

UCLA

UCLA Electronic Theses and Dissertations

Title

High Performance Heterogeneous Acceleration: Exploiting Data Parallelism and Beyond

Permalink

<https://escholarship.org/uc/item/921805kz>

Author

Grigorian, Beayna

Publication Date

2014

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA

Los Angeles

**High Performance
Heterogeneous Acceleration:
Exploiting Data Parallelism and Beyond**

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

by

Beayna Grigorian

2014

© Copyright by
Beayna Grigorian
2014

ABSTRACT OF THE DISSERTATION

High Performance Heterogeneous Acceleration: Exploiting Data Parallelism and Beyond

by

Beayna Grigorian

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2014

Professor Glenn Reinman, Chair

Real-time, low-energy constraints as well as large amounts of data continue to challenge high performance computing (HPC). As a result, it has become increasingly important to advance the capabilities of high performance architectures. Single instruction multiple data (SIMD) designs are ideal for targeting data- and compute-intensive HPC workloads. Accelerator-rich architectures, in particular, implement application-specific functionality directly in hardware via on-chip accelerators, providing many orders of magnitude improvement in power efficiency and performance. Unlike instruction-based SIMD architectures, such as graphics processing units (GPUs), accelerator-rich designs avoid the overhead for processing instructions while maintaining flexibility by way of accelerator composition and virtualization.

This dissertation explores various aspects of hardware-based acceleration, including fine-grained vs. coarse-grained designs, ASIC-based vs. FPGA-based implementations, and domain-specific vs. domain-adaptive systems. While accelerator-rich designs are well-suited for exploiting data-level parallelism, they are highly susceptible (as are all SIMD architectures) to performance degradation

due to divergence in control flow. Since HPC workloads can contain various types and amounts of control flow, this SIMD divergence issue must be addressed in order for accelerator-based designs to yield more effective HPC platforms. As such, this work also investigates an approximation-based approach for eliminating control flow. We exploit the intelligent learning capabilities of neural networks to approximate and regularize the control flow regions of applications, thereby trading off precision for performance gains. Furthermore, we develop light-weight checks to ensure output reliability at runtime, allowing our neural-network-based approximations to be leveraged in a dynamically adaptive fashion.

Our work culminates in the following hybrid approach: a heterogeneous SIMD platform with both precise (conventional) and approximate (neural) accelerators, which are managed using online error control mechanisms. For the implementation, ASIC components are incorporated into the platform; also, approximation control methods, including NN training tools, static software interfaces, and dynamic hardware components, are developed to maintain acceptable error rates. Taking inspiration from the partial-observability and stochasticity of the world around us, this work combines data-parallel acceleration with neural approximation in an effort to advance high performance computation.

The dissertation of Beayna Grigorian is approved.

Adnan Darwiche

William Kaiser

Todd Millstein

Glenn Reinman, Committee Chair

University of California, Los Angeles

2014

*“Out of clutter, find simplicity.
From discord, find harmony.
In the middle of difficulty lies opportunity.”*

– Albert Einstein –

To my family . . .
who are my continued inspiration for
seeking out the opportunities within life’s difficulties.

Table of Contents

1	Introduction	1
1.1	Motivation Part 1: Hardware Acceleration	6
1.2	Motivation Part 2: SIMD Divergence	8
2	Accelerating Vision and Navigation Applications on a Customizable Platform	13
2.1	Application Domain	14
2.2	Methodology	16
2.3	Evaluation Approach	24
2.4	Experimental Results	25
3	ARC: Architecture Support for Accelerator-Rich CMPs	30
3.1	Overview of ARC	31
3.2	Evaluation Approach	42
3.3	Experimental Results	49
4	CHARM: Composable Heterogeneous Accelerator-Rich Microprocessor	55
4.1	Overview of CHARM	56
4.2	Evaluation Approach	63
4.3	Experimental Results	67
5	CAMEL: Composable Accelerator-Rich Microprocessor Enhanced for Longevity	73
5.1	Overview of CAMEL	75
5.2	Evaluation Approach	80
5.3	Experimental Results	86
6	Neural Acceleration of Divergent Applications on SIMD Architectures	90
6.1	Kernel Characterization	91
6.2	Methodology	93
6.3	Evaluation Approach	101
6.4	Experimental Results	103
7	Dynamically Reliable Approximate Computing Using Light-Weight Error Analysis	110
7.1	Methodology	111

7.2	Evaluation Approach	117
7.3	Experimental Results	118
7.4	Limitations	124
8	BRAINIAC: Bringing Reliable Accuracy Into Neurally-Implemented Approximate Computing	125
8.1	Overview of BRAINIAC	126
8.2	Neural Accelerator Design	129
8.3	Evaluation Approach	133
8.4	Experimental Results	137
9	Related Work	148
9.1	Accelerator-Rich Design	148
9.2	SIMD Divergence	150
9.3	Approximate Computing	151
9.4	Neural Network Implementation	153
10	Conclusion	155
	Bibliography	158

List of Figures

1.1	Example of warp divergence in a GPU	8
1.2	Pseudocode and profiling results for triangle intersection algorithm (<code>jmeint</code>)	10
1.3	Impact of divergence on GPU performance of Newton-Raphson algorithm	11
1.4	Performance gains from approximating Newton-Raphson algorithm with NNs of different topologies	11
2.1	3D CMP-FPGA platform	14
2.2	Performance and energy improvements using accelerators	26
2.3	Functional, application, and domain speedups based on varying interconnect models	28
3.1	ARC microarchitecture	31
3.2	Communication between CPU, GAM, memory, and accelerator	31
3.3	Light-weight interrupt support	32
3.4	Accelerator creation methodology	36
3.5	ARC development flow	36
3.6	Regression models for medical imaging benchmarks	38
3.7	An example of accelerator composition	41
3.8	Accelerator composition steps	41
3.9	Process used to generate simulation structures and accelerated programs	43
3.10	Speedup over <i>SW-only</i> (medical imaging domain)	44
3.11	Speedup over <i>SW-only</i> (computer vision and navigation domain)	45
3.12	Energy gain over <i>SW-only</i> (medical imaging domain)	46
3.13	Energy gain over <i>SW-only</i> (computer vision and navigation domain)	47
3.14	Speedup over <i>OS+Acc</i> (medical imaging domain)	48
3.15	Speedup over <i>OS+Acc</i> (computer vision and navigation domain)	49
3.16	Energy gain over <i>OS+Acc</i> (medical imaging domain)	50
3.17	Energy gain over <i>OS+Acc</i> (computer vision and navigation domain)	51
3.18	Error in accelerator runtime and wait time estimations	51
3.19	FFT virtualization (2D and 3D)	52
3.20	Benefit of using light-weight interrupts	52
3.21	Benefit of using hardware GAM (HW-GAM) over software GAM (SW-GAM)	53
4.1	Data flow graph for Denoise LCA	57
4.2	Microarchitecture of CHARM	57

4.3	LCA composition example: (A) a core sends a request for an LCA to the ABC; (B) an LCA instance is allocated; (C) another LCA instance is allocated with consideration for balancing DMA utilization; (D) the ABC signals completion to the core	61
4.4	Details of Poly ABB	64
4.5	Performance improvements of medical imaging applications	67
4.6	Energy gains of medical imaging applications	67
4.7	Effect of increasing accelerators	68
4.8	Utilization of ABBs given a task-grain of 8	69
4.9	Utilization of ABBs given a task-grain of 128	70
4.10	Performance improvements of computer vision and navigation applications	71
5.1	CAMEL Microarchitecture	75
5.2	Design of ABB island	75
5.3	Design of programmable fabric (PF)	76
5.4	Motivational example of applying rate-matching on PF	78
5.5	PF allocation algorithm	79
5.6	Compiler framework	79
5.7	Performance comparison between acceleration schemes	85
5.8	Energy usage comparison between acceleration schemes	85
5.9	Geometric mean of all speedups and energy savings as the percentage of PF increases	86
5.10	Geometric mean of speedup and energy savings for each domain as %-PF increases .	86
5.11	Domain longevity and graph partitioning impact for increasing percentages of PF . .	89
6.1	Overview of the Neuralizer software flow for automated neural acceleration of divergent applications	92
6.2	(A) Sample MLP and its mathematical representation based on labeled nodes and edge weights; (B) Example of integrating MLP of (A) into sample code (shaded regions represent the kernel-to-NN conversion)	97
6.3	CDF plot of benchmark error; point (x, y) signifies that y percent of the outputs see x percent of error or less	104
6.4	Performance gains of the various schemes	105
6.5	Energy savings of the various schemes	105
6.6	Dynamic instruction counts to verify the source of the performance and energy gains	106
6.7	Performance gains and accuracy for <code>jmeint</code> benchmark as kernel scope varies	107
6.8	Performance gains and accuracy for <code>invkin</code> benchmark as NN topology varies	108
7.1	Integration of LWC into application code	114
7.2	Performance comparison shown as speedup of <i>ORIG-n%</i> , <i>ACC+LWC</i> , and <i>ACC-LWC</i> over <i>ORIG-1%</i>	119
7.3	Performance breakdown for the <i>ACC+LWC</i> scheme; execution time broken down into time for (1) computation of neural accelerator (ACC), (2) evaluation of light-weight check (LWC), and (3) recovery using exact computation (Recovery)	119
7.4	Reliability issues (shown as failure to satisfy QoS constraints) when approximating without LWC support	121

7.5	Amount of coverage for out-of-range inputs	122
7.6	Speedup from coverage for out-of-range inputs	122
7.7	Amount of coverage with less accurate approximation	123
7.8	Speedup from coverage with less accurate approximation	123
8.1	Multi-stage acceleration consisting of N+1 stages (N approximate computation stages, 1 precise computation stage)	126
8.2	Microarchitecture of the BRAINIA C platform	127
8.3	Three main categories of architectural designs for NN hardware implementation: (A) fixed connections, fixed weights/biases; (B) flexible connections, flexible weights/bi- ases; (C) fixed connections, flexible weights/biases	130
8.4	(A) Internal architecture of reconfigurable neural accelerator (RNA); (B) Pseudocode for functionality of computational pipeline within RNA	131
8.5	Performance results for <code>invkin</code> benchmark	138
8.6	Performance results for <code>nrpoly</code> benchmark	138
8.7	Performance results for <code>physim</code> benchmark	139
8.8	Performance results for <code>roblloc</code> benchmark	139
8.9	Energy results for <code>invkin</code> benchmark	140
8.10	Energy results for <code>nrpoly</code> benchmark	140
8.11	Energy results for <code>physim</code> benchmark	141
8.12	Energy results for <code>roblloc</code> benchmark	141
8.13	Performance results for hardware acceleration of <code>invkin</code> benchmark in comparison to loop perforation implemented in software	142
8.14	Energy results for hardware acceleration of <code>invkin</code> benchmark in comparison to loop perforation implemented in software	142
8.15	Performance breakdown in terms of cycles executed for <i>RNA</i> scheme running <code>invkin</code> benchmark	143
8.16	Performance breakdown in terms of cycles executed for <i>RNA</i> scheme running <code>nrpoly</code> benchmark	143
8.17	Performance breakdown in terms of batches executed for <i>RNA</i> scheme running <code>invkin</code> benchmark	144
8.18	Performance breakdown in terms of batches executed for <i>RNA</i> scheme running <code>nrpoly</code> benchmark	144
8.19	Performance overhead for configuring RNAs	145
8.20	Varying coverage of NN2 in <code>physim</code> benchmark	145
8.21	Visualization of physics-based simulation with various error tolerance thresholds	147

List of Tables

1.1	OS overhead (cycles) to access accelerators	7
1.2	Accelerator building blocks (ABBs) used in medical imaging	7
2.1	Cycles, energy, and maximum benefits from accelerator candidates	18
2.2	Cycles, energy, and speedup statistics for implemented accelerator candidates	24
2.3	FPGA resource utilization for implemented accelerator candidates	24
2.4	Interconnect model parameters	27
3.1	Instructions used to interact with accelerators	33
3.2	Instructions to handle light-weight interrupts	33
3.3	Accelerated medical imaging benchmarks	42
3.4	Accelerated computer vision and navigation benchmarks	42
3.5	Simics+GEMS configuration	43
3.6	Sample synthesis results	44
4.1	Simulation parameters	63
4.2	Area/Power results – ABBs	64
4.3	Area/Power results – LCAs	65
4.4	Area (mm^2) for various chip components	65
5.1	Simulation parameters	81
5.2	Tools for timing and power models	81
5.3	ABB types, PF synthesis, domain numbers, and functionality	83
5.4	Power and area values for components of the <i>CAMEL</i> base platform	84
5.5	Number of ABBs and PF slices in <i>CAMEL-x%</i>	84
6.1	Summary of benchmark descriptions, domain categorizations, control flow characteristics, and justifications for approximability	100
6.2	NN characteristics, training results, and benchmark evaluation results for neural approximations	100
7.1	Examples of applications, algorithms, domains, and LWCs	114
8.1	RNA hardware parameters	132
8.2	Simulation parameters	134

8.3	Tools for timing and power models	134
8.4	Summary of benchmarks and LWCs	134
8.5	Summary of trained neural network models	134
8.6	Power and area values of PAs and CNAs (NN1 and NN2)	136

ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest gratitude to my advisor, Professor Glenn Reinman, for his continuous support and guidance throughout my education. I first met Professor Reinman as an undergraduate, where his introductory course on computer systems architecture initially sparked my desire to pursue research in this field. I am sincerely grateful for the countless hours and energy he spent as both my undergraduate and graduate research advisor, where our meeting discussions ranged from computer design perspectives to pop culture references. As someone I both respect and admire, Professor Reinman was truly the ideal advisor for me and my greatest inspiration.

I would like to offer special thanks to my doctoral committee members for their invaluable feedback on my dissertation. I also wish to acknowledge Narayan Srinivasa, my manager at HRL Laboratories, who provided me with the most rewarding internship of my career, introduced me to the world of neuromorphic computing, and inspired me to incorporate neural networks in my research. In addition, I would very much like to thank the National Science Foundation (Graduate Research Fellowship Grant # DGE-0707424 and Expedition in Computing Award # CCF-0926127), the UCLA Center for Domain-Specific Computing (CDSC), and the Center for Future Architectures Research (C-FAR, which is one of six centers of STARnet, an SRC program sponsored by MARCO and DARPA). Their financial support was vital to my graduate career as it allowed me to remain focused on my research and education.

The following are my acknowledgments of co-authored work, broken down by chapter:

- ▶ Chapter 2 is based on “Accelerating Vision and Navigation Applications on a Customizable Platform,” in the proceedings of the 22nd IEEE International Conference on Application-specific Systems, Architectures, and Processors (ASAP), 2011, pages 25–32. This work was a collaborative effort with Marco Vitanza, and was supervised by Professors Glenn Reinman and Jason Cong.
- ▶ Chapter 3 is a version of “Architecture Support for Accelerator-Rich CMPs,” in the proceedings of the 49th ACM/EDAC/IEEE Design Automation Conference (DAC), 2012,

pages 843–849, and Chapter 4 is a version of “CHARM: A Composable Heterogeneous Accelerator-Rich Microprocessor,” in the proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED), 2012, pages 379–384.

These projects were done collaboratively with Mohammad Ali Ghodrat, Michael Gill, Professor Glenn Reinman, and Professor Jason Cong.

- ▶ Chapter 5 is based on “Composable Accelerator-rich Microprocessor Enhanced for Adaptivity and Longevity,” in the proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED), 2013, pages 305–310. This research was conducted in collaboration with Michael Gill, Mohammad Ali Ghodrat, Hui Huang, Professor Glenn Reinman, and Professor Jason Cong.
- ▶ Chapter 6 is a version of “Accelerating Divergent Applications on SIMD Architectures Using Neural Networks,” which is to appear in the proceedings of the 32nd IEEE International Conference on Computer Design (ICCD), 2014, and Chapter 7 is a version of “Dynamically Adaptive and Reliable Approximate Computing Using Light-Weight Error Analysis,” in the proceedings of the 9th NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2014, pages 248–255. The work for these papers was overseen by Professor Glenn Reinman.
- ▶ Chapter 8 is based on “BRAINIAC: Bringing Reliable Accuracy Into Neurally-Implemented Approximate Computing”, submitted to the 21st IEEE International Symposium on High Performance Computer Architecture (HPCA), 2015. This work was performed in collaboration with Nazanin Farahpour and Professor Glenn Reinman.

I am truly grateful to my fellow PhD students and labmates, who in many ways provided help and support over the years. This includes Michael Gill, Mohammad Ghodrat, Robert Chen, Marco Vitanza, Nazanin Farahpour, Konstantine Tsotsos, Karthika Mohan, and Alessandra Scalfuro. Whether it was staying up late working in lab, having long-winded intellectual discussions, or taking all-too-necessary coffee breaks, these people were central figures in my graduate school experience.

Last but certainly not least, I would like to express my deep appreciation for my family: my parents, for always encouraging me to put education first and strive for goals beyond my grasp; my sisters,

with whom I have a bond that transcends familial ties; my brother-in-law, Harut(ik), for being the brother I never had; and my aunts, uncles, and all 20 of my first cousins, for making my life feel like an Armenian version of *My Big Fat Greek Wedding* [1]. I also sincerely appreciate having the following special people in my life: my best friends since high school, Manda Paul and Meghedy Shanazarian, and my trusty accomplice, Dianne Pulido. Finally, I would like to thank Alex... for entering my life with impeccable timing.

VITA

- 2007-2010 Undergraduate Researcher, Center for Embedded Networked Sensing,
University of California, Los Angeles
- 2008-2009 Software Engineering Intern, Jet Propulsion Laboratory, Pasadena, CA
- 2010 Shivakumar Scholarship for Undergraduate Research,
Computer Science Department, University of California, Los Angeles
- 2010 Engineering Achievement Award for Student Welfare,
University of California, Los Angeles
- 2010 Outstanding Bachelor of Science Award, Computer Science Department,
University of California, Los Angeles
- 2010 B.S. (Computer Science and Engineering),
University of California, Los Angeles
- 2010 Software Engineering Intern, Qualcomm Incorporated, San Diego, CA
- 2010-2014 Graduate Researcher, Computer Science Department,
University of California, Los Angeles
- 2011 Google Anita Borg Memorial Scholarship,
Google Incorporated, Mountain View, CA
- 2011 Graduate Technical Intern, Intel Corporation, Santa Clara, CA
- 2011-2014 Graduate Research Fellowship,
National Science Foundation, Arlington, VA
- 2012 M.S. (Computer Science), University of California, Los Angeles
- 2012 Computer Engineer, HRL Laboratories, Malibu, CA
- 2014 Registration Fee Grant for Distinguished Students, Graduate Division,
University of California, Los Angeles
- 2014 Contract Research and Development Achievement Award,
HRL Laboratories, Malibu, CA

PUBLICATIONS

- Beayna Grigorian and Glenn Reinman, “Accelerating Divergent Applications on SIMD Architectures Using Neural Networks,” to appear in Proceedings of the 32nd IEEE International Conference on Computer Design (ICCD), 2014.
- Beayna Grigorian and Glenn Reinman, “Dynamically Adaptive and Reliable Approximate Computing Using Light-Weight Error Analysis,” in Proceedings of the 9th NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2014, pp. 248–255.
- Jason Cong, Mohammad Ali Ghodrati, Michael Gill, Beayna Grigorian, Karthik Gururaj, and Glenn Reinman, “Accelerator-Rich Architectures: Opportunities and Progresses,” in Proceedings of the 51st ACM/EDAC/IEEE Design Automation Conference (DAC), 2014, pp. 1–6.
- Jason Cong, Mohammad Ali Ghodrati, Michael Gill, Beayna Grigorian, and Glenn Reinman, “Architecture Support for Domain-Specific Accelerator-Rich CMPs,” in ACM Transactions on Embedded Computing Systems (TECS), 2014, vol. 13, no. 4s, pp. 131:1–131:26.
- Narayan Srinivasa, Deying Zhang, and Beayna Grigorian, “A Robust and Scalable Neuromorphic Communication System by Combining Synaptic Time-Multiplexing with MIMO-OFDM,” in IEEE Transactions on Neural Networks and Learning Systems (TNNLS), 2014, vol. 25, no. 3, pp. 585–608.
- Jason Cong, Mohammad Ali Ghodrati, Michael Gill, Beayna Grigorian, Hui Huang, and Glenn Reinman, “Composable Accelerator-rich Microprocessor Enhanced for Adaptivity and Longevity,” in Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED), 2013, pp. 305–310.
- Jason Cong, Mohammad Ali Ghodrati, Michael Gill, Beayna Grigorian, and Glenn Reinman, “CHARM: A Composable Heterogeneous Accelerator-Rich Microprocessor,” in Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED), 2012, pp. 379–384.
- Jason Cong, Mohammad Ali Ghodrati, Michael Gill, Beayna Grigorian, and Glenn Reinman, “Architecture Support for Accelerator-Rich CMPs,” in Proceedings of the 49th ACM/EDAC/IEEE Design Automation Conference (DAC), 2012, pp. 843–849.
- Jason Cong, Beayna Grigorian, Glenn Reinman, and Marco Vitanza, “Accelerating Vision and Navigation Applications on a Customizable Platform,” in Proceedings of the 22nd IEEE International Conference on Application-specific Systems, Architectures, and Processors (ASAP), 2011, pp. 25–32.

Chapter 1

Introduction

Some of the most challenging applications facing computer scientists today feature real-time performance constraints, ultra-low power requirements, and significant computational complexity. These requirements are often at odds with one another, however. Applications trying to attain high levels of situational awareness, for instance, would have a difficult time meeting both low-energy and real-time constraints. An example of this could be found in autonomous robotics, where the system must ensure that intelligent decisions are made to allow a robot to safely explore and interact with its environment. While it is imperative that the robot responds in real-time to its dynamically-changing surroundings, this type of mobile system is likely also limited by the total amount of energy it can consume.

Real-time processing is also continuously challenged by ever-increasing amounts of data. A recent statement from the Defense Advanced Research Projects Agency (DARPA) describes exponential increases in volumes of sensor data and similar increases in complexity of analysis, which have led to an outpacing of the processing capabilities of our computing systems [2]. Following this onset of very large data sets, research has also found that problems can often be reformulated into more data-intensive ones to achieve better results [3, 4]. Consequently, more and more algorithms are being implemented with the assumption that large amounts of data are accessible at their disposal, resulting in increasing amounts of data-intensive workloads.

Many of these challenging modern-day applications fall under the umbrella of what would be considered high performance computing (HPC). HPC, which refers to compute-intensive workloads that operate on large data sets and are in general need of acceleration [5], includes such areas as computer vision, interactive entertainment, medical imaging, and financial modeling. Such workloads are often targeted by single instruction multiple data (SIMD) architectures [6–10] to exploit the data parallelism that is often inherent in these applications. Accelerator-rich platforms [11–16], in particular, are well-suited for targeting these workloads. By implementing application-specific functionality directly in hardware, on-chip accelerators can provide many orders of magnitude improvement in power efficiency and performance for a variety of tasks ranging from multiply-accumulate operations to complex encoding/decoding algorithms. Furthermore, unlike instruction-based SIMD architectures [9, 17], accelerator-rich designs avoid the overhead for processing instruction streams (e.g. instruction fetch/decode, register value migration, etc.), while maintaining flexibility by way of accelerator chaining and virtualization (refer to Chapters 2, 3, 4, and 5).

Despite these significant benefits, accelerator-rich platforms, much like other SIMD architectures, suffer from performance degradation due to divergence in control flow (i.e. *branch divergence*). The reason for this is because, by definition, SIMD architectures compute multiple data sets in lock-step via wide datapaths under a single control flow [18, 19]. While this design choice allows us to amortize the cost of instruction processing over multiple datapaths, it fundamentally makes SIMD less effective at dealing with diverse control flow, particularly when control flow is data-dependent. For HPC workloads, there can exist various types and amounts of control flow, often having dependencies on input data [20]. In linear algebraic processing, for instance, robust stencil operations are used for dense matrices while sparse matrices are solved using iterative methods. It is therefore evident that in order for accelerator-rich architectures to be more effective at high performance computation, the SIMD divergence issue must be addressed.

Applications that exhibit SIMD divergence problems, or “divergent applications”, have been dealt with in a variety of ways in prior art. One approach is predication, in which all paths are executed sequentially and masks are used to ensure data correctness [21, 22]. With this approach, branch divergence may at best be avoided by conversion into a data dependency. However, this results in

inefficiency due to added overhead for data generation and processing, as well as reduced *useful* utilization from uncommitted results. Similar to predication, accelerator-rich designs either defer the handling of control flow to the core, or bloat the circuitry of the custom accelerators (i.e. the results for all possible paths are computed and multiplexed to arrive at the final correct result). While these methods ensure program correctness, they can be costly in terms of performance as well as area and power consumption. Other prior art has focused on application-specific modification to avoid divergence (e.g. data migration [23]), algorithmic modification to restructure control flow [24], or compiler-based optimization [25], but these techniques do not work in all situations, such as with applications containing data-dependent control flow.

Unlike prior art that handles divergence by tolerating the aftermath of control flow, we explore an approach that views this problem from an entirely different angle: we exploit the intelligent learning capabilities of neural networks (NNs) to approximate away the control flow, and as a result, the divergence itself (see Chapter 6).

An artificial neural network (ANN) is a fundamental machine learning paradigm for performing non-parametric multivariate regression [26]. It innately has the ability to subsume computation in exchange for an approximate representation [27], making it an ideal computational tool for nonlinear regression and function approximation. Developments in advanced learning techniques for training ANNs [28] have also made them more flexible than various polynomial regression models. The self-organizing, nonlinear learning capabilities of ANNs and their success as noise-immune, general-purpose classifiers [29] suggest they are well suited for approximating general regions of computation, including regions with diverse control flow.

From an architectural standpoint, general-purpose SIMD designs are performance-wise sensitive to data dependencies, while NNs are custom trained to incorporate data dependencies directly into their network configurations so as to successfully map input values to output values. The structural parallelism of NNs also enables efficient implementation in hardware, including ultra low-power neuromorphic implementations [30]. Furthermore, since the evaluation of NNs does not include any control flow, divergent computation can be transformed into a regularized, approximate

form, thereby translating the challenges of data dependency from the performance domain to the accuracy domain.

Although regularization of control flow could improve performance, we must account for the resulting loss in computational accuracy. First, we note that many high performance applications, such as the “Recognition, Mining, and Synthesis (RMS)” workload from Intel [31], are based on heuristics that can be approximated (e.g. with the use of neural networks [27]). Second, applications that use exact computation often also include regions of computation that are tolerant to imprecision, or “approximable”, even if these regions can only be circumstantially approximated (e.g. when the application is used as part of a compute pipeline that yields a final approximate result). In either case, the accuracy of the approximations can additionally be customized to fit the needs of specific users. Based on these observations, we see a general need for approximation control mechanisms in our approach.

Prior art has extensively studied computational resilience to imprecision in both hardware [32–36] and software [37–41]. These existing approaches often couple the overall application quality with the accuracy of the approximation unit. Though this allows for efficient quality analysis by way of offline, static techniques [42], *coverage* and *reliability* are potentially compromised.

Coverage is lost when cases that are statically determined to lead to unacceptably inaccurate solutions are exempted from approximation. This may occur, for instance, with inputs outside the range of training data or with the use of otherwise inaccurate approximate accelerators. One approach to combat this loss in coverage is to exploit algorithmic and cognitive resilience [35], such as with high performance workloads from Recognition, Mining, and Synthesis (RMS) [31], potentially uncovering larger slack in accuracy. Related work includes methodologies that employ high-level, application-specific metrics for assessing output quality [34, 38, 43, 44]. These metrics, however, often determine quality degradation of approximate solutions by measuring against exact solutions, such as by finding the average difference between images produced by approximate and exact versions of an image processing application.

Dynamic *reliability* entails providing absolute guarantees for satisfying quality of service (QoS) constraints. With static quality analysis, guarantees on worst-case accuracy cannot be provided unless the space of possible inputs is exhaustively explored. As this is an inherent issue for approximation techniques, instead of providing worst-case guarantees and ensuring dynamic reliability, measures are often taken to statically mitigate low-quality results [45]. Despite its statistical unlikelihood, however, a low-quality result may potentially render an application’s output meaningless. This would be unacceptable for circumstances that involve strict QoS expectations. As such, we recognize a general need for mechanisms that allow dynamic analysis and control of approximation errors (much like with circuit-level errors [33]). Moreover, since acceptable ranges of error could vary across different uses of an application, user-based specification of QoS constraints must also be featured. An example of a use case for this would be an inverse kinematics application for robot control, which requires high precision for performing surgery, yet tolerates lower precision for moving large blocks.

With coverage and reliability in mind, we have designed an approach based on light-weight checks (LWCs) (refer to Chapter 7). These high-level, application-specific metrics allow us to decouple error analysis of the approximate accelerator from quality analysis of the overall application. As a result, rather than relying on statically-constructed models of error distribution [42, 45, 46], we can dynamically guarantee the worst-case error for an application, as well as gain additional coverage from leveraging slack that may not be evident at the level of the approximation unit.

Returning to our overarching goal of enhancing high performance computation, this work culminates in the following hybrid approach: a heterogeneous SIMD platform that combines conventional acceleration techniques with neural-network-based approximate computing, while reliably enforcing constraints on accuracy at runtime (detailed in Chapter 8). To gain maximum performance benefits and energy efficiency from our neural approximations, we shy away from a platform-agnostic approach and opt for hardware-based implementations that can be incorporated into our platform as accelerators. We refer to conventional accelerators as *precise* accelerators to distinguish them from our *approximate* accelerators.

The accelerators in our platform are leveraged in a multi-stage flow that begins with simple approximations and resorts to more complex ones as needed. We employ LWCs to throttle this multi-stage acceleration flow and dynamically maintain user-specified quality in outputted results. In terms of implementation, ASIC-based hardware accelerators are incorporated into the platform. Additionally, approximation control methods, including NN training tools, static software interfaces, and dynamic hardware components, are developed to maintain acceptable error rates.

The remainder of this doctoral dissertation is organized as follows. Chapters 2, 3, 4, and 5 describe preliminary work on accelerator platforms. These architectures vary in both characterization and usage of accelerators, thereby providing a broad spectrum of design choices for high performance acceleration. Next, Chapter 6 presents a software-based approach for neural acceleration of divergent SIMD applications, and Chapter 7 describes an LWC-based technique for approximation control. Combining the principles introduced in the previous chapters, Chapter 8 then presents a heterogeneous accelerator-rich platform that carefully leverages hardware-based precise (conventional) and approximate (neural) accelerators to gain further performance benefits while dynamically ensuring acceptable output quality using LWCs. Finally, we review related work in Chapter 9 and conclude with Chapter 10.

1.1 Motivation Part 1: Hardware Acceleration

1.1.1 Accelerator Management

In a typical accelerator-based system, a core accesses an accelerator by way of a driver (OS call) [47, 48]. Using the Simics+GEMS [49, 50] simulation platform to model a system consisting of UltraSPARC-III-i processors running Solaris 10, we have measured the delay for different system call operations. These results are shown in Table 1.1, where “ioctl” refers to the system call for device-specific operations. We see that for an accelerator-rich platform, this software-based approach to accelerator management can become very inefficient, in terms of performance as well

Table 1.1: OS overhead (cycles) to access accelerators

Operation	1 Core	2 Cores	4 Cores	8 Cores	16 Cores
Open Driver	214,413	256,401	266,133	308,434	316,161
ioctl (Average)	703	725	781	837	885
Interrupt Latency	16,383	20,361	24,022	26,572	28,572

Table 1.2: Accelerator building blocks (ABBs) used in medical imaging

ABBs	Denoise	Deblur	Registration	Segmentation
Float Reciprocal (FInv)	✓	✓		✓
Float Square-Root (FSqrt)	✓	✓	✓	✓
Float Polynomial-16 (Poly16)	✓	✓	✓	✓
Float Divide (FDiv)	✓	✓	✓	✓

as energy. Therefore, to minimize overhead, we investigate efficient hardware-based accelerator management of loosely-coupled accelerators.

1.1.2 Accelerator Composition

Tasks performed by coarse-grained *loosely-coupled accelerators (LCAs)* tend to have a great deal of data parallelism, which can be exploited by the use of multiple LCAs of the same type. However, there is often a variety of LCAs (with possible overlap) required by different applications, resulting in sporadic usage of any particular LCA. In support of these observations, our experiments demonstrate that while LCAs have dramatic potential to improve performance and reduce power dissipation, their overall utilization is low. Due to variations in accelerator resource demand, LCAs in the medical imaging domain, for instance, are idle over 90% of the time, revealing an opportunity for accelerator sharing.

In addition to the under-utilization of coarse-grained LCAs, we have found that applications in a single domain can have a large degree of similarity in the type of computation primitives being used. This indicates that the applications can be accelerated using a limited number of smaller, more general *accelerator building blocks (ABBs)*. Table 1.2 depicts how the applications in the medical

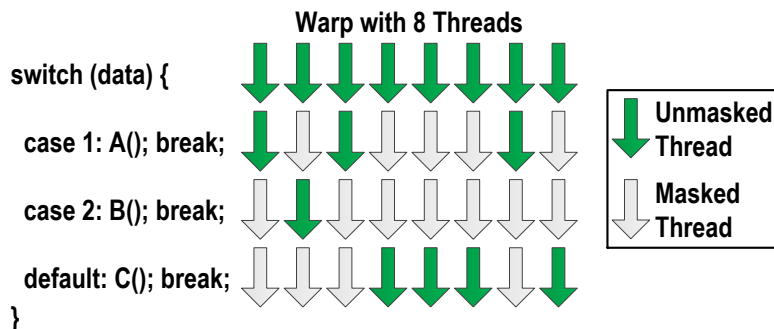


Figure 1.1: Example of warp divergence in a GPU

imaging domain [51] can be decomposed into 4 main types of ABBs, emphasizing the considerable overlap in application functionality, which could be leveraged to improve accelerator utilization.

While sufficient diversity of LCAs allows acceleration of a variety of applications, sufficient quantity of a particular LCA is needed to handle the parallelism within a particular kernel. Therefore, rather than attempting to provide monolithic LCAs for all possible execution scenarios, we opt to provide a set of ABBs. These fine-grained building blocks could then be dynamically linked in a variety of ways to compose desired coarse-grained LCAs on-the-fly.

1.2 Motivation Part 2: SIMD Divergence

1.2.1 Branch Divergence in SIMD

Divergent control flow, also known as *branch divergence*, can have a significant impact on the performance of a SIMD architecture. With GPUs, for instance, this problem is often referred to as “warp divergence”. In this case, threads clustered in a given warp are all subjected to execution of any path that even a single one of them follows. Since the system masks away the compute results of threads that were not intended to execute that path (i.e. their compute values are invalidated and remain uncommitted in order to maintain program correctness), this could result in a large amount of excessive computation. In fact, recent performance studies on GPUs have shown that

as we increase the number of divergent threads in warps, the overhead of divergent control flow increases linearly [52].

In Figure 1.1 we provide an example of warp divergence, where a warp of eight threads comes across a data-dependent switch statement (shown at the left of the figure). The switch statement has three possible outcomes (i.e. three cases), each of which invokes a particular function (A, B, or C). In the right part of the figure, the green “unmasked thread” arrows indicate which of the eight threads follow the various paths of the switch statement – i.e. three of the threads fall within case 1 (which invokes function A), one thread falls within case 2 (which invokes function B), and four threads fall within the third “default” case (which invokes function C). We can see there is a large degree of divergence in this example because all three of the functions are computed by all threads, while the majority of the results are masked away. For example, while all threads will execute function B, only one thread will actually make use of the results of this execution. Note further that if this one thread did not fall within case 2 of the switch statement, none of the threads would need to execute function B, and there would be less divergence. We now briefly examine two motivational case studies.

1.2.2 Case Study I: Triangle Intersection

The triangle intersection algorithm used in graphics applications is an example of practical code that can be quite control-flow-heavy. It takes as input the vertices of two triangles and determines if they intersect. The pseudocode for the algorithm (based on the `jmeint` benchmark described in Section 6.3.1) is shown in Figure 1.2. Within this pseudocode, the number of floating point operations (*FP Ops*) and comparisons (*Compares*) is specified on the right-hand side of each compute line. Also, the bottom of the figure lists the portions of execution time (*% of Program Execution Time*) consumed by each of the four highlighted kernel scopes. The percentages of execution time are shown as average ranges based on a series of randomized inputs, allowing us to see that the breakdown of execution time can vary drastically depending on data. Overall, we observe that a series of significant chunks of computation are segregated by non-predictable, data-dependent

```

intersection(Tri1Vertex1, Tri1Vertex2, Tri1Vertex3, Tri2Vertex1, Tri2Vertex2, Tri2Vertex3):
1  Compute distances between Tri2 vertices and Tri1 plane    ▶ 40 FP Ops, 12 Compares
2  if (Tri2 does not intersect plane of Tri1)
3    return false
4  else
5    Compute distances between Tri1 vertices and Tri2 plane ▶ 40 FP Ops, 12 Compares
6    if (Tri1 does not intersect plane of Tri2)
7      return false
8    else
9      Compute values for coplanarity test                    ▶ 48 FP Ops, 15 Compares
10     if (Tri1 and Tri2 are coplanar)
11       Compute values for edge crossing test                ▶ 126 FP Ops, 102 Compares
12       if (Any Tri1 edge crosses any Tri2 edge)
13         return true
14       else
15         Compute values for containment test                 ▶ 29 FP Ops, 2 Compares
16         if (Tri2 contains Tri1 || Tri1 contains Tri2)
17           return true
18         else
19           return false
20     else
21       Compute values for non-coplanar intersection test    ▶ 17 FP Ops, 4 Compares
22       if (Non-coplanar intersection exists)
23         return true
24       else
25         return false

```





	Kernel Scope (Level of Branch Nesting)	% of Program Execution Time (Varies Based on Data)
	1st Level	10.85% – 38.51%
	2nd Level	11.58% – 27.23%
	3rd Level	13.42% – 26.90%
	4th Level	7.35% – 64.15%

Figure 1.2: Pseudocode and profiling results for triangle intersection algorithm (jmeint)

control flow. As the divergence is data-dependent, there is no way to format or reorganize the data a priori to avoid it. Static branch resolution and precomputation are therefore ineffective options. Alternatively, if one were to break up the kernel into multiple sub-kernels based on points of divergence, this would result in (1) overhead for binning intermediate values, (2) loss in data locality, and (3) overhead for sorting output data of each sub-kernel and reinitializing computation for subsequent sub-kernels. Hence, the inadequacies of these existing solutions call for a better approach to overcoming branch divergence.

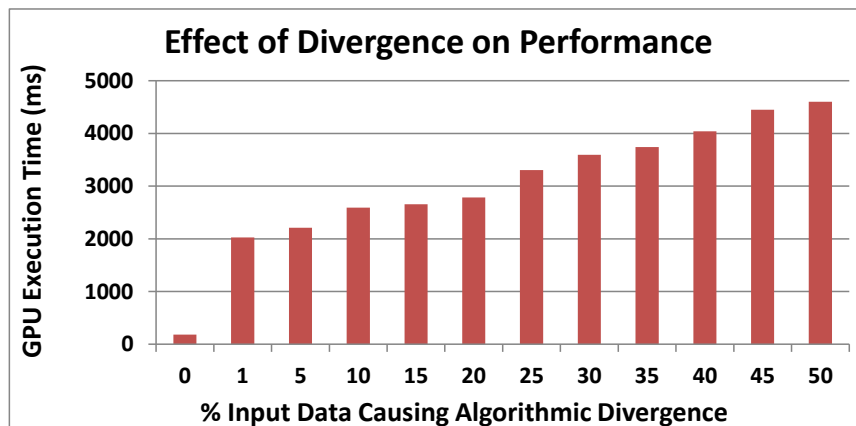


Figure 1.3: Impact of divergence on GPU performance of Newton-Raphson algorithm

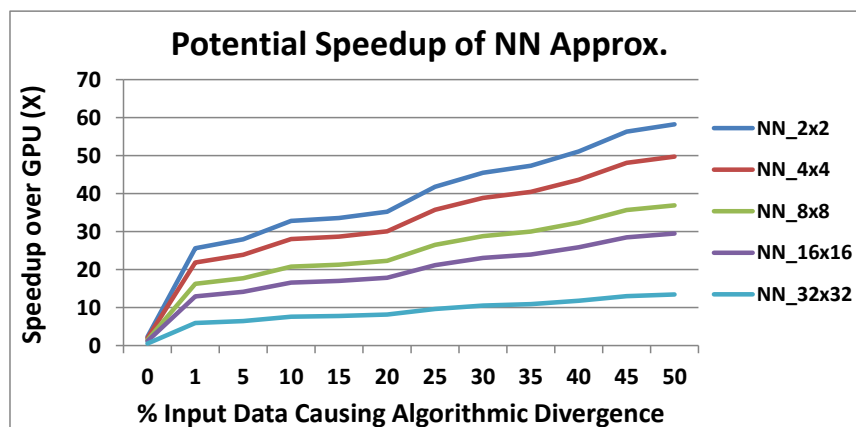


Figure 1.4: Performance gains from approximating Newton-Raphson algorithm with NNs of different topologies

1.2.3 Case Study II: Newton-Raphson Method

The Newton-Raphson method [53] is an iterative approach for finding roots of an equation. The algorithm starts from an initial guess provided by the user, and iteratively refines the solution until either it finds an exact root of the given equation, or is able to provide an approximation of the root within tolerable error bounds. While it is a powerful approximation-based approach, this algorithm is not guaranteed to converge and return an acceptable result. In other words, when there is *algorithmic divergence*, the program continues processing for some number of iterations before determining it cannot converge to an acceptable result, and consequently returns a failure indicator. The four main cases of *algorithmic divergence* for the Newton-Raphson method are:

(1) divergence at inflection points, (2) zero-valued derivatives (leading to division by zero), (3) oscillations near local maxima or minima, and (4) root jumping (e.g. for oscillating functions with many roots). These cases result from both the characteristics of the equations themselves as well as the initial guess fed into the algorithm. To demonstrate the impact of divergence on GPU performance, we have taken the Newton-Raphson method and generated synthetic input data, where we intentionally include values that will disallow the algorithm to converge to an acceptable result (i.e. will lead to *algorithmic divergence*). Figure 1.3 displays the impact on GPU performance when a given percentage of the data causes *algorithmic divergence*. Notably, the performance of the GPU degrades by $11\times$ going from 0% to even 1% divergence.

To overcome this performance degradation, we introduce the use of NNs that approximate the divergent code. Since this algorithm is essentially one large while loop, we approximate the entire benchmark with a single NN and evaluate the mathematical representation of that NN (i.e. compute a series of multiplication, summation, and sigmoid operations – please refer to Section 6.2.3 for more information on integrating an NN into code) instead of the original control-flow-heavy Newton-Raphson algorithm. We first see the potential performance benefits of neural approximation, temporarily holding off on evaluating loss of accuracy, which is addressed in Section 6.4. Figure 1.4 demonstrates the performance gains achieved as we alter the topology of the NN. Here, an $NN_M \times N$ topology represents an NN with two hidden layers, the first with M number of neurons and the second with N – Section 6.2.2 provides a detailed description of the NN model. As expected, the smaller NN topologies achieve higher speedups as they require less computation to evaluate their outputs. Yet we note that even the largest topology, which has two hidden layers of 32 neurons each, achieves a $6\times$ performance gain when as little as 1% of the data leads to *algorithmic divergence*. Moreover, all the other topologies see performance gains even when there is 0% divergence, which highlights the added benefit of including non-divergent computation in the code region being approximated. All in all, while this case study could be considered an extreme example of branch divergence, it shows how severely a SIMD architecture can be bogged down and how impactful it can be to have a solution that converts divergent control flow into non-divergent computation.

Chapter 2

Accelerating Vision and Navigation Applications on a Customizable Platform

For high performance applications in the vision and navigation domain, we investigate acceleration with a customizable, FPGA-based design [54]. Our overall goal is to achieve low-energy real-time performance, demonstrated through reductions in energy-delay product (EDP), for a set of key algorithms within this domain – specifically, feature tracking along with localization and mapping. To avoid degradation in the quality of results, our approach uses hardware accelerators to speed up existing full-featured application code.

Our system model is a 3D integration of a field-programmable gate array (FPGA) atop a standard chip multiprocessor (CMP), with through-silicon vias (TSVs) as the communication medium between the two layers (Figure 2.1) – a more detailed description of the architecture can be found in [55]. With the use of TSVs, our design provides efficient and direct CMP-FPGA communication; however, we also make comparisons with other potential interconnects to ensure that our acceleration techniques are not limited to the assumed model.

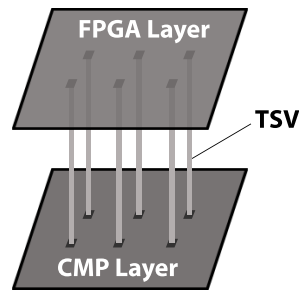


Figure 2.1: 3D CMP-FPGA platform

In this scheme, applications run normally on the CMP, except for key portions that are executed by hardware accelerators implemented on the FPGA. FPGAs are an attractive option because their customizability can lead to large reductions in both computation time and energy consumption when compared to software running on a CPU, yet they have a relatively low development cost compared to ASIC designs. Furthermore, with the use of a C-to-RTL high-level synthesis tool (AutoPilot [56]), we are able to quickly create and analyze a wide range of potential accelerators in a semi-automated fashion. With this approach, we model an accelerator platform that can be readily customized to the needs of specific workloads.

2.1 Application Domain

Pattern detection, object recognition, and feature tracking are among the most notable areas in computer vision [57]. Feature tracking uses various methods for initially detecting features (i.e. points of interest) within a given data set, followed by extracting descriptions of those features [58–60]. The detected features are subsets of the data, for instance edges or corners in an image, and are described using additional computations, such as surrounding gradients or local histograms. There also exist several methods [61–64] to solve the simultaneous localization and mapping problem (SLAM) [65]; these entail dynamically collecting data from the environment in order to create a local map that identifies free space and obstacles while simultaneously self-localizing within this map.

In this study, we focus on four specific algorithms to represent the vision and navigation domain: SLAM based on the extended Kalman filter (EKF) [66, 67]; feature detection and description using the scale-invariant feature transform (SIFT) algorithm [68]; feature detection and description using the speeded-up robust features (SURF) algorithm [69]; and feature detection using the Kanade-Lucas-Tomasi corner-detection algorithm (KLT) [70] with feature description using the intensity-domain spin image (IDSI) algorithm [71].

We define the domain using this set of applications for several reasons. First, EKF is a widely-accepted, standard method for nonlinear state estimation and navigation [67, 72–74]. Second, feature tracking is a large area of research within computer vision, and includes many widely-used algorithms. Although typically only one would be used by a particular application, by surveying multiple algorithms we intend to identify which ones are more amenable to our method of acceleration. In doing so, we are able to identify algorithmic characteristics that should be sought after in other applications to allow for successful acceleration.

As benchmarks, we use the sample applications `kf-slam` and `features-matching` from the Mobile Robot Programming Toolkit (MRPT) [75], which uses the Open Source Computer Vision (OpenCV) library [76] for the implementations of the SURF and KLT algorithms.

The `kf-slam` application runs an EKF-based SLAM iteration for each action-observation pair in a sample data set. An action-observation pair represents a state change by way of a pose transition and an observation of the new state. A complete iteration of EKF-based SLAM involves the following steps: (1) acquire the action data and linearize the dynamics of the transition based on the previous filtered state estimate; (2) predict the next state estimate, compute transition Jacobians, and predict the covariance of the next state estimate; (3) acquire the observation data and linearize the dynamics of the observation based on the predicted next state estimate; (4) compute observation Jacobians and predict observation covariances; (5) finally, using the near-optimal Kalman gain, update the next state estimate and its covariance.

The `features-matching` application has three main stages. First, it runs a feature detection algorithm (SIFT/SURF/KLT) on a sample image to find feature points. Next, it computes descriptors

(SIFT/SURF/IDSI) for each feature. Finally, it matches the features to those in a previously-analyzed image by computing the minimum descriptor distance between each newly-found feature and every feature in the previous image. By matching features between successive images, the movement of each feature can be tracked.

With these benchmarks serving as baseline implementations, we aim to create custom accelerators that will allow them to perform in real-time with reduced energy consumption. In an effort to match data collection frequencies of current autonomous systems [77, 78], we have assumed input rates of 30 Hz for image data and 1 Hz for range-and-bearing sensor data. Our measurements indicate that speedups of at least $30\times$ for SIFT-based `features-matching`, $9\times$ for SURF-based `features-matching`, $3.6\times$ for KLT-IDSI-based `features-matching`, and $12.4\times$ for `kf-slam` are needed in order to meet real-time constraints.

2.2 Methodology

Given the area constraints of the FPGA and the limited bandwidth of the off-chip interconnect, our design goal is to find valid accelerators that achieve enough speedup to push application execution within the thresholds set by real-time data-collection frequencies. While our primary focus is on reducing execution time, saving energy also remains a significant goal; as such, we may choose to include accelerators that achieve little speedup so long as they reduce energy consumption.

The creation of effective accelerators begins with thorough application profiling to determine the sets of operations that consume the largest amounts of time (and energy). We measure program runtime statistics for our benchmarks using hardware counters accessed with the PAPI library [79]. Energy consumption is estimated using the McPAT framework [80]. Based on profiling results, we identify accelerator candidates (Section 2.2.1) and synthesize RTL designs using the AutoPilot tool-chain to determine latency and resource consumption (Section 2.2.2), allowing us to systematically select the optimal subset of candidates for our domain (Section 2.2.3). Finally, we implement the selected RTL designs for the target FPGA device using Xilinx ISE [81] map, place, and route tools,

followed by analysis of latency and energy consumption of the completed accelerators using Xilinx XPower Analyzer [82].

2.2.1 Accelerator Candidate Identification

In order to characterize accelerator candidates based on the profiling results, we introduce the terms functional speedup (FS), application speedup (AS), and domain speedup (DS). We define FS as the speedup of a particular function using an FPGA-based accelerator, AS as the speedup of the entire application due to one or more accelerators, and DS as the speedup of the entire domain of applications due to a selected set of accelerators. Hence, for an accelerator that transforms a function from T cycles to T' cycles, the functional speedup is simply expressed as:

$$FS = \frac{T}{T'}$$

Consider an application that takes a total of T_A cycles and has a set of m accelerators that achieve speedups of FS_1, \dots, FS_m for functions that originally take T_1, \dots, T_m cycles and are run R_1, \dots, R_m times, respectively; we then express the speedup of that application due to all the accelerators as follows:

$$AS = \frac{T_A}{(T_A - R_1T_1 - \dots - R_mT_m) + R_1\frac{T_1}{FS_1} + \dots + R_m\frac{T_m}{FS_m}}$$

For a domain of n applications having T_{A_1}, \dots, T_{A_n} total cycles and AS_1, \dots, AS_n application speedups, respectively, the overall domain speedup becomes:

$$DS = \frac{T_{A_1} + \dots + T_{A_n}}{\frac{T_{A_1}}{AS_1} + \dots + \frac{T_{A_n}}{AS_n}}$$

Unlike FS , AS and DS are sensitive to the run-counts of the code being replaced by the accelerator. Note that these speedup metrics also take into account CMP-FPGA communication overhead with respect to the proposed interconnect model (i.e. the T' values include cycles dedicated to communication).

Table 2.1: Cycles, energy, and maximum benefits from accelerator candidates

	Cycles (at 2 GHz)	Energy (J)	AS_{max}	AER_{max}	DS_{max}	DER_{max}
kf-slam (EKF-SLAM)	69,943,334,047	342.43	–	–	–	–
jacobiansPoseComposition	43,768,831,168	211.89	2.64	2.62	2.44	2.42
sphericalCoordinates	21,911,609,872	106.73	1.46	1.45	1.42	1.42
features-matching (SIFT)	2,399,480,515	12.12	–	–	–	–
SIFTExtremum	253,435,933	1.26	1.12	1.12	1.00	1.00
SIFTdesc	828,914,977	4.15	1.53	1.52	1.01	1.01
features-matching (SURF)	782,557,518	4.09	–	–	–	–
findMaximaInLayer	9,352,586	0.05	1.01	1.01	1.00	1.00
SURFdesc	378,550,056	1.99	1.94	1.95	1.01	1.01
features-matching (KLT-IDS)	478,421,485	2.44	–	–	–	–
cornerMinEigenvals	26,038,388	0.14	1.06	1.06	1.00	1.00
computeIDS	393,453,604	2.00	5.63	5.54	1.01	1.01

In accordance with Amdahl’s Law, a single accelerator α can achieve up to the following amount of application speedup (i.e. assuming $FS \rightarrow \infty$):

$$AS_{max} = \frac{T_A}{T_A - R_\alpha \cdot T_\alpha} = \frac{1}{1 - \frac{R_\alpha \cdot T_\alpha}{T_A}} = \frac{1}{1 - P}$$

where P is the proportion of the application ($0 \leq P \leq 1$) affected by that particular accelerator. Assuming that the $n-1$ other applications within the domain execute in T_{A_2}, \dots, T_{A_n} total numbers of cycles and in the worst case they cannot be accelerated by α , the maximum amount of domain speedup this accelerator can achieve is:

$$DS_{max} = \frac{T_A + T_{A_2} + \dots + T_{A_n}}{T_A \cdot (1 - P) + T_{A_2} + \dots + T_{A_n}}$$

The maximum application and domain energy reductions (i.e. AER_{max} and DER_{max}) are defined similarly.

Table 2.1 shows AS_{max} , AER_{max} , DS_{max} , and DER_{max} for each of the accelerator candidates in our benchmark suite. Note that as there is strong correlation between speedup and reduction in energy usage (e.g. AS_{max} is roughly equal to AER_{max}), energy is not explicitly considered for accelerator identification/selection. Descriptions of the accelerator candidates can be found in Section 2.2.2.

As Amdahl’s Law dictates, a candidate must make up a significant portion of an application in order to have a non-negligible impact on overall performance. However, large code regions, or *kernels*, lead to bloated accelerators that require excessive amounts of FPGA resources. The process of identifying viable candidates therefore involves balancing the tradeoff between AS_{max} and size of the code kernel.

We characterize an effective accelerator candidate as one with a relatively high ratio of AS_{max} to kernel size (i.e. resource consumption). At one extreme, the entire application is accelerated, meaning $AS_{max} = \infty$, which could lead to very high FPGA resource consumption (possibly over 100%). At the other extreme, multiple small portions of an application are accelerated, in which case overall FPGA resource consumption is lower and better controlled. However, AS_{max} is also considerably reduced.

Ideal candidates, we find, likely exist in the compute-intensive inner sections of large loops within a given application. In this case, not only is resource consumption low (i.e. the same accelerator is reused during each iteration), but overall acceleration due to functional speedup is found to far outweigh the overhead of repeated FPGA communication. This can be seen by the large speedups achieved with EKF-SLAM (refer to Figure 2.2), which uses accelerators of this ideal type. Consistent with our characterization of effective candidates, this methodology leads to accelerators with high AS_{max} values and small kernel sizes, resulting in high ratios of the two.

2.2.2 Accelerator Candidate Synthesis

We synthesize our accelerator candidates by manually extracting and translating portions of our applications’ C++ code into AutoPilot-friendly C. The AutoPilot tool then generates a register-transfer level (RTL) description of an algorithm from a C-based functional description. Automatic loop pipelining, unrolling, and/or merging can be applied if desired. AutoPilot supports most C constructs such as functions, arrays, and structures. Pointers are supported, but all pointer values must be statically determinable; this means all dynamic memory allocations must be removed. Moreover, we reimplement necessary mathematical functions, such as exponent and square-root

operations, in C as hardware implementations are not automatically generated. Aside from making these modifications, the functionality of the code is kept intact so as to maintain the fidelity of the original applications. We now detail the functionalities of the accelerator candidates.

2.2.2.1 `jacobiansPoseComposition`

A pose is defined as a 3D point representing location combined with a quaternion representing orientation. Given two poses x and u , where x is the robot’s pose and u is the pose of a sensor on the robot, this EKF-SLAM accelerator computes the pose composition $f(x, u) = x + u$. It then computes the two Jacobians $\frac{df}{dx}$ and $\frac{df}{du}$.

Since this accelerator contains no loops and consists primarily of floating-point arithmetic, it is easily parallelized by our C-to-RTL tools. In addition to being compute-intensive, this set of operations is repeated for each action-observation data-pair, making this an ideal accelerator.

2.2.2.2 `sphericalCoordinates`

Given a sensor pose and the absolute position of a landmark, this second EKF-SLAM accelerator computes the landmark’s position in spherical coordinates with respect to the sensor. Also computed are the derivatives of the spherical coordinates with respect to the landmark point and the sensor pose.

Like `jacobiansPoseComposition`, this code is purely computational with no loops and few conditional expressions. As these computations are also repeated for each data-pair throughout the application, this function has been identified as an ideal accelerator candidate.

2.2.2.3 `SIFTExtremum`

To accelerate SIFT-based feature matching, this accelerator consists of the core feature-detection algorithm. Given a point in the difference of Gaussians (DoG) pyramid, we check whether it is

the maximum or minimum of its 26 scale-space neighbors. If so, it is a feature candidate, and we iteratively interpolate it through scale-space to make the feature location more precise.

The extremum check is well-suited for a hardware implementation because all 26 neighbors can be checked in parallel. However, numerous memory accesses are required to continuously obtain the different sets of 26 neighbor values. Furthermore, the interpolation process is still roughly sequential since it is implemented as a loop with several input-dependent stopping conditions.

2.2.2.4 SIFTdesc

This accelerator computes the SIFT descriptor for a given feature point by constructing a histogram of gradients in an oriented patch around the feature location. As the computation is similar to that of SURFdesc and we did not ultimately use this accelerator, we do not describe it in further detail here. However, it is important to note that this candidate is the most complex in nature. Not only are patch sizes dependent on feature-scales, thereby requiring larger data sets to be read, but each point in the patch is linearly interpolated into the histogram, contributing to multiple bins and resulting in added computation.

2.2.2.5 findMaximaInLayer

To accelerate SURF feature detection, this accelerator consists of a critical part of the algorithm. It identifies potential features by finding maxima in a 2D array of determinants. Like SIFT, it also interpolates feature-points through scale-space to increase the accuracy of the feature location. Moreover, the process of finding maxima is also memory-intensive and the interpolation similarly exhibits sequential behavior.

2.2.2.6 SURFdesc

With this accelerator, the SURF descriptor is computed for a given feature point. First, it determines feature orientation by sliding a radial window around the feature point and computing

the distance-weighted sum of gradient angles in the window. The window orientation with the largest gradient response is saved as the feature orientation. Next, the image gradient is sampled in an oriented rectangular patch around the feature point. Finally, the descriptor is assembled as a histogram of gradients in 16 sub-rectangles within the patch.

This accelerator is the most complex of our selections. We use BRAM to store pre-computed sample points and weights for determining feature orientation. We also pipeline the many loops with our C-to-RTL tools.

2.2.2.7 cornerMinEigenvals

The main part of the KLT algorithm is contained in this accelerator. It first computes the gradient of pixel intensities throughout the source image. Using this gradient, it computes at each point the minimum eigenvalue of the structure tensor, which is a symmetric matrix of partial-derivatives of image intensity. Similar to SIFT and SURF feature detection, this algorithm applies relatively simple operations across a large image data set, resulting in memory-intensive rather than compute-intensive behavior.

2.2.2.8 computeIDSI

Given a feature location, this accelerator computes the intensity-domain spin image (IDSI) around the feature. An IDSI is a 2D histogram with the axes being intensity difference and distance from the feature point. Histogram values are exponentially weighted so that values close in intensity or distance are more prominent. The histogram is normalized to the range $[0, 1]$ to provide some invariance to illumination.

This accelerator is much more deterministic than `SURFdesc` since the patch size is statically known and there is no concept of feature orientation. Using AutoPilot, most loops in this implementation are pipelined, while a few simpler ones are unrolled.

2.2.3 Accelerator Selection

Once we synthesize the hardware for our accelerator candidates, we select the most suitable accelerators to bundle together on a single FPGA. Being constrained by the size of the FPGA, we are faced with limitations on the number of flip-flops (FFs) and lookup tables (LUTs), which are enclosed into fixed-size logic slices. There are also a limited number of digital signal processing slices (DSPs) and block RAM modules (BRAMs). In order to use these resources optimally, we analyze the domain speedup (DS) of each accelerator and compare this to its overall resource consumption.

We approach this problem of optimal candidate-set selection from a dynamic-programming perspective. Specifically, we reduce this to a multi-dimensional knapsack problem where we maximize cumulative DS while being restricted by the available FPGA resources. This method differs from the traditional knapsack problem in that three sets of weights must be tracked simultaneously – logic slices, DSPs, and BRAMs. In addition, a direct summation of DS values does not provide a correct representation of the combined speedup. Instead, the following formula must be used to represent the effective speedup of a group of n accelerators with DS values DS_1, \dots, DS_n :

$$DS_{effective} = \frac{1}{\sum_{i=1}^n \left(\frac{1}{DS_i}\right) - (n - 1)}$$

Despite this added complexity, approximate solutions are achievable in polynomial time [83]. Specifically, for n accelerators and 3 constraints, the time complexity of a solution with maximum error ϵ is roughly $O(n^k)$, where $k = \min(n, \lceil 3 \times (1 - \epsilon)/\epsilon \rceil)$.

We note that for the small set of candidates we present here, a brute-force selection algorithm would suffice. However, our dynamic-programming approach will scale well to larger sets of candidates. Using these methods, we select the following accelerators: `jacobiansPoseComposition`, `sphericalCoordinates`, `SIFTtextremum`, `SURFdesc`, and `computeIDSI`.

Table 2.2: Cycles, energy, and speedup statistics for implemented accelerator candidates

	Lines of C Code	FPGA Cycles	# Reads & Writes	# Runs	Total CPU Cycles	Total Energy (J)	FS	AS	DS
<code>jacobiansPoseComposition*</code>	186	77	70	13,994	21,550,760	0.02	2017.42	2.64	2.44
<code>sphericalCoordinates*</code>	203	104	40	12,750	26,520,000	0.03	826.23	1.46	1.42
<code>SIFTExtremum*</code>	196	10.6 †	42	1,183,935	251,040,500	0.27	1.01	1.00	1.00
<code>SIFTdesc</code>	255	121,323	60,788	1,095	2,656,973,700	2.88	0.31	0.57	0.98
<code>findMaximaInLayer</code>	222	6,061,448	8,182,740	8	969,831,680	1.05	0.01	0.45	0.99
<code>SURFdesc*</code>	304	13,716	5,452	682	187,086,240	0.20	2.02	1.32	1.00
<code>cornerMinEigenvals</code>	153	11,275,054	14,209,497	1	112,750,540	0.12	0.23	0.85	1.00
<code>computeIDSI*</code>	119	4,442	1,484	254	22,565,360	0.02	17.44	4.45	1.01

Note: All accelerators run at 100 MHz, except for `cornerMinEigenvals` which runs at 200 MHz.

* Selected accelerator. † Weighted average for two nested conditions; the outer condition executes every time the accelerator is called, but is true only about 0.2% of the time.

Table 2.3: FPGA resource utilization for implemented accelerator candidates

	BRAM	DSP	FF	LUT	SLICE
<code>jacobiansPoseComposition*</code>	0	128	9282	20781	6983
<code>sphericalCoordinates*</code>	0	304	20721	37140	11275
<code>SIFTExtremum*</code>	0	36	6513	13386	4142
<code>SIFTdesc</code>	0	38	10097	18274	5594
<code>findMaximaInLayer</code>	0	50	14191	25148	7318
<code>SURFdesc*</code>	35	139	19887	39783	11825
<code>cornerMinEigenvals</code>	0	36	11300	13528	3870
<code>computeIDSI*</code>	0	17	3580	6706	2087
Total Available Resources	832	768	301440	150720	37680
FPGA Utilization (* only)	4.21%	81.25%	19.90%	78.16%	96.37%

* Selected accelerator.

2.3 Evaluation Approach

Our system is evaluated using Red Hat 4.1.2 (x86-64) on a 2.0 GHz Intel Xeon processor with 8 GB of main memory. We implement our accelerators for the Xilinx Virtex 6 FPGA (resources summarized in Table 2.3). The benchmark applications are built from MRPT 0.9.3 and OpenCV 2.2, and the sample images used by `features-matching` are 640×480 pixels.

To model our system’s interconnect, we assume a single 128-bit time-multiplexed TSV bus. Since the FPGA clock is 20 times slower than the CMP, this effectively results in 20 individual TSV buses, where 4 are allocated to each of our 5 accelerators. For each effective TSV bus, we assume 1 FPGA cycle and $\frac{1}{2}CV^2$ Joules per access, where $C = 0.8$ femtofarads and $V = 1.2$ volts. Note that the energy consumed by the TSVs is found to be negligible. Additionally, we treat every load

and store on the FPGA (to external data) as a separate, single-cycle access. As we do not attempt any data packing over the buses, our results are conservative and we would expect to see further performance improvement with more efficient use of buses.

2.4 Experimental Results

Table 2.2 lists the cycles, energy, and speedup statistics for all the implemented accelerator candidates. We note that the number of lines of code roughly correlates with the complexity of an accelerator, but not necessarily its runtime. From these results, we see that the accelerators which were not selected (i.e. `SIFTdesc`, `findMaximaInLayer`, and `cornerMinEigenvals`) require the highest numbers of bus accesses. Although some of these accesses can occur in parallel (i.e. multiple buses could be in use simultaneously for a single accelerator), the resulting overhead remains large and significantly limits functional speedup. In comparison, for the accelerator with the highest FS (i.e. `jacobiansPoseComposition` with $FS = 2017$), the number of bus accesses (70) is very low. Since this is close to the total number of FPGA cycles (77), this verifies that the accelerator is made up of highly parallel computations which are almost fully hidden by the external load and store operations.

Table 2.3 presents the FPGA resource utilization corresponding to each implemented accelerator candidate. Although our selected accelerators consume almost the entire FPGA (96% slices), only `SURFdesc` uses BRAM. For the memory-intensive feature-detection algorithms, it will likely be useful to modify the original code to stream large portions of the input images to FPGA-local BRAM modules. In this way, the FPGA resources will be better utilized, while memory-access latency will also decrease.

Finally, Figure 2.2 summarizes performance improvements, including speedup, energy savings, and reduction in the energy-delay product (EDP) for each application as well as the overall domain. Notably, we achieve a $15\times$ speedup for EKF-SLAM and a $4.4\times$ speedup for KLT-IDS, resulting in enough performance improvement to meet our previously established real-time goals. We see

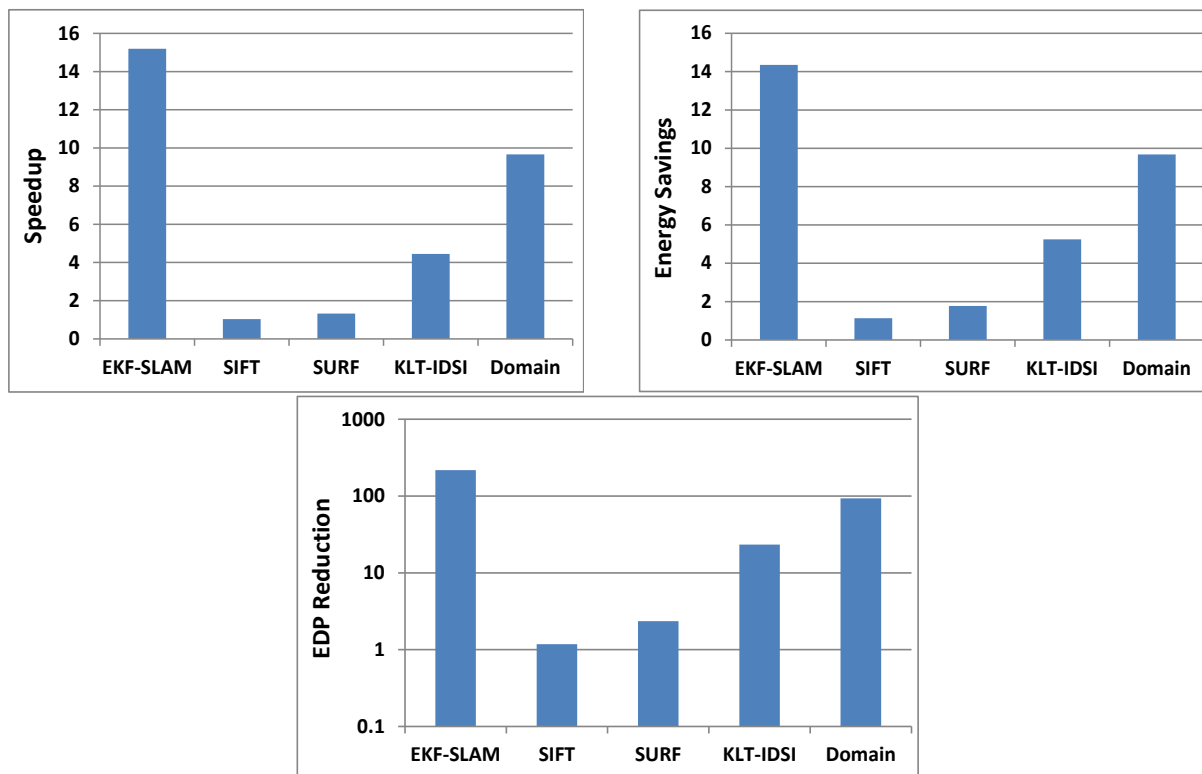


Figure 2.2: Performance and energy improvements using accelerators

similar improvements in terms of energy savings, ultimately resulting in a $94\times$ EDP reduction for the entire domain. We further point out that despite seeing virtually no speedup, the SIFT accelerator nevertheless provides enough energy savings to justify occupying the otherwise unused FPGA resources.

In a single application, one would not likely use more than one type of feature detection. However, with our results we are able to identify which algorithms for feature detection are more amenable to this type of acceleration, and hence should be preferred for the assumed architecture. We find that as feature detection is primarily a search algorithm over a large data set, implementations for the SIFT, SURF, and KLT detectors become highly memory-intensive computations that bear much interconnect communication overhead for marginal functional speedups. Consequently, our approach of partial acceleration using C-to-RTL tools turns out to be less effective than, for instance, a full custom FPGA implementation. EKF-SLAM and IDS, on the other hand, are compute-intensive, making our approach more successful.

As ideal accelerators are likely found in inner sections of critical loops and are run many times during an application’s execution, their relative performance may be highly dependent on communication overhead. For this reason, we evaluate the performance of our accelerators using additional interconnect models. This can also be viewed as a sensitivity study of varying communication bandwidths and latencies.

We model the total execution time of an accelerator with interconnect i using the latency of a memory access L_i and the interconnect’s raw 1-way bandwidth BW_i . Since our original measurements account for the latency and bandwidth of accesses over the TSV bus, we define latency penalty LP_i and bandwidth penalty BP_i to characterize the additional time taken over slower interconnects:

$$LP_i = L_i - L_{TSV} ; BP_i = \frac{BW_{TSV}}{BW_i} - 1$$

Given an accelerator that originally runs in N cycles of length T_{clk} and has M external memory accesses, our simplified model for the total execution time T_i using interconnect i is then:

$$T_i = LP_i + T_{clk} \cdot (N + M \cdot BP_i)$$

Note that when i is TSV, $T_i = T_{clk} \cdot N$.

Table 2.4: Interconnect model parameters

	L_i (ns)	BW_i (Gbps)	Description
TSV	0.5	256	128 bits · 2 GHz
QPI	1.25	102.4	20 bits · 64/80 ovhd. · 2 · 3.2 GHz
FSB	105	102.4	64 bits · 4 · 400 MHz
PCIe3	63	64	8 lanes · 1 bit · 8 GHz
PCIe2	126	32	8 lanes · 1 bit · 5 GHz · 4/5 ovhd.

We calculate values for FS , AS , and DS for the following interconnects: TSV bus, QuickPath Interconnect (QPI), front-side bus (FSB), PCI-Express 3.0 (PCIe3), and PCI-Express 2.0 (PCIe2). Table 2.4 lists the interconnect model parameters, while Figure 2.3 displays the corresponding speedup metrics. The latency over QPI is estimated as $2 \text{ flits} \times 2 \text{ cycles/flit} \times 3.2 \text{ GHz} = 1.25$

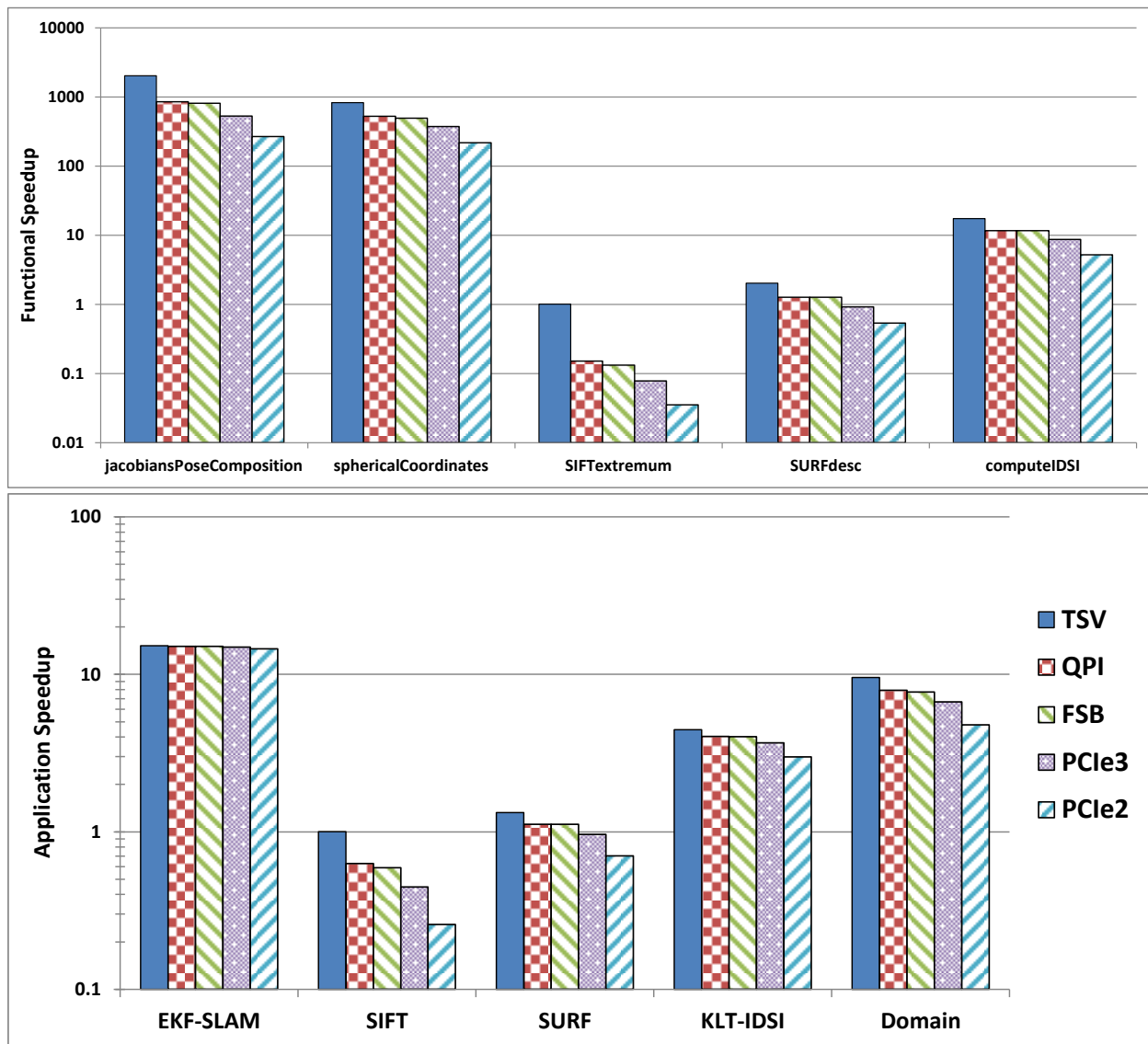


Figure 2.3: Functional, application, and domain speedups based on varying interconnect models

ns [84], the latency over FSB is estimated using manufacturer data [85], and the latency over PCIe is estimated by extrapolating results from [86].

As seen in Figure 2.3, because the FS values for the EKF-SLAM accelerators are high, the slower interconnects only slightly reduce AS (2% average loss). In contrast, AS for SIFT is drastically reduced (52% average loss) because its large number of initial simultaneous memory accesses make

it highly bandwidth-dependent. These impacts on accelerator speedups support the previous evaluations based on bus accesses for accelerators of compute-intensive versus memory-intensive applications. To mitigate increased communication overhead due to decreased interconnect bandwidth, overlap of computation and communication is needed, as done in [87]. However, the functional-speedup reductions seen for our compute-intensive accelerators are not significant enough to affect our accelerator choices.

Chapter 3

ARC: Architecture Support for Accelerator-Rich CMPs

In an accelerator-rich platform, major issues exist involving the low utilization and challenging usability of accelerators. To increase the utilization of accelerators and allow application developers to benefit from their potential performance efficiency, it is necessary to reduce the overhead involved in their usage, namely the overhead of operating system (OS) interaction for resource management. Another key issue in such accelerator-rich architectures is achieving efficient management when sharing accelerators among different cores and across different applications. Furthermore, it is important for the generation of unique accelerators to be automated so that an application author can produce code that is reusable across other accelerator-rich platforms as well.

We therefore provide support for accelerator-rich CMPs (*ARC*) [14] in the form of an efficient architectural framework and an associated set of algorithms that minimize the overhead of statically integrating and dynamically managing loosely-coupled accelerators (LCAs). This approach focuses on minimally invading the core, incurring low overhead by avoiding OS interaction for resource management, enabling flexibility in accelerator sharing and virtualization, and allowing user-friendly development for application programmers. Our work features the following: (1) an

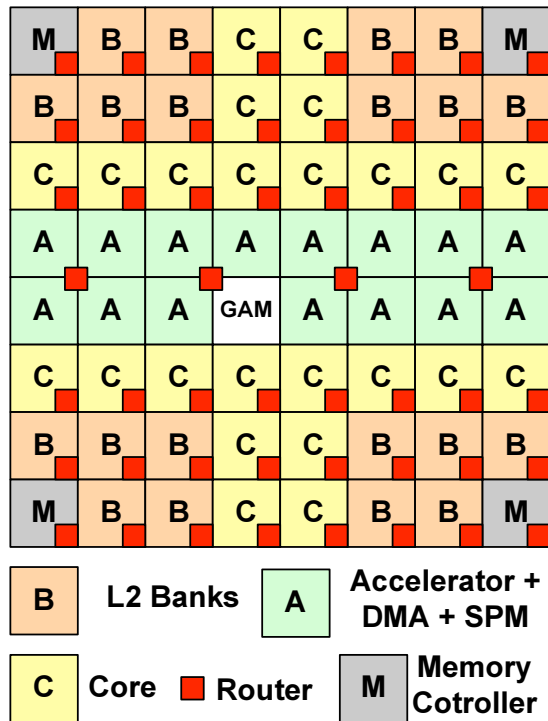
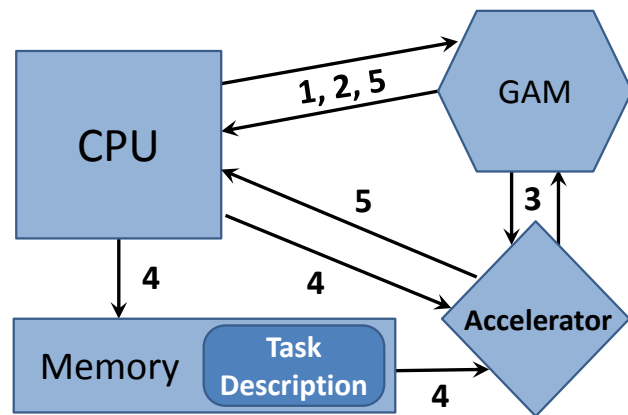
Figure 3.1: *ARC* microarchitecture

Figure 3.2: Communication between CPU, GAM, memory, and accelerator

accelerator allocation protocol to avoid OS overhead in scheduling tasks to shared LCAs, (2) an approach to accelerator composability that allows multiple LCAs to work collaboratively as a single complex virtual accelerator while maintaining transparency to program authors, and (3) a fully automated simulation tool-chain to support accelerator generation and management.

3.1 Overview of *ARC*

Figure 3.1 presents the microarchitecture of *ARC*, which is composed of cores, accelerators, the global accelerator manager (GAM), shared L2 cache banks, and memory controllers. All of the mentioned components are connected through the network on chip (NoC) using routers. Accelerator nodes include a dedicated DMA controller (DMAC), scratchpad memory (SPM) for local storage, and a small translation look-aside buffer (TLB) for virtual to physical address translation. The GAM is introduced to handle accelerator sharing and arbitration.

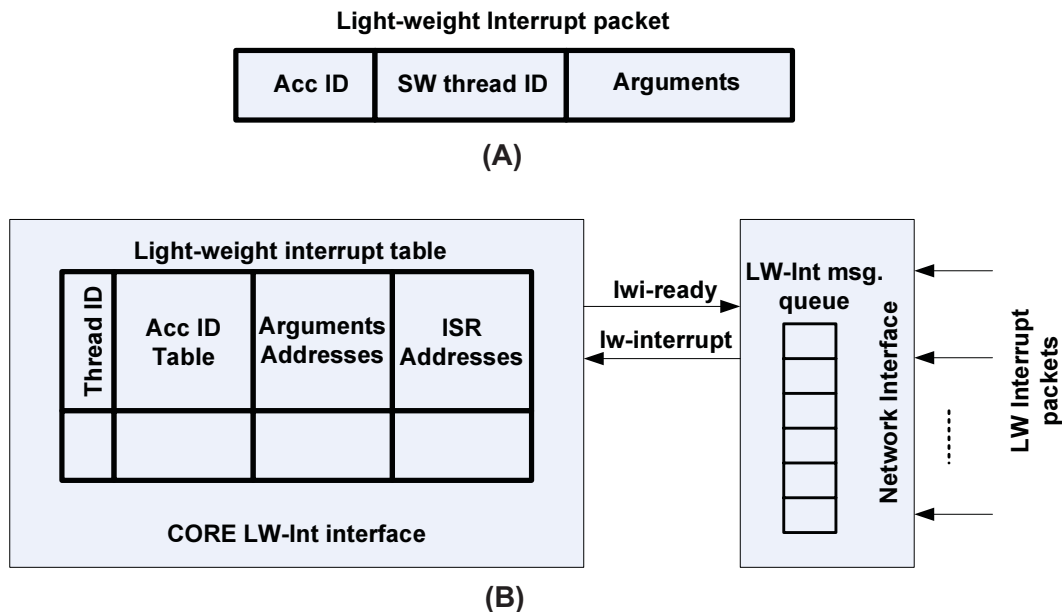


Figure 3.3: Light-weight interrupt support

3.1.1 Instruction Set Extension

In order to interact with accelerators more efficiently, we have introduced an extension to the instruction set, which consists of four new instructions used specifically for interacting with accelerators. These instructions are briefly described in Table 3.1. A processor uses *lacc-req* to request information about accelerator availability; the returned information consists of pairs of accelerator identifiers and predicted wait times for each available accelerator. A processor will then use *lacc-rsv* to request the accelerator corresponding to a specific identifier. Once an accelerator is reserved, *lacc-cmd* is then used for directly interacting with it. When a job is completed, *lacc-free* is used to release the accelerator so that it can be used by another processor. These instructions are accessible directly from user code, and do not require OS interaction. Communication with accelerators is carried out with the use of virtual addresses, maintaining accessibility of resources from user code. Execution of one of these instructions results in a message being sent to a device on the network (either the GAM or an accelerator). Attached to each message is the thread ID of the executing thread, which can be used to track requesting threads in an environment where context switches are possible.

Table 3.1: Instructions used to interact with accelerators

<i>lcacc-req x</i>	Request information from GAM about availability of accelerators implementing functionality <i>x</i> .
<i>lcacc-rsv x y</i>	Reserve the accelerator with ID <i>x</i> for a predicted duration <i>y</i> .
<i>lcacc-cmd accl cmd addr x y z</i>	Send a command (<i>cmd</i>) to an accelerator (<i>accl</i>) with parameters <i>x</i> , <i>y</i> , and <i>z</i> ; perform an address translation on <i>addr</i> , sending both logical and physical addresses.
<i>lcacc-free accl</i>	Send a message to GAM to release accelerator <i>accl</i> .

Table 3.2: Instructions to handle light-weight interrupts

<i>lwi-reg x y z</i>	Register service routine <i>y</i> to service interrupts arriving from accelerator <i>x</i> . LWI message packet will be written to <i>z</i>
<i>lwi-ret</i>	Return from an interrupt service routine.

Figure 3.2 shows the communication between a core, the GAM, an accelerator, and the shared memory. The numbers on the arrows indicate the steps taken when a core uses a single accelerator.

These steps are described as follows:

1. The core requests from the GAM an enumeration of all accelerators it may potentially need (*lcacc-req*). The GAM responds with a list of accelerator IDs and estimated wait times for the corresponding accelerators.
2. The core sends to the GAM a sequence of reservations for specific accelerators (*lcacc-rsv*). The core then waits for the GAM to give it permission to use these accelerators.
3. The GAM configures the reserved accelerators for use by the requesting core.
4. The core writes to the shared memory a task description detailing the computation to be performed. Next, the core sends to the accelerator a command identifying the memory address of the task description (*lcacc-cmd*). The accelerator loads this task description, and begins working.
5. When the accelerator finishes working, it notifies the core. The core then sends a message to the GAM freeing the accelerator (*lcacc-free*).

3.1.2 Light-Weight Interrupt Support

A platform that features accelerators requires a mechanism for a processor to be notified of the progress of an accelerator. In the *ARC* platform, we handle this issue with the use of light-weight interrupts. *ARC* light-weight interrupts are interrupts handled entirely as user code, and do not involve OS interaction, as this interaction can be a major source of inefficiency. Refer back to Table 1.1 in Section 1.1.1 for a more detailed analysis of the overhead incurred when interacting with accelerators through device drivers and OS interrupts.

There are three main sources of interrupts associated with accelerator interaction: (1) GAM responses, (2) TLB misses, and (3) notifications that the accelerators have finished working. GAM responses come either because a core sent a request message or a reserve message. TLB misses occur when an accelerator fails to perform address translation with the use of its own private TLB, and requires a core's assistance in performing the lookup. Interrupts notifying the completion of work arrive when an accelerator has completed all work given to it.

Figure 3.3 shows the microarchitectural components added to the cores in *ARC* in order to support the light-weight interrupt. An interrupt is sent via an interrupt packet (shown in Figure 3.3A) through the NoC to the core. Each interrupt packet includes the thread ID identifying the thread to which this interrupt belongs, and a set of interrupt-specific information. The main microarchitectural components added to support the light-weight interrupt are listed below:

1. Interrupt controller located at the core's network interface, which receives the interrupt packets and queues them until they can be serviced by the core.
2. Light-weight interrupt interface in the core, which is responsible for receiving the interrupts from the interrupt controller, and providing a software interface to setup the information needed for servicing the interrupts.

The interrupt controller has a queue for buffering the received interrupt packets, ensuring they do not get lost if the core is busy handling other interrupts. Without loss of generality, we assume that

for each thread we can only have a single-level nest for interrupts; this means multiple light-weight interrupts cannot be simultaneously serviced. If an interrupt arrives for a thread that is currently scheduled, it is executed immediately. If the thread is not scheduled, a normal OS-based interrupt occurs.

In order to support light-weight user-level interrupts, we introduce a set of instructions to enable user code to handle interrupts. These instructions are described in Table 3.2. The *lwi-reg* instruction registers the interrupt handlers, while the *lwi-ret* instruction returns from an interrupt-handler routine. Using these instructions, a program segment that utilizes accelerators is then designed as a series of interrupt service routines.

3.1.3 Accelerator Creation Methodology

Figure 3.4 is a block diagram highlighting the process of generating accelerators for a given application. Using a combination of static analysis and dynamic profiling, code kernels of the application are first assessed and a list of accelerator candidates are extracted. These candidates are then weighted using a series of selection criteria, such as area, performance, energy, criticality, and physical design constraints; this selection step generates an accelerator database. A *virtualizer* module then outputs a DLL that is used to link to executable files, as shown in Figure 3.5. By introducing a series of transformation rules, accelerator composition can also be carried out to create larger or smaller accelerators from the available accelerators on the platform.

3.1.4 Programming Interface to ARC

The **ARC** development flow, including the application programming interface (API) involved in using accelerators, is presented in Figure 3.5. For each type of accelerator, one dynamic linked library (DLL) is provided. This DLL is specific to a target platform, and provides a mapping from accelerator calls to actual invocations of physical accelerators. Calls to accelerators have their implementations dynamically linked to application code.

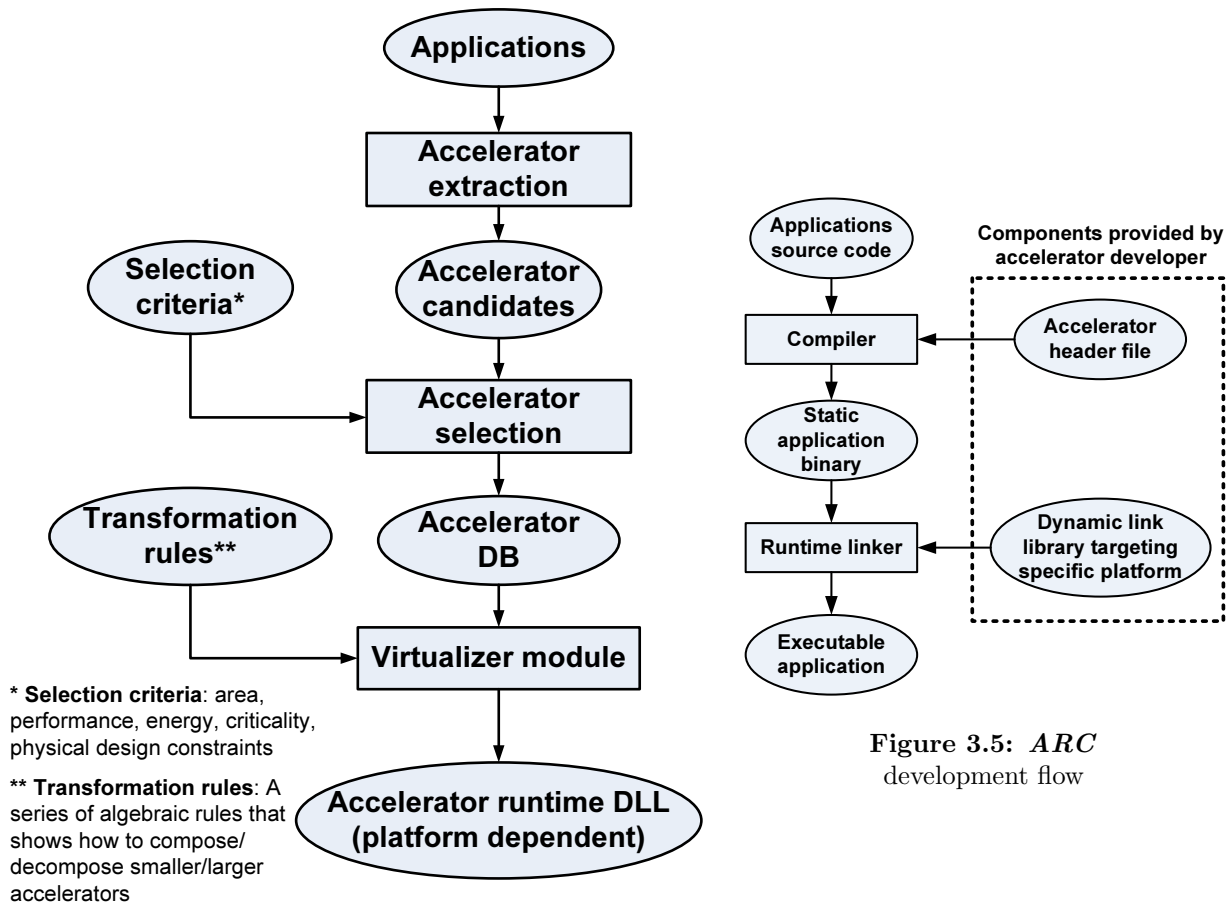
Figure 3.5: *ARC* development flow

Figure 3.4: Accelerator creation methodology

3.1.5 Invoking Accelerators

The initial overhead associated with acquiring permissions to use an accelerator is significant enough that it should be amortized over a large amount of work. We therefore assume an accelerator will be used to process a relatively large amount of data. To that end, we introduce two accelerator features to deal with this efficiently: (1) task descriptions to limit communication between accelerators and the controlling core, and (2) methods to handle TLB misses.

To communicate with an accelerator, a program would first write to a region of shared memory a description of the work to be performed. This description includes the location of arguments, data layout, which accelerators are involved in the computation, the computation to be performed, and

the order in which to perform necessary operations. This detail is included to allow accelerators to be general as well as allow coordination of accelerators in groups that perform more complex tasks (described in Section 3.1.7). Evaluating the task description yields a series of steps to be performed in order, with each step consisting of a set of memory transfers and computations that can be executed concurrently. This allows accelerators to overlap computation with memory transfer within a given step. When all computations and memory transfers in that step are completed, the accelerator moves onto the next step. We refer to these individual steps as *tasks*, and the structure detailing a sequence of tasks as a *task description*.

To further decouple the accelerator from the controlling core, each accelerator contains a small local TLB. This is required because the accelerator operates within the same virtual address space as the software thread that is using the accelerator. The accelerator relies on the controlling core to service any detected TLB misses. When a TLB miss occurs, the accelerator sends to the core a light-weight interrupt containing the address that caused the miss. The core then executes the same TLB miss handler that is executed when the core normally encounters a miss in its own TLB. Because this is an OS action and involves trapping to an OS handler, it is not necessary for the original software thread that is using the accelerator to be scheduled. If it is scheduled, the lightweight interrupt interface can be used to limit overhead associated with interrupt handling. Otherwise, the OS can be notified directly (e.g. by invoking a software interrupt or real hardware interrupt) without having to force a context switch to reschedule the controlling thread. The resolved address is then sent back to the accelerator that had encountered the TLB miss, where it is then stored in the local TLB.

3.1.6 Sharing Accelerators

When accelerators are shared among all the on-chip cores, it is possible for there to be several cores competing for the same accelerator. Even in architectures with large numbers of accelerators, there may be a limited number of a particular type of accelerator that is suddenly in high demand. In this situation, some of these cores may choose to eschew the use of the accelerator and simply execute

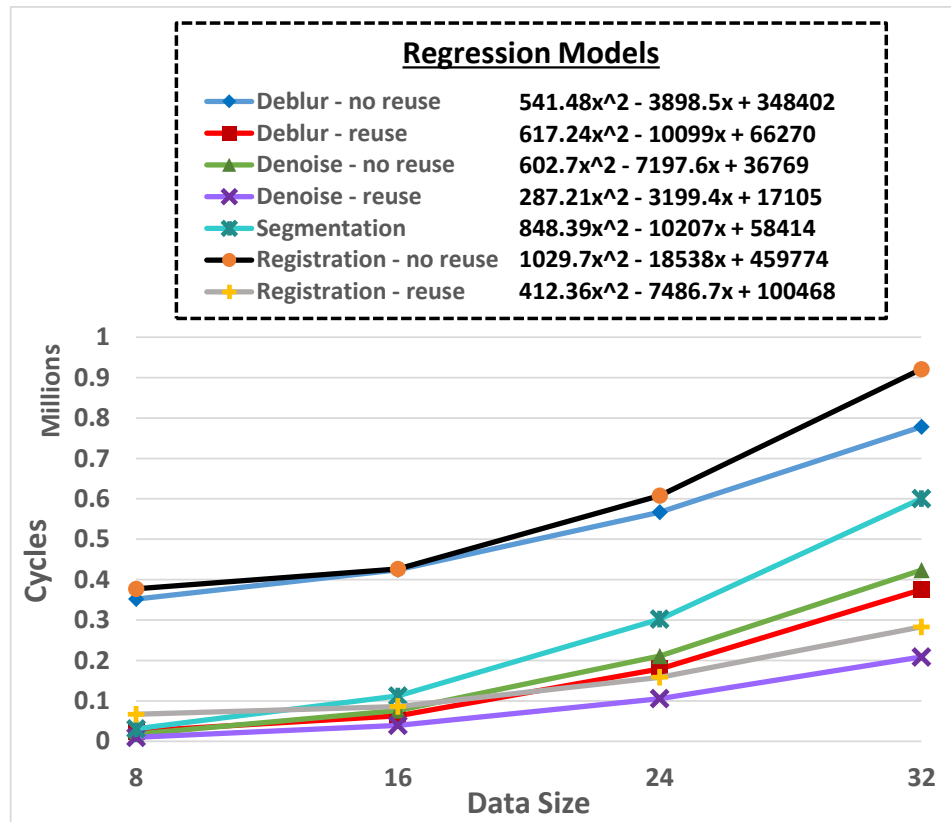


Figure 3.6: Regression models for medical imaging benchmarks

the task using their own core resources. While the core is certainly less power efficient in executing this task, it may make sense for it to do so in situations where the wait time for an accelerator will eliminate any potential gains. As such, we propose a sharing and management scheme that can use an estimated wait time to dynamically determine whether the core should wait to use an accelerator or instead choose a software path. This sharing and management strategy is performed by the GAM, which tracks the following: (1) the types of available accelerators; (2) the number of accelerators of each type; (3) the jobs currently running or waiting to run on accelerators, including their starting times and estimated execution times (Section 3.1.6.1); (4) the waiting list for each accelerator and the estimated runtime for each job in the waiting list (Section 3.1.6.2).

3.1.6.1 Accelerator Runtime Estimation (by the Core)

The execution time of a given job on an accelerator is data-dependent. When an accelerator is reserved, the requesting thread submits an estimation of the duration for which the accelerator will be used. This estimate is determined with the use of a data-size-parameterized regression model, which has been constructed based on profiled executions. We have empirically found that a simple second-order polynomial is sufficient to estimate execution time on average within 1%–2% of the actual execution time (at most 6%). Once a model is generated for an accelerator, it is provided to the rest of the development flow via the accelerator DLL (see Figure 3.5). Figure 3.6 presents some of the models we use in this work.

3.1.6.2 Wait Time Estimation (by the GAM)

After receiving a reservation message from the core, the GAM adds the requesting core’s ID to the tail of the waiting list for that accelerator. Note that the tasks being tracked in this waiting list are issued on a first-come-first-served (FCFS) basis. Hence, the estimated wait time for the task being added to the end of the list can be derived by summing up the expected execution times of all jobs that already exist in the waiting list for that accelerator. This estimation algorithm is both simple and practical for hardware implementation.

3.1.7 Accelerator Composition

A key contribution of our work is the increased utilization of the available resources by either chaining accelerators in a pipelined fashion or composing them in a more complex formation to virtualize larger accelerators. The next two subsections discuss these techniques.

3.1.7.1 Accelerator Chaining

In an accelerator-rich platform, it is common for the output of one accelerator to feed the input of another accelerator (e.g. when running streaming applications). For a traditional system, these two accelerators would communicate through system memory, meaning the controlling core would read the output of the first accelerator from its SPM, store that output in shared memory, and subsequently write the output to the second accelerator's SPM. To remove this inefficiency, we employ the DMACs of the two accelerator nodes; as these two controllers communicate, the source DMAC sends the content of its SPM to the destination DMAC to be written into its own SPM.

3.1.7.2 Accelerator Virtualization

For many types of problems, it is not practical to provide a monolithic accelerator to directly solve each instance of that problem. Additionally, it is not practical to demand an application author to target a single architecture. To decouple hardware design and software development, we provide a set of virtual accelerators. A *virtual accelerator* is an accelerator that is implemented as a series of calls to other physical accelerators, which are available in hardware (Figure 3.7A). A large library of virtual accelerators can be provided to the application author as if they were implemented in hardware. These accelerators would actually be implemented as a series of decomposition rules that break down a large problem into a number of smaller problems (Figure 3.7B), similar in style to the approach presented in [88]. The smaller problems would then be solved directly by hardware. Decomposition rules must describe two things: (1) computation that is performed by accelerators capable of solving sub-problem instances, and (2) communication of data to, from, and between the various smaller accelerators. Rules would be applied recursively to express an implementation of a virtual accelerator in terms of calls to physical accelerators.

These statically determined decomposition rules can be applied at runtime. Figure 3.8 describes the process of invoking a virtual accelerator from within the application binary. When an accelerator is called, an *lacc-req* message is sent to the GAM for wait times for all functional units that

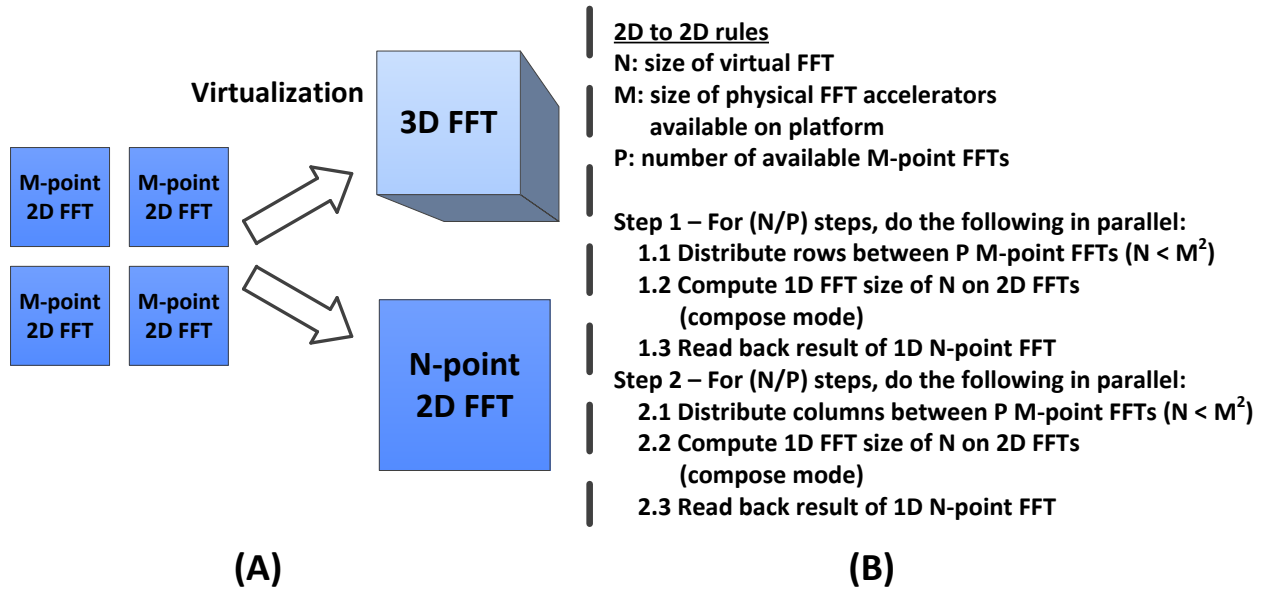


Figure 3.7: An example of accelerator composition

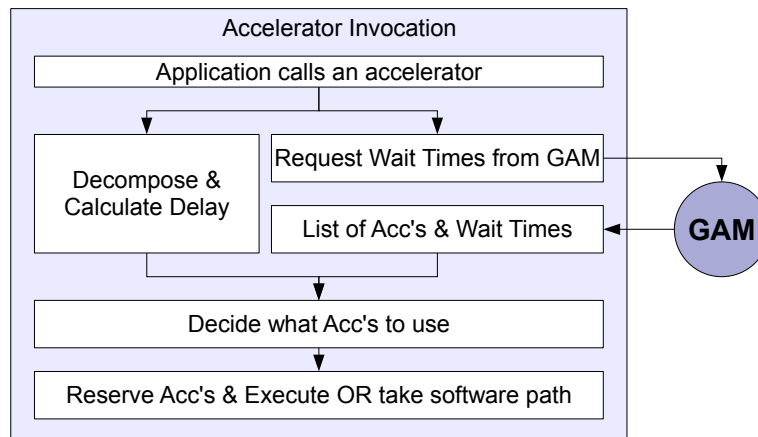


Figure 3.8: Accelerator composition steps

may be required by the decomposition result. While waiting on this request, the requesting core either begins calculating the decomposition or begins fetching the data structures associated with the statically computed solution. Once the GAM responds and the requesting core has a fully decomposed problem available, the core calculates the wait time for the entire computation. It does so by adding the runtime delay of the virtual accelerator (calculated with the use of a regression model) to the largest of the delays provided by GAM. The core then executes a series of *lacc-rsv* instructions (one for each required accelerator), specifying the estimated duration of use of

Table 3.3: Accelerated medical imaging benchmarks

Application	Algorithmic Functionality	# LCAs
Deblur [89]	Total variation minimization and deconvolution	4
Denoise [90]	Total variation minimization	3
Registration [91]	Linear algebra and optimizations	7
Segmentation [92]	Dense linear algebra, spectral methods, MapReduce	1

Table 3.4: Accelerated computer vision and navigation benchmarks

Application	Algorithmic Functionality	# LCAs
EKF-SLAM [66]	Partial derivative, covariance, and spherical coordinate computations	2
IDSI [71]	Computation of histograms based on intensities and distances of pixels	1
LPCIP [93]	Log-polar forward transformation of image patch surrounding each feature	1
SURF [69]	Feature orientation and computation of gradient histograms	1

each reserved accelerator as the wait time for the entire operation. GAM will not assign any accelerators until it can assign all accelerators requested. The core releases accelerators in the same way as it normally would. With these mechanisms, an application author can use a simple API to invoke virtual accelerators, and a hardware developer can implement accelerators based on need and availability of resources.

3.2 Evaluation Approach

3.2.1 Benchmarks

To illustrate the effectiveness of our *ARC* platform, we evaluate a number of compute-intensive benchmarks from both the medical imaging domain as well as the computer vision and navigation domain. Using shared LCAs, we accelerate four algorithms from each of these two domains.

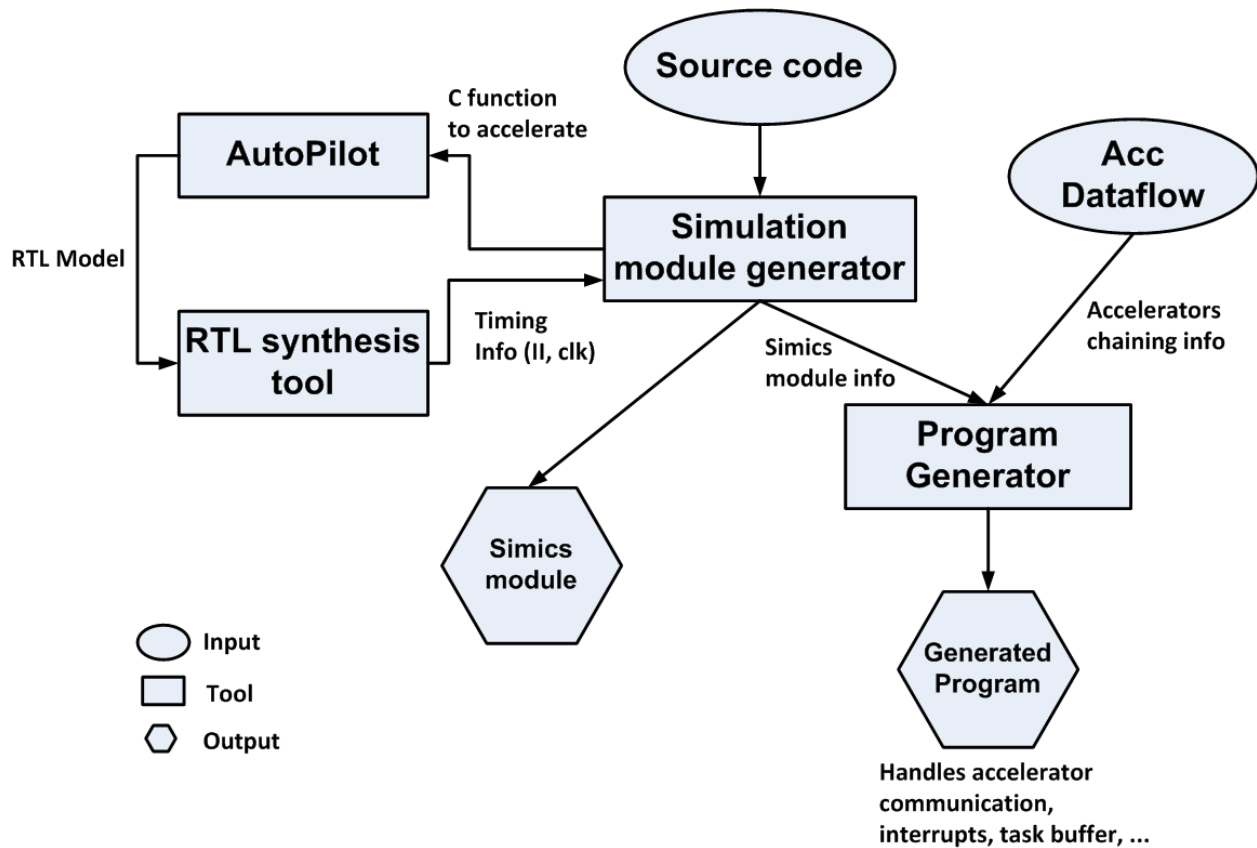


Figure 3.9: Process used to generate simulation structures and accelerated programs

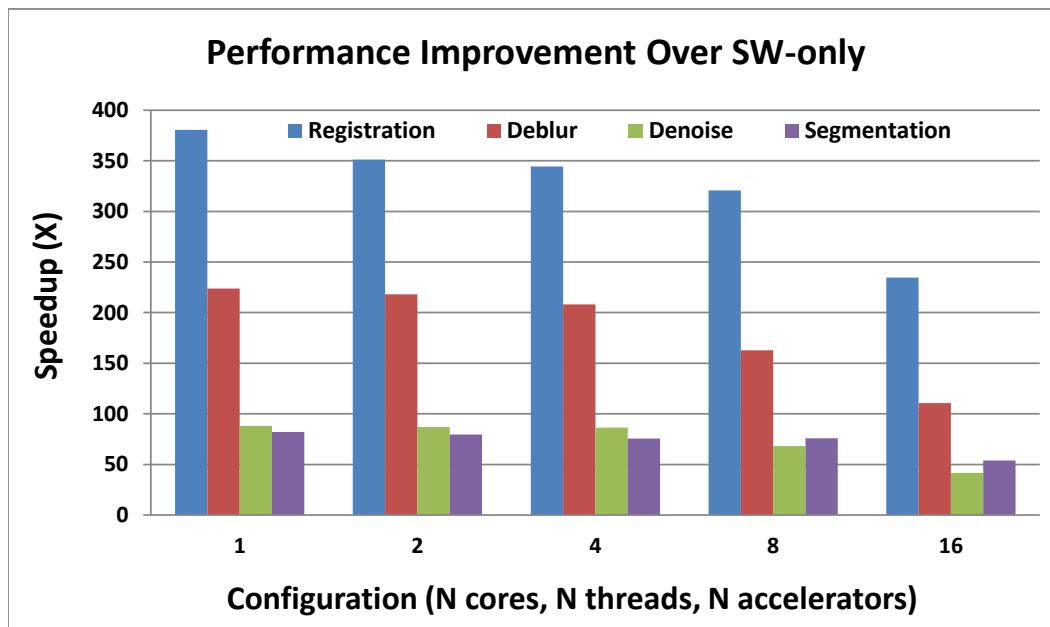
Table 3.5: Simics+GEMS configuration

Processor	UltraSPARC-III-i @ 2.0 GHz
Operating system	Solaris 10
Number of cores	1, 2, 4, 8, 16
Coherence protocol	MSI_MOSI_CMP_directory
L1 cache	32 KB, 4-way set-associative; latency: 1 cycle
L2 cache	8 MB, 8-way set-associative; latency: 10 cycles
Main memory	Latency: 1000 cycles; directory: 6 cycles
Network topology	4x8 mesh

Tables 3.3 and 3.4 provide brief descriptions of each application’s computational characteristics and include the number of accelerators used. Also, we note that the data of the medical imaging benchmarks is in a cubic form (e.g. a cube of $32 \times 32 \times 32$ data elements), whereas the data of the computer vision and navigation benchmarks is linear (e.g. 1024 data elements).

Table 3.6: Sample synthesis results

	DMAC	GAM	Deblur	Denoise	Registration	Segmentation
Clock (ns)	2	2	4	4	4	4
Area (μm^2)	10071	12270	4419917	1935539	12253775	2890354
Power (mW)	0.59	2.64	98.28	57.69	256.3	80.93

**Figure 3.10:** Speedup over *SW-only* (medical imaging domain)

The Fast Fourier Transform (FFT) is a computation common to a wide range of scientific computing and signal processing algorithms, including use in a number of our chosen medical imaging benchmarks. We use FFT to demonstrate our virtualization results. As a point of comparison, we use FFTW [94] v3.3 for our software implementations.

When analyzing contention between multiple threads executing the same benchmark, we insert a barrier immediately before entering the benchmark kernel being targeted for acceleration. This maximizes the observable effects of contention and models a worst-case scenario. All threads executing a benchmark can then be expected to enter the given kernel at approximately the same time.

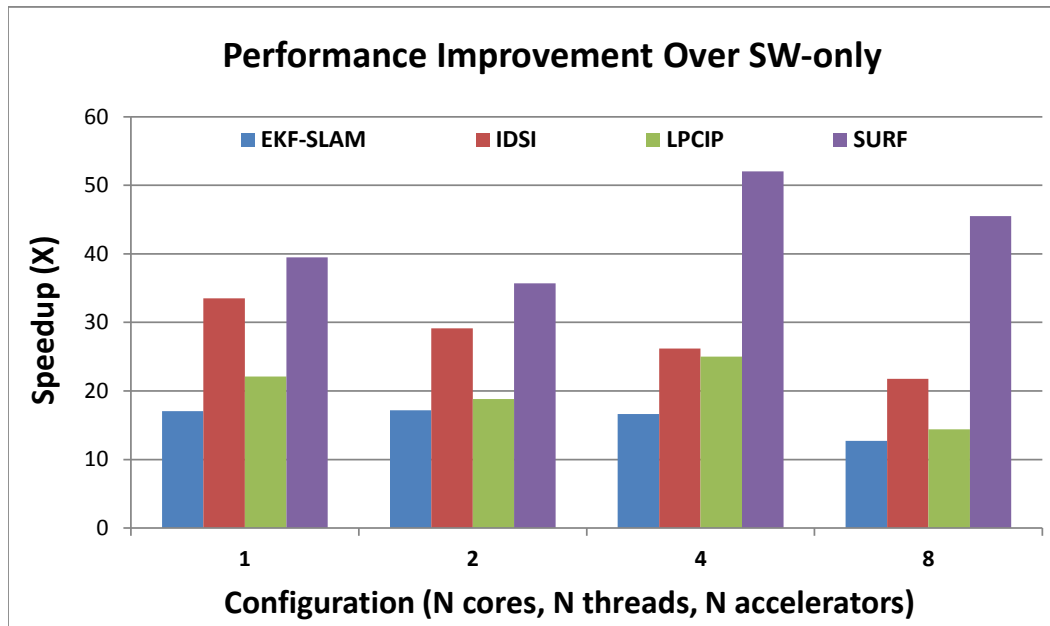


Figure 3.11: Speedup over *SW-only* (computer vision and navigation domain)

3.2.2 Tool-Chain

In order to make the exploration of this topic practical, a number of supporting tools have been created. These tools simplify the authoring of programs that use accelerators, and automate the process of implementing our chosen accelerators in our simulator framework. These tools are used in place of hand-written implementations and hand-adapted benchmarks to allow us to simulate systems that would have been prohibitively complex to manually author, such as those that utilize many accelerators or feature complicated inter-accelerator communication. Additionally, we believe our tool-chain is representative of what will be needed for the development of future accelerator-exploiting libraries, simplifying the job of programmers who would use these libraries without compromising any of the capabilities of these accelerators.

With this tool-chain, generation of accelerators is only a matter of identifying a function in an application's source code to accelerate. We have automated the process of extracting the function, compiling it as a VHDL-based hardware module, and synthesizing the module to extract timing and energy information. This process yields a module that plugs into our cycle-accurate simulation

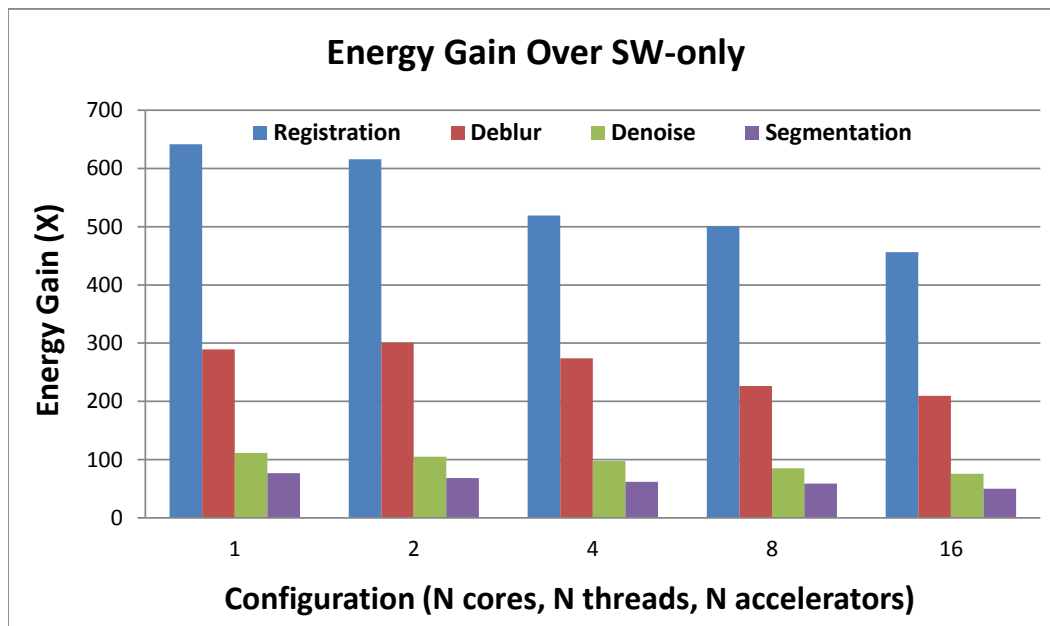


Figure 3.12: Energy gain over *SW-only* (medical imaging domain)

infrastructure to model the hardware unit, and coordinates the execution of the selected function in a pipelined fashion.

Once we select the functions we want to accelerate, typically encompassing the compute-intensive kernels of the benchmark, we procedurally generate a program segment to use these accelerators. We describe communication between accelerators in a simple data-flow language that we use to generate C source code. These program segments together make up the platform-specific DLL mentioned previously. This code is responsible for coordinating interactions between accelerators, registering/handling interrupts, managing task descriptions and accelerator resources, and dealing with accelerator-CPU synchronization. Figure 3.9 illustrates the work flow described here.

3.2.3 Simulation Platform

Our experiments are conducted using a heavily modified version of the Simics [49] and GEMS [50] simulation platform. The machine we model is based on a multicore system consisting of a mix of UltraSPARC-III-i processors and accelerators. In order to create a fair comparison between machines of different configurations, we maintain a fixed cache and network configuration. Our

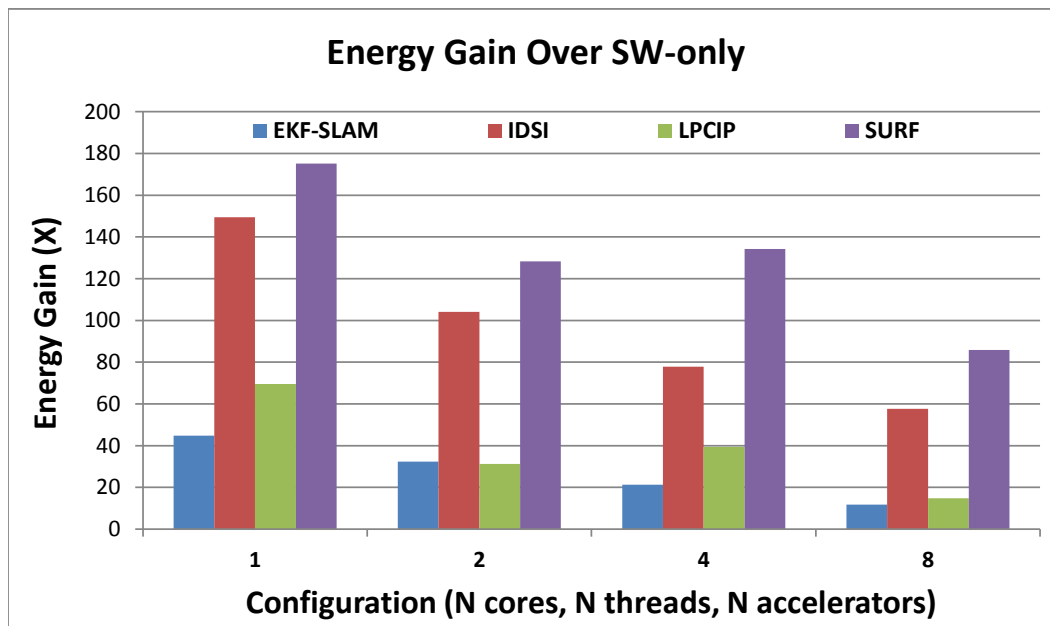


Figure 3.13: Energy gain over *SW-only* (computer vision and navigation domain)

network topology is a mesh modeled on a system normally used to support 32 processors. These nodes are then configured to either be processors, accelerators, or empty sockets. We feature a per-processor split L1 cache, and a distributed L2 spread across all nodes that rely on a directory-based coherence protocol. Table 3.5 shows the machine configurations we model in our simulations.

3.2.4 Area/Timing/Power Measurements

The AutoPilot behavioral synthesis tool [56] in combination with the Synopsys design compiler [95] are used to synthesize the C modules into ASIC. The timing information produced by the synthesis process is back-annotated to our accelerator modules to model cycle-accurate accelerators. For computing energy we use power reports from Synopsys for accelerators and *ARC* hardware components, and McPAT [80] for CPU power. As an example, Table 3.6 shows the synthesis results for a DMAC, the GAM, and accelerators for the medical imaging benchmarks.

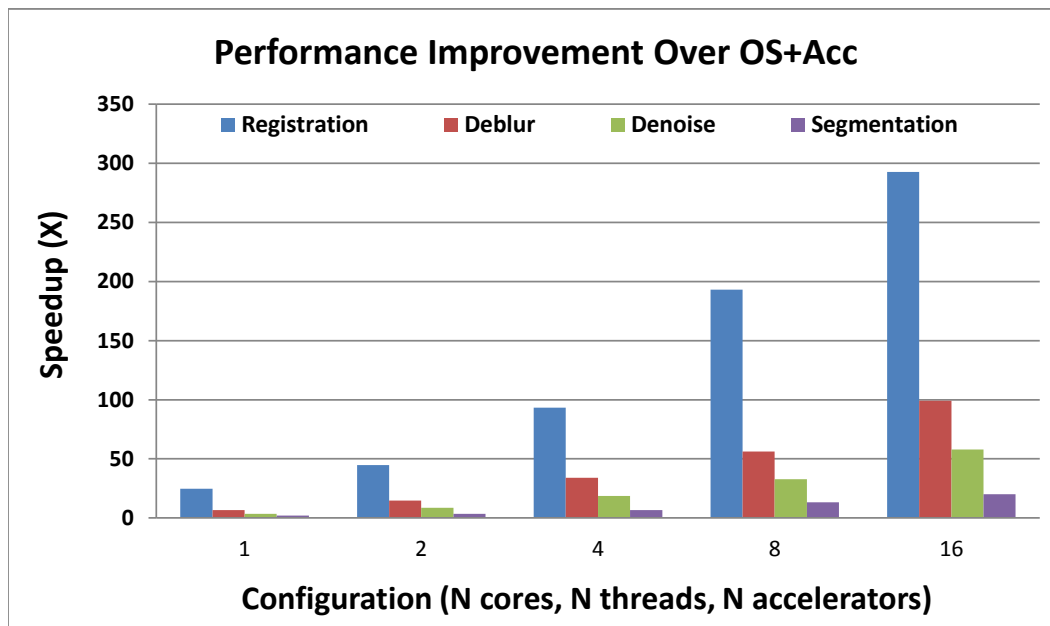


Figure 3.14: Speedup over *OS+Acc* (medical imaging domain)

3.2.5 Evaluation Schemes

The following schemes are used to evaluate our *ARC* platform:

- **Original benchmark (*SW-only*):** The baseline for the experiments is the execution of these multi-threaded benchmarks on a multiprocessor (one thread per processor).
- **Accelerators + OS management (*OS+Acc*):** This is a system which has accelerators managed by device drivers and OS interrupts.
- **Accelerators + HW management (*ARC*):** This system features all the enhancements discussed thus far, including resource arbitration managed by the hardware-based GAM.

We specify each simulation configuration using the Cc-Tt-Aa pattern, where “C” is the number of cores, “T” is the number of threads, and “A” is the number of replications of the accelerators needed by a benchmark. For example, a benchmark featuring 4 cores, 2 threads, and 1 instance of each accelerator would be described as 4c-2t-1a.

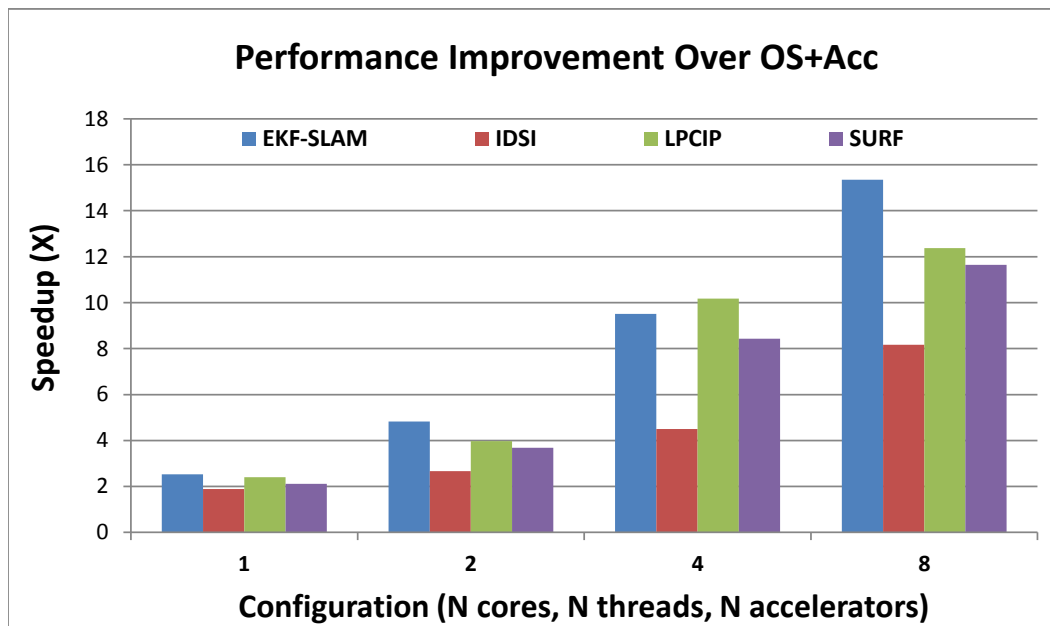


Figure 3.15: Speedup over *OS+Acc* (computer vision and navigation domain)

3.3 Experimental Results

3.3.1 Speedups and Energy Gains

Figures 3.10, 3.11, 3.12, and 3.13 show the speedup and energy gain results for the *ARC* base configuration (Nc-Nt-Na) compared to running the software-only version of the benchmark with the same number of processors and threads, and the same data size. The highest speedup is for **Registration** ($485\times$ for the 1c-1t-1a case) and the lowest is for **EKF-SLAM** ($13\times$ for 16c-16t-16a case). The best energy gain is $641\times$ improvement with **Registration**. On average, we achieve $241\times$ energy improvement over all the benchmarks and configurations. The computer vision and navigation benchmarks are shown to benefit relatively less from acceleration than the medical imaging benchmarks, yet this is mostly due to their smaller data sizes. As the data sizes increase, more computation can be streamed through the accelerators, resulting in more utilization and efficient execution.

We observe a reduction in speedup as we increase the number of cores and threads. This reduction is attributed to several sources. First, we measure the time from the start of all threads to

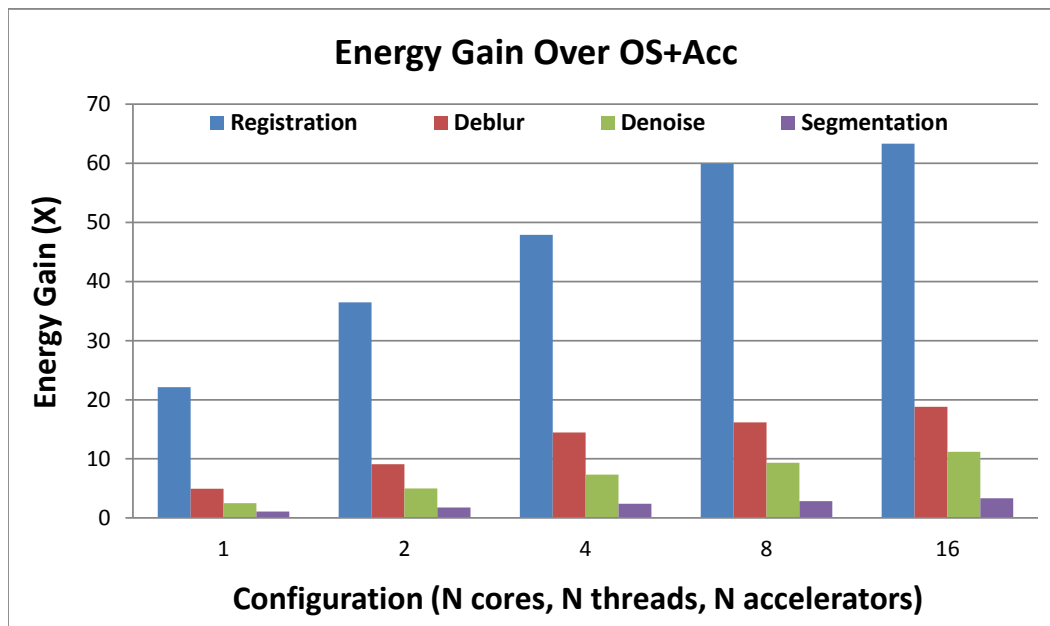


Figure 3.16: Energy gain over *OS+Acc* (medical imaging domain)

the end of the last thread, which means each result shown is the time measured for the longest running thread. Adding more threads increases the likelihood of observing normal fluctuations in runtime. Moreover, while we increase the number of cores and accelerators, we do not correspondingly increase network resources, memory bandwidth, or cache capacity. As a result, increasing the number of cores and threads results in additional contention for communication and memory resources. This impacts accelerated cases more significantly than software-only cases because, while the same amount of data is accessed, the accelerated cases access the data over a much shorter period of time.

Figures 3.14 and 3.15 show the performance gain *ARC* achieves compared to the *OS+Acc* scheme. Here we see increases in speedup for larger base configurations of *ARC* compared to the OS-managed system. The reasons for this are two-fold: (1) by increasing the number of threads and processors, the OS management overhead (thread context switching, TLB services, etc.) increases; (2) for larger configurations, the number of interrupts also increase, which makes our *ARC* system perform better due to the use of the light-weight interrupt in place of OS interrupts.

Similar to the performance results, Figures 3.16 and 3.17 show energy improvement of *ARC* over

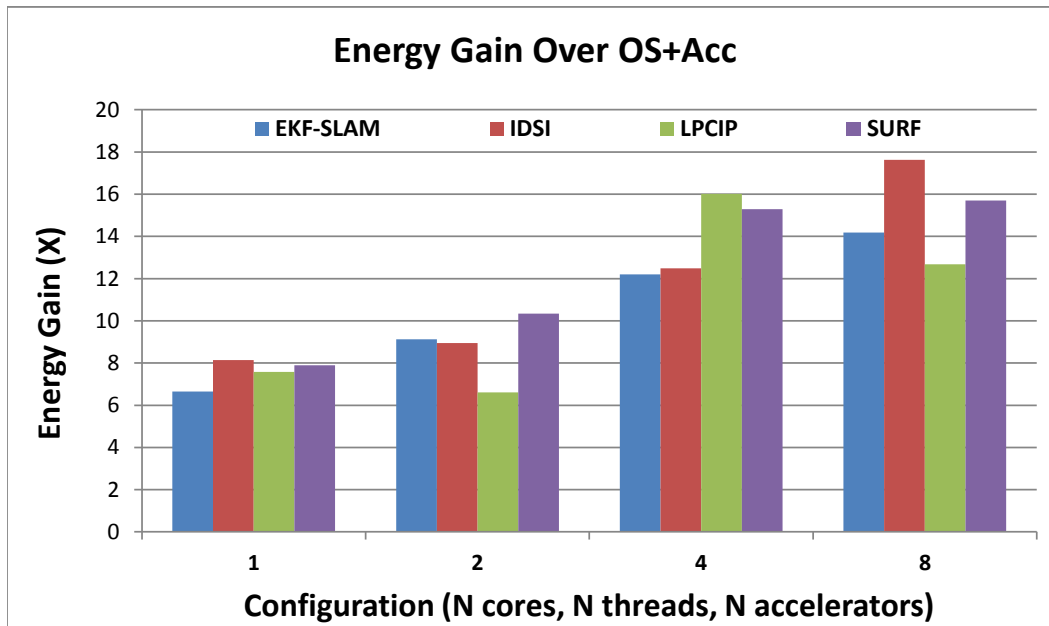


Figure 3.17: Energy gain over *OS+Acc* (computer vision and navigation domain)

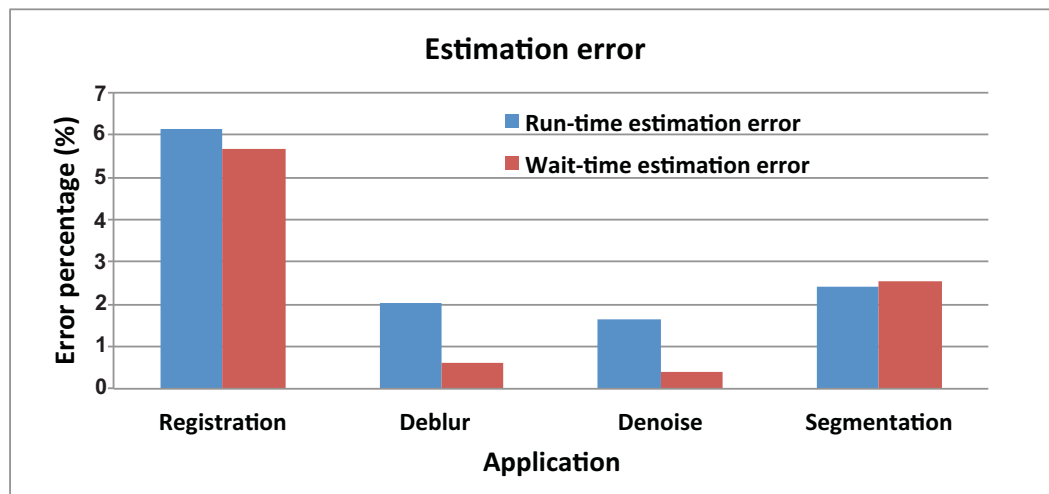


Figure 3.18: Error in accelerator runtime and wait time estimations

the *OS+Acc* configurations. As we enlarge the configurations, we see increased energy gains over the *OS+Acc* system. The **Registration** benchmark once again performs best with $63\times$ reduction in energy consumption. On average, the benchmarks run on the *ARC* scheme achieve $17\times$ energy gain over the *OS+Acc* scheme.

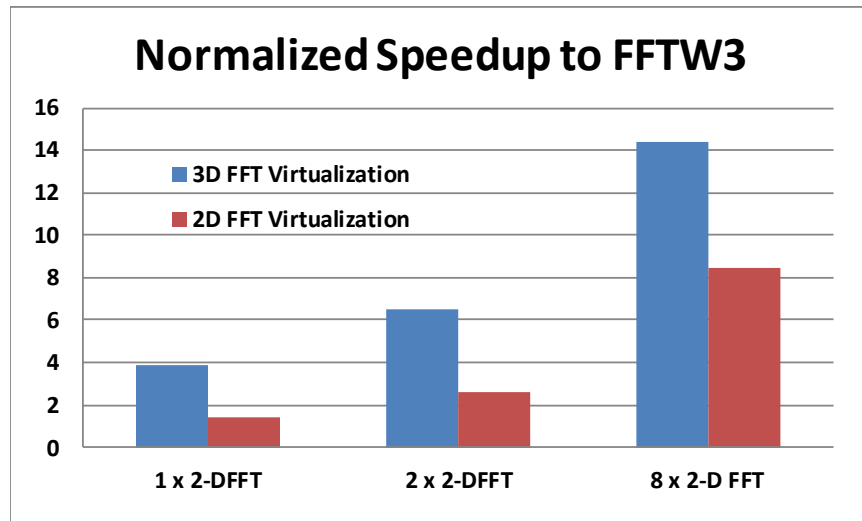


Figure 3.19: FFT virtualization (2D and 3D)

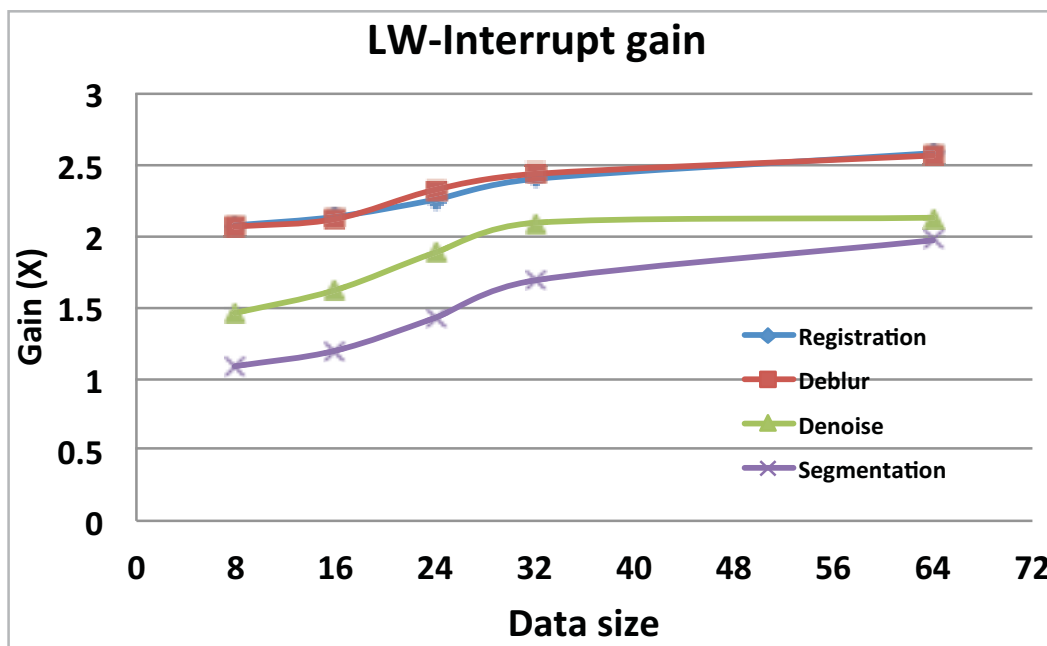


Figure 3.20: Benefit of using light-weight interrupts

3.3.2 Accelerator Sharing Results

Figure 3.18 shows the observed error for both the runtime and wait time estimations of the medical imaging benchmarks. We see that our estimated errors range from less than 1% to a maximum of

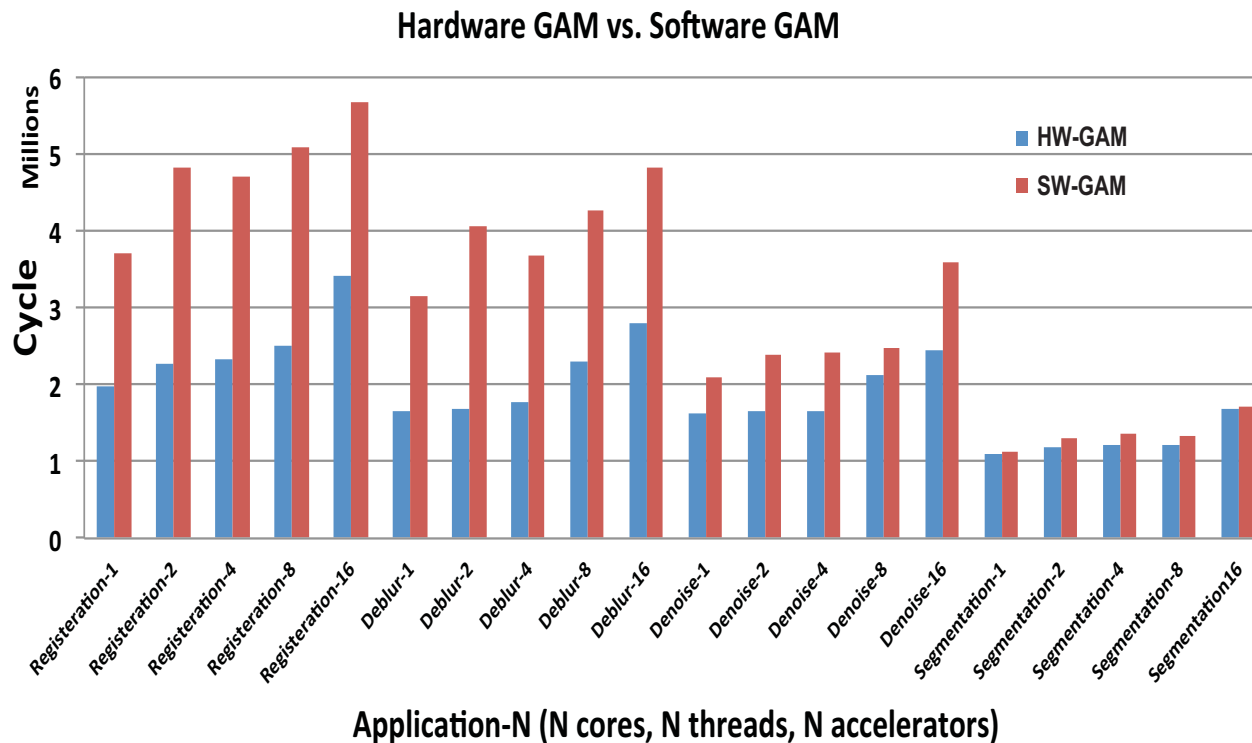


Figure 3.21: Benefit of using hardware GAM (HW-GAM) over software GAM (SW-GAM)

6%. As such, the execution times on our accelerators are sufficiently predictable for this to be a practical approach.

3.3.3 Accelerator Virtualization Results

Figure 3.19 shows the result of virtualizing a 512×512 2D FFT and a $128 \times 128 \times 8$ 3D FFT using multiple 128×128 2D FFTs. We compare the *SW-only* case to an *ARC* system with 1, 2, and 8 copies of the 128×128 2D FFT accelerator. For the *ARC* configuration, we assign a maximum of 5% of the chip area to FFT accelerators, while the *SW-only* case uses the general-purpose cores to run FFTW3 [94]. In the best case, we obtain $14.4\times$ speedup for the 3D FFT and $8.4\times$ speedup for the 2D FFT.

3.3.4 Benefits of Light-Weight Interrupt

To measure the benefits of the light-weight interrupts, we examine the performance of the medical imaging benchmarks and compare it to the performance on a system that relies instead on OS-based handling of interrupts. This comparison is shown in Figure 3.20, where **ARC** achieves up to $2.5\times$ speedup over an otherwise identical system that lacks light-weight interrupts. As the data size grows, more interrupts are generated, thereby increasing the benefits of employing light-weight interrupts.

3.3.5 Benefits of Hardware-Based GAM

Lastly, we examine the possibility of using an OS process, called the SW-GAM, to handle the responsibilities normally associated with the GAM. This approach differs from the previous *OS+Acc* scheme in that it avoids relying on accelerator drivers for communication; with SW-GAM, the interface for light-weight interrupts and the newly added instructions for accelerator communication are still included. To provide a fair opportunity for the SW-GAM to compete with the GAM implemented in dedicated hardware (HW-GAM), we allocate an extra processor exclusively for the SW-GAM. Figure 3.21 displays the comparison results for running the medical imaging benchmarks on configurations with N cores, N threads, and N accelerators, where N equals 1, 2, 4, 8, and 16. The most benefit is seen for **Registration**, with almost $2\times$ performance gain; since this benchmark utilizes more accelerators, the GAM is responsible for allocating more resources, allowing it make a larger impact on performance. On the other hand, **Segmentation** has only one accelerator, which diminishes the benefits of HW-GAM to as little as 10% speedup. As the configuration size increases, there is more interaction with the GAM, which translates into more benefits for the hardware-based GAM.

Chapter 4

CHARM: Composable Heterogeneous Accelerator-Rich Microprocessor

To address concerns regarding the low utilization of monolithic LCAs (refer back to Section 1.1.2), we explore more fine-grained accelerator composition. The composable heterogeneous accelerator-rich microprocessor (*CHARM*) [15] is a multicore design where loosely-coupled accelerator building blocks (ABBs) are distributed in islands around the chip and may be dynamically composed to virtualize a monolithic loosely-coupled accelerator (LCA). Thus, the ABBs are effectively the building blocks of our application-specific accelerators. Instead of having discrete LCAs for every desired accelerator function, the use of ABBs gives us tremendous flexibility in composing different accelerators on-the-fly, thereby dynamically matching application demand. In addition, we may better share resources at this fine granularity and improve ABB utilization. From a design perspective, the use of ABBs creates more regularity and static homogeneity in a design that still provides virtual heterogeneity through dynamic composition.

We first analyze LCA demands for the application domain of medical imaging. We then validate the system flexibility by targeting two independent application domains, namely navigation and

computer vision. For all our benchmarks, we examine the potential for composability and accelerator sharing. In terms of resource arbitration, we propose a hardware mechanism to control and compose ABBs that may be distributed anywhere in our system. This hardware provides virtualization of LCAs and allows software to interact with shared accelerators without being burdened by arbitration or contention management. Finally, we demonstrate how our approach provides load-balancing to better distribute work among ABBs, and improves utilization for either a single application with multiple accelerator tasks or across multiple applications.

4.1 Overview of *CHARM*

In this section, we address our design goals by way of an architecture that provides flexibility, scalability, and design reuse.

4.1.1 Software Infrastructure

The software component of *CHARM* is responsible for: (1) *LCA candidate selection* – identifying program hotspots that would benefit from LCA implementation [96]; (2) *ABB selection* – creating a set of ABBs to cover a set of LCAs under physical design constraints (area, timing, power) [97]; and (3) *data flow graph creation* – generating a mapping used to compose LCAs from ABBs. While the LCA candidate selection is done manually with the help of profiling tools, the processes of ABB selection and data flow graph creation are fully automated.

The structure of the data flow graph of ABBs is the same as that of a task flow graph [98] (see Figure 4.1). Each node is a task that is represented by a desired ABB invocation, with edges representing memory transfers between ABBs. In memory, this graph consists of a list of ABBs that would be used to compose the LCA, followed by an enumeration of memory transfers. Each ABB node consists of a type, an enumeration of starting addresses for locating argument streams in a virtually addressed private scratchpad memory (SPM) region, and settings for any local configuration registers. Each memory transfer consists of an identifier for a source and destination device

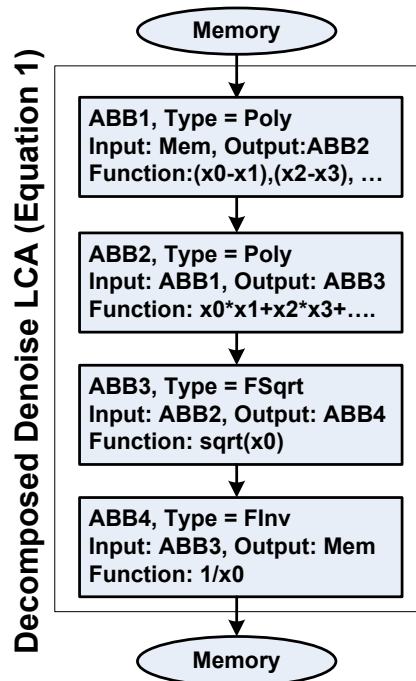


Figure 4.1: Data flow graph for Denoise LCA

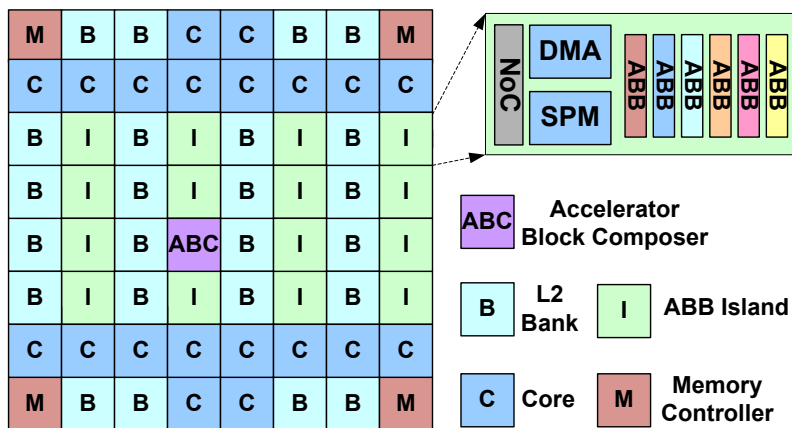


Figure 4.2: Microarchitecture of *CHARM*

(either memory or an ABB node). It also includes a starting address and a series of size-stride pairs describing a polyhedral (regular, high-dimensional) space for both the source and the destination. This graph is easy to parse, and consists mostly of values that are directly usable by various control registers on the ABBs and associated DMAs. We further note that when a data flow graph is created, it is not tied to any physical instance of the ABBs. The graph simply connects virtual ABBs together as a template of an LCA. Figure 4.1 illustrates this for one of the LCAs used in *Denoise*, whose functionality is formulated by the following equation:

$$1/\sqrt{\sum_{i=0}^6 (x_c - x_i)^2} \quad (4.1)$$

Once this LCA template is generated, the hardware would be responsible for mapping to it physical instances of ABBs to instantiate a virtual LCA on-the-fly.

4.1.2 Hardware Infrastructure

While the software is responsible for specifying candidates for acceleration and detailing how ABBs may construct particular LCAs, it is the hardware’s responsibility to allocate ABB resources to particular threads to satisfy software demand. For this work, we will restrict each ABB to be allocated to at most one LCA at a time. Our hardware will arbitrate usage of the ABBs among multiple competing threads/cores, and allocate available resources in a way that maximizes their utilization (i.e. load-balances requests from one or more cores among multiple accelerators). Note that additional complications exist due to varying contention for the use of any given ABB along with variation in latency when data is streamed to accelerators (e.g. caused by TLB/cache misses and congestion on the NoC). A dynamic solution is preferable in order to adapt to nondeterminism in memory latency and to the varying accelerator demand across different cores.

Figure 4.2 presents an implementation of the *CHARM* microarchitecture. It consists of cores, L2 cache banks, memory controllers, ABB islands, and an *accelerator block composer* (ABC), which is the means of control for composing ABBs and the mechanism by which we provide dynamic adaptation. We describe the ABC in more detail in Section 4.1.2.1.

Each ABB island has a small dedicated SPM, a dedicated DMA engine, and an NoC interface. The SPM allows ABBs, when composed into an LCA, to have a fixed data access latency. By using memory streaming and task partitioning, and by overlapping communication with computation, the SPM size can be kept small. The allocation of SPM regions to each ABB is handled by the ABC.

The dedicated DMA engine in each ABB island is responsible for transferring data between the SPM and the L2 cache, and also between SPMs in different ABB islands (i.e. accelerator chaining or remote DMA [99]). In addition, each DMA has a small internal TLB, allowing accelerators to work with virtual addresses. In the event of a TLB miss, the DMA will forward its request to the ABC.

4.1.2.1 ABC Design

In our scheme, the ABC is contacted by cores that need access to a virtual LCA. It then allocates ABBs to satisfy this request. A virtual LCA can consist of any number of ABBs, provided that number is less than the number of ABBs that is available in the system. The ABC uses five components to manage its collection of ABBs: a *resource table*, a *composed LCA table*, a collection of *task lists*, a *TLB*, and a *data flow graph interpreter*.

Resource Table: The ABC has a resource table that it uses to track the allocation of different ABBs to LCAs. When a core requests the use of an ABB, the resource table is queried to determine which ABBs are available. If enough ABB resources are available, multiple instances of a particular type of LCA may be instantiated, assuming the computation to be done is large enough for these multiple instances to each perform non-trivial amounts of work. The ABC uses a two-tiered allocation policy to decide which ABBs to compose into a given LCA. First, the ABC will attempt to balance the concentration of memory-accessing ABBs across the entire system. The purpose of this is to limit contention in the DMA associated with each node. Second, the ABC will employ a simple greedy approach to select ABBs that are local to other ABBs they communicate with. This is done in order to minimize the cost of communication between ABBs. To further reduce latency, ABBs within the same island may use a common SPM for communication (rather than each using their own SPM in their respective islands) and eliminate the need to communicate through the NoC. When ABB resources are scarce, the above metrics degrade to greedily constructing LCAs out of any available ABBs, rather than waiting for more optimal choices to become available.

Composed LCA Table: To eliminate the need to repeatedly compose the same LCA out of the same ABBs when tasks are completed, a composed LCA table is introduced. This table tracks ABB allocation, and is used to remove the overhead of remapping patterns when an LCA is already composed.

Task Lists: When the ABC receives a request for an LCA, the requested computation is split into a number of fixed-size chunks of data to enable efficient parallelism. Each of these chunks is

referred to as a task and the ABC maintains these in a task list. Each entry in the task list consists of a marker identifying which LCA the task belongs to, which task of the whole computation the entry belongs to (for that specific LCA invocation), and a bit flag marking it as runnable or not runnable. As tasks are added to the task list, the ABC iterates over the memory addressed by the task, and checks its local TLB. If all addresses in a task are resolvable by the internal TLB, the task is marked as runnable. Otherwise, it is marked as not runnable, and the ABC issues a TLB miss to the requesting core. The ABC uses a round-robin scheduling policy to iterate through all LCAs that have at least one task marked as runnable. So long as there are runnable tasks for which enough ABBs are available, the ABC continues attempting to compose more LCAs, and continues issuing tasks.

TLB: The ABC maintains a shared TLB that caches address translations among all tasks in its task list. This allows the ABC to prescreen tasks for TLB misses prior to composition. If multiple ABBs under control of the ABC would have encountered the same TLB miss, the ABC can avoid sending duplicate requests to the corresponding core and simply satisfy these misses locally with its own TLB.

Data Flow Graph Interpreter: Our software framework provides composition instructions in the form of a data flow graph. These graphs are fed as resource instantiation templates from the cores to the ABC. Each node in the data flow graph needs to be allocated to a particular ABB; at any given time, each ABB is only assigned to a single graph node and a single LCA. When an ABB finishes with the work for a single task, it notifies the ABC that it is free for reassignment. If there are more tasks marked as runnable associated with the LCA to which the ABB was allocated, it is given another task from that set. If there are no runnable tasks associated with that LCA, the ABB becomes eligible for composition into a different LCA. We considered keeping LCAs composed for a longer duration to exploit potential locality of use of a particular LCA, but found that the overhead involved in mapping a set of ABBs to an LCA template is small enough such that releasing resources immediately is preferable due to the improved utilization of ABBs across multiple LCAs. This means that ABB utilization varies over the course of execution of a particular task, and it may be possible for there to be multiple constructed copies of a particular LCA at a given time, even if

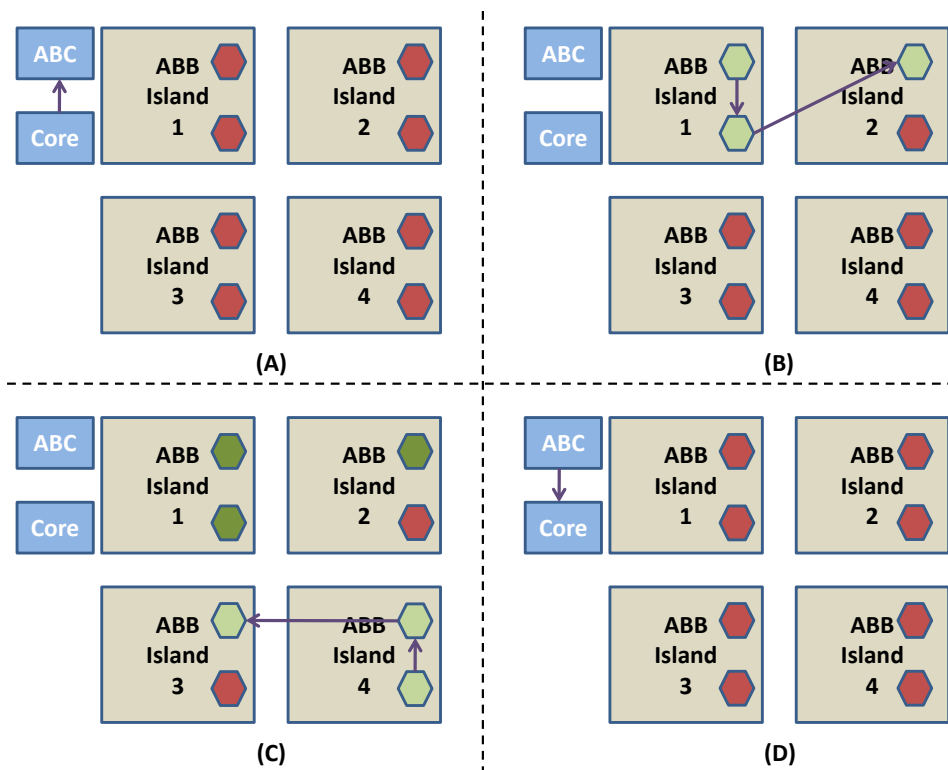


Figure 4.3: LCA composition example: (A) a core sends a request for an LCA to the ABC; (B) an LCA instance is allocated; (C) another LCA instance is allocated with consideration for balancing DMA utilization; (D) the ABC signals completion to the core

this is not possible when a given core initially requests an LCA. Therefore, as long as the ABC has runnable tasks in the task lists for a particular LCA, we allow it to attempt to compose additional copies of that LCA. In this way, the ABC can eventually make use of all available resources. Note that we do not allow ABB preemption, except in the event of an error, such as an access violation in the requesting core.

4.1.2.2 Example of Composition

Figure 4.3 shows an example of LCA composition for an architecture with 4 ABB islands. This sample architecture has eight ABBs (shown as hexagons), with two of them in each ABB island. For simplicity, we assume that all ABBs in this example are of the same type. The ABC and a requesting core are shown in the upper-left corner of each quadrant of the figure. In this example,

the core requests the composition of an LCA consisting of three ABBs in sequence, with the first ABB reading from memory and the last ABB writing to memory. The core first sends a data flow graph of the desired LCA to the ABC (Figure 4.3A). The ABC then interprets the data flow graph, splits the request into tasks, and begins cycling over the addresses each computation will access. It puts each of these chunks in the task list. For this example, we assume there is more than one task associated with this LCA invocation, and that the ABC's local TLB has the required pages to make all tasks immediately runnable. The ABC then examines the availability of ABBs, discovering that they are all free, and begins allocating.

Since at least one task is made runnable, the ABC proceeds to execute the allocation algorithm described in Section 4.1.2.1. After finding a match, consisting of two ABBs in Island 1 and a single ABB in Island 2 (Figure 4.3B), the ABC makes an entry in its composed LCA table, marking these ABBs as belonging to this specific LCA. At this point, it chooses a runnable task from the task list belonging to this LCA type, and dispatches a task. The ABC then begins attempting to map another instance of the requested LCA to the remaining available ABBs, and finds two ABBs in Island 4 and one in either Island 2 or Island 3. The allocation algorithm chooses to use Island 3 instead of Island 2 for obtaining the last ABB in order to distribute memory-access load across more DMAs (Figure 4.3C). The LCA instantiation process is then stopped since there are not enough ABBs to construct any additional LCAs. As ABBs complete their assigned work, they signal to the ABC that they are finished. Each time the first ABB in an LCA signals the ABC of completion, the ABC checks its task list for runnable tasks. If it finds a task, it begins sending a new task to each ABB in that composed LCA. If it does not find a task, it marks this LCA as retiring, and marks the associated ABBs as free. Each time an ABB that was part of a retiring LCA is marked as free, it is made available to be recomposed into a new LCA. When all ABBs of all clones of a retired LCA are freed in this manner, an interrupt is sent to the requesting core marking the completion of the requested computation (Figure 4.3D).

Table 4.1: Simulation parameters

Parameter	Value
Processor	UltraSPARC-III-i @ 2.0 GHz
Operating system	Solaris 10
L1	32 KB, 4-way set-associative; latency: 1 cycle
L2	8 MB, 8-way set-associative; latency: 10 cycles
Coherence protocol	Shared banked L2 cache; L2: MOSI; L1:MSI
Main memory	Latency: 1000 cycles; directory: 6 cycles
Network topology	4x8 mesh; link latency: 1 cycle; router latency: 5 cycles

4.2 Evaluation Approach

4.2.1 Simulation Platform

In order to conduct our experiments, we have modified Simics [49] and GEMS [50] to model accelerator-rich many-core architectures. Table 4.1 shows the parameters we use in our simulations. We have also implemented a series of supporting tools to automatically generate accelerators, as well as application code that makes use of these accelerators. When calculating energy, we use the power results outputted from the Synopsys Design Compiler (32nm SAED library) [95] for the LCAs and ABBs; for cores and caches, we generate power values using McPat [80] and CACTI 5.3 [100]. Tables 4.2 and 4.3 provide the area and power overhead for the selected ABBs and LCAs corresponding to each benchmark. We have also included the synthesis results for the ABC that implements the ABB allocation algorithm mentioned in Section 4.1.

For studying the overhead of ABBs, we have synthesized the Poly ABB with 16 inputs/outputs (i.e. Poly16), the results of which are shown in Table 4.2. To provide more detail on the functionality of a Poly ABB, Figure 4.4 displays the internal structure of one with 8 inputs/outputs (i.e. Poly8). It consists of adder/subtractor/multiplier (ASM) modules, an SPM bank, and logic for controlling access to the SPM. The SPM bank has 3 sub-banks (for simultaneous read/compute/write), each with one read/write port. One sub-bank is connected to the ASMs and two are ported to the DMA controller (DMAC).

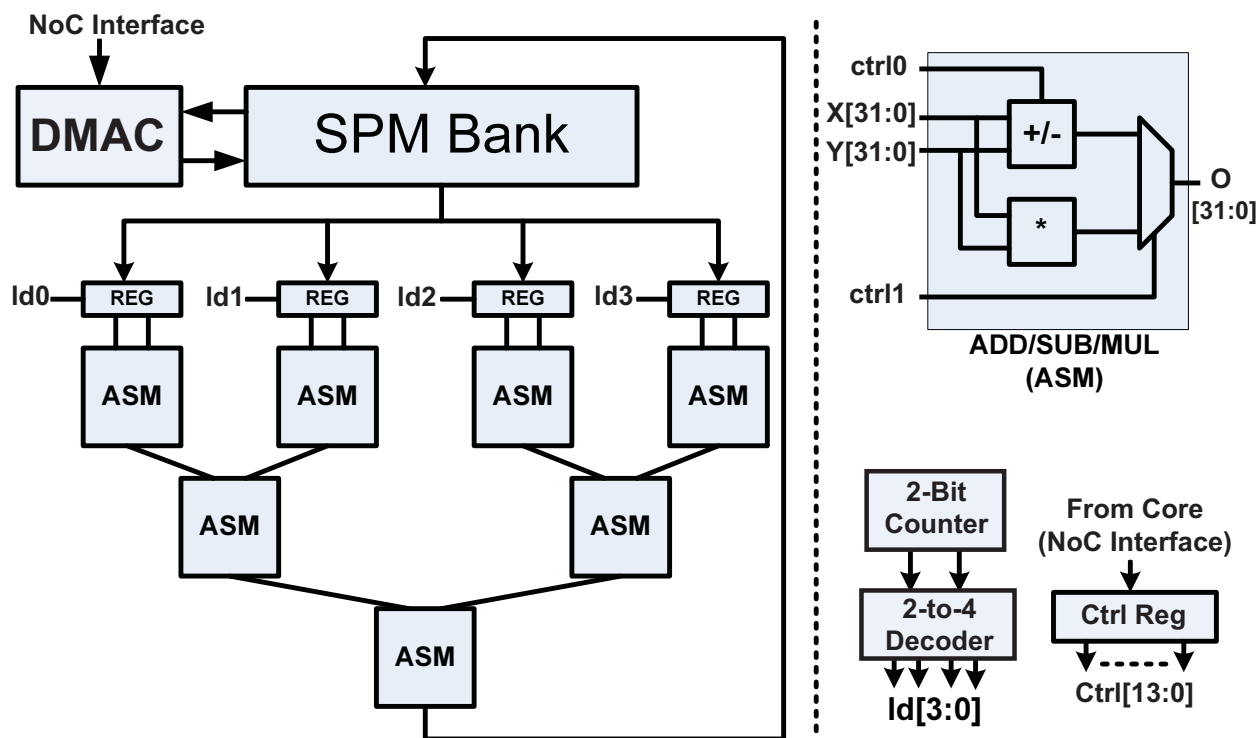


Figure 4.4: Details of Poly ABB

Table 4.2: Area/Power results – ABBs

Name	Area (μm^2)	Power (mW)	# of Units
FDiv	4949	0.264	12
Poly16	38276	1.608	96
FInv	3503	0.141	12
FSqrt	58683	1.83	8
ABC	8383	0.066	1
SPM - 4 KB, 1 read/write port	13591	17.6	288
SPM - 768 B, 1 read/write port	2545	7	72

In our experiments, we investigate systems consisting of 1 to 8 processors, and a set of either physical LCAs or ABBs. When modeling a system consisting of physical LCAs, the baseline (i.e. $1 \times$ LCA area) includes all the accelerators required to run a single instance of each benchmark without contention. When modeling a system featuring ABBs, the number of ABBs in the baseline (i.e. $1 \times$ ABB area) corresponds to the total amount of area that would have otherwise been devoted to LCAs. As such, the baseline *CHARM* platform is designed with a total of 128 ABBs that are organized into 8 ABB islands. Each island contains 16 ABBs (3 FInv/FDiv, 1 FSqrt, and 12

Table 4.3: Area/Power results – LCAs

Name	Area (μm^2)	Power (mW)	# of SPM Banks
Deblur	2013228	110.9	9
Denoise	496908	16.5	6
Registration	3853098	183.9	18
Segmentation	688298	27.3	6
EKF-SLAM	1188252	42.0	24
LPCIP	239159	6.11	6
SPM - 2 KB, 2 read ports, 1 read/write port	37043	17.5	–

Table 4.4: Area (mm^2) for various chip components

Core	NoC	Cache & Dir	CHARM HW	CHARM Total	LCA HW
10.8 [101] (scaled to 32nm)	0.3 [102]	39.8 [100]	8.3 (14% of chip) (see Table 4.2)	59.2	8.5 (14.3% of chip) (see Table 4.3)

Poly16) along with 16 SPM banks to provide concurrent access to all the ABBs. Table 4.4 shows the area for the main components of the chip (note that “CHARM HW” accounts for the area of the ABB islands and the ABC, while “CHARM Total” also includes the core, NoC, and memory).

To illustrate the load-balancing capacity of our ABC, we also include additional experiments where the LCA- and ABB-based systems have some multiple of the number of accelerators used in the baseline platforms. For the LCA-based systems, all LCA numbers are multiples of the base amount; similarly, the ABB-based systems are configured to correctly scale the number of ABBs present on each island. We also scale the amount of SPM space on each ABB island proportionally.

4.2.2 Evaluation Schemes

We evaluate *CHARM* by comparing the following architectures.

Physical LCA sharing with Global Accelerator Manager (*LCA+GAM*): In this architecture, physical LCAs can be shared between multiple cores. Each benchmark in our domain is accelerated with special-purpose accelerators. A global accelerator manager (GAM) is implemented in hardware to dynamically allocate physical LCAs to cores. We examine cases where there are between 1 and 8 replications of each required accelerator. This allows for the concurrent execution

of multiple instances of any specific benchmark, one for each accelerator in the system. Also, LCAs are powered off when not in use. This approach is similar to the architecture in [14].

Physical LCA sharing with ABC ($LCA+ABC$): In this architecture, a core may share physical LCAs using a centralized hardware-based ABC. In addition, the ABC can load-balance the available physical LCAs. We examine cases where there are between 1 and 8 replications of each required LCA. Since the ABC can split tasks among multiple LCAs, we are able to take advantage of all LCAs of a given type, even when only a single instance of that benchmark is executing. In the cases where there are more LCAs than can be allocated to available tasks, extra LCAs remain powered off.

ABB composition and sharing with ABC ($ABB+ABC$): In this architecture, a centralized hardware-based ABC is responsible for composing and managing available ABBs, load-balancing the tasks, and managing TLB requests from ABBs. We examine multiple ABB quantities. For the purposes of making a comparison, we will refer to a quantity of ABBs with area equal to a single replication of each LCA in the domain to be comparable to the case where we have one of each physical LCA in the domain. Typically these ABBs can be used to make multiple virtual LCAs, but this gives us a metric by which to make a fair comparison to the $LCA+GAM$ and the $LCA+ABC$ cases. In the cases where there are more ABBs than can be constructed into LCAs, the extra ABBs are left powered off.

For all cases, unless otherwise stated, we run four benchmarks from the medical imaging domain (i.e. *Deblur*, *Denoise*, *Registration*, and *Segmentation*). The benchmarks are run with volumetric images of 32-pixel cubes in multiple iterations.

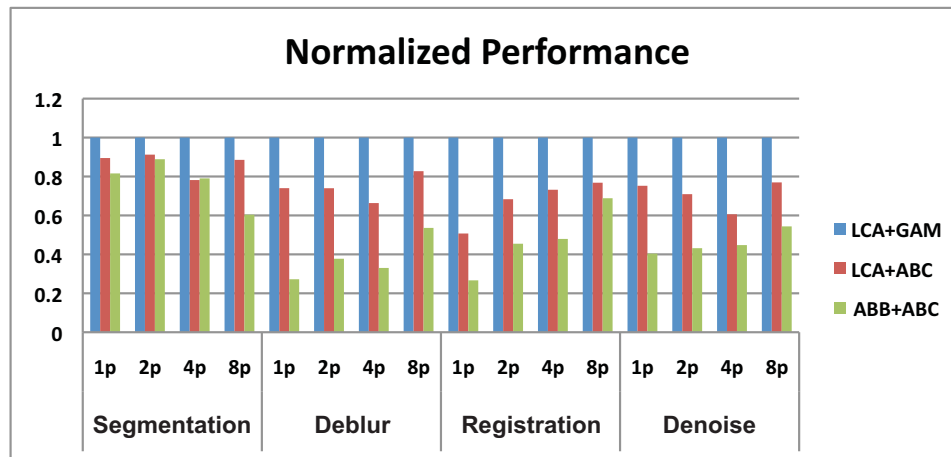


Figure 4.5: Performance improvements of medical imaging applications

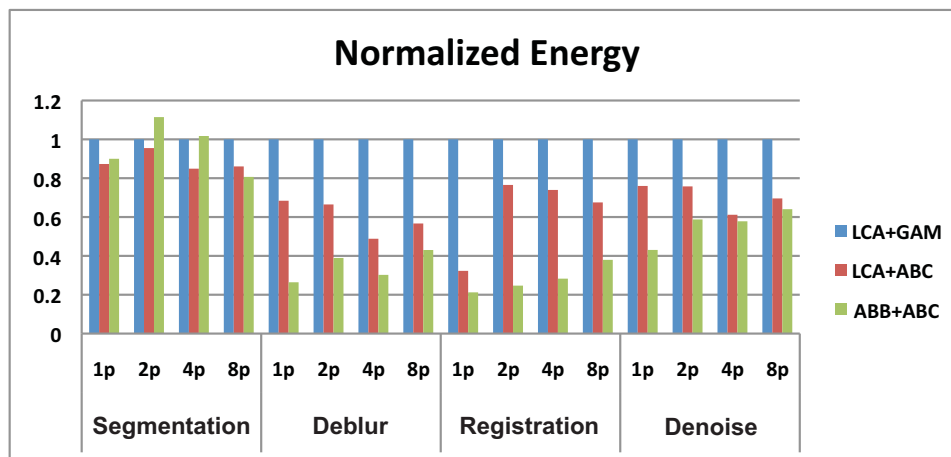


Figure 4.6: Energy gains of medical imaging applications

4.3 Experimental Results

4.3.1 Improvement over LCA-Based Systems

Figures 4.5 and 4.6 show the performance (i.e. execution time) and energy consumption of the *ABB+ABC*, *LCA+ABC*, and *LCA+GAM* evaluation schemes. All numbers shown here are normalized to the corresponding *LCA+GAM* result. In each case, we have the same number of processors, threads, and accelerators (e.g. the 4p case has 4 processors, 4 threads, and $4\times$ accelerators). On average, the *ABB+ABC* scheme achieves more than $2.4\times$ energy improvement over

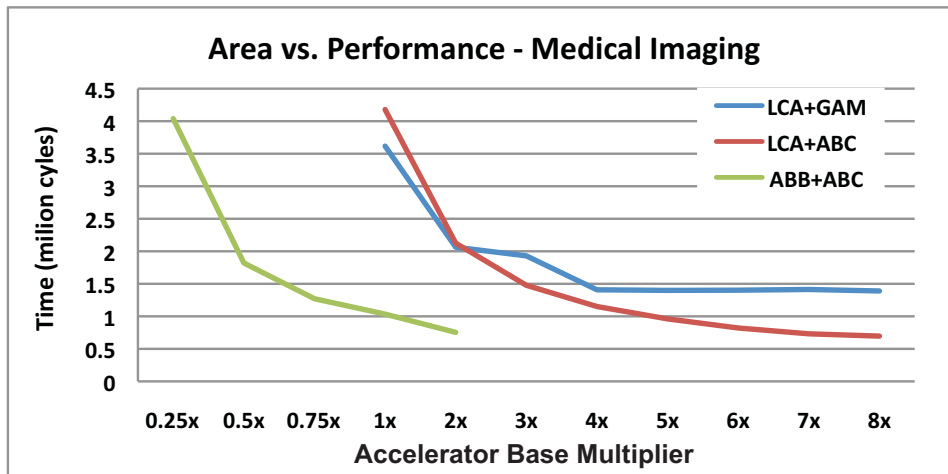


Figure 4.7: Effect of increasing accelerators

LCA+GAM (maximum $4.7\times$) and $1.6\times$ energy improvement over *LCA+ABC* (maximum $3\times$), while achieving even larger improvements in terms of performance. In general, as the number of independent tasks increases, *ABB+ABC* shows better performance because the ABC starts assembling ABBs to virtualize new LCAs (so long as ABBs are available in the system). This creates more parallel tasks, thereby achieving better performance and consuming less energy. We note that *ABB+ABC* running *Segmentation* with two processors (i.e. the 2p case) shows higher energy consumption compared to the other schemes. Since the performance for *Segmentation* improves only slightly, the overhead for constructing LCAs and coordinating communication between ABBs consumes more energy than the amount conserved by the slight reduction in execution time.

4.3.2 Effect of Increasing Accelerators

Figure 4.7 shows the effect of increased accelerators on the performance of the various schemes we study. For this experiment, we fix the number of processors and threads at four. To evaluate the LCA-based schemes (*LCA+GAM* and *LCA+ABC*), we vary the number of LCAs from 1 to 8 times the amount of LCAs in the base platform (i.e. the accelerator base multiplier ranges from $1\times$ – $8\times$). For the ABC+ABB scheme, the quantity of ABBs ranges from $0.25\times$ – $2\times$ of the base amount.

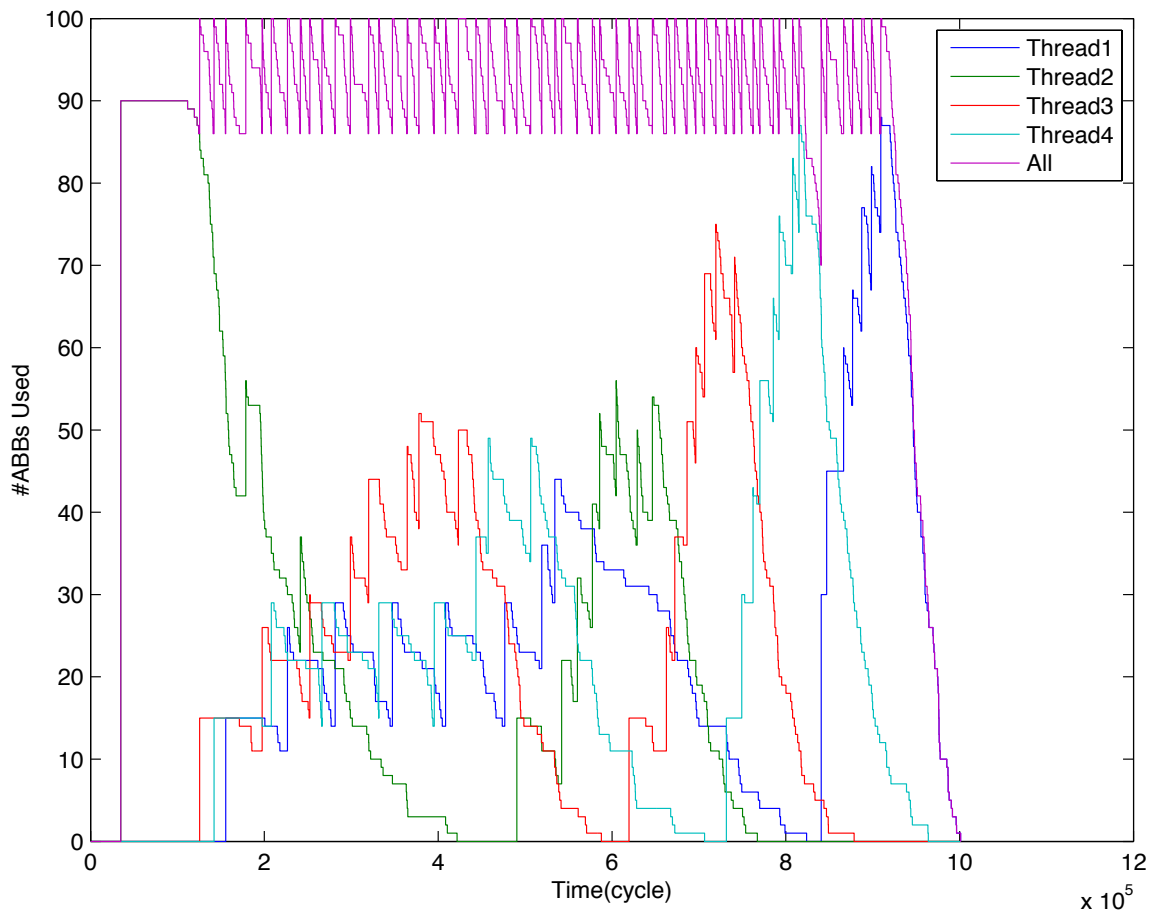


Figure 4.8: Utilization of ABBs given a task-grain of 8

There are several significant observations for these results. First, we see that adding more accelerator resources in general improves speedup, and we observe very similar results for energy improvement as well. Second, as accelerator resources are increased, performance improvements can be seen much earlier in the *ABB+ABC* scheme than in the LCA-based schemes (e.g. notice the base multiplier range of $1.5\times-2\times$ in *ABB+ABC* vs. the $6\times-8\times$ range in *LCA+GAM* and *LCA+ABC*). The reason for this is because a benchmark using physical LCAs only uses those of a specific type, and consequently only a small number of the total LCAs. With an ABB-based scheme, on the other hand, even $1\times$ area allocation can reconstruct many copies of a virtual LCA to run concurrently. The *ABB+ABC* is therefore free to replicate virtual LCAs out of the entire sum of accelerator resources, rather than leaving resources unutilized. An implication of this is that the *ABB+ABC* case saturates much more quickly in the acceleration that it can offer, either

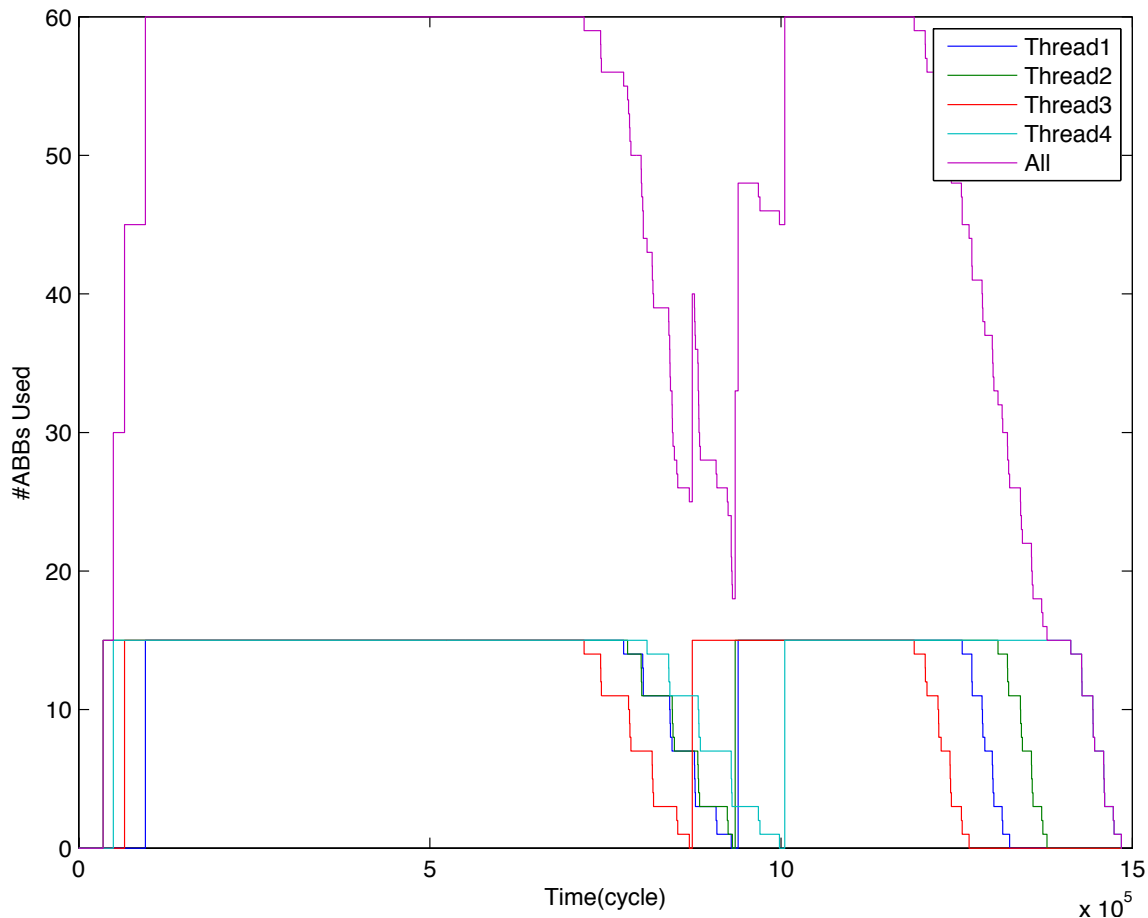


Figure 4.9: Utilization of ABBs given a task-grain of 128

exhausting potential parallelism or becoming memory-bound. Third, after reaching a base multiplier of $4\times$, the *LCA+ABC* scheme continues improving performance, while the performance of the *LCA+GAM* scheme flattens. We explain this by noting that the ABC splits each individual LCA invocation into multiple tasks and dynamically load-balances these tasks among accelerator resources. Hence, it benefits from having more than one LCA available per accelerator invocation. The GAM, however, allocates accelerators directly to the calling thread and is incapable of assigning tasks to more of them without the software having explicitly requested multiple accelerators.

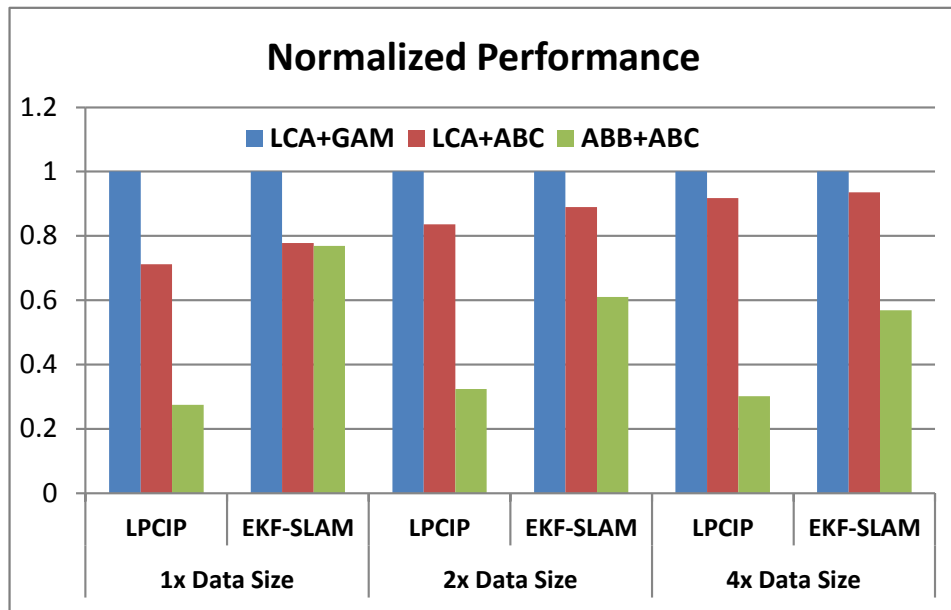


Figure 4.10: Performance improvements of computer vision and navigation applications

4.3.3 Effect of Modifying Task-Grain

Task-grain is the maximum number of individual computations in each task that is assigned to a set of composed ABBs. The smaller the task-grain, the more parallelism there can be in cases where computations can be performed independently of one another. In order to gauge the impact of task-grain on the ABB usage of each thread, we measure the number of ABBs allocated to LCA invocations by each thread for every moment of execution. For brevity, we only show the results for **Registration**, but the other benchmarks exhibit these same characteristics. We show this utilization for two cases: a task-grain of 8 (Figure 4.8) and a task-grain of 128 (Figure 4.9). Each figure shows the ABB usage by each thread and the total number of ABBs used (the upper-most curve). When the task-grain is 8, there is more parallelism and so more ABBs can be quickly allocated to a given thread (e.g. the initial spike seen in Figure 4.8). When the task-grain is 128, only one set of ABBs is used by each thread. The values shown in Figure 4.9 describe the ABBs allocated for a single LCA instance per thread. Also shown in Figure 4.8 is the impact of our round-robin scheduling, which ensures a measure of fairness when allocating ABBs. The jagged

nature of the curve representing the total ABB usage is the result of freeing ABBs prior to their reassignment.

4.3.4 Platform Flexibility

An original argument we put forth as a justification for fine-grained acceleration was reusability of the system in terms of both block design and retargetability. To substantiate this argument, we examine two applications from two domains that are completely unrelated to medical imaging: computer vision and navigation. Computer vision and navigation require compute-intensive data processing, heavily consisting of linear algebra and floating point computation, to attain high levels of situational awareness. We examine feature extraction based on log-polar coordinate image patches (LPCIP) [93] from computer vision and simultaneous localization and mapping based on the extended Kalman filter (EKF-SLAM) [66, 67] from navigation. A more detailed description of these two applications and existing acceleration strategies can be found in [54]. Figure 4.10 shows results comparing the use of our ABB-based *CHARM* platform originally designed for medical imaging (unmodified) against custom physical LCAs that specifically target these new domains. The consistent performance gains illustrate that our ABB-based platform is flexible, and is much more broadly targetable than a typical platform featuring custom LCAs.

Chapter 5

CAMEL: Composable Accelerator-Rich Microprocessor Enhanced for Longevity

In the face of new algorithmic innovations or migrations to other application domains, adaptivity becomes a key characteristic for domain-specific platforms. While there is nearly unlimited potential for longevity and flexibility, an approach entirely based on programmable fabric (PF) results in accelerators that are considerably larger and slower than ASIC accelerators. Specifically, an FPGA implementation is on average $40\times$ larger and $3.2\times$ slower, with $12\times$ higher dynamic power consumption than its ASIC counterpart [103].

For this reason we introduce the composable accelerator-rich microprocessor enhanced for longevity (*CAMEL*), a hybrid approach that combines the performance of composable ASIC accelerators with the flexibility and longevity of PF-based accelerators. The PF will enable the instantiation of new accelerator building blocks, while the performance impact of the fabric will be mitigated by the fact that we still maintain a rich set of domain-specific building blocks implemented in ASIC.

We will therefore support composition of accelerators that are a mixture of both ASIC and PF components.

This approach provides a number of benefits. First, the PF serves as a design catch-all: we need not implement in ASIC the infrequently used accelerator building blocks of a given domain, as these blocks can be covered by the PF. This frees up silicon resources for more critical building blocks. Second, the PF can help adapt to variations in a domain or algorithm: we can more efficiently employ our design for domains/algorithms that differ from those for which the design was originally intended by instantiating new building blocks in the PF, while still employing useful building blocks that were already implemented in ASIC. The main aspects of this work include the following:

- **Compiler and Runtime Framework to Support ASIC and PF Allocation** – Our compilation framework generates a data flow graph of interconnected building blocks for a given kernel; it can also perform platform-aware partitioning of the data flow graph into subgraphs that can be accommodated by on-chip resources; at runtime, our resource manager uses these graphs to compose accelerators by allocating either ASIC- or PF-based building blocks.
- **Slack Analysis and Rate Matching** – Our compiler statically identifies imbalance in the data flow graph, and compensates for the slack in shorter computational paths by allocating extra buffer space; our hardware reduces PF performance overhead through *rate-matching*, where it instantiates multiple PF-based building blocks to collectively match the ASIC design throughput.
- **Design Space Exploration** – We demonstrate the enhanced flexibility of our approach through analysis on four distinct application domains, where we examine the benefits our approach provides to both design extensibility and longevity; while we evaluate our results on a single candidate architecture that supports accelerator composition, our techniques are generally applicable to other composable architectures as well.

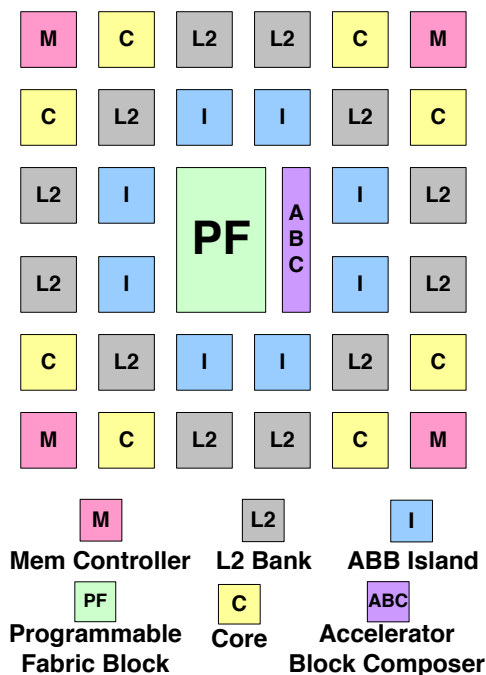
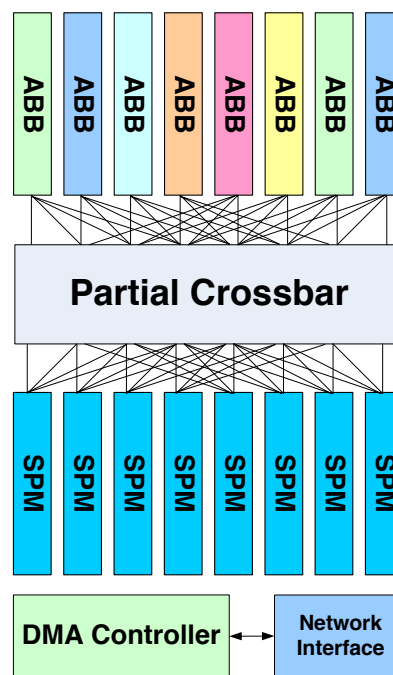
Figure 5.1: *CAMEL* Microarchitecture

Figure 5.2: Design of ABB island

5.1 Overview of *CAMEL*

The *CAMEL* architecture uses a combination of software and hardware components to improve flexibility and longevity. The hardware components are responsible for accelerator composition, where the virtual accelerators, or loosely-coupled accelerators (LCAs), are dynamically constructed using either the available ASIC-based accelerator building blocks (ABBs), or ABBs that have been instantiated in PF. While our contributions in the *CAMEL* architecture are generally applicable to composable architectures, in this work we implement our techniques and analyze results on the *CHARM* architecture [15]. An overview of the *CAMEL* microarchitecture is presented (not to scale) in Figure 5.1. This figure consists of a set of cores with private L1 caches, shared L2 cache banks, and the following specialized *CAMEL* components: (1) ABBs grouped into a series of islands (shown as “I”); (2) accelerator block composer (ABC) responsible for accelerator composition, PF assignment, and resource arbitration; and (3) PF for instantiating additional ABBs.

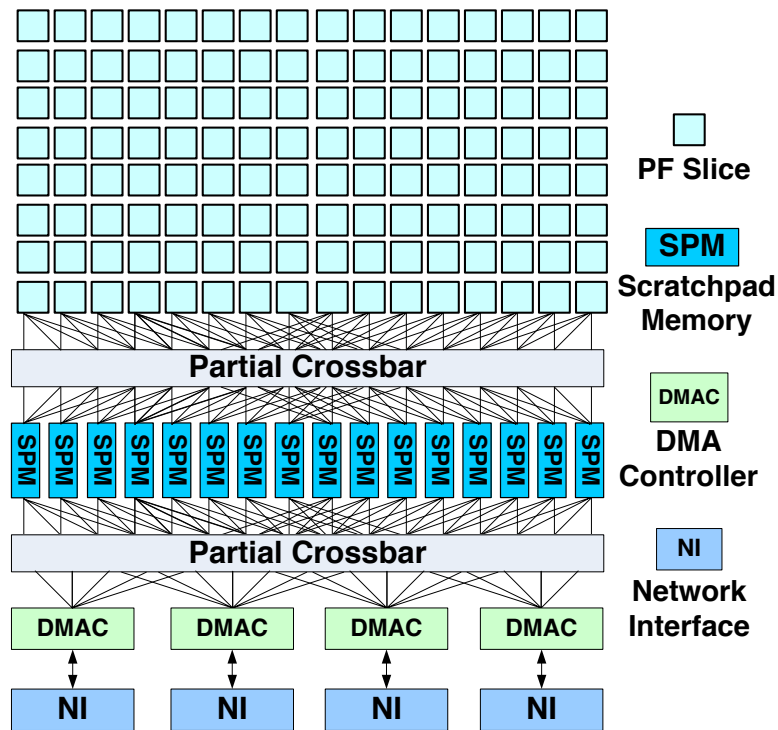


Figure 5.3: Design of programmable fabric (PF)

5.1.1 ABB Islands

Figure 5.2 shows the internal structure of an ABB island; in this sample figure there are 8 ABBs, 8 scratchpad memory (SPM) banks, and 1 multi-channel DMA controller (DMAC). Each ABB has access to only 4 of the SPM banks using a partial 8x8 crossbar [104]. These SPMs are in turn connected to the multi-channel DMAC. The numbers and types of the ABBs are determined using software-driven design space exploration, and the ABBs of a given type are distributed evenly across the islands in a round-robin fashion.

5.1.2 Programmable Fabric

The PF is used for hosting the ABBs required by new applications (in existing or even entirely new domains). The internal design of the PF in *CAMEL* is shown in Figure 5.3. It consists of PF slices, 16 SPM banks, 4 DMACs, 4 network interfaces (NIs), and 2 crossbars: one to connect a

selected set of PF slices to SPMs and one to connect SPMs to DMACs. Although a monolithic PF presents challenges in its shared usage (i.e. ports, NoC congestion, etc.), it accommodates ABBs of any size and avoids performance hits due to static partitioning of resources. The main advantage of using a PF is its reusability and runtime reconfigurability. However, ABBs implemented on the PF are less area- and power-efficient, and have lower performance compared to ABBs implemented in ASIC technology. While the area and power issues are largely technology-dependent, we address energy consumption and performance using hardware techniques that compensate for the mismatch in computational speed.

When a virtual LCA is invoked, software sends to the ABC an encoded data flow graph representing the LCA's functionality. Nodes in this graph represent functionalities of individual ABBs, while edges represent data transfers. This functionality is executed in a pipelined fashion, with each ABB in the graph communicating with others by means of bulk transfers from its local SPM to remote SPMs or memory, and vice versa. If a PF-implemented (presumably less efficient) ABB is on the critical path, it can negatively impact the performance of the entire LCA. Figure 5.4 exemplifies this scenario and demonstrates how rate-matching can help. In this figure, the same data flow graph is instantiated for three different hardware allocation scenarios, and we see how four independent data sets (illustrated by four different shading patterns) would flow through the connected ABBs. As Figure 5.4A shows, when all ABBs are operating at the same frequency (e.g. $f = 1$), the LCA they compose will have that same throughput. However, as shown in Figure 5.4B, if one of the ABBs is slower than the others (e.g. ABB3 has $f = 1/2$), this ABB becomes a bottleneck and the other ABBs are forced to stall. This results in the LCA as a whole progressing at the rate of this single slow component. Since the ABBs allocated in the PF typically have less throughput than ASIC ones, the inclusion of a PF-based ABB could often result in such a bottleneck.

To address this, *CAMEL* allocates multiple copies of the slower ABB to bring the aggregate throughput of the collection of slow ABBs up to match that of the faster ABBs. This is referred to as *rate-matching*, and is shown in Figure 5.4C. Provided there are sufficient PF resources for multiple ABB instantiations, this technique interleaves independent data sets between the duplicated PF-based ABBs and allows for the LCA to make more efficient use of the ASIC-based ABBs. As

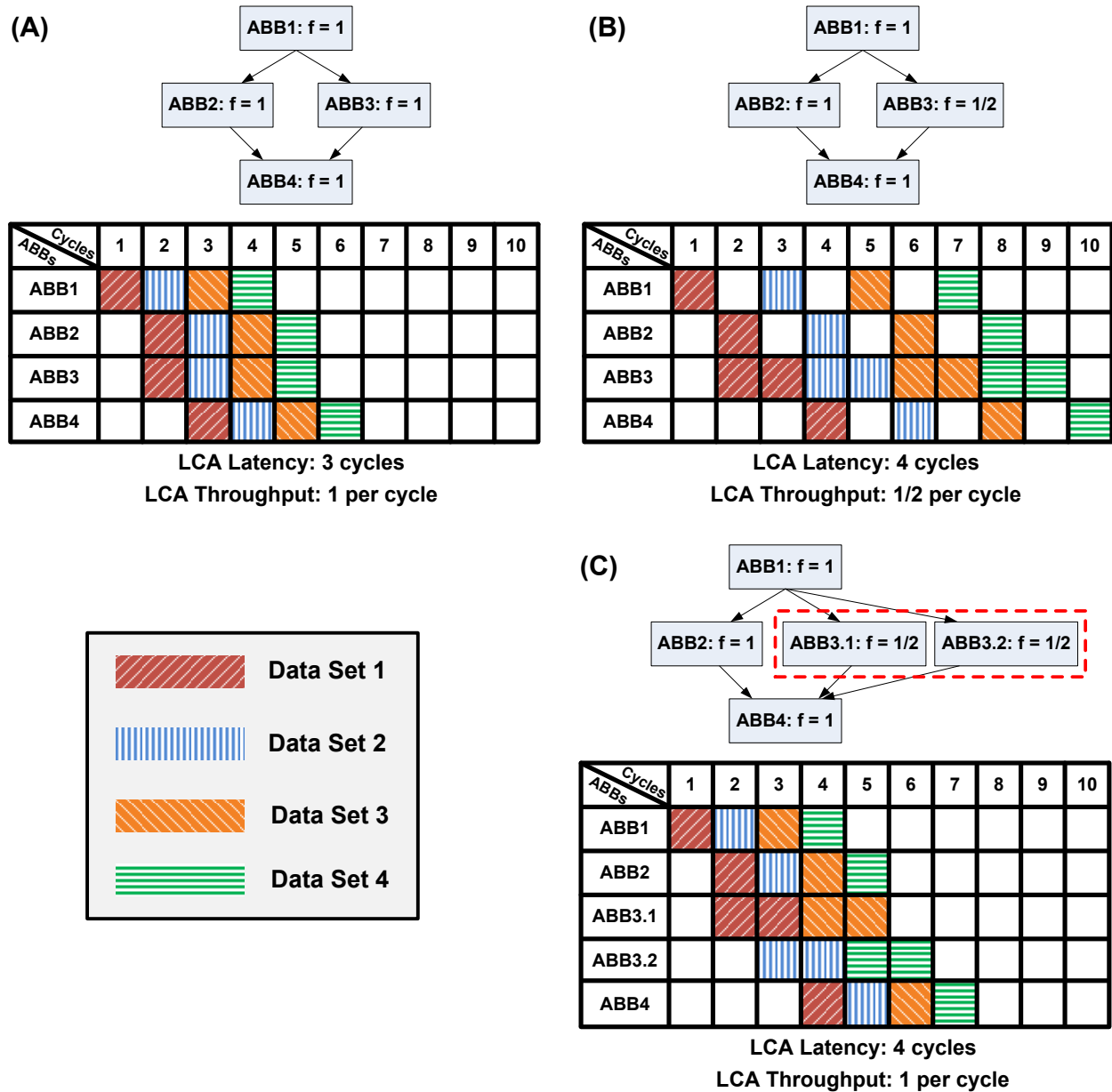


Figure 5.4: Motivational example of applying rate-matching on PF

throughput is increased, the other ABBs and overall system components are left idle for a shorter period of time, thereby reducing static energy consumption. Although dynamic power is slightly increased, dynamic energy remains relatively constant and so overall energy consumption is reduced. Thus, rate matching simultaneously improves performance, resource utilization, and energy efficiency. The implementation of this technique is described in Section 5.1.3.

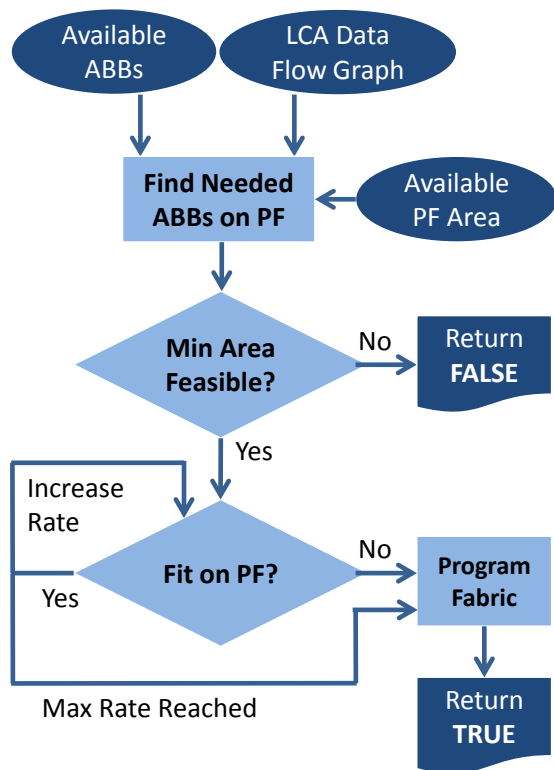


Figure 5.5: PF allocation algorithm

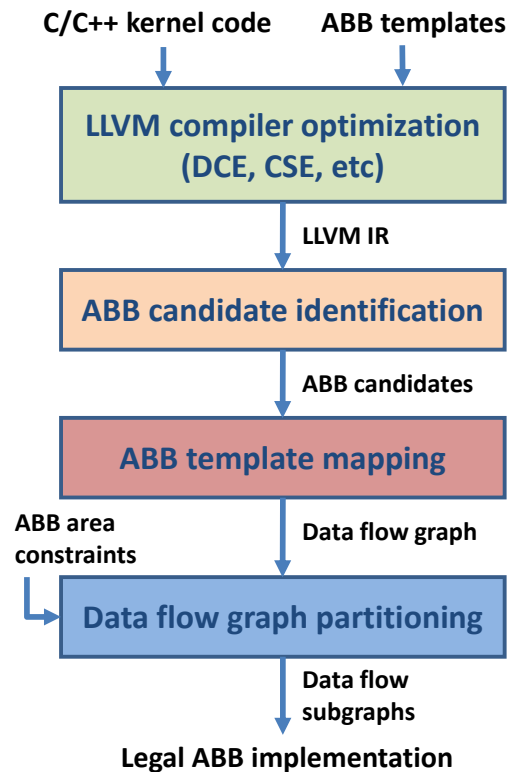


Figure 5.6: Compiler framework

5.1.3 Runtime PF Allocation

The ABC performs PF-based ABB allocation using the algorithm shown in Figure 5.5. It receives information on the available space on the PF, along with the list of available ASIC ABBs and the LCA data flow graph. Using these it determines what ABBs to allocate in PF. To achieve the best allocation, it starts with the minimum configuration as a feasibility test; if the minimum temporarily cannot fit, it keeps track of the current task until enough space is available on the PF. If the minimum cannot be implemented at all, the ABC informs the requesting core of the failure to implement. After passing the feasibility check, the ABC attempts rate-matching: it iteratively increases the PF-based allocation of critical ABBs (i.e. those on the critical path of the data flow graph) until either no space is left on the PF or the best rate-match is achieved.

5.1.4 Compiler Support

An overview of the *CAMEL* compiler framework is shown in Figure 5.6. Given information on ABB types to potentially use, the compiler is responsible for mapping a given program kernel to a set of those ABB types, producing a data flow graph whose nodes are ABBs and whose edges are data transfers. The algorithm used is similar to that described in [105]. Provided supplemental information on the available ASIC ABBs and PF for a given platform, the compiler can also determine if a kernel being mapped is too large for the total number of ASIC ABBs combined with the total PF. In these cases, the kernel’s data flow graph is partitioned into the fewest number of regions such that allocation is possible. Partitioning is done along specific regions of the graph so as to minimize data transfers between partitions; temporary storage is also allocated to store intermediate data. The partitioned regions become subgraphs that can then be run sequentially. An example of this is shown in Section 5.3.4. After a mapping solution exists, addressing for the local SPM of each ABB is calculated. Part of this calculation is an optimization for graphs that feature multiple paths of different lengths (i.e. slack) between a pair of nodes. Once this slack is identified, computational correctness is ensured by allocating extra buffer space along shorter paths. By avoiding stalls, this method allows for higher ABB utilization and overall throughput along all paths.

5.2 Evaluation Approach

5.2.1 Simulation Platform

In order to evaluate our approaches, we have extended Simics [49] and GEMS [50] with the cycle-accurate models needed by *CAMEL*. Table 5.1 describes the simulation parameters used. We also implemented a complete tool-chain for generating simulator models starting from C-based kernel code. Table 5.2 provides information on the additional tools used for acquiring accurate timing and power values for these models. In addition, our compiler framework has been implemented in LLVM, and has an average compilation time of 6.1 seconds per kernel for our benchmarks.

Table 5.1: Simulation parameters

Parameter	Value
L2 cache	8 MB, 8-way set-associative, 32 banks; latency: 10 cycles
Coherence protocol	Shared banked L2 cache; L2: MOSI; L1: MSI
Main memory	Latency: 280 cycles; bandwidth: 10 B/cycle per controller
Network topology	4x8 mesh; link latency: 1 cycle; router latency: 5 cycles; bandwidth: 72 B/cycle per link
ABB islands (base)	16 islands; 14 ABBs and 14 SPMs (4 KB each) per island

Table 5.2: Tools for timing and power models

Tool	Purpose
Xilinx Vivado Design Suite [106]	Accelerator high-level synthesis
Synopsys Design Compiler (32nm) [95]	ASIC synthesis (power, performance)
Xilinx ISE [106]	PF synthesis (performance)
Xilinx Virtex 6 XPower Estimator [106]	PF power analysis
CACTI [100]	Cache and scratchpad modeling
Orion [102]	NoC power and area
McPat [80]	Core and cache power analysis

5.2.2 Domains

In this work, we target four application domains: medical imaging, commercial, computer vision, and navigation. These four domains not only provide coverage of real-world applications with interesting computational demands, they also represent classes of applications that are algorithmically diverse in nature. Table 5.3 shows the numbers and types of ABBs used for accelerating each domain using one set of accelerators. Note that by *one set of accelerators* we mean as many ABBs as it would take to instantiate one of each virtual LCA in the domain. In our experiments, we have used four sets of accelerators.

5.2.2.1 Medical Imaging (Med)

Medical imaging is an important tool for diagnosis and treatment. Because of the high volumes of data and high computational demands, the algorithms cannot be easily used in real-time clinical diagnosis, making them excellent candidates for acceleration. The medical imaging pipeline includes denoising (**Denoise**), deblurring (**Deblur**), fluid registration (**Registration**), and image

segmentation (**Segmentation**). These algorithms and their acceleration strategies are described further in [107].

5.2.2.2 Commercial (Com)

To represent the commercial domain, we have selected three applications from the PARSEC [108] benchmark suite: **Blackscholes**, **Streamcluster**, and **Swaptions**. These applications solve partial differential equations, online clustering problems, and probability distribution estimations.

5.2.2.3 Computer Vision (Vis)

Computer vision is a compute-intensive domain with inherent parallelism that makes it ideal for the streaming-data style of acceleration. Two main categories of applications in this domain are feature extraction, for which we include implementations of **SURF** from OpenCV [76] and **LPCIP** from MRPT [75], and image processing, for which we include the **Texture Synthesis** application from SD-VBS [109]. These applications provide a variety of computation including complex matrix-based, trigonometric, log-polar, and gradient-histogram computations, with fluctuating memory usage.

5.2.2.4 Navigation (Nav)

Navigation is a compute-intensive, AI-related domain that aims to achieve high levels of situational awareness. We include **EKF-SLAM** from MRPT [75], along with **Robot Localization** and **Disparity Map** from SD-VBS [109]. These applications provide diverse computation in the form of partial derivatives, covariance, spherical coordinates, probabilistic models, particle filters, and search for minimal sum of absolute differences.

Table 5.3: ABB types, PF synthesis, domain numbers, and functionality

ABB Type	FPGA Slices	Power (mW)	Freq (GHz)	ABBs per Domain				Functionality
				Med	Com	Vis	Nav	Description
poly	3536	571	1/4	47	95	143	167	16 I/O Polynomial (floating pt.)
sqrtf	672	176	1/3	2	1	1	3	Square root (floating pt.)
divf	255	84	1/3	6	5	6	7	Divide (floating pt.)
powf	672	176	1/3	1	3	1	0	Power function (floating pt.)
logf	672	176	1/3	0	3	0	0	Log base e (floating pt.)
rr1D	25	2	1/2	0	2	0	0	Random read in 1 dimension
rr2D	90	70	1/2	0	0	2	0	Random read in 2 dimension
rr3D	145	91	1/2	0	0	73	0	Random read in 3 dimension
rw1D	25	2	1/2	0	1	0	0	Random write in 1 dimension
selff	58	54	1/2	0	4	50	0	MUX (float inputs, float select)
selfi	57	54	1/2	0	3	4	0	MUX (float inputs, int select)
selif	27	84	1	0	4	8	0	MUX (int inputs, float select)
selii	30	85	1	0	1	0	0	MUX (int inputs, int select)
sum	134	77	1/2	0	1	8	1	Accumulate a vector
castfi	94	32	1/4	0	0	43	0	Cast float to integer
castif	108	35	1/4	0	0	1	0	Cast integer to float
mod	255	84	1/3	0	0	2	0	Modulo
min	65	54	1/2	0	0	3	0	Find minimum value in vector

5.2.3 ABB Characterization

The ASIC ABBs for our system have all been synthesized with a frequency of 1.0 GHz and an initiation interval (II) of 1. Although the PF ABBs also have II's of 1, they have different operating frequencies depending on their type. Table 5.3 details the results of synthesizing the various ABB types for a Xilinx Virtex6 FPGA; this table also includes the numbers of ABBs needed by the four domains and the functionalities of the ABBs. Note that the ABB granularities and functionalities have been determined according to a domain-space optimization done primarily for Med (which is the base domain of *CAMEL* in our evaluation schemes – see Section 5.2.4), with additional ABB types added as needed.

Table 5.4: Power and area values for components of the *CAMEL* base platform

Unit Type	Num. Units	Power per Unit (mW)	Area per Unit (μm^2)
ABC	1	66.00	8383
poly ABB	188	6.65	362570
sqrtf ABB	8	9.49	368819
divf ABB	24	0.52	15117
powf ABB	4	9.49	368819
SPM	240	17.60	40773
DMAC	20	0.59	10071
L2 Bank *	32	148.97	881990
Core *	1	686.46	9868400
NoC *	1	4923.52	557978

* Power varies with execution; average power values are shown.

Table 5.5: Number of ABBs and PF slices in *CAMEL-x%*

	<i>CAMEL-0%</i>	<i>CAMEL-10%</i>	<i>CAMEL-20%</i>	<i>CAMEL-30%</i>	<i>CAMEL-40%</i>	<i>CAMEL-50%</i>
# ABBs	224	192	168	148	128	108
# PF Slices	0	2466	4935	7404	9873	12342

5.2.4 Evaluation Schemes

For the purposes of this work, we consider the case where single benchmarks are run and accelerator needs are known. As such, PF reconfiguration is done statically and reconfiguration time is excluded from all results. In our experiments, we have considered the following schemes, each representing a different class of accelerator-based architectures:

- **GPU** – Tesla M2075; performance measures only consider computation (not data transfer).
- **LCA-ASIC** – accelerator-rich platform where all LCAs are monolithic and ASIC-based [14].
- **LCA-FPGA** – accelerator-rich platform where all LCAs are monolithic and FPGA-based [14].
- **CHARM** – composable accelerator-rich platform with Med base domain and no PF [15].
- **CAMEL-x%** – *CAMEL* architecture with Med base domain and “x” percent of the total ABB area substituted (by removing “x” percent of ABBs of each type, maintaining even ABB distribution across islands) for equivalent area of PF; x ranges from 0%–50%.

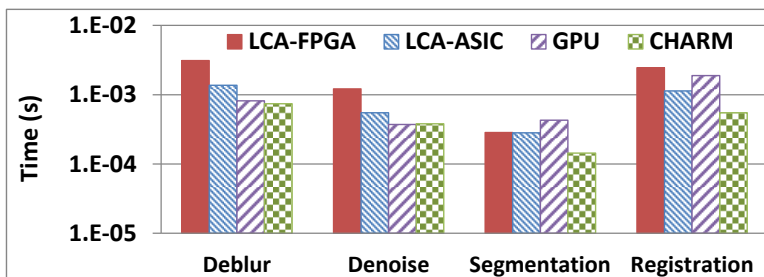


Figure 5.7: Performance comparison between acceleration schemes

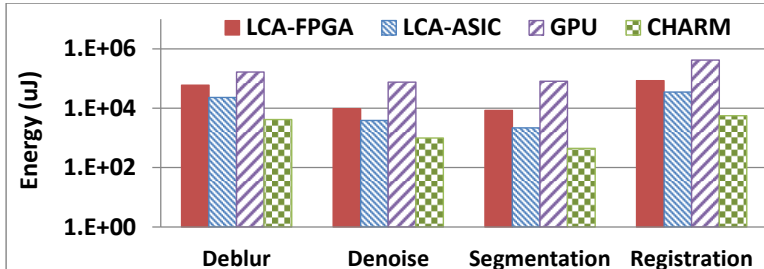


Figure 5.8: Energy usage comparison between acceleration schemes

The power and area values modeled for the *CAMEL-0%* base platform can be found in Table 5.4, where the total area of the chip is 122 mm^2 . To determine the number of PF slices that can fit in *CAMEL-x%*, we have used the die area size of Virtex6 (measured by taking X-ray photos) and have estimated 2955 um^2 for each slice in 32nm. Table 5.5 shows numbers of PF slices and remaining ASIC-based ABBs for each *CAMEL-x%* case. Note that ABB types vary in both area and quantity – the distribution shown corresponds specifically to our platform. As PF slices are linearly increased for the *CAMEL-x%* cases, different numbers of various types of ABBs are removed to make room for the PF area, so the total number of remaining ABBs does not necessarily decrease linearly.

5.3 Experimental Results

In this section, we present and discuss our simulated results. Although our Simics+GEMS framework simulates an UltraSPARC-III-i 1.0 GHz processor (running Solaris 10), we conservatively measure our performance gains in terms of a wall-time-based comparison to fully parallelized runs

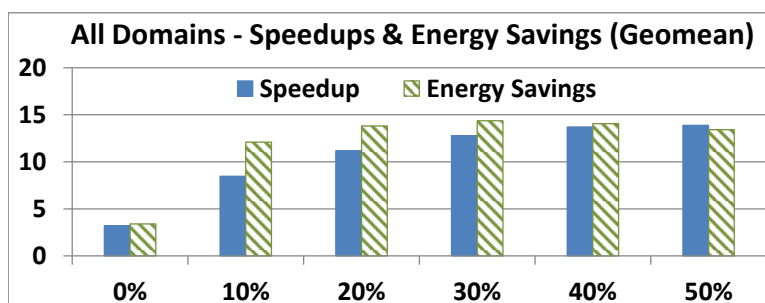


Figure 5.9: Geometric mean of all speedups and energy savings as the percentage of PF increases

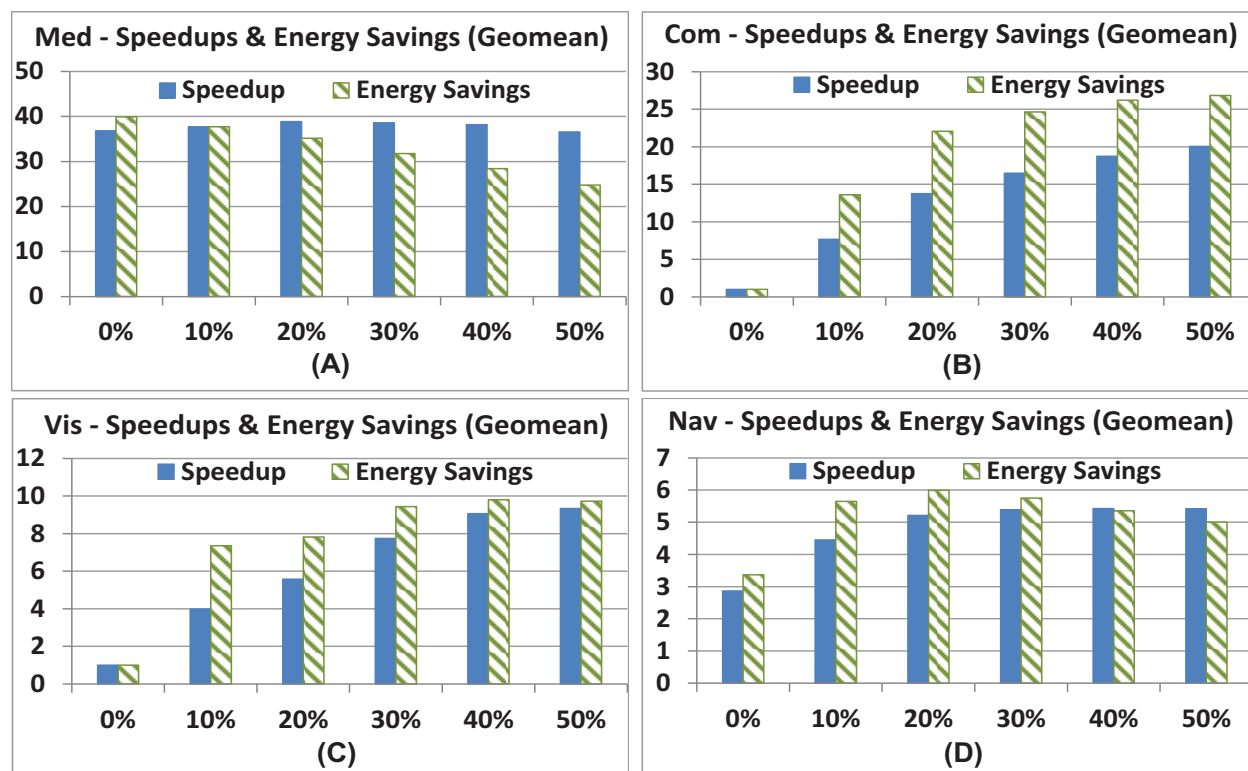


Figure 5.10: Geometric mean of speedup and energy savings for each domain as %-PF increases

on a 4-core 2.0 GHz Intel Xeon E5405 processor. When there are insufficient accelerator resources to run a benchmark, we fall back to running on the CPU, and thus exhibit no benefit.

5.3.1 Comparison Between Acceleration Schemes

Figures 5.7 and 5.8 compare four accelerator-based architectures running benchmarks from the Med domain. As it features domain-specific acceleration, *CHARM* (i.e. *CAMEL-0%*) outperforms by

2.1 \times and saves energy by 93 \times compared to the power-hungry *GPU*. Furthermore, with its ability to load-balance and dynamically virtualize LCAs, *CHARM* on average outperforms *LCA-FPGA* by 3.5 \times and *LCA-ASIC* by 1.8 \times , resulting in energy savings of 14.5 \times and 5.1 \times , respectively. For an optimal design, we desire the performance and energy usage of *CHARM* with the adaptivity of GPUs and FPGAs. We show next how *CHARM* is made adaptive for greater performance and energy savings across domains.

5.3.2 Effect on Domain-Span

To evaluate *CAMEL* support of domain-span, we use Med as a base domain (for ASIC ABBs) and choose three other target domains: Com, Vis, and Nav (as mentioned in Section 5.2). In all of these experiments, we keep the overall area constant by removing 0%–50% of the ASIC ABB area in increments of 10% (maintaining even distributions of ABB types across islands) and adding PF slices equivalent to the removed area.

In comparison to software-only versions of the benchmark executions, Figure 5.9 shows the aggregate speedup and energy savings for all four domains, while Figure 5.10 shows the average speedup and energy savings of each domain. Since most of the new applications from the target domains are unable to run on the base platform without the PF (i.e. exhibit 1 \times performance gain as they fall back to running on the CPU), the aggregate speedup of *CAMEL-0%* (i.e. *CHARM*) across all benchmarks is relatively low. As seen in Figure 5.10A, the Med applications, for which this base was originally optimized, see performance improvement with the addition of a small amount of PF, followed by a decrease in performance as more PF is added. This is intuitively correct, because the platform being considered was originally provisioned with the ASIC-based ABBs designed specifically for accelerating Med applications. A small amount of PF (10%–20%) provides adaptivity to improve load-balancing and resource utilization for each individual benchmark, while larger amounts of PF begin to starve the system of the improved performance efficiency of the ASIC ABBs. However, even the small performance improvement initially seen with the addition of PF is not enough to counterbalance the reduction in power-efficiency as ASIC ABBs are replaced

by PF. As a result, we see an initially small reduction in energy savings for *CAMEL-10%* and *CAMEL-20%*, followed by larger reductions for *CAMEL-30%* and onward.

For Com (Figure 5.10B), no applications can be implemented without PF because they all require a variety of new ABB types that do not appear on the base platform (refer to Table 5.3). As PF is added, these ABBs can be instantiated and rate-matched, resulting in large performance gains and energy savings. With Vis (Figure 5.10C), we see behavior similar to that of the Com applications. For Nav (Figure 5.10D), we observe an initial speedup even without the PF because this domain shares a lot of the same ABBs as Med, allowing some benchmarks to be minimally implemented on the base platform. As we initially increase PF, we are able to instantiate the missing ABBs and run all benchmarks, resulting in increased average gains in both performance and energy. However, similar to the trends we see with *CAMEL-10%* and *CAMEL-20%* for Med, as more ASIC is replaced by PF, the performance of the Nav domain continues improving slightly while its energy savings begin dropping (e.g. *CAMEL-30%* and onward).

In summary, as ASIC ABBs are removed and replaced by PF, more useful ABBs become available and rate-matching takes effect. This translates into better adaptivity, and often times higher performance and energy savings for new domains. While these trends depend on the specific workload being considered, as intuitively suspected, the less similar a workload is to the base domain of the platform, the more useful the PF. As with the law of diminishing returns, however, increasing the PF past a certain point starts reducing the improvements because the system begins removing too many of the useful ASIC ABBs and replacing them with equivalent PF-based ones. We see this turning point with $\sim 30\%$ PF for domains similar to the base (e.g. Nav) and $\sim 50\%$ PF for other domains (e.g. Com and Vis).

5.3.3 Effect on Domain Longevity

In order to evaluate the longevity of the base domain, we have added a new application to Med: compressive sensing magnetic resonance (CS_MR) [110]. This application requires one additional ABB, namely the “sum” ABB, which is not found on the Med base domain of *CAMEL*. This

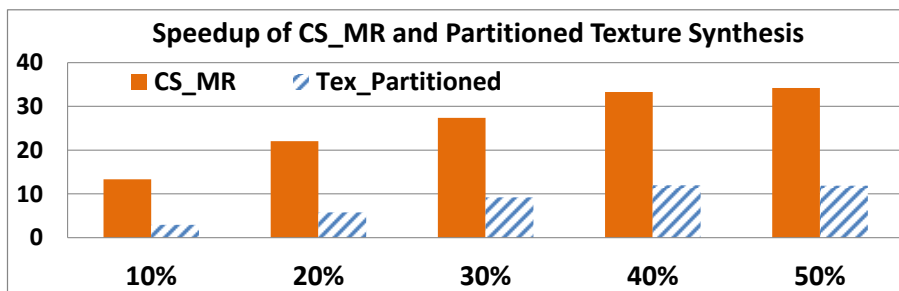


Figure 5.11: Domain longevity and graph partitioning impact for increasing percentages of PF

“sum” ABB is one that accumulates the values of a given vector, and is used to implement the internal FFT engine of *CS_MR*. The speedup result for *CS_MR* is shown in Figure 5.11. Since *CS_MR* does not use many of the ASIC-based ABBs on *CAMEL*, the increasing amounts of PF being provided are used to implement more “sum” ABBs, allowing more virtual LCAs to be instantiated and more speedup to be gained.

5.3.4 Graph Partitioning for Lower-Capacity Hardware

As described in Section 5.1.4, it is sometimes the case that a benchmark demands a massive LCA for a large kernel and requires more resources than are available on *CAMEL*, even with PF. Benchmarks like *Texture Synthesis*, *Swaptions*, *Streamcluster*, and *SURF* contain kernels that can never be implemented in their original form. To overcome this, our compiler partitions the data flow graph of each of these kernels into a number of subgraphs that can each fit on *CAMEL-x%* (e.g. *Texture Synthesis* requires 6 partitions for *CAMEL-50%*). As an example, Figure 5.11 shows the result of accelerating *Texture Synthesis* after applying this graph partitioning technique, where we are able to achieve up to 11.96× speedup.

Chapter 6

Neural Acceleration of Divergent Applications on SIMD Architectures

In this chapter, we present neural approximation as a solution to the problem of branch-divergence-induced performance degradation in single instruction multiple data (SIMD) architectures. Applying a neural network (NN) to branch divergence is a complex and non-trivial process, as the effectiveness of an NN relies on how well it adapts to fit a specific problem. The NN training, therefore, is critical for ensuring acceptable error rates for applications. For this reason, we present a complete methodology, including a software flow and supplementary optimization techniques.

We initially characterize the types of control flow seen with SIMD architectures and analyze performance degradation resulting from branch divergence. As we identify a potentially-divergent code region (or “kernel”), we train an artificial neural network (ANN) offline to approximate that kernel, and inject the ANN computation directly into the code in place of the extracted kernel. By converting control flow regions into non-divergent, approximate computation, we **remove the divergence problem entirely as a trade-off for introducing imprecision into the results**. Note that as our technique directly manipulates code without the need for costly hardware modifications, this is a *platform-agnostic* approach and can be readily adopted by data-parallel architectures that suffer

from branch divergence. To demonstrate the effectiveness of our neural-network-based solution, we evaluate our approach on a graphics processing unit (GPU) across a range of divergent applications from various domains.

6.1 Kernel Characterization

In order for kernels to be properly targeted for approximation, the following criteria must be satisfied:

- ▶ **Pure function** – no side effects (e.g. cannot modify external state)
- ▶ **Fixed-size inputs/outputs (I/O)** – no dynamic, variable-length inputs or outputs
- ▶ **“Approximable” region of code** – imprecision does not result in program failure

The function purity and I/O constraints are similar to the kernel constraints imposed by other hardware acceleration schemes [12, 14, 27]. Additionally, since the NNs will be approximating application kernels, the “approximability” constraint means the application itself must also be able to tolerate imprecision. For example, one cannot use an NN to approximate a kernel that computes the exact value of a memory address used to access data. However, if subsequent computation on that data is approximated (e.g. a heuristic-based feature detection algorithm from computer vision), approximation of the final result of that algorithm would be tolerable.

There is a large body of prior work [34–36, 40, 111] that has performed in-depth examination of the topic of approximability. The majority of these works have deemed it the responsibility of the programmer to determine which code regions are suitable for approximation. We operate under these same assumptions. To aid programmers, prior art also includes programming language support for controlling precision [38] and verifying quantitative reliability [45] in applications.

Aside from the criteria listed above, there are also several characteristics to consider when identifying appropriate kernels. First, it is important for the kernel to have a relatively small number of inputs and outputs. Kernels with large numbers of inputs/outputs will not only lead to larger NN

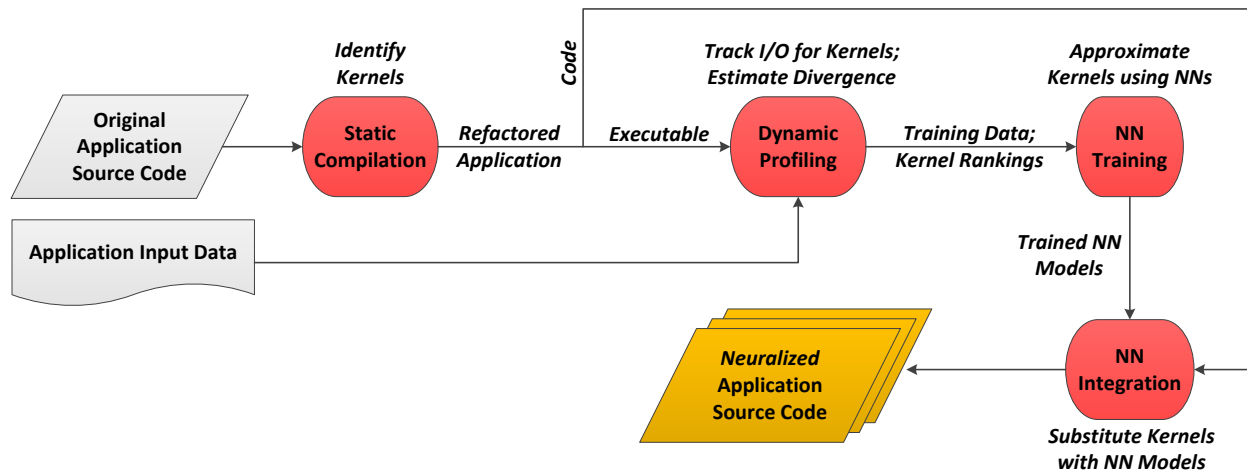


Figure 6.1: Overview of the *Neuralizer* software flow for automated neural acceleration of divergent applications

computations (and less of a performance improvement), but will also require more time for the NN training. An example of a non-ideal kernel, for instance, would be one that iterates over a large array and executes one trivial computation for each element. In this case, the overhead of passing the inputs through the NN would likely outweigh the benefits of the approximation. Similarly, another important characteristic to consider is the complexity of the relation between the inputs and outputs of the kernel. If this relation is high-dimensional, one would require an NN large enough to handle the approximation of that complex functionality, and consequently, the large NN would be much more difficult to train with an acceptable error rate. The reason for this is that more hidden layers are required for learning higher-dimensional functionality, yet the iterative, locally-optimal algorithms commonly used for NN training, such as the backpropagation algorithm [28], break down as the number of hidden layers increases. While these characteristic-based considerations we have mentioned are not hard-set rules, they serve as valuable guidelines (e.g. when refining a large search space of kernels) for finding ideal regions to target with neural approximation.

6.2 Methodology

In existing acceleration schemes [14, 15, 27], the process of choosing which kernels to target is often conducted in an ad-hoc manner, and is difficult to optimize as the space of targetable kernels grows exponentially. By specifically targeting potentially divergent regions, we maintain the ability to achieve significant gains by addressing the most critical issue for SIMD architectures, yet we narrow the scope enough to allow for automation beginning with kernel identification. Various aspects of our software flow, the *Neuralizer*, are detailed in the following sections. To aid the process of creating effective neural approximations, we also discuss novel optimization techniques for improving ANN performance and accuracy. As previously mentioned, this overall approach is platform-agnostic; however, for the purposes of this work, we examine branch divergence within the scope of a GPU. As such, our software flow leverages the CUDA-enabled ROSE [112] compiler infrastructure.

6.2.1 Neuralizer

The *Neuralizer* is a software flow that operates offline; this tool-chain automates kernel identification, divergence estimation, training data collection, NN training, and NN integration, ultimately generating a set of *neuralized* versions of a given application. For this process, the programmer is only responsible for determining the approximability of the code. Figure 6.1 is a high-level illustration of this flow.

Static Compilation. The *neuralization* process begins by accepting an application’s source code and performing static, compiler-based analysis to identify eligible, potentially-divergent kernels. At first, the compiler marks all regions of control flow as kernel candidates. It then eliminates kernel candidates based on the criteria described in Section 6.1. Specifically, the compiler checks for purity and fixed-size I/O using internally-generated data flow graphs, and uses programmer annotations to ensure approximability. After identifying eligible kernels, the compiler refactors the application’s source code, converting each kernel into a function with well-defined I/O (similar to the processing

of OpenMP pragmas). The compiler also instruments the source code to enable profiler-driven probing of I/O values of the kernels. Once refactoring and instrumentation are complete, the code is compiled for the given SIMD platform.

Dynamic Profiling. The next step of the *Neuralizer* involves dynamic profiling. The profiler receives the compiled executable corresponding to the refactored, instrumented source code, and runs it using real input data of the application. Leveraging the statically instrumented I/O probes, the profiler collects the I/O values of each kernel. These I/O values form each kernel’s respective data set, and are later used for training NN approximations of the given kernels. Also, this tool gauges the amount of divergence in each kernel (represented by the percentage of thread instructions that were not executed by all threads in the warp), and ranks the kernels based on decreasing amounts of divergence.

NN Training. Using the data sets collected by the dynamic profiler (one data set per kernel), our software flow proceeds by training NNs to approximate the kernels, and outputs the single “best” NN for each kernel. Details regarding the modeling of NNs are provided in Section 6.2.2. If an excessive number of kernels have been identified, the length of the training process can be shortened by trimming the list of kernels based on the rankings provided by the dynamic profiler. For instance, in our scheme we include the top ten most divergent kernels.

NN Integration. The final stage of the *Neuralizer* integrates the mathematical representations of the trained NNs directly into the refactored source code, replacing the potentially-divergent kernels they approximate. For details regarding the integration of NNs into applications, please refer to Section 6.2.3. An example of kernel-to-NN conversion is shown for sample code in Figure 6.2B using the MLP model shown in Figure 6.2A.

In order to properly examine application-level quality-vs-performance tradeoffs, our software flow does not simply replace all the kernel candidates of a given application; rather, it considers replacing various subsets of kernels with their corresponding NNs. Given a set of K kernels, there are 2^K possible kernel replacement combinations. Note that some kernels are nested within others, which invalidates a subset of the combinations. In addition, if there is an overabundance of combinations,

ones with low cumulative kernel rankings can be eliminated. Ultimately, this software flow outputs a set of *neuralized* versions of a given application, each version having a unique combination of kernel replacements. The versions are compiled using a standard compiler for the given SIMD architecture, and performance gains and quality loss are evaluated for each. Performance gain depends on the size of the code regions being replaced by NNs, the total amount of neurons and neuron-connections for all NNs being used, and the amount of divergence being removed. This essentially means making a worthwhile trade as time and energy spent on divergence are substituted for time and energy spent evaluating the NNs. We determine the best *neuralized* version of a given application using the following speedup-to-error ratio:

$$\frac{\textit{Speedup}}{(\textit{Error}/\textit{Error_Threshold})} \tag{6.1}$$

Here, the speedup and error values correspond to the entire *neuralized* application, not individual kernels. Since error is not the same across different applications, this ratio also takes into account a user-defined error threshold set specifically for each application. This allows a user to define an acceptable range for the quality of results of the overall application, and to have this range control the kernel approximations being integrated into the application. To further augment this process, a static analysis approach, such as Rely [45], can be integrated with our tool-chain, allowing it to statically enforce acceptable error rates by restricting the search space of kernels and NN topologies.

Aside from requiring the programmer, who has knowledge of the code, to determine approximable regions (e.g. using code annotations or type qualifiers [38]), this software flow is entirely automated. Kernel identification and NN integration run on the order of seconds, incurring negligible overhead. Also, training-data collection and divergence estimation generally only require a few minutes. NN training, however, incurs relatively more overhead. Training time depends on both the number of input/output neurons and the size of the training data set. However, since the individual code kernels are independent, as are the NN topologies being explored, the training process can be effectively parallelized. Furthermore, while automatically selecting optimal combinations of kernel replacements is a non-trivial problem, exploring the various combinations is trivially parallelizable,

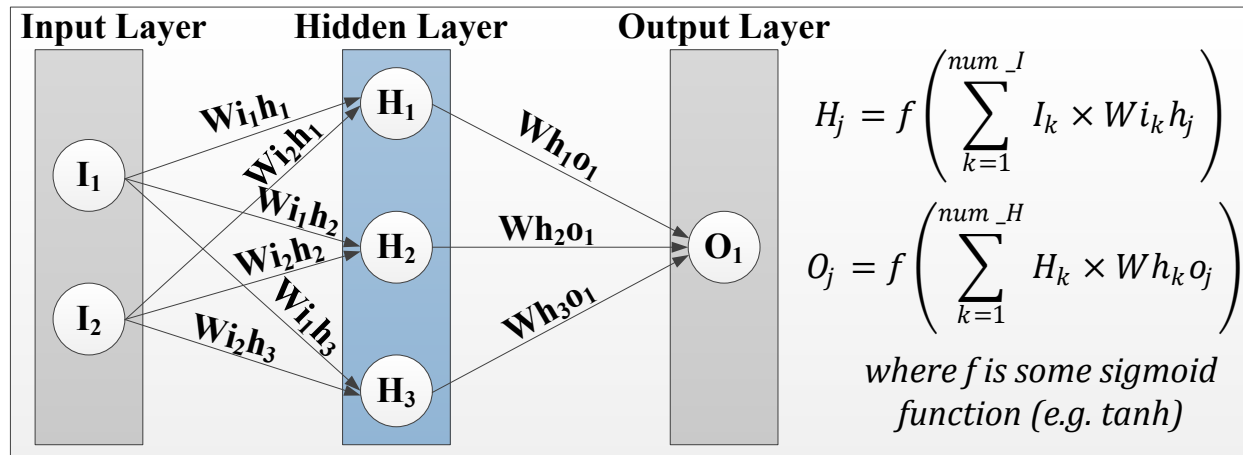
transforming this from a time-consuming process to a resource-intensive one.

On average, training a kernel for the given NN search space using fully parallelized execution on our 2.0 GHz Intel Quad-Core-i7 CPU has a duration of approximately 15 minutes. Additionally, since this training is an offline process, users may take advantage of libraries of pre-trained approximations for commonly used kernels. In terms of longevity, NN retraining is required only when the distribution of the original training data no longer represents that of user data, resulting in unacceptable quality of results. However, this does not mean the entire software flow would have to be rerun, particularly because the kernels of interest are likely to not change. As such, we would need to only rerun the dynamic profiler for the given kernels using the non-represented evaluation data, and retrain our NN with the newly consolidated training set.

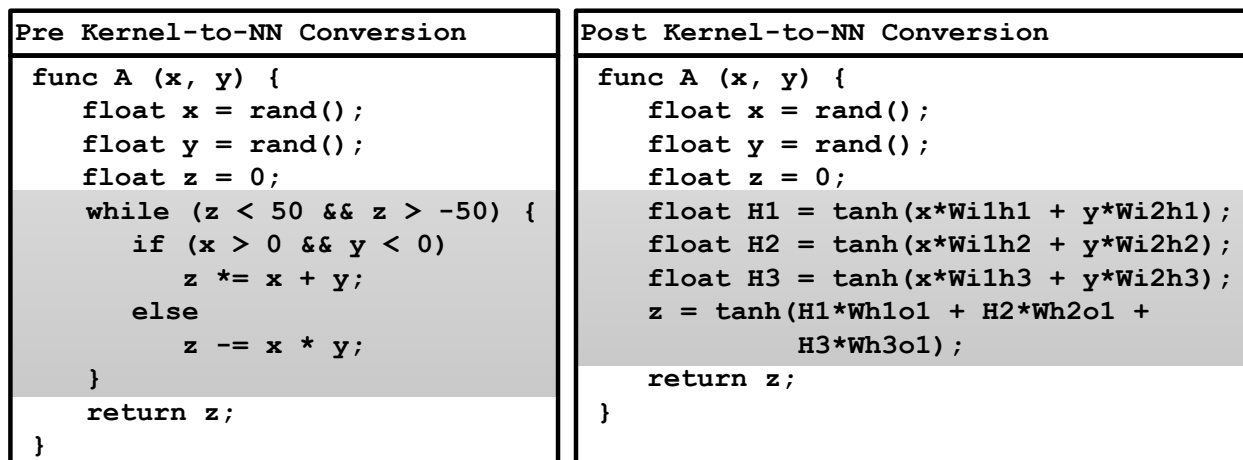
6.2.2 Details of NN Modeling and Training

We employ supervised learning via the conventional backpropagation algorithm [28] to train multilayer perceptrons (MLPs) [113]. Backpropagation is a gradient-descent-based function that iteratively adjusts edge weights of an NN, incrementally optimizing its solution. The MLP model is a feedforward network structured as an input layer, followed by one or more hidden layers, and finally an output layer. Functionally, the NN is evaluated using a series of weighted-sum and sigmoid operations without requiring any control flow. An example of a single-hidden-layer MLP (with labeled nodes and edge weights), along with the mathematical representation of its functionality, is shown in Figure 6.2A. We choose the MLP as our NN model not only because it is a simple network to manage, but also because of its flexible approximation capabilities, as described by the Universal Approximation Theorem [113].

Although a given kernel requires a fixed number of neurons in the input/output layers, we can search for optimal NN topologies by modifying the numbers of neurons in the hidden layers. Empirically, we have found that including three or more hidden layers generally results in excessive overhead without much gain in accuracy. Therefore, we limit our search to NNs with 1–2 hidden layers, and explore 1–32 neurons per layer (increasing by powers of two), resulting in an exploration space of 42



(A)



(B)

Figure 6.2: (A) Sample MLP and its mathematical representation based on labeled nodes and edge weights; (B) Example of integrating MLP of (A) into sample code (shaded regions represent the kernel-to-NN conversion)

topologies. We begin with the smallest NN topology, train it, and compute the cross-validated mean squared error (MSE) value (i.e. the MSE corresponding to a “test” subset of the kernel’s training data). We then incrementally explore larger topologies, saving those that minimize MSE. The best topology for a given kernel is determined as the smallest topology that achieves the minimal MSE, prioritizing accuracy over topology size.

For NN training to be successful, the distribution of the training data must be representative of the evaluation data; otherwise, no guarantees can be provided for the quality of the results [26].

The accuracy of an NN approximation also relies on the ability for the training process to find an acceptable mapping from inputs to outputs. However, if a mapping with sufficient accuracy requires too complex of an NN, this would not break our methodology – due to performance reasons, the required NN topology would merely fall outside the given search space. Though this would render the given kernel ineligible for approximation, other kernels within the same application would continue to be approximated by our software flow.

6.2.3 Details of NN Integration

We consider several important implementation decisions for integrating an NN into existing code. First, the sigmoid operation (the hyperbolic tangent function in our case) could be computed in one of two ways: (1) using a lookup table (LUT) with precomputed values [114], in which case only the address for the lookup would need to be computed instead of the entire sigmoid operation, or (2) computing the actual function (e.g. using a math library). In our scheme, we found that the *tanh* function from the CUDA math library [115] performed very efficiently on the GPU and even matched the performance of the LUT regardless of where we stored the LUT in memory. The reason for this is because memory accesses are very costly on a GPU, not to mention the need to still compute addresses for each table lookup.

The second important implementation decision is regarding the storage of weight and bias values of the NN. In our implementation, we statically integrate these values into the code, thereby having them stored as “immediate” values held in program memory. Other options include storage into shared or cached constant memory. While program memory may not be the best storage option for all hardware platforms, we found it to be the most efficient implementation for the GPU.

6.2.4 Optimization Techniques

The creation of an NN-based approximation that is both accurate and performance-wise effective is a nontrivial task. It is especially difficult when a generalized methodology is used to create NNs for a

variety of applications. For this purpose, we provide supplementary optimization techniques to help improve the accuracy of the NNs, as well as the performance and energy gains of the applications. As previously discussed, targeting divergent kernels is not a limitation of our approach – it is a way to intelligently guide kernel identification by focusing on the ultimate weakness of SIMD architectures. Once these main benefits have been reaped, further optimization can include subsumption of non-divergent code as well.

Our first optimization technique enlarges kernel scope, potentially allowing more divergent as well as non-divergent code to be encompassed by a single NN. This is done by modifying the kernel ranking criteria of the *Neuralizer* such that kernels are first prioritized by decreasing levels of scope before being prioritized by decreasing amounts of divergence. For example, a kernel with non-divergent control flow (e.g. deterministic “for” loop) encompassing a series of divergent control flow regions (e.g. data-dependent “if-else” statements) would be given priority over those individual divergent kernels. Allowing larger kernels to be subsumed by a given NN, we reduce the number of NNs needed, thereby increasing maximum potential gains. Also, we note that functional complexity is not directly related to kernel size, meaning larger scopes may even allow for more accurate NNs. An example of results from optimizing across different kernel scopes for the triangle intersection algorithm can be found in Section 6.4.3.

The second optimization technique we employ enables generation of better data sets for NN training. If the kernel is from an approximate algorithm (e.g. an iterative solver), we reverse engineer a data set with exact solutions to properly train the NN, potentially achieving a lower error rate than the original application. For instance, with our inverse kinematics benchmark (*invkin*, described in Section 6.3.1), we use *forward kinematics* to generate a “correct” data set to train the NN, providing a better representation of the input-output relation. With this optimization we see an average of 24% improvement in our training results and are even able to achieve lower error rates than the original applications (e.g. reducing error from 7.1% to 1.8% for inverse kinematics).

Our final optimization technique is for any kernel that can be described as a classifier (e.g. one that returns a boolean true/false value). For these kernels, the NN approximation can be augmented

Table 6.1: Summary of benchmark descriptions, domain categorizations, control flow characteristics, and justifications for approximability

Benchmark	Algorithm Description	App. Domain(s)	Control Flow	Approximability
invkin	Solves 3-joint inverse kinematics problem based on Cyclic Coordinate Descent (CCD)	Robotics, Graphics	Data-dependent multiple compute iterations	Heuristic-based algorithm: Iterative solver with no closed-form solution; not guaranteed to compute globally optimal result
nrpoly3	Uses Newton-Raphson method to find roots of a cubic polynomial	Nonlinear Algebra, Fluid Mechanics	Data-dependent multiple compute iterations	Heuristic-based algorithm: Iterative solver; not guaranteed to algorithmically converge
julfrac	Solves equations for points in a complex plane to generate Julia set fractals	Complex Dynamics	Data-dependent multiple compute iterations	Heuristic-based algorithm: Iterative solver; approximation of infinite fractals over discrete values
jmeint	Determines whether or not 2 given triangles intersect in 3D space	Graphics, Gaming, Computer Vision	Data-dependent single compute iteration (highly divergent)	Circumstantially approximable: E.g. output of <i>jmeint</i> could be used in approximate physics-based simulation
swaptions	Computes pricing for various stock options using Heath-Jarrow-Morton (HJM) framework	Finance	Data-dependent single compute iteration encompassed by non-data-dependent multiple compute iterations	Heuristic-based algorithm: Monte Carlo (MC) simulation used to generate inexact solution; region following MC output is approximated

Table 6.2: NN characteristics, training results, and benchmark evaluation results for neural approximations

Benchmark	NN Topology	Input Data Set *	NN MSE	Eval. Metric	Eval. Error
invkin	2 x 32 x 32 x 3	10000 sets of 2D coordinate values	0.0106%	Distance	1.75%
nrpoly3	5 x 2 x 0 x 1	16384 sets of cubic polynomial parameters and initial guess value	0.0139%	Avg. Relative Error	0.91%
julfrac	2 x 4 x 2 x 1	100x100 matrix of values	5.1071%	Miss Rate	6.56%
jmeint	18 x 1 x 0 x 1	102400 pairs of 3D triangle vertex coordinate values	7.4798%	Miss Rate	0.02%
swaptions	33 x 16 x 0 x 33	16384 sets of 33-element array values	0.7942%	RMS	8.79%

* Note: All inputs are represented as floating point values.

by filtering its outputs with a threshold-based classifier. For instance, the NN used for triangle intersection detection (i.e. *jmeint* benchmark described in Section 6.3.1) is supplemented with a stump classifier rooted at zero, improving the average error rate from 7% to 0.02%.

6.3 Evaluation Approach

6.3.1 Benchmarks and Evaluation Metrics

For evaluation purposes, we have selected divergent applications from a variety of domains. The benchmarks are selected because they (1) suffer from branch divergence, (2) potentially tolerate imprecision, and (3) could be beneficial for general applications. We purposely do not select from GPGPU benchmark suites because they primarily include compute-intensive workloads specifically optimized for GPUs (e.g. minimal control flow). We instead aim to enlarge the space of SIMD-targetable benchmarks. A summary of our benchmarks, which includes algorithm descriptions, control flow characterizations, and justifications for approximability, can be found in Table 6.1. These benchmarks originate from various sources [38, 53, 108, 116, 117] and have been converted to CUDA for execution on the GPU. Leveraging maximum data parallelism and memory coalescence, the CUDA-based GPU versions of these benchmarks run on average $20\times$ faster (minimum $6\times$, maximum $46\times$) than their C++-based, multi-threaded versions running on a 2.0 GHz Intel Quad-Core-i7 machine.

Since the type of final output varies across applications, evaluation metrics must be application-specific. The inverse kinematics benchmark (`invkin`) receives coordinate values for a target location and computes angle values for the three joints; to evaluate the error, we use forward kinematics to find the location of the end effector based on the computed angle values, and calculate its distance from the target location. With the Newton-Raphson method for finding roots of an equation (`nrpoly3`), we compare the outputted root value to the correct root value as an average relative error. Since the benchmarks for finding Julia set fractals (`julfrac`) and detecting triangle intersection (`jmeint`) each return a boolean value, error is evaluated as a miss rate. The final benchmark, `swaptions`, outputs arrays of values; as such, its evaluation metric is based on the root mean square (RMS) of array difference (much like how RMS of image difference is used for evaluating image processing). These evaluation metrics are reiterated in Table 6.2.

6.3.2 Experimental Setup

With respect to the generation, training, and testing of our NN models, we use the open-source, C-based Fast Artificial Neural Network (FANN) library [118] with support for floating point values. To allow for steady convergence of the backpropagation algorithm, we use a learning rate of 0.01 along with a maximum number of epochs of 5000. Also, while 75% of the benchmark’s real input data is used for collecting the NN training data sets for the kernels, the other 25% is used for post-*neuralization* evaluation of benchmark error. For performance and energy evaluations, our CUDA benchmarks are compiled with version 5.5 of the Nvidia CUDA compiler [119]; we then execute the benchmarks on an Nvidia GeForce GTX 480 GPU [120], which features 448 cores running at 607 MHz with 16 warps per block and 32 threads per warp.

Our performance metric is execution time, measured using the standard CUDA-based event timing constructs. For gauging power, we employ an electricity load monitoring device [121], which measures from a system’s main power source. We first measure power when the system is idle. Then, we run the GPU benchmark long enough for it to reach steady-state, remeasure the power, and take the difference between the two measurements. Energy is then computed as the product of this power result and the execution time from the performance result. Though our measurements could include dynamic power consumption of non-GPU components, these are negligible compared to the tens-to-hundreds of Watts consumed by the GPU. This form of direct power measurement also allows us to obtain more realistic results than possible with a simulator.

In our results, we compare the following schemes:

- **GPU**: Original GPU benchmarks
- **GPU_Ideal**: Non-divergent version of benchmarks (i.e. still include control flow, but have all threads process the same data values)
- **NN**: Benchmarks integrated with trained NN approximations
- **NN_Ideal**: Benchmarks integrated with zero-hidden-layer NN approximations

GPU_Ideal represents the best possible scenario for the GPU, where all threads not only fall within the same path of control flow (thereby eliminating divergence), they fall within the path that would result in the highest performance possible for the GPU; as such, *GPU_Ideal* represents a performance-wise optimal version of an approximation-based technique known as “branch herding” [122]. Also, the *GPU_Ideal* version of a benchmark is considered “ideal” because in reality, a user does not have control over data-divergence. Similarly, the zero-hidden-layer NN implementation presented with *NN_Ideal* is considered “ideal” because an MLP with no hidden layers does not have the capability to approximate any functionality and cannot realistically be used. Using these schemes, we are able to see the impact of divergence (*GPU_Ideal* vs. *GPU*), the benefit of using neural approximation to remove divergence while potentially subsuming non-divergent code (*NN* vs. *GPU_Ideal*), and the upper-bound on performance and energy gains for neural approximations (*NN_Ideal* vs. *NN*).

6.4 Experimental Results

6.4.1 NN Approximation

The characteristics of our NN-based approximations, along with application-specific evaluation metrics and error values, are summarized in Table 6.2. Note that “NN MSE” represents the cross-validated mean squared error of the NN, while the “Eval. Error” represents the application-level error assessment (i.e. based on the 25% of the benchmark inputs used for post-*neuralization* evaluation). For each of our benchmarks, optimal gains were achieved with the use of a single NN to subsume all of the divergent control flow. Our results, therefore, correspond to these single-NN configurations. In terms of the average evaluation error rates, all five of the benchmarks are well within 10%. To examine quality degradation in more detail, related work in approximate computing [27] uses a plot of the cumulative distribution function (CDF) of error in an application’s output. We adopt this same approach for analysis and provide a CDF plot of benchmark evaluation

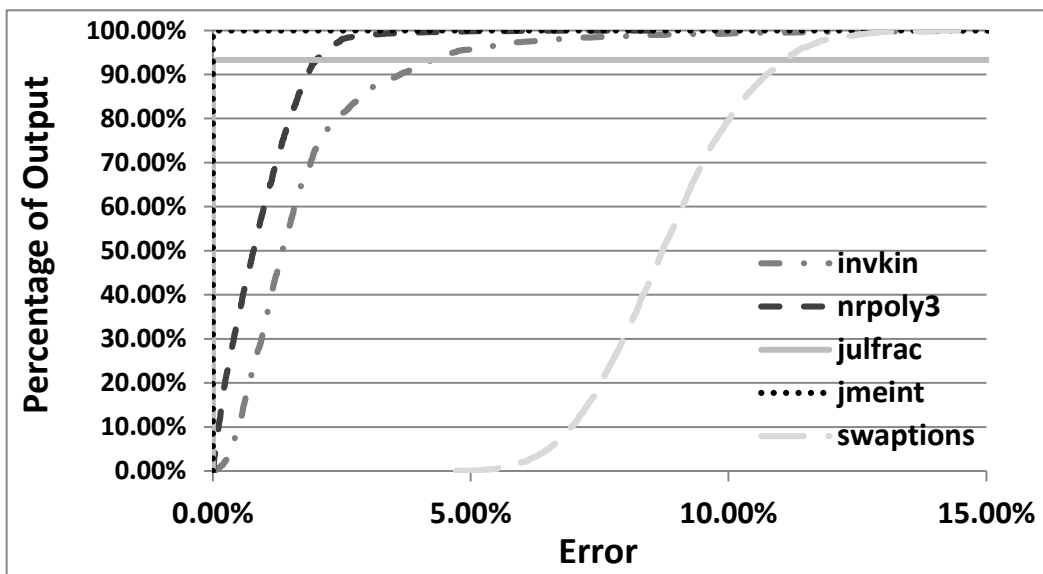


Figure 6.3: CDF plot of benchmark error; point (x, y) signifies that y percent of the outputs see x percent of error or less

error in Figure 6.3. This distribution reveals that 80%–100% of the outputs for all five benchmarks have less than 10% error.

While we do not claim the error rates demonstrated for our benchmarks are acceptable by all standards, we observe them to be on par with the range of quality loss seen in other approximation schemes [27, 34–36, 38, 40]. As with all approximate computing, statistically improbable errors could still render an application’s output meaningless. Different users may also have different notions of acceptable ranges of error for even the same application (e.g. `invkin` used for controlling robot-assisted surgery versus robot-assisted movement of large blocks); the user would therefore need to deem these approximations acceptable. For these reasons, we see an opportunity to combine our approach with existing mechanisms for online error validation and user-based error-threshold specification [123].

6.4.2 Performance and Energy Gains

Figures 6.4 and 6.5, which are normalized to the original *GPU* benchmarks, display our performance and energy results, respectively. We see that the three iterative, constraint-based solvers, namely

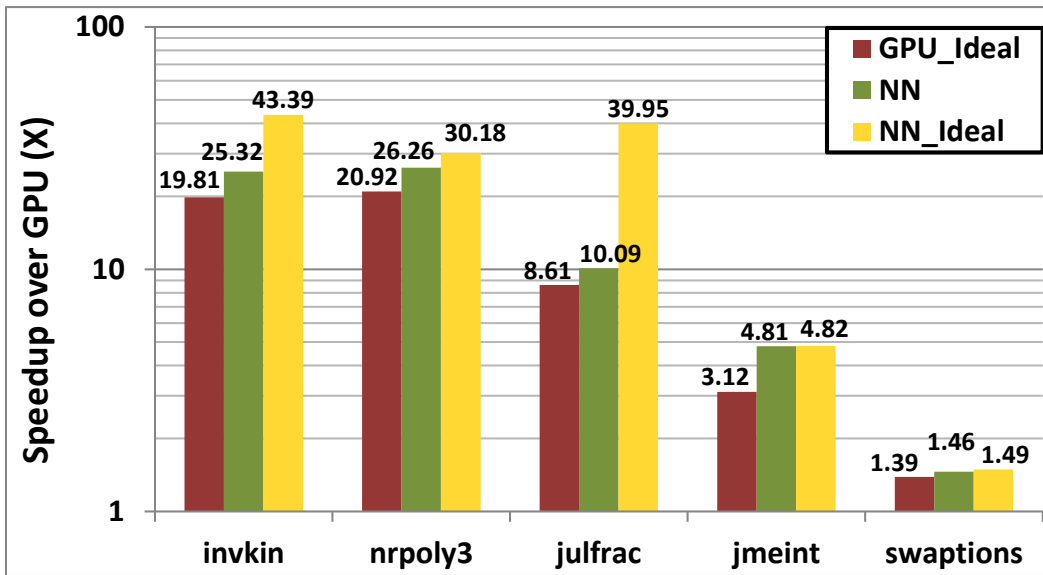


Figure 6.4: Performance gains of the various schemes

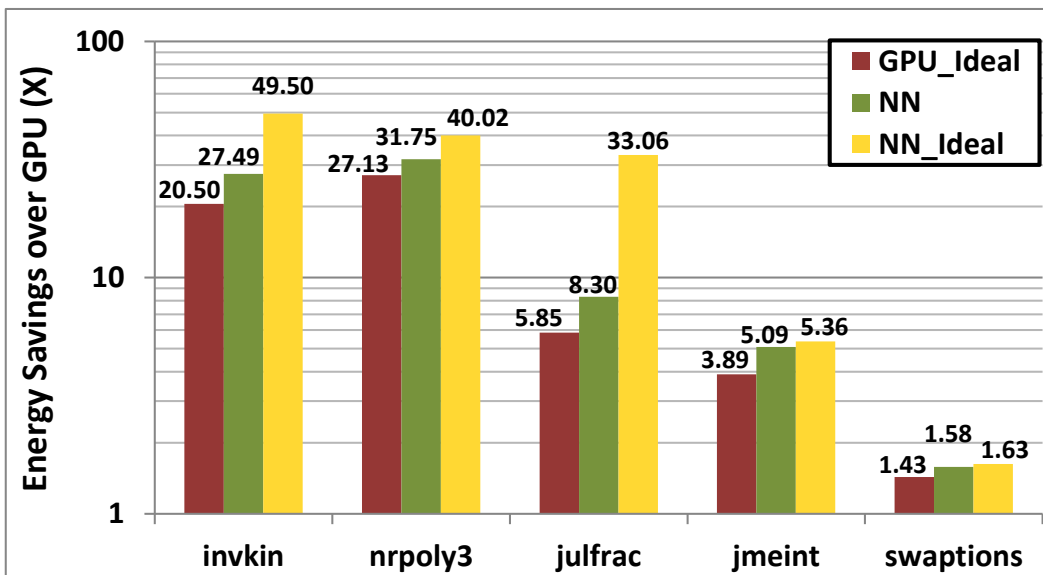


Figure 6.5: Energy savings of the various schemes

invkin, nrpoly3, and julfrac, have the highest speedups because of the extent of the divergence being converted to non-divergent computation. If divergence is removed (i.e. *GPU_Ideal* results), $9\times$ – $21\times$ speedup can be achieved. As a result of further reducing the number of instructions executed, the trained neural approximations (i.e. *NN*) achieve speedups of up to $26\times$. Although the *jmeint* benchmark lacks data-dependent loops, it still contains a significant amount of divergence

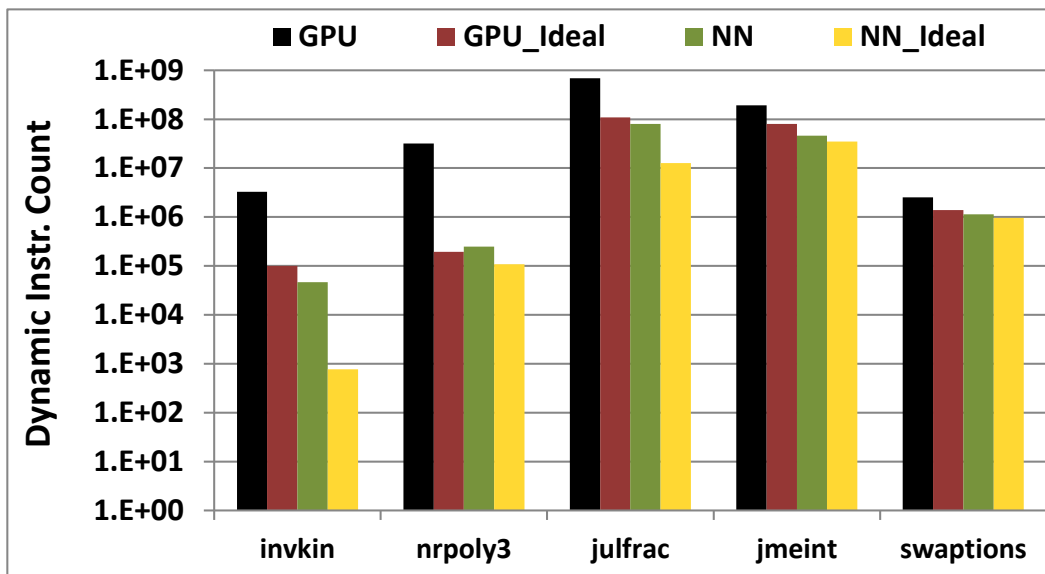


Figure 6.6: Dynamic instruction counts to verify the source of the performance and energy gains

due to data-dependent branches. As a result, performance improves by $3.1\times$ when the divergence is ideally removed, and neural approximation gains $4.8\times$ speedup, which is achievable because a very small NN topology (single hidden layer with one neuron) is able to satisfy this benchmark’s approximation requirements. Compared to the other benchmarks, `swaptions` has less divergence and requires a large NN with many input/output neurons; it therefore exhibits relatively modest improvements using neural approximation. In Figure 6.5, we see similar trends in energy savings, with $8\times$ – $32\times$ savings for the iterative solvers, $5\times$ for the highly-divergent triangle intersection detection, and $1.6\times$ for the many-input-output `swaptions` benchmark.

These performance and energy benefits are achieved with a combination of divergence elimination and reduction of dynamic instructions. For our benchmarks, we gauge the impact of divergence elimination using the “warp execution efficiency” profiling metric supported by CUDA; this metric measures the average percentage of active threads (i.e. threads performing useful work) in each executed warp. As expected, branch divergence leads to inefficient resource utilization and lowers warp execution efficiency. Our benchmarks originally have an average warp execution efficiency of 55%; after *neuralization*, however, the warp execution efficiency is transformed to 100% for all benchmarks. To further verify the source of our performance and energy gains, we include

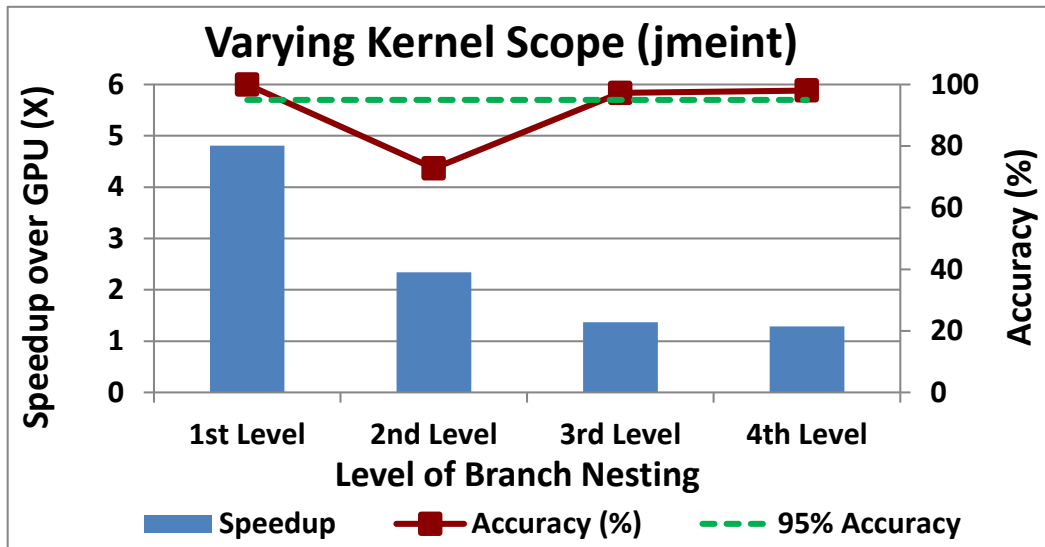


Figure 6.7: Performance gains and accuracy for `jmeint` benchmark as kernel scope varies

dynamic instruction counts in Figure 6.6. These results reveal a greater disparity between *GPU* and *GPU_Ideal* than between *GPU_Ideal* and *NN* for all the benchmarks, and in some cases (e.g. `nrpoly3`), *GPU_Ideal* may even execute fewer instructions than *NN*. In other words, while the amount of work in these applications certainly changes as instructions are subsumed by NN approximations, **divergence elimination is the key source of these gains** as it reduces the execution of unnecessary instructions and improves the efficiency of threads. This demonstrates the significance of changing the *nature* (and not just the *amount*) of the workload in these divergent applications. Furthermore, the gains of the NN implementations match closely with the upper bounds set by the idealized NNs for most benchmarks, thereby providing support for the effectiveness and low overhead of our technique.

6.4.3 Varying Kernel Scope

We now present results corresponding to the technique for exploring various levels of kernel scope. Figure 6.7 shows the results of using the same NN topology (one hidden layer with one neuron) to approximate the four different levels of scope for the `jmeint` benchmark (for an overview of the kernel scopes, please refer back to the pseudocode in Figure 1.2). The purpose of these results

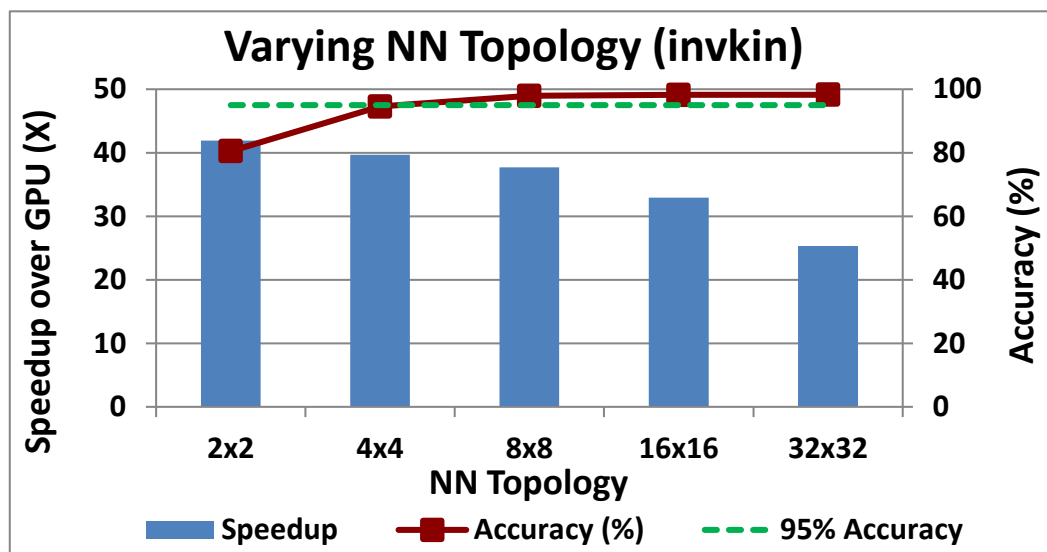


Figure 6.8: Performance gains and accuracy for `invkin` benchmark as NN topology varies

is to demonstrate how increasing/decreasing scope can impact the functional dimensionality of a kernel in application-specific ways. In other words, increasing scope does not necessarily increase dimensionality and decreasing scope does not necessarily have the opposite effect. For instance, we see from these results that the first level of branch nesting in fact has the highest accuracy when the same NN topology is used, thereby suggesting that its functionality is the easiest for the NN to learn. Furthermore, since it subsumes the largest portion of the benchmark, it has the highest potential for speedup. The second level of scope, conversely, diminishes in accuracy as its large number of inputs and high-dimensional functionality are too complex for the single neuron in the hidden layer to approximate accurately. The third and fourth levels of scope, however, reduce the functional dimensionality once again and achieve acceptable accuracy, though their speedups are limited as they subsume smaller portions of the benchmark.

6.4.4 Varying NN Topology

To demonstrate the need to explore the space of NN topologies for a given kernel, we present the results of varying the NN topology when approximating the `invkin` benchmark (Figure 6.8). Much like with the `nrpoly3` results presented earlier as motivation (Figure 1.4), we see the performance

gains reduce as the size of the NN increases. This figure also displays the benchmark evaluation accuracy corresponding to each NN topology used, and we see how (though it is not guaranteed to always be the case) a larger NN topology could likely result in higher approximation accuracy. For instance, in this case an accuracy within 95% could be achieved with a 4x4 or larger topology. Balancing these trade-offs, we find that a topology within the range of 4x4 and 8x8 (e.g. the 4x8 used for our final performance and energy results) would be ideal for this benchmark. As the performance of a given NN topology can vary drastically from one kernel to the next, we find that exploration of NN topologies is key in optimizing the gains achieved by neural approximations.

Chapter 7

Dynamically Reliable Approximate Computing Using Light-Weight Error Analysis

As a response to increasingly important concerns of power and energy consumption, along with lofty performance goals, computer architecture research has gravitated towards methodologies that provide inexact, yet acceptable solutions. Taking inspiration from techniques in fault tolerance, *approximate computing* [124] has evolved into a means for not only tolerating imprecision, but leveraging it as a trade-off for performance and energy gains. An “approximate accelerator”, or compute unit based on approximate hardware (e.g. low-precision functional unit) or software (e.g. algorithmic shortcuts), leverages imprecision tolerance (i.e. “slack” in computational accuracy) to improve performance and reduce energy consumption.

Prior art in approximate computing has extensively studied computational resilience to imprecision. However, existing approaches often rely on static techniques, which potentially compromise *coverage* and *reliability*. For achieving dynamic error analysis, it is unreasonable to compare an application’s exact output to the approximate output, as the generation of the exact output would merely add

overhead, defeating the purpose of the approximation. As a more high-level approach to quality analysis, we employ *light-weight checks* (LWCs).

The key inspiration behind this approach is that while finding a solution may be complex, checking the quality of that solution could be simple. By definition, an LWC is a metric that is **application-specific**, yet **algorithm-independent**, meaning it is not necessarily bound by a specific implementation. Although LWCs are unique to each application, they may be fairly easy to determine for certain categories of applications (e.g. problems that could be solved using iterative refinement). These metrics are also light-weight relative to the application, allowing them to be utilized dynamically instead of solely for the purpose of offline error analysis. The resulting methodology adapts to output quality at runtime, providing guarantees on worst-case application-level error. With this approach, we also decouple error analysis of the approximate accelerator from quality analysis of the overall application, allowing us to gain coverage by exploiting imprecision tolerance at the application level. To ensure platform agnosticism, these light-weight metrics are integrated directly into the application, enabling compatibility with any approximate acceleration technique. This chapter presents a case study of dynamic error control for inverse kinematics. Using software-based neural acceleration with LWC support, we demonstrate improvements in coverage, reliability, and overall performance.

7.1 Methodology

7.1.1 Light-Weight Checking

Static error analysis, such as establishing confidence intervals [46] or finding average error across a range of training values [27], is commonly employed to control the usage of approximate accelerators. In our approach, we advocate dynamic quality analysis and approximation control by way of LWCs. The key characteristics of LWCs are (1) application-specific quality assessment, and (2) light-weight evaluation relative to the application.

LWCs are not specific to any particular algorithm or implementation. They are only specific to the application itself. For instance, the application for physics-based simulation could involve many variations on algorithms for collision detection and constraint solving. However, an LWC for analyzing simulated scenes would remain relevant across the different implementations.

By nature, these LWCs could be based on values accessed from within the application (i.e. inputs, approximated outputs, and intermediate values), or via external values (e.g. additional sensor-based data). In a mobile robot application, for instance, an approximate accelerator used for augmenting the navigation program may be provided with supplemental sensory feedback, such as rangefinder data. This feedback may or may not be directly used for planning high-level navigation solutions, but it could be helpful in warning the robot about possible nearby collisions, rendering it an ideal LWC.

7.1.2 Deriving LWCs

We organize LWCs into three general categories based on the types of values being processed:

- (1) Approximate Output
- (2) Approximate Output + Input
- (3) Approximate Output + External Value

Though the exact output is never required, all LWCs must take into account the approximate output that has been produced. The first type of LWC, therefore, is one that relies only on the output. With applications based on iterative refinement, for instance, the quality of the computed output is continuously assessed in each iteration without the need for considering inputs or externally supplied values. However, another type of LWC may consider the output with respect to the input. For example, an LWC for physics-based simulation may gauge the quality of a simulation by comparing the input scene (i.e. the state of the simulation before a series of actions is taken) with the output scene. Finally, a third type of LWC is one that assesses output quality using external values (e.g. feedback provided by a supplementary sensor). We also note that intermediate values

of an application (i.e. non-input and non-output values) may be used in LWCs that fall in any of these categories, and do not give rise to a distinct category of their own.

Similar to existing methods for error control in approximate computing [34, 37], it is the responsibility of the user to define these light-weight quality metrics. However, while LWCs are application-specific, they may be found fairly easily for certain categories of applications. For instance, problems that could be solved using iterative refinement, such as inverse kinematics [116], may use the refinement criteria as an LWC. Likewise, an LWC for image quality assessment could be reused for various image processing applications, such as image enhancement, restoration, manipulation, etc.

With our organization of LWCs into categories, a user is able to more clearly identify the nature of the LWC they wish to derive, allowing for better visibility of overlap with similar LWCs. Furthermore, this categorization of LWCs may be used to establish a more rigid and detailed structure for users to follow when specifying an application. This structured specification could in turn be used by a compiler tool-chain for automatically extracting candidate LWCs.

7.1.3 Integrating LWCs

Once an LWC has been established for a given application, the LWC is integrated directly into the application code, providing a platform-agnostic mechanism for gauging output quality. Assuming the original code already contains an approximate computation, the modified code would need to subsequently evaluate the LWC, determine the QoS level achieved, and initiate recovery as needed. The overall structure of the LWC-integrated code is provided in Figure 7.1.

The application flow shown in Figure 7.1 is a fixed structure. However, the adaptive nature of our approach arises from the LWC and QoS constraint: the adjustable LWC allows the computational flow to accommodate user-specific definitions of quality, and the QoS constraint allows for adaptation to variations in data as well as variations in user preferences. Based on the structure of the LWC integration, we see that for each input the LWC is checked only once (not multiple times, as in incremental refinement [39]). Additionally, overall quality of the application is assessed

Structure of LWC-Integrated Application

- (1) Call approximate accelerator
 - (2) Evaluate LWC and determine QoS achieved
 - (3) *If QoS constraint is met*
 ➔ Continue to next input
 - (4) *Else*
 ➔ Recover by reprocessing current input with exact
 (non-approximated) version of computation
-

Figure 7.1: Integration of LWC into application code**Table 7.1:** Examples of applications, algorithms, domains, and LWCs

Application	Sample Algorithm	Application Domains	Light-Weight Check (LWC)
Inverse Kinematics	Cyclic Coordinate Descent Optimization	Robotics, Graphics, Gaming, Virtual Reality	Forward Kinematics
State Estimation	Kalman Filter	Navigation, Finance, Signal Processing	Measurement Comparison
Physics-Based Simulation	Gilbert-Johnson-Keerthi Distance	Gaming, Fluid Dynamics, Control Systems	Energy Conservation
Image Denoising	Total Variation Minimization	Computer Vision, Graphics, Medical Imaging	Universal Image Quality Index
SAT Solver	Davis-Putnam-Logemann-Loveland Search	Design Automation, AI Planning, Bioinformatics	Satisfiability Check

independently of the approximate accelerator, and recovery is dynamically initiated as necessary. While overall performance is undoubtedly affected by the performance and accuracy of the chosen approximate accelerator, it is also largely impacted by the specified QoS constraint and the nature of the inputs being processed.

7.1.4 Application Examples

Table 7.1 contains examples of applications and their corresponding LWCs. The following subsections describe these applications and LWCs in more detail.

7.1.4.1 Inverse Kinematics

Inverse kinematics based on cyclic coordinate descent optimization [116] is a well-known iterative algorithm for computing joint angles given a target location. This iterative technique is commonly applied in robotics (e.g. robotic arm motion) and graphics (e.g. character animation), especially for complex kinematic problems where analytical solutions are not always possible. By adjusting joints in a series of steps, this method continuously refines a solution until the end effector reaches the target within a given tolerance bound. At each iteration, the error in the solution is gauged using a quick forward kinematics computation. This forward kinematics check is inherently part of the iterative algorithm and is computationally simpler than inverse kinematics. It is therefore application-specific and light-weight, and serves as a valid LWC. Note that if the application entails hardware sensors for robotic motion, for example, sensory feedback would also serve as an LWC.

7.1.4.2 State Estimation

Another application that could benefit from LWC support is state estimation using the Kalman filter [67]. The Kalman filter is a recursive technique for state estimation of linear systems with applications in object tracking, localization, multimodal data fusion, exchange rate modeling, voltage estimation, and much more. Algorithmically, it includes a “prediction” step followed by an “update” step for comparing the prediction to actual measurements (e.g. from sensory feedback) and using a gain matrix to adjust the state accordingly. The primary purpose of the gain matrix is to optimally correct the state in order to minimize the difference between measurements and predictions. Though the computation of the gain is expensive, the measurement comparison is light-weight and may be leveraged as an LWC.

7.1.4.3 Physics-Based Simulation

With physics-based simulation, numerous algorithms could be used for object motion evaluation, collision detection, collision response, and constraint solving. However, regardless of those algorithms, the LWC metric would entail analysis of the simulated scene itself. Prior work in accelerating physics engines [32] has found energy conservation across scenes to be a useful metric for analyzing approximation error. Since solving for energy is considerably more light-weight than the entire simulation process, this metric could serve as an LWC.

7.1.4.4 Image Denoising

The image denoising application may have many different implementations, such as an implementation based on total variation minimization [90]. Quality assessment for the outputs of this application could rely on an overarching image analysis metric, such as the Universal Image Quality Index (UIQI) [125]. Similar to the Peak Signal-to-Noise Ratio (PSNR), which is commonly used for application-specific quality analysis, UIQI is generally applicable to images. UIQI is also easy to compute dynamically because it may serve as a comparison metric based on input-output images without the need for an exact output. In addition, unlike traditional error summation methods, UIQI models image distortion as a combination of loss of correlation, luminance distortion, and contrast distortion, allowing it to provide more meaningful comparisons across different images. UIQI could therefore be leveraged as a general-purpose LWC for image processing applications.

7.1.4.5 SAT Solver

Boolean satisfiability (SAT) is the problem of determining whether there exists a set of variable assignments that can satisfy a given propositional logic formula [126]. As a modeling framework for solving combinatorial problems, SAT has been widely used within many domains, including electronic design automation, artificial intelligence, and bioinformatics. Since SAT is an NP-complete problem, there is no known efficient solution for all of its possible instances. However, there are

numerous heuristic-based algorithms for solving SAT, many of which are implemented based on the seminal Davis-Putnam-Logemann-Loveland (DPLL) search algorithm [127]. Though solutions to the SAT problem are difficult to find, they are computationally cheap to verify with a simple “satisfiability check”, which entails plugging-in the solution and evaluating whether or not the logic formula is satisfied. As such, this satisfiability check is an ideal LWC.

7.2 Evaluation Approach

To demonstrate the benefits of dynamic quality analysis using LWCs, we examine inverse kinematics for a 3-joint arm. In this case study, the application continuously receives target x-y coordinates as input (e.g. for continuous robotic motion or character animation). Though the application operates on a non-data-parallel input stream, the computation for each input could be accelerated using approximate computing techniques. Also, the error tolerance threshold represents the maximum percentage of error the user is willing to accept for any given input, where error is measured as the distance (relative to the total length of the arm) from the end effector to the target location.

Though our approach is compatible with other approximation techniques, for the purposes of this work we exemplify benefits through software-based neural acceleration [128] (i.e. integrating a trained neural network directly into application code). To approximate inverse kinematics, a neural network (NN) is trained using 7500 input-output sets. Each set contains 2 inputs for the target x-y coordinates and 3 outputs for the joint angles; all values are expressed as floating point numbers. The NN has a total of 4 layers: 1 input layer with 2 neurons, 2 hidden layers with 8 neurons each, and 1 output layer with 3 neurons. For our performance evaluations, we use 1024 input-output sets, where the average error of our trained 8x8 NN is 4.1% with a standard deviation of 3.6% (i.e. variance of 0.13%). Note that it is important for the training and evaluation data sets to be completely distinct in order to ensure we are assessing the neural network’s ability to generalize rather than its ability to learn the training data.

Our experiments are run on a 2.0 GHz quad-core Intel Xeon E5405 processor. In our results, we compare the following evaluation schemes:

- **ORIG_1%**: Original benchmark with 1% set threshold
- **ORIG_n%**: Original benchmark with user-specified adjustable threshold “n”, which represents QoS as the maximum error being tolerated
- **ACC+LWC**: Benchmark integrated with NN and LWC, where the LWC is used to determine when to make use of exact computation (i.e. revert to *ORIG_n%* computation)
- **ACC-LWC**: Benchmark integrated with NN, but no LWC is used; recovery is therefore never initiated and the approximate solution is always employed, regardless of error

Note that, as iterative inverse kinematics repeatedly refines its solution to match a given error threshold, the original benchmark could either be statically set to an “acceptable” threshold (i.e. *ORIG_1%*), or could have the threshold be adjustable based on user specification (i.e. *ORIG_n%*), which resembles a software-based implementation of the *incremental refinement* technique [39].

7.3 Experimental Results

7.3.1 Performance

Performance corresponding to each of the schemes described in Section 7.2 is presented in Figure 7.2. On average, compared to the original benchmark with a set threshold of 1% (*ORIG_1%*), we see speedups (i.e. reductions in execution time) of $3.4\times$ for software-based incremental refinement (*ORIG_n%*), $28.6\times$ for neural acceleration with LWC support (*ACC+LWC*), and $44\times$ for neural acceleration without LWC support (*ACC-LWC*).

Approximation via software-based incremental refinement (*ORIG_n%*) and approximation with LWC support (*ACC+LWC*) are comparable techniques because they are both platform-agnostic approaches for dynamically ensuring QoS. Our results for inverse kinematics demonstrate an average

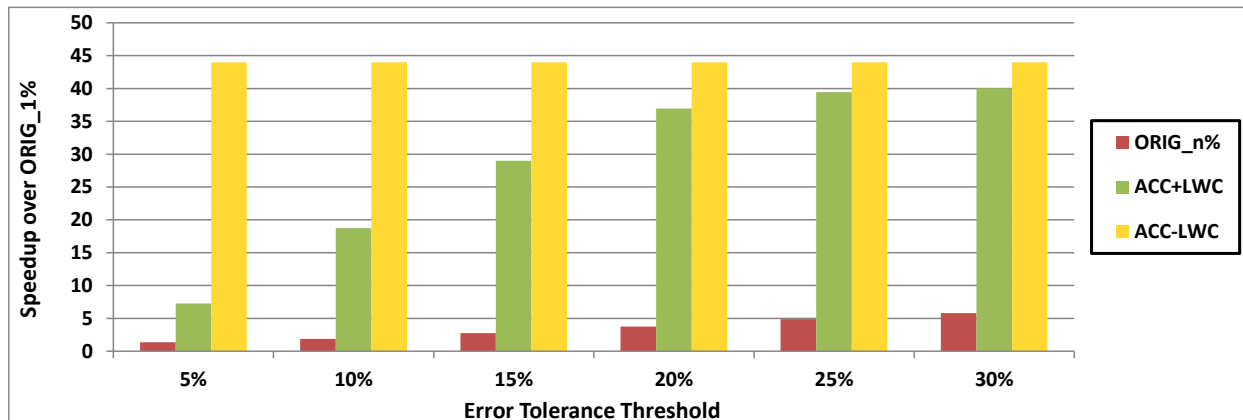


Figure 7.2: Performance comparison shown as speedup of $ORIG_n\%$, $ACC+LWC$, and $ACC-LWC$ over $ORIG_1\%$

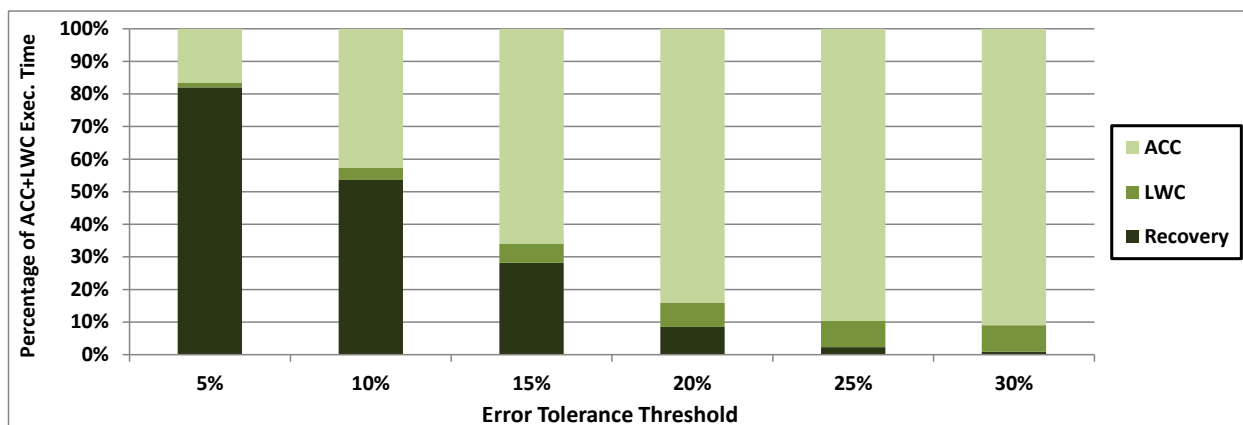


Figure 7.3: Performance breakdown for the $ACC+LWC$ scheme; execution time broken down into time for (1) computation of neural accelerator (ACC), (2) evaluation of light-weight check (LWC), and (3) recovery using exact computation (Recovery)

speedup of $8.5\times$ for $ACC+LWC$ relative to $ORIG_n\%$. Although the main source of the performance improvement for $ACC+LWC$ is the neural approximation, the efficiency of its dynamic error analysis is due to LWC support, which involves a single quick check for each approximation. The incremental refinement in $ORIG_n\%$, on the other hand, loses efficiency as it satisfies QoS constraints by continuously checking potential solutions at each iteration, incurring unnecessary costs that render the light-weight nature of its checks ineffective.

In Figure 7.2, $ACC-LWC$ provides a notion of maximum speedups that can be achieved by the approximation-based acceleration if worst-case accuracy is not guaranteed (i.e. LWC is excluded

and recovery is never initiated). From the given trends, we see the performance of *ACC+LWC* rapidly approaching that of *ACC-LWC* for tolerance thresholds ranging from 5%–30%, thereby emphasizing the light-weight nature of our dynamic error management approach. For a more detailed understanding of the performance of *ACC+LWC*, Figure 7.3 illustrates the percentages of execution time spent on (1) computation of the neural accelerator (ACC), (2) evaluation of the light-weight check (LWC), and (3) recovery using exact computation (Recovery). This performance breakdown reveals that the majority of the overhead preventing *ACC+LWC* from achieving *ACC-LWC* performance is caused by recovery stages. Though the need for recovery could be reduced by using more accurate approximation techniques, the higher accuracy would likely increase the execution time of the accelerator.

Excluding the overhead for recovery, the remaining overhead, which is for evaluating the LWC, is negligible (i.e. 6% on average). We note that as the error tolerance threshold increases, the amount of time spent on accelerator computation and LWC evaluation remains constant. However, their proportions of execution time increase because the total execution time decreases as the need for recovery is reduced. Moreover, though the overhead of an LWC will vary across different benchmarks due to its application-specific nature, it will remain beneficial for dynamic error analysis so long as it is light-weight relative to the overall application.

7.3.2 Reliability

As previously described, *ACC-LWC* lacks LWC support and an ability to initiate dynamic recovery, thereby demonstrating maximal speedups achievable by the given approximate accelerator. In this case, since dynamic analysis is not an option, static analysis would need to be used for understanding the output quality degradation resulting from the approximations. When the output quality of an approximate accelerator is statically analyzed, the assumption is that for the majority of executions, the approximation unit will have errors near its average error value (i.e. within a standard deviation given a normal distribution). Based on this assumption, it would be acceptable to use an approximate accelerator for an application if its average error falls within a given tolerance

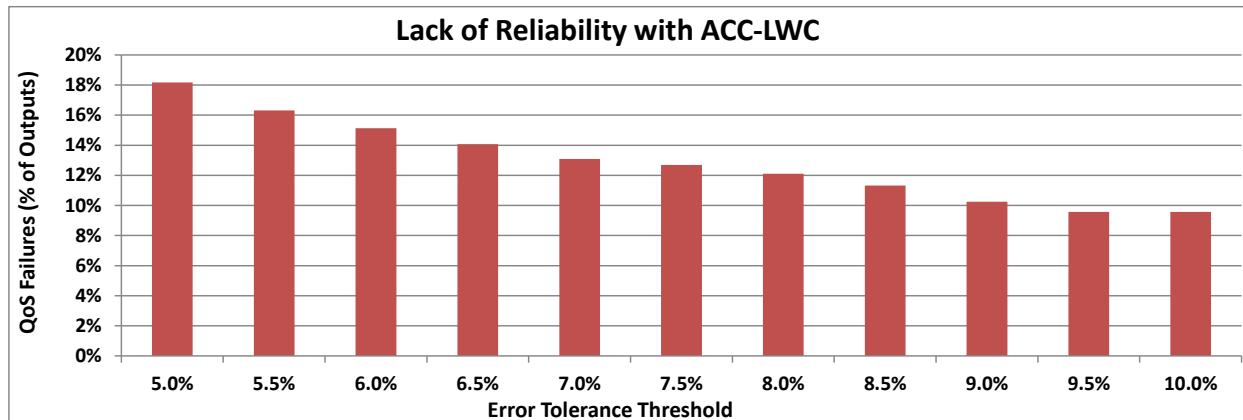


Figure 7.4: Reliability issues (shown as failure to satisfy QoS constraints) when approximating without LWC support

threshold, assuming the inputs also fall within an acceptable range (e.g. the range of the training data). However, though statistically less likely, there would still be cases where the error exceeds the threshold, resulting in a QoS failure. In Figure 7.4, we present the QoS failures that occur when an NN with an average error of 4.1% and a standard deviation of 3.6% is used for accelerating inverse kinematics with tolerance thresholds of 5%–10%. On average, we see QoS failures in 13% of the outputs. Notably, for thresholds of 8%–10%, which are past a standard deviation of the average error, there are QoS failures in 10% of the outputs, highlighting the unreliability of approximation without LWC support.

In addition, while the NN used in this example approximates the entire application, there may be cases where only a portion of the application is approximated. For instance, if the floating point operations are computed using an approximate floating point unit in hardware, this would result in approximation of various code regions throughout the application. For cases such as this, QoS guarantees are even less reliable if based on the approximation error of the accelerator (e.g. the accuracy of the approximate floating point unit), as even small errors may accumulate and result in unacceptably large quality loss in the overall application. We therefore advocate analyzing and guaranteeing QoS based on application-level quality metrics.

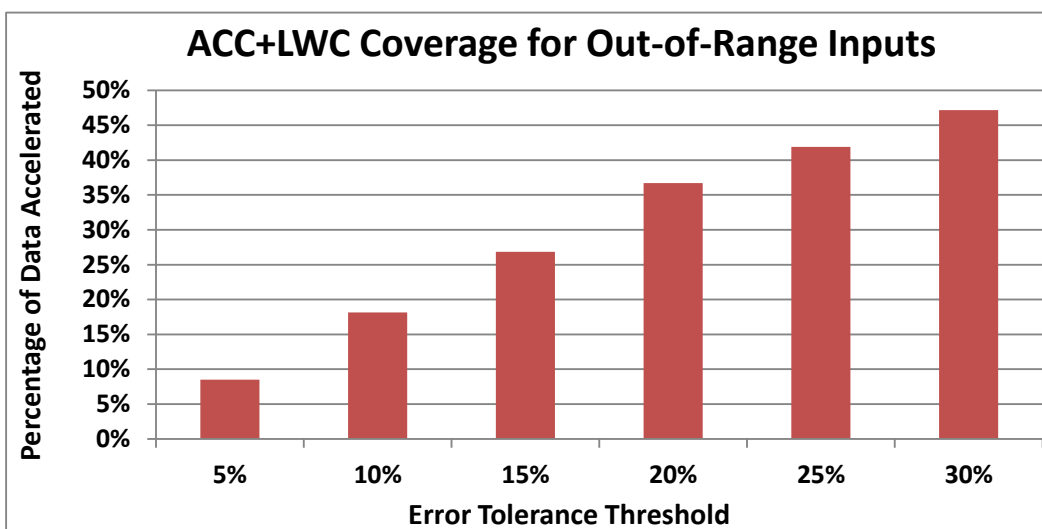


Figure 7.5: Amount of coverage for out-of-range inputs

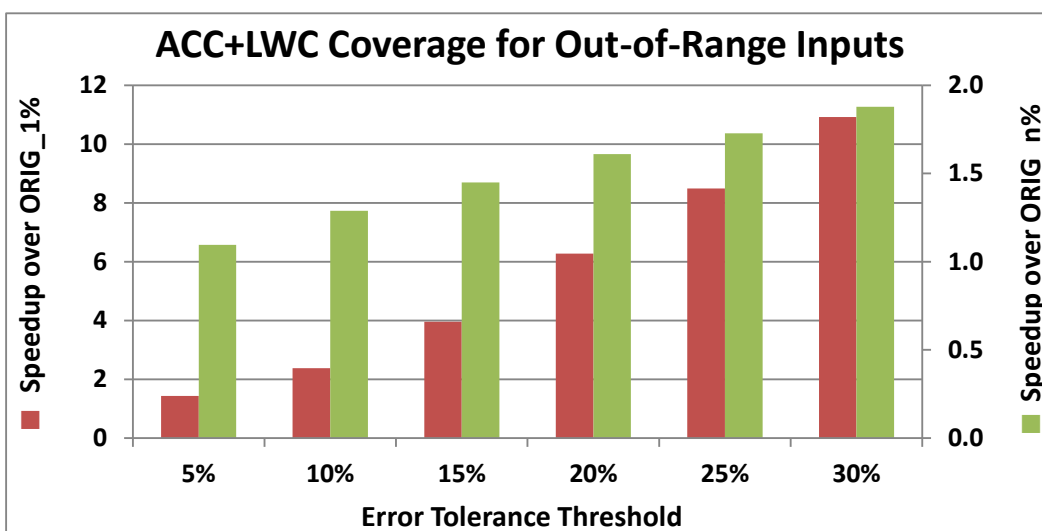


Figure 7.6: Speedup from coverage for out-of-range inputs

7.3.3 Coverage

Approximate accelerators are often unused when input data lies outside an acceptable range, such as the range of the training data for a neural accelerator [27]. This decision is based on the assumption that out-of-range inputs result in poor approximations. To demonstrate how this may be a costly oversight, we feed 1024 out-of-range inputs to our 8x8 NN, and reveal how 9%–47%

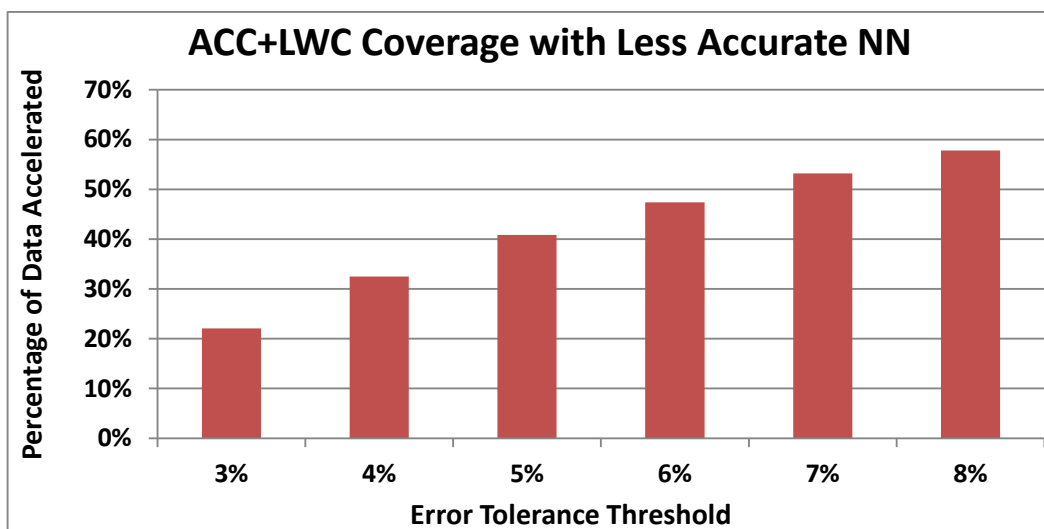


Figure 7.7: Amount of coverage with less accurate approximation

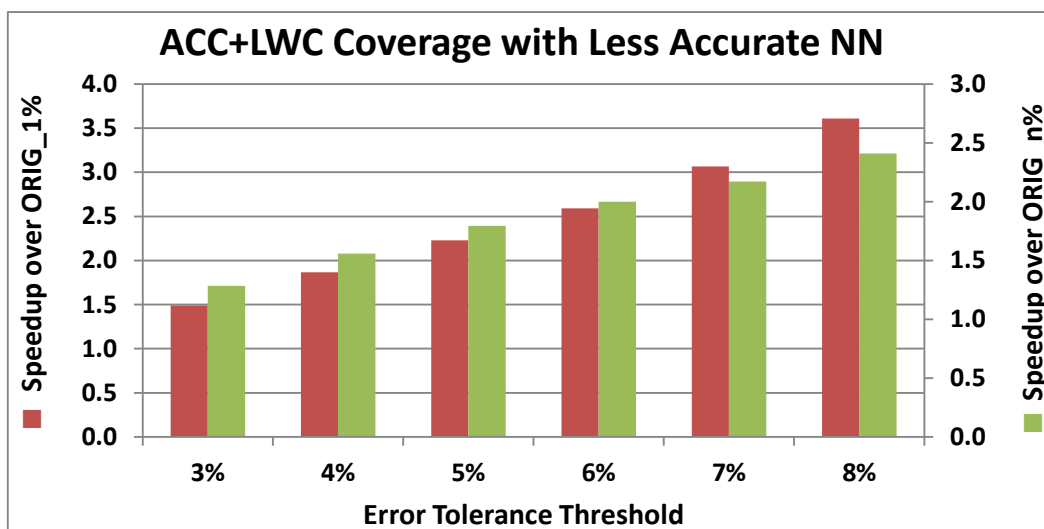


Figure 7.8: Speedup from coverage with less accurate approximation

of the data could be approximated acceptably for tolerance thresholds of 5%–30% (Figure 7.5), leading to average speedups of $5.6\times$ over *ORIG_1%* and $1.6\times$ over *ORIG_n%* (Figure 7.6).

Similarly, static techniques for error analysis (e.g. statistical models) disallow the use of an accelerator when its average error is above the tolerance threshold. Relying on the assumption that there would be too many cases of unacceptable quality loss, these static techniques lose coverage with portions of the data for which the low-accuracy accelerator could produce acceptable results.

For instance, we train a smaller NN (same as the previous NN, only the 2 hidden layers each have 4 neurons instead of 8) and find it has an average error of 8.9% (with a standard deviation of 11.7% and variance of 1.4%) for the evaluation data. Although this error exceeds tolerance thresholds of 3%–8%, significant portions of the data (i.e. 22%–58%) may still be reliably accelerated (Figure 7.7), resulting in average speedups of $2.5\times$ over *ORIG_1%* and $1.9\times$ over *ORIG_n%* (Figure 7.8). Furthermore, while these gains are with respect to an entirely software-based accelerator, efficient hardware-based approximate accelerators may be coupled with LWC support to yield even greater performance improvements.

7.4 Limitations

Our LWC-based scheme is generally applicable to the domain of approximate computing. However, this approach is fundamentally limited in two ways: (1) for a given application, a user must be able to define an LWC that is both expressive (i.e. in terms of adequately gauging output quality) and computationally efficient; (2) when integrated into a system based on approximate computing, the overhead of initiating recovery via exact computation must not outweigh the benefits of acceleration via approximation.

Though LWCs must be defined carefully, ideal ones can be readily extracted for many categories of applications. For instance, applications that can be implemented using iterative refinement inherently contain an LWC as the refinement criteria. Also, for NP-hard problems (e.g. boolean satisfiability, traveling salesman, bin-packing, etc.) in general, it is much more computationally difficult to *find* solutions than to *check* potential solutions. As such, implementations are often already embedded with light-weight checks that either guide the heuristic-based solvers or check the quality of given solutions. Nevertheless, it must be ensured that the overhead of computing the LWC does not dramatically diminish the gains achieved by approximation.

Chapter 8

BRAINIAC: Bringing Reliable Accuracy Into Neurally-Implemented Approximate Computing

Combining the previously discussed topics of this dissertation, we now introduce *BRAINIAC*: a heterogeneous accelerator-rich platform that combines conventional acceleration techniques with neural-network-based approximate computing. To gain maximum performance benefits and energy efficiency from our neural approximations, we shy away from a platform-agnostic approach. Instead, we opt for hardware-based implementations of the neural (approximate) accelerators, and we incorporate them into our platform alongside the conventional (precise) accelerators. Also, in order to reliably ensure acceptable quality of results, we implement dynamic error control based on light-weight checks (LWCs).

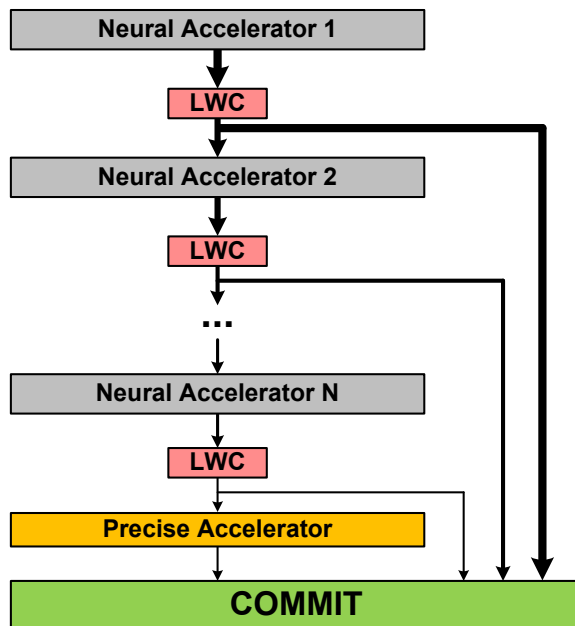


Figure 8.1: Multi-stage acceleration consisting of $N+1$ stages (N approximate computation stages, 1 precise computation stage)

8.1 Overview of BRAINIAC

The focus of our design is to gain performance benefits via approximation, while maintaining reliability via recovery. Our multi-stage acceleration technique, shown in Figure 8.1, aims to achieve this by combining light-weight checking with computations of varying accuracy. The flow of execution begins with weak approximations (which are simpler to compute) and continues onto stronger (more computationally complex) approximations as needed, ultimately ending with a precise computation. In order to ensure dynamic error control, we employ high-level, application-specific checks, similar to other runtime error detection schemes [123, 129]. The light-weight check, or LWC, computed between each stage is used to evaluate output quality. If the output produced for a given task is acceptable, the results are committed; otherwise, the task is recomputed by the next stage in the flow. Note that the LWC need not be evaluated for outputs of the final stage, which is the non-approximated version of the computation. This multi-stage flow is designed such that the approximations in the earlier stages would have enough coverage to produce significant performance gains and render the overhead of recovery negligible for the overall application execution.

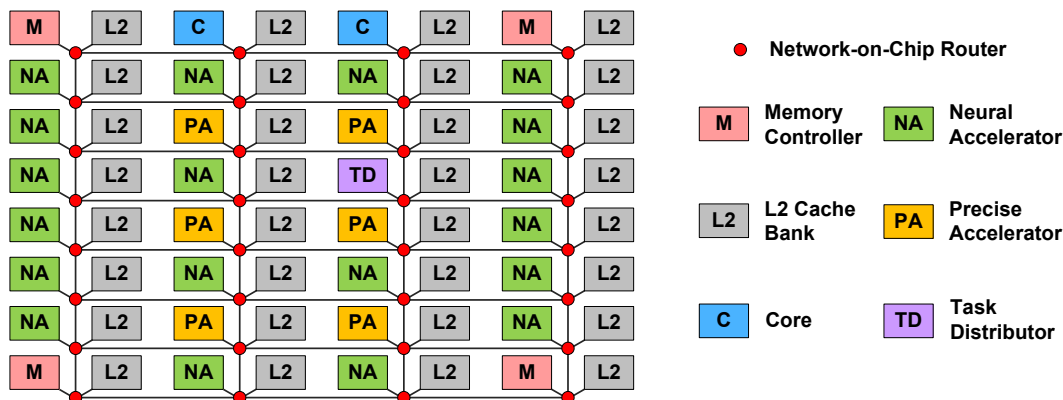


Figure 8.2: Microarchitecture of the *BRAINIAC* platform

In our system, the approximations are based on neural networks, and the precise computation is a monolithic accelerator that functionally emulates the original non-approximated code.

8.1.1 Hardware Infrastructure

The microarchitecture of our hybrid accelerator-rich platform is shown in Figure 8.2. It includes multiple cores and a task distributor (TD) surrounded by a sea of loosely-coupled accelerators and L2 cache banks. In our design, we advocate dynamic hardware-based accelerator management, which limits a core’s interaction with accelerators and removes it as a performance bottleneck. The design of our TD is based on the task distribution mechanism used in the *CHARM* architecture [15], and is not discussed in detail here. To enable our multi-stage acceleration flow, our TD and hardware drivers include new functionality to enable running batches of tasks asynchronously. The new functionality entails the following: (1) enabling multi-thread interrupts to be sent from a single software thread (i.e. provide one sub-thread per batch of tasks); (2) preparing the TD for asynchronous calls from software; (3) firing off batches of tasks; (4) queuing up batches in the TD in-order based on invocation time-stamps; (5) waiting on batches to be completed by checking necessary status flags. To avoid the complexity of integrating analog components or hardware with sub-critical voltage, we implement all our accelerators in digital ASIC technology. Section 8.2 provides more detail on the design of the neural accelerators.

8.1.2 Software Infrastructure

We implement our multi-stage acceleration scheme in software using hardware-supported asynchronous accelerator calls. However, we do not run each task with an independent accelerator call. Since loosely-coupled accelerators are optimized for streaming-data type of computation, running groups of tasks is much more efficient than running individual tasks. Furthermore, running tasks in groups avoids unnecessary reconfiguration of the neural accelerators and reduces overhead for handling interrupts. We therefore divide the tasks into a fixed number of groups (or “batches”) and employ batch-based execution.

At each approximation stage, once a given batch completes execution on the hardware accelerators, the results of each task in that batch are evaluated by an application-specific, software-based LWC. Tasks with acceptable quality of outputs are retired and their results are committed. Tasks with unacceptable results, on the other hand, are bundled into new batches and forwarded to the next acceleration stage to attempt recovery. Note that as the number of batches is fixed, the tasks are consolidated into as few batches as possible; once a batch has been filled with tasks that need to be run, the batch is fired off asynchronously. The number of batches is a design parameter that should be adjusted to balance trade-offs of excessive overhead (i.e. too many batches) vs. low accelerator utilization (i.e. too few batches). We have found a batch count of 32 to be an optimal design point for our system.

Each batch includes buffers that hold information on the type of accelerator needed for the tasks, as well as data for configuring the accelerator itself. Using this information, the accelerators are reconfigured at a granularity of a given number of tasks (e.g. we use a task granularity of 16). For more information on configuring the neural accelerators, refer to Section 8.2.2.

Our software infrastructure also includes tools for extracting code regions (or “kernels”), training neural accelerators, implementing accelerators (both precise and neural), synthesizing the accelerators in hardware, and simulating them for our platform. The tool-chain for generating neural network approximations profiles the original code kernels to extract input/output sets and uses this

data to train multilayer perceptrons, similar to the *Parrot Transformation* [27] and *Neuralizer* [128] processes. The other tools responsible for synthesis, implementation, and simulation are described in Section 8.3.

8.2 Neural Accelerator Design

The functionality of a given neural network (NN) is determined by its topology and the values associated with its network connections. As such, an NN can be “programmed” by simply updating the memory used for keeping track of connections, weights, and biases without needing to modify the functionality of compute engines. In other words, new functionality can be trained and instantiated without modifying hardware topology. This inherently adaptive nature of NN models enables us to use efficient ASIC technology and maintain design flexibility without resorting to programmable hardware fabrics (e.g. FPGAs).

Technology aside, however, several important issues remain when implementing our neural accelerators in hardware. Questions regarding composition, communication, and granularity present tradeoffs that must be explored and optimized, resulting in a variety of design choices for NN architectures. These design choices can be organized into three categories:

- (1) Monolithic network structure with fixed connections and fixed weights/biases
- (2) Composable network structure with flexible connections and flexible weights/biases
- (3) Monolithic network structure with fixed connections and flexible weights/biases

The first category (Figure 8.3A) is essentially the monolithic ASIC implementation of a code kernel’s NN approximation, which translates into a series of hardware-implemented weighted-sum and sigmoid operations. This option offers high efficiency in performance, yet virtually no flexibility. On the opposite end of the spectrum lies the second category (Figure 8.3B), which allows any set of fine-grained neural nodes to be connected in an arbitrary network formation with variable weights and biases associated with the connections. When realized in hardware, this entirely fluid structure closely resembles a sea of accelerator building blocks that are dynamically composed into

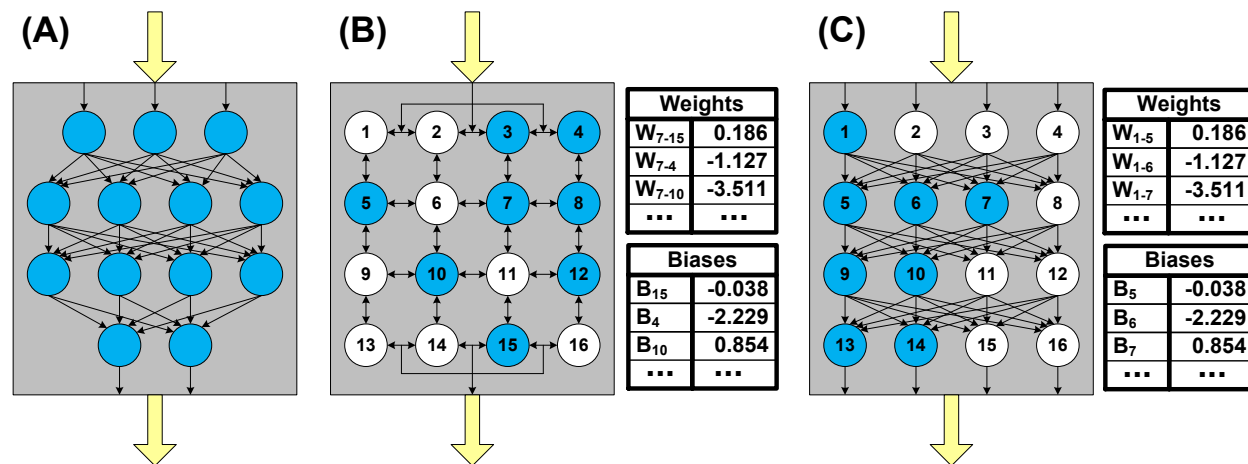
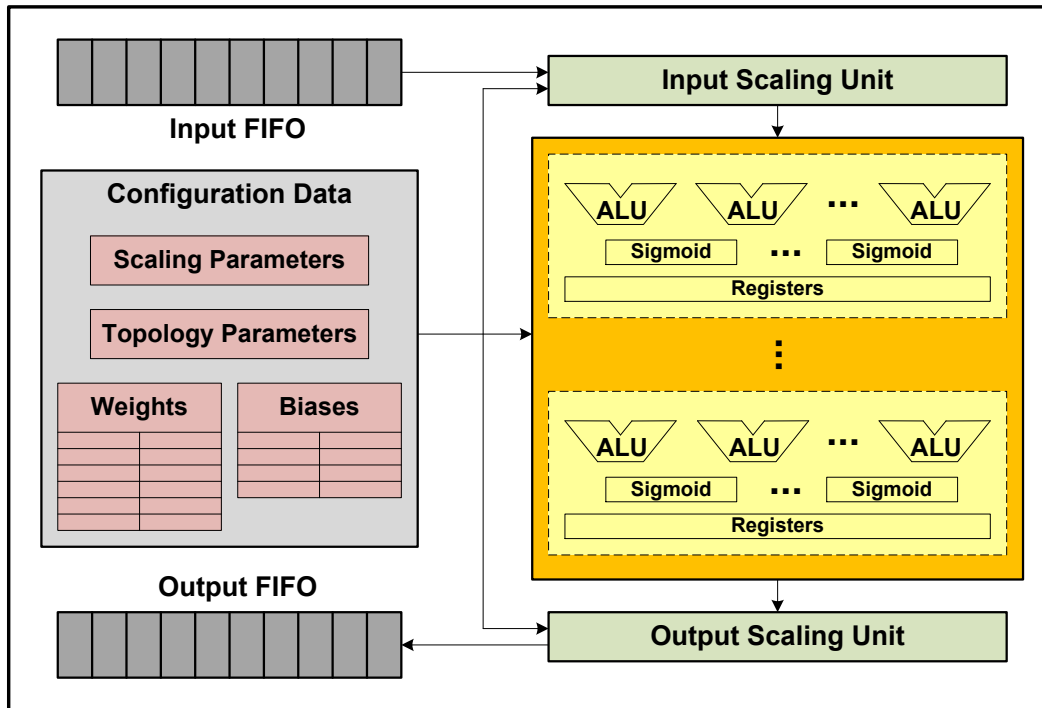


Figure 8.3: Three main categories of architectural designs for NN hardware implementation:
 (A) fixed connections, fixed weights/biases;
 (B) flexible connections, flexible weights/biases;
 (C) fixed connections, flexible weights/biases

virtual monolithic accelerators (i.e. the building blocks are neurons and the monolithic accelerator is a complete neural network). This architecture would allow for optimal load-balancing and resource utilization. However, it would incur maximum overhead for dynamic composition and packet-switched communication. The third category (Figure 8.3C) presents a compromise between the two aforementioned designs. The fixed network connections allow for direct circuit-switched communication between neurons, while the flexible weights/biases still achieve moderate adaptivity in composition. To leverage these benefits, we have designed our NAs based on this category.

8.2.1 Reconfigurable Neural Accelerator

In order to support NNs of varying topology at runtime, we design our NAs to be dynamically reconfigurable. As such, though our system features a homogeneous set of NAs, the reconfigurability allows them to be heterogeneous by nature. Unlike the tightly-coupled neural processors used in related work on neural acceleration [27], our NAs are loosely-coupled. This allows for computational tasks to be offloaded from the core to be processed on the NAs, much like with the PAs.



(A)

```

Reset prevNodeVals
Set currNodeVals to values of input neurons
biasIndex ← 0
weightIndex ← 0
layerIndex ← 1

while (layerIndex < MAX_LAYERS && nodesPerLayer[layerIndex] > 0) do
  Set prevNodeVals to currNodeVals
  Reset currNodeVals
  prevNodeCount ← nodesPerLayer[layerIndex - 1]
  currNodeCount ← nodesPerLayer[layerIndex]
  for (nodeIndex ← 0 to currNodeCount) do
    currNodeVals[nodeIndex] ← biases[biasIndex]
    biasIndex ← biasIndex + 1
    for (srcIndex ← 0 to prevNodeCount) do
      v ← prevNodeVals[srcIndex] * weights[weightIndex]
      currNodeVals[nodeIndex] ← currNodeVals[nodeIndex] + v
      weightIndex ← weightIndex + 1
    end for
    currNodeVals[nodeIndex] ← sigmoid(currNodeVals[nodeIndex])
  end for
end while
return currNodeVals

```

(B)

Figure 8.4: (A) Internal architecture of reconfigurable neural accelerator (RNA);
 (B) Pseudocode for functionality of computational pipeline within RNA

Table 8.1: RNA hardware parameters

Max Weights	Max Biases	Max Nodes Per Layer	Max Layers	Pipe Depth	Cycle Time (ns)	Min II	Power (uW)	Area (um²)
4096	128	32	4	926	1	8	18100	994270

The design of the reconfigurable neural accelerator (RNA) is illustrated in Figure 8.4A. This RNA design features a first-in-first-out buffer (FIFO) for receiving inputs, as well as a FIFO for sending outputs. The use of FIFOs allows for compatibility with varying numbers of inputs and outputs. The RNA also includes updatable look-up tables (LUTs) for storing weights and biases associated with the configured NN. The sigmoid operation, which is based on the hyperbolic tangent function, is also implemented using LUTs, yet these LUTs are configured statically. For evaluating outputs of a given NN, the required weighted-sum and sigmoid operations are carried out by the pipeline of combinational and sequential components. The functionality of this pipeline is summarized by the pseudocode provided in Figure 8.4B. Our RNA is designed to support 1 input layer, 1 output layer, and either 1 or 2 hidden layers; as such, the `MAX_LAYERS` parameter shown in the pseudocode would be equal to 4 in our case. As shown, the pipeline incrementally steps through each layer of the configured NN. For each neuron in the current layer, it collects the weighted-sum of the values from the neurons in the previous layer (i.e. `prevNodeVals`), applies the sigmoid function, and stores the value (i.e. in `currNodeVals`) to be used for processing the next layer. The final output is the set of values computed for the last layer of neurons (i.e. the final set of values stored into `currNodeVals`).

The corresponding hardware parameters for the RNA implementation are provided in Table 8.1. While the module runs at a constant 1.0 GHz frequency, the initiation interval (II) for each usage at runtime is determined by the amount of input and output data. For the given pipeline, the minimum II supported is equal to 8 cycles, which is for a range of 1–8 inputs and 1–8 outputs. Larger numbers of inputs and outputs result in longer initiation intervals at runtime.

8.2.2 RNA Configuration

As described in the previous section, the RNA structure can be adapted to evaluate any NN topology within the physical bounds of the hardware structure (e.g. without overflowing weight and bias buffers). In order to configure the correct functionality, the software infrastructure that makes use of these neural accelerators must correctly initialize the configuration data (shown in Figure 8.4A). This includes the following: (1) *scaling parameters* – values for scaling inputs down and scaling outputs back up; (2) *topology parameters* – specification of the number of layers and the number of neurons per layer (i.e. `nodesPerLayer` in Figure 8.4B) to allow the finite state machine controlling the computational pipeline to ensure correct evaluation of outputs; (3) *weights and biases* – values necessary for emulating the particular functionality of a given trained NN. Note that when configuring the weight and bias LUTs, the software passes in the values in a specific order; by knowing the topology of the NN, the hardware can then easily consume the weight and bias values in the same predefined order.

8.3 Evaluation Approach

8.3.1 Simulation Platform

In our experiments, we use the Simics [49] and GEMS [50] simulation frameworks, and extend them with the cycle-accurate models needed for our hybrid acceleration platform. The simulation parameters we use can be found in Table 8.2. We have also developed a complete tool-chain for generating simulator models starting from C-based code. The NN models, which need to also be trained and tested, are implemented using the open-source, C-based Fast Artificial Neural Network (FANN) library [118] (which includes support for floating point values). Table 8.3 lists the tools used for acquiring accurate timing and power values for all the hardware components we simulate.

Table 8.2: Simulation parameters

Parameter	Value
L2 cache	8 MB, 8-way set-associative, 32 banks; latency: 10 cycles
Coherence protocol	Shared banked L2 cache; L2: MOSI; L1: MSI
Main memory	Latency: 280 cycles; bandwidth: 10 B/cycle per controller
Network topology	4x8 mesh; link latency: 1 cycle; router latency: 5 cycles; bandwidth: 72 B/cycle per link

Table 8.3: Tools for timing and power models

Tool	Purpose
Xilinx Vivado Design Suite [106]	Accelerator high-level synthesis
Synopsys Design Compiler (32nm) [95]	ASIC synthesis (power, performance)
CACTI [100]	Cache memory modeling
Orion [102]	NoC power and area
McPat [80]	Core and cache power analysis

Table 8.4: Summary of benchmarks and LWCs

Benchmark	Algorithm	Light-Weight Check (LWC)
invkin	Solves 3-joint inverse kinematics problem based on Cyclic Coordinate Descent (CCD) algorithm	Forward Kinematics
nrpoly	Uses Newton-Raphson method to find roots of a cubic polynomial	Convergence Check
physim	Generates physics-based simulation of objects moving in 2D space	Energy Conservation
robloc	Performs robot localization using probabilistic model and particle filter	Sensor Measurement

Table 8.5: Summary of trained neural network models

Benchmark	Input Data	Evaluation Metric	NN Topology	Eval. Error
invkin	10000 sets of 2D coordinate values	Distance from end effector relative to arm length	NN1: 2 x 4 x 4 x 3	5.8%
			NN2: 2 x 8 x 8 x 3	3.4%
nrpoly	16384 sets of cubic polynomial parameters and initial guess values	Convergence test for iterative refinement	NN1: 5 x 4 x 0 x 1	5.9%
			NN2: 5 x 16 x 0 x 1	3.9%
physim	30000 sets of velocities, positions, and contact values for collisions	Average error (relative to maximum)	NN1: 11 x 8 x 0 x 4	5.4%
			NN2: 11 x 16 x 0 x 4	2.8%
robloc	1024 sets of quaternion and 3D acceleration values	Average error (relative to maximum)	NN1: 7 x 2 x 0 x 6	6.6%
			NN2: 7 x 16 x 0 x 6	3.3%

8.3.2 Benchmarks

The benchmarks used for our experimentation are applications for inverse kinematics (**invkin**) [116], the Newton-Raphson method (**nrpoly**) [53], physics-based simulation (**physim**) [130], and robot localization (**robloc**) [109]. Descriptions of these, along with their associated LWCs, can be found

in Table 8.4.

For the `invkin`, `nrpoly`, and `physim` benchmarks, the LWCs we employ are derived from the applications themselves. The LWC for inverse kinematics, for instance, is an implementation of forward kinematics to determine the location of the end effector of the 3-joint arm; the error would then be the distance from the end effector to the target location, relative to the length of the arm. Similarly, for physics-based simulation, we use energy conservation across simulated scenes as a metric for determining approximation error. Also, for the Newton-Raphson benchmark, the LWC is the refinement criteria (i.e. convergence check) of the iterative cyclic coordinate descent (CCD) algorithm used by the application. Unlike these other three benchmarks, with robot localization we emulate a hypothetical light-weight sensor, such as a proximity monitor or pressure sensor, that acts as a caution flag to force the system to execute more precise computation when navigating in the current environment. We configure this LWC as a module that, with a probability P , will generate a signal; this signal is a random number between 0 and 1, which is scaled by some value S and checked to see if it exceeds the error tolerance threshold. Within our benchmark, we use a value of 0.5 for both P and S .

For each benchmark, we have accelerated a single region that spans the majority of the application. Our implemented system employs a three-stage acceleration flow consisting of a weak neural approximation (NN1), a strong neural approximation (NN2), and a precise computation. The strong neural approximations have benchmark evaluation errors in the range of 0%–5%, while the weak neural approximations have errors in the range of 5%–10%. Depending on user-specified error tolerance, the use of a given NN may cause an average benchmark evaluation error to exceed the error threshold. Using that NN may nevertheless be beneficial, however, because the NN may still have significant coverage and be computationally simpler than the precise version of the computation; this performance benefit can be seen in our results (Section 8.4.1). The topologies and benchmark evaluation errors of our trained NNs are shown in Table 8.5.

Table 8.6: Power and area values of PAs and CNAs (NN1 and NN2)

Benchmark	Power (<i>uW</i>)			Area (<i>um</i> ²)		
	PA	NN1	NN2	PA	NN1	NN2
invkin	224000	216000	438000	15264739	12496073	25468980
nrpoly	10700	123000	404000	480396	6432129	23583138
physim	6610	422000	724000	267455	24273726	43125017
robloc	21300	185000	700000	962203	10690125	40512637

8.3.3 Evaluation Schemes

We evaluate power and performance benefits of our platform by comparing the following schemes:

- ***SW***: Original benchmarks run entirely in software
- ***PA***: Accelerated benchmarks that make use of precise accelerators, but not neural accelerators
- ***CNA***: Accelerated benchmarks that make use of both precise and neural accelerators; NAs are implemented as custom neural accelerators (CNAs) with fixed connections and fixed weights (i.e. based on design shown in Figure 8.3A)
- ***RNA***: Accelerated benchmarks that make use of both precise and neural accelerators; NAs are implemented as reconfigurable neural accelerators (RNAs) (i.e. based on design described in Section 8.2.1)

The software-only benchmarks are multi-threaded applications run on a 4-core 2.0 GHz Intel Xeon E5405 processor, while the hardware-based schemes are run on our simulation platform. We design our accelerator-rich platform such that it includes 4 of each precise accelerator (i.e. 16 in total) for the *PA* evaluation scheme. For the *CNA* scheme, we add 4 of each NA as well (i.e. 16 NN1 and 16 NN2). Note that the *CNA* scheme is meant to represent a performance-wise optimal yet unrealistic design that is not limited by power and area constraints. The *RNA* scheme, on the other hand, will be compared against *CNA* while being constrained to the total area utilized by the *PA* scheme. In designing the *RNA* scheme, we start with the *PA* platform, remove half of the PAs (i.e. we are left with 2 of each type, for a total of 8 PAs), and add enough RNAs such that the total area of the RNAs does not exceed the area of the PAs. The power and area values of each hardware-implemented PA and CNA can be found in Table 8.6. Given these area values, and an

RNA area of 994270 um^2 (refer to Section 8.2.1), we see that the RNA-based scheme includes 34 RNAs. We now compare performance and energy results across these different evaluation schemes.

8.4 Experimental Results

8.4.1 Performance and Energy

The performance improvements achieved by the 3 accelerator-based schemes (*PA*, *CNA*, and *RNA*) over the software-based scheme (*SW*) are shown in Figures 8.5, 8.6, 8.7, and 8.8. While the *PA* scheme achieves moderate performance benefits, the other two schemes are able to achieve even larger speedups by leveraging neural approximations whenever possible. Also, we see the gains achieved by *CNA* and *RNA* increase as the error tolerance threshold for each benchmark is increased, while the gains of *PA* remain static.

As previously noted, the *CNA* scheme includes custom neural accelerators that are computationally more efficient, yet require significantly more power and area. Though this system would be performance-wise ideal, it would be impractical to implement. In comparison to an RNA, a CNA can compute its designated NN eight times faster on average. However, CNAs are limited in their usage while the reconfigurability of RNAs allows all of them to be accessible to any of the applications. The RNA-based system therefore achieves higher resource utilization and increased parallel processing, as well as more evenly distributed workloads with fewer bottlenecks in the on-chip network. As a result, we see comparable performance between the *RNA* and *CNA* platforms.

On average, for error tolerance thresholds of 5%–25%, *PA* achieves performance gains of $9\times$, *CNA* achieves $22\times$ – $40\times$, and *RNA* achieves $15\times$ – $35\times$. The energy results (provided in Figures 8.9, 8.10, 8.11, and 8.12) follow trends similar to those of the performance results.

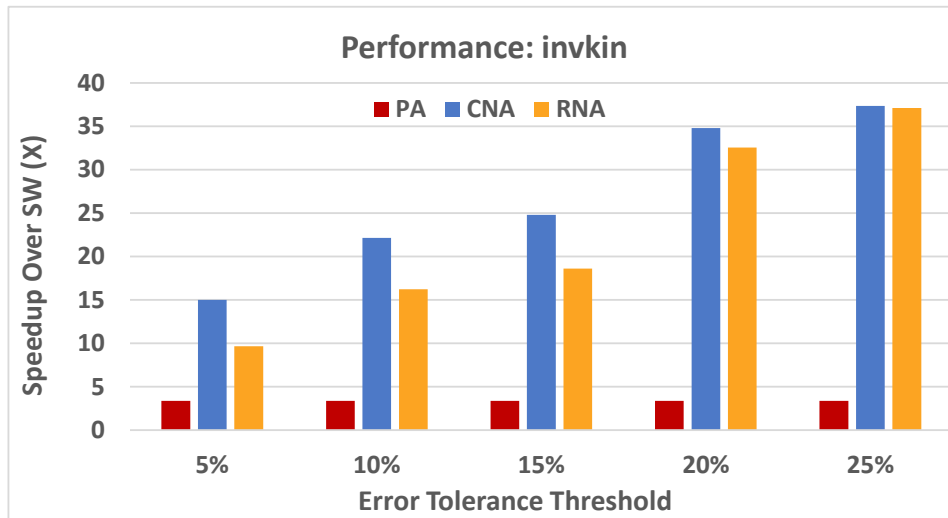


Figure 8.5: Performance results for `invkin` benchmark

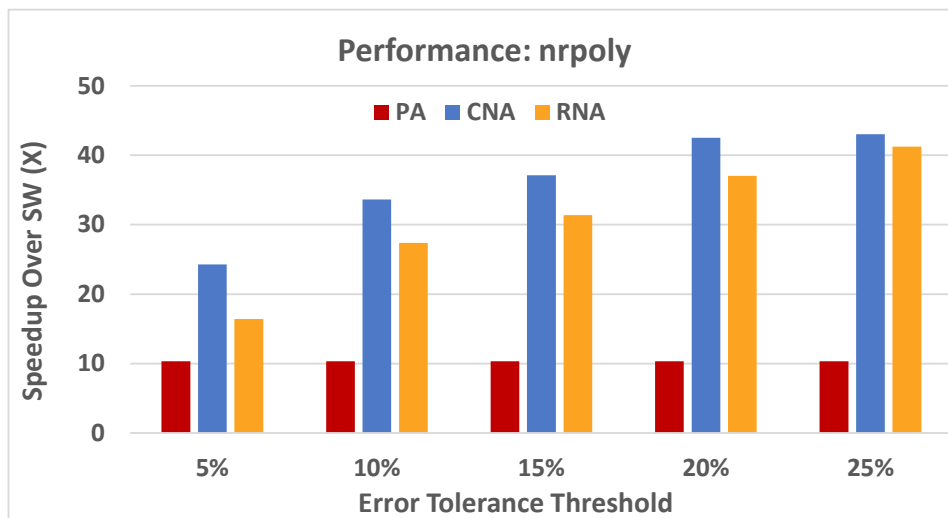


Figure 8.6: Performance results for `nrpoly` benchmark

8.4.2 Comparison to Loop Perforation

Loop perforation [40] is a common method of approximate computing used in applications with iterative computation. This technique uses a high-level convergence check (which is the same as our LWC for `invkin`) to break out from loops early, essentially cutting down on computation while adding minor overhead for the check. We compare our accelerator-based schemes to a software-based implementation of loop perforation for the `invkin` benchmark. As shown by the results

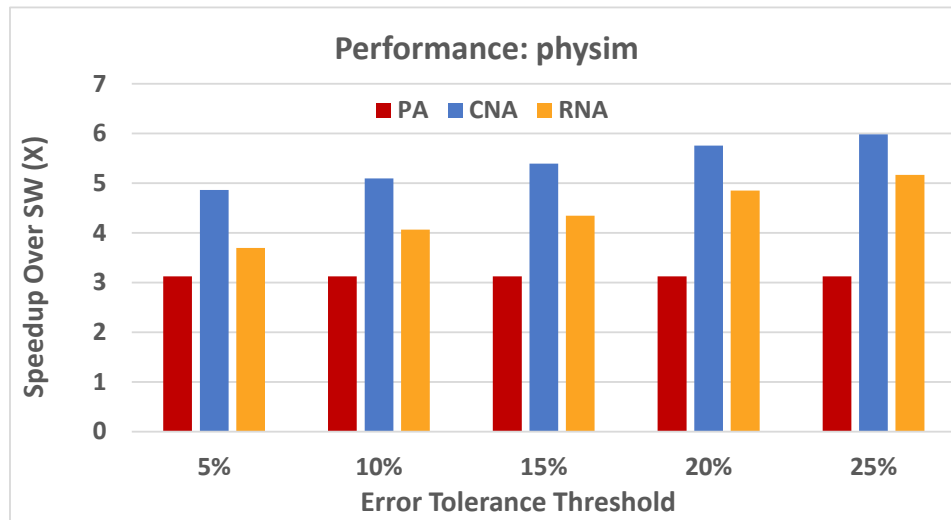


Figure 8.7: Performance results for *physim* benchmark

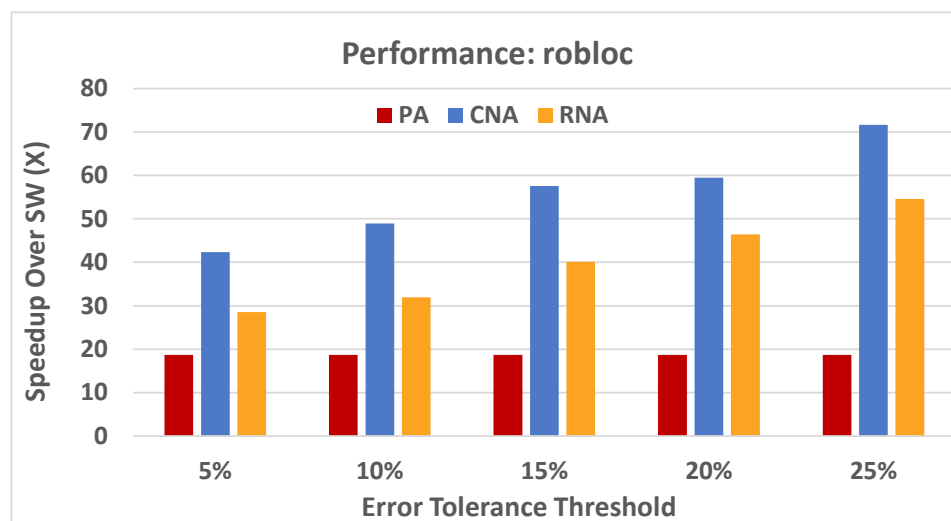
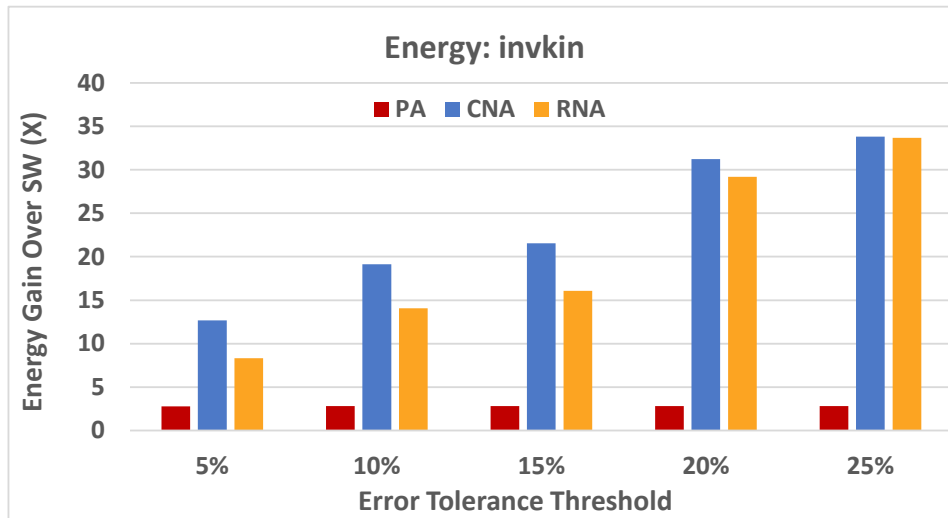
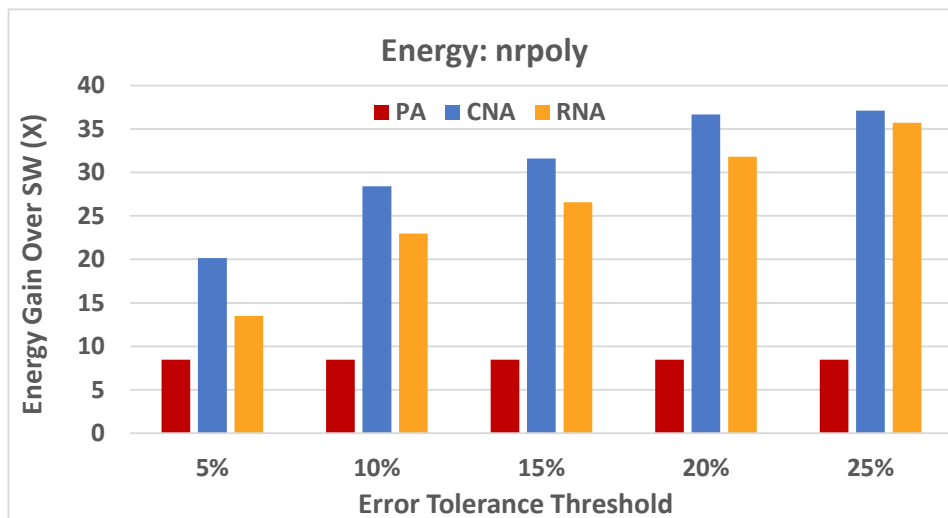


Figure 8.8: Performance results for *robloc* benchmark

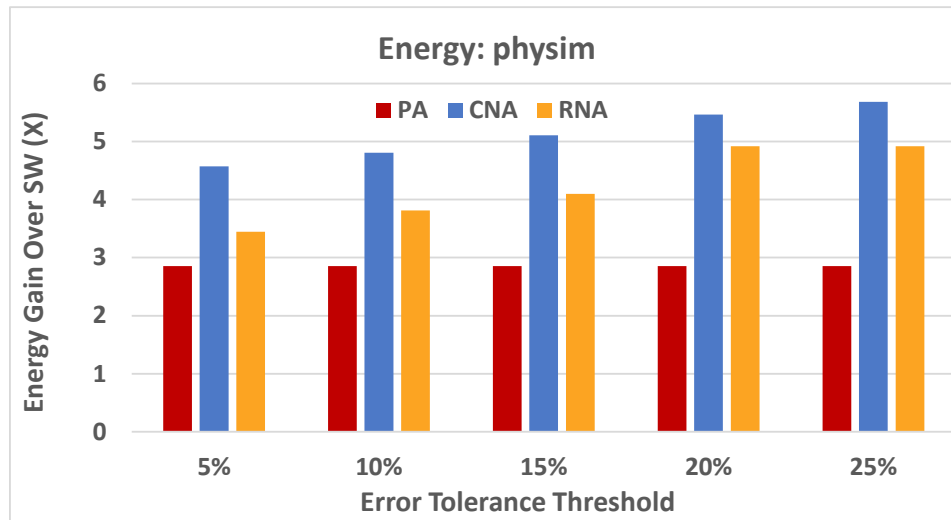
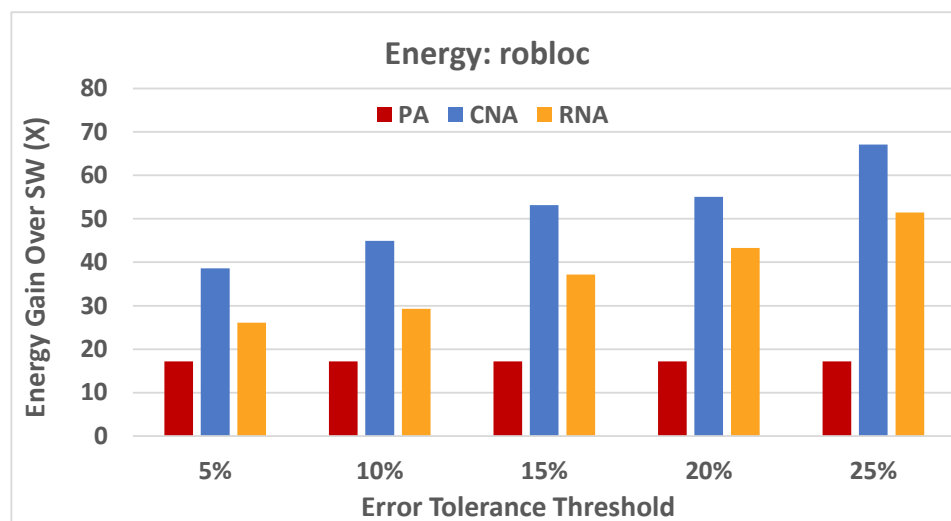
in Figures 8.13 and 8.14, the *CNA* and *RNA* schemes maintain speedups of over $6\times$ and energy savings of over $5\times$. The performance and energy gains of *PA*, however, diminish exponentially.

8.4.3 Performance Breakdown of RNA Scheme

We now discuss the breakdown of performance in terms of cycles and batches executed by the various stages of the multi-stage acceleration flow. All four benchmarks follow very similar trends;

**Figure 8.9:** Energy results for *invkin* benchmark**Figure 8.10:** Energy results for *nrpoly* benchmark

to avoid redundancy, we provide the performance breakdown for two of them: *invkin* and *nrpoly*. From the cycle execution results (Figures 8.15 and 8.16) we see that as error tolerance increases, PA and NN2 contribute to smaller portions of the total cycles executed while NN1 and LWC contribute to larger portions. Observing the batch execution results (Figures 8.17 and 8.18), we see similar trends for PA, NN2, and NN1. We also note that although NN1 contributes to less than 20% of the executed cycles, it is responsible for the majority of the computational work (i.e. over 55% of the batches that are executed).

**Figure 8.11:** Energy results for physim benchmark**Figure 8.12:** Energy results for robloc benchmark

8.4.4 RNA Configuration Overhead

The reconfigurability of RNAs is a key characteristic that allows them to achieve significant performance and energy gains while maintaining an implementation limited by practical power and area constraints. This reconfigurability, however, comes at a cost. We quantify the overall configuration overhead for the *RNA* scheme in Figure 8.19. This configuration overhead is largest for an error tolerance threshold of 5% because this is the scenario that requires the most batches to transition

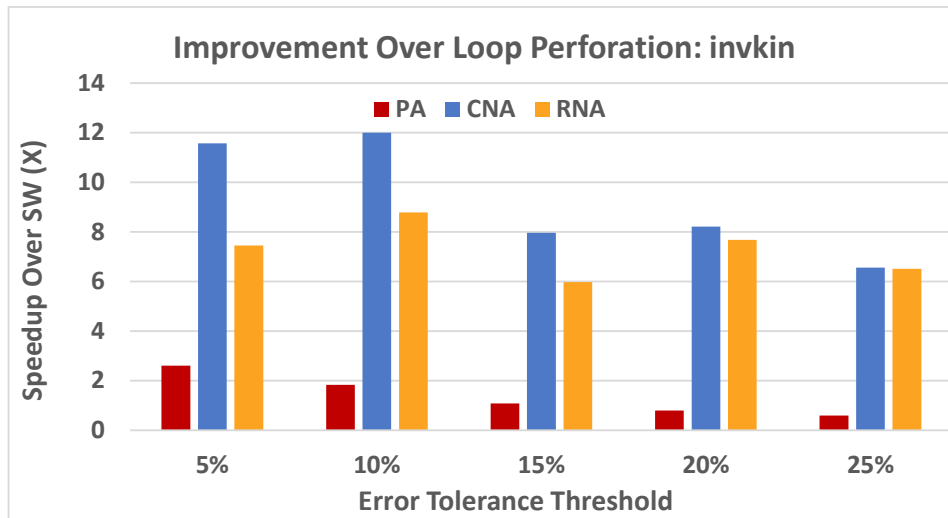


Figure 8.13: Performance results for hardware acceleration of `invkin` benchmark in comparison to loop perforation implemented in software

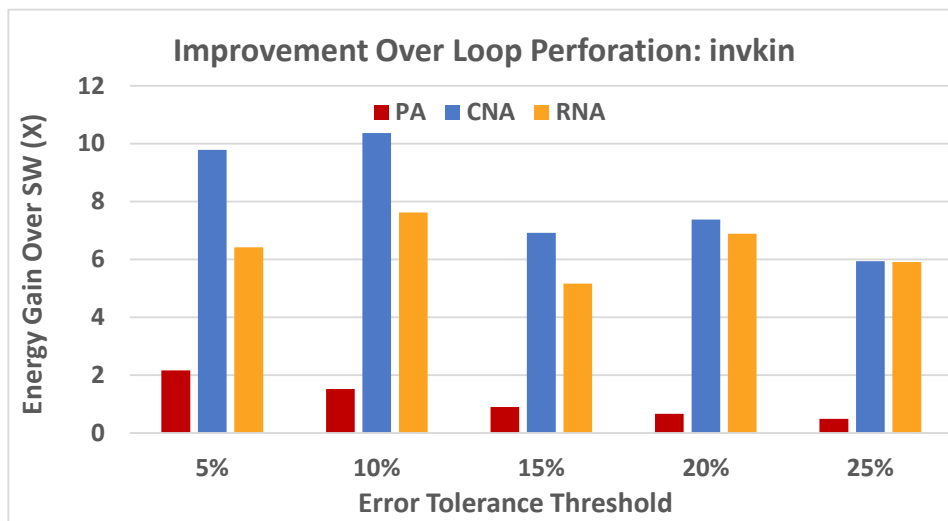


Figure 8.14: Energy results for hardware acceleration of `invkin` benchmark in comparison to loop perforation implemented in software

down the multi-stage acceleration flow, which translates into more RNA reconfigurations. Also, as the error threshold increases and the total execution time of a benchmark decreases, the configuration overhead may appear to contribute to larger portions of the execution time (though the actual amount of overhead may be decreasing), resulting in a slight trend upwards. For an error tolerance of 10% and higher, however, the average configuration overhead remains less than 5%.

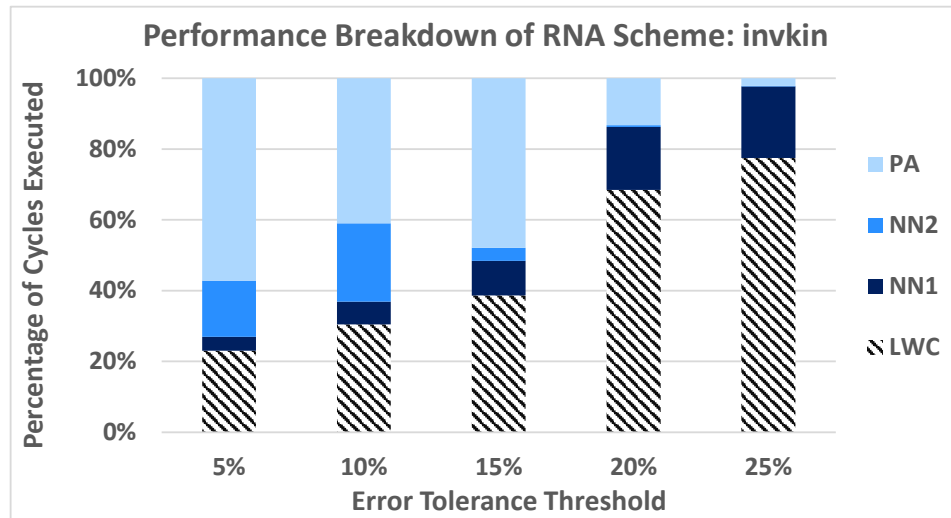


Figure 8.15: Performance breakdown in terms of cycles executed for *RNA* scheme running *invkin* benchmark

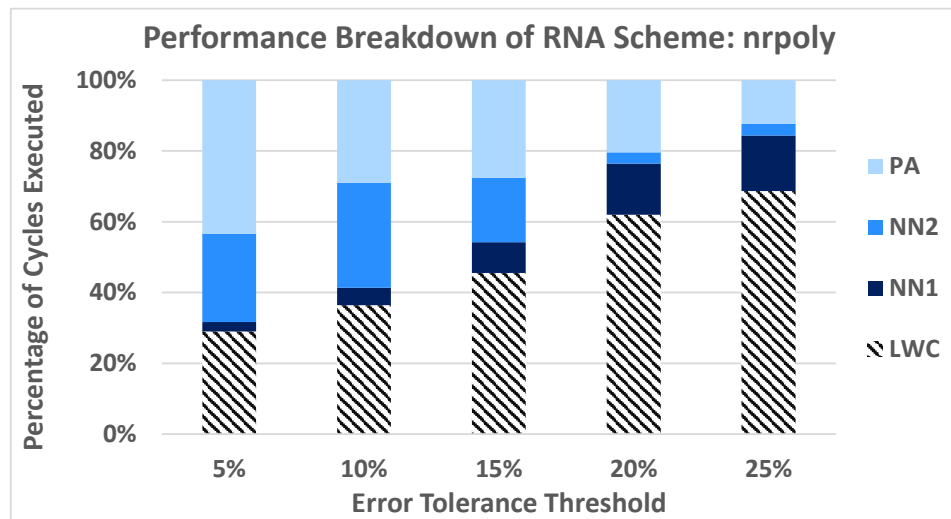


Figure 8.16: Performance breakdown in terms of cycles executed for *RNA* scheme running *nrpoly* benchmark

8.4.5 RNA Design Space Exploration

As previously mentioned, our RNA design has a single input FIFO, a single output FIFO, and a minimum II of 8. To test the optimality of our RNAs, we perform design space exploration by varying the number of input/output FIFOs, which in turn impacts the minimum II. More specifically, for a given design point, the use of M FIFOs reduces the minimum II by a factor of M .

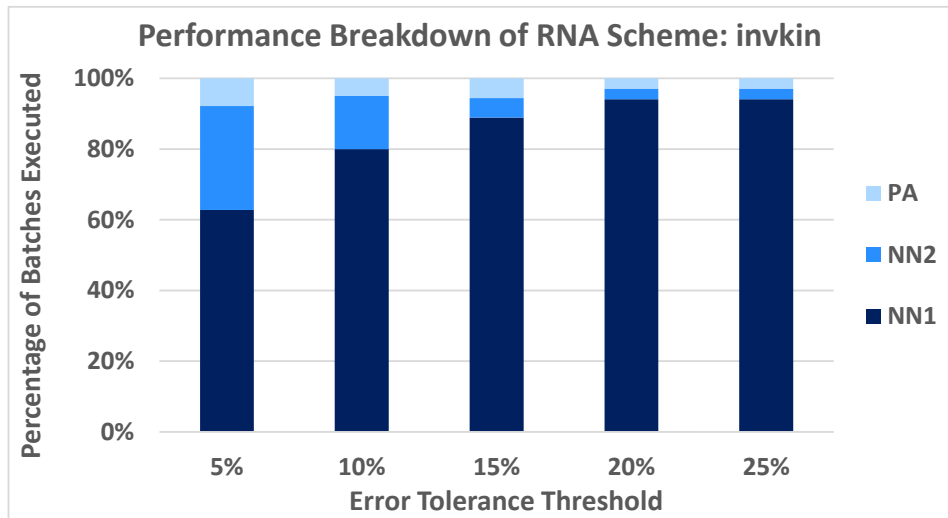


Figure 8.17: Performance breakdown in terms of batches executed for *RNA* scheme running *invkin* benchmark

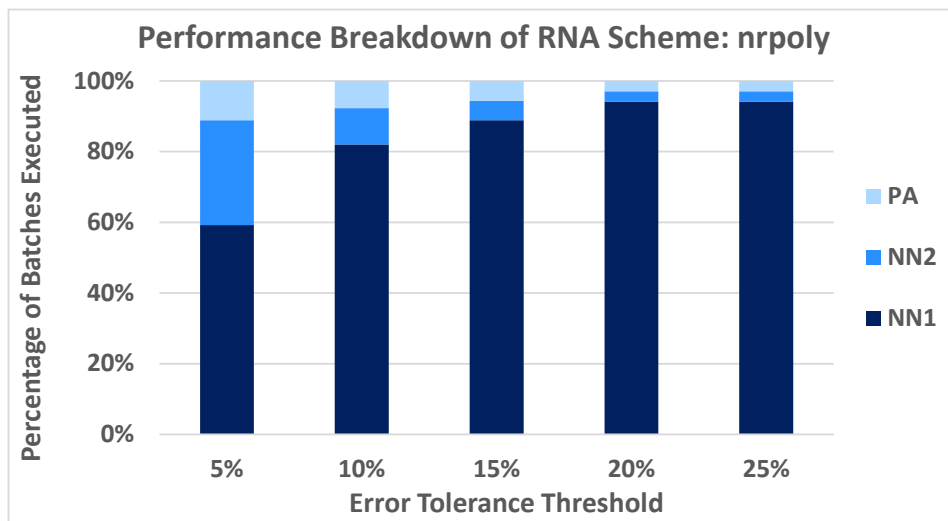


Figure 8.18: Performance breakdown in terms of batches executed for *RNA* scheme running *nrpoly* benchmark

In order to support this II reduction, however, the RNA hardware must widen its computational pipeline and increase the number of logic elements by a factor of approximately M . For example, implementing an RNA with 2 FIFOs means integrating almost twice as many components into one RNA and allowing it to accept inputs twice as fast. As a result, we found there is negligible reduction in area when compared to the total area consumed by M individual 1-FIFO RNAs. Also, since the network is not the primary bottleneck in our system, the M -FIFO RNA achieves

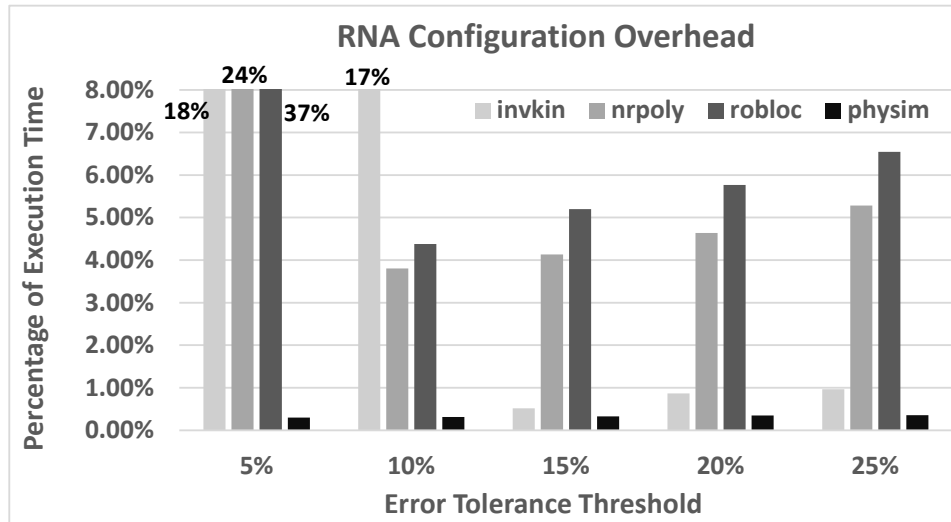


Figure 8.19: Performance overhead for configuring RNAs

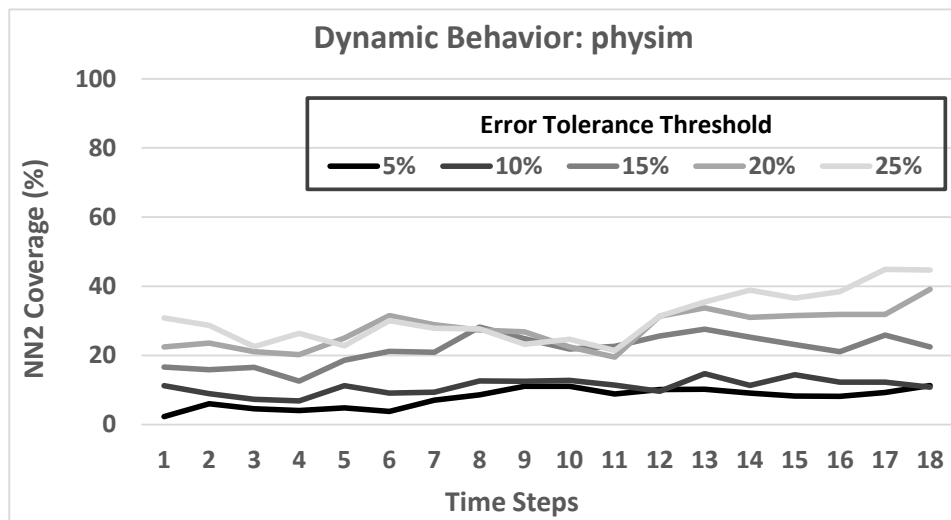


Figure 8.20: Varying coverage of NN2 in physim benchmark

virtually no gain in overall throughput relative to M separate 1-FIFO RNAs running in parallel. Furthermore, clumping the components into larger RNAs and allowing fewer of them to exist on the same platform (i.e. due to area constraints) restricts the number of independent tasks that can be run simultaneously. For these reasons, we find the 1-FIFO RNA to be the optimal design point in our system.

8.4.6 Dynamically Adaptive Acceleration Flow

With our methodology, the goal is to combine approximation with recovery in a way that achieves higher performance than the non-approximated version of the computation. Carefully trading-off the accuracy versus performance of the accelerators is critical, yet finding an optimal design point is non-trivial in practice. In addition, the impact of recovery cannot always be minimized. Unpredictable trends in input data, for example, may result in unanticipated loss of accuracy in the approximate accelerators, thereby inducing a surge of recovery attempts. Figure 8.20 demonstrates how NN2 of the `physim` benchmark has varying coverage for a series of time-steps in the simulation; for instance, with an error tolerance threshold of 15%, NN2 coverage ranges from 12% to 28%. Depending on the performance overhead of the precise computation, it may or may not be worthwhile to attempt the NN2 approximation. Moreover, data locality potentially exists across the time-steps, and can be leveraged to more easily predict and avoid periods of inaccurate approximation. We therefore see a potential for adding dynamic flexibility to multi-stage acceleration. In our heterogeneous platform, feedback from LWCs could be used to continuously gauge quality versus performance. By tracking such trends, this system would be able to dynamically adapt its multi-stage flow and accelerator usage in order to maximize performance gains and energy savings.

8.4.7 Visualization of Physics-Based Simulation

To demonstrate our system’s capacity for error control, we provide a visualization of the `physim` benchmark for a sample scene containing four spheres moving at different velocities and colliding within a rigid box (Figure 8.21). Each column presents a simulation of the same scene, only with a different error tolerance threshold; each row shows the same frame across the different simulations. The yellow arrow protruding from each sphere represents the magnitude and direction of that sphere’s velocity. Using the first column (i.e. 0% error) as a reference, we see that the other simulations diverge further and further away from the “correct” simulation as the error tolerance threshold increases. However, since our design ensures that energy is conserved across the frames, there are no sudden variations that would dramatically impact or spoil the simulation quality.

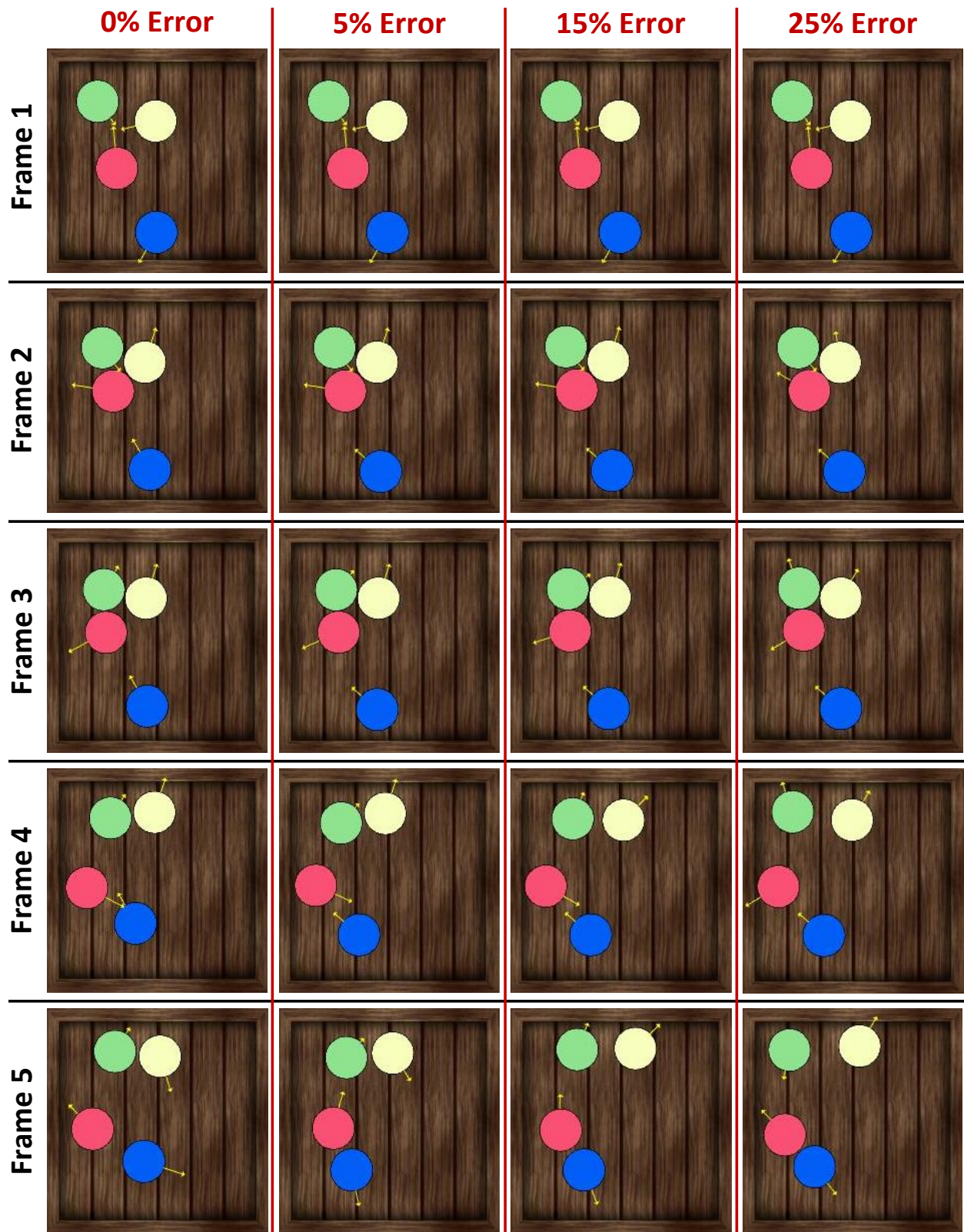


Figure 8.21: Visualization of physics-based simulation with various error tolerance thresholds

Chapter 9

Related Work

9.1 Accelerator-Rich Design

There is a large amount of prior work that implements an application-specific coprocessor or accelerator through either ASIC or FPGA [131–133] technology. These works mostly consider a single accelerator dedicated to a single application. Convey [134] and Nallatech [85], on the other hand, target reconfigurable computing through diverse sets of accelerators; these customized accelerators are off-chip from the processors, much like our customizable acceleration platform [54]. Prior art in both academia and industry has also evaluated integrating accelerators and processing cores on a single chip, which falls more in line with our research objectives. Garp [135], UltraSPARC T2 [136], and Intel’s Larrabee [6], feature designs where accelerators are tightly coupled with processing cores, or groups of cores. IBM’s WSP processor [13] is designed with looser coupling, both in terms of programmability and usability. Our work [14–16] has similarly focused on loosely-coupled accelerators that are shared among multiple cores and could be composed to form larger virtual accelerators.

Heterogeneous architectures have also been widely examined in prior art. Examples of these works include EXOCHI [137], HiPPAI [138], SARC [99], and QsCores [12], among many others. EXOCHI

presents an architecture and associated programming environment for a multicore system. HiPPAI, like our work, eliminates operating system overhead, instead using a light-weight software layer. SARC similarly relies on software management of accelerators. QsCores features specialized cores to provide energy efficiency and high performance for frequently seen code patterns. We distinguish ourselves from these works by advocating the implementation of special-purpose hardware with dynamic accelerator-management and load-balancing capabilities [15], which limits the core's interaction with accelerators and eliminates it as a performance bottleneck.

Prior work has also investigated the composition of large and complex computational elements out of simpler structures. Examples of this are core fusion [139], core spilling [140], and TRIPS [141]. In these works, the goal of composition is to construct a mostly general-purpose processing element, while our work has focused on composing highly specialized structures that are capable of performance and efficiency beyond the capabilities of general-purpose cores.

Accelerator virtualization has likewise been studied by works such as VEAL [142], PPA [143], and DySER [144]. VEAL proposes an architecture template for loop accelerators, along with a hybrid static-dynamic approach to mapping a loop to that structure. PPA features an array of processing elements which are configured to collaborate together. DySER proposes integrating a configurable accelerator into a core's execution engine to allow programs to dynamically encode kernels into custom instructions. These designs all lack support for virtualizing accelerators outside the scope of their available building blocks. The PF-based techniques [16] we have contributed could essentially be integrated into any of these composable accelerator architectures to enhance their adaptivity and longevity. Furthermore, while our composable acceleration scheme could be implemented on a programmable SoC, such as the Zynq [145] device from Xilinx, integrating PF into an ASIC-based accelerator architecture allows for added performance benefits from using the customized ASIC accelerators.

9.2 SIMD Divergence

Prior work has proposed various techniques to address branch divergence issues in SIMD architectures. Most of these approaches require complex hardware modifications that consume area, power, and energy, and nevertheless experience lowered utilization when processing diverse control flow. SIMD instruction set extensions [8, 146, 147] rely on the core they are tied to for implicit handling of control flow. Some GPUs [9, 17] use “priority scheduling” of warps to hide latency of divergence, but this thread scheduling procedure incurs overhead while only resolving the latency of stalled threads instead of the latency of the divergent threads themselves. Other techniques [18, 21, 148–151] dynamically modify (regroup, divide, or compact) the thread warps so as to reduce latency and memory divergence, yet this involves costly hardware modifications with limited impact on overall system utilization. Even static techniques [24, 25] and prediction-based thread compaction [152] lose efficiency when faced with complex data-dependent control flow.

Similar to the active masks on GPUs, VPU architectures [153–155] use “vector bit masks” to control the outputs of processing elements. Likewise, scalar processing uses *guarded instructions* [156]. These techniques ensure correctness and potentially reduce execution time, but still result in low *useful* utilization. Furthermore, note that branch predictors or speculative compute engines are rendered ineffective on SIMD architectures, particularly when dealing with data-dependent divergence, because even one misprediction can cause a significant stall. For instance, on a GPU if a single thread is mispredicted and diverges, it stalls the progress of all other threads in the same warp.

Compared to computation on a general-purpose core, accelerator-rich designs [12, 14, 15, 144] can often reduce a great deal of the overhead associated with data movement and control flow [157]. However, these designs must deal with branch divergence in the following ways: they either (1) are forced to subsume all of the control flow into a single monolithic accelerator, (2) decompose the different cases of the control flow into separate accelerators, compute all the cases, and multiplex the correct result at the end, or (3) avoid the control flow entirely by offloading it to the core.

Though they ensure program correctness, each of these alternatives can be costly in terms of area, power, resource utilization, and performance.

While much of the prior art on SIMD branch divergence focuses on reducing the *amount* of computation, we explore the efficiency to be gained if the *nature* of the computation is regularized. The approximation-based technique known as “branch herding” [122] similarly aims to transform computation into a non-divergent form. This technique reduces branch divergence by “herding” threads of GPU warps down the same path for various control flow regions, and uses static analysis and profiling to minimize output degradation. In comparison, our NN implementations consistently outperform the performance-wise optimal implementations of branch herding (i.e. *GPU_Ideal*) while maintaining reasonable accuracy. The reason for this is because we approximate control flow regions by exploiting the intelligent learning capabilities of neural networks, allowing us to emulate the functionality of the different branch paths using the same non-divergent computation. As a result, our trained NNs achieve higher accuracy while subsuming more computation.

9.3 Approximate Computing

Approximate computing has been studied extensively for the purposes of improving performance, energy consumption, and resource utilization. There exist numerous hardware-based approaches, including stochastic or probabilistic technology [158–160], approximate circuitry for arithmetic [32, 42, 161] as well as general logic [162–164], architectures based on voltage scaling [33, 36, 165], and processing units for computing neural-network-based approximations [27]. In relation to these approaches, LWCs are software-based and platform-agnostic, allowing them to be interchangeably coupled with different approximate accelerators. With our approach, we also make the distinction to decouple error analysis of the approximation unit from quality analysis of the overall application, demonstrating benefits in terms of coverage and reliability.

Prior art has also explored purely software-based approximate computing. Incremental refinement [39] and loop perforation [40], for instance, approximate iteratively-constructed solutions by

reducing compute iterations. Likewise, SAGE [166] explores static compilation of kernels with varying levels of approximation. Other dynamic approaches include selective bit-width adaptation [41] and transformation of static configuration parameters into adjustable knobs [37]. Though LWCs are similarly software-based, we employ high-level quality metrics and avoid unnecessary error checking. Also, with the use of recovery methods, we provide absolute guarantees on worst-case error and obviate the need for statically-constructed models of error distribution.

In addition to pure hardware- or software-based solutions, approximate computing experts have also looked to hardware-software codesign. These systems [34–36, 44] typically include software support (e.g. new programming language and compiler) along with a series of architectural innovations (e.g. ISA extensions). Similar to these techniques, the *BRAINIA*C platform combines our software-based LWC methodology with hardware-based approximate accelerators to form a synergistic hardware-software design. Unlike these existing approximate computing systems, however, we endorse a methodology based on dynamic error analysis at the *application level*, gaining additional coverage from leveraging slack that may not be evident at the level of the approximation unit.

9.3.1 Error Control

Language features, static analysis, and program logic may be used to control the impact of errors and ensure reliability during program execution [45, 167]. With EnerJ [38], for example, language support enables protection of specific values and compute regions, allowing hardware to readily perform approximate computation without being burdened by online error detection. Novelty detection [168], which enables recognition of out-of-range inputs using statistical estimations, may also be used to avoid potentially poor approximations. Likewise, correction mechanisms, such as those based on algorithmic noise tolerance [165, 169], allow for detection of errors during the course of the algorithm, albeit at an additional hardware cost. Error acceptance [170] also relates to our research, as it allows erroneous results to proceed so long as overall application output quality is not compromised. However, though the growth of error rates is controlled during program execution,

there is no mechanism for dynamically leveraging the error information to initiate recovery for cases with unacceptable quality loss.

9.3.2 Error Analysis

Common metrics for error analysis include error rate (ER), error significance (ES), mean squared error (MSE), root mean squared error (RMSE), mean error distance (MED), and peak signal-to-noise ratio (PSNR) [124, 171]. Error could also be predicted using estimations of confidence intervals [46, 172] and error bounds [173]. Error prediction is an orthogonal topic relative to our methodology, and could be readily combined with error analysis and recovery mechanisms to provide a more holistic approach to imprecision tolerance.

Application-specific error analysis [43] is arguably most relevant to our work. Approaches such as ERSA [35] and EnerJ [38] conduct high-level error analysis using application-specific quality metrics. However, output quality degradation of approximate executions is most often measured with respect to precise executions (e.g. deviations in classification results), deeming the QoS metrics unsuitable for conducting light-weight error analysis at runtime. Low-cost error detectors [129], on the other hand, are application-specific and can dynamically detect various forms of data corruption, yet they are not as high-level as LWCs and do not actually gauge the quality of the program's output. All of these works nevertheless present thorough studies pertaining to application-level correctness, such as its impact on fault tolerance [43], thereby providing further support for our LWC-based methodology.

9.4 Neural Network Implementation

Neural networks have been widely studied for both general-purpose [29, 174, 175] and application-specific [176–180] pattern recognition, classification, and machine learning. In an effort to broaden the applicability of NNs, Chen et al. [181] have developed software NN implementations of high performance applications from the PARSEC [108] benchmark suite. While it promotes the potential

for general-purpose hardware-based neural accelerators, the methodology calls for complete manual reimplementations of entire benchmarks as opposed to a guided kernel extraction and conversion process.

Hardware implementations for NNs have been developed using various forms of technology [182]. These include ASIC (both digital and analog) [183–187], FPGA [188], and neuromorphic hardware [30, 189], along with specialized fault-tolerant designs [190–192]. Also related to our work are NN implementations based on GPUs [193, 194]. As they focus on accelerating the evaluation of NNs on GPUs, these implementations can be combined with our platform-agnostic *Neuralizer* approach for accelerating divergent applications on SIMD architectures. Similarly, although absolute worst-case error rates cannot be guaranteed when NN training is done using non-exhaustive data sets, research on estimating NN error bounds, such as [173], can be used in conjunction with our NN-based techniques to strengthen the confidence in our outputted results.

Among the related work on NN implementations, our work finds significant common ground with the general-purpose neural acceleration [27] developed by Esmailzadeh et al. Similar to our approach, they select code kernels from compute-intensive, approximable applications, train NNs to mimic the functionality of those kernels, and compute that approximate functionality at runtime. However, they require ad-hoc kernel identification, whereas our divergence-guided *Neuralizer* approach strategically automates kernel identification while addressing major microarchitectural inefficiencies of SIMD designs. In addition, while their neural accelerators are tightly-coupled to the core, the accelerators on our *BRAINIA C* platform are loosely-coupled, allowing our system to avoid instruction processing overhead when using the accelerators. Our loosely-coupled accelerators would also be easier to load-balance among and share in a multi-core design (e.g. eschew difficulties of context-switching). Furthermore, their system does not employ mechanisms for dynamic error control, thereby having no need for a heterogeneous design that accommodates recovery via precise accelerators.

Chapter 10

Conclusion

Accelerators, meaning customized series of operations, inherently compromise generality in execution in order to gain efficiency. Chapters 2, 3, 4, and 5 discuss various architectural designs of accelerator-rich platforms. The approaches we investigate include FPGA-based vs. ASIC-based implementations, OS-based vs. hardware-based resource management, and global arbitration vs. task-based workload distribution. We also explore several other aspects of accelerator-rich designs, including accelerator chaining, virtualization, programmability, scalability, and longevity. This work is motivated by our belief that future high performance computers, especially green supercomputers, will improve their performance and power efficiency through extensive use of accelerators. Moreover, these efforts work to bridge the gap between inefficient general-purpose execution and high-performance custom computation.

SIMD branch divergence, a well-known problem that affects accelerator-rich designs, is then examined in Chapter 6. We present our approach based on neural networks, where we approximate control flow regions in order to trade-off precision for gains in performance and energy. Our approach includes a complete methodology with an automated software flow and supplemental optimization techniques. This research also highlights the importance of exploring different neural network topologies and kernel scopes in order to find optimal neural approximations. While we evaluate

our approach on a GPU, the platform-agnostic techniques we introduce are generally applicable for approximation-based acceleration of divergent applications on SIMD architectures.

In order to address the reliability concerns associated with approximate computing, Chapter 7 presents a methodology for performing adaptive error analysis and recovery based on light-weight checks (LWCs). These high-level metrics leverage application-specific imprecision tolerance to dynamically harness benefits in terms of coverage and reliability. Unlike existing online approaches, such as incremental refinement, our approach effectively employs LWCs to minimize the overhead for error analysis and initiate recovery as needed. Platform-agnostic in nature, LWCs allow for an elegant solution to dynamic error control.

Ultimately, Chapter 8 presents *BRAINIAC*, a heterogeneous platform that combines custom accelerators, approximate computing, and online error control. The design of our reconfigurable neural accelerators allows for a wide range of neural networks to be configured and employed for approximating compute-intensive workloads. We use LWCs to dynamically throttle our multi-stage acceleration flow and carefully trade-off precision for performance gain. Evaluation of our platform shows significant speedup and energy gains compared to a system that performs software-based execution or one that employs only precise accelerators. With heterogeneous systems like *BRAINIAC*, high performance applications that tolerate imprecision can be targeted more effectively than with conventional acceleration techniques.

Future work in the area of accelerator-based computing may explore ways to create more viable accelerators for memory-intensive applications, such as techniques for more efficient data packing over buses and interconnects. In order to increase accelerator usage in modern-day machines, the usability of accelerators must also be addressed, meaning we must improve on methods for seamless kernel identification, extraction, implementation, and integration. This may also be achieved, for example, by creating libraries of optimized domain-specific accelerators (e.g. for image processing) that are reusable across different applications within a given domain. To further improve on robustness and reliability, static error control methods (e.g. programming language support or

compiler-based verification of quantitative reliability) could be used in combination with our dynamic LWC-based approach. Learning techniques, such as retraining of neural networks based on detection of out-of-range inputs, could also be incorporated into this methodology. Additionally, while the neural approximations in this work are based on multilayer perceptrons, other models, such as recurrent or convolutional neural networks as well as ensemble learning models, could be explored. In terms of neural network implementation in hardware, future work may incorporate neuromorphic hardware components, allowing for ultra low-power designs.

In their textbook on the fundamentals of modern-day artificial intelligence [4], Russell and Norvig state, “Partial observability and stochasticity are ubiquitous in the real world, and so, therefore, is decision making under uncertainty.” Partially observable and stochastic environments, such as those perceived by an autonomous robot performing simultaneous localization and mapping, can be made more observable by increasing sensing capabilities. The resulting increase in input data, however, calls for more advanced and adaptive techniques for processing, filtering, and learning information. We face this challenge by designing power-efficient systems that exploit data-level parallelism via loosely-coupled accelerators. Furthermore, we demonstrate the benefits of a heterogeneous system that intelligently leverages approximate computing, thereby embracing the inherent partial observability and stochasticity in our world in order to improve the capabilities of high performance computing architectures.

Bibliography

- [1] Joel Zwick, Rita Wilson, Tom Hanks, Gary Goetzman, Nia Vardalos, John Corbett, Lainie Kazan, Michael Constantine, Gia Carides, Louis Mandylor, Bess Meisler, Andrea Martin, Joey Fatone, Jeffrey Jur, Mia Goldman, Chris Wilson, and Alexander Janko. *My Big Fat Greek Wedding*, 2003.
- [2] Microsystems Technology Office. Unconventional Processing of Signals for Intelligent Data Exploitation (UPSIDE), DARPA-BAA-12-53. Broad Agency Announcement, Defense Advanced Research Projects Agency (DARPA), Aug 2012.
- [3] Michele Banko and Eric Brill. Scaling to Very Very Large Corpora for Natural Language Disambiguation. In *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*, pages 26–33, 2001.
- [4] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall PTR, 3rd edition, 2010.
- [5] Altera. High Performance Computing Applications. <http://www.altera.com/end-markets/computer-storage/computer/hpc/applications/cmp-applications.html>.
- [6] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A Many-Core x86 Architecture for Visual Computing. *ACM Transactions on Graphics*, 27(3):18:1–18:15, Aug 2008.
- [7] Michael Gschwind. Chip Multiprocessing and the Cell Broadband Engine. In *Proceedings of the 3rd Conference on Computing Frontiers*, pages 1–8, 2006.
- [8] Chris Lomont. Introduction to Intel Advanced Vector Extensions. In *Proceedings of the 2nd Annual ASCI Conference*, pages 132–137, 2011.
- [9] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, 2008.
- [10] Yunsup Lee, Rimas Avizienis, Alex Bishara, Richard Xia, Derek Lockhart, Christopher Batten, and Krste Asanović. Exploring the Tradeoffs Between Programmability and Efficiency in Data-Parallel Accelerators. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, pages 129–140, 2011.

- [11] Michael J. Lyons, Mark Hempstead, Gu-Yeon Wei, and David Brooks. The Accelerator Store: A Shared Memory Framework for Accelerator-Based Systems. *ACM Transactions on Architecture and Code Optimization*, 8(4):48:1–48:22, 2012.
- [12] Ganesh Venkatesh, Jack Sampson, Nathan Goulding-Hotta, Sravanthi Kota Venkata, Michael Bedford Taylor, and Steven Swanson. QsCores: Trading Dark Silicon for Scalable Energy Efficiency with Quasi-Specific Cores. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 163–174, 2011.
- [13] Hubertus Franke, Jimi Xenidis, Claude Basso, Brian M. Bass, Sandra S. Woodward, Jeffrey D. Brown, and Charles L. Johnson. Introduction to the Wire-Speed Processor and Architecture. *IBM Journal of Research and Development*, 54(1):3:1–3:11, 2010.
- [14] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. Architecture Support for Accelerator-Rich CMPs. In *Proceedings of the 49th Annual Design Automation Conference*, pages 843–849, 2012.
- [15] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. CHARM: A Composable Heterogeneous Accelerator-Rich Microprocessor. In *Proceedings of the IEEE/ACM International Symposium on Low Power Electronics and Design*, pages 379–384, 2012.
- [16] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Hui Huang, and Glenn Reinman. Composable Accelerator-rich Microprocessor Enhanced for Adaptivity and Longevity. In *Proceedings of the IEEE/ACM International Symposium on Low Power Electronics and Design*, pages 305–310, 2013.
- [17] AMD. AMD FireStream GPU Compute Accelerators. <http://www.amd.com/us/products/workstation/graphics/legacy/Pages/firestream.aspx>.
- [18] Jiayuan Meng, David Tarjan, and Kevin Skadron. Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, pages 235–246, 2010.
- [19] Ujval J. Kapasi. *Conditional Techniques for Stream Processing Kernels*. PhD thesis, Stanford University, 2004.
- [20] Razvan Cheveresan, Matt Ramsay, Chris Feucht, and Ilya Sharapov. Characteristics of Workloads Used in High Performance and Technical Computing. In *Proceedings of the 21st Annual International Conference on Supercomputing*, pages 73–82, 2007.
- [21] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 407–420, 2007.
- [22] Ujval J. Kapasi, William J. Dally, Scott Rixner, Peter R. Mattson, John D. Owens, and Brucec Khailany. Efficient Conditional Operations for Data-Parallel Architectures. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 159–170, 2000.

- [23] Ingo Wald. Active Thread Compaction for GPU Path Tracing. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, pages 51–58, 2011.
- [24] Haicheng Wu, Gregory Diamos, Jin Wang, Si Li, and Sudhakar Yalamanchili. Characterization and Transformation of Unstructured Control Flow in Bulk Synchronous GPU Applications. *International Journal of High Performance Computing Applications*, 26(2):170–185, 2012.
- [25] Gregory Diamos, Benjamin Ashbaugh, Subramaniam Maiyuran, Andrew Kerr, Haicheng Wu, and Sudhakar Yalamanchili. SIMD Re-Convergence at Thread Frontiers. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 477–488, 2011.
- [26] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, 2nd edition, 1998.
- [27] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural Acceleration for General-Purpose Approximate Programs. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 449–460, 2012.
- [28] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning Representations by Back-Propagating Errors. *Nature*, 323(6088):533–536, 1986.
- [29] Guoqiang P. Zhang. Neural Networks for Classification: A Survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 30(4):451–462, 2000.
- [30] N. Srinivasa and J. M. Cruz-Albrecht. Neuromorphic Adaptive Plastic Scalable Electronics: Analog Learning Systems. *IEEE Pulse*, 3(1):51–56, 2012.
- [31] Pradeep Dubey. A Platform 2015 Workload Model: Recognition, Mining and Synthesis Moves Computers to the Era of Tera. White Paper, Intel Corporation, 2007.
- [32] Thomas Yeh, Petros Faloutsos, Milos Ercegovac, Sanjay Patel, and Glenn Reinman. The Art of Deception: Adaptive Precision Reduction for Area Efficient Physics Acceleration. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 394–406, 2007.
- [33] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, and Trevor Mudge. Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 7–18, 2003.
- [34] Woongki Baek and Trishul M. Chilimbi. Green: A Framework for Supporting Energy-Conscious Programming using Controlled Approximation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 198–209, 2010.
- [35] H. Cho, L. Leem, and S. Mitra. ERSA: Error Resilient System Architecture for Probabilistic Applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(4):546–558, 2012.

- [36] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Architecture Support for Disciplined Approximate Programming. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 301–312, 2012.
- [37] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic Knobs for Responsive Power-Aware Computing. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 199–212, 2011.
- [38] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. EnerJ: Approximate Data Types for Safe and General Low-Power Computation. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 164–174, 2011.
- [39] S. H. Nawab, A. V. Oppenheim, A. P. Chandrakasan, J. M. Winograd, and J. T. Ludwig. Approximate Signal Processing. *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, 15(1–2):177–200, 1997.
- [40] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing Performance vs. Accuracy Trade-Offs with Loop Perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 124–134, 2011.
- [41] J. Park, J. H. Choi, and K. Roy. Dynamic Bit-Width Adaptation in DCT: An Approach to Trade Off Image Quality and Computation Energy. *IEEE Transactions on Very Large Scale Integration Systems*, 18(5):787–793, 2010.
- [42] Jiawei Huang, John Lach, and Gabriel Robins. A Methodology for Energy-Quality Tradeoff Using Imprecise Hardware. In *Proceedings of the 49th Annual Design Automation Conference*, pages 504–509, 2012.
- [43] X. Li and D. Yeung. Application-Level Correctness and its Impact on Fault Tolerance. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, pages 181–192, 2007.
- [44] Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. Relax: An Architectural Framework for Software Recovery of Hardware Faults. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, pages 497–508, 2010.
- [45] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 33–52, 2013.
- [46] G. Chryssolouris, M. Lee, and A. Ramsey. Confidence Interval Prediction for Neural Network Models. *IEEE Transactions on Neural Networks*, 7(1):229–232, 1996.
- [47] P. Garcia and K. Compton. Kernel Sharing on Reconfigurable Multiprocessor Systems. In *International Conference on Field Programmable Technology*, pages 225–232, 2008.

- [48] Ning Sun and Chi-Chang Lin. *Using the Cryptographic Accelerators in the the UltraSPARC T1 and T2 Processors*. Oracle, Sun BluePrints Online, Nov 2007.
- [49] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *Computer*, 35(2):50–58, 2002.
- [50] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet’s General Execution-Driven Multiprocessor Simulator Toolset. *ACM SIGARCH Computer Architecture News*, 33(4):92–99, 2005.
- [51] Alex Bui, Kwang-Ting Cheng, Jason Cong, Luminita Vese, Yi-Chu Wang, Bo Yuan, and Yi Zou. Platform Characterization for Domain-Specific Computing. In *Proceedings of the 17th Asia and South Pacific Design Automation Conference*, pages 94–99, 2012.
- [52] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA. *Journal of Parallel and Distributed Computing*, 68(10):1370–1380, 2008.
- [53] J. M. Ortega and W. C. Rheinboldt. *Iterative Solution of Nonlinear Equations in Several Variables*. Academic Press, 1970.
- [54] Jason Cong, Beayna Grigorian, Glenn Reinman, and Marco Vitanza. Accelerating Vision and Navigation Applications on a Customizable Platform. In *Proceedings of the 22nd IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 25–32, 2011.
- [55] Jason Cong, Karthik Gururaj, Muhuan Huang, Sen Li, Bingjun Xiao, and Yi Zou. Domain-Specific Processor with 3D Integration for Medical Image Processing. In *Proceedings of the 22nd IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 247–250, 2011.
- [56] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong. *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer Publishers, 2008.
- [57] C. H. Chen. *Handbook of Pattern Recognition and Computer Vision*. World Scientific, 2009.
- [58] Huiyu Zhou, Yuan Yuan, and Chunmei Shi. Object Tracking Using SIFT Features and Mean Shift. *Computer Vision and Image Understanding*, 113(3):345–352, 2009.
- [59] S. T. Birchfield and S. J. Pundlik. Joint Tracking of Features and Edges. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–6, 2008.
- [60] David A. Forsyth and Jean Ponce. *Computer Vision: A Modern Approach*. Prentice Hall PTR, 1st edition, 2002.
- [61] Lu-Fang Gao, Yu-Xian Gai, and Sheng Fu. Simultaneous Localization and Mapping for Autonomous Mobile Robots Using Binocular Stereo Vision System. In *Proceedings of the International Conference on Mechatronics and Automation*, pages 326–330, 2007.

- [62] Francisco Bonin-Font, Alberto Ortiz, and Gabriel Oliver. Visual Navigation for Mobile Robots: A Survey. *Journal of Intelligent and Robotic Systems*, 53(3):263–296, 2008.
- [63] Kok Seng Chong and L. Kleeman. Sonar Based Map Building for a Mobile Robot. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1700–1705, 1997.
- [64] Xuefeng Dai, Hongmin Zhang, and Yan Shi. Autonomous Navigation for Wheeled Mobile Robots - A Survey. In *Proceedings of the 2nd International Conference on Innovative Computing, Information and Control*, pages 551–551, 2007.
- [65] Sebastian Thrun. Simultaneous Localization and Mapping. In *Robotics and Cognitive Approaches to Spatial Mapping*, volume 38, pages 13–41. Springer Berlin Heidelberg, 2008.
- [66] Jose-Luis Blanco. Derivation and Implementation of a Full 6D EKF-based Solution to Bearing-Range SLAM. Technical Report, University of Malaga, Spain, Mar 2008.
- [67] R. E. Kalman. A New Approach to Linear Filtering and Prediction Problems. *Journal of Basic Engineering*, 82:35–45, 1960.
- [68] D. G. Lowe. Object Recognition from Local Scale-Invariant Features. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1150–1157, 1999.
- [69] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool. SURF: Speeded Up Robust Features. *Computer Vision and Image Understanding*, 110(3):346–359, 2008.
- [70] J. Shi and C. Tomasi. Good Features to Track. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 593–600, 1994.
- [71] S. Lazebnik, C. Schmid, and J. Ponce. A Sparse Texture Representation Using Local Affine Regions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27:1265–1278, 2005.
- [72] M. W. M. G. Dissanayake, P. Newman, S. Clark, H. F. Durrant-White, and M. Csorba. A Solution to the Simultaneous Localization and Map Building (SLAM) Problem. *IEEE Transactions on Robotics and Automation*, 17(3):229–241, 2001.
- [73] Udo Frese. A Discussion of Simultaneous Localization and Mapping. *Autonomous Robots*, 20(1):25–42, 2006.
- [74] H. Durrant-Whyte and T. Bailey. Simultaneous Localization and Mapping: Part I. *IEEE Robotics Automation Magazine*, 13(2):99–110, 2006.
- [75] Jose-Luis Blanco. The Mobile Robot Programming Toolkit. <http://mrpt.org>.
- [76] Itseez. Open Source Computer Vision. <http://opencv.willowgarage.com>.
- [77] Anastasios I. Mourikis, Nikolas Trawny, Stergios I. Roumeliotis, Andrew E. Johnson, Adnan Ansar, and Larry Matthies. Vision-Aided Inertial Navigation for Spacecraft Entry, Descent, and Landing. *IEEE Transactions on Robotics*, 25(2):264–280, 2009.

- [78] Jose-Luis Blanco, Francisco-Angel Moreno, and Javier Gonzalez. A Collection of Outdoor Robotic Datasets with Centimeter-Accuracy Ground Truth. *Autonomous Robots*, 27(4):327–351, 2009.
- [79] PAPI. Performance API. <http://icl.cs.utk.edu/papi>.
- [80] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 469–480, 2009.
- [81] Xilinx ISE Design Suite. <http://www.xilinx.com/tools/designtools.htm>.
- [82] Xilinx XPower Analyzer. http://www.xilinx.com/products/design_tools/logic_design/verification/xpower_an.htm.
- [83] A. M. Frieze and M. R. B. Clarke. Approximation Algorithms for the M-Dimensional 0-1 Knapsack Problem: Worst-Case and Probabilistic Analyses. *European Journal of Operational Research*, 15(1):100–109, 1984.
- [84] Intel. Intel QuickPath Interconnect. <http://www.intel.com/technology/quickpath>.
- [85] Nallatech. Nallatech: Intel Xeon FSB-FPGA Accelerator Module. <http://www.nallatech.com/Intel-Xeon-FSB-Socket-Fillers/fsb-development-systems.html>.
- [86] David J. Miller, Philip M. Watts, and Andrew W. Moore. Motivating Future Interconnects: A Differential Measurement Analysis of PCI Latency. In *Proceedings of the 5th IEEE/ACM Symposium on Architectures for Networking and Communications Systems*, pages 94–103, 2009.
- [87] Jason Cong and Yi Zou. FPGA-Based Hardware Acceleration of Lithographic Aerial Image Simulation. *ACM Transactions on Reconfigurable Technology and Systems*, 2(3):17:1–17:29, Sep 2009.
- [88] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.
- [89] Eitan Tadmor, Suzanne Nezzar, and Luminita Vese. Multiscale Hierarchical Decomposition of Images with Applications to Deblurring, Denoising and Segmentation. *Communications in Mathematical Sciences*, 6(2):281–307, 2008.
- [90] Luminita A. Vese and Stanley J. Osher. Image Denoising and Decomposition with Total Variation Minimization and Oscillatory Functions. *Journal of Mathematical Imaging and Vision*, 20(1–2):7–18, Jan 2004.
- [91] Igor Yanovsky, Carole Le Guyader, Alex Leow, Paul Thompson, and Luminita Vese. Nonlinear Elastic Registration with Unbiased Regularization in Three Dimensions. In *Computational Biomechanics for Medicine III, MICCAI 2008 Workshop*, pages 56–67, 2008.

- [92] Tony F. Chan and Luminita A. Vese. Active Contours without Edges. *IEEE Transactions on Image Processing*, 10(2):266–277, 2001.
- [93] Frédéric Jurie. A New Log-Polar Mapping for Space Variant Imaging: Application to Face Detection and Tracking. *Pattern Recognition*, 32(5):865–875, May 1999.
- [94] Matteo Frigo and Steven G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.
- [95] Synopsys. Synopsys Design Compiler. <http://www.synopsys.com/Tools/Pages/default.aspx>.
- [96] Jason Cong, Mohammad Ali Ghodrati, Michael Gill, Chunyue Liu, Glenn Reinman, and Yi Zou. AXR-CMP: Architecture Support in Accelerator-Rich CMPs. In *Proceedings of the 2nd Workshop on SoC Architecture, Accelerators and Workloads*, pages 19–26, 2011.
- [97] Jason Cong, Hui Huang, and Wei Jiang. A Generalized Control-Flow-Aware Pattern Recognition Algorithm for Behavioral Synthesis. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1255–1260, 2010.
- [98] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the IEEE/ACM Conference on Supercomputing*, pages 83:1–83:13, 2006.
- [99] A. Ramirez, F. Cabarcas, B. Juurlink, M. Alvarez Mesa, F. Sanchez, A. Azevedo, C. Meenderinck, C. Ciobanu, S. Isaza, and G. Gaydadjiev. The SARC Architecture. *IEEE Micro*, pages 16–29, 2010.
- [100] HP Labs. CACTI 5.3. <http://quid.hpl.hp.com:9081/cacti>.
- [101] Sun Microsystems. UltraSPARC III. http://en.wikipedia.org/wiki/UltraSPARC_III.
- [102] A. B. Kahng, B. Li, L. Peh, and K. Samadi. ORION 2.0: A Fast and Accurate NoC Power and Area Model for Early-Stage Design Space Exploration. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 423–428, 2009.
- [103] Ian Kuon and Jonathan Rose. Measuring the Gap between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, 2007.
- [104] Jason Cong and Bingjun Xiao. Optimization of Interconnects Between Accelerators and Shared Memories in Dark Silicon. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 630–637, 2013.
- [105] Amir Hormati, Nathan Clark, and Scott Mahlke. Exploiting Narrow Accelerators with Data-Centric Subgraph Mapping. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 341–353, 2007.
- [106] Xilinx. <http://www.xilinx.com>.

- [107] Jason Cong, Glenn Reinman, Alex Bui, and Vivek Sarkar. Customizable Domain-Specific Computing. *IEEE Design and Test of Computers*, 28(2):6–15, 2011.
- [108] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, 2008.
- [109] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. SD-VBS: The San Diego Vision Benchmark Suite. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 55–64, 2009.
- [110] Michael Lustig, David Donoho, and John M. Pauly. Sparse MRI: The Application of Compressed Sensing for Rapid MR Imaging. *Magnetic Resonance in Medicine*, 58(6):1182–1195, 2007.
- [111] Swarat Chaudhuri, Sumit Gulwani, Roberto Lubliner, and Sara Navidpour. Proving Programs Robust. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 102–112, 2011.
- [112] Dan Quinlan. ROSE: Compiler Support for Object-Oriented Frameworks. *Parallel Processing Letters*, 10:215–226, 2000.
- [113] Kurt Hornik. Approximation Capabilities of Multilayer Feedforward Networks. *Neural Networks*, 4(2):251–257, Mar 1991.
- [114] P. K. Meher. An Optimized Lookup-Table for the Evaluation of Sigmoid Function for Artificial Neural Networks. In *Proceedings of the 18th IEEE/IFIP VLSI System on Chip Conference*, pages 91–95, 2010.
- [115] Nvidia CUDA Math Library. <http://developer.nvidia.com/cuda-math-library>.
- [116] Li-Chun Tommy Wang and Chih Cheng Chen. A Combined Optimization Method for Solving the Inverse Kinematics Problem of Mechanical Manipulators. *IEEE Transactions on Robotics and Automation*, 7(4):489–499, Aug 1991.
- [117] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Pro., 2010.
- [118] Steffen Nissen. Implementation of a Fast Artificial Neural Network Library (FANN). Technical Report, Department of Computer Science University of Copenhagen (DIKU), 2003.
- [119] Nvidia CUDA 5.5 Production Release. <http://developer.nvidia.com/cuda-downloads>.
- [120] Nvidia GeForce GTX 480. <http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-480>.
- [121] P3 International. Kill A Watt. <http://www.p3international.com/products/p4400.html>.
- [122] J. Sartori and R. Kumar. Branch and Data Herding: Reducing Control and Memory Divergence for Error-Tolerant GPU Applications. *IEEE Transactions on Multimedia*, 15(2): 279–290, Feb 2013.

- [123] Beayna Grigorian and Glenn Reinman. Dynamically Adaptive and Reliable Approximate Computing Using Light-Weight Error Analysis. In *NASA/ESA Conference on Adaptive Hardware and Systems*, pages 248–255, 2014.
- [124] J. Han and M. Orshansky. Approximate Computing: An Emerging Paradigm for Energy-Efficient Design. In *Proceedings of the 18th IEEE European Test Symposium*, pages 1–6, 2013.
- [125] Z. Wang and A. C. Bovik. A Universal Image Quality Index. *IEEE Signal Processing Letters*, 9(3):81–84, 2002.
- [126] T. J. Schaefer. The Complexity of Satisfiability Problems. In *Proceedings of the 10th Symposium on Theory of Computing*, pages 216–226, 1978.
- [127] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM*, 5(7):394–397, Jul 1962.
- [128] Beayna Grigorian and Glenn Reinman. Accelerating Divergent Applications on SIMD Architectures Using Neural Networks. In *Proceedings of the 32nd IEEE International Conference on Computer Design*, page To Appear, 2014.
- [129] S. K. S. Hari, S. V. Adve, and H. Naeimi. Low-Cost Program-Level Detectors for Reducing Silent Data Corruptions. In *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 1–12, 2012.
- [130] Erin Catto. Box2D: A 2D Physics Engine for Games. <http://box2d.org>.
- [131] Pan Min and Sun Yihe. ASIC Design of Gabor Transform for Speech Processing. In *Proceedings of the 4th International Conference on ASIC*, pages 401–404, 2001.
- [132] D. Bouris, A. Nikitakis, and I. Papaefstathiou. Fast and Efficient FPGA-Based Feature Detection Employing the SURF Algorithm. In *Proceedings of the 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 3–10, 2010.
- [133] Jason Cong and Yi Zou. FPGA-Based Hardware Acceleration of Lithographic Aerial Image Simulation. *ACM Transactions on Reconfigurable Technology and Systems*, 2(3):17:1–17:29, Sep 2009.
- [134] Convey Computer. The Hybrid-Core Series. <http://www.conveycomputer.com/products/hcseries>.
- [135] J. R. Hauser and J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *Proceedings of the 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 12–21, 1997.
- [136] Tim Johnson and Umesh Nawathe. An 8-core, 64-thread, 64-bit Power Efficient SPARC SoC (Niagara2). In *Proceedings of the 2007 International Symposium on Physical Design*, pages 2–2, 2007.

- [137] Perry H. Wang, Jamison D. Collins, Gautham N. Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y. Yang, Guei-Yuan Lueh, and Hong Wang. EXOCHI: Architecture and Programming Environment for a Heterogeneous Multi-core Multithreaded System. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 156–166, 2007.
- [138] P.M. Stillwell, V. Chadha, O. Tickoo, S. Zhang, R. Illikkal, R. Iyer, and D. Newell. HiPPAI: High Performance Portable Accelerator Interface for SoCs. In *Proceedings of the International Conference on High Performance Computing*, pages 109–118, 2009.
- [139] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. Core Fusion: Accommodating Software Diversity in Chip Multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 186–197, 2007.
- [140] J. Cong, H. Guoling, A. Jagannathan, G. Reinman, and K. Rutkowski. Accelerating Sequential Applications on CMPs Using Core Spilling. *IEEE Transactions on Parallel and Distributed Systems*, 18(8):1094–1107, 2007.
- [141] Mark Gebhart, Bertrand A. Maher, Katherine E. Coons, Jeff Diamond, Paul Gratz, Mario Marino, Nitya Ranganathan, Behnam Robatmili, Aaron Smith, James Burrill, Stephen W. Keckler, Doug Burger, and Kathryn S. McKinley. An Evaluation of the TRIPS Computer System. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–12, 2009.
- [142] N. Clark, A. Hormati, and S. Mahlke. VEAL: Virtualized Execution Accelerator for Loops. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, pages 389–400, 2008.
- [143] Hyunchul Park, Yongjun Park, and Scott Mahlke. Polymorphic Pipeline Array: A Flexible Multicore Accelerator with Virtualized Execution for Mobile Multimedia Application. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 370–380, 2009.
- [144] V. Govindaraju, C. Ho, and K. Sankaralingam. Dynamically Specialized Datapaths for Energy Efficient Computing. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture*, pages 503–514, 2011.
- [145] Xilinx Zynq-7000 All Programmable SoC. <http://www.xilinx.com/products/silicon-devices/soc/zynq-7000>.
- [146] Linley Gwennap. AltiVec Vectorizes PowerPC. *Microprocessor Report*, 12(6):1–6, 1998.
- [147] L. Kohn, G. Maturana, M. Tremblay, A. Prabhu, and G. Zyner. The Visual Instruction Set (VIS) in UltraSPARC. In *COMPCON: Technologies for the Information Superhighway, Digest of Papers*, pages 462–469, 1995.
- [148] Aniruddha S. Vaidya, Anahita Shayesteh, Dong Hyuk Woo, Roy Saharoy, and Mani Azimi. SIMD Divergence Optimization through Intra-Warp Compaction. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 368–379, 2013.

- [149] Nicolas Brunie, Sylvain Collange, and Gregory Diamos. Simultaneous Branch and Warp Interweaving for Sustained GPU Performance. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, pages 49–60, 2012.
- [150] W. W. L. Fung and T. M. Aamodt. Thread Block Compaction for Efficient SIMT Control Flow. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture*, pages 25–36, 2011.
- [151] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. Improving GPU Performance via Large Warps and Two-Level Warp Scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 308–317, 2011.
- [152] Minsoo Rhu and Mattan Erez. CAPRI: Prediction of Compaction-Adequacy for Handling Control-Divergence in GPGPU Architectures. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, pages 61–71, 2012.
- [153] W. J. Bouknight, S. A. Denenberg, D. E. McIntyre, J. M. Randall, A. H. Sameh, and D. L. Slotnick. The Illiac IV System. *Proceedings of the IEEE*, 60(4):369–388, 1972.
- [154] Richard M. Russell. The CRAY-1 Computer System. *Communications of the ACM*, 21(1): 63–72, Jan 1978.
- [155] J. E. Smith, G. Faanes, and R. Sugumar. Vector Instruction Set Support for Conditional Operations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 260–269, 2000.
- [156] P. Y. T. Hsu and E. S. Davidson. Highly Concurrent Scalar Processing. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 386–395, 1986.
- [157] Brandon Reagen, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. Quantifying Acceleration: Power/Performance Trade-Offs of Application Kernels in Hardware. In *Proceedings of the IEEE International Symposium on Low Power Electronics and Design*, pages 395–400, 2013.
- [158] L. N. Chakrapani, B. E. S. Akgul, S. Cheemalavagu, P. Korkmaz, K. V. Palem, and B. Seshasayee. Ultra-Efficient (Embedded) SOC Architectures Based on Probabilistic CMOS (PCMOS) Technology. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1–6, 2006.
- [159] S. Narayanan, J. Sartori, R. Kumar, and D. L. Jones. Scalable Stochastic Processors. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 335–338, 2010.
- [160] W. Qian, X. Li, M. D. Riedel, K. Bazargan, and D. J. Lilja. An Architecture for Fault-Tolerant Computation with Stochastic Logic. *IEEE Transactions on Computers*, 60(1):93–105, 2011.
- [161] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy. Low-Power Digital Signal Processing Using Approximate Adders. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(1):124–137, 2013.

- [162] Shih-Lien Lu. Speeding Up Processing with Approximation Circuits. *Computer*, 37(3):67–73, 2004.
- [163] M. R. Choudhury and K. Mohanram. Approximate Logic Circuits for Low Overhead, Non-Intrusive Concurrent Error Detection. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 903–908, 2008.
- [164] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan. SALSA: Systematic Logic Synthesis of Approximate Circuits. In *Proceedings of the 49th Annual Design Automation Conference*, pages 796–801, 2012.
- [165] R. Hegde and N. R. Shanbhag. Soft Digital Signal Processing. *IEEE Transactions on Very Large Scale Integration Systems*, 9(6):813–823, 2001.
- [166] Mehrzad Samadi, Janghaeng Lee, Davoud Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. SAGE: Self-tuning Approximation for Graphics Engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 13–24, 2013.
- [167] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. Shoestring: Probabilistic Soft Error Reliability on the Cheap. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 385–396, 2010.
- [168] C. M. Bishop. Novelty Detection and Neural Network Validation. *Vision, Image and Signal Processing*, 141(4):217–222, 1994.
- [169] G. V. Varatkar and N. R. Shanbhag. Energy-Efficient Motion Estimation Using Error-Tolerance. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 113–118, 2006.
- [170] K. He, A. Gerstlauer, and M. Orshansky. Controlled Timing-Error Acceptance for Low Energy IDCT Design. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1–6, 2011.
- [171] R. Venkatesan, A. Agarwal, K. Roy, and A. Raghunathan. MACACO: Modeling and Analysis of Circuits for Approximate Computing. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 667–673, 2011.
- [172] B. Efron and R. Tibshirani. Bootstrap Methods for Standard Errors, Confidence Intervals, and Other Measures of Statistical Accuracy. *Statistical Science*, 1(1):54–75, 1986.
- [173] N. W. Townsend and L. Tarassenko. Estimations of Error Bounds for Neural-Network Function Approximators. *IEEE Transactions on Neural Networks*, 10(2):217–230, 1999.
- [174] K. Fukushima. A Neural Network for Visual Pattern Recognition. *Computer*, 21(3):65–75, 1988.
- [175] Kunihiro Fukushima. Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position. *Biological Cybernetics*, 36(4): 193–202, 1980.

- [176] S. Lawrence, C. L. Giles, A. C. Tsoi, and A. D. Back. Face Recognition: A Convolutional Neural-Network Approach. *IEEE Transactions on Neural Networks*, 8(1):98–113, 1997.
- [177] M. W. Roth. Survey of Neural Network Technology for Automatic Target Recognition. *IEEE Transactions on Neural Networks*, 1(1):28–43, 1990.
- [178] R. Parisi, E. D. Di Claudio, G. Lucarelli, and G. Orlandi. Car Plate Recognition by Neural Networks and Image Processing. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, volume 3, pages 195–198, 1998.
- [179] K. Kamijo and T. Tanigawa. Stock Price Pattern Recognition - A Recurrent Neural Network Approach. In *Proceedings of the IEEE International Joint Conference on Neural Networks*, pages 215–221, 1990.
- [180] P. D. Gader, M. Mohamed, and J. Chiang. Handwritten Word Recognition with Character and Inter-Character Neural Networks. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 27(1):158–164, 1997.
- [181] T. Chen, Y. Chen, M. Duranton, Q. Guo, A. Hashmi, M. Lipasti, A. Nere, S. Qiu, M. Sebag, and O. Temam. BenchNN: On the Broad Potential Application Scope of Hardware Neural Network Accelerators. In *IEEE International Symposium on Workload Characterization*, pages 36–45, 2012.
- [182] Janardan Misra and Indranil Saha. Artificial Neural Networks in Hardware: A Survey of Two Decades of Progress. *Neurocomputing*, 74(1):239–255, 2010.
- [183] K. W. Przytula and V. K. Prasanna. *Parallel Digital Implementations of Neural Networks*. Prentice Hall, 1st edition, 1993.
- [184] Giovanni Danese, Francesco Loporati, and Stefano Ramat. A Parallel Neural Processor for Real-Time Applications. *IEEE Micro*, 22(3):20–31, 2002.
- [185] M. Holler, S. Tam, H. Castro, and R. Benson. An Electrically Trainable Artificial Neural Network (ETANN) with 10240 ‘Floating Gate’ Synapses. In *Proceedings of the IEEE International Joint Conference on Neural Networks*, pages 191–196, 1989.
- [186] J. Schemmel, J. Fieres, and K. Meier. Wafer-Scale Integration of Analog Neural Networks. In *Proceedings of the IEEE International Joint Conference on Neural Networks*, pages 431–438, 2008.
- [187] Renée St. Amant, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, Hadi Esmailzadeh, Arjang Hassibi, Luis Ceze, and Doug Burger. General-Purpose Code Acceleration with Limited-Precision Analog Computation. In *Proceedings of the 41st Annual International Symposium on Computer Architecture*, pages 505–516, 2014.
- [188] Jihan Zhu and Peter Sutton. FPGA Implementations of Neural Networks – A Survey of a Decade of Progress. In *Field Programmable Logic and Application*, pages 1062–1066. 2003.
- [189] Atif Hashmi, Andrew Nere, James Jamal Thomas, and Mikko Lipasti. A Case for Neuromorphic ISAs. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 145–158, 2011.

- [190] Bilel Belhadj, Antoine Joubert, Zheng Li, Rodolphe Hélot, and Olivier Temam. Continuous Real-World Inputs Can Open Up Alternative Accelerator Designs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 1–12, 2013.
- [191] Olivier Temam. A Defect-Tolerant Accelerator for Emerging High-Performance Applications. In *Proceeding of the 39th Annual International Symposium on Computer Architecture*, pages 356–367, 2012.
- [192] Atif Hashmi, Hugues Berry, Olivier Temam, and Mikko Lipasti. Automatic Abstraction and Fault Tolerance in Cortical Microarchitectures. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, pages 1–10, 2011.
- [193] Honghoon Jang, Anjin Park, and Keechul Jung. Neural Network Implementation Using CUDA and OpenMP. In *Digital Image Computing: Techniques and Applications*, pages 155–161, 2008.
- [194] Kyoung-Su Oh and Keechul Jung. GPU Implementation of Neural Networks. *Pattern Recognition*, 37(6):1311–1314, 2004.