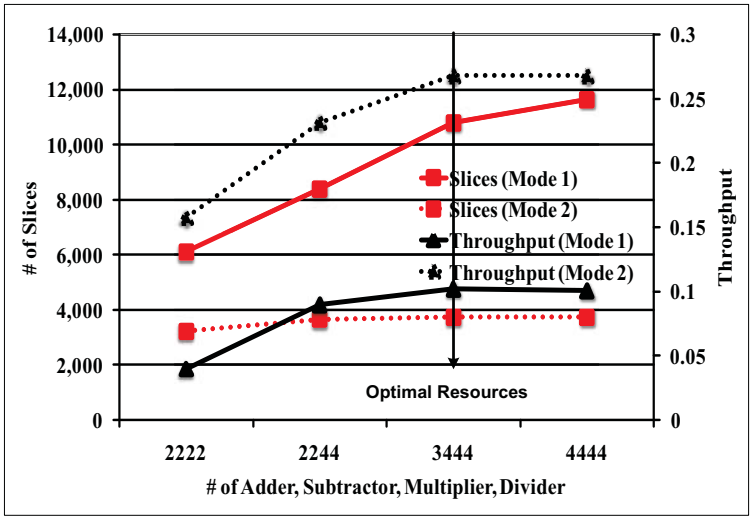


Figure 5.20: **Design space exploration for QR decomposition based matrix inversion architectures using different resource allocation options.**

We investigate different resource allocations for QR decomposition based matrix inversion architectures using different processor types that are provided by GUSTO and present the results in Figure 5.20. As expected from general purpose processing element, Figure 5.20 shows an increase in area and throughput as the

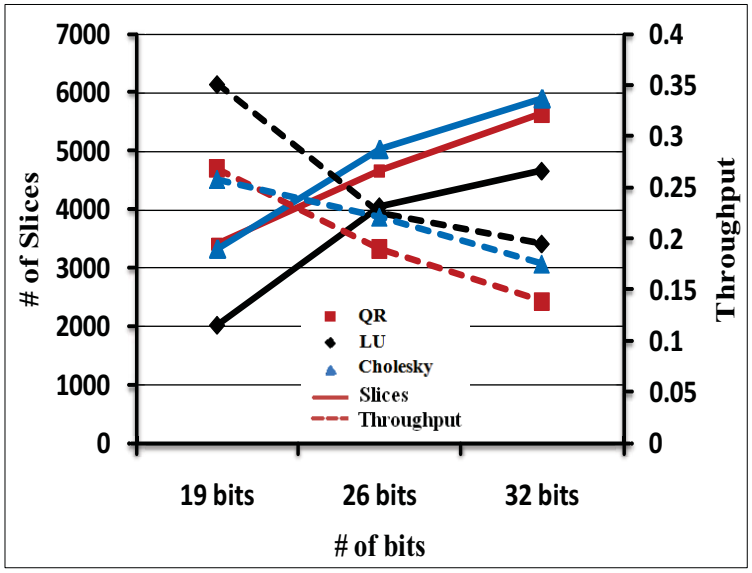


Figure 5.21: **Design space exploration for decomposition based matrix inversion architectures using different bit widths.**

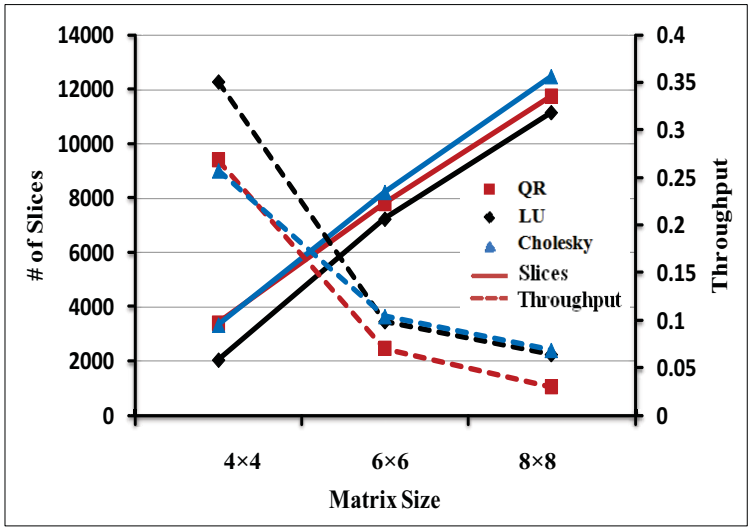


Figure 5.22: **Design space exploration for decomposition based matrix inversion architectures using different matrix sizes.**

number of resources increase up to optimal number of resources. Adding more than the optimal number of resources decreases throughput while still increasing area. However, application specific processing element generated by GUSTO finds the optimal number of resources which maximizes the throughput while minimiz-

ing area which is shown in Figure 5.20. Application specific processing element's optimized architecture can therefore provide an average of 59% decrease in area and 3X increase in throughput over general purpose (non optimized) processing element. Bit width of the data is another important input for the matrix inversion. The precision of the results is directly dependent on the number of bits used. The usage of a high number of bits results in high precision at a cost of higher area and lower throughput. We present 3 different bit widths, 19, 26 and 32 bits in Figure 5.21 for these three different decomposition based matrix inversion architectures. Usage of LU decomposition for matrix inversion results in smallest area and highest throughput compared to the other methods. Cholesky decomposition offers higher throughput at a cost of larger area compared to QR decomposition.

We also present three different matrix dimensions, 4×4, 6×6 and 8×8, implementation results in Figure 5.22 showing how the area and performance results scale with matrix dimension. We again observe that LU decomposition based matrix inversion architectures offer better area and throughput results compared to other methods.

5.4.3 Comparison

Comparisons between our results and previously published implementations for analytic based matrix inversion and decomposition based matrix inversion for 4×4 matrices is presented in Table 5.2 and 5.3 respectively. For ease of comparison we present all of our implementations with bit width 20 as this is the largest bit width value used in the related works. Though it is difficult to make direct comparisons between our designs and those of the related works (because we used fixed point arithmetic instead of floating point arithmetic and fully used FPGA resources (like DSP48s) instead of LUTs), we observe that our results are comparable. The main advantages of our implementation are that, it provides the designer the ability to study the tradeoffs between architectures with different design parameters and provides a means to find an optimal design.

Table 5.2: Comparisons between our results and previously published articles for Matrix Inversion using Analytic Method. NR denotes not reported.

			GUSTO		
Method	[17]	[17]	**Impl A**	**Impl B**	**Impl C**
Bit width	16	20	**20**	**20**	**20**
Data type	floating	floating	**fixed**	**fixed**	**fixed**
Device type (Virtex)	IV	IV	**IV**	**IV**	**IV**
Slices	1561	2094	**702**	**1400**	**2808**
DSP48s	0	0	**4**	**8**	**16**
BRAMs	NR	NR	**0**	**0**	**0**
Throughput $(10^6 \times s^{-1})$	1.04	0.83	**0.38**	**0.72**	**1.3**

Table 5.3: Comparisons between our results and previously published articles for Decomposition based Matrix Inversion Architectures. NR denotes not reported.

	[18]	[19]	**GUSTO**		
Method	QR	QR	**QR**	**LU**	**Cholesky**
Bit width	12	20	**20**	**20**	**20**
Data type	fixed	floating	**fixed**	**fixed**	**fixed**
Device type (Virtex)	II	IV	**IV**	**IV**	**IV**
Slices	4400	9117	**3584**	**2719**	**3682**
DSP48s	NR	22	**12**	**12**	**12**
BRAMs	NR	NR	**1**	**1**	**1**
Throughput $(10^6 \times s^{-1})$	0.28	0.12	**0.26**	**0.33**	**0.25**

5.5 Conclusion

This chapter describes our matrix computation core generator tool, GUSTO, that we developed to enable easy design space exploration for various matrix computation architectures which targets reconfigurable hardware designs. GUSTO provides different parameterization options including bit widths and resource allocations which enable us to study area and performance tradeoffs over a large number of different architectures. We present QR, LU, and Cholesky decompositions, AWC calculation, and decomposition and analytic method based matrix inversion architectures, to observe the advantages and disadvantages of all of these different methods in response to varying parameters. GUSTO is the only tool that allows design space exploration across different matrix computation architectures.

Its ability to provide design space exploration, which leads to an optimized application specific processing element architecture, makes GUSTO an extremely useful tool for applications requiring matrix computations.

The text of Chapter 5 is in part a reprint of the material as it appears in the proceedings of the Transactions on Embedded Computing Systems. The dissertation author was the primary researcher and author and the co-authors listed on this publication [52–54] directed and supervised the research which forms the basis for Chapter 5.

Chapter 6

GUSTO's Single Processing Core Architecture

We believe that a general purpose processing core is an ideal starting point to create an application specific processing core since it provides us with fully connected highly custom architecture to perform efficient design space exploration. We see that there are substantial opportunities to eliminate unneeded functionality and create an application specific core for executing the assigned part of the algorithm. Connecting these optimized application specific cores, that each employs only the required amount of memory resources, can result in highly efficient multi-core architectures. Since matrix computation algorithms are expensive computational tasks, their hardware implementation requires a significant amount of time and effort. We believe that it is crucial to have a domain specific design tool which can automatically create these type of parallel platforms for a given matrix computation algorithm and a set of user decisions.

Therefore, we design our tool, GUSTO, for automatic multi-core architecture generation for matrix computation algorithms. We utilize FPGAs for these type of parallel processing architectures since they provide reconfigurability and let us to cheaply develop a prototype system. GUSTO can also generate ASIC designs. GUSTO receives the matrix computation algorithm from the user, generates instructions and presents a data flow graph to the user. The user performs partitioning to divide the given algorithms into small highly parallelizable processing

cores. The user also defines the type and number of arithmetic resources and the data representation (integer and fractional bit width) for each core. GUSTO then uses these inputs to generate *general purpose processing cores* and performs simulation & elimination to generate optimized *application specific processing cores*. It then connects these small cores to create a multi-core architecture. GUSTO employs a different amount of memory resources for each core in such a way that the memory accesses between cores are reduced and the overall platform generated has only the required connectivity between processing cores. As explained in Section 3, processing cores that are generated by GUSTO are simple but ALU oriented for high performance computation.

The major contributions of this chapter are:
1) Detailing the properties of the general purpose architectures generated by GUSTO, and optimizations performed to achieve application specific architectures;
2) A trimming optimization methodology to generate ***application specific processing cores*** *by simulating & eliminating* ***general purpose processing cores*** *that use dynamic scheduling and binding with out-of-order execution;*

The rest of the chapter is organized as follows. In section 2, we present related work including commercially available design tools to generate hardware, previously published papers for domain-specific design tools and multi-core platforms. In section 3, we introduce GUSTO, its design flow and describe the optimizations performed to generate an application specific core from a user defined general purpose core. Our optimizations include static architecture generation and trimming for optimization. Section 4 details the multi-core architecture generation by introducing GUSTO's steps: partitioning and generation of the connectivity between cores. We conclude in Section 5.

6.1 Related Work

In this section, we consider **1)** multi-core platforms that are specifically designed to exploit inherent parallelism in the given algorithms and **2)** commercially available design tools to generate hardware from a higher level language and

previously published papers to advance their technology.

There are different platforms that employ various types of computational cores on a single die to achieve higher throughput. Some examples to these platforms are Graphical Processor Units (GPUs) and Massively Parallel Processor Arrays (MPPAs), each of which has a different type of architectural organization, processor type and memory management. GPUs and MPPAs employ 960 cores (NVIDIA Tesla S1070) and 336 cores (Ambric AM2045) in their architectures respectively. This large number of cores provides massive parallelism, but it is also very hard to partition the given algorithm onto hundreds of processors. These platforms employ general purpose processors with full connectivity and predefined local memories and cache sizes.

High level design tools are attractive to achieve optimized architectures for the application at hand while decreasing the design time. There are various different design tools that can aid a designer to generate hardware from a higher level language such as MATLAB and C. Matrix computation algorithms are typically written and tested using MATLAB code or Simulink, a model based environment, and there are a number of tools that translate such algorithms to a hardware description language. Model based design tools include System Generator from Xilinx, Simulink HDL Coder from Mathworks and Synplify DSP from Synplicity. The drawbacks of model based design tools are: *1)* The generation of control units requires manual effort and in particular, resource synchronization becomes more complicated as the application complexity increases, *2)* the lack of built-in IP block support and *3)* inefficient performance since designed IP cores are treated as black boxes by current synthesis tools; therefore current synthesis tools do not perform optimization that cross module boundaries.

A solution to the synchronization problem and the lack of built-in block support is to use AccelDSP from Xilinx which provides MATLAB code, *.m*, to HDL conversion automatically. Therefore, a user can design controller units and/or undefined blocks using AccelDSP easily. However, Nissbrandt et al. [156] compared AccelDSP and AccelWare DSP library results with hand-code and Xilinx Core Generator implementations for various signal processing algorithms. The au-

thors concluded that AccelDSP should primarily be used as a method for platform decision due to inefficiencies in the AccelDSP generated designs; performance, reliability of the description, readability and the problems of maintaining the source are common problems. All of which are more important than the reduced design time. Furthermore, Xilinx announced that AccelDSP will be discontinued starting from 2010, thus creating a need for MATLAB code, *.m*, based design tool [157].

Examples of C-based design tools are Catapult-C, Forte Cynthesizer [167] and the PICO Project [168]. Catapult-C and Forte Cynthesizer perform simultaneous architecture generation for the data path and the controller where the datapath is built *on the fly* and affected by the control generation. These tools generate FSMs as controllers and perform the best on small applications (those with less code) [169]. PICO generated architectures consist of a EPIC/VLIW processor and an optional nonprogrammable accelerator (NPA) subsystem where it employs only synchronous parallel nearest-neighbor communication. Another example is OptimoDE [175] that consists of a VLIW-styled Data Engine architecture allowing data path configurations and ISA customizations. To generate pipelines of accelerators from streaming applications, Mahlke et al. proposed an automated system called Streamroller [177]. However, VLIW datapaths do not scale efficiently and there has been extensive research on multicluster architecture generation [170]. Our methodology is significantly different than these previous approaches and can be seen as a custom MPPA design. General purpose processing cores, generated by GUSTO, are simple RISC CPUs where the user defines the number as well as the types of the processing cores. After the optimizatons, each processing core employs required connectivity between its architectural components and only required amounts of memory resources.

Optimization of a processor by removing unused and/or underutilized resources has also been studied. Fan et al. proposed a method to remove underutilized bypass paths to optimize the processor architectures that employ register bypassing logic [173]. Optimization of a processor by trimming is presented in [164,165] where the trimming is performed for only unused functional resources or limited to an architecture style. Gajski et al. presented a methodology for design-

ing energy-efficient processing-elements, that use statically scheduled nanocode-based architectures, by extending trimming method for the optimization of interconnects [166]. The drawback of their approach is the large code size of the parallel nanocoded architecture which is up to four times larger than a RISC processor. GUSTO's flow also does not require updating the instruction scheduler due to the removed functional resources and interconnects and extends trimming optimization to multi-core architectures by optimizing the connectivity between processing cores.

There has been extensive research on efficient partitioning of operations. Exploiting ILP leads to high performance values while centralized memory becomes bottleneck in the architectures. A strategy to eliminate the bottleneck is to create a decentralized architecture using several smaller register files, each serving to particular FUs [171]. These types of architectures are known as clustered architecture and their main challenge is the compilation support for efficient partitioning operations into available resources. Chu et al. presented a methodology to solve efficient partitioning based on graph partitioning methods using a performance-based operation partitioning algorithm [172]. [174] extends their performance-based approach to another level by effectively balancing cost and performance. [176] proposed a method for synthesizing the local memory architecture by breaking it into simple sub-problems and solving it with using a phase-ordered approach. An efficient memory access partitioning method that partitions memory operations across cores to decrease the memory stall time is presented in [178]. GUSTO generates processing cores that have local memories, however gives partitioning responsibilites to the user.

6.2 Automatic Generation and Optimization of Matrix Computation Architectures

There are many architectural design choices that need to be made while implementing the hardware for matrix computation algorithms. These implementation choices include resource allocation, number of functional units, organization

of controllers and interconnects, and bit widths of the data. Not only does generating the hardware for a given set of requirements involve tedious work, but performing design space exploration to find the optimum hardware design is also a time consuming process. Therefore, a high level tool for design space exploration and fast prototyping is essential.

A general purpose processing core is an ideal starting point to create an application specific processing core since it provides us with fully connected highly custom architecture and there are substantial opportunities to eliminate unneeded functionality. This includes unused registers, register ports, functional units, as well as the interconnect that have substantial overhead on performance. We use GUSTO to optimize general purpose processing cores to have only the required connectivity between its components for a given part of the algorithm. Therefore, it is not surprising to see that there is large amount of area savings and throughput increase by moving to application specific processing cores. Even though general purpose processing cores require substantial overhead in order to maintain the flexibility to execute any application, there are of course many benefits for employing general purpose processing cores in a multi-core architecture. The main goals of this chapter are to show GUSTO's methodology in more detail and provide an idea to the user about the cost of employing a general purpose processing core in the multi-core architecture generated by GUSTO.

In this section we present the flow of operation, describe the general purpose processing cores generated, and discuss optimizations performed in each core by GUSTO when generating matrix computation architectures. The next section introduces design flow steps for the multi-core architecture generation in more detail. To the best of our knowledge we are the first to propose such an automated design tool.

6.2.1 Flow of Operation

GUSTO is a high level design tool that provides automatic generation and optimization of a variety of general purpose processing cores with different parameterization options. It then optimizes the general purpose processing cores to

improve area efficiency and design quality which results in an application specific core. As shown in Figure 6.1, GUSTO first receives the algorithm from the user and allows him/her to perform partitioning. A user then chooses the type and number of arithmetic resources and the data representation for each core. GUSTO automatically generates optimized application specific cores and combines these small cores to create an architecture for a given matrix computation algorithm. The application specific architectures that are generated by GUSTO employ the optimal number of resources which maximizes the throughput while minimizing area.

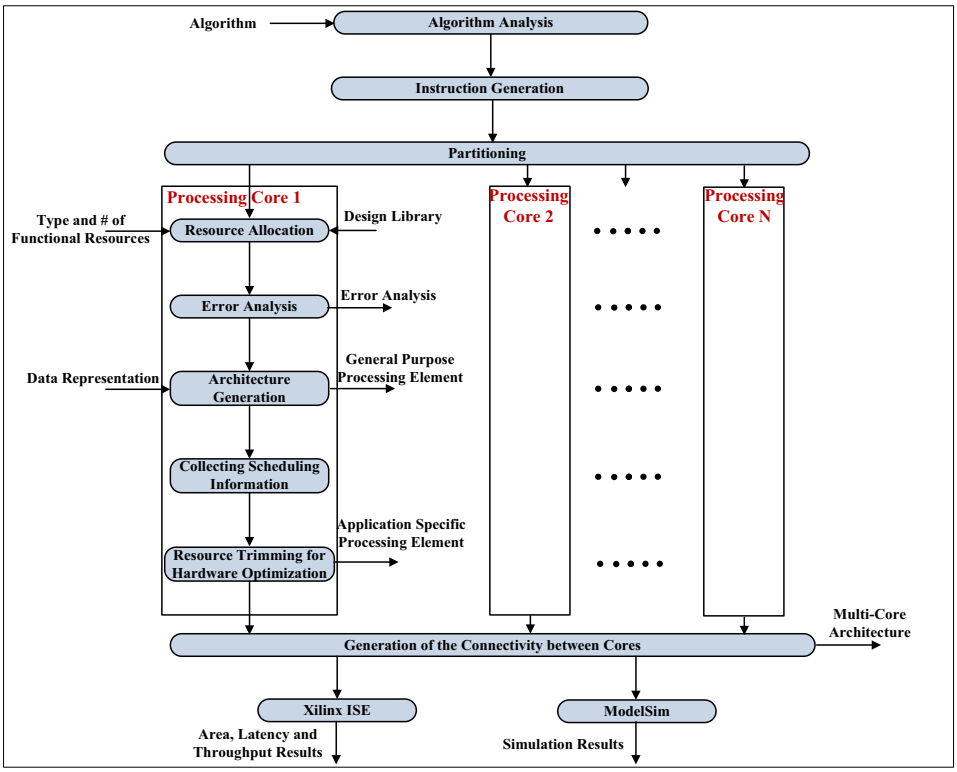


Figure 6.1: **Design Flow of GUSTO.**

6.2.2 Designing the General Purpose Processing Core

In the architecture generation step, GUSTO creates a general purpose processing core which exploits instruction level parallelism using dynamically sched-

uled out-of-order execution. The general purpose processing core consists of an instruction scheduler, a memory controller and functional resources. *The instruction scheduler* reads instructions that are pre-generated by GUSTO from its instruction memory. The format of a fetched instruction defines the operation type to be performed, the destination and two required operands for the calculation. The main duty of the instruction scheduler is to generate scheduled instructions i.e., assigning operations to the functional resources, performing scheduling and binding. This is achieved by **1)** tracking the availability of the functional resources and **2)** tracking the functional units that will produce the operand(s) (Figure 6.2(a)). The instruction scheduler prevents WAR and WAW hazards as well as structural hazards. Each *functional resource* receives the scheduled instructions and waits for the required operands for its execution. The required operands can be received either from memory through its controller or from functional resources, shown as *interconnect matrix*. *The memory controller* resides the memory, watches for the data, updates its memory entries and prevents RAW hazards (Figure 6.2(b)). GUSTO generated architectures can define register files or BRAMs as memory elements which depends on the required number of inputs/outputs for the memory and the required memory size.

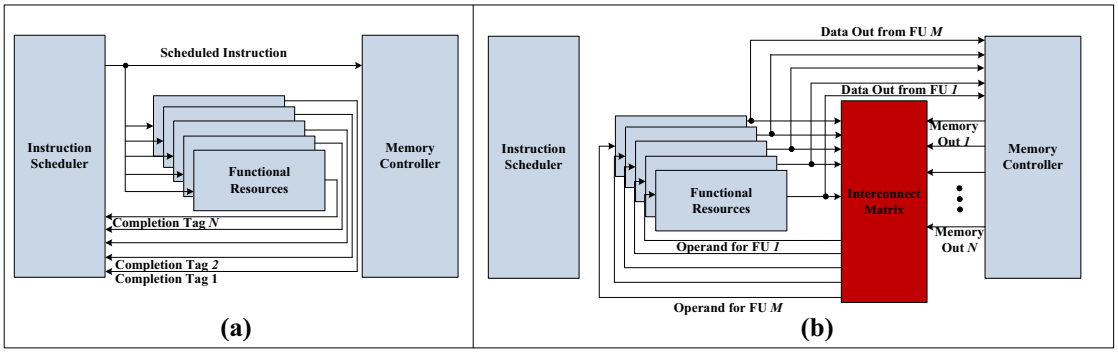


Figure 6.2: **(a) The instruction scheduler generates scheduled instructions i.e., assigning operations to the functional resources, performing scheduling and binding. (b) Each functional resource receives scheduled instructions and waits for the required operands to begin execution.**

6.2.3 Designing the Application Specific processing core

GUSTO performs several optimizations on the general purpose architecture and creates application specific processing cores while ensuring that the correctness of the solution is maintained. We present details about our optimizations in the following subsections.

Optimizing the Instruction Scheduler

The instruction scheduler generates scheduled instructions from fetched instructions by scheduling and binding the arithmetic operations. The instruction scheduler prevents WAR and WAW hazards as well as structural hazards through tracking the availability of functional resources and the operands 6.3. The instruction scheduler has a crucial role while implementing a dynamic scheduling processor architecture with out-of-order execution capability, therefore it consumes a lot of silicon in the overall architecture. However, we show that the dynamic instruction scheduler can be optimized to create an efficient application specific processing core.

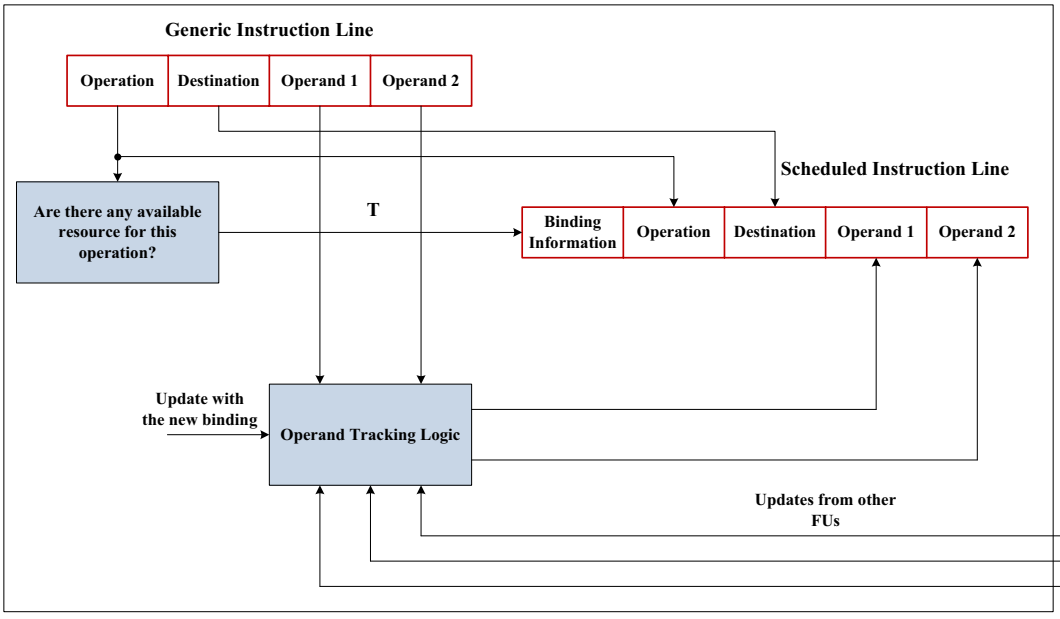


Figure 6.3: **Detailed Architecture for the Instruction Scheduler.**

There are two variables that have a significant effect on the instruction scheduler's silicon usage: **number of functional resources** (to track the availability of resources) and **number of memory entries** (to track the location of the operands for scheduled instructions that will be produced from one of the functional resources). We perform two analyses to determine the effect of these two variables on GUSTO generated general purpose processing cores in terms of area (Slices) and the critical path (ns). We use FPGAs to perform these analyses but could also target ASICs. Figure 6.4(a) presents the increase in the area and critical path *with the increasing number of functional resources* where we assume that the memory has 20 entries. Figure 6.4(b) presents the increase in the area and the critical path while the size of the memory entries increases from 20 to 100, assuming that there are 8 functional resources in the processor architecture. We determine that increasing the number of functional resources and memory entries dramatically affects the area and critical path of the architecture. Therefore, we concentrate our efforts on removing the dynamic binding process (tracking the availability of functional resources) and the tracking unit for operands.

We explore two different possible optimization methods for the instruction scheduler:

- **Removing the instruction scheduler** by creating a static architecture (such as finite state machines (FSMs) as controllers) for functional resources and memory resources. GUSTO is capable of removing the instruction scheduler and creating a datapath for execution of the given algorithm through a set of functional resources, memory resources and FSM controllers. However, we prefer not to use this method since FSMs usually grow in complexity as the required number of clock cycles for the execution of the application increases (especially with the tools that we used);

- **By simulating the general purpose processing core with the fetched instructions and collecting the scheduled instructions,** GUSTO can remove the binding and the operand tracking logic since this information is already included in the scheduled instructions. What is not included is the time to send the scheduled instructions to the other resources which

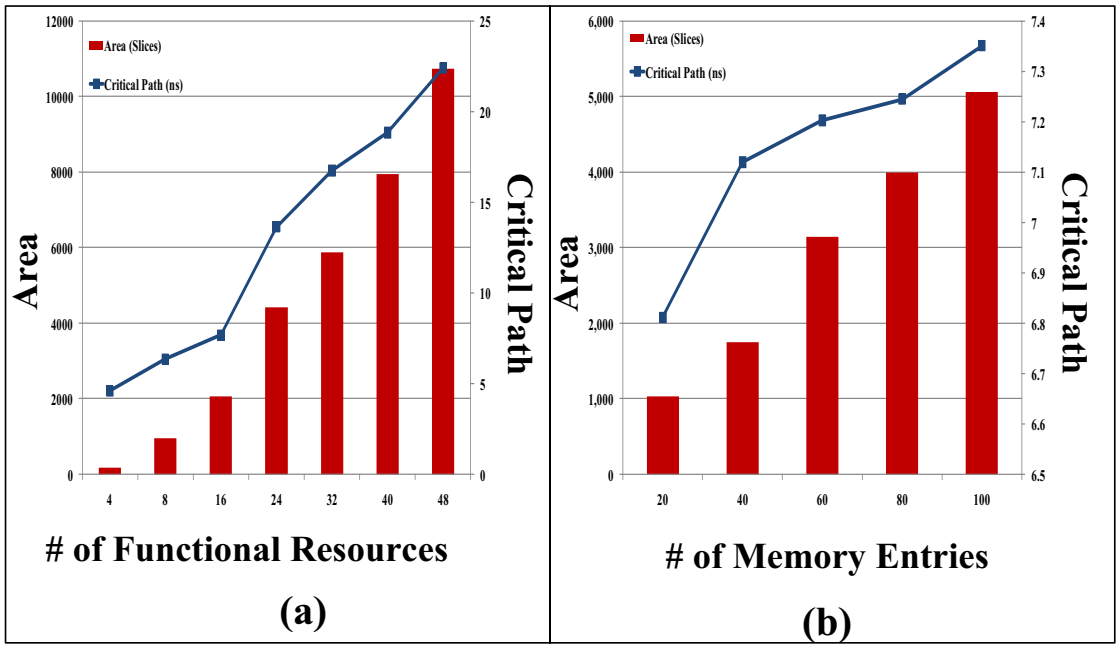


Figure 6.4: **(a) Increasing the number of functional resources results in an increase in the area (Slices) and the critical path where we assume that the memory has 20 entries. (b) Increasing the size of the memory entries from 20 to 100 results in an increase in the area (Slices) and the critical path where we assume that there are 8 functional resources.**

can easily be performed with the functional resource availability. Therefore, GUSTO generates a processor architecture that uses static binding and dynamic scheduling of resources. This approach gives us the opportunity to perform various optimizations on the other resources instead of generating a fully static architecture.

Figure 6.5 presents a comparison between unoptimized (Unopt.) and optimized (Opt.) instruction scheduler architectures in terms of area (Slices) and critical path (ns) with the increasing number of functional resources. As can be seen from Figure 6.5, we keep the complexity of the instruction scheduler steady even with the increasing number of functional resources by saving tremendous silicon (83%-97%). We achieve this optimization by avoiding dynamic binding and operand tracking which are very costly in terms of architectural implementation.

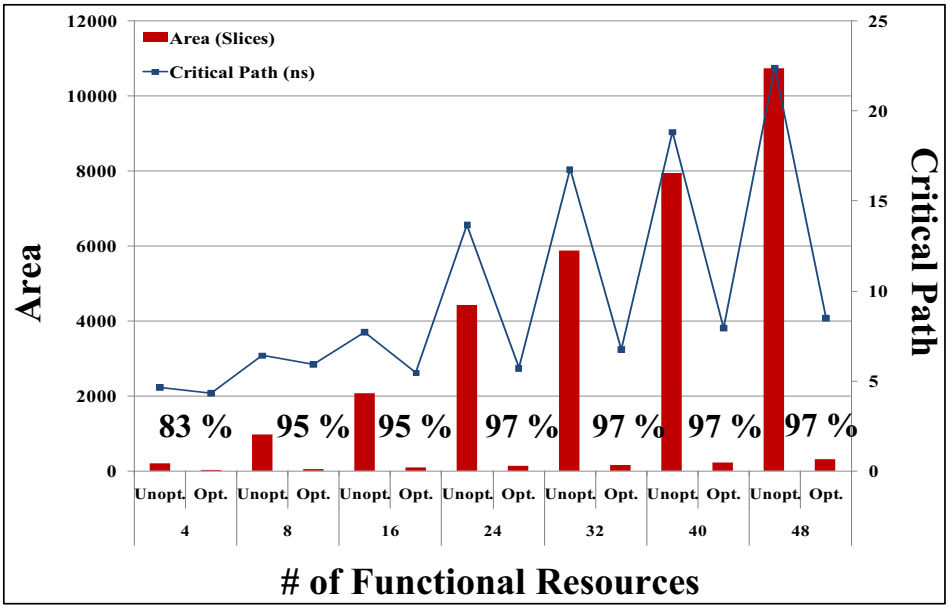


Figure 6.5: **A comparison between unoptimized (Unopt.) and optimized (Opt.) instruction scheduler architectures in terms of area (Slices) and critical path (ns) with the increasing number of functional resources.**

Optimizing the Functional Resources

After the instruction scheduler decodes the fetched instruction and assigns a free functional resource for the calculation, the scheduled instruction is sent to every other functional resource. The assigned arithmetic resource receives the scheduled instruction, starts monitoring the common data bus, and goes into the execution stage when all the required operands are available. The required operands can be received from either outputs of a functional resource or memory element.

Each functional resource consists of an arithmetic calculation unit (adder, matrix multiplier, etc.) and the controllers to track required operands for the calculation which is shown in Figure 6.6. Employing a high number of utilized functional resources in a core provides higher throughput which also increases the silicon consumption. For better understanding the distribution of the silicon consumption in arithmetic calculation units and controllers, we investigate their distribution in terms of area while employing an increasing number of functional resources as shown in Figure 6.7(a). As can be seen from the Figure, most of the

silicon (61%-78%) is consumed as controllers in a functional resource in order to be able to track the operands.

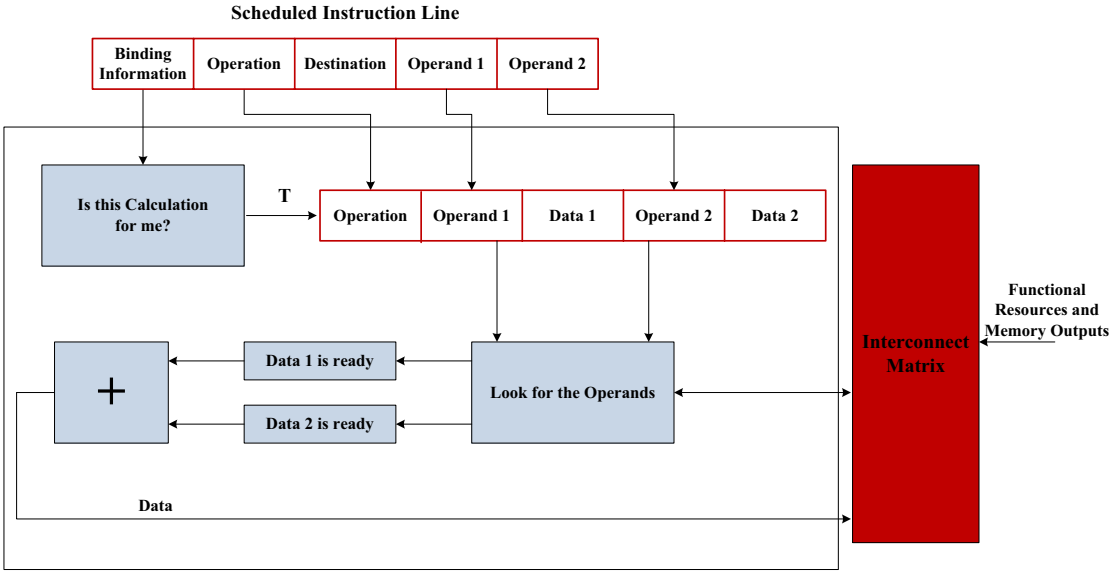


Figure 6.6: **Detailed Architecture for a Functional Resource.**

The possible optimizations to reduce the controller overhead include:

- **Generation of a static controller.** The complexity of the generated controllers increases dramatically for more complex algorithms, therefore careful FSM optimizations need to be performed to achieve low area results;

- **Trimming for control units.** GUSTO simulates the general purpose architecture to define the usage of arithmetic units, multiplexers, register entries and input/output ports and trims away the unused components with their interconnects. GUSTO defines an optimization matrix for each functional resource employed in the processing core that shows the connections with respect to other functional resources. These optimization matrices are used to trim away unused resources with its interconnects and therefore lead to a processing core which has only the required functional resources and connectivity for the given algorithm.

Figure 6.7(b) presents the optimization results for functional resources using

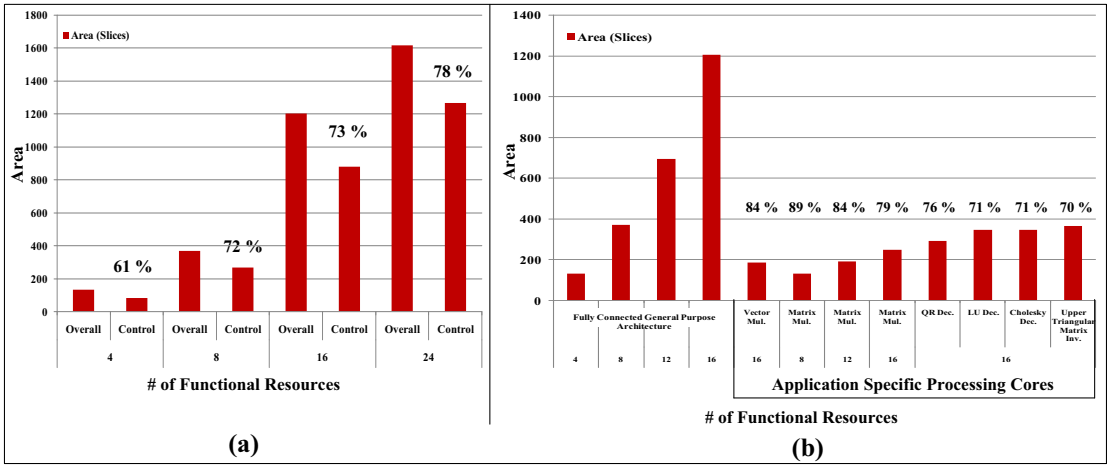


Figure 6.7: **(a) shows that controllers in a functional resource consumes most of the silicon (61%-78%) in order to be able to track the operands. (b) presents the optimization results for functional resources using trimming optimizations.**

trimming optimizations. The first four bars in the figure show the area results for unoptimized general purpose functional resources as the number of functional resources increases. The area of the functional resources increases significantly with the addition of the new resources into the architecture. The remaining bars show the area for optimized functional resources using different applications. Trimming performance depends on the applications's simplicity and regularity. For example, vector multiplication is the simplest among our examples. Our results show 84% area reduction compared to the equivalent general purpose architecture with 16 functional resources. Another example is the matrix multiplication which is more complex and requires more calculations compared to the vector multiplication. We use GUSTO to show different area results of matrix multiplication with different number of functional resources. Results for QR, LU and Cholesky decompositions and upper triangular matrix inversion are also shown in Figure 6.7(b). Towards upper triangular matrix inversion algorithm, optimization results drop to 70% and trimming optimization becomes less effective. The reason behind this is that the complexity and irregularity of the algorithms increase.

It is important to state one more important feature of the GUSTOs' trimming optimization. While simulating the general purpose architecture, GUSTO

creates different usage frequency graphs for the functional resources and their interconnects: **1) Usage frequency graphs for the functional resources:** a user sees the utilization of the functional resources in order to determine underutilized resources. As an example, assume that there are 10 adders in a core and one of the adders is being used just 2 times compared to the rest of the adders (used more than 20 times each). The user can remove this underutilized adder which will force dynamic scheduling and the binding process to assign its calculations to another adder. **2) Usage frequency graphs for the interconnects:** a user sees the utilization of the interconnects in order to determine underutilized interconnects and performs different scheduling methods to eliminate the underutilized interconnects. The benefit from these binding approaches is to save significant amount of area from underutilized resources/interconnects and there may be also a decrease in the throughput of the architecture.

Optimizing the Memory Controller

After receiving the scheduled instruction from the instruction scheduler, the memory controller checks if the required operands are in the memory entries and if they are, it sends them to the functional resources. The memory controller also keeps the destination information from the scheduled instruction, thus it does not send the data until its execution is completed and the destination is updated with the new value to prevent RAW hazards (Figure 6.8).

GUSTO creates a static architecture for the memory controller by collecting the memory assignments (inputs and outputs) with the clock cycle. Therefore, GUSTO uses this information to remove dynamic assignments and create a stateless Moore FSM to control the memory assignments. We are currently working on optimizing the FSMs that are generated by GUSTO by finding sequences of control signal patterns and reducing the area by using counter logic instead of just linear states. However, these optimizations are not in the scope of this article.

Figure 6.9 shows the area (slices) of the memory controller and required clock cycles to execute for the given matrix computation algorithm using a different number of functional resources. The first three bars show the dynamic

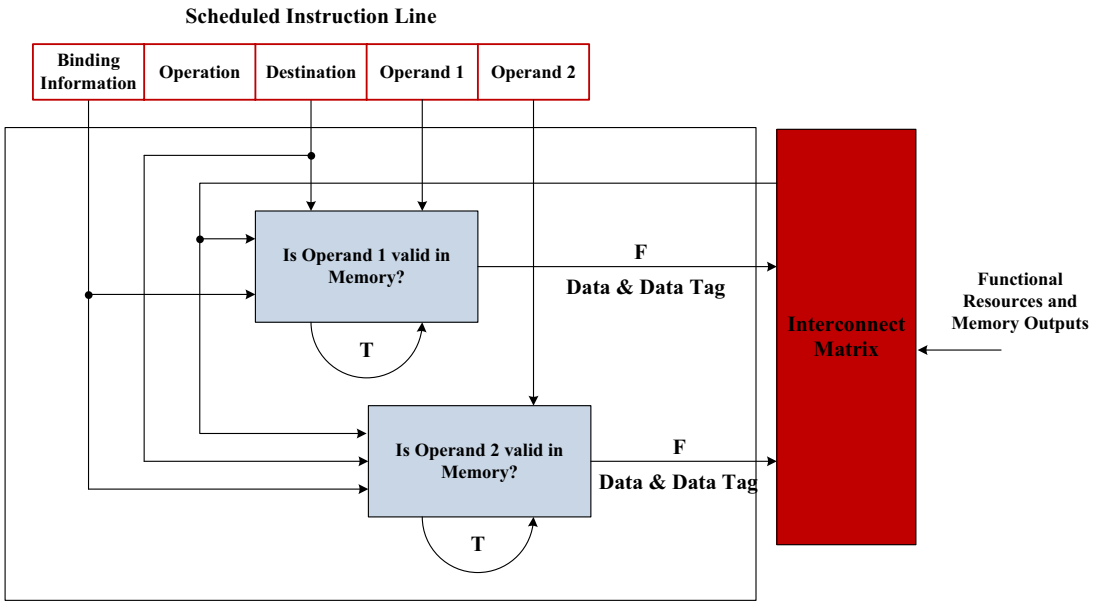


Figure 6.8: **Detailed Architecture for the Memory Controller.**

general purpose memory controller area results with a different number of functional resources. The area of the memory controller increases significantly with the increasing number of functional resources. The required clock cycles for the general purpose architecture is not shown since this value is different for each matrix computation algorithm. We use GUSTO to optimize the memory controller for different matrix computation algorithms such as vector multiplication, matrix multiplication, LU decomposition, etc. GUSTO generated static memory controllers provide large area savings. The required clock cycles to execute the given matrix computation algorithms directly affects the area of the static architecture. As can be seen from the Figure 6.9, the area increases while the required clock cycles to execute increases.

Figure 6.10 shows the area (slices) in the y-axis and percentage inside the bar describing the distribution of arithmetic resources and controllers for a single core design. GUSTO provides single core architectures that consumes most of the area in arithmetic resources such as 87% and 85% for Cholesky decomposition and upper triangular matrix inversion algorithms respectively. This distribution changes with the complexity and irregularity of the algorithms such that LU

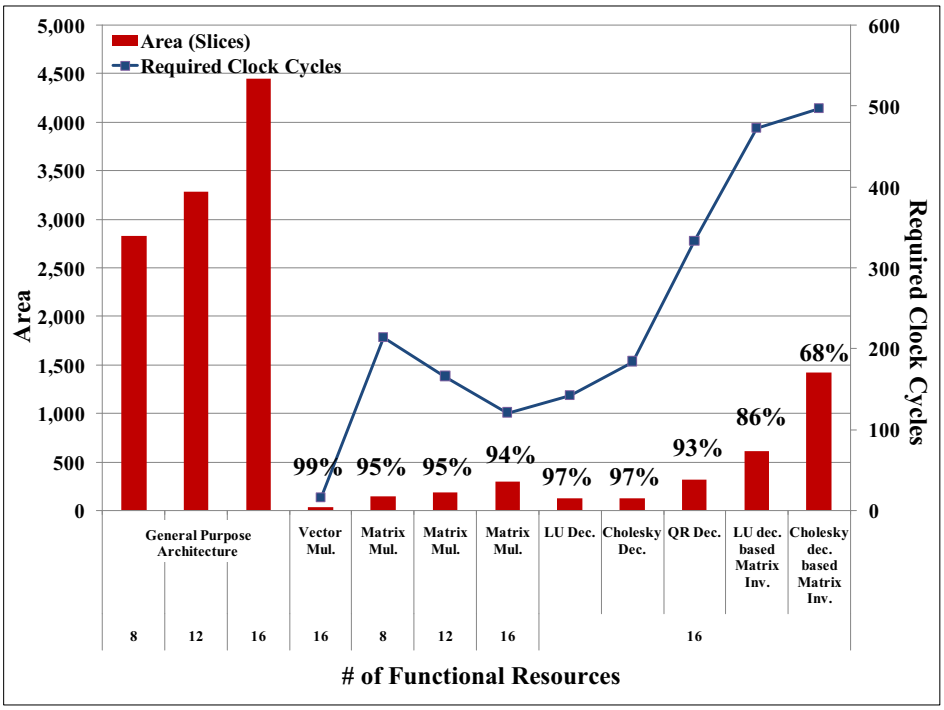


Figure 6.9: **Shows the area results for the memory controller and required clock cycles to execute the given matrix computation algorithm using different number of functional resources.**

decomposition and Cholesky decomposition based matrix inversion architectures consume larger area in the controllers.

6.3 Designing a Multi-Core Architecture

Designing a single processing core for matrix computations does not provide a complete design space to the user because the user is only capable of changing the number of functional resources in terms of the organization of the design and concentrating on the instruction level parallelism without task level parallelism. Since GUSTO finds the optimal number of resources for a single core, which maximizes the throughput while minimizing area, it is not possible to achieve a higher throughput. Designing a single core for matrix computations also does not scale well with the complexity of the algorithms typically required to take advantage of the instruction level parallelism present. The two major reasons for these scalabil-

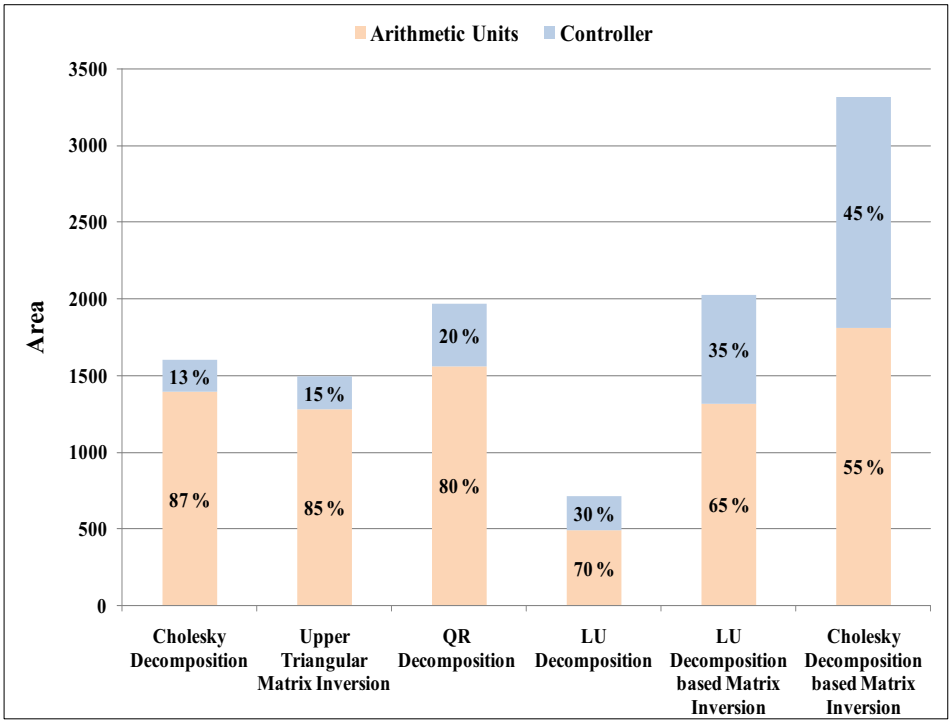


Figure 6.10: **Shows the percentage distribution of arithmetic resources and controllers for a single core design.**

ity issues are: **1)** The internal storage and communication between functional units becomes dominated by the increasing number of functional resources in terms of delay, power dissipation, and area [159], **2)** Because of the increase in the usage of functional units, the optimization that is performed by GUSTO, to remove the unused arithmetic units, multiplexers and input/output ports with their interconnects, becomes less effective.

Even though moving from general purpose processing cores to application specific processing cores provides tremendous area and throughput improvements due to the performed optimizations, the complexity of the given algorithm directly affects these optimization results. For example, a single application specific processing core design for LU decomposition provides 94% area savings compared to general purpose processing core implementation. A single application specific processing core design for a more complex algorithm like LU decomposition based matrix inversion provides 77% area savings. The effectiveness of the optimiza-

tion decreases with the complexity of algorithms. Therefore, instead of creating one application specific processing core for the entire algorithm, generating multi-core architectures with homogeneous/heteregenous cores to exploit instruction and task level parallelism will result in more detailed design space exploration and may result in a more efficient hardware implementation.

Multi-core architecture generation requires us to introduce the following steps in more detail:

- *Partitioning* is the process of dividing the given algorithm into parallelizable parts to be implemented in different cores;
- *Generation of the connectivity between cores* is the process to connect the cores together with the required connectivity.

As shown in Figure 6.1, GUSTO starts with algorithm analysis, instruction generation and partitioning for multi-core implementation. GUSTO allows the user to define the desired parallelism in the partitioning step and generates general purpose processing cores for each part of the algorithm after the user defines the type and number of arithmetic resources and the data representation (the integer and fractional bit width). After the architecture generation step, GUSTO performs different optimizations to every core independent from each other and creates application specific processing cores. The last step is generation of the connectivity between cores for multi-core architecture. The *Partitioning* and *generation of the connectivity between cores* steps are introduced in more detail in the following subsections.

6.3.1 Partitioning

Partitioning is one of the most important steps, when designing a multi-core architecture using GUSTO, to exploit task level parallelism. There are many different ways to design a multi-core architecture by partitioning the given algorithm into different cores. Examples include **1)** design of a platform that employs large number of cores to achieve a high throughput but large area and **2)** design of a platform that employs small number of cores to achieve a small area but low

throughput. The quality of the design also depends on the inherent parallelism of the given algorithm independent from the user decisions. Our tool provides a large freedom to the user to create different types of parallel architectures and to compare them for choosing the one that fits to his/her situation.

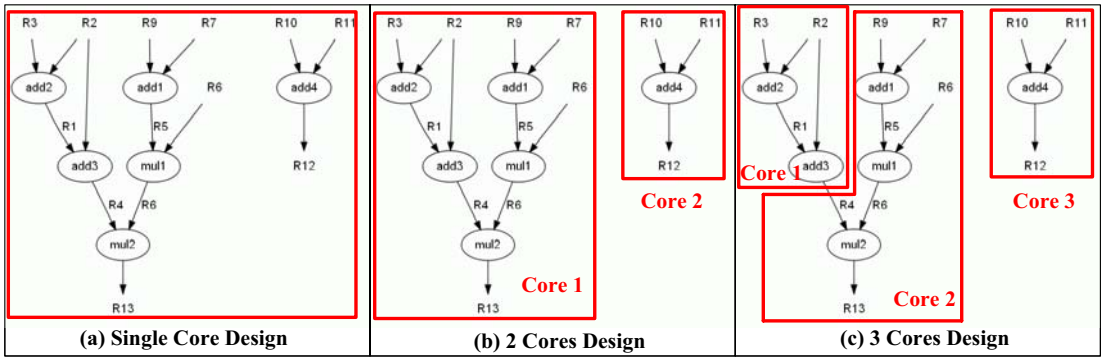


Figure 6.11: **There are several different ways that one can partition the given algorithm into different cores. We show three different possibilities: (a), (b), (c), that are examples of designing one core, two cores and three cores for a given matrix computation algorithm respectively.**

After the algorithm analysis and instruction generation steps, GUSTO produces a data flow graph representing the data dependencies and the parallelism options to the user. As a very simple example, consider the data flow graph that is generated by GUSTO in Figure 6.11. There are several different ways that one can partition the algorithm into different cores. Therefore we show three different possibilities in Figure 6.11 where (a) presents 1 core design to execute all instructions in one core and (a) does not require the execution of *generation of the connectivity between cores* step of the GUSTO's flow. Unlike (a), (b) and (c) show different ways to exploit the parallelism in the given algorithm using 2 or 3 cores. Even though (b) is a multi-core architecture, there is no connectivity needed between cores due to the fact that there are no data dependencies between them. There is no need to perform *the generation of the connectivity between cores* for (b). On the other hand (c) generates a multi-core architecture where there is data dependency between its core 1 and core 2. Therefore, GUSTO performs *the generation of the connectivity between cores* step to make core 2 able to access core 1 to get the data *R4* when it is valid. The *generation of the connectivity between cores* step will be

introduced in more detail in the later subsection of this section.

6.3.2 Generation of the Connectivity Between Cores

GUSTO generates general purpose processing cores with the user-defined inputs. Furthermore, GUSTO performs several optimizations to generate application specific cores where a user can choose to keep some general purpose processing cores in the architecture. The application specific cores have only the required connectivity internally which is shown in Figure 6.12(a). Each row in the platform shows the cores that have the data dependency to each other. These cores process the data in a streaming fashion. As an example, Application Specific Core, *ASP 1,1*, processes the data and another core in its row can start execution upon the completion of its required inputs. This architecture employs a shared memory structure where each core has access to the specific parts of the memory, defined as **local** and **shared** variables. Local variables are specific to a core; shared variables are used between a core and its successors. As an example, *ASP 1,1* uses its local memory to execute given instructions, and puts the data, that are required for *ASP 1,x*'s execution, into the shared memory; therefore *ASP 1,x* can start its execution. Each row in the architecture executes independent from each other since they have no data dependencies. However, using shared memory is not the most efficient solution for multi-core architectures since the complexity of the controllers increases dramatically.

GUSTO partitions the shared memory into each processing core in such a way that the memory accesses between cores are reduced. In a fully connected architecture, there are two different types of connectivity: *1) Instruction scheduler to other cores's local memory.* Scheduled instructions need to be broadcasted in a way that each core sends the instruction to its predecessors since they may access data from a predecessors's shared variables. *2) each shared memory to every functional resource in its row.* There should be full connectivity between each shared memory and every functional resource in its row. However, full connectivity might not be required in application specific architectures. Therefore, GUSTO removes the connections, *instruction scheduler to other cores' memory controller*

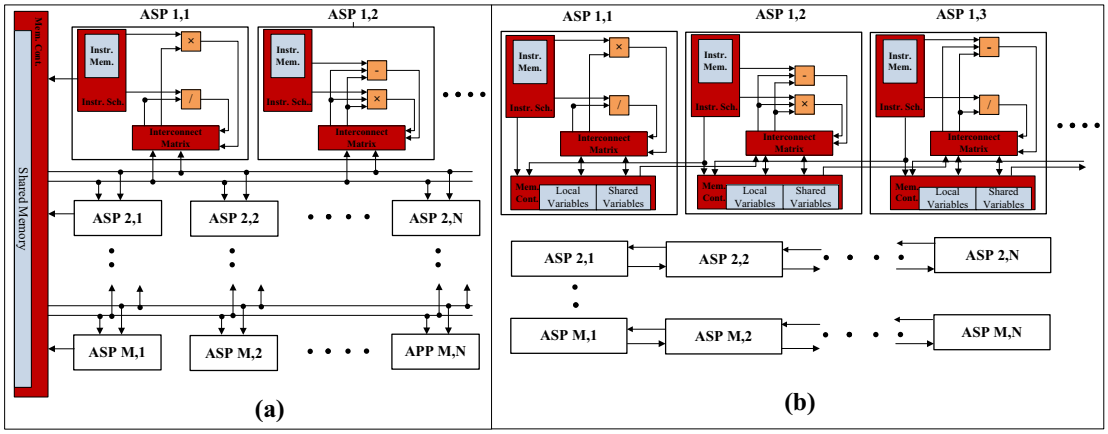


Figure 6.12: **(a) GUSTO the optimizes general purpose processing elements to generate application specific processing cores that use a shared memory. (b) GUSTO partitions the data into individual processing cores. The data is classified as local variables, that used only for that core, and shared variables that are written only once and used by the next processing core.**

and *shared memory to every functional resource in another core*, if there is no data dependecy between two cores. There also might be a data dependency between two cores, however not all the interconnects between shared memory and functional resources is used. GUSTO also removes these unused interconnects resulting in a platform which has only the required connectivity between processing cores.

6.4 Conclusion

This chapter details the automatic multi-core architecture generation and optimization methods for matrix computation algorithms. Our tool, GUSTO, is developed to enable easy design space exploration by providing different design methods and parameterization options which enable us to study area and performance tradeoffs over a large number of different architectures.

Chapter 7

Hardware Implementation Trade-offs of Matrix Computation Architectures using Hierarchical Datapaths

Matrix computations is a topic of great interest in numerical linear algebra. Since many of these matrix computation algorithms are computationally expensive and memory demanding tasks, it is an area where there is much interest in parallel architectures. There has been extensive research for the design of a platform that can process the same amount of data in a smaller amount of time or handle a larger amount of data in the same amount of time. Due to the decrease in the clock frequency of processors, the new trend is to provide various types of computational cores on a single die. Examples to these platforms are Chip Multiprocessors (CMPs), Graphical Processor Units (GPUs), Massively Parallel Processor Arrays (MPPAs) etc. where they have different types of architectural organizations, processor types, memory management etc.

Single core generation using GUSTO for a given application provides the following advantages:

- Automization process is a lot simpler from the tool development side since

tool does not need to have to generate connections between different number and types of cores;

- Partitioning process is simpler since an user does not need to worry about dividing the given algorithms onto different number and types of processors efficiently.

However if a user wants to design and implement a platform that exploits large amount of parallelism that leads to very high throughput implementation of algorithms, generation of multi-core architectures is essential, especially while implementing highly parallelizable matrix computation algorithms in hardware (which is the main topic of this thesis). Furthermore, there are several disadvantages for single core design for a given application:

- The datapaths of the processing elements that are generated by GUSTO are based on a single memory which is shared by all functional units. The functional units, with the number and type specified by the user, are employed for the computation of the given algorithm using resource sharing. Unfortunately this simple organization of the architecture does not provide a complete design space to the user for exploring better design alternatives since the user is only capable of changing the number of functional resources and number of instructions executed in a clock cycle in terms of the organization of the design.

- This simple organization also does not scale well with the complexity of the algorithms and focuses primarily on instruction level parallelism even with the implementation of superscalar PEs that can execute more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to redundant functional units on the processing element. There are two major reasons for these scalability issues:

  - As the number of functional units increases, internal storage and communication between functional units quickly becomes the bottleneck in terms of delay, power dissipation, and area [23]. It has been shown that

for N functional units connected to a register file, the area of the register file grows as N, the delay as $N^{3/2}$, and power dissipation as N^3 [24];

– The optimization performed by GUSTO to remove the unused arithmetic units, multiplexers and input/output ports with their interconnects becomes less effective due to the increased usage of functional units.

Therefore, we take our design tool, GUSTO, to another level by enabling multi-core architecture generation where a user can define **the number** as well as **the type** of the processors with required connectivity in processor architecture level as well as the platform level by incorporating the following options into GUSTO:

- Hierarchical datapaths implementation for multi-core architecture generation;
- Heterogeneous architecture generation to be capable of designing different type of cores in one platform.

GUSTO employs a different amount of memory resources for each processor in a way that the memory accesses between cores are reduced and the overall platform generated by GUSTO has only the required connectivity between processors. As explained in Chapter 6, processor architectures that are generated by GUSTO are simple but ALU oriented for high performance computation. **By using these new features, a user can divide the given algorithms into small highly parallelizable parts, generate hardware cores for each specific part of the algorithm and combine these small PEs with hierarchical datapaths using GUSTO to create a multi-core architecture. This provides us a more detailed design space exploration and more efficient hardware implementations; and enables us to exploit both instruction and task level parallelism for achieving higher performance/throughput.**

The major contributions of this chapter are:

1) Automatic generation and optimization of matrix computation architectures with

parameterized bit widths, resource allocation and architecture types which also support hierarchical datapaths for multi-core architecture generation;
2) Implementation of heterogeneous architectures to enlarge our design space exploration thereby giving the user more options to produce efficient and highly optimized designs;
3) Comparison of different matrix computations architectures such as matrix multiplication and inversion using various GUSTO parameters and design methods.

The rest of this chapter is organized as follows. In section 7.1, we introduce our tool as well as explain incorporating hierarchical datapaths for multi-core architecture generation and heterogeneous architecture generation. Section 7.2 presents our implementation results of matrix multiplication and inversion architectures in terms of area and throughput and compares our results with previously published work. We conclude in Section 7.3.

7.1 Hierarchical Datapaths Implementation and Heterogeneous Architecture Generation using GUSTO

There are many architectural design choices that need to be made while implementing the hardware for applications employing computationally intensive matrix computations such as in signal processing, computer vision and financial computations. These implementation choices include resource allocation, number of functional units, organization of controllers and interconnects, which is constrained by hardware design to offer time or area efficiency, and bit widths of the data which specify precision required. Not only is generating the hardware for given set of requirements involves tedious work, but performing design space exploration to find the optimum hardware design is also a time consuming process. Therefore, a high level tool for design space exploration and fast prototyping is essential and required. As introduced in the previous chapters, we designed a tool, GUSTO ("**G**eneral architecture design **U**tility and **S**ynthesis **T**ool for **O**ptimization"), that

provides automatic generation and optimization of a variety of general purpose PEs with different parameterization options. It also optimizes the general purpose PEs to improve area efficiency and design quality which results in an optimized application specific PE.

In this section, we present the required modifications to the flow of operation in GUSTO to be capable of designing multi-core architectures, then we describe the optimizations performed by our tool when generating hardware architectures. We further present the implementation of hierarchical datapaths and heterogeneous architectures using GUSTO. Therefore our tool allows the user to explore wider design choices and make better decisions with regards to design space and performance. To the best of our knowledge we are the first to propose such automated design tool.

7.1.1 Hardware Implementation Trade-offs of Matrix Computation Architectures using Hierarchical Datapaths

The organization of the architectures, that are generated by GUSTO, do not provide a complete design space to the user because the user is only capable of changing the number of functional resources in terms of the organization of the design. We show the application specific architecture results of matrix inversion algorithm using QR decomposition in Figure 7.1 with regards to increasing functional units. As can be seen from the figure, the area and throughput increase up to the optimal number of resources as the number of resources increase. However, adding more than the optimal number of resources decreases throughput while still increasing area. GUSTO finds the optimal number of resources which maximizes the throughput while minimizing the area while designing a core for the given algorithm. Therefore, it is not possible to achieve a higher throughput, smaller area, or to find the middle points of these design results using single core generation. Designing multi-core architectures using hierarchical datapaths provides opportunites to achieve smaller area results, higher throughput results and even better results in terms of both area and throughput compared to single core generation.

The application specific single core architectures, that are generated by

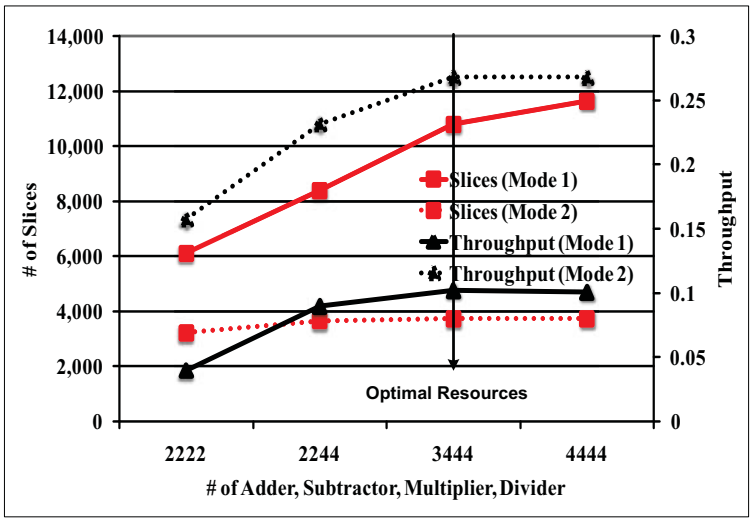


Figure 7.1: **Design space exploration using different resource allocation options for matrix inversion using QR decomposition.**

GUSTO, also do not scale well with the complexity of the algorithms and focus primarily on instruction level parallelism. The two major reasons for these scalability issues are:

- The internal storage and communication between functional units become dominated by the increasing number of functional resources in terms of delay, power dissipation, and area [23];
- Because of the increase in usage of functional units, the optimization that is performed by GUSTO, to remove the unused arithmetic units, multiplexers and input/output ports with their interconnects, becomes less effective.

Even though moving from general purpose PEs to application specific PEs provides tremendous area and throughput improvements due to performed optimizations, the complexity of the given algorithm and the organization of the architecture directly affect these optimization results. For example, sole decomposition methods: QR, LU and Cholesky, application specific PEs provide 83%, 94% and 86% area savings and 16%, 68% and 14% throughput increase respectively compared to general purpose PEs. However, such as for matrix inversion using decomposition methods provide 69%, 77% and 68% area savings by moving

to application specific PEs which is less area savings compared to decomposition algorithms due to matrix inversion algorithms' higher complexity which requires larger finite state machines in memory controller unit and more resource usage that may lead to ineffective trimming results.

Therefore, instead of creating one application specific core for the entire algorithm, we can divide the algorithm into different segments and create a homogeneous/heterogeneous architecture which includes smaller parallelizable PEs for each segment connected with hierarchical datapaths (Figure 7.2). An homogeneous architecture employs the same type of PEs whereas a heterogeneous architecture employs the same type of PEs along with different type of PEs. A homogeneous architecture provides simplicity in terms of organization for communication and distribution of the given algorithm since the algorithm is mapped onto identical PEs. A heterogeneous architecture, on the other hand, employs different type of PEs in its architectures that enlarges the design space provided with the price of more complex communication and algorithm mapping. As an example if there is a fully parallel architecture that processing time of the one type PE is larger compared the other PEs, the throughput of the architecture is determined with the PE that has the largest execution time as bottleneck. A heterogeneous architecture is an efficient or required solution for creating hardware for *not highly* parallelizable algorithms where a user cannot or do not want to employ identical PEs in the architecture. Thus, different design methods provided by GUSTO result in more detailed design space exploration and more efficient hardware implementations that enable us to exploit both instruction and task level parallelism.

7.1.2 Flow of GUSTO for Multi-Core Designs

Multi-core architecture generation requires us to introduce the following steps in more detail:

- *Partitioning* is the process of dividing the given algorithm into parallelizable parts to be implemented in different cores;
- *Generation of the connectivity between cores* is the process to connect the

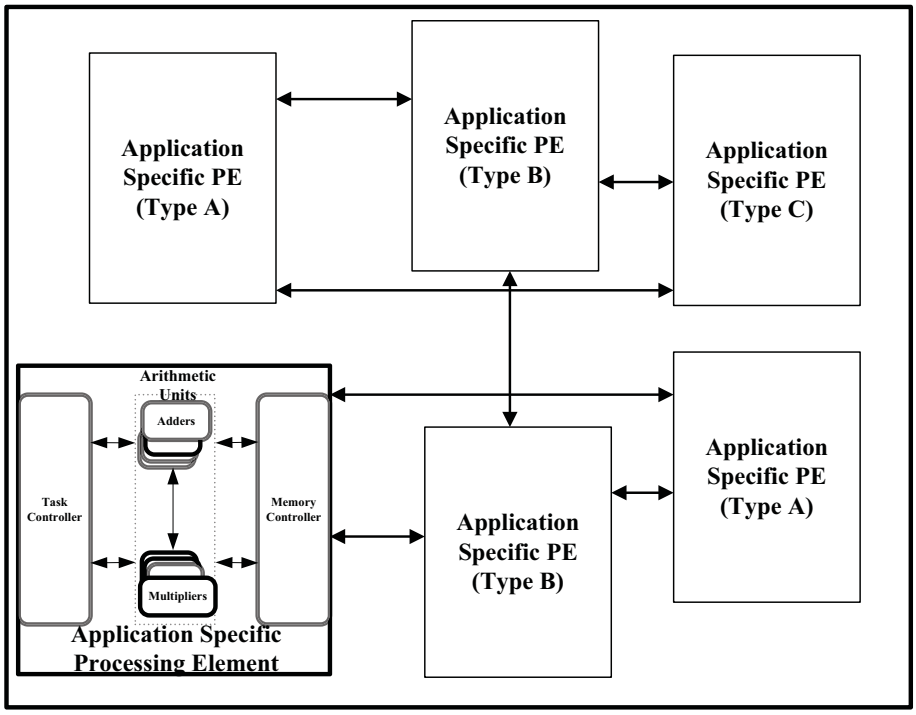


Figure 7.2: **Presents hierarchical datapath design for multi-core architecture using different number/type of application specific processing elements which execute a specific part of the given algorithm.**

cores together with the required connectivity.

As shown in Figure 7.3, GUSTO starts with algorithm analysis, instruction generation and partitioning for multi-core implementation. GUSTO allows the user to define the desired parallelism in partitioning step and generates general purpose processing elements for each part of the algorithm after user defines the type and number of arithmetic resources and the data representation (the integer and fractional bit width). After the architecture generation step, GUSTO performs different optimization to the every PE independent from each other, and creates application specific PEs. The last step is generation of the required connections between cores for multi-core architectures.

Partitioning is one of the most important steps while designing a multi-core architecture using GUSTO since the choices made determines the quality of the results. There are many different ways to design an multi-core architecture

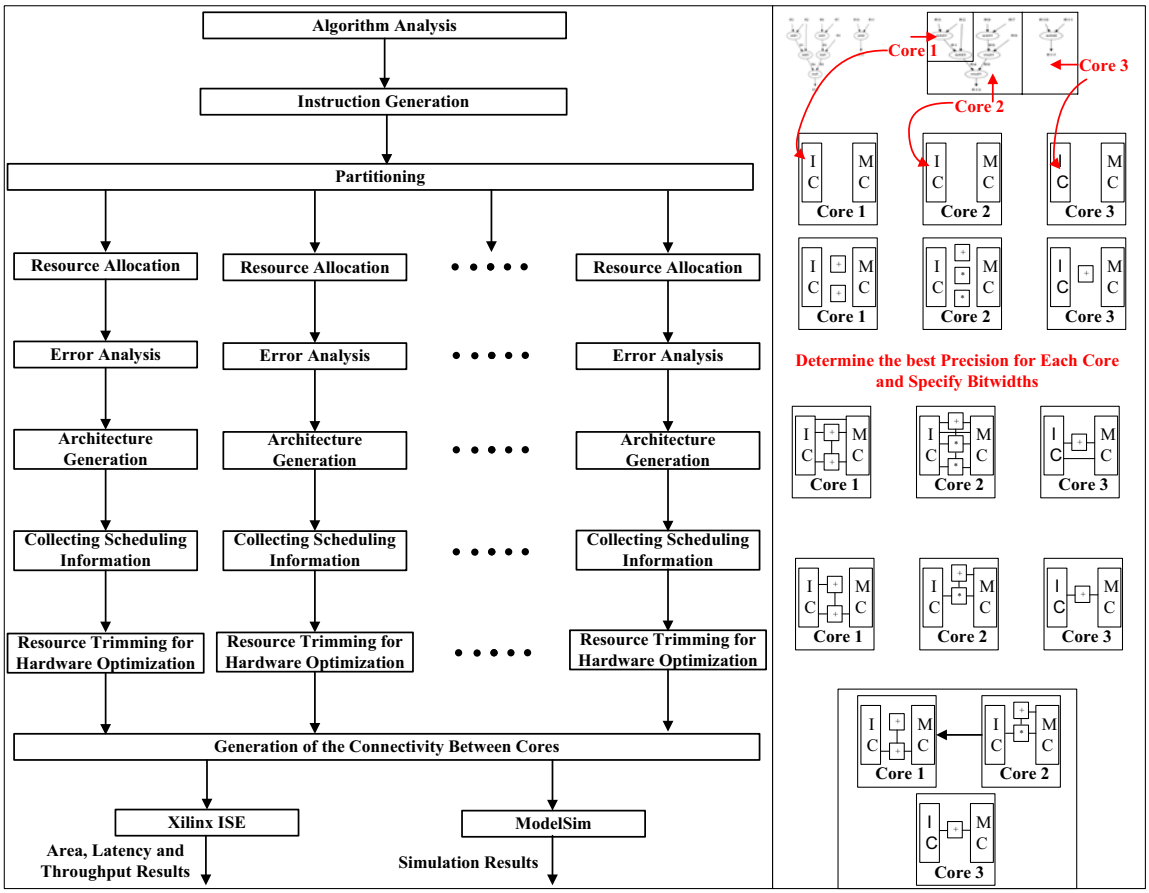


Figure 7.3: **Flow of GUSTO showing various parameterization options and output results for multi-core architecture design.**

by dividing the given algorithm such that achieving a high throughput by large number of cores with a price of large area and achieving a lower area by combining some of the cores and decreasing the number of cores in the platform results in a lower throughput. The quality of the design also depends on the parallelism level of the given algorithm independent from the user decisions. We provide a large freedom to the user to create different type of parallel architectures and to compare them for choosing the one that fits to her situation.

As a very simple example, consider the following instruction that are generated by GUSTO after the algorithm analysis and instruction generation:

$$R1 = R3 + R2$$
$$R4 = R1 + R2$$

$$R5 = R9 + R7$$
$$R6 = R5 + R6$$
$$R13 = R4 + R6$$
$$R12 = R10 + R11$$

GUSTO uses graphviz [134] which is an open source graph visualization software to represent structural information as diagrams of abstract graphs and networks. The Graphviz graph visualization software receives descriptions of graphs in a simple text language, and makes diagrams in several useful formats such as images, postscript for inclusion in PDF or displays in an interactive graph browser. GUSTO analyzes the instructions that are generated in the instuction generation step, and creates a file to be read via graphviz. The generated data flow graph provides data dependencies as well as parallelism options to the user. Instructions above generates the following graph in Figure 7.4 using graphviz.

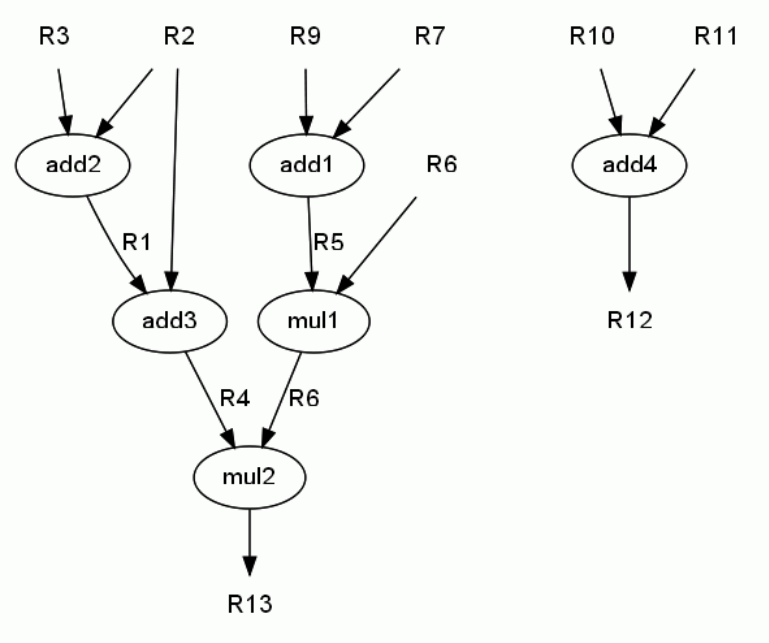


Figure 7.4: **GUSTO analyzes the instructions that are generated in the instruction generation step, and creates a file to be read via graphviz.**

As can be seen from Figure 7.4, there are several different ways that one can partition the algorithm into different cores. Therefore we show three different possibilities in Figure 7.5 where (a) presents 1 processor core design to perform all

instruction in one core and (a) does not require the execution of *generation of the required inputs* step of the GUSTOs' flow. Unlike (a), (b) and (c) show different ways to exploit the parallelism in the given algorithm using 2 or 3 cores. Even though (b) has a multi-core architecture, there is no connectivity needed between cores due to the fact that there are no data dependencies between cores. There is no need to perform *generation of the required inputs* step of the flow for (b). On the other hand (c) generates a multi-core architecture where there is data dependency between its Core 1 and Core 2. Therefore, GUSTO performs *generation of the required inputs* step to make Core 2 be able to access Core 1 to get the data *R4*. It is important to note that GUSTO provides this visual information for user to see the data dependencies and different ways to design a multi-core architecture. Thus, the resources are never shared between instructions in this visualization and there is no relation between the number of arithmetic resources and the GUSTO generated architectures.

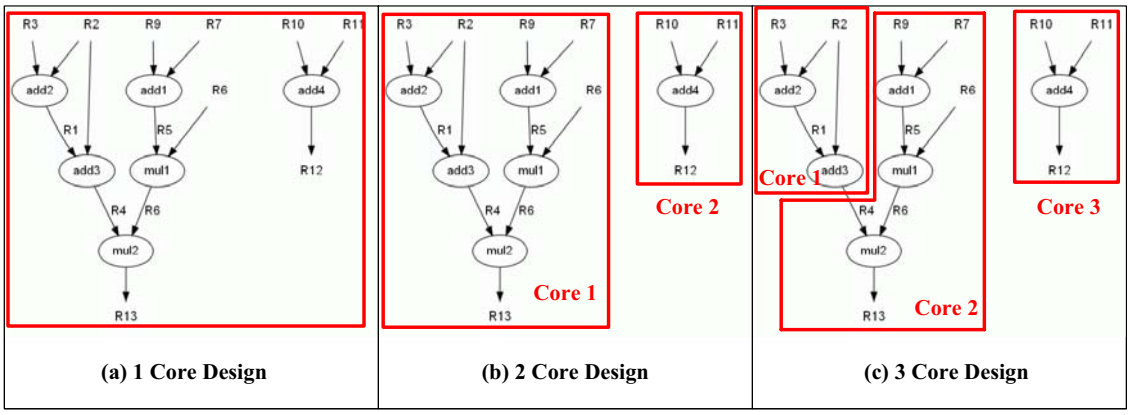


Figure 7.5: **There are several different ways that one can partition the given algorithm into different cores. We show three different possibilities: (a), (b), (c), that are examples of designing one core, two core and three core for a given matrix computation algorithm respectively.**

After a user inputs the number and type of arithmetic resources as well as the data reprentation, GUSTO creates a general purpose PE which can be seen in Figure 7.6 where the proposed processor architecture consists of controller units: instruction and memory controllers and desired arithmetic units: adders, subtractors, multipliers, dividers and square root units etc with full connectivity.

We introduced the details of our processor architecture in Chapter 6, therefore we do not provide further information in this chapter.

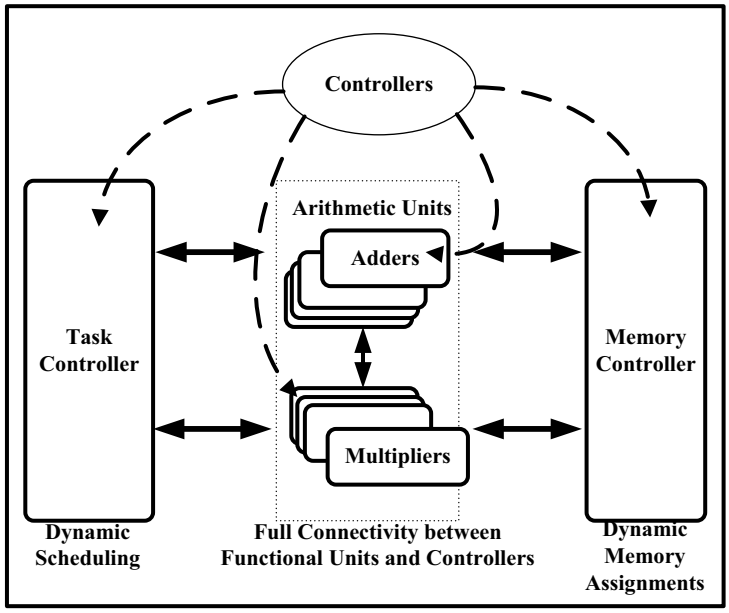


Figure 7.6: **The general purpose processing element which uses dynamic scheduling and dynamic memory assignments. This architecture has full connectivity between functional units and controllers.**

With the architecture generation step, GUSTO designs a general purpose processor for each specific part of the given algorithm. Furthermore GUSTO performs several optimizations to create an optimized application specific PE while ensuring the correctness of the solution is maintained (Figure 7.7). We divided these optimizations into two sections: static architecture generation and trimming for optimization. These optimization are introduced in more detail in Chapter 6, and we summarized these below:

Static architecture generation: In the architecture generation step, GUSTO generates a general purpose PE and its datapath by using resource constrained list scheduling after the required inputs, type and number of resources and data representation, are given. This architecture employs a dynamic scheduling mechanism which is considered as a complex architecture since instruction controller makes decisions on the fly by decoding the instructions. GUSTO simulates this general purpose architecture and simulations help GUSTO to reveal the assignments done

to the arithmetic units and the memory elements during the scheduling process. Gathering this information and using it to cancel the scheduling process and dynamic memory assignments results in a static architecture with significant area and timing savings.

Trimming for optimization: GUSTO performs trimming/removing the unused resources from the general purpose PE while ensuring that correctness of the solution is maintained. GUSTO simulates the architecture to define the usage of arithmetic units, multiplexers, register entries and input/output ports, creates a optimization matrix for each resource and trims away the unused components with their interconnects. GUSTO also provides the usage frequency of the resources, therefore a user can perform resource binding by basically removing the under utilized resources, and GUSTO binds the required instructions to another resource with higher utilization.

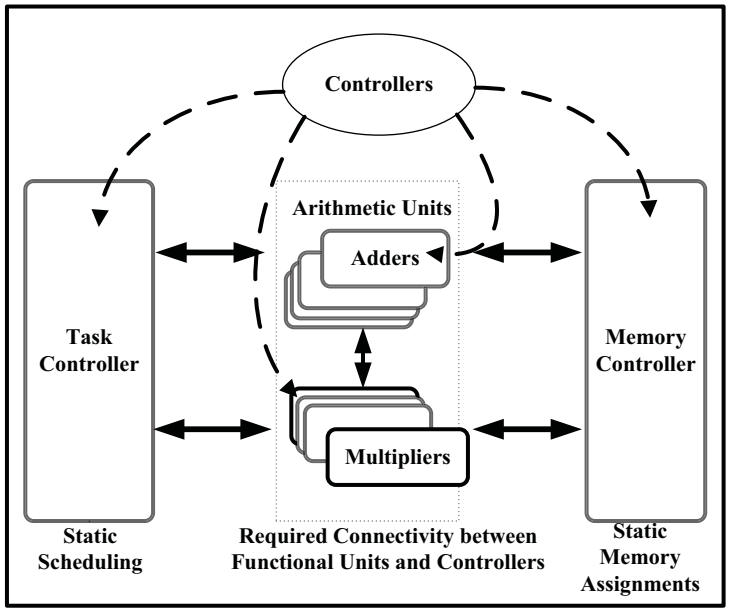


Figure 7.7: **The application specific processing element which uses static scheduling and static memory assignments and has only the required connectivity between functional units and controllers for specific algorithm/s.**

7.2 Architectural Implementation Results of Different Matrix Computation Algorithms

In this section, we present different design space exploration results using different features of GUSTO, general purpose PE design, application specific PE design as well as hierarchical datapath design using homogeneous/heterogeneous cores; and compare our results with previously published FPGA implementations. We concentrate on two different examples of matrix computations:

- matrix multiplication;
- matrix inversion algorithms using decomposition methods.

Matrix multiplication is also employed in matrix inversion and it is highly parallelizable since each entry calculation is independent from each other as shown in Figure 7.8(b), (c) and (d) for 4×4 matrices, $A \times B = C$ (a). This kind of matrix computation represents a class of algorithms that it is possible to generate a platform with there is no data dependencies/data accesses between cores, therefore *generation of the required connections between cores* step in GUSTO's flow do not need to be executed. These kind of algorithms result in a family of architectures that more silicon, designing more number of cores in a platform, will result in higher throughput since there is no timing overhead that is wasted by accessing the data in other cores.

On the other hand, matrix inversion algorithms using decomposition methods (QR, LU and Cholesky) and triangular matrix inversion inherent less parallelism compared to the matrix multiplication since there is only one way to avoid data accesses between cores which is designing a single core for the given algorithm. A user needs to pay a particular attention to the data accesses between cores. It is important to remember that dividing a particular part of the algorithm with many data dependencies into two different cores, will introduce high data access latencies, therefore the silicon wasted for more resources to exploit higher parallelism will not be able to provide higher throughput results.

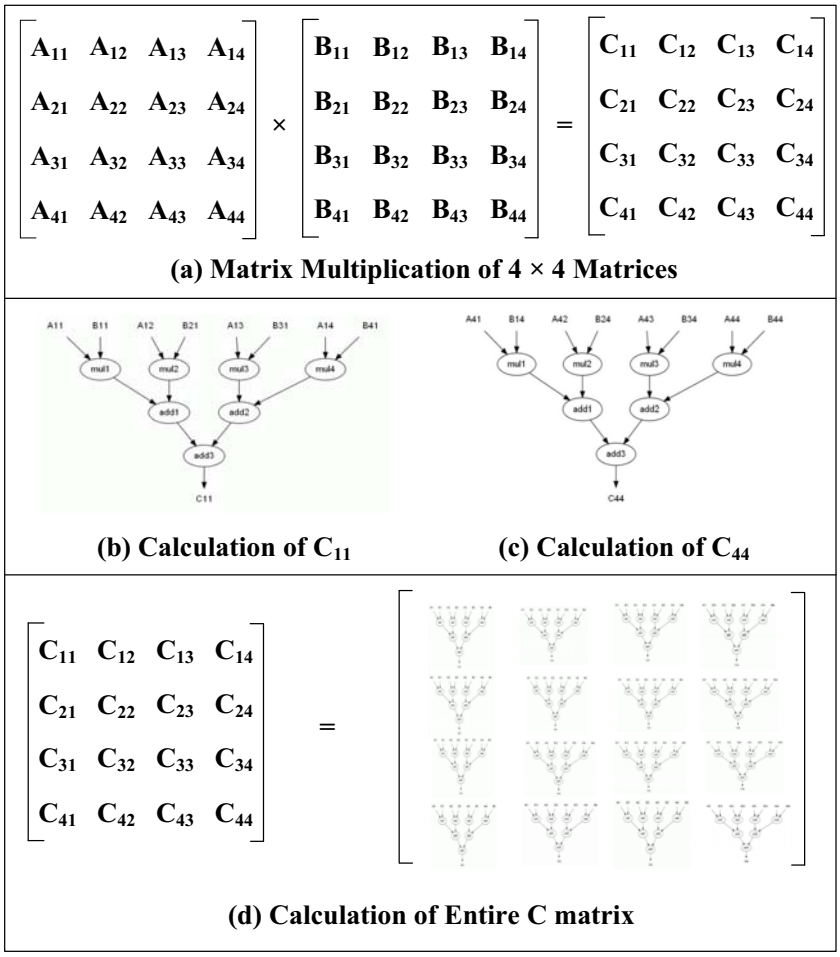


Figure 7.8: **A** 4×4 **matrix multiplication example is shown in (a). Example calculations for the resulting matrix** C **entries,** C_{11} **and** C_{44} **are also shown in (b) and (c). (d) shows a fully parallel architecture for the matrix multiplication.**

In this section we present our various different architectural designs using the hierarchical datapath implementations and heterogeneous core designs for matrix multiplication and matrix inversion algorithms. We present area results in terms of slices and performance results in terms of throughput. Throughput is calculated by dividing the maximum clock frequency (MHz) by the number of clock cycles to perform matrix multiplication/inversion. All designs are generated in Verilog and synthesized using Xilinx ISE 9.2. Resource utilization and design frequency are post place and route values are obtained using a Virtex 4 SX35 FPGA.

7.2.1 Matrix Multiplication

We consider multiplication of two 4×4 matrices using 19 bits of precision for our experiments. We investigate three different design methods:

- Using one PE for entire matrix multiplication procedure;
- Dividing the computation into small homogeneous PEs;
- Designing a heterogeneous architecture with different types of PEs.

1) Matrix multiplication using 1 PE: GUSTO creates one processing element for the computation of the matrix multiplication. These implementations are the application specific architectures that are generated by GUSTO without using hierarchical datapaths. The design flow for these architectures follow the simple GUSTO flow which is presented in Figure 7.3. The parameter that an user can change to create a design space is the number of resources (assuming that the data representation is given as is 19 bits). Therefore, we show three different implementations with different number of functional units in Figure 7.9: *Implementations 1-3*.

- *Implementation 1* employs 4 adders and 4 multipliers;
- *Implementation 2* employs 2 adders and 4 multipliers;
- *Implementation 3* employs 2 adders and 2 multipliers.

In this design method, a user is only capable of changing the number of functional units. GUSTO finds the optimal number of resources which maximizes the throughput while minimizing area by trimming the unused resources. As the number of resources, adders and multipliers, decrease, the area as well as the throughput decreases. Adding more resources than *Implementation 1* cannot increase the throughput due to the fact that the resources are not being used, and trimmed away by GUSTO. This design method provide a limited design space for possible architecture implementation of matrix multiplication algorithm, since the designer cannot generate multi-core architectures to achieve smaller area or higher throughput (Figure 7.10).

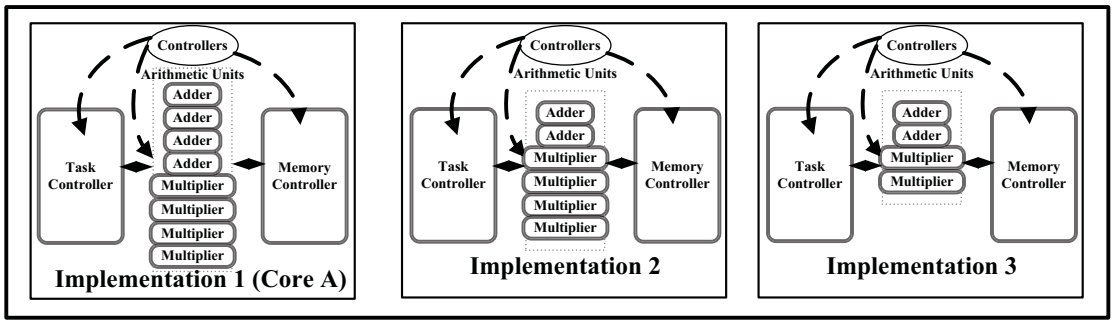


Figure 7.9: **Implementations 1-3 are the application specific architectures that are generated by GUSTO with different number of functional units.**

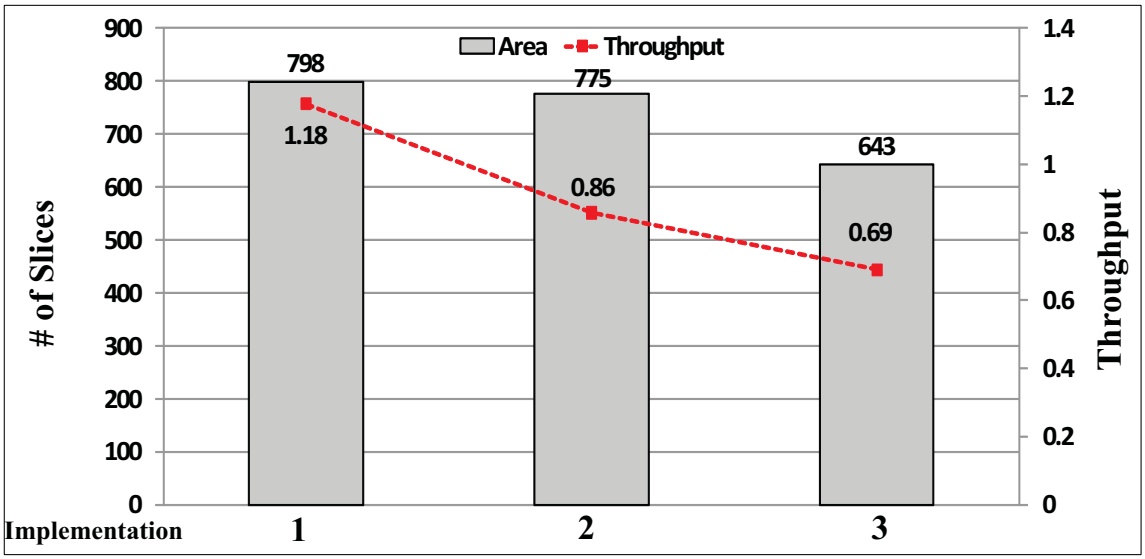


Figure 7.10: **Hardware implementation of matrix multiplication architectures with one PE for entire computation using GUSTO. Implementation 1-3 employs different number of resources for the computation of the matrix multiplication algorithm.**

It is important to note that we name initial *Implementation 1* general purpose architecture, which has 4 adders and 4 multipliers, as **Core A**, because we use this implementation in other design methods.

2) Matrix multiplication using different number of homogeneous PEs: We use **Core A** architecture to create small PEs by dividing the matrix multiplication algorithm into small and parallelizable parts and optimizing **Core A** for

the assigned instructions. **Core** A_1 is a type of a core which means Core A is optimized for the calculation of one entry in the resulting matrix like C_{11}. The required calculations for C_{11} is shown in Figure 7.8(b) and the instructions for this calculation put in **Core A**'s instruction memory in partitioning step of GUSTO's flow. **Core** A_2 is another type of core which means **Core A** architecture is assigned instructions to calculate two different entries of the resulting matrix, such as C_{11} and C_{44} that are shown in Figure 7.8(b) and (c) in partitioning step of GUSTOs' flow.

We present 6 different implementations in Figure 7.11: implementations 4-9, to demonstrate the effectiveness of multi-core architecture generation using homogeneous PEs.

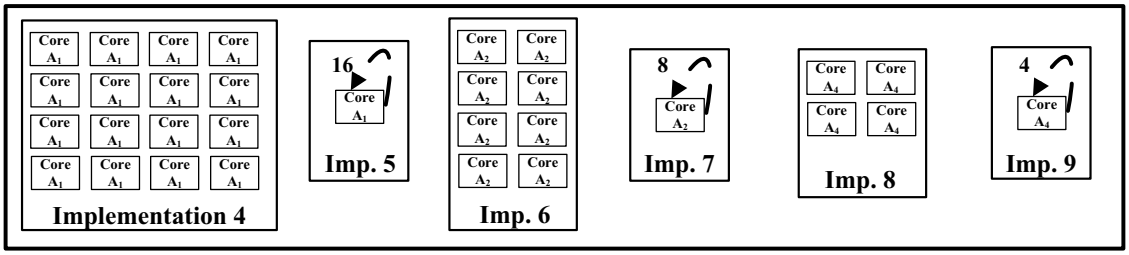


Figure 7.11: **Implementations 4-9 are the application specific architectures that are generated by GUSTO with different number of PEs.**

- *Implementation 4* employs 16 (sixteen) homogeneous A_1 type of PEs where each PE computes one entry of the resulting C matrix (a given matrix can be represented as rectangular table of entries, for example in our example there are 16 entries in a 4×4 matrix). This architecture can receive all A and B matrices in the same clock cycle, place the data in the particular cores such as A_{11}, B_{11}, A_{12}, B_{21}, A_{13}, B_{31}, A_{14} and B_{41} in one core, and compute the respective entry, C_{11}, of the resulting matrix. Each core runs in parallel and generates the resulting matrix with the highest throughput.

- *Implementation 5* employs 1 (one) A_1 type of PE in its architecture. This core is scheduled 16 times to calculate all 16 entries of the resulting matrix for the 4×4 matrix which is shown with an arrow around the core. This

core receives only the elements required for one entry calculation such as A_{11}, B_{11}, A_{12}, B_{21}, A_{13}, B_{31}, A_{14} and B_{41} to compute the respective entry, C_{11}. After the completion of the calculation of C_{11}, the required elements of A and B matrices for the next entry calculation, C, are received. Therefore, the instructions that reside in its instruction memory are the same, but the data inside in its data memory changes and the architecture executes the same instructions for the calculation of different entries using different data.

- *Implementation 6* employs 8 (eight) homogeneous A_2 type of PEs where each PE computes two entries of the resulting C. As an example, a core has the required instructions for the calculation of C_{11} and C_{44} that are shown in Figure 7.8(b) and (c) respectively. Each core resides the same instruction sets for calculation of the matrix multiplication and executes these instruction in parallel with the other cores using different data. Like *Implementation 4*, this architecture can receive all A and B matrix entries in one clock cycle and compute the respective entries in each core in parallel.

- *Implementation 7* employs 1 (one) A_2 type of PE in its architecture where A_2 type of PE is optimized for the calculation of two different entries in the resulting matrix C, therefore scheduled 8 times to compute whole 4×4 matrix multiplication. This core receives the required data for the calculation of the two entries; and after the completion of the calculation, it receives the required data for the calculation of another two entries.

- *Implementation 8* employs 4 (four) homogeneous A_4 type of PEs in its architectures. Each A_4 type PEs calculate different 4 entries such that one PE calculates C_{11}, C_{12}, C_{13} and C_{14} while another one calculates C_{21}, C_{22}, C_{23} and C_{24} in parallel. This core like Implementation 4 and 6 receives all the required data in once, calculates resulting matrix entries in parallel.

- *Implementation 9* employs 1 (one) A_4 type of PE in its architecture where A_4 type PE can calculate 4 different entries in the resulting matrix such as C_{11}, C_{12}, C_{13} and C_{14}. Upon the completion of the calculation of these

entries, this core receives required data for other 4 entries, therefore this core is scheduled 4 times to compute all entries of the resulting matrix.

Implementation 4, 6 and 8 are examples of fully parallel architectures that employ different homogeneous type PEs in their architecture to exploit task level parallelism. The difference between A_1, A_2 and A_4 is the complexity of the architecture where A_1 is the simplest. There is a tradeoff between the simplicity of the PEs and the number of PEs employed in one platform since a platform with A_1 type of PEs required to employ more number of PEs in the platform compared to a platform that employs A_2 type of cores. One would expect *Implementation 4* to provide the highest throughput with the price of highest area results, and *Implementation 8* to provide the smaller throughput compared to *Implementation 4* with a smaller area. Hardware implementation results of these fully parallel architectures can be seen in Figure 7.12 where *Implementation 4* provides the highest throughput with the largest area. Moving to *Implementation 6 and 8*, the throughput as well as the area of the platform decreases which creates a design space for user to choose. In general, these architectures are for users who wants to achieve high throughput values with different area results.

Implementation 5, 7 and 9 are examples of architectures that do not exploit task level parallelism and employs scheduling of the cores to be capable of achieving smaller area results in the design space. Since only one PE, A_1, A_2 or A_4, is employed in these architectures, the increasing complexity of the PEs does not have large impact on the area results. Complexity increases in the architectures, such as moving from A_1 to A_4, stems from the fact that there are larger finite state machines in the memory controller unit, more number of instructions in the instruction memory, and more arithmetic resource usage which may result in less trimming optimization. Since the 4×4 matrix multiplication is an algorithm with less complexity compared to matrix inversion algorithms, we see a slight increase in terms of area while moving from *Implementation 5* to *9* with a slight increase in throughput as can be seen from Figure 7.12.

3) Matrix multiplication with a heterogeneous architecture using different types of PEs: We used GUSTO to generate different type of PEs

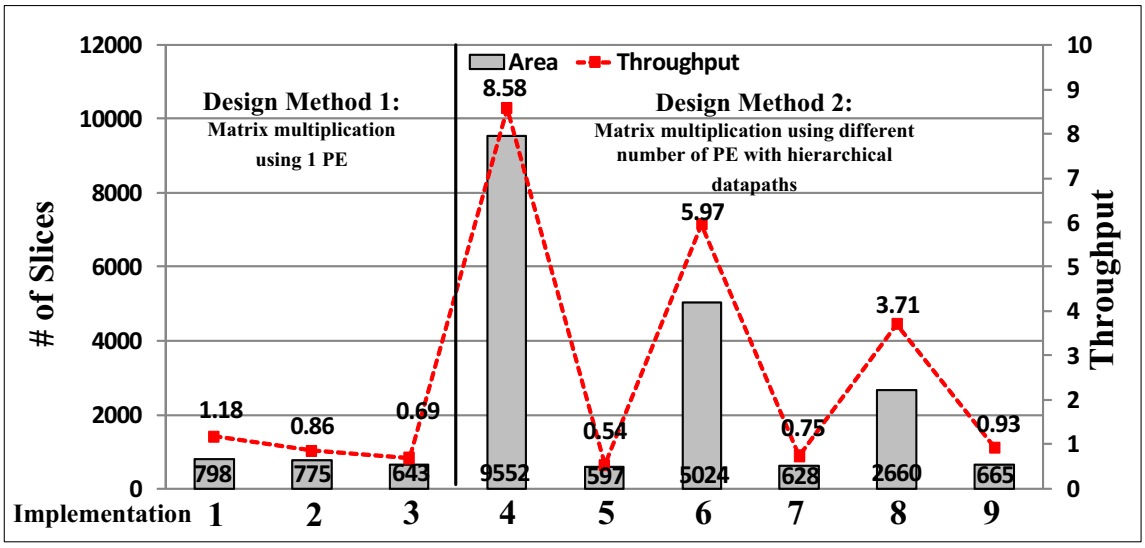


Figure 7.12: **Hardware implementation of matrix multiplication architectures with different design methods, implementation 1-9, using GUSTO.**

to create heterogeneous architectures. Different type of PEs in a platform let us to exploit different architectural implementations that previous design methods cannot provide and enlarge our design space even more.

There are many different ways to combine these different type of PEs in a platform, here we present 3 different implementations in Figure 7.13: implementations 10-12.

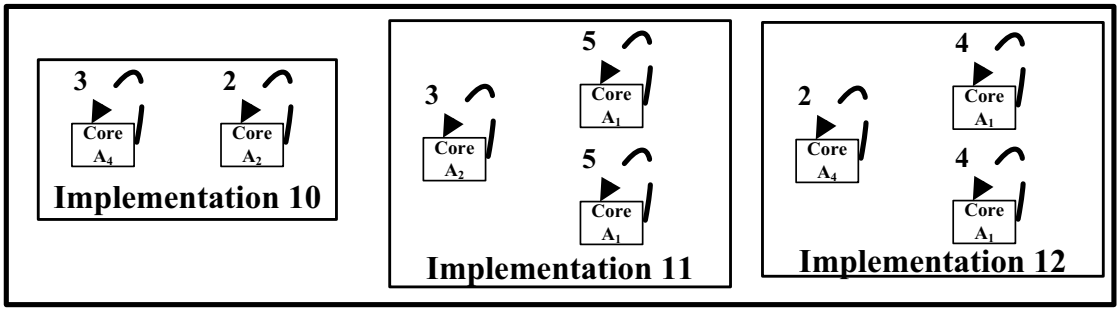


Figure 7.13: **Implementations 10-12 are application specific heterogeneous architectures that are generated by GUSTO with different types of PEs.**

- *Implementation 10* employs 2 (two) PEs which have different types, A_4 and A_2. A_4 and A_2 type of PEs compute 4 different entries and 2 different entries

in the resulting matrix respectively. Each PE, A_4 or A_2, is scheduled 3 times and 2 times respectively and computes different entries with the required input data. It is important to note that the time that data provided to the PEs is different for each PE since A_4 type of PE requires more processing time than A_2 type of PE (A_4 computes 4 entries where A_2 computes 2 entries). Therefore, A_4 type PE runs 3 times to calculate 12 entries and A_2 type of PE runs 2 times to calculate 4 entries of the resulting matrix.

- *Implementation 11* employs three PEs with 2 different types: one A_2 and two A_1 type of PEs. A_2 type of PE is scheduled 3 times to calculate 6 entries and each A_1 type of PEs are scheduled 5 times to calculate 5 entries of the resulting matrix.

- *Implementation 12* employs three PEs with 2 different types: one A_4 and two A_1 type of PEs. A_4 type of PE is scheduled 2 times to calculate 8 entries and each A_1 type of PEs are scheduled 4 times to calculate 4 entries of the resulting matrix.

Implementation of heterogeneous architectures lets a user to find middle points between previous design method results by incorporating mix type of PEs in one platform. The complexity of these architectures is that the execution time of each core such as A_1 and A_2 are different than each other which requires user to provide new data upon the completion of the calculation of each PE in different times. We present the architectural results for *Implementation 10, 11 and 12* in Figure 7.14. *Implementation 10* has the smallest area among the three implementation by employing 2 different PEs. The area and throughput increases by employing more PEs in a platform like in *Implementation 11*. The difference between *Implementation 11* and *12* is that *Implementation 12* employs more complex PEs (A_4) which requires less scheduling, therefore area results of Implementation 12 increases with the complexity of the PE, and throughput of the platform increases as well with less scheduling.

Each design method: *matrix multiplication using 1 PE*, *Matrix multiplication using different number of homogeneous PEs* and *Matrix multiplication with*

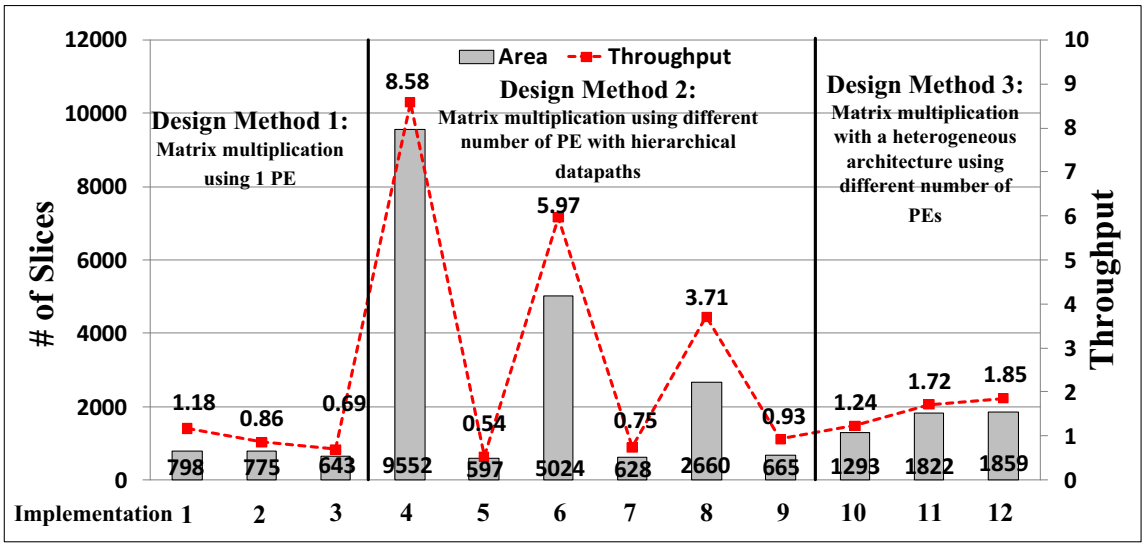


Figure 7.14: **Hardware implementation of matrix multiplication architectures with different design methods, implementation 1-12, using GUSTO.**

a heterogeneous architecture using different types of PEs increases the number of possible implementations and provides a very large design space that a designer can choose depending on his/her needs. We present some of the important findings in our results in Figure 7.15.

To present GUSTO's optimization efficiency, we compared two different *1 PE designs* for matrix multiplication. We implemented another architecture, *Implementation 13*, using 1 PE with 8 multipliers and 8 adders and compared its results with the *Implementation 1* which employs 4 multipliers and 4 adders. One can expect a higher throughput by employing more resources assuming that all of the arithmetic units are used in the computation. However this is not the case and additional resources may not be used. The additional resources, which are not used, still increases the area of the platform while decreasing the throughput due to the decreasing clock frequency of the device. As can seen from Figure 7.15, GUSTO trims away the unused resources from *Implementation 13* since additional resources are not utilized, and generates an architecture with same area and throughput result of *Implementation 1*. This case shows that GUSTO minimizes the area by removing unused resources from architecture while maximizing the throughput. Providing more design methods such as *Matrix multiplication using*

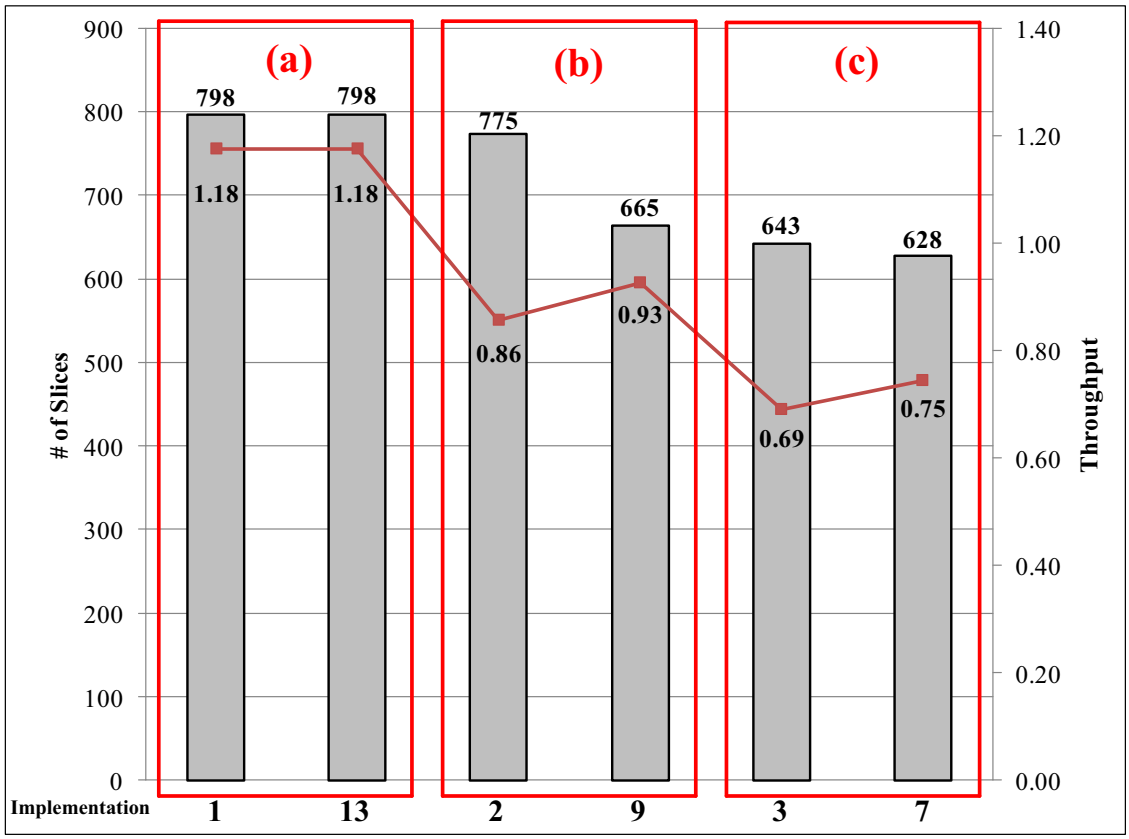


Figure 7.15: **We present some of the important points of matrix multiplication hardware implementation results: 1) finding the optimum hardware to minimize the hardware while maximizing the throughput, 2) improvement in both area and throughput with hierarchical datapath compared to single core design.**

different number of homogeneous PEs and *Matrix multiplication with a heterogeneous architecture using different types of PEs* enlarge our design space and may lead to better results in terms of both area and throughput compared to *1 PE designs*. We present two different examples where we achieve better results compared to *1 PE designs*:

- Figure 7.15 (b) shows that *Implementation 9* provides better results compared to *Implementation 2* in terms of both area and throughput. Therefore every user should prefer *Implementation 9* to *Implementation 2*.

- Figure 7.15 (c) shows that *Implementation 7* provides better results com-

pared to *Implementation 3* in terms of both area and throughput. Therefore every user should prefer *Implementation 7* to *Implementation 3*.

As the number of instructions executed in a processor increases, **the complexity of memory controller** increases as well which results in a higher area usage with a lower clock frequency for the controller. After the optimization of the general purpose processing element, the memory controller becomes static. Thus, memory controller resides a *stateless Moore finite state machine* that includes the controls to send data to the functional units or receive data from functional units (and update the memory entries) in each clock cycle which is introduced in more detail in Chapter 6. One processing element design requires to put this information into one memory controller which requires longer finite state machines in the memory controller that requires larger area. Large number of states lower the maximum clock frequency of the memory controller, and executing the given algorithm with a single instruction in each clock cycle architecture requires more clock cycles to execute given application that decreases the throughput. Therefore, we perform an analysis to see the effects in area, required clock cycles to execute and the throughput of the architecture in Figure 7.16 by generation of different memory controllers for A_1, A_2, A_4, A_8 and A_{16} type of processing elements for 4×4 matrix multiplication.

- A_1 type of processor computes one entry of the resulting matrix. A fully parallel architecture that employs 16 A_1 processors is shown in *Implementation 4* in Figure 7.11.

- A_2 type of processor computes two entries of the resulting matrix. A fully parallel architecture that employs 8 A_2 processors is shown in *Implementation 6* in Figure 7.11.

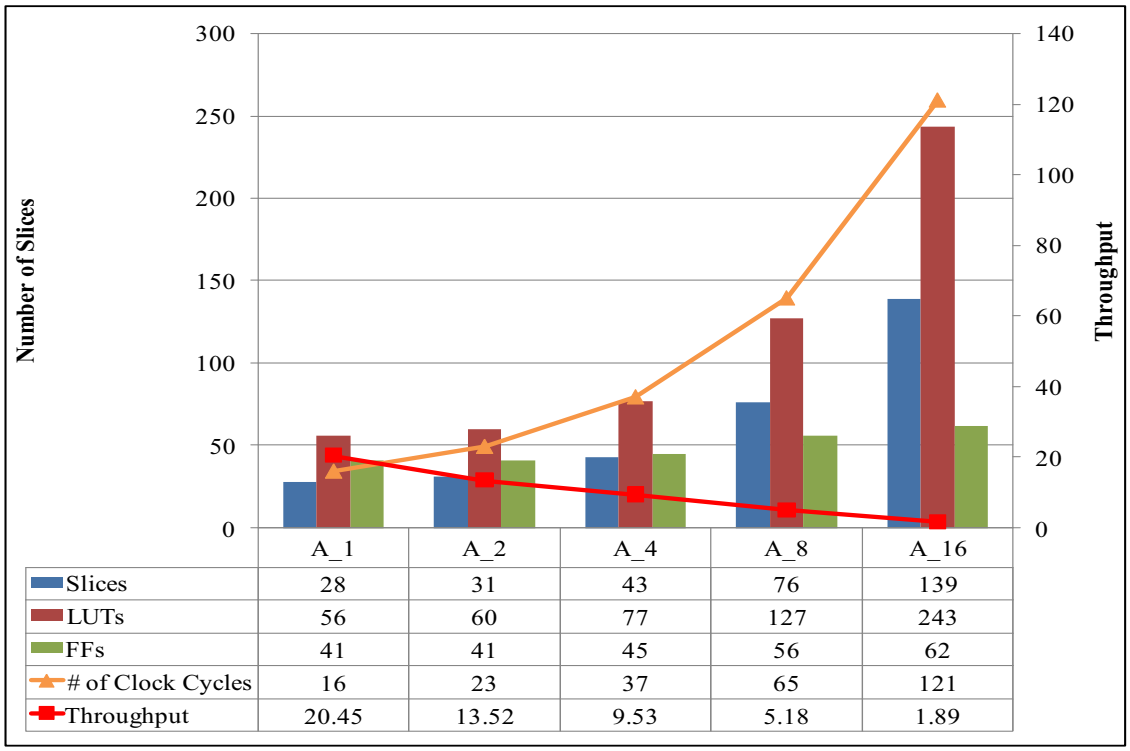
- A_4 type of processor computes four entries of the resulting matrix. A fully parallel architecture that employs 4 A_4 processors is shown in *Implementation 9* in Figure 7.11.

- A_8 type of processor computes eight entries of the resulting matrix. A fully parallel architecture employs 2 A_8 processors.

- A_{16} type of processor computes all of the entries of the resulting matrix. This is one PE design for matrix multiplication and can be seen in *Implementation 1* in Figure 7.9.

As expected, the complexity increases while moving from the simplest memory controller: A_1 to the most complex memory controller: A_{16} for 4×4 matrix multiplication as can be seen from the increase in area (slices). Increasing number of states requires more LUTs to be used and more number of clock cycles to be executed to calculate more entries for the resulting matrix. However it is important to see that the increase in the area is not double between A_1 and A_2; and A_2 and A_4 etc. Therefore employing 16 A_1 type processor memory units in one architecture requires a larger area than employing 8 A_2 type processor memory units. Therefore one can say that the best architecture in terms of the area is the one PE design which is *Implementation 1* in Figure 7.9 with the memory controller results shown for A_{16} in Figure 7.16. However, to achieve a higher throughput which requires high clock frequencies and low number of clock cycles while executing a given algorithm is possible by dividing the given algorithm into different number of processing elements. As can be seen from Figure 7.16, while the complexity increases, the throughput of the architecture decreases significantly (A_1 type of processor offers a throughput value of **20.45** while A_{16} type of processor offers **1.89**).

Comparison: We also compared our matrix multiplication implementation results with a hand coded design which is implemented using System Generator model based design tool from Xilinx as well as the previously published work. The architectural implementation using System Generator design tool is shown in Figure 7.17 for a 4×4 matrix multiplication using 18 bits of data as precision. This design includes an address generation logic **(a)** which is used to calculate the address to be read from the input data memory **(b)**. This implementation assumes that the input data, matrices A and B, are already in the input data memory. After reading the required inputs from input data memory, these data are sent to the multiply-accumulate block **(c)** to calculate respective matrix entry. After calculation of each resulting matrix entry, data is written to the result data memory



	A_1	A_2	A_4	A_8	A_16
Slices	28	31	43	76	139
LUTs	56	60	77	127	243
FFs	41	41	45	56	62
# of Clock Cycles	16	23	37	65	121
Throughput	20.45	13.52	9.53	5.18	1.89

Figure 7.16: **An analysis to see the effects in area, required clock cycles to execute and the throughput of the architectures by generation of memory controllers with the increasing complexity: A_1, A_2, A_4, A_8 and A_{16} type of processing elements for 4×4 matrix multiplication.**

(d) using the address calculated by address generation logic. This design is an example of existing design tool and their usage in matrix computation algorithms. Design of a 4×4 matrix multiplication core using System Generator took around a week.

We also compared our results with the previously published articles in Table 7.2. El-Atfy et. al [25] proposed an architecture for 8×8 matrix multiplication with 16-bit fixed point arithmetic on a Virtex-4 device. The architecture is based on a parallel model of the matrix multiplication sequence where an array of processing elements (PEs) are employed in fully parallel fashion to calculate each elements of the resulting matrix. The authors are also utilized BRAM and DSP48 blocks in their design. Mencer et. al [26] implemented 4×4 matrix multiplication using Booth encoding with bit-serial multipliers on the Xilinx-4 device. Amira et.

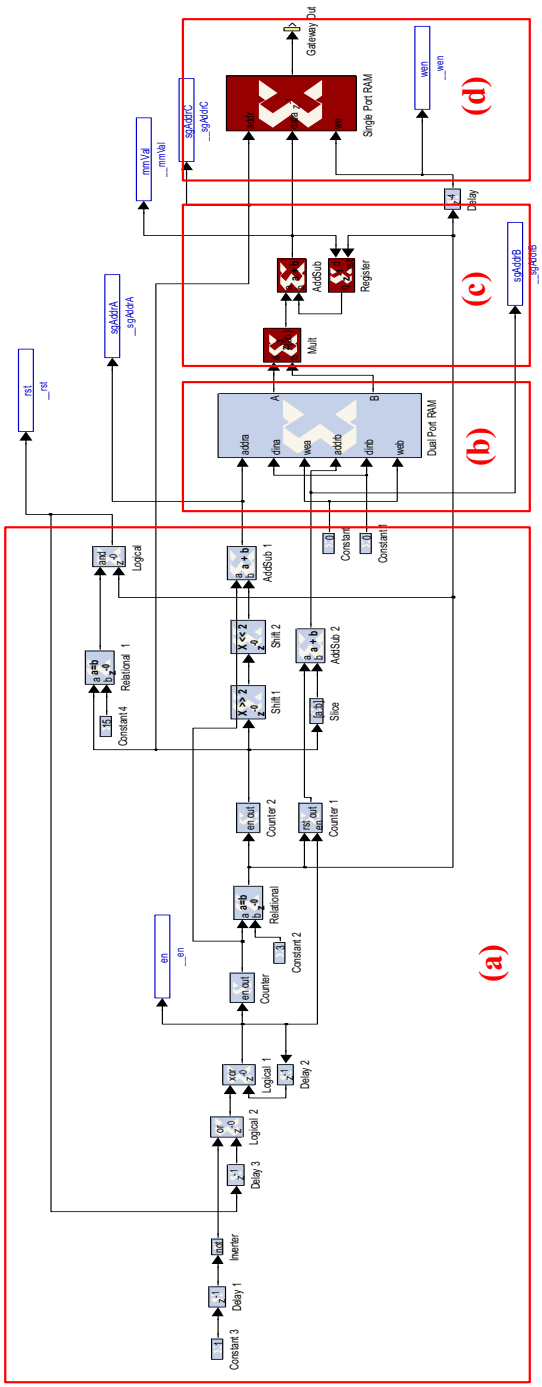


Figure 7.17: **Hardware implementation of 4×4 matrix multiplication core with 18 bits of precision using System Generator for DSP design tool from Xilinx. (a), (b), (c) and (d) are the address generation logic, input data memory, multiply-accumulate units and destination data memory respectively.**

al [27] improved [26] using modified Booth-encoder multiplication with Wallace tree addition on a Xilinx XCV1000E device. Prasanna et. al [28] provided the theoretical lower bound in latency for a matrix multiplication design that is based on a linear array providing design trade-offs between the number of registers and the latency. These results are improved in Jang et. al [29] by developing new algorithms and architectures for matrix multiplication.

7.2.2 Matrix Inversion

For matrix inversion, we consider all three matrix inversion methods (with QR, LU and Cholesky decompositions) which are explained in Chapter 4 in detail. There are many different ways to divide matrix inversion algorithms since its higher complexity and parallelism. Matrix inversion algorithms include matrix decomposition, upper triangular matrix inversion and matrix multiplication. Matrix multiplication inherents high level of parallelism as presented in the previous subsection. Unlike matrix multiplication, matrix decomposition and upper triangular matrix inversion algorithms present many data dependencies which makes their multi-core hardware implementation harder. As an example, consider the QR decomposition algorithm given in Algorithm 4 for a 4×4 matrices. The data dependencies and the data flow of QR decomposition algorithm is shown in Figure 7.18. There are four different stages in QR decomposition algorithm which are presented as stage 1-4 in respective boxes. Each of these stages, computes different entries of upper triangular matrix, R, and Orthogonal matrix, Q: Stage 1-4 produces diagonal entries of upper triangular matrix, columns of orthogonal matrix, non-diagonal upper triangular matrix entries and intermediate entries of the orthogonal matrix respectively. Stage 2 creates one column for orthogonal matrix in each iteration where Stage 1 and 3 creates one row of upper triangular matrix in each iteration. Therefore in the first iteration, one column (first column) of matrix Q, Q_1, and one row (first row) of matrix R, R_1, are computed and the the rest of the columns of Q matrix are updated which are seen as intermediate, X, values. The next iteration starts after updating the columns of Q matrix in every Stage 4, therefore we cannot parallellize its computation over columns. This

is the similar case for other decomposition methods.

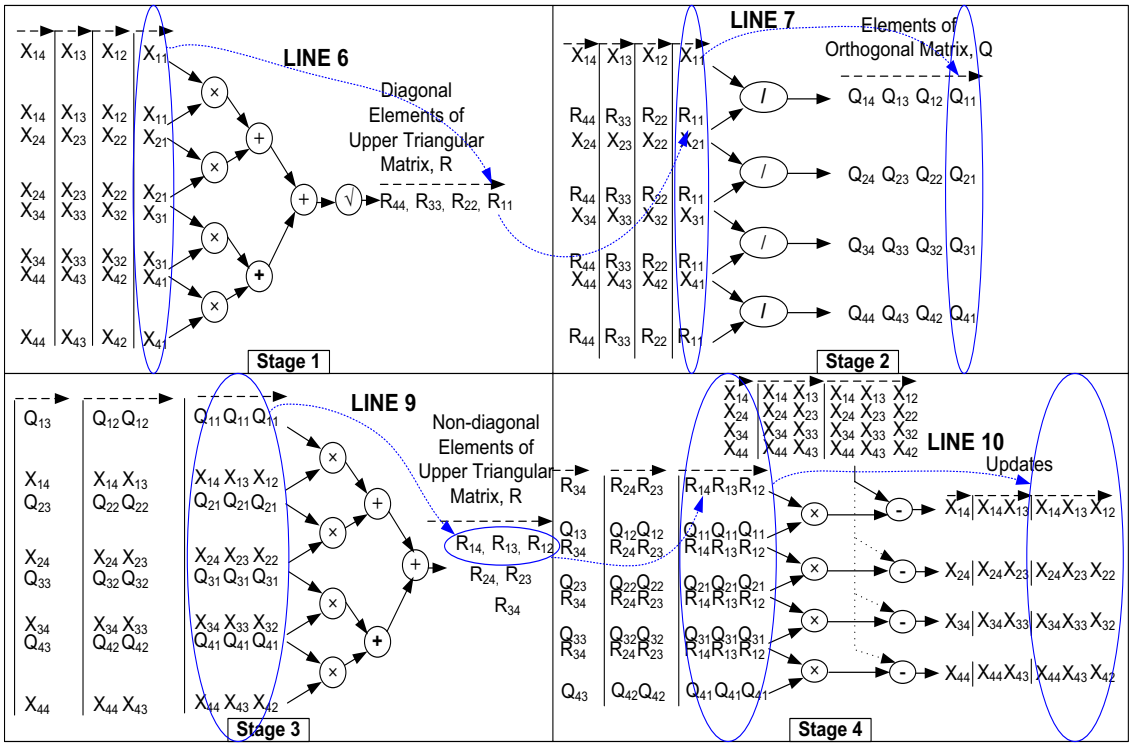


Figure 7.18: **The data dependencies and the data flow of QR decomposition algorithm.**

Therefore, we used GUSTO to generate heterogeneous architectures by dividing the matrix inversion steps into different PEs. Each heterogeneous architecture includes a decomposition PE, one/two matrix inversion PE for the upper triangular matrix and a matrix multiplication PE.

- *Matrix inversion using QR decomposition:* We divided matrix inversion into three different PEs including one PE for QR decomposition to generate upper triangular matrix, R, and orthogonal matrix, Q; one PE for upper triangular matrix inversion using upper triangular matrix, R, entries with an identity matrix, I, already residing its memory units and one PE for matrix multiplication for multiplication of inverted upper triangular matrix entries, R^{-1}, with transpose of orthogonal matrix, Q^T. Transpose of matrix Q is another matrix which is created by writing the rows of Q as the columns of transpose

matrix or writing the columns of Q as the rows of transpose matrix and do not require any computation. GUSTO generates and assigns required instructions into to Instruction Controller of matrix multiplication PE to read values in transposed order. The proposed architecture for matrix inversion using QR decomposition can be seen in Figure 7.19.

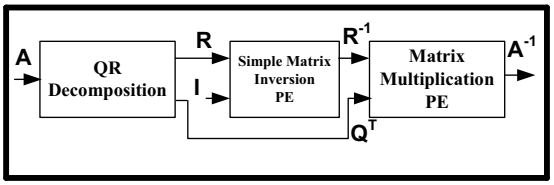


Figure 7.19: **Heterogeneous matrix inversion architecture for matrix inversion using QR decomposition.**

- *Matrix inversion using LU decomposition:* We divided matrix inversion into four different PEs including one PE for LU decomposition that generates lower and upper triangular matrices, L and U respectively; two PEs for inversion of triangular matrices generating inverted triangular matrices, U^{-1} and L^{-1}; and one PE for multiplication of upper triangular matrices. LU decomposition generates its outputs column by column; since zero entries in triangular matrices are already known, upper triangular matrix inversion can start its computation after accessing the required first column. We employed two triangular matrix inversion PEs in parallel to be able to compute their inversion in parallel. The proposed architecture for matrix inversion using LU decomposition can be seen in Figure 7.20.

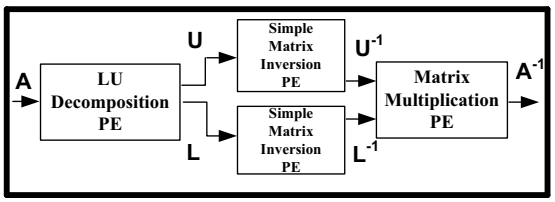


Figure 7.20: **Heterogeneous matrix inversion architecture for matrix inversion using LU decomposition.**

- *Matrix inversin using Cholesky decomposition:* We divided matrix inversion into four different PEs like LU decomposition based matrix inversion architecture including one PE for Cholesky decomposition generating Cholesky triangle; two PEs for triangular matrix inversion where each uses Cholesky triangle, G, and transpose of the Cholesky triangle, G^T, respectively generating inverted triangular matrices, G^{-1} and $(G^T)^{-1}$; and one PE for matrix multiplication of the inverted triangular matrices. Cholesky decomposition PE generates Cholesky triangle and transpose of the matrix is provided by register renaming embedded in the instructions of matrix inversion PE Instruction Controller. The proposed architecture for matrix inversion using LU decomposition can be seen in Figure 7.21.

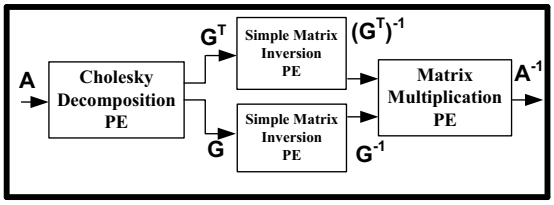


Figure 7.21: **Heterogeneous matrix inversion architecture for matrix inversion using Cholesky decomposition.**

We first investigate the total number of operations used in different decomposition methods which is shown in Figure 7.22 in log domain. Decomposition methods for the given matrices, QR, LU and Cholesky, are always the dominant calculation while inverting the matrix (except 2×2 matrix inversion using LU due to its low complexity). If one is indifferent between differen decomposition methods, it is important to notice that there is an inflection point between LU and Cholesky decompositions at 4×4 matrices with a significant difference from QR decomposition. Furthermore, this inflection point is shifted to 5×5 matrices for matrix inversion implementations where LU and Cholesky have more significant differences in terms of total number of operations; besides the difference between QR and the other decomposition methods increases. Since decomposition methods are highly serial computations, one would expect benefit from the overlapping computations between cores to hide latency while designing a heterogeneous matrix

inversion architecture.

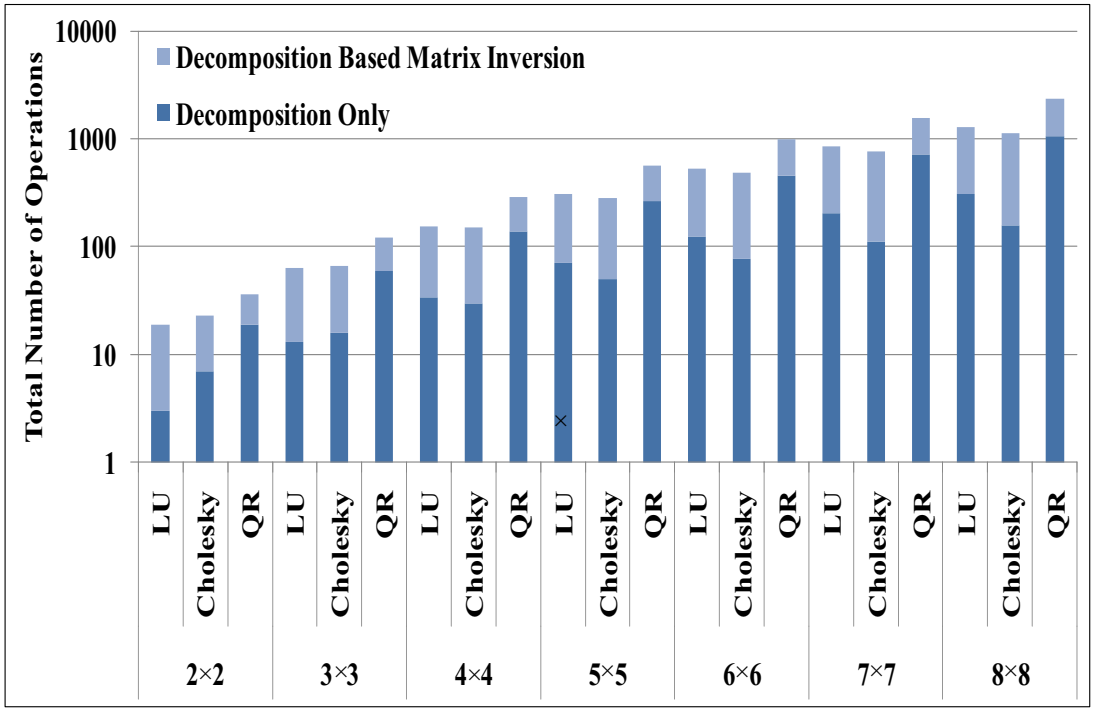


Figure 7.22: **Total number of operations in log domain for decomposition based matrix inversion (light) and decompositions only (dark). Note that the dark bars overlap the light bars.**

Therefore, we presented and compared heterogeneous matrix inversion architecture with the single PE design for matrix inversion in Figure 7.23 in terms of area and throughput. Even though the matrix multiplication algorithm is highly parallelizable and can provide efficient implementation through multi-core architecture generation, the other parts of the matrix inversion algorithm, decomposition and upper triangular matrix inversion, do not provide the same parallelism. Therefore, the area results of the architecture increase since the most complex part of the matrix inversion algorithms is the decomposition calculation, and additional simple computations do not decrease the optimization and trimming efforts significantly. In more detail, the additional calculations require low number of instructions compared to decomposition as can be seen in Figure 7.22 which do not increase the complexity of the memory controller as well as the task controller significantly. On the other hand, the throughput increases with heterogeneous

multi-core architecture generation for matrix inversion algorithms. This is due to reason that the cores are capable of hiding latencies between each other and decreasing the required clock cycles to compute given matrix inversion. Such as matrix multiplication PE starts its computation after receiving the last row of R^{-1} and first column of Q^T to compute A_{41}^{-1} and does not need to stall until all input entries are received.

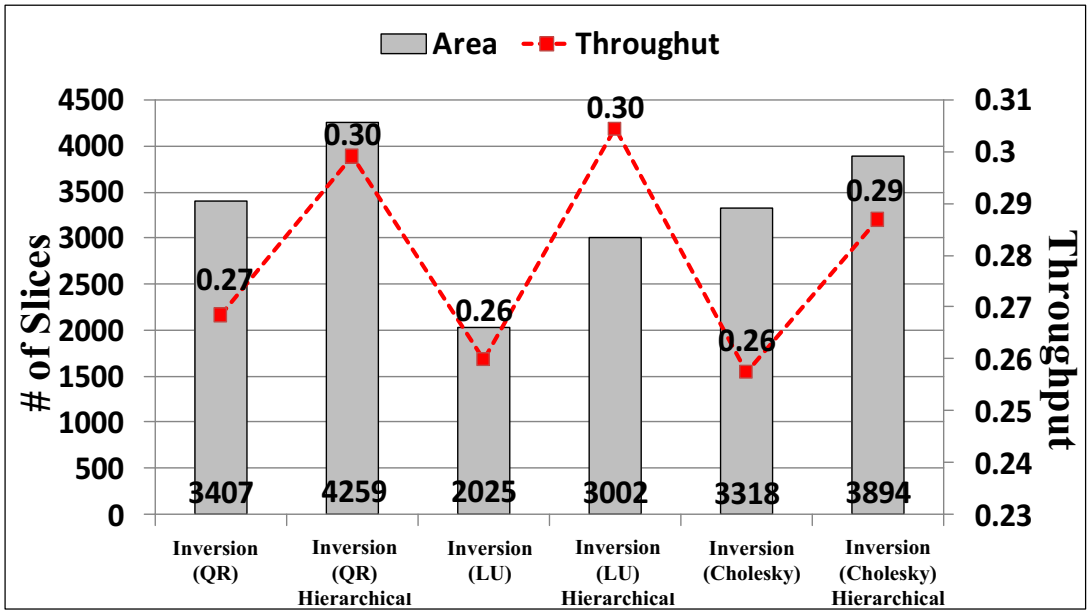


Figure 7.23: **Hardware implementation of matrix inversion architectures with different design methods (using QR, LU and Cholesky decompositions) using GUSTO.**

In chapter 2, we stated that GPU architectures employs large number of ALUs by removing the scheduling logic to exploit instruction level parallelism and caches that remove memory latency. Therefore GPUs are simply very powerful number crunching machines. Thus, the future's high performance parallel computation platform should have an ALU dominant architecture to employ more resources for computation by removing as much control logic as possible. To create an ALU dominant architecture, we perform different optimizations to GUSTO generated general purpose architectures: *static architecture generation* and *trimming for optimization*. We further investigate our results for matrix decomposition, multiplication and inversion architectures to see if our architectures are ALU ori-

ented. We present our analysis in Figure 7.24. Each section, (a) - (g), includes 4 small boxes: total area results for the processor, and the individual units of the processor including instruction controller, arithmetic units and memory controller. For example (a) shows results for a QR decomposition PE including total area for the processor following the individual results for instruction controller, arithmetic units and memory controller. Our designs include:

- **(a):** 1 PE design for QR decomposition algorithm. As can be seen from the distribution of the area, our processor architecture is ALU oriented by using 87 and 317 slices for the controller units whereas arithmetic units consume 1562 slices, 80% of the total area usage.

- **(b):** 1 PE design for matrix inversion architecture that uses LU decomposition. This architecture consumes 65% of its silicon to the arithmetic units where control units consumes only 5% and 30% for instruction controller and memory controller respectively.

- **(c):** 1 PE design for LU decomposition algorithm. This architecture also consumes significant portion, 70%, of its silicon into arithmetic units where control units consume 30% of the processor area.

- **(d):** 1 PE design for matrix inversion architecture that uses Cholesky decomposition. The processor consumes 55% of the silicon into arithmetic units where control units consume a total of 45% of the silicon.

- **(e):** 1 PE design for Cholesky decomposition algorithm. This architecture consumes 87% of silicon into arithmetic units which is the highest precentage among all decomposition methods.

- **(f):** 1 PE design for upper triangular matrix inversion algorithm. This is another architecture that consumes very high percentage, 85%, of its silicon into arithmetic units.

- **(g):** 1 PE design for matrix multiplication algorithm. This architecture consumes 73% of its silicon into arithmetic units where controller units consume 27% of the total silicon.

As can be seen from the Figure 7.24, GUSTO generated architectures for matrix decomposition, multiplication and inversion are all ALU oriented indeed. It is also important to see that while the silicon consumed to the arithmetic units is 55% for one PE design of Cholesky decomposition based matrix inversion (d), heterogeneous matrix inversion architecture for Cholesky decomposition which consists of decomposition, upper triangular matrix inversion and matrix multiplication units consumes 87%, 85% and 73% of their silicon into arithmetic units. This is the similar case for LU and QR decomposition based matrix inversion architectures. Therefore, multi-core architecture implementations provide architectures that consumes higher percentage of their silicon into ALU compared to the one PE designs. This is another benefit of multi-core architecture implementation option provided by GUSTO.

Comparison: We provide a comparison between our results with the architectures employing heterogeneous cores using hierarchical datapaths and previously published implementations for 4×4 matrices in Table 1. We present all of our implementations with bit width of 20 as this is the largest bit width value used in the related works. Though it is difficult to make direct comparisons between our designs and those of the related works (because we used fixed point arithmetic instead of floating point arithmetic and fully used FPGA resources (like DSP48s) instead of LUTs), we observe that our results are comparable. The main advantages of our implementation are that it provides the designer the ability to study the tradeoffs between architectures with different design parameters and design methods, and provides a means to find an optimal design.

7.3 Conclusion

This chapter presents automatic generation and optimization of matrix computation architectures using hierarchical datapaths through a generator tool, GUSTO, that is developed to enable easy design space exploration. GUSTO provides different design methods and parameterization options which enable us to study area and performance tradeoffs over a large number of different architec-

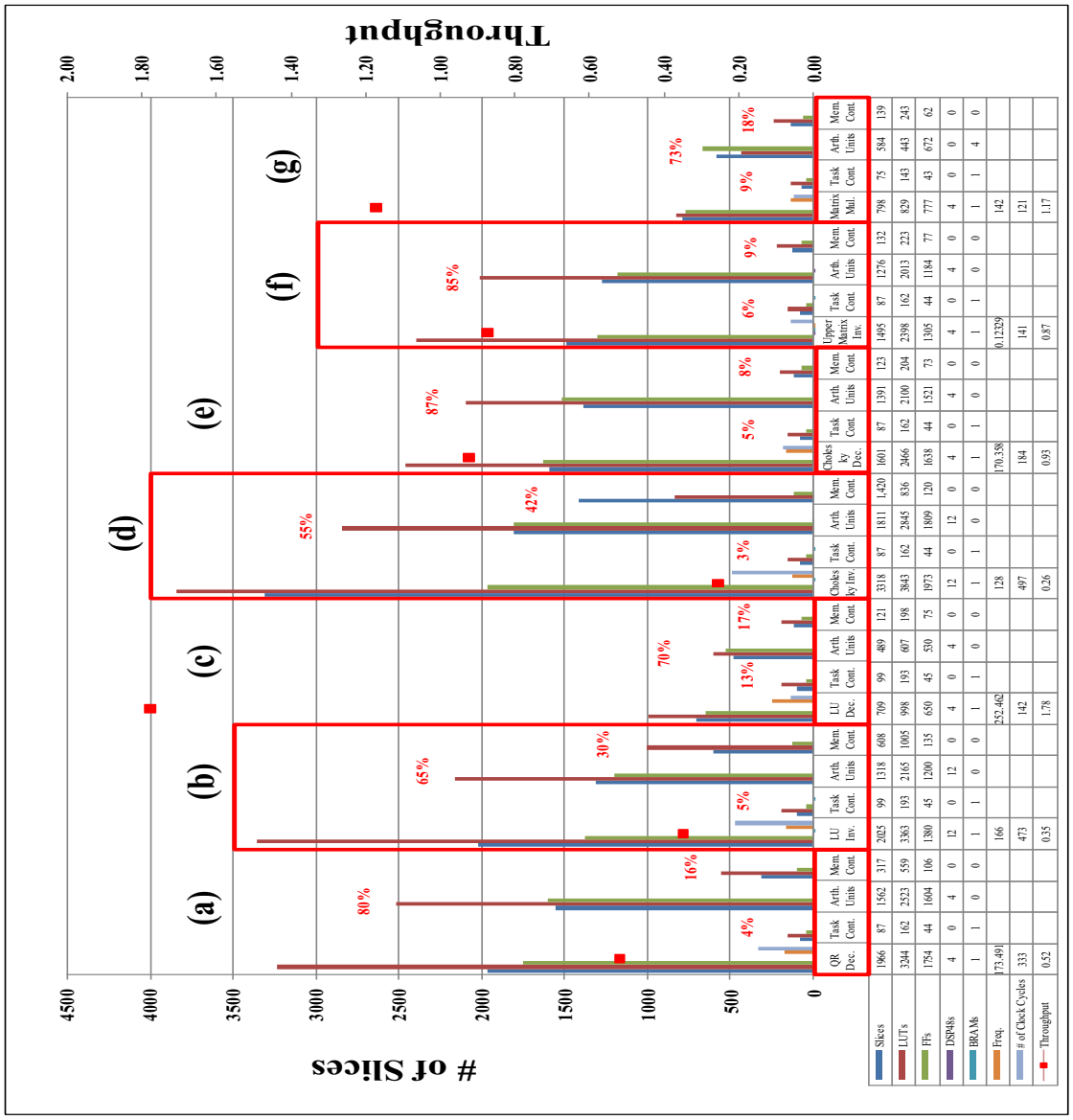


Figure 7.24: **GPU architectures employs large number of ALUs by removing the scheduling logic to exploit instruction level parallelism and caches that removes memory latency. Therefore GPUs are simply very powerful number crunching machines. Thus, the future's high performance parallel computation platform should have an ALU dominant architecture to employ more resources for computation by removing as much control logic as possible. We show that GUSTO generated architectures for matrix decomposition, multiplication and inversion are all ALU oriented indeed by consuming 65% - 87% of their silicon into ALUs.**

Table 7.1: Comparisons between our results with the architectures employing heterogeneous cores using hierarchical datapaths and previously published articles for Decomposition based Matrix Inversion Architectures. NR denotes not reported.

	[18]	[19]	**GUSTO**		
Method	QR	QR	**QR**	**LU**	**Cholesky**
Bit width	12	20	**20**	**20**	**20**
Data type	fixed	floating	**fixed**	**fixed**	**fixed**
Device type (Virtex)	II	IV	**IV**	**IV**	**IV**
Slices	4400	9117	**4259**	**3002**	**3894**
DSP48s	NR	22	**12**	**12**	**12**
BRAMs	NR	NR	**3**	**3**	**3**
Throughput $(10^6 \times s^{-1})$	0.28	0.12	**0.3**	**0.3**	**0.29**

tures. In this chapter, we specifically concentrate on matrix multiplication and matrix inversion methods to observe the advantages and disadvantages of different design methods in response to varying parameters. We show that employing hierarchical datapaths results in more detailed design space exploration and more efficient hardware implementation.

The text of Chapter 7 is in part a reprint of the material as it appears in the proceedings of the International Conference on Wireless Communications and Networking. The dissertation author was the primary researcher and author and the co-authors listed on this publication [55] directed and supervised the research which forms the basis for Chapter 7.

Table 7.2: Comparisons between our results and previously published articles for Matrix Multiplication Architectures. NR denotes not reported.

	GUSTO															
Design	Imp. 4	Imp. 6	Imp. 8	Imp. 12	Imp. 11	Imp. 10	Imp. 1	Imp. 9	Imp. 7	Imp. 5	Hand-Coded	[25]	[26]	[27]	[28]	[29]
Bit width	20	20	20	20	20	20	20	20	20	20	18	16	16	16	16	16
Data type	fixed	fixed	fixed	fixed	fixed	fixed	fixed	fixed	fixed	fixed	fixed	fixed	fixed	fixed	fixed	fixed
Device type (Virtex)	IV	IV	IV	IV	IV	IV	IV	IV	IV	IV	IV	IV	IV	XCV1000E	II	II
Slices	9552	5024	2660	1859	1822	1293	798	665	628	597	93	668	3816	1184	620	560
DSP48s	64	32	16	12	12	8	4	4	4	4	1	64	0	0	0	0
BRAMs	0	0	0	0	0	0	0	0	0	0	2	24	0	0	0	0
Throughput	8.58	5.97	3.71	1.85	1.72	1.24	1.18	0.93	0.75	0.54	1.27	40	1.73	NR	6.64	6.64

Chapter 8

FPGA Acceleration of Mean Variance Framework for Optimal Asset Allocation

With the increasing on die resources, FPGAs become attractive to the scientific computing community. In this chapter, we investigate potential speed-ups for a computationally intensive financial application, the mean variance framework for optimal asset allocation, using simulations of the hardware architectures. The mean variance framework's inherent parallelism in its solution steps (due to many matrix computations and its use of Monte Carlo simulations) and its need for reprogramability (to allow for modifications based on different investor characteristics) make the framework an ideal candidate for an FPGA implementation. However, reconfigurability of FPGAs is both a blessing and a curse since it provides great flexibility in terms of the design of an application with increasing programming complexity. This study proves the need for automatic generation of multi-core architectures on FPGAs for acceleration of financial computations.

Asset allocation is the core part of portfolio management. With asset allocation, an investor distributes his wealth across different asset classes which include different securities such as bonds, equities, investment funds, derivatives, etc. in a given market to form a portfolio. Because each asset class responds differently to shifts in financial markets, an investor can minimize the risk of loss and maximize

the return of his portfolio by diversifying his assets. The goal of the portfolio manager in a financial institution is to provide the asset allocation with the greatest return for some level of risk for investors [30, 31].

A portfolio manager needs to include two pieces of information to determine the best allocation for a given investor: the investor's profile and the market data. The investor profile includes the current asset allocation of the investor, the budget, the investment time horizon, and the investor's objectives and satisfaction indices to be able to evaluate the portfolio's performance. The market data include the joint distribution of the prices at the investment horizon and the implementation costs for trading these securities.

Determining the best allocation for a given investor requires solving a constrained optimization problem [32–34]. Convex programming problems represent a broad class of constrained optimization problems which can be solved numerically [35]; however an optimal asset allocation problem includes a large number of variables that need to be processed which requires a long computation time. Therefore, using an approximation method for the allocation optimization is crucial.

The most popular approximation approach for optimal asset allocation is Markowitz's mean variance framework [21]. In this framework, the investor tries to maximize the portfolio's expected return for a given risk and investment constraints. Mean variance framework is a two-step approach which approximates the solution of the optimal asset allocation problem as a tractable problem. The first step of the mean variance optimization selects efficient allocations for different risks among all the possible combinations of assets to form the efficient frontier; and the second step searches for the best allocation among all efficient allocations found in the first step.

Increasing the number of assets in a portfolio significantly improves the efficient frontier as shown in Figure 8.1. Adding new diversified assets to a portfolio shifts the frontier to the upper left which gives better return opportunities with less risk compared to the lower number asset portfolios. An efficient way to find an optimal allocation for small investors is to use commercially available asset alloca-

tion software: World Markets [36], Allocation Master [37], Encorr [38], PACO [39], Expert Allocator [40], Horizon [41] and Power Optimizer [42]. However financial institutions which make larger investments or control large individual investor portfolios face more complicated problems to obtain the optimal asset allocation. Their higher number of assets and more complex diversification require significant computation that currently only high performance computing can provide.

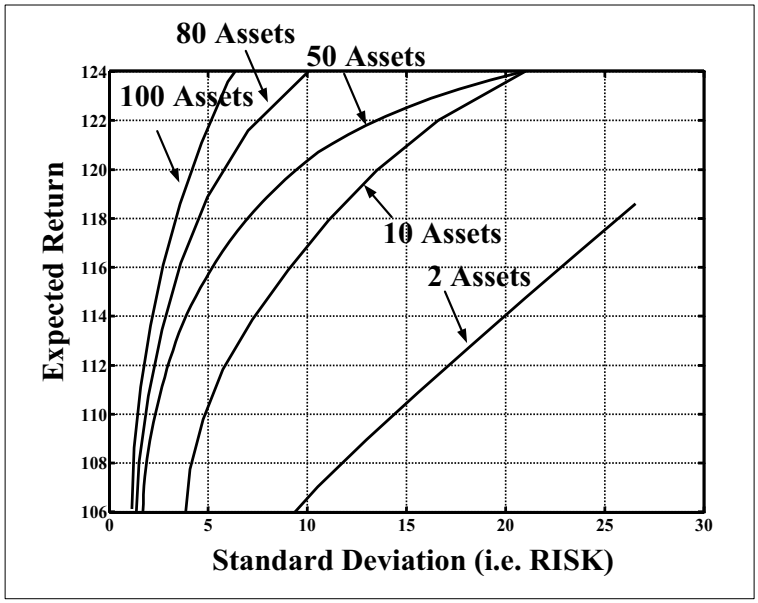


Figure 8.1: **Increasing the number of assets in a portfolio significantly improves the efficient frontier, the efficient allocations of different assets for different risks. Adding new assets to a portfolio shifts the frontier to the upper left which gives better return opportunities with less risk compared to the lower number of assets portfolios.**

There is an increasing interest for FPGAs from high performance computing (HPC) community since conventional microprocessors are struggling to keep up with the Moore's law that degrades the performance gains with increasing power requirements. The addition of FPGAs to the existing high performance computers can boost the application performance and design flexibility. The advantages of FPGAs over conventional compute clusters are less power consumption, increased compute density and significant performance improvement for highly parallel applications. The mean variance framework's inherent parallelism and its need for reprogramability make the framework an ideal candidate for an FPGA implementation.

There are some previous works which consider the hardware acceleration of different financial problems, mainly concentrated on Monte-Carlo simulations [43–49]. Zhang et al. [43] and Morris et al. [44] focused on single option pricing where Kaganov et al. [49] considered credit derivative pricing. Also interest rates and Value-at-Risk simulations are being considered by Thomas et al. in [45, 46]. To the best of our knowledge, we are the first to propose hardware acceleration of the mean variance framework for optimal asset allocation using FPGAs.

The major contributions of this chapter are:

1) A detailed description of the mean variance framework for optimal asset allocation, incorporating investor objectives and satisfaction indices used in practical implementations;

2) Identification of bottlenecks for the mean variance framework which can be adapted to work in hardware;

3) Design of the proposed hardware for the FPGA implementation of the mean variance framework;

4) A study of potential performance improvements through simulations of the hardware architectures and a comparison between a software implementation running on two 2.4 Ghz Pentium-4 CPUs, and an FPGA architecture, showing potential performance ratios of 9.6 × and 221 × for different steps. This study proves the need for automatic generation of multi-core architectures on FPGAs for acceleration of financial computations.

The rest of this chapter is organized as follows: In section 8.1, we describe the steps of the mean variance framework used for optimal asset allocation. In section 8.2, we present our proposed implementation of the mean variance framework. Section 8.3 presents our results in terms of timing and throughput and compares these results with a purely software implementation. We conclude in section 8.4.

8.1 The Mean Variance Framework for Optimal Asset Allocation

In this section, we present the mean variance framework for optimal asset allocation. The framework is a popular two-step approach used in all practical asset allocation applications. Step 1 selects efficient allocations among all the possible combinations of assets and computes the efficient frontier. Step 2 performs a search for the best among the efficient allocations using Monte-Carlo simulations. We divide our discussion of the framework into three sections (shown in Figure 8.2):

A. The computation of inputs required for Step 1 of the mean variance framework,

B. Step 1 of the mean variance framework,

C. Step 2 of the mean variance framework.

Note that all equations listed in the following subsections are found in [30] unless otherwise specified.

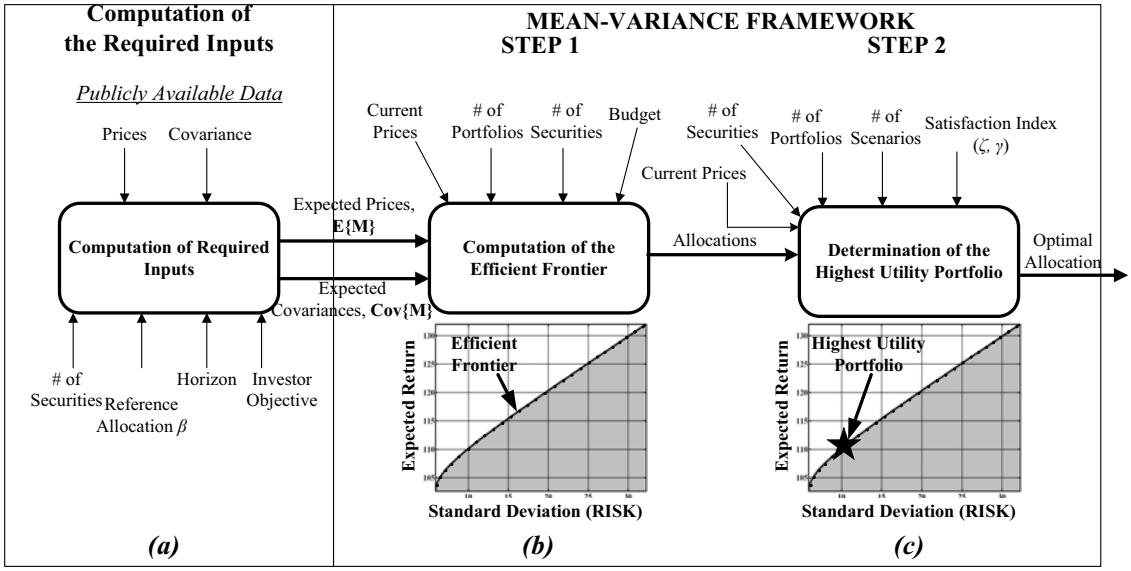


Figure 8.2: **The required steps for optimal asset allocation are shown in (a), (b) and (c). After the required inputs to the mean variance are generated in (a), computation of the efficient frontier and determination of the highest utility portfolio are shown in (b) and (c) respectively. This figure also presents the inputs and outputs provided to the user.**

8.1.1 Computation of the Required Inputs

The expected values of the market vector $E\{M\}$ and the respective covariance matrix $Cov\{M\}$ are needed as inputs to the mean variance framework. Calculating these inputs requires the use of already known publically available data: prices, standard deviation, and covariances plus the investor's objective, number of securities, N_s, reference allocation and horizon τ (as shown in Figure 8.2(a)). Calculating $E\{M\}$ and $Cov\{M\}$ using these investor and market parameters requires the 5 stage procedure (shown in Figure 8.3) explained in detail below [30].

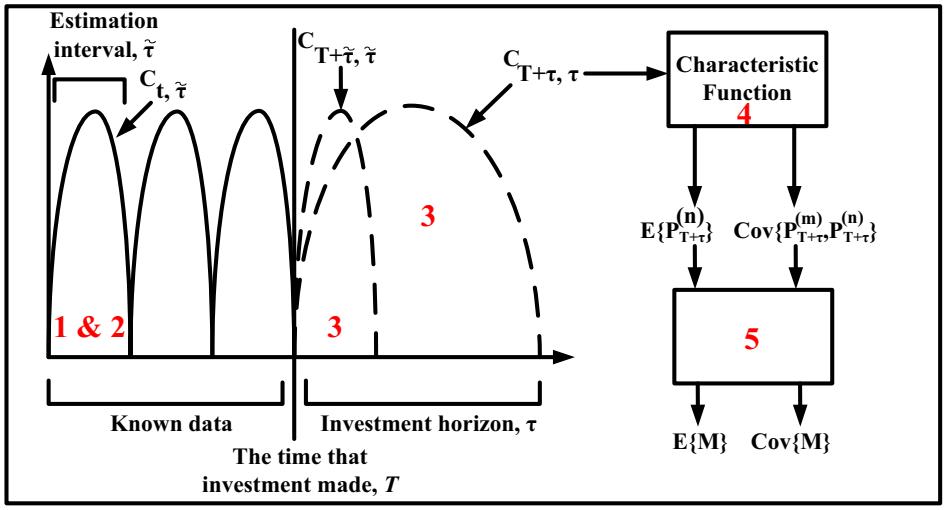


Figure 8.3: **The procedure to generate required inputs is described. The numbers 1-5 refers to these computation steps which are explained in subsections in more detail.**

1. Detection of the invariants, $X_{t,\tilde{\tau}}$

Invariants are identical repetitions in the market within a given estimation interval, $\tilde{\tau}$. The estimation interval, $\tilde{\tau}$, is different than the horizon τ, which was mentioned before. The estimation interval, $\tilde{\tau}$, refers to the time which we suspect a repetition in data, where investment horizon, τ, refers to the time the investor plans to invest. Detection of the invariants is an essential step and linear return of stocks $L_{t,\tilde{\tau}} = \frac{P_t}{P_{t-\tau}} - 1$, or compounded return of stocks $C_{t,\tau} = \ln(\frac{P_t}{P_{t-\tau}})$ can be used as invariants for the market. We chose to use compounded return of stocks.

2. Determination of the distribution of the invariants

We can determine the distribution of the invariants based on estimators (maximum likelihood estimators, nonparametric estimators etc.) based on current market information. As an example, we assume that the invariants $\mathrm{X}_{t,\tilde{\tau}} = C_{t,\tilde{\tau}}$ are multivariate normal distribution with $N(\hat{\mu}, \hat{\Sigma})$ where $\hat{\mu}$ and $\hat{\Sigma}$ are vectors of sample mean and covariance matrix respectively.

3. Projection of the invariants $X_{t,\tilde{\tau}}$ to the investment horizon to obtain the distribution of $X_{T+\tau,\tau}$

After the determination of the distribution of the invariants $\mathrm{X}_{t,\tilde{\tau}}$ in an estimation interval, $\tilde{\tau}$, we project them to the investment horizon $\mathrm{X}_{T+\tau,\tau} \sim N(\frac{\tau}{\tilde{\tau}}\hat{\mu}, \frac{\tau}{\tilde{\tau}}\hat{\Sigma})$. Furthermore we use this distribution to determine the distribution of the market prices, $\mathrm{P}_{T+\tau}$.

4. Computation of the expected return $E\{P_{T+\tau}\}$, and the covariance matrix $Cov\{P_{T+\tau}\}$ from the distribution of the market prices

We use the characteristic function of the compounded returns to formulize the expected returns as

$$E\{P^{(n)}_{T+\tau}\} = P^{(n)}_{T} e^{\frac{\tau}{\tilde{\tau}}(\hat{\mu}_n + \frac{\hat{\Sigma}_{nn}}{2})} \tag{8.1}$$

and covariance matrix of the market as:

$$Cov\{P^{(m)}_{T+\tau}, P^{(n)}_{T+\tau}\} = P^{(m)}_{T} P^{(n)}_{T} e^{\frac{\tau}{\tilde{\tau}}(\hat{\mu}_m + \hat{\mu}_n)} e^{\frac{1\tau}{2\tilde{\tau}}(\hat{\Sigma}_{mm} + \hat{\Sigma}_{nn})} (e^{\frac{\tau}{\tilde{\tau}}(\hat{\Sigma}_{mn})} - 1) \tag{8.2}$$

5. Computation of the expected return $E\{M\}$, and the covariance matrix $Cov\{M\}$ of the market vector

An investor objective is a function for which every investor desires the largest value as an output of that function. There are different objectives such as absolute wealth, relative wealth and net profits [30]. An *absolute wealth* investor tries to maximize the value of the portfolio in the investment horizon. A *relative wealth* investor tries to achieve better portfolio return compared to a reference portfolio where the reference portfolio is denoted as β with γ as a normalization factor. A *net profits* investor always tries to increase the value of the portfolio compared to the value of the portfolio today. The specific forms of the equations for these objectives are shown in Table 8.1(a).

These different objectives can be seen as a linear function of the investor's

Table 8.1: Different Investor Objectives: Specific and Generalized Forms

	Standard Investor Objectives		
	Absolute Wealth	Relative Wealth	Net Profits
(a) Specific Form	$\psi_\alpha = W_{T+\tau}(\alpha)$	$\psi_\alpha = W_{T+\tau}(\alpha) - \gamma(\alpha)W_{T+\tau}(\beta)$	$\psi_\alpha = W_{T+\tau}(\alpha) - w_T(\alpha)$
(b) Generalized Form	$a \equiv 0, B \equiv I_N$ $\psi_\alpha = \alpha^{'} P_{T+\tau}$	$a \equiv 0, B \equiv K$ $\psi_\alpha = \alpha^{'} K P_{T+\tau}$	$a \equiv -p_T, B \equiv I_N$ $\psi_\alpha = \alpha^{'}(P_{T+\tau} - p_T)$

allocation α, and the market vector M, shown as follows:

$$\psi_\alpha = \alpha M \tag{8.3}$$

M is a transformation of the market prices at the investment horizon as:

$$M \equiv a + BP_{T+\tau} \tag{8.4}$$

where a and B are a suitable conformable vector and an invertible matrix respectively. These generalized forms of investor objectives are also shown in Table 8.1(b) with different a and B values where $\gamma(\alpha) \equiv \frac{w_T(\alpha)}{w_T(\beta)}$ (Normalization factor), $K \equiv I_N - \frac{p_T \beta^{'}}{\beta^{'} p_T}$ and I_N is identity matrix. Computation of the market vector combines the expected returns and covariance matrix with the investor objectives using different a and B values for different investor objectives which is shown as:

$$E\{M\} = a + BE\{P_{T+\tau}\} \tag{8.5}$$

$$Cov\{M\} = BCov\{P_{T+\tau}\}B^{'} \tag{8.6}$$

Notice that each step requires the financial analyst to make assumptions (such as what type of invariant distribution to assume, and what estimation interval to use). Each assumption affects the outcome of the computation and hence each of the five steps described is a broad research area in economics. For our purposes we use the following assumptions with the knowledge these could be easily changed: we use the past 3 years of the data with 1 week estimation interval. We use compounded returns of stocks as market invariants and assume that they are multivariate random variables. We assume our estimation horizon is 1 year.

8.1.2 Mean Variance Framework Step 1: Computation of the Efficient Frontier

Computing the efficient frontier, the efficient allocations of different assets for different risks, is the first step of the mean variance framework (Figure 8.2(b)). The inputs to this step are current prices (already known), expected prices, $E\{M\}$, and expected covariance matrix, $Cov\{M\}$,(which are calculated as described in 8.1.1), number of portfolios, N_p, number of securities, N_s and investor's budget. This step calculates N_p amount of efficient portfolios. These different portfolios create the curve in Figure 8.2(b) which is called the efficient frontier.

Assume an investor who purchases α_n units of the $n-th$ security in a market of N securities at time T (the time that the investment is made). If $P_T^{(n)}$ and α denote the price of the $n-th$ security at the time T and the allocation at the time the decision is made respectively, the value of the portfolio is calculated as:

$$W_T(\alpha) \equiv \alpha P_T \tag{8.7}$$

However, the market prices of the securities are multivariate random variables at the investment horizon, therefore the portfolio is a random variable which can be seen as:

$$W_{T+\tau}(\alpha) \equiv \alpha^{'} P_{T+\tau} \tag{8.8}$$

where $\alpha^{'}$ refers to the allocation at the horizon. Because the portfolio's value is a random value since the market prices are unknown, the expected prices at the investment horizon $E\{P_{T+\tau}\}$ and the covariance matrix $Cov\{P_{T+\tau}\}$ need to be computed and then investor objective function needs to be included to give us $E\{M\}$ and $Cov\{M\}$ (These calculations are shown in the previous section). The efficient frontier is then found by maximizing the investor objective value by a constrained variance. This computation can be seen as :

$$\alpha(v) \equiv \arg \max_{\alpha \in \mathbb{C}, \alpha^{'} Cov\{M\} \alpha = v} \alpha^{'} E\{M\}, v \geq 0 \tag{8.9}$$

where an investor's objective value and variance is calculated as follows:

$$E\{\psi_\alpha\} = \alpha' E\{M\} \tag{8.10}$$

$$Var\{\psi_\alpha\} = \alpha' Cov\{M\}\alpha \tag{8.11}$$

With the efficient frontier depending on how much risk an investor wants to face, there is a corresponding expected return. The region which is below the black curve (the shaded region in Figure 8.2(b)) corresponds to the achievable risk-return space for the specific frontier which includes at least one portfolio constructible from the investments that has the risk and return corresponding to that point. The upper region is the unachievable risk-return space. The black curve running along the top of the achievable region is the efficient frontier. The portfolios that correspond to points on that curve are optimal according to Equation 8.9.

8.1.3 Mean Variance Framework Step 2: Computing the Optimal Allocation

Now that we have generated the inputs for the mean variance framework and used these inputs to compute the efficient frontier, we have to consider satisfaction indices to determine which *point* along the efficient frontier represents the optimal allocation for the given investor. The required inputs to this step are the allocations computed in step 1, current prices, number of portfolios, N_p, number of securities, N_s, number of scenarios, N_m, and investor satisfaction index (as shown in Figure 8.2(c)).

The investor objective function produces one value. However this value is random since the market prices at the investment horizon are stochastic and therefore the market vector, M, contains random variables. Therefore, using the investor function alone does not allow us to select the optimal allocation because we have no way of determining which random value output is *better* for the investor than another. Therefore we need to compute the expected value of the investor objective value by introducing satisfaction indices [30]. Satisfaction indices represent all the features of a given allocation with one single number and quantify the investor's satisfaction. Therefore, an investor prefers an allocation to the other if

it provides more satisfaction. There are mainly three different classes of indices being used to model the investor's satisfaction: certainty-equivalent, quantile and coherent indices. We use certainty-equivalent indices because they are based on a concave function and promote diversification [30].

Certainty-equivalent indices are represented by the investor's utility function and objective. A *utility function* $u(\psi)$ is defined for an investor to explain his enjoyment. There are different utility functions which we can use to represent an investor's satisfaction such as exponential, quadratic etc. Even though this function is specific for every investor, it is possible to investigate the most commonly used functions and generalize them [30]. We show these different utility functions in Table 8.2. To generalize the creation of utility functions, we use Hyperbolic Absolute Risk Aversion (HARA) class of utility functions which are specific forms of the Arrow-Pratt risk aversion model and defined in [30, 50] as

$$A(\psi) \equiv \frac{\psi}{\gamma\psi^2 + \zeta\psi + \eta} \tag{8.12}$$

where $\eta \equiv 0$. The HARA class of utility functions gives us most of the utility functions by varying the constants, ζ and γ as shown in Table 8.2.

Table 8.2: Different Utility Functions for Satisfaction Indices

Utility Functions				
Exponential Utility	Quadratic Utility	Power Utility	Logarithmic Utility	Linear Utility
($\zeta > 0$ and $\gamma \equiv 0$)	($\zeta > 0$ and $\gamma \equiv -1$)	($\zeta \equiv 0$ and $\gamma \geq 1$)	($lim_{\gamma \to 1}\gamma$)	($lim_{\gamma \to \infty}\gamma$)
$u(\psi) = -e^{\frac{1}{\zeta}\psi}$	$u(\psi) = \psi - \frac{1}{2\zeta}\psi^2$	$u(\psi)^{1-\frac{1}{\gamma}}$	$u(\psi) = \ln\psi$	$u(\psi) = \psi$

Therefore, an investor compares different allocations using the index of satisfaction and chooses the maximum value as the optimal asset allocation. Computing the optimal allocation is a maximization problem using different market scenarios since market values are uncertain and its analytical solution is not possible in many practical implementations [30]. Therefore approximation methods are employed for finding the best allocation on the efficient frontier. To solve this problem with approximations, a large number of market scenarios are simulated through Monte-Carlo simulations.

8.2 Implementation of the Mean Variance Framework

Now that we have described how optimal asset allocation works, we now discuss our proposed implementation of the mean variance framework. We first present a series of figures to provide the motivation for our implementation and determine the bottlenecks of optimal asset allocation. We then describe the proposed architectures and possible ways to benefit from their inherent parallelism.

8.2.1 Implementation Motivation

As previously shown in Figure 8.1, increasing the number of securities in a portfolio allows the investor to achieve better investment opportunities, thus our goal is to allow for a large number of diversified securities in a portfolio. **But how much computation time does increasing the number of securities add to the computation of the optimal asset allocation?** To address this question we looked at how varying the number of securities affected the computation time in relation to number of portfolios and number of scenarios, two other important parameters that affect computation time (an increase in the number of portfolios increases the number of points on the efficient frontier and increasing the number of scenarios increases the number of runs of the Monte-Carlo simulation). Figure 8.4(a), 8.4(b) and 8.4(c) compare number of securities, N_s, versus number of portfolios, N_p; number of portfolios, N_p, versus number of scenarios, N_m; number of securities, N_s, versus number of scenarios, N_m, respectively. By looking at the slopes of the lines in the figures it can be easily seen that N_s dominates computation time (has a steeper slope) over N_p (a), and N_p dominates computation time over N_m (b). These results suggest that the number of securities is the most computationally time sensitive input to the optimal asset allocation problem, thus if a large number of securities are to be allowed as input to the framework, a faster implementation must be developed.

To identify the bottlenecks of the computation of the optimal asset allocation, we look at the runtime of each solution step (**1.** generation of the required

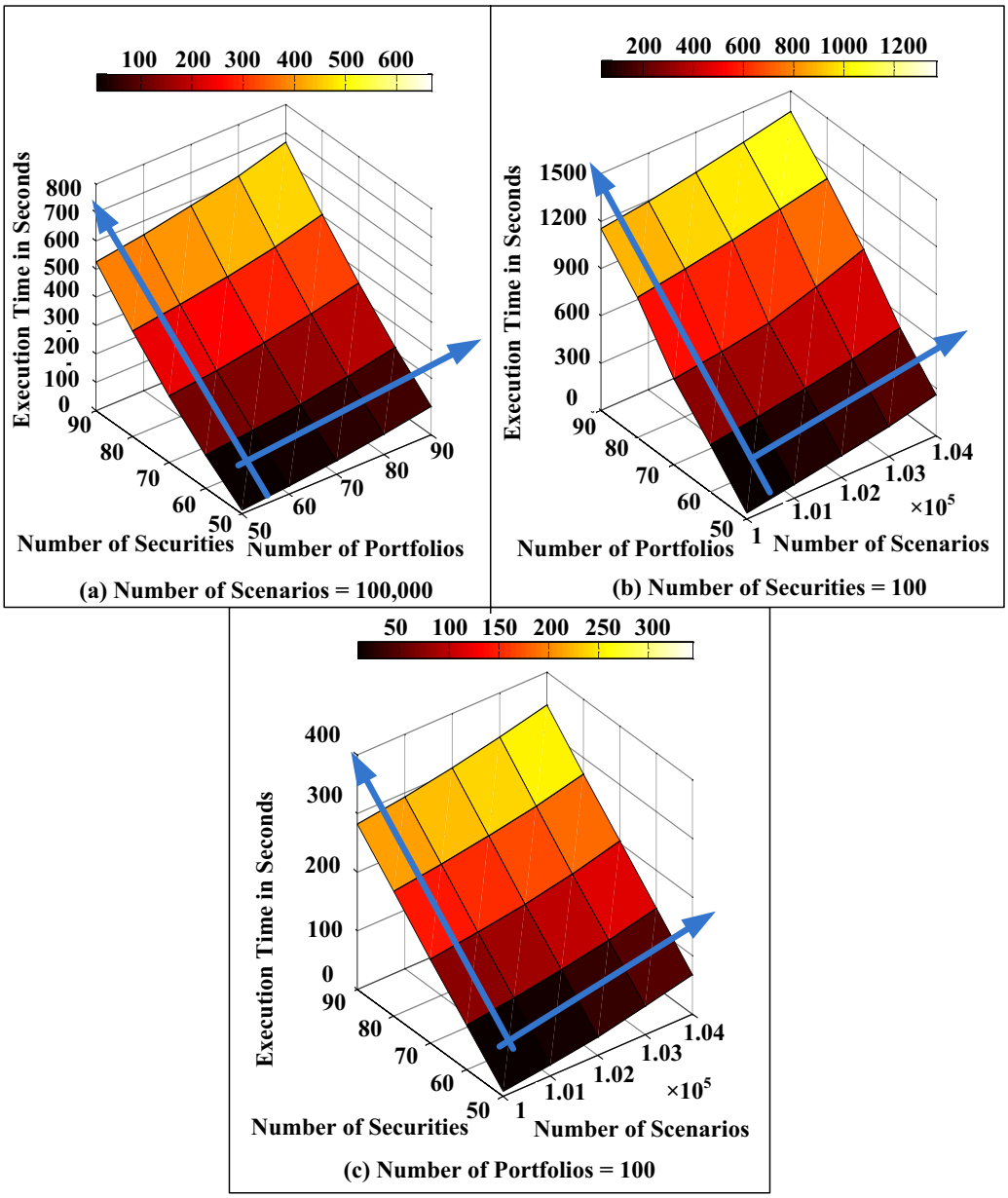


Figure 8.4: **To determine the computation time of different variables, we compare number of securities, N_s, versus number of portfolios, N_p, and number of portfolios, N_p, versus number of scenarios, N_m, respectively. By looking at the slopes of these lines in the figures it can be easily seen that N_s dominates computation time (has a steeper slope) over N_p (a), N_p dominates computation time over N_m (b).**

inputs, **2.** Step 1 of the mean variance framework, and **3.** Step 2 of the mean variance framework) with respect to varying the number of securities (Figure 8.5(a)),

number of portfolios (Figure 8.5(b)) and number of Monte-Carlo simulations (Figure 8.5(c)). As can be seen from Figure 8.5(a), the generation of the required inputs does not consume a significant amount of time, thus it is best to keep this implementation step in software if the computation cannot be parallelized. On the other hand, step 1 and 2 of the mean-variance framework consume a significant amount of time providing the motivation for an alternative implementation. It is also important to note there is a cutoff point between step 1 and 2, showing that the computational time for step 1 becomes more significant after 60 securities. In Figure 8.5(b), we only compare step 1 and 2 for different number of portfolios (because we already determine that the computational time for the generation of required steps is not significant), and we conclude that most time consuming part is step 1. Figure 8.5(c) shows the increase in the timing with varying number of scenarios.

8.2.2 Hardware/Software Interface

As determined in 8.2.1, Step 1 and Step 2 of the mean variance framework are the bottlenecks for computing the optimal asset allocation. FPGA implementations can provide a substantial performance improvement for processes that can be easily parallelized. Fortunately, finding the maximum return for different risk values to create the efficient frontier (Step 1) and implementing Monte-Carlo simulations to apply different market scenarios (Step 2) can be easily parallelized making them good candidates for hardware implementations. Although the generation of required inputs is not a bottleneck for optimal asset allocation, further performance improvement can be gained by implementing phase 5 of this step (computation of $E\{M\}$ and $Cov\{M\}$) which includes parallizable matrix computations in hardware. Thus, our implementation combines software (a Host PC) to compute phases 1-4 of the generation of required inputs and hardware (FPGA) to compute phase 5 of the generation of required inputs and step 1 and step 2 of the mean variance framework to obtain maximal performance gain.

Because our implementation combines hardware and software, we must pay particular attention to the hardware/software interface, especially to the data that

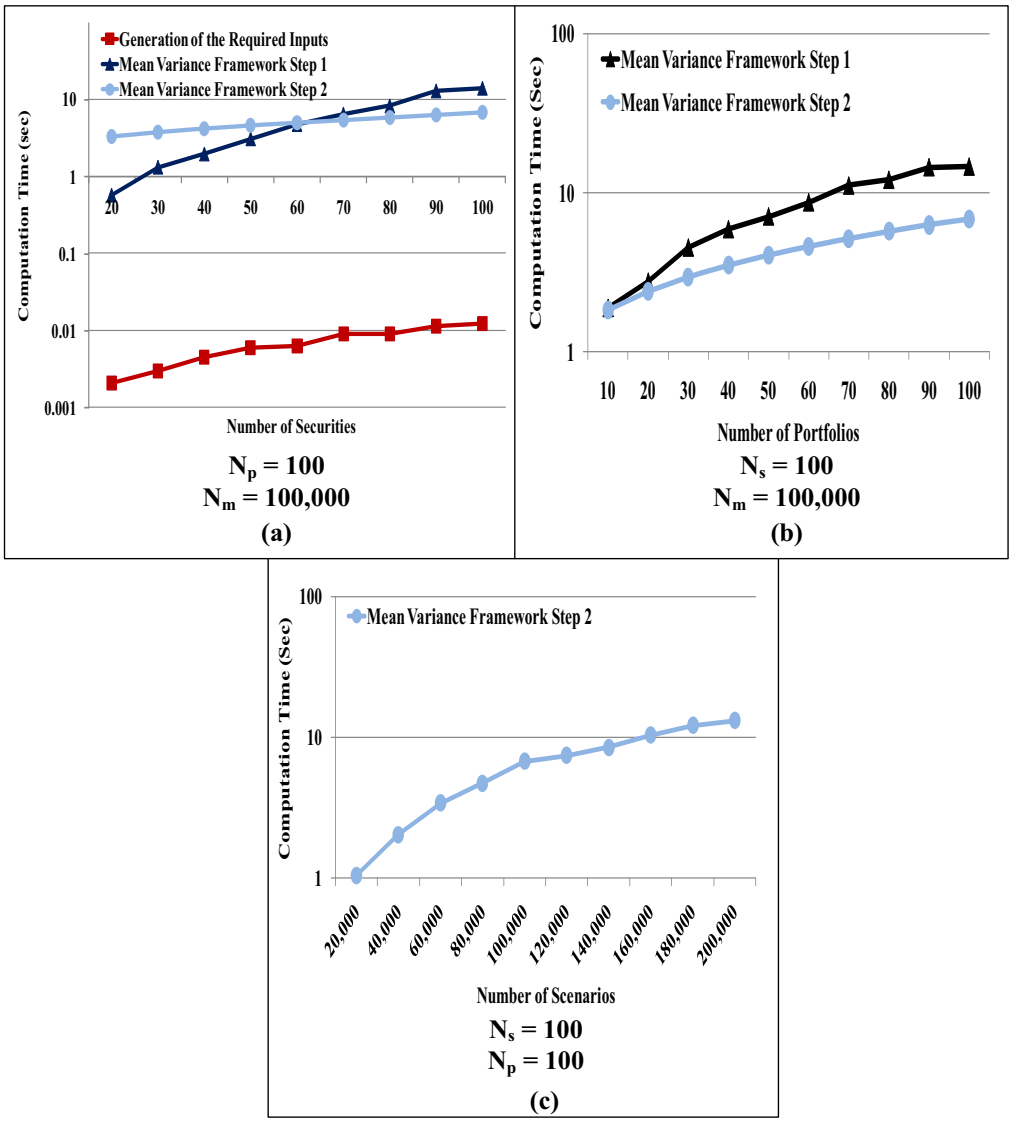


Figure 8.5: **Identification of the bottlenecks in the computation of the optimal asset allocation. We run two different test while holding all but one variable constant. We determined that generation of the required input does not consume significant amount of time. On the other hand, step 1 and 2 of the mean variance framework consumes significant amount of time.**

needs to be transferred, to insure we do not lose the performance gain we added through the hardware software separation. The information that needs to be transferred between the software and hardware are current prices, expected prices, and expected covariances which are of the dimensions $N_s \times 1$, $N_s \times 1$ and $N_s \times N_s$ respectively. In the following subsections, we present our architectural design and

parallelization possibilities for the generation of the required inputs phase 5, mean variance framework step 1 and 2.

8.2.3 Generation of Required Inputs - Phase 5

We implement *"Market Vectors Calculator IP Core"* for the calculation of phase 5 in the generation of the required inputs (Figure 8.6). This IP Core can compute three different objectives: absolute wealth, relative wealth and net profits or any combination of these which are described in 8.1.1. This IP Core includes the *K building block* which computes the constant matrix K and used if the investor's objective is relative wealth. The required inputs to this hardware are current prices, P_T, reference allocation, β', identity matrix, I_N, and expected returns, $P_{T+\tau}$. We use two control inputs: *cntrl_a* and *cntrl_b* to select the desired investor profile. These control relationships are described in Table 8.3.

After these control units are given, $E\{M\}$, the market vectors at the investment

Table 8.3: Control Inputs for Different Investor Objectives

	Control Inputs	
Investor Objective	cntrl_a	cntrl_b
Absolute Wealth	0	0
Relative Wealth	1	0
Net Profits	0	1

horizon, is calculated. Figure 8.7 shows how the *Market Vectors Calculator IP Core* can be easily parallelized. $Cov\{M\}$, computed by Equation 8.6 is only needed when the investor objective is relative wealth. Because it also includes many matrix multiplications and accumulations, a similar parallelized hardware can be implemented.

8.2.4 Hardware Architecture for Mean Variance Framework Step 1

Mean variance framework step 1 is a constrained maximization problem which is shown in Equation 8.9. This step receives market vectors, $E\{M\}$ and

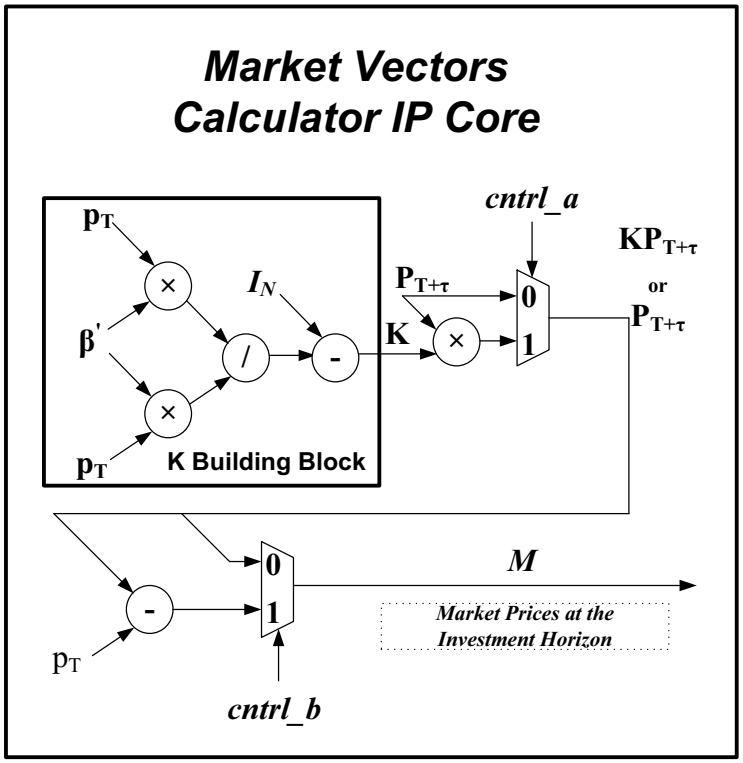


Figure 8.6: **Parameterizable serial hardware architecture for the generation of the required inputs - phase 5.**

$Cov\{M\}$ as inputs and maximizes the expected return for a specific standard deviation (risk) to find the efficient portfolio. This maximization problem needs to be solved for different risks number of portfolios, N_p, times. A simple example of this maximization problem can be seen as:

$$\alpha(v) \equiv \arg \max_{\alpha' Cov\{M\}\alpha = v} \alpha' E\{M\}, v \geq 0 \tag{8.13}$$

where two possibly important constraints: the budget of the investor and $\sum_{i=1}^{N_s} \alpha_i = 1$ are not added the for ease of understanding.

A popular approach to solve constrained maximization problems is to use the *Lagrangian multiplier method* [51] which introduces an additional variable, λ, to equalize the number of equations and number of unknowns. The equations for the solution of the Equation 8.13 for 2 securities can be seen as:

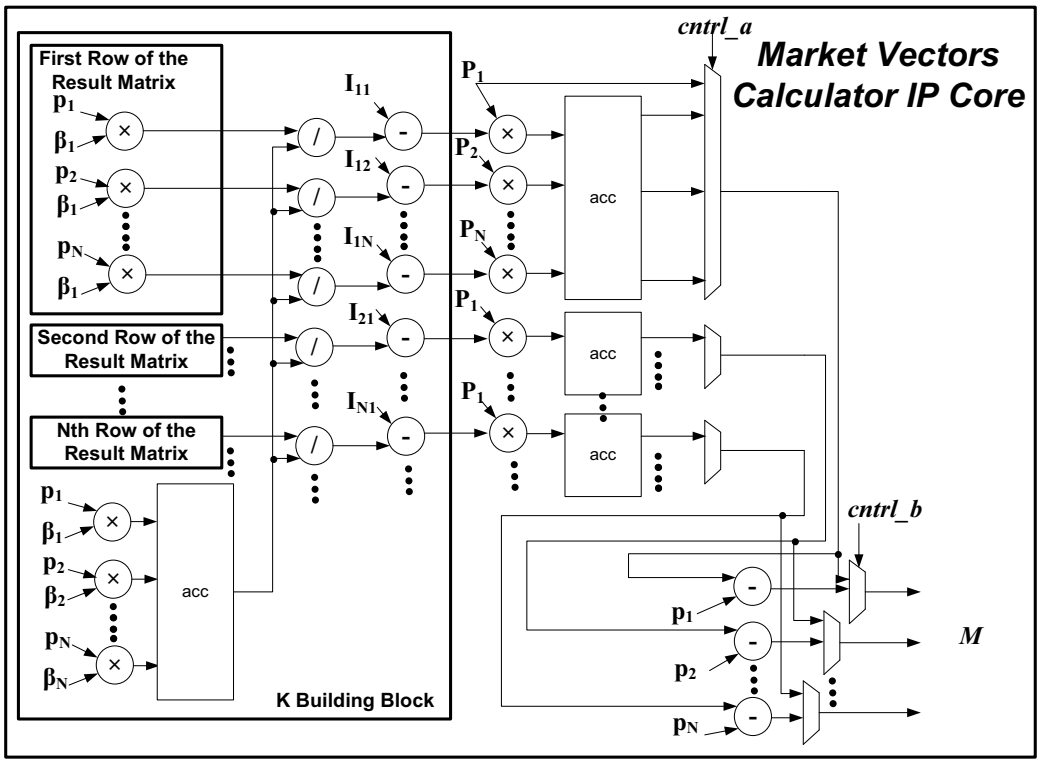


Figure 8.7: **Parameterizable fully parallel hardware architecture for the generation of the required inputs - phase 5. As can be seen from the parallel architecture, phase 5 has very high potential for the parallel implementation, therefore a good candidate for decreasing the computational time of the optimal asset allocation.**

$$\mathcal{L} = \alpha^{'} E\{M\} + \lambda(v - \alpha^{'} Cov\{M\}\alpha) \tag{8.14}$$

$$\mathcal{L} = [\alpha_1 \alpha_2] \begin{bmatrix} P_1 \\ P_2 \end{bmatrix} + \lambda(v - [\alpha_1 \alpha_2] \begin{bmatrix} Cov_{11} & Cov_{12} \\ Cov_{21} & Cov_{22} \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \end{bmatrix}) \tag{8.15}$$

$$\frac{\partial \mathcal{L}}{\partial \alpha_1} = P_1 - \lambda[2\alpha_1 Cov_{11} + \alpha_2(Cov_{21} + Cov_{12})] = 0 \tag{8.16}$$

$$\frac{\partial \mathcal{L}}{\partial \alpha_2} = P_2 - \lambda[2\alpha_2 Cov_{22} + \alpha_1(Cov_{21} + Cov_{12})] = 0 \tag{8.17}$$

$$\frac{\partial \mathcal{L}}{\partial \lambda} = \alpha_1^2 Cov_{11} + \alpha_1 \alpha_2 (Cov_{21} + Cov_{12}) + \alpha_2^2 Cov_{22} = v \tag{8.18}$$

By solving three equations for three unknowns, α_1, α_2, λ, one can derive the optimal α_1 and α_2 values where calculation of these values can be written

as functions of the known constants such as $v(Risk)$, P_1 and Cov_{22}. A number of securities, N_s, amount of functions need to be computed for determination of the efficient allocation for a given risk. These equations will be the same for different risks and hence can be easily parallelized (as shown in Figure 8.8). There are different α calculator blocks in every core. This core can be used serially by applying different variances as inputs or can be parallelized since the equations these cores include are the same.

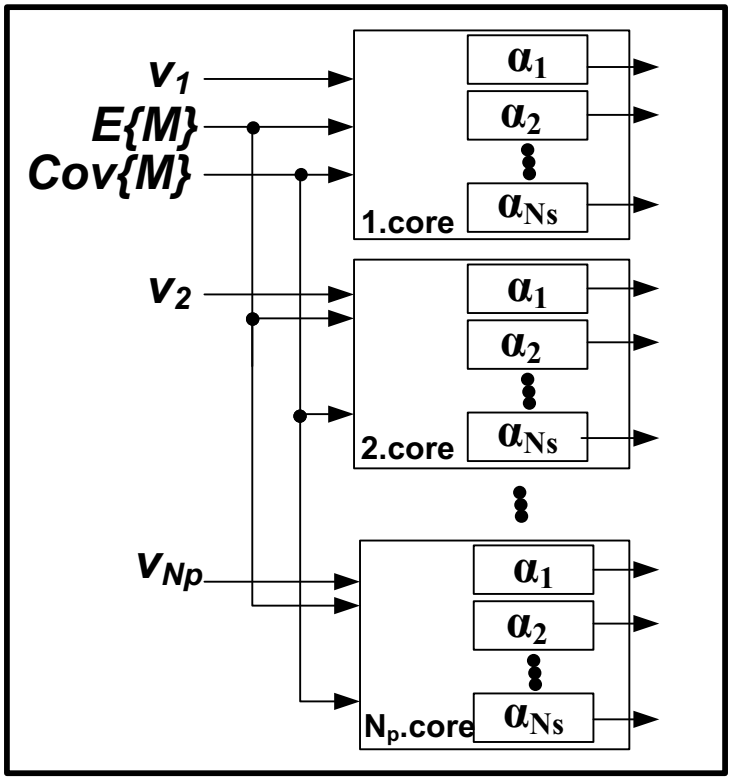


Figure 8.8: **Parallel optimum allocation calculator IP Cores.**

8.2.5 Hardware Architecture for Mean Variance Framework Step 2

After computing the efficient frontier, we determine the highest utility allocation (optimal allocation) among these different allocations using satisfaction

indices. Computing the optimal allocation is a maximization problem by simulating a large amount of market scenarios through Monte-Carlo simulations. *The Satisfaction Function Calculator IP core* (Figure 8.9) has required inputs of the investor objective function values ψ, and the constants ζ and γ which are defined in Equation 8.12. *The Satisfaction Function Calculator IP core* can evaluate linear, logarithmic, exponential, quadratic, and power utility functions. The control input, *cntrl_c*, defines which utility to use.

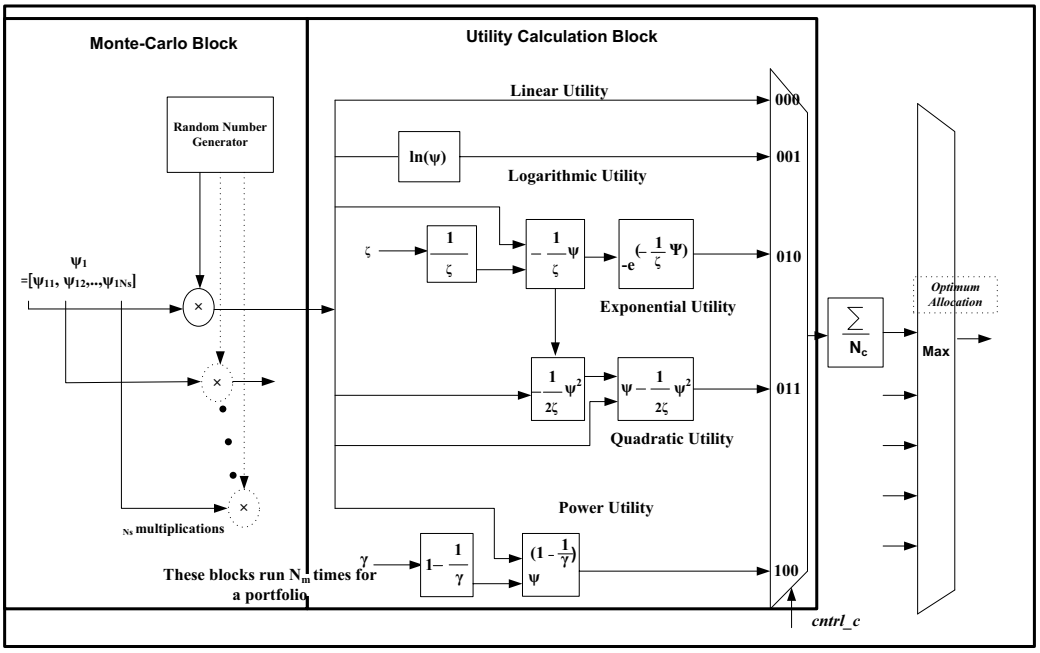


Figure 8.9: **Parallel parameterizable hardware architecture for the mean variance framework step 2.**

For the determination of the highest utility allocation, *the Monte-Carlo block* and *the Utility Calculation Block* (as part of the Satisfaction Function Calculator block) are run number of simulations, N_m, times. The whole Satisfaction Function Calculator IP core is then run number of portfolios, N_p, times. Therefore, the Monte-Carlo block, Utility Calculation Block, and Satisfaction Function Calculator IP core can be easily parallelized a maximum of N_m, N_m and N_p times respectively as shown in Figure 8.10.

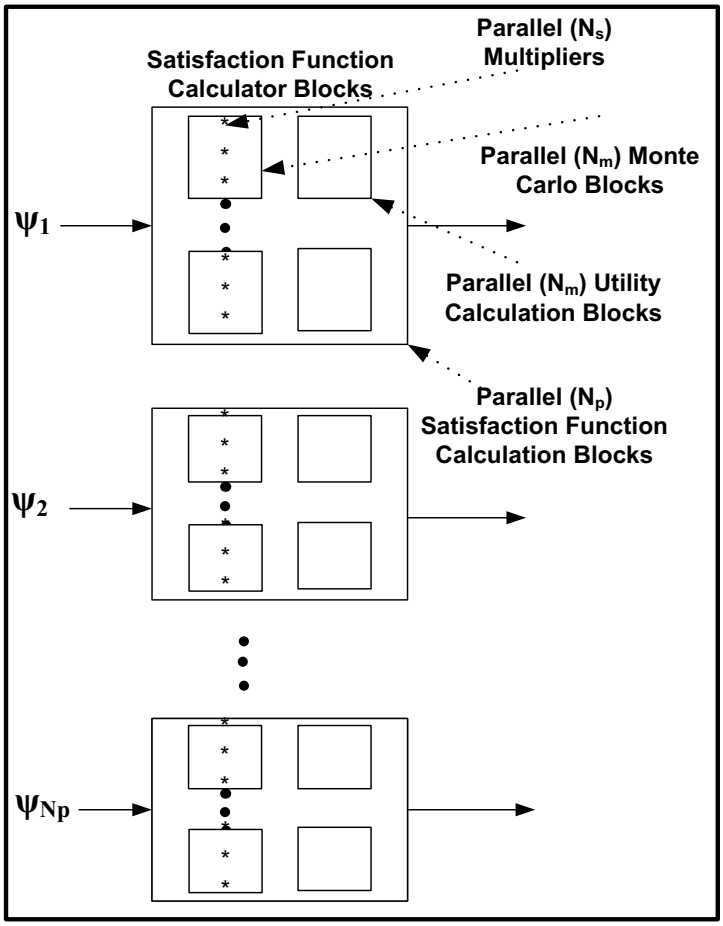


Figure 8.10: **Parallel parameterizable hardware architecture for the mean variance framework step 2. The Monte-Carlo block, Utility Calculation Block, and Satisfaction Function Calculator IP core can be easily parallelized a maximum of N_m, N_m and N_p times respectively.**

8.3 Results

In this section, we investigate potential speed-ups for the mean variance framework using simulations of the hardware architectures we described in 8.2. We concentrate on "Generation of the required inputs - Phase 5" and "the mean variance framework - step 2." We consider serial and different level of parallel implementations of these steps and compare our results with a software implementation running on two 2.4 Ghz Pentium-4 CPUs (every test is run 1000 times and average runtime is presented). We use 32 bit fixed-point arithmetic for our implementa-

tions and assume that our clock frequency achieves 200 MHz. The complexity of the mean variance framework step 1 increases dramatically with increased securities, and hence its potential runtime cannot be determined until we investigate alternative parallelism methods (such as employing Monte-Carlo simulations) and hence is not presented.

As can be seen from Figure 8.6 and 8.7, the market vector calculator IP Core can be implemented with different levels of parallelism levels where we are bound by hardware resources rather than by the parallelism that this step offers. *The serial implementation* of this step (no parallelism exploited) performs poorly compared to the software implementation. *The parallel implementation* uses a reasonable parallelism level by employing N_s number of arithmetic resources in parallel: for 50 securities there are 50 multipliers, dividers, subtractors etc. This level of parallelism achieves a potential performance ratio between **6 × and 9.6 ×** compared to the software implementation. *A fully parallel implementation* which might not be realistic due to hardware limitations, presents a best potential bound offering a performance ratio **629 ×** (for 50 securities). This comparison is shown in Figure 8.11.

We investigate the difference in timing for mean variance framework step 2 in Figure 8.12. We use 100,000 scenarios, N_m, for Monte-Carlo simulations and 50 portfolios, N_p, to evaluate. We present two parallel architectures, parallel 1 employs 10 Satisfaction Function Calculator blocks where each consists of 1 Monte-Carlo block with 10 multipliers and 10 Utility Function Calculator blocks. Parallel 2 employs 10 Satisfaction Function Calculator blocks where each consists of 1 Monte-Carlo block with 20 multipliers and 20 Utility Function Calculator blocks. Parallel 1 and Parallel 2 offer a potential performance ratio between **151 × and 221 ×** and between **302 × and 442 ×**.

As can be seen from the potential performance ratios, both "Generation of the required inputs - phase 5" and "mean variance framework - step 2" offer significant speed-up when parallelized and implemented in hardware.

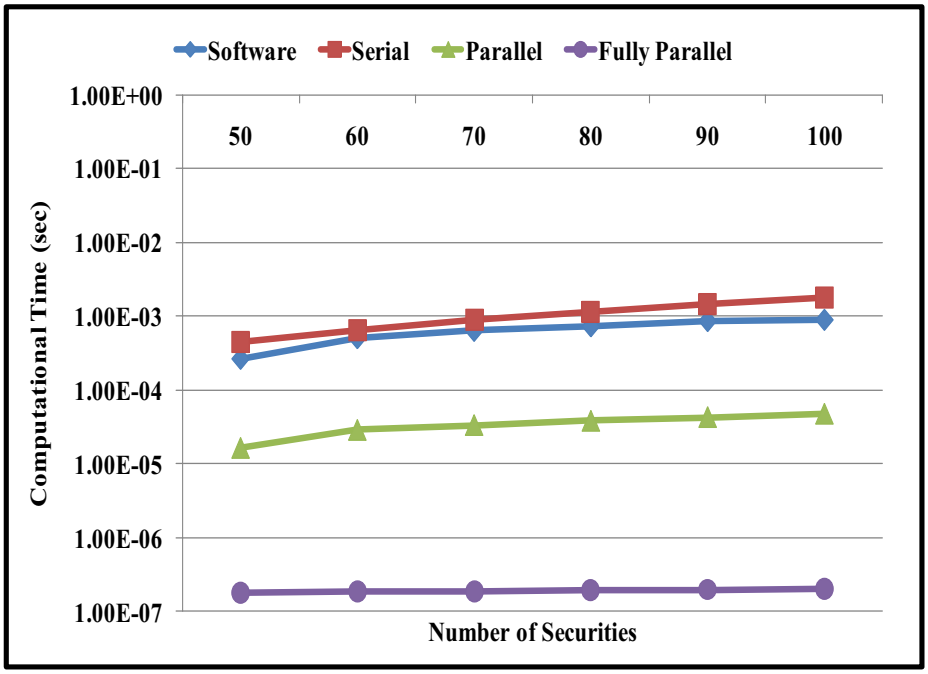


Figure 8.11: **Possible speed-ups for "generation of the required inputs - phase 5"**

8.4 Conclusions

The addition of FPGAs to the existing high performance computers can boost an application's performance and design flexibility. The mean variance framework's inherent parallelism in its solution steps (due to many matrix computations and its use of Monte Carlo simulations) and its need for reprogramability (to allow for modifications based on different investor characteristics) make the framework an ideal candidate for an FPGA implementation. In this work, we are the first to propose hardware acceleration of optimal asset allocation through an FPGA implementation of Markowitz' mean variance framework. We concentrate on "Generation of the required inputs - Phase 5" and "Mean-variance Framework - Step 2" in this work and present a study of potential performance improvements through simulations of the hardware architectures. We provide a comparison between a software implementation running on two Pentium-4 CPUs, and an FPGA architecture, showing potential performance gains of **9.6** $\times$ for Phase 5 and **221**

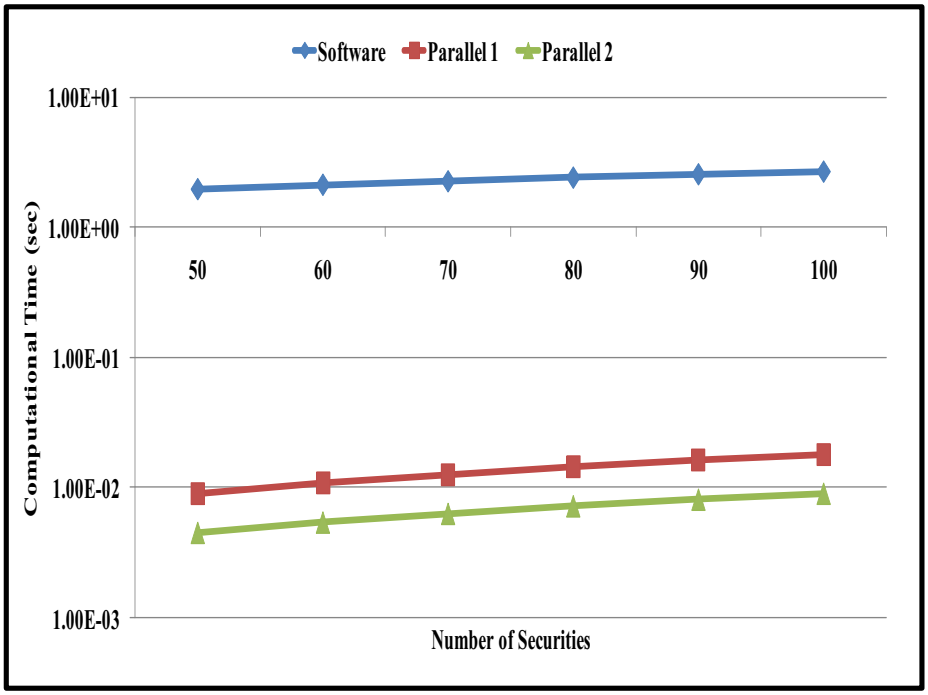


Figure 8.12: **Possible speed-ups for "Mean Variance Framework Step 2".**

× for Step 2. This study proves the need for automatic generation of multi-core architectures on FPGAs for acceleration of financial computations.

The text of Chapter 8 is in part a reprint of the material as it appears in the proceedings of the Workshop on High Performance Computational Finance. The dissertation author was the primary researcher and author and the co-authors listed on this publication [180] directed and supervised the research which forms the basis for Chapter 8.

Chapter 9

Future Research Directions

There are many possible directions for future research. I will touch on a few directions that could be explored using and improving my tool, GUSTO.

Vector Processing Unit Designs: Techniques for improving optimization results by automatic generation of vector processing elements with hierarchical datapaths.

Study of Trimming Effects for Different Scheduling Algorithms: GUSTO currently uses list scheduling. Different scheduling algorithms will provide us different results in terms of area, timing and throughput. Customization of matrix computation algorithms by exploiting instruction scheduling algorithms that have different effects on optimizations performed.

Supporting Conditional Statements: GUSTO currently doesnt support conditional statements in algorithms. Supporting conditional statements will improve GUSTO's capabilities.

Study of Effects on Architectural Result with Different Arithmetic Systems: Hardware implementation trade-offs for standard fixed-point and floating point arithmetic as well as non-standard arithmetic such as logarithmic number systems.

Appendix A

Matrix Computations

A.1 Matrix Decomposition Methods

A.1.1 QR Decomposition

QR decomposition is one of the most important operations in linear algebra. It can be used to find matrix inversion, to solve a set of simulations equations or in numerous applications in scientific computing. It represents one of the relatively small numbers of matrix operation primitive from which a wide range of algorithms can be realized.

QR decomposition is an elementary operation, which decomposes a matrix into an orthogonal and a triangular matrix. QR decomposition of a real square matrix A is a decomposition of A as $A = Q \times R$, where Q is an orthogonal matrix ($Q^T \times Q = I$) and R is an upper triangular matrix. And we can factor $m \times n$ matrices (with $m \geq n$) of full rank as the product of an $m \times n$ orthogonal matrix where $Q^T \times Q = I$ and an $n \times n$ upper triangular matrix.

There are different methods which can be used to compute QR decomposition. The techniques for QR decomposition are Gram-Schmidt orthonormalization method, Householder reflections, and the Givens rotations. Each decomposition method has a number of advantages and disadvantages because of their specific solution process. Each of these techniques are discussed in detail in the following sections.

1. Gram-Schmidt Orthonormalization Method

Gram-Schmidt method is a formulation for the orthonormalization of a linearly independent set. QR decomposition states that if there is an A matrix where $A \in \mathbb{R}^{n\times n}$, there exists an orthogonal matrix, Q, and an upper triangular matrix, R such that $A = Q \times R$, is the most important result of this orthonormalization. This method is used as algorithm to implement QR decomposition.

This decomposition can be seen as

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & \dots & A_{1m} \\ A_{21} & A_{22} & A_{23} & \dots & A_{2m} \\ A_{31} & A_{32} & A_{33} & \dots & A_{3m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & A_{n3} & \dots & A_{nm} \end{bmatrix} =$$

$$\begin{bmatrix} Q_{11} & Q_{12} & Q_{13} & \dots & Q_{1m} \\ Q_{21} & Q_{22} & Q_{23} & \dots & Q_{2m} \\ Q_{31} & Q_{32} & Q_{33} & \dots & Q_{3m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ Q_{n1} & Q_{n2} & Q_{n3} & \dots & Q_{nm} \end{bmatrix} \times \begin{bmatrix} R_{11} & R_{12} & R_{13} & \dots & R_{1m} \\ 0 & R_{22} & R_{23} & \dots & R_{2m} \\ 0 & 0 & R_{33} & \dots & R_{3m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & R_{nm} \end{bmatrix}$$

We can use another representation for simplification that is shown below:

$$\begin{bmatrix} a_1 & a_2 & a_3 & \dots & a_m \end{bmatrix} =$$

$$\begin{bmatrix} Q_1 & Q_2 & Q_3 & \dots & Q_m \end{bmatrix} \times \begin{bmatrix} r_{11} & r_{12} & r_{13} & \dots & r_{1m} \\ 0 & r_{22} & r_{23} & \dots & r_{2m} \\ 0 & 0 & r_{33} & \dots & r_{3m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & r_{nm} \end{bmatrix}$$

In order to illustrate the decomposition process, we supposed a set of column vectors Q_1, Q_2, Q_3, …, $Q_k \in \mathbb{R}^n$ which constructs the Q matrix as:

$$Q = \begin{bmatrix} Q_{11} & Q_{12} & Q_{13} & \dots & Q_{1m} \\ Q_{21} & Q_{22} & Q_{23} & \dots & Q_{2m} \\ Q_{31} & Q_{32} & Q_{33} & \dots & Q_{3m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ Q_{n1} & Q_{n2} & Q_{n3} & \dots & Q_{nm} \end{bmatrix} = \begin{bmatrix} Q_1 & Q_2 & Q_3 & \dots & Q_n \end{bmatrix}$$

These column vectors can be orthonormal if the vectors are pairwise orthogonal and each vector has euclidean norm of 1 [13]. In other words, Q is an orthogonal matrix where $Q \in \mathbb{R}^{n \times n}$ if and only if its columns form an orthonormal set which is the result of $Q \times Q^T = I$.

If we look at the result of the multiplication between Q and its transpose:

$$Q \times Q^T = \begin{bmatrix} Q_1 & Q_2 & Q_3 & \dots & Q_n \end{bmatrix} \times \begin{bmatrix} Q_1^T \\ Q_2^T \\ Q_3^T \\ \vdots \\ Q_n^T \end{bmatrix} =$$

$$\begin{bmatrix} Q_1Q_1^T & Q_2Q_1^T & Q_3Q_1^T & \dots & Q_nQ_1^T \\ Q_1Q_2^T & Q_2Q_2^T & Q_3Q_2^T & \dots & Q_nQ_2^T \\ Q_1Q_3^T & Q_2Q_3^T & Q_3Q_3^T & \dots & Q_nQ_3^T \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ Q_1Q_n^T & Q_2Q_n^T & Q_3Q_n^T & \dots & Q_nQ_n^T \end{bmatrix}$$

We see that the entries of $(Q \times Q^T)$ matrix are the inner products of the (Q_i, Q_j).Thus, $Q \times Q^T$ will be equal to I, *identity matrix*, if and only if the columns of the Q matrix form an orthonormal set. This can be shown as

$$(Q_i, Q_j) = \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j \end{cases} \quad \text{which results as } Q \times Q^T = I$$

Definition 1 - Given two vectors $x = (x_1, x_2, \dots, x_n)^T$ and $y = (y_1, y_2, \dots, y_n)^T$ in $\mathbb{R}^n$, we can define the inner product of x and y by

$$(x, y) = \sum_{i=1}^{n} x_i y_i \tag{A.1}$$

denotes as (x, y). It is important to note that the inner product has the following properties [13]:

$$(x, y) = (y, x) \tag{A.2}$$

$$(\alpha_1 x_1 + \alpha_2 x_2, y) = \alpha_1(x_1, y) + \alpha_2(x_2, y) \tag{A.3}$$

$$(x, \alpha_1 y_1 + \alpha_2 y_2) = \alpha_1(x, y_1) + \alpha_2(x, y_2) \tag{A.4}$$

for all $x, x_1, x_2, y, y_1, y_2 \in \mathbb{R}^n$ and $\alpha_1, \alpha_2 \in \mathbb{R}$.

Definition 2 - The Euclidean norm can be shown as $\|x\|_2 = \sqrt{(x, x)}$. There is an important relationship between inner product and Euclidean norm definitions.

To understand the decomposition process, we looked at the subspaces of $\mathbb{R}^n$ since we are working with columns of the matrices. Assume that there is a nonempty subset, ℓ, which is a subspace of $\mathbb{R}^n$ and this subset is closed under addition and scalar multiplication which is basically the result of inner product. That is, ℓ is a subspace of $\mathbb{R}^n$ if and only if whenever $a, w \in \ell$ and $c \in \mathbb{R}$, then $a + w \in \ell$ and $ca \in \ell$. Given vectors $a_1, a_2, \ldots, a_m \in \mathbb{R}$, a linear combination of $a_1, a_2, \ldots, a_m$ is a vector of the form $c_1a_1 + c_2a_2 + \cdots + c_ma_m$, where $c_1, c_2, \ldots, c_m \in \mathbb{R}$. We can call the numbers $c_1, c_2, \ldots, c_m$ as the coefficients of the linear combination. In sigma notation a linear combination looks like $\sum_{k=1}^{m} c_k a_k$. The span of $a_1, a_2, \ldots, a_m$ is the set of all linear combinations of $a_1, a_2, \ldots, a_m$ and the notation for span is $\langle a_1, a_2, \ldots, a_m \rangle$. In particular $\langle a \rangle$ denotes the set of all scalar multiples of a. It is important to note that $\langle a_1, a_2, \ldots, a_m \rangle$ is closed under addition and scalar multiplication; that is, it is a subspace of $\mathbb{R}^n$ [13].

If ℓ be a subspace of $\mathbb{R}^n$, and $a_1, a_2, \ldots, a_m \in \ell$, as a result $\langle a_1, a_2, \ldots, a_m \rangle \subseteq \ell$. We can say $v_1, v_2, \ldots, v_m$ span ℓ if $\langle a_1, a_2, \ldots, a_m \rangle$. This means that every member of ℓ can be expressed as a linear combination of $a_1, a_2, \ldots, a_m$. In this case we say that $a_1, a_2, \ldots, a_m$ form a spanning set for ℓ. Every subspace of $\mathbb{R}^n$ has a basis, and any two bases of ℓ have the same number of elements. This number is called the dimension of the subspace. Thus, for example, if $a_1, a_2, \ldots, a_m$ are independent, then $\langle a_1, a_2, \ldots, a_m \rangle$ has dimension m.

The Gram-Schmidt process is an algorithm that produces orthonormal bases. Let ℓ be a subspace of $\mathbb{R}^n$, and let $a_1, a_2, \ldots, a_m$ be a basis of ℓ. The Gram-Schmidt process uses $a_1, a_2, \ldots, a_m$ to produce $Q_1, Q_2, \ldots, Q_m$ that form a

basis of ℓ. Thus $\ell = \langle a_1, a_2, \ldots, a_m \rangle = \langle Q_1, Q_2, \ldots, Q_m \rangle$. And the column vectors $Q_1, Q_2, \ldots, Q_m$ also satisfy

$$\langle Q_1 \rangle = \langle a_1 \rangle \tag{A.5}$$

$$\langle Q_1, Q_2 \rangle = \langle a_1, a_2 \rangle \tag{A.6}$$

$$\vdots \tag{A.7}$$

$$\langle Q_1, Q_2, \ldots, Q_m \rangle = \langle a_1, a_2, \ldots, a_m \rangle \tag{A.8}$$

We are given linearly independent vectors $a_1, a_2, \ldots, a_m \in \mathbb{R}^n$, and we seek orthonormal $Q_1, Q_2, \ldots, Q_m$ satisfying the Equation A.8.

In order to satisfy $\langle Q_1 \rangle = \langle a_1 \rangle$, we must choose Q_1 to be a multiple of a_1. Since we also require Euclidean form of, $\|Q_1\| = 1$, we define

$$Q_1 = \left(\frac{1}{r_{11}} \right) a_1, \text{where } r_{11} = \|a_1\|_2 \tag{A.9}$$

We know that $r_{11} \neq 0$ which causes divide by 0 hazard, because $a_1, a_2, \ldots, a_m$ are linearly independent, so $a_1 \neq 0$. The equation $Q_1 = \left(\frac{1}{r_{11}} \right) a_1$ implies that $Q_1 \in \langle a_1 \rangle$; hence $\langle Q_1 \rangle \subseteq \langle a_1 \rangle$. Conversely the equation $Q_1 r_{11} = a_1$ implies that $a_1 \in \langle Q_1 \rangle$, and therefore $\langle a_1 \rangle \subseteq \langle Q_1 \rangle$. Thus $\langle Q_1 \rangle \subseteq \langle a_1 \rangle$.

The second step of the algorithm is to find Q_2 such that Q_2 is orthogonal to Q_1, $\|Q_2\| = 1$, and $\langle Q_1, Q_2 \rangle = \langle a_1, a_2 \rangle$. We can produce a vector $\tilde{Q_2}$ that lies in the plane and is orthogonal to Q_1 by subtracting just the right multiple of Q_1 from a_2. We can then obtain Q_2 by scaling $\tilde{Q_2}$. Thus let

$$\tilde{Q_2} = a_2 - r_{12} Q_1 \tag{A.10}$$

where the scalar r_{12} is to be determined. We must choose r_{12} so that $(\tilde{Q_2}, Q_1) = 0$. This equation implies $0 = (a_2 - r_{12} Q_1, Q_1) = (a_2, Q_1) - r_{12}(Q_1, Q_1)$, and since $(Q_1, Q_1) = 0$:

$$r_{12} = (a_2, Q_1) \tag{A.11}$$

On the other hand, this choice of r_{12} guarantees that $(\tilde{Q_2}, Q_1) = 0$.

We can find orthogonal Q matrix by satisfying Equation A.8. And suppose that we have found orthonormal vectors $Q_1, Q_2, \ldots, Q_{k-1}$ such that $\langle Q_1, Q_2, \ldots, Q_i \rangle = \langle a_1, a_2, \ldots, a_i \rangle$, for $i = 1, \ldots, k-1$ by repeating the same process. Now, we can determine Q_k, which is a general formula for the solution and very useful for us. We seek $\tilde{Q_k}$ of the form

$$\tilde{Q_k} = a_k - \sum_{j=1}^{k-1} r_{jk} Q_j \tag{A.12}$$

where $\tilde{Q_k}$ is orthogonal to $Q_1, Q_2, \ldots, Q_{k-1}$.
The equations $(\tilde{Q_k}, Q_i) = 0$, $i = 1, \ldots, k-1$, imply that

$$(a_k, Q_i) - \sum_{j=1}^{k-1} r_{jk}(Q_j, Q_i) = 0 \quad \text{i = 1, 2, ..., k-1} \tag{A.13}$$

Since $(Q_i, Q_j) = 0$ when $i \neq j$, and $(Q_i, Q_i) = 1$, these equations reduce to

$$r_{ik} = (a_k, Q_i) \quad \text{i = 1, 2, ..., k-1} \tag{A.14}$$

If r_{ik} are defined by the equation above, then $\tilde{Q_k}$ is orthogonal to $Q_1, Q_2, \ldots, Q_{k-1}$.
Let

$$r_{kk} = \|\tilde{Q_k}\|_2 \neq 0 \tag{A.15}$$

And define

$$Q_k = \frac{1}{r_{kk}} \tilde{Q_k} \tag{A.16}$$

Then clearly $\|\tilde{Q_k}\|_2 = 1$ and $(Q_i, Q_k) = 0$, $i = 1, 2, \ldots, k-1$. Combining these equations:

$$a_k = \tilde{Q_k} + \sum_{j=1}^{k-1} r_{jk} Q_j \tag{A.17}$$

And using this:

$$\tilde{Q_k} = Q_k r_{kk} \tag{A.18}$$

And combining these equations, $a_k = Q_k r_{kk} + \sum_{j=1}^{k-1} r_{jk} Q_j$ There are actually m such equations, one for each value k. Writing out these equations, we have

$$a_1 = Q_1 r_{11} \tag{A.19}$$

$$a_2 = Q_1 r_{12} + Q_2 r_{22} \tag{A.20}$$

$$a_3 = Q_1 r_{13} + Q_2 r_{23} + Q_3 r_{33} \tag{A.21}$$

$$a_4 = Q_1 r_{14} + Q_2 r_{24} + Q_3 r_{34} + Q_4 r_{44} \tag{A.22}$$

$$\vdots\!\vdots \tag{A.23}$$

$$a_m = Q_1 r_{1m} + Q_2 r_{2m} + Q_3 r_{3m} + \ldots + Q_m r_{mm} \tag{A.24}$$

These can be seen in a single matrix equation

$$\begin{bmatrix} a_1 & a_2 & a_3 & \ldots & a_m \end{bmatrix} =$$

$$\begin{bmatrix} Q_1 & Q_2 & Q_3 & \ldots & Q_m \end{bmatrix} \times \begin{bmatrix} r_{11} & r_{12} & r_{13} & \ldots & r_{1m} \\ 0 & r_{22} & r_{23} & \ldots & r_{2m} \\ 0 & 0 & r_{33} & \ldots & r_{3m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & r_{mm} \end{bmatrix}$$

Defining

$$A = \begin{bmatrix} a_1 & a_2 & a_3 & \ldots & a_m \end{bmatrix} \in \mathbb{R}^{n \times m}$$

$$Q = \begin{bmatrix} Q_1 & Q_2 & Q_3 & \ldots & Q_m \end{bmatrix} \in \mathbb{R}^{n \times m}$$

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} & \ldots & r_{1m} \\ 0 & r_{22} & r_{23} & \ldots & r_{2m} \\ 0 & 0 & r_{33} & \ldots & r_{3m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & r_{mm} \end{bmatrix} \in \mathbb{R}^{m \times m}$$

Equations A.12, A.14, A.15 and A.16 are used to implement the kth step of classical Gram-Schmidt algorithm. After performing this step for $k = 1, 2, \ldots, m$, we have the desired $Q_1, Q_2, \ldots, Q_m$ and $R_1, R_2, \ldots, R_m$.

Unfortunately the classical Gram-Schmidt process is numerically unstable since its algorithm updates only the nearest columns and this increases the round-off errors. These small round-off errors sometimes cause the computed vectors to

be far from orthogonal. However, a slight modification of the algorithm suffices to make it stable. In the classical Gram-Schmidt algorithm, Q_1 is taken to be multiple of v_1. Then the appropriate multiple of Q_1 is subtracted from v_2 to obtain a vector that is orthogonal to Q_1. The modified Gram-Schmidt procedure calculates Q_1 just as before. It then subtracts multiples of Q_1 not just from v_2, but from $v_3, v_4, \ldots, v_m$ as well, so that the resulting vectors $v_2^{(1)}, v_3^{(1)}, \ldots, v_m^{(1)}$ are all orthogonal to Q_1. This is the method that we used in our implementation and it is shown in Figure A.1.

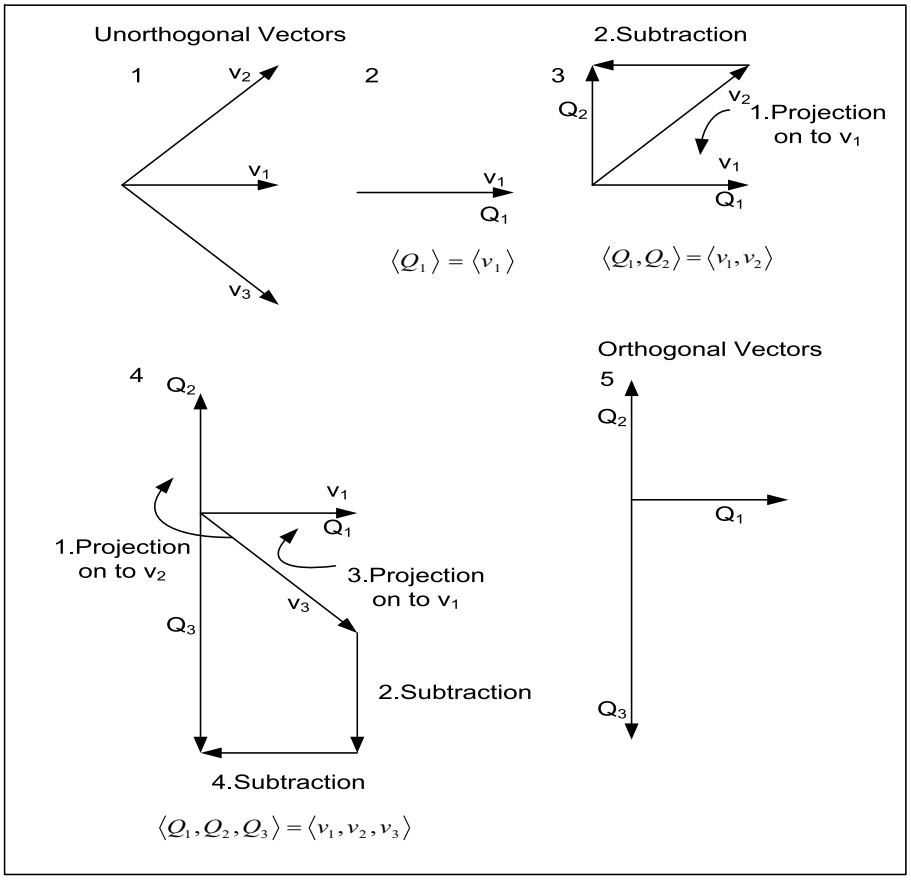


Figure A.1: **Visualizing orthonormalization method.**

The next two sections are devoted to the general solutions of the other two QR decomposition methods.

2. Householder Reflections

Householder reflections (transformations) is another method to decompose a matrix, A, into Q and R matrices. To describe this method, we chose to use a matrix of size 2. If we choose a line, l, in $\mathbb{R}^2$ which passing through the origin, a linear transformation can be described as reflecting any vector in $\mathbb{R}^2$ through the line l using an operator. We can describe this operator as a matrix with size 2. Suppose a vector, v, which is on the l, and another vector which is orthogonal to the v, named u. We can say that u, v is a basis for $\mathbb{R}^2$. Then choose any vector in $\mathbb{R}^2$ to reflect on the line l. Assume that there is a vector x that can be expressed as the linear combination of u and v vectors as $x = \alpha u + \beta v$. The reflection of x to the line l is $-\alpha u + \beta v$ as show in Figure A.2.

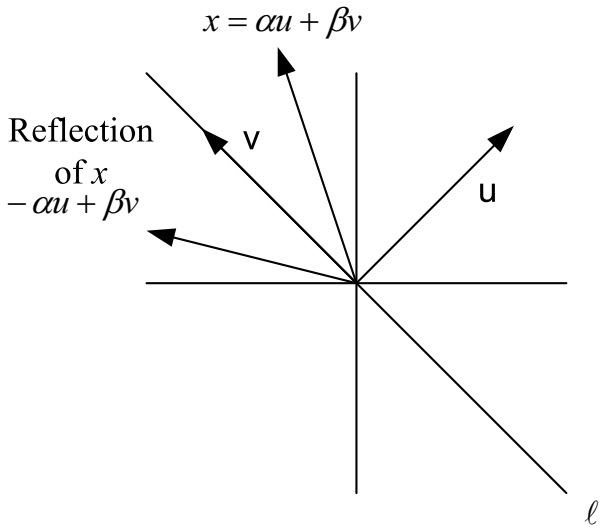


Figure A.2: **Visualizing reflection method.**

We define the reflection operator, Q, and Q must satisfy the equation:

$$Q(\alpha u + \beta v) = -\alpha u + \beta v \tag{A.25}$$

for all α and β. It is important to note that the value α and β can change, and the graph can be drawn differently, however this wouldn't change our general results. Using this result, we find that

$$Q\alpha u + Q\beta v = -\alpha u + \beta v \tag{A.26}$$

$$Qu = -u; Qv = v \tag{A.27}$$

We need to find a Q matrix satisfying these conditions. Let $Q = I - 2P$, where u is a unit vector and $P = uu^T$. Discussing on obtaining Q matrix as the reflector is beyond the scope of this part. However it can be proven that this equation satisfies the conditions:

$$Qu = u - 2Pu = -u \tag{A.28}$$

$$Qv = v - 2Pv = v \tag{A.29}$$

As a result, we found a Q matrix which reflects vectors through the line l and the equation for Q can be written as $Q = I - 2uu^T$. However, if we choose not to take u as a unit matrix, we can define Q matrix as

$$Q = I - \gamma uu^T \quad \text{where} \quad \gamma = \frac{2}{\|u\|_2^2} \tag{A.30}$$

The only unexplained variable is the unit vector u. u is described as $a - y$, where a represents one of the columns in our matrix and y is a column vector which can be described as $a = [\sigma, 0, 0, 0, 0]^T$ where $\sigma = \|x\|_2$.

After finding the first Q matrix which can be denoted as Q_1, we calculate $Q_1 A$ which is equal to $Q_1^T A$ since Q_1 is symmetric. The expected result is to be R matrix which is upper triangular. If not, same process continues to find Q_2 till we find a resulting R matrix.

3. Givens Rotations

Definition 3 - If there are two nonzero vectors, x and y, in a plane, the angle, θ, between them can be formulized as:

$$\cos\theta = \frac{(x, y)}{\|x\|_2 \|y\|_2} \tag{A.31}$$

This formula can be extended to n vectors.

Definition 4 - The angle, θ, can be defined as:

$$\theta = \arccos \frac{(x, y)}{\|x\|_2 \|y\|_2} \tag{A.32}$$

These two vectors are orthogonal if $\theta = \frac{\pi}{2}$ radians where x or y equals to 0.

Using Givens Rotation method, we find an operator which rotates each vector through a fixed angle, θ, and this operator can be represented as a matrix. If we use a 2×2 matrix, this operator can be described as:

$$Q = \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix}$$

This Q matrix can be determined by using two column vectors: $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$. The result of the multiplications between these column vectors and the Q matrix are the columns of the Q matrix.

Thus, we can write the operator Q as:

$$Q = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \text{ for } 2 \times 2 \text{ matrices}$$

We solve the $A = Q \times R$, so this can be written as $Q^T \times A = R$. And we know that R is an upper triangular matrix. Let there be a matrix, $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$, this can be seen as:

$$\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix}$$

It can be easily seen that:

$$(-\sin\theta A_{11} + \cos\theta A_{21}) = 0 \tag{A.33}$$

$$\cos\theta A_{21} = \sin\theta A_{11} \tag{A.34}$$

$$\frac{\sin\theta}{\cos\theta} = \frac{A_{21}}{A_{11}} = \tan\theta \tag{A.35}$$

$$\theta = \arctan\left(\frac{A_{21}}{A_{11}}\right) \tag{A.36}$$

After determining θ, we can determine Q matrix. To determine R matrix, we need to work column by column:

$$Q^T \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} = \begin{bmatrix} R_{11} \\ 0 \end{bmatrix} \text{ to solve for the first column of } R \text{ matrix}$$

$$Q^T \begin{bmatrix} A_{12} \\ A_{22} \end{bmatrix} = \begin{bmatrix} R_{12} \\ R_{22} \end{bmatrix} \text{ to solve for the second column of } R \text{ matrix}$$

For $n \times n$ matrices, the 2×2 representation of Q matrix can be generalized. The decomposition process stays the same.

A.1.2 LU Decomposition

If A is an $n - by - n$ matrix and its leading principal submatrices are all nonsingular, we can decompose A into unique unit lower triangular and upper triangular matrices [13] as shown below:

$$A = L \times U \tag{A.37}$$

where L is unit lower triangle and U is upper triangular matrix. Consider the equation $A = L \times U$ in detail for 4×4 matrix:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{bmatrix} \times \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix}$$

The goal is the find the value for the entries which are not 0. The first row of L matrix is completely known and there is only one non-zero element. If we multiply the first row of L matrix by the jth column of the U matrix, we find

$$A_{11} = u_{11}, \quad A_{12} = u_{12}, \quad A_{13} = u_{13}, \quad \text{and} \quad A_{14} = u_{14} \tag{A.38}$$

which can be described as $A_{1j} = u_{1j}$ in general which means that the first row of U matrix can be uniquely determined. Thus, the assumption that A is a nonsingular matrix gives us a U matrix which is nonsingular too. We will continue to determine U matrix entries as shown previously.

One can see that if we know the first row of U matrix that means that we already know the first column of the U matrix too. And the first column of the U matrix has only one non-zero element. If we multiply the first column of the U matrix by the rows of the L matrix, we find

$$A_{21} = l_{21} \times u_{11}, \quad A_{31} = l_{31} \times u_{11}, \quad \text{and} \quad A_{41} = l_{41} \times u_{11} \tag{A.39}$$

can be written as

$$A_{i1} = l_{i1} \times u_{11} \tag{A.40}$$

At this point, we know the values for A_{i1}, u_{11} and using the assumption that U is a nonsingular matrix ($u_{ij} \neq 0$), we don't have a problem of dividing by zero. This leads to

$$l_{i1} = \frac{A_{i1}}{u_{11}}, i = 2, 3, 4 \tag{A.41}$$

We uniquely determined the first row of U and the first column of L using these steps. This is the way we determined L matrix entries. If we continue to solve for U matrix

$$A_{22} = l_{21} \times u_{12} + 1 \times u_{22} \tag{A.42}$$

$$A_{23} = l_{21} \times u_{13} + 1 \times u_{23} \tag{A.43}$$

$$A_{24} = l_{21} \times u_{14} + 1 \times u_{24} \tag{A.44}$$

where we know the values except for u_{22}, u_{23} and u_{24}

$$u_{22} = A_{22} - l_{21} \times u_{12} \tag{A.45}$$

$$u_{23} = A_{23} - l_{21} \times u_{13} \tag{A.46}$$

$$u_{24} = A_{24} - l_{21} \times u_{14} \tag{A.47}$$

For L matrix

$$A_{32} = u_{12} \times l_{31} + u_{22} \times l_{32} \tag{A.48}$$

$$A_{42} = u_{12} \times l_{41} + u_{22} \times l_{42} \tag{A.49}$$

where we know the values except for l_{32} and l_{42}

$$l_{32} = \frac{A_{32} - l_{31} \times u_{12}}{u_{22}} \tag{A.50}$$

$$l_{42} = \frac{A_{42} - l_{41} \times u_{12}}{u_{22}} \tag{A.51}$$

For U matrix:

$$A_{33} = l_{31} \times u_{13} + l_{32} \times u_{23} + u_{33} \tag{A.52}$$

$$A_{34} = l_{31} \times u_{14} + l_{32} \times u_{24} + u_{34} \tag{A.53}$$

where we know the values except for u_{33} and u_{34}

$$u_{33} = A_{33} - l_{31} \times u_{13} - l_{32} \times u_{23} \tag{A.54}$$

$$u_{34} = A_{34} - l_{31} \times u_{14} - l_{32} \times u_{24} \tag{A.55}$$

For L matrix

$$A_{43} = l_{41} \times u_{13} + l_{42} \times u_{23} + l_{43} \times u_{33} \tag{A.56}$$

where we know the values except for l_{43}

$$l_{43} = \frac{A_{43} - l_{41} \times u_{13} - l_{42} \times u_{23}}{u_{33}} \tag{A.57}$$

Lastly,

$$A_{44} = l_{41} \times u_{14} + l_{42} \times u_{24} + l_{43} \times u_{34} + u_{44} \tag{A.58}$$

$$u_{44} = A_{44} - l_{41} \times u_{14} - l_{42} \times u_{24} - l_{43} \times u_{34} \tag{A.59}$$

As a result, if we look at the equations for L and U matrix entries, we can come up with two general equations which define the non-zero entries of the two matrices:

$$l_{ij} = \frac{A_{ij} - \sum_{t=1}^{j-1} l_{it} \times u_{tj}}{u_{jj}} \tag{A.60}$$

$$u_{ij} = a_{ij} - \sum_{t=1}^{i-1} l_{it} \times u_{tj} \tag{A.61}$$

These general equations uniquely determine L and U matrices if they are applied in the correct order.

A.1.3 Cholesky Decomposition

Cholesky decomposition is another elementary operation, which decomposes a symmetric positive definite matrix into a unique lower triangular matrix with positive diagonal entries [13]. Cholesky decomposition of a matrix A is shown as $A = G \times G^T$, where G is a unique lower triangular matrix, Cholesky triangle, and G^T is the transpose of this lower triangular matrix (as shown below for 4×4 matrices).

$$G = \begin{bmatrix} G_{11} & 0 & 0 & 0 \\ G_{21} & G_{22} & 0 & 0 \\ G_{31} & G_{32} & G_{33} & 0 \\ G_{41} & G_{42} & G_{43} & G_{44} \end{bmatrix}; G^T = \begin{bmatrix} G_{11} & G_{21} & G_{31} & G_{41} \\ 0 & G_{22} & G_{32} & G_{42} \\ 0 & 0 & G_{33} & G_{43} \\ 0 & 0 & 0 & G_{44} \end{bmatrix}$$

The given matrix should be symmetric. If $A \in \mathbb{R}^{n \times n}$ is symmetric then it has a LU decomposition which results in a unit lower tirangular matrix, L, and a diagonal matrix $D = diag(D_{11}, ..., D_{nn})$ such that $A = LDL^T$ which can be seen as:

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} =$$

$$\begin{bmatrix} L_{11} & 0 & 0 & 0 \\ L_{21} & L_{22} & 0 & 0 \\ L_{31} & L_{32} & L_{33} & 0 \\ L_{41} & L_{42} & L_{43} & L_{44} \end{bmatrix} \times \begin{bmatrix} D_{11} & 0 & 0 & 0 \\ 0 & D_{22} & 0 & 0 \\ 0 & 0 & D_{33} & 0 \\ 0 & 0 & 0 & D_{44} \end{bmatrix} \times \begin{bmatrix} L_{11} & L_{21} & L_{31} & L_{41} \\ 0 & L_{22} & L_{32} & L_{42} \\ 0 & 0 & L_{33} & L_{43} \\ 0 & 0 & 0 & L_{44} \end{bmatrix}$$

The given matrix also should be positive definite. A matrix $A \in \mathbb{R}^{n\times n}$ is *positive definite* if $x^TAx > 0$ for $x \in \mathbb{R}^n$ and x $\neq 0$ and if it is *symmetric positive definite matrix* then $A^T = A$. A positive definite matrix is always nonsingular and its determinant is always positive. Since a unit lower triangular matrix, L, and a diagonal matrix, $D = diag(D_{11}, d_{22}, ..., D_{nn})$, exist such that $A = LDL^T$; and Ds are all positive, there exists a real lower triangular matrix with positive diagonal entries such that $G = Ldiag\sqrt{D_{11}}, ..., \sqrt{D_{nn}}$ and satisfies $A = G \times G^T$. Consider $A = LDL^T$ which can be seen as follows for a 2×2 matrix:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \times \begin{bmatrix} D_{11} & 0 \\ 0 & D_{22} \end{bmatrix} \times \begin{bmatrix} L_{11} & L_{21} \\ 0 & L_{22} \end{bmatrix} =$$
$$\begin{bmatrix} L_{11}D_{11} & 0 \\ L_{21}D_{11} & L_{22}D_{22} \end{bmatrix} \times \begin{bmatrix} L_{11} & L_{21} \\ 0 & L_{22} \end{bmatrix} = \begin{bmatrix} L_{11}^2D_{11} & L_{11}L_{21}D_{11} \\ L_{21}L_{11}D_{11} & L_{21}^2D_{11} + L_{22}^2D_{22} \end{bmatrix}$$

We can see $A = G \times G^T$ as follows for a 2×2 matrix:

$$A = \begin{bmatrix} L_{11}\sqrt{D_{11}} & 0 \\ L_{21}\sqrt{D_{11}} & L_{22}\sqrt{D_{22}} \end{bmatrix} \times \begin{bmatrix} L_{11}\sqrt{D_{11}} & L_{21}\sqrt{D_{11}} \\ 0 & L_{22}\sqrt{D_{22}} \end{bmatrix} =$$
$$\begin{bmatrix} L_{11}^2D_{11} & L_{11}L_{21}D_{11} \\ L_{21}L_{11}D_{11} & L_{21}^2D_{11} + L_{22}^2D_{22} \end{bmatrix}$$

where

$$G = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \times \begin{bmatrix} \sqrt{D_{11}} & 0 \\ 0 & \sqrt{D_{22}} \end{bmatrix}$$

As can be seen both approaches result in the same matrix A. As an example, assume a matrix A is given as follows:

$$A = \begin{bmatrix} 2 & -2 \\ -2 & 5 \end{bmatrix}$$

The Cholesky triangle is equal to the following matrix as shown previously:

$$G = \begin{bmatrix} L_{11}\sqrt{D_{11}} & 0 \\ L_{21}\sqrt{D_{11}} & L_{22}\sqrt{D_{22}} \end{bmatrix}$$

where the values for D_{11}, L_{21} and D_{22} are not known (the lower triangular matrix diagonal elements are all 1s, L_{11} and L_{22} respectively, since it is a unit triangle) so we can rewrite it as:

$$G = \begin{bmatrix} \sqrt{D_{11}} & 0 \\ L_{21}\sqrt{D_{11}} & \sqrt{D_{22}} \end{bmatrix}$$

If we go back to equation $A = LDL^T$ which can be rewritten as the following with the values on the diagonal of lower triangular matrix are all 1's:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ L_{21} & 1 \end{bmatrix} \times \begin{bmatrix} D_{11} & 0 \\ 0 & D_{22} \end{bmatrix} \times \begin{bmatrix} 1 & L_{21} \\ 0 & 1 \end{bmatrix}$$

where D_{11}, L_{21} and D_{22} can be computed. If we multiply these matrices, we find that:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} D_{11} & D_{11}L_{21} \\ D_{11}L_{21} & L_{21}^2 D_{11} + D_{22} \end{bmatrix}$$

which shows that

$$D_{11} = A_{11} \tag{A.62}$$

and

$$L_{21} = \frac{A_{12}}{D_{11}} \tag{A.63}$$

and

$$D_{22} = A_{22} - L_{21}^2 D_{11} \tag{A.64}$$

And if we place these equations, D_{11}, L_{21} and D_{22}, into the Cholesky triangle:

$$G = \begin{bmatrix} \sqrt{A_{11}} & 0 \\ \frac{A_{12}}{D_{11}}\sqrt{A_{11}} & \sqrt{A_{22} - L_{21}^2 D_{11}} \end{bmatrix}$$

For the given example, the resulting the Cholesky triangle becomes:

$$G = \begin{bmatrix} \sqrt{2} & 0 \\ \frac{-2}{2}\sqrt{2} & \sqrt{5 - (-1)^2 2} \end{bmatrix} = \begin{bmatrix} \sqrt{2} & 0 \\ -\sqrt{2} & \sqrt{3} \end{bmatrix}$$

The following equations uniquely determine matrix G for a 2×2 matrix if they are applied in the correct order:

$$G_{11} = \sqrt{A_{11}} \tag{A.65}$$

$$G_{21} = \frac{A_{12}}{D_{11}}\sqrt{A_{11}} \tag{A.66}$$

$$G_{22} = \sqrt{A_{22} - L_{21}^2 D_{11}} \tag{A.67}$$

Bibliography

[1] Y. Meng, A.P. Brown, R. A. Iltis, T. Sherwood, H. Lee, R. Kastner, "MP core: algorithm and design techniques for efficient channel estimation in wireless applications," Proceedings of the Design Automation Conference, pp. 297-302, 2005.

[2] R. A. Iltis, S. Mirzaei, R. Kastner, R. E. Cagley,B. T. Weals, "Carrier Offset and Channel Estimation for Cooperative MIMO Sensor Networks," Proceedings of the Global Telecommunications Conference, pp. 1-5, 2006.

[3] R. E. Cagley, B. T. Weals, S. A. McNally, R. A. Iltis, S. Mirzaei, R. Kastner, "Implementation of the Alamouti OSTBC to a Distributed Set of Single-Antenna Wireless Nodes," Proceedings of the Wireless Communications and Networking Conference, pp. 577-581, 2007.

[4] L. Zhou, L. Qiu, J. Zhu, "A novel adaptive equalization algorithm for MIMO communication system", Proceedings of the Vehicular Technology Conference, pp. 2408-2412, 2005.

[5] T. Abe, S. Tomisato,T. Matsumoto, "A MIMO turbo equalizer for frequency-selective channels with unknown interference", IEEE Transactions on Vehicular Technology, Volume 52, Issue 3, pp. 476-482, 2003.

[6] T. Abe and T. Matsumoto, "Space-time turbo equalization in frequency selective MIMO channels," IEEE Transactions on Vehicular Technology, pp. 469-475, 2003.

[7] K. Kusume, M. Joham, W. Utschick, G. Bauch, "Efficient Tomlinson-Harashima precoding for spatial multiplexing on flat MIMO channel," Proceedings of the International Conference on Communications, pp. 2021-2025, 2005.

[8] C. Hangjun, D. Xinmin, A. Haimovich, "Layered turbo space-time coded MIMO-OFDM systems for time varying channels," Proceedings of the Global

Telecommunications Conference, pp. 1831-1836, 2003.

[9] S. Haykin, "Adaptive Filter Theory," Prentice Hall, Fourth Edition.

[10] J. Ma, K.K. Parhi, E.F. Deprettere, "Annihilation-reordering lookahead pipelined CORDIC-based RLS adaptive filters and their application to adaptive beamforming," IEEE Transactions on Signal Processing, 2000.

[11] "IEEE 802.11 LAN/MAN Wireless LANS," IEEE Standards Association, http://standards.ieee.org/getieee802/802.11.html.

[12] "IEEE 802.16 LAN/MAN Broadband Wireless LANS," IEEE Standards Association, http://standards.ieee.org/getieee802/802.16.html.

[13] G.H. Golub, C.F.V. Loan, Matrix Computations, 3rd ed. Baltimore, MD: John Hopkins University Press.

[14] A. Björck, C. Paige, "Loss and recapture of orthogonality in the modified Gram-Schmidt algorithm," SIAM J. Matrix Anal. Appl., vol. 13 (1), pp 176-190, 1992.

[15] A. Björck, "Numerics of Gram-Schmidt orthogonalization," Linear Algebra and Its Applications, vol. 198, pp. 297-316, 1994.

[16] C. K. Singh, S.H. Prasad, P.T. Balsara, "VLSI Architecture for Matrix Inversion using Modified Gram-Schmidt based QR Decomposition", Proceedings of the International Conference on VLSI Design, pp. 836-841, 2007.

[17] J. Eilert, D. Wu, D. Liu, "Efficient Complex Matrix Inversion for MIMO Software Defined Radio", Proceedings of the International Symposium on Circuits and Systems, pp. 2610-2613, 2007.

[18] F. Edman, V. Öwall, "A Scalable Pipelined Complex Valued Matrix Inversion Architecture", Proceedings of the International Symposium on Circuits and Systems, pp. 4489-4492, 2005.

[19] M. Karkooti, J.R. Cavallaro, C. Dick, "FPGA Implementation of Matrix Inversion Using QRD-RLS Algorithm", Proceedings of the Asilomar Conference on Signals, Systems and Computers, pp. 1625-1629, 2005.

[20] C. Dick, F. Harris, M. Pajic, D. Vuletic, "Real-Time QRD-Based Beamforming on an FPGA Platform", Fortieth Asilomar Conference on Signals, Systems and Computers, 2006. ACSSC '06.

[21] H. M. Markowitz, "Portfolio Selection: Efficient Diversification of Investments", 2nd Edition, 1991.

[22] S. Maya-Rueda and M. Arias-Estrada, "FPGA Processor for Real-Time Optical Flow Computation," Lecture Notes in Computer Science. v2778. 1016-1103.

[23] M. F. Jacome, G. de Veciana, V. Lapinskii, "Exploring Performance Tradeoffs for Clustered VLIW ASIPS," Proceedings of the International Conference on Computer-Aided Design, 2000.

[24] S. Rixner, W. Dally, B. Khailany, P.Mattson, U. Kapasi, and J. Owens, "Register Organization for Media Processing," Proceedings of the International Symposium on High-Performance Computer Architecture, May 1999.

[25] R. El-Atfy, M.A. Dessouky, H. El-Ghitani, "Accelerating Matrix Multiplication on FPGAs," Proceedings of the International Design and Test Workshop, 2007, Pages 203-204, 2007.

[26] O. Mencer, M. Morf, M. Flynn, "PAM-Blox: High Performance FPGA Design for Adaptive Computing," Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, Pages 167-174, 1998.

[27] A. Amira, A. Bouridane, P. Milligan, "Accelerating Matrix Product on Reconfigurable Hardware for Signal Processing," Proceedings of the International Conference on Field-Programmable Logic and Applications, Pages 101-111, 2001.

[28] V.K. Prasanna and Y. Tsai, "On Synthesizing Optimal Family of Linear Systolic Arrays for Matrix Multiplication," IEEE Transactions on Computers, Vol. 40, no. 6, 1991.

[29] J. Jang, S. Choi, V.K.K. Prasanna, "Area and Time Efficient Implementations of Matrix Multiplication on FPGAs," Proceedings of the IEEE International Conference on Field-Programmable Technology, Pages 93-100, 2002.

[30] A. Meucci, "Risk and Asset Allocation," Springer Finance Press, 2005.

[31] F. Fabozzi, "Handbook of Portfolio Management," Frank J. Fabozzi Associates, New Hope, Pennsylvania, 1998.

[32] M. Lobo, L. Vandenberghe, S. Boyd and H. Lebret, " Applications of second order cone programming," Linear Algebra and its Applications, Special Issue on Linear Algebra in Control, Signals and Image Processing 284, pp. 193-228,

1998.

[33] A. Ben-Tal and A. Nemirovski, "Optimal Design of Engineering Structures," Optima pp. 4-9.

[34] S. Boyd and L. Vandenberghe, "Convex Optimization", Cambridge University Press, 2004.

[35] Y. Nesterov, and A. Nemitovski, "Interor-Point Polynomial Algorithms in Convex Programming", Society for Industrial and Applied Mathemetics, 1995.

[36] http://www.barra.com

[37] http://www.sungard.com/AllocationMaster/

[38] http://www.ibbotson.com

[39] http://www.northinfo.com

[40] http://invest-tech.com/allocator.html

[41] http://www.wilshire.com

[42] http://wilsonintl.com

[43] G.L. Zhang, P.H.W. Leong, C.H. Ho, K.H. Tsoi, C.C.C. Cheung, D.-U. Lee, R.C.C. Cheung, W. Luk, "Reconfigurable acceleration for Monte Carlo based financial simulation," Proceedings of the International Conference on Field-Programmable Technology, pp. 215-222, 2005.

[44] G.W. Morris, M. Aubury, "Design Space Exploration of the European Option Benchmark using Hyperstreams," Proceedings of the International Conference on Field Programmable Logic and Applications, pp.5-10, 2007.

[45] D.B. Thomas, J.A. Bower, W. Luk, "Automatic Generation and Optimisation of Reconfigurable Financial Monte-Carlo Simulations," Proceedings of the International Conference on Application-specific Systems, Architectures and Processors, pp.168-173, 2007.

[46] D.B. Thomas, W. Luk, "Sampling from the Multivariate Gaussian Distribution using Reconfigurable Hardware," Proceedings of the Symposium on Field-Programmable Custom Computing Machines, pp.3-12, 2007.

[47] D.B. Thomas, W. Luk, "Credit Risk Modelling using Hardware Accelerated Monte-Carlo Simulation," Proceedings of Field-Programmable Custom Computing Machines, 2008.

[48] N. A. Woods, and T. VanCourt, "FPGA Acceleration of Quasi-Monte Carlo in Finance," accepted for publication in Proceedings of Field Programmable Logic and Applications, 2008.

[49] A. Kaganov, A. Lakhany and P. Chow, "FPGA Acceleration of Monte-Carlo Based Credit Derivative Pricing," accepted for publication in Proceedings of Field Programmable Logic and Applications, 2008.

[50] M. LiCalzi and A. Sorato, "The Pearson system of utility functions," Game Theory and Information 0311002, EconWPA 2003.

[51] W. Nicholson, "Microeconomic Theory: Basic Principles and Extensions," South Western Educational Publishing, 2004.

[52] A. Irturk, B. Benson, S. Mirzaei, and R. Kastner, "GUSTO: An Automatic Generation and Optimization Tool for Matrix Inversion Architectures," Transactions on Embedded Computing Systems.

[53] A. Irturk, B. Benson, and R. Kastner, "Automatic Generation of Decomposition based Matrix Inversion Architectures," Proceedings of the International Conference on Field-Programmable Technology, 2008.

[54] A. Irturk, B. Benson, S. Mirzaei and R. Kastner, "An FPGA Design Space Exploration Tool for Matrix Inversion Architectures," Proceedings of the Symposium on Application Specific Processors, 2008.

[55] A. Irturk, B. Benson, N. Laptev and R. Kastner, "Architectural Optimization of Decomposition Algorithms for Wireless Communication Systems," Proceedings of the International Conference on Wireless Communications and Networking, 2009.

[56] Tensilica: Xtensa LX $http://www.tensilica.com/products/xtensa\_LX.htm$, 2005.

[57] Automated Configurable Processor Design Flow, White Paper, Tensilica, Inc. $http://www.tensilica.com/pdf/Tools\_white\_paper\_final-1.pdf$, January 2005.

[58] Diamond Standard Processor Core Family Architecture, White Paper, Ten-

silica, Inc. *http* : *//www.tensilica.com/pdf/Diamond_WP.pdf*, October 2006.

[59] D. Goodwin and D. Petkov, "Automatic Generation of Application Specific Processors," Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems, 2003.

[60] Stretch. Inc.: S5000 Software-Configurable Processors, *http* : *//www.stretchinc.com/products/devices.php.*

[61] Forte Design System Cynthesizer, *http* : *//www.forteds.com/products/cynthesizer.asp*, 2008

[62] NEC CyberWorkBench, *http* : *//www.necst.co.jp/product/cwb/english/index.html*, 2008.

[63] Mentor Graphics Technical Publications: Designing High Performance DSP Hardware using Catapult C Synthesis and the Altera Accelerated Libraries, *http* : *//www.mentor.com/techpapers/fulfillment/upload/mentorpaper_*36558*.pdf*, 2008.

[64] Mentor Graphics Technical Publications: Alcatel Conquers the Next Frontier of Design Space Exploration using Catapult C Synthesis, *http* : *//www.mentor.com/techpapers/fulfillment/upload/mentorpaper_*22739*.pdf*, 2008.

[65] Mentor Graphics Technical Publications, *http* : *//www.mentor.com/training_and_services/tech_pubs.cfm*, 2008.

[66] K. Wakabayashi, "C-based synthesis experiences with a behavior synthesizer, Cyber," Proceedings of the conference on Design, Automation and Test in Europe, page 83, 1999.

[67] B. Landwehr, P. Marwedel, and R. Dömer, "OSCAR: Optimum Simultaneous Scheduling, Allocation and Resource Binding Based on Integer Programming," Proceedings of the European Design Automation Conference, pages 9095, Grenoble, France, 1994.

[68] P. Paulin and J. Knight, "Force-directed scheduling for the behavioral synthesis of ASICs," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, June 1989.

[69] S. Devadas and R. Newton, "Algorithms for hardware allocation in data path synthesis," IEEE Transactions on Computer-Aided Design, July 1989.

[70] P. Gutberlet, J. Müller, H. Krämer, and W. Rosenstiel, "Automatic module allocation in high level synthesis," Proceedings of the Conference on European Design Automation (EURO-DAC 92), pages 328333, 1992.

[71] C. Tseng and D. Seiwiorek, "Automated synthesis of data paths in digital systems," IEEE Transactions on Computer-Aided Design, 1986.

[72] F.-S. Tsai and Y.-C. Hsu, "STAR: An automatic data path allocator," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2(9):10531064, September 1992.

[73] F. Brewer and D. Gajski. Chippe, "A system for constraint driven behavioral synthesis," IEEE Transactions on Computer-Aided Design, July 1990.

[74] P. Marwedel, "The MIMOLA system: Detailed description of the system software," Proceedings of Design Automation Conference, June 1993.

[75] S. Che, J. Li, J.W. Sheaffer, K. Skadron, J. Lach, "Accelerating Compute-Intensive Applications with GPUs and FPGAs," In Proceedings of Symposium on Application Specific Processors, Pages 101-107, 2008.

[76] S.D. Haynes, J. Stone, P.Y.K. Cheung, W. Luk, "Video Image Processing with the SONIC Architecture" Computer Magazine, Pages 50-57, 2000.

[77] N. P. Sedcole, P.Y.K. Cheung, G.A. Constantinides, W. Luk, "A Reconfigurable Platform for Real-Time Embedded Video Image Processing," In Proceedings of Field-Programmable Logic and Applications, LNCS 2778, Pages 606-615, 2003.

[78] F. Bensaali and A. Amira, "Design & Efficient FPGA Implementation of an RGB to YCbCr Colour Space Converter Using Distributed Arithmetic," In Proceedings of Field-Programmable Logic and Applications, LNCS 3203, Pages 991-995, 2004.

[79] H. Styles and W. Luk, "Customising Graphics Applications: Techniques and Programming Interface," In Proceedings of Symposium on Field-Programmable Custom Computing Machines, Pages 77-87, 2000.

[80] R. Fernando and M. Kilgard, "Cg: The Cg Tutorial", Addison Wesley, 2003.

[81] P. Colantoni, N. Boukala, J. D. Rugna, "Fast and Accurate Color Image Processing Using 3D Graphics Cards", Proc. of Vison, Modeling and Visualization, Pages 1-9, 2003.

[82] R. Strzodka and C. Garbe, "Real-Time Motion Estimation and Visualization on Graphics Cards", Proceedings of Visualisation, Pages 545-552, 2004.

[83] C. Bruyns and B. Feldman, "Image Processing on the GPU: a Canonical Example", Project at Berkeley, 2003.

[84] Xilinx Product Specification "Virtex-4 Family Overview" (2007). *http://www.xilinx.com/support/documentation/data_sheets/ds*112.*pdf*

[85] Ali Irturk, "Implementation of QR Decomposition Algorithms using FPGAs," M.S. Thesis, Department of Electrical and Computer Engineering, University of California, Santa Barbara, June 2007.

[86] FPGA Design Flow Overview, *http://www.xilinx.com/itp/xilinx*8*/help/iseguide/html/ise_fpga_design_flow_overview.htm*

[87] D. Thomas, L. Howes, W. Luk, "A Comparison of CPUs, GPUs, FPGAs, and Massively Parallel Processor Arrays for Random Number Generation," Proceedings of the International Symposium on Field Programmable Gate Arrays, Pages 63-72, 2009.

[88] Ambric, Inc. Am2000 Family Architecture Reference, May 2008.

[89] Eclipse Foundation, *http://www.eclipse.org*

[90] M. Butts, A.M. Jones, P. Wasson, "A Structural Object Programming Model, Architecture, Chip and Tools for Reconfigurable Computing," Proceedings of the International Symposium on Field-Configurable Custom Computing Machines, Pages 55-64, 2007.

[91] M. Butts, "Synchronization through Communication in a Massively Parallel Processor Array," Proceedings of the International Symposium on Microarchitecture, Pages 32-40, 2007.

[92] A.M.Jones, M. Butts, "TeraOPS Hardware: A New Massively-Parallel MIMD Computing Fabric IC," Proceedings of the Symposium on High Performance Chips, 2006.

[93] Ambric, Inc. Ambric Technology Backgrounder, *http://www.ambric.com/technology/technology − overview.php*

[94] Ambric, Inc. Am2000 Family Instruction Set Reference, January 2008.

[95] J.D.Owens, D. Luebke, N. Govindaraju, M. Harris, J.Krüger, A.E.Lefohn, T.J.Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," Proceedings of Eurographics 2005, State of the Art Reports, Pages 21-51, 2005.

[96] J.D.Owens, D. Luebke, N. Govindaraju, M. Harris, J.Krüger, A.E.Lefohn, T.J.Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," Proceedings of the Computer Graphics Forum, Pages 80-113, 2007.

[97] Khronos Group. Open GL - The Industry's Foundation for High Performance Graphics. *http* : *//www.opengl.org*, 2007.

[98] Microsoft Corporation, Microsoft DirectX.
http : *//www.microsoft.com/directx*, 2007.

[99] PeakStream, The PeakStream Platform: High Productivity Software Development for Multi-core Processors.
http : *//peakstreaminc.com/reference/peakstream_platform_technote.pdf*, 2006.

[100] M.D.McCool and B.D'Amora, "Programming using RapidMind on the Cell BE," Proceedings of the ACM/IEEE Conference on Supercomputing, Page 222, 2006.

[101] NVIDIA Corporation, CUDA Programming Guide Version 0.8.1.
http : *//developer.download.nvidia.com/compute/cuda/*0_81/*NVIDIA_CUDA_Programming_Guide*_0.8.1.*pdf*, 2007.

[102] A. R. Brodtkorb, "A MATLAB Interface to the GPU", Master's thesis, Department of Informatics, Faculty of Mathematics and Natural Sciences, University of Oslo, 2007.

[103] Advanced Micro Devices Inc. Radeon X1900 Graphics Technology - GPU Specification.
http : *//ati.de/companyinfo/researcher/documents/ATI_CTM_Guide.pdf*, 2006.

[104] NVIDIA Corporation, Geforce 8800.
http : *//www.nvidia.com/page/geforce_*8800.*html*, 2007.

[105] NVIDIA Corporation, Tesla S1070 Specifications.
http : *//www.nvidia.com/object/product_tesla_s*1070_*us.html*, 2009.

[106] D. Göddeke, R. Strzodka, S. Turek, "Accelerating Double Precision FEM Simulation with GPUs," Peoceedings of the Symposium on Simulation Technique, Pages 139-144, 2005.

[107] D. Göddeke, R. Strzodka, S. Turek, "Performance and Accuracy of Hardware-Oriented, Native-Emulated and Mixed-Precision Solvers in FEM Simulations," Proceedings of International Journal of Parallel, Emergent and Distributed Systems, 2007.

[108] C. Jiang and M. Snir, "Automatic Tuning Matrix Maltiplication Performance on Graphics Hardware," Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, Pages 185 - 196, 2005.

[109] B. Kovar, J. Kloub, J. Schier, A. Hermanek, P. Zemcik, A. Herout, "Rapid Prototyping Platform for Reconfigurable Image Processing," Proceedings of Mezinarodni Conference Technical Computing Program, 2008.

[110] Z. Pavel, F. Otto, R. Miroslav, V. Pavel, "Imaging Algorithm Speedup Using Co-Design," In Summaries Volume Process Control, Strbske Pleso, Pages 96-97, 2001.

[111] T. Halfhill, "Parallel Processing with CUDA," The Insider's Guide to Microprocesssor Hardware, 2008.

[112] J. Hull, Options, Futures and Other Derivatives, 6th ed. Prentice Hall, 2005.

[113] Q. Jin, D. B. Thomas, W. Luk, "Exploring Reconfigurable Architectures for Explicit Finite Difference Option Pricing Models," In Proceedings of International Conference on Field Programmable Technology.

[114] J. Cong and Y. Zou, "FPGA-Based Hardware Acceleration of Lithographic Aerial Image Simulation", to appear at ACM Transactions on Reconfigurable Technology and Systems.

[115] O. Mencer and R. G. Clapp, "Accelerating 2d FFTs and Convolutions for Seismic Processing," Brief Notes by Maxeler Technologies, 2007.

[116] V. Podlozhnyuk, FFT based 2D Convolution, NVIDIA white paper, 2007.

[117] Xtremedata, *http* : *//www.xtremedatainc.com*, XD1000 Development System, 2007.

[118] D. Haessig, J. Hwang, S. Gallagher, M. Uhm, "Case-Study of a Xilinx System

Generator Design Flow for Rapid Development of SDR Waveforms," Proceedings of the SDR 05 Technical Conference and Product Exposition, 2005.

[119] Mathworks Inc., Simulink Documentation, *http://www.mathworks.com/access/helpdesk/help/toolbox/simulink/index.html?/access/helpdesk/help/toolbox/simulink/*

[120] Xilinx Inc., System Generator User Guide, *http://www.xilinx.com/support/documentation/sw_manuals/sysgen_bklist.pdf*

[121] Xilinx Inc., Put Hardware in the Loop with Xilinx System Generator for DSP, *http://www.xilinx.com/publications/xcellonline/xcell_47/xc_pdf/xc_sysgen47.pdf*, Xcell Journal Archives, 2003.

[122] Xilinx Inc., AccelDSP Synthesis Tool, User Guide, *http://www.xilinx.com/support/sw_manuals/acceldsp_user.pdf*, 2008.

[123] K. Lindberg and K. Nissbrandt, "An evaluation of methods for FPGA implementation from a Matlab description," Master's Degree Project, Stockholm, Sweden 2008.

[124] Xilinx Inc., MATLAB for Synthesis, Style Guide, *http://www.xilinx.com/support/sw_manuals/acceldsp_style.pdf*, 2008.

[125] Xilinx Inc., AccelWare Reference Designs, User Guide, *http://www.xilinx.com/support/sw_manuals/accelware_user.pdf*, 2008.

[126] Mathworks Inc., Simulink HDL Coder, User's Guide, *http://www.mathworks.com/access/helpdesk/help/pdf_doc/slhdlcoder/slhdlcoder_ug.pdf*

[127] Mathworks Inc., Stateflow and Stateflow Coder 7, *http://www.mathworks.com/access/helpdesk/help/pdf_doc/stateflow/sf_ug.pdf*

[128] R. Pasko, P. Schaumont, V. Derudder, V. Vernalde, and D. Durackova, "A new algorithm for elimination of common subexpressions," IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, 1999.

[129] H. Safiri, M. Ahmadi, G.A. Jullien, and W.C. Miller, "A New Algorithm for the Elimination of Common Subexpressions in Hardware Implementation of Digital Filters by Using Genetic Programming," Proceedings of the IEEE

International Conference on Application-Specific Systems, Architectures and Processors (ASAP), 2000.

[130] A. Hosangadi, F. Fallah, and R. Kastner, "Common subexpression elimination involving multiple variables linear DSP synthesis," Proceedings of the 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors, Pages 202-12, 2004.

[131] A. Hosangadi, F. Farzan, and R. Kastner, "Optimizing high speed arithmetic circuits using three-term extraction," Proceedings of the Design, Automation and Test in Europe, Pages 6, 2006.

[132] A. Azivienis, "Signed-Digit Number Representations for fast Parallel Arithmetic", IRE Trans. Elect. Comp., EC- 10, Pages 389-400, Sept. 1961.

[133] Mathworks Inc., Filter Design HDL Coder, User's Guide, *http* : *//www.mathworks.com/access/helpdesk/help/pdf_doc/hdlfilter/hdlfilter.pdf*

[134] Graphviz, Graph Visualization Software, *http* : *//www.graphviz.org/*

[135] Spiral, Software/Hardware Generation for DSP Algorithms, Finite/Infinite Impulse Response Filter Generator, *http* : *//www.spiral.net/hardware/filter.html*.

[136] B. Holland, M. Vacas, V. Aggarwal, R. DeVille, I. Troxel, and A. George, "Survey of C-based Application Mapping Tools for Reconfigurable Computing," Proceedings of 8th International Conference on Military and Aerospace Programmable Logic Devices (MAPLD), Washington, DC, September 7-9, 2005.

[137] Mitrionics AB, Inc., *http* : *//www.mitrionics.com/*.

[138] Mitrionics AB, Inc., Mitrion Users' Guide, *http* : *//forum.mitrionics.com/uploads/Mitrion/_Users/_Guide.pdf*.

[139] Celoxica, Inc., *http* : *//www.celoxica.com/*.

[140] Celoxica, Inc., Handel-C Language Reference Manual, *http* : *//www*28.*cs.kobe* − *u.ac.jp/ kawapy/class/proj/proj*07/*HandelC.pdf*.

[141] Nallatech, Inc., *http* : *//www.nallatech.com/*?*node_id* = 1.1.

[142] SRC Computers, LLC., *http* : *//www.srccomp.com/index.asp.*

[143] SRC Computers, LLC., MAP Processors,
http : *//www.srccomp.com/techpubs/map.asp.*

[144] SRC Computers, LLC., Carte Programming Environment, *http* : *//www.srccomp.com/techpubs/carte.asp.*

[145] Open SystemC Initiative (OSCI), *http* : *//www.systemc.org/home.*

[146] Open SystemC Initiative (OSCI), Defining and Advancing SystemC Standards, *http* : *//www.systemc.org/downloads/standards/*

[147] Cameron Project, Computer Science Department at Colorado State University,
http : *//www.cs.colostate.edu/cameron/index.html.*

[148] C.A.R. Hoare, Communicating Sequential Processes, Communications of the ACM, 21(8):666-677, August 1978.

[149] M. Gokhale, J. Stone, J. Arnold, M. Kalinowski,"Stream-oriented FPGA computing in the Streams-C high level language," In Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines, Pages 49-56, 2000.

[150] Impulse Accelerated Technologies, Inc., *http* : *//www.impulseaccelerated.com/.*

[151] Impulse Accelerated Technologies, Inc., *http* : *//www.impulseaccelerated.com/support*$_a$*ppnotes.htm.*

[152] Mentor Graphics Corp., *http* : *//www.mentor.com/.*

[153] Mentor Graphics Corp., Electronic System Level Design,
http : *//www.mentor.com/products/esl/high_level_synthesis/catapult_syn* − *thesis/*

[154] T. Bollaert, "Catapult Synthesis: A Practical Introduction to Interactive C Synthesis," SpringerLink Book Chapter, Pages 29-52, ISBN 978-1-4020-8587-1.

[155] M.B. Gokhale, J.M. Stone, "NAPA C: compiling for a hybrid RISC/FPGA architecture," In Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, Pages: 126 - 135, 1998.

[156] K. Lindberg and K. Nissbrandt, "An evaluation of methods for FPGA implementation from a Matlab description," Master's Degree Project, Stockholm, Sweden 2008.

[157] Xilinx Inc., Product Discontinuation Notice, AccelDSP Synthesis Tool, *http* : *//www.xilinx.com/support/documentation/customer_notices/xcn*09018.*pdf*

[158] Ali Irturk, Bridget Benson, Shahnam Mirzaei and Ryan Kastner, "GUSTO: An Automatic Generation and Optimization Tool for Matrix Inversion Architectures," ACM Transactions on Embedded Computing Systems.

[159] M. F. Jacome, G. de Veciana, V. Lapinskii, "Exploring Performance Tradeoffs for Clustered VLIW ASIPS," IEEE/ACM International Conference on Computer-Aided Design, 2000.

[160] F. Edman, V. Öwall, "A Scalable Pipelined Complex Valued Matrix Inversion Architecture," IEEE International Symposium on Circuits and Systems. (2005) 4489 - 4492.

[161] Karkooti, J.R. Cavallaro, C. Dick, "FPGA Implementation of Matrix Inversion Using QRD-RLS Algorithm," Thirty-Ninth Asilomar Conference on Signals, Systems and Computers (2005) 1625 - 162.

[162] C. Dick, F. Harris, M. Pajic, D. Vuletic, "Real-Time QRD-Based Beamforming on an FPGA Platform," Fortieth Asilomar Conference on Signals, Systems and Computers, 2006.

[163] R. Uribe, T. Cesear, "Implementing Matrix Inversions in Fixed-Point Hardware" *http* : *//www.dspdesignline.com/howto/*193104965, 2006.

[164] O. Hebert, I. Kraljic, Y. Savaria, "A Method to Derive Application-Specific Embedded Processing Cores," Eighth International Workshop on Hardware/Software Codesign (CODES), 2000.

[165] M. Pflanz, H. Viehaus, "Generating Reliable Embedded Processors," IEEE Micro, 1998.

[166] B. Gorjiara, D. Gajski, "Automatic Architecture Refinement Techniques for Customizing Processing Elements," 45th ACM/IEEE Design Automation Conference (DAC), 2008.

[167] Forte Design System Cynthesizer, *http* : *//www.fortеds.com/products/cynthesizer.asp*, 2008

[168] V. Kathail, S. Aditya, R. Schreiber, B. Ramakrishna Rau, D.C. Cronquist, M. Sivaraman, "PICO: Automatically Designing Custom Computers," Computer, 2002.

[169] J. Trajkovic, D. Gajski, "Custom Processor Core Construction from C Code," Symposium on Application Specific Processors (SASP), 2008.

[170] H. Zhong, K. Fan, S. Mahlke, M. Schlansker, "A Distributed Control Path Architecture for VLIW Processors," 14th Intl. Conference on Parallel Architectures and Compilation Techniques (PACT), 2005.

[171] J. Fisher, "Very long instruction word architectures and the ELI-52", 10th Annual International Symposium on Computer Architecture (ISCA), 1983.

[172] M. Chu, K. Fan, and S. Mahlke, "Region-based Hierarchical Operation Partitioning for Multicluster Processors," ACM SIGPLAN 2003 Conference on Programming Languages Design and Implementation (PLDI), 2003.

[173] K. Fan, N. Clark, M. Chu, K.V. Manjunath, R. Ravindran, M. Smelyanskiy, and S. Mahlke, "Systematic Register Bypass Customization for Application-Specific Processors," IEEE 14th Intl. Conference on Application-Specific Systems, Architectures and Processors (ASAP), 2003.

[174] M. L. Chu, K. C. Fan, R. A. Ravindran and S. A. Mahlke, "Cost-Sensitive Operation Partitioning for Synthesizing Custom Multicluster Datapath Architectures," 2nd Workshop on Application Specific Processors (WASP), 2003.

[175] N. Clark, H. Zhong, K. Fan, S. Mahlke, K. Flautner, and K. V. Nieuwenhove, "OptimoDE: Programmable Accelerator Engines Through Retargetable Customization," Hot Chips 16, 2004.

[176] M. Kudlur, K. Fan, M.Chu, and S. Mahlke, "Automatic Synthesis of Customized Local Memories for Multicluster Application Accelerators," 15th Intl. Conference on Application-Specific Systems, Architectures and Processors (ASAP), 2004.

[177] M. Kudlur, K. Fan, and S. Mahlke, "Streamroller: Automatic Synthesis of Prescribed Throughput Accelerator Pipelines," Proc. 2006 Intl. Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2006.

[178] M. Chu, R.Ravindran, and S. Mahlke, "Data Access Partitioning for Fine-grain Parallelism on Multicore Architectures," Proc. 40th Intl. Symposium on Microarchitecture (MICRO), 2007.

[179] A. Arfaee, A. Irturk, N. Laptev, R. Kastner, F. Fallah, "Xquasher: A Tool for Efficient Computation of Multiple Linear Expressions," Proc. of the Design Automation Conference (DAC 2009), July 2009.

[180] A. Irturk, B. Benson, N. Laptev and R. Kastner, "FPGA Acceleration of Mean Variance Framework for Optimum Asset Allocation." Proc. of the Workshop on High Performance Computational Finance at SC08 International Conference for High Performance Computing, Networking, Storage and Analysis, November 2008.