Power-Efficient Accelerators for High-Performance Applications

by

Ganesh Suryanarayan Dasika

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2011

Doctoral Committee:

Associate Professor Scott Mahlke, Chair
Professor David Blaauw
Professor Jeffrey A. Fessler
Professor Trevor N. Mudge

© Ganesh Suryanarayan Dasika 2011
All Rights Reserved

To my parents.

ACKNOWLEDGEMENTS

This work would not have been possible without the help and support of a number of my colleagues, friends and family.

Thanks first go to Professor Scott Mahlke, my advisor through all my years in graduate school and during the last few years of college. A constant source of ideas and always full of energy and alacrity, Scott was a great advisor and a great teacher.

I would like to thank my thesis committee for providing their thoughts and suggestions for a number of my projects. Professor David Blaauw was a tremendous resource during my investigation of various power-reduction techniques. Professor Jeffrey Fessler was instrumental in providing a real-world look at medical imaging and scientific computing. Professor Trevor Mudge was effectively my co-advisor and heavily influenced the direction of my research into SIMD and power-efficient architectures.

While they were not my advisors in any official capacity, Professor James Freudenberg and Professor Mark Brehob were both very positive influences on my undergraduate and graduate careers at Michigan and provided sound advice at many pivotal points over the last several years.

My work in industry heavily influenced the ideas and solutions in this thesis. For providing me with a window into “the real world”, I thank Krisztian Flautner, David Bull, Sami Yehia, and Shidhartha Das at ARM; and Mikhail Smelyanskiy and Victor Lee at Intel.

I had some amazing colleagues while at Michigan. Thanks first go to Ankit Sethia and Vincentius Robby who put up with my requests for the projects they helped me complete. Nathan Clark, Shidhartha Das, Kevin Fan, Sangwon Seo, and Mark Woh were helpful co-authors on various publications. Thanks to all of my office-mates – Amin Ansari, Hyoun Kyu Cho, Kevin Fan, Shuguang Feng, Shantanu Gupta, Jeff Hao, Amir Hormati, Po-Chun Hsu, Ashlesha Joshi, Samuel King, Manjunath Kudlur, Dominic Lucchetti, Mojtaba Mehrara, Hyunchul Park, Yongjun Park, and Mark Woh – who were always prepared to discuss work, politics, history, languages and the world. Thanks also to the rest of my colleagues from the “Compilers Creating Custom Processors” group: Jason Blome, Gaurav Chadha, Michael Chu, Nathan Clark, Anousheh Jamshidi, Steven Lieberman, Yuan Lin, Andrew Lukefahr, Robert Mullenix, Rajiv Ravindran, Mehrzad Samadi, Ankit Sethia, Mikhail Smelyanskiy and Griffin Wright. Long live the CCCP quotes page! Finally, many thanks to Gabriel Black, Ronald Dreslinski and Timur Alperovich for their warm friendship.

My friends from outside of work provided a much-needed escape from the drudgery of graduate school; I won’t name all of them here, but I trust that they know who they are. Many fond moments were shared during the various movie nights, game nights and Eastern Flame nights.

Above all, the utmost of thanks go to my dear family for their unconditional love and support. Their encouragement kept me going through grad school.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF FIGURES viii

LIST OF TABLES xiii

ABSTRACT xiv

CHAPTER

I. Introduction 1

II. Background and Related Work 6

2.1 Architecture Styles 6
2.2 Application Domains of Interest 10

III. Voltage and Frequency Scaling for Loop-Accelerators 13

3.1 Introduction 13
3.2 The BLADES Architecture 18
3.2.1 Loop Accelerator Architecture 18
3.2.2 Applying Razor 19
3.2.3 Overheads of Using Razor 23
3.2.4 Shaving the Overheads of Razor 25
3.3 Experiments 28
3.3.1 Setup 28
3.3.2 Proof of Concept 29
3.3.3 Dynamic Frequency Scaling 30
3.3.4 Effect of CFUs 32
3.4 Related Work 33

3.5 Conclusion 33

IV. Flexible Loop-Accelerator Systems 35

4.1 Introduction 35
4.1.1 The Advantages of GPUs 36
4.1.2 The Disadvantages of GPUs 37
4.1.3 Programmable Loop Accelerators 38
4.2 Targeting Medical Applications 39
4.3 PUMA 41
4.3.1 Background 41
4.3.2 PUMA Architecture 44
4.3.3 PUMA System Architecture 49
4.4 Experiments and Results 50
4.4.1 Setup 50
4.4.2 PLA Characteristics 50
4.4.3 System Characteristics 52
4.4.4 Commodity GPGPU Comparison 55
4.5 Conclusion 57

V. Accelerator Systems for High-Throughput, Bandwidth-Constrained Applications 58

5.1 Introduction 58
5.2 Computational Requirements of Image Reconstruction 62
5.2.1 Benchmark Overview 66
5.2.2 Benchmark Analysis 68
5.3 The MEDICS Processor Architecture 70
5.3.1 FPU Pipeline 71
5.3.2 Data Compression 77
5.3.3 Memory System 79
5.4 Experimental Evaluation 82
5.4.1 FPU Chaining 82
5.4.2 Local Storage 85
5.4.3 Compression 88
5.5 The MEDICS System 90
5.6 Related Work 94
5.7 Conclusion 95

VI. General Accelerators for High-Throughput, Data-Parallel Applications 97

6.1 Introduction 97
6.2 Analysis of Scientific Applications on GPUs 101
6.2.1 Application Analysis 101
6.2.2 GPU Utilization 103

6.3 The PEPSC Architecture . . . 106
6.3.1 A Two-Dimensional SIMD Datapath . . . 107
6.3.2 Reducing Memory Stalls . . . 111
6.3.3 Reducing Serialization . . . 114
6.4 Results . . . 118
6.4.1 Datapath optimizations . . . 119
6.4.2 Memory System . . . 121
6.4.3 Serialization . . . 122
6.4.4 Application to GPU . . . 124
6.4.5 PEPSC Specifications . . . 125
6.5 Related Work . . . 126
6.6 Conclusion . . . 128

VII. Data-Parallel Accelerators to Extract Instruction-Level Parallelism . . . 130

7.1 Introduction . . . 130
7.2 Motivation . . . 134
7.3 SIMD-Morph . . . 137
7.3.1 Hardware . . . 137
7.3.2 Configuration . . . 139
7.3.3 Compilation . . . 140
7.3.4 Baseline Observations . . . 141
7.4 Design-Space Exploration and Results . . . 145
7.4.1 Varying Register Inputs . . . 146
7.4.2 Varying Number of Memory Units . . . 147
7.4.3 Varying Interconnect Topology . . . 149
7.4.4 Varying Register Outputs . . . 150
7.5 Results . . . 152
7.6 Related Work . . . 154
7.7 Conclusion . . . 156

VIII. Future Work . . . 157

8.1 Efficient CPU-GPU Fusion . . . 157
8.2 System Design for Portable, Low-Power Medical Ultrasound . . . 158

IX. Conclusions . . . 162

BIBLIOGRAPHY . . . 165

LIST OF FIGURES

Figure

2.1 Comparison of peak performance, power efficiency, and programmability of different architecture design styles. 7
3.1 Abstract view of the Razor flip-flop. 14
3.2 An example modulo-scheduled loop. 17
3.3 Hardware schema of a loop accelerator. 18
3.4 Loop accelerator design flow. 19
3.5 Structural modifications to a loop accelerator to support Razor. Data signals are dotted lines and control signals are solid lines. 20
3.6 Life-cycle of an error 22
3.7 The effect of using CFUs 24
3.8 Dataflow graph for a portion of the `fsed` loop kernel. Operations chained to form COPs are in the numbered, shaded regions. Some operations that were not selected to be combined are shown within dotted lines. 26
3.9 Benchmarks used in this work. 28
3.10 Energy savings and slowdown with voltage-scaling for the `sobel` benchmark. 29
3.11 Dynamic execution of a `sobel` BLADES processor. 30
3.12 `sobel` BLADES processor with 256 bytes of memory at 65nm. The accelerator communicates with an ARM1176 via an APB interface. 31
3.13 Dynamic energy savings when using DVS. The top portion is the contribution to this number by using CFUs. 31
4.1 PUMA. Each tile comprises of a programmable loop accelerator (template pictured) and the control and data memories required for its operation. On-chip routers transfer data between each tile and the external interface. 42
4.2 Template for single-function loop accelerator. 43
4.3 The PUMA ring architecture. Two bus segments, moving data in opposite directions, connect each bus/FU connector. In this 8-FU example, the longest move latency between any two FUs is 2 cycles. 45
4.4 ILP formulation for FU arrangement on the PUMA ring. 46
4.5 Normalized performance of benchmarks on LA and PUMA PLA designed for `MRI.FH`. 51

4.6 Normalized $\frac{Performance}{Power}$ efficiency of benchmarks relative to `MRI.FH`. . 52
4.7 Overall system performance for each of the PUMA systems. 53
4.8 Run-time of benchmarks running on different PLAs, normalized to that of the native benchmark. 53
4.9 Average energy consumed (per iteration) by each benchmark while running on PUMA systems designed around different PLAs. 54
4.10 Achieved performance of the MRI.FH benchmark (in trillions of operations) on the `MRI.FH` PUMA system and on various NVIDIA GPUs based on the GT200 architecture. 56
4.11 $\frac{Performance}{Power}$ efficiency improvement (in multiples, *not* in percentages) of running benchmarks on PUMA systems rather than on commodity GPUs. 56
5.1 Performance and power requirements for the domains of current and advanced image reconstruction techniques in both tethered and untethered environments. Diagonal lines indicate "Mops/mW" performance/power efficiency. For comparison, the peak performance and power of several commercial processors and GPGPUs are provided: Intel Pentium M, Intel Core 2, Intel i7, IBM Cell, Nvidia GTX 280, Nvidia GTX 295, and Nvidia Tesla S1070. 59
5.2 Capturing and deciphering x-rays for tomographic image reconstruction. (a) Detected x-ray attenuations vary based on the density of the object; the opaque cylinder allows much fewer x-rays to pass through it than the transparent cube. (b) Internal density values must be computed using x-ray attenuations measured by the detector array. (c) A slice of 3D helical x-ray CT scan reconstructed by the conventional FBP method (left) and by a MBIR method (right). For these thin-slice images, the MBIR method exhibits much lower noise than the FBP images, enabling better diagnoses. 63
5.3 Instruction type breakdown showing the % of instructions used for FP computation (FPU), loads and stores (Mem), address generation (AGU), control-flow (CF) and integer ALU (I-ALU). 69
5.4 Components required for a medical imaging compute system. 71
5.5 FPU architecture. (a) Internal structure for FPU. (b) The Normalizer stage may be removed for all but the last in a chain of FPUs. 73
5.6 Example using chained FPUs (CFPs) from the Radon benchmark. (a) Operation identification. (b) Latency hiding via software pipelining. . . . 74
5.7 Memory Interfaces. (a)Datapath-to-DRAM memory system. A large L1 or a smaller L1 with input and output streaming FIFOs may be used. (b)Off-chip memory system. 80
5.8 Datapath latency, power and area effects of varying the number of FPUs when increasing FPU chain length. 83
5.9 Speedup and FPU utilization with increasing chain length. 84

5.10 Local storage characteristics. (a) Energy consumption when using 8-entry and 32-entry RFs, normalized to energy consumption of 16-entry RF. (b) Power consumption of each additional RF context, using a FIFO streaming buffer and using an L1 cache to hide L2 memory latency. The FIFO and context-based solutions include an additional 16-byte/lane L1 cache for register spill. 86
5.11 Speedup using a FIFO streaming buffer instead of an L1 cache. 87
5.12 Image compression. (a) Datapath-to-DRAM memory system with two different possible configurations for using compression and decompression engines. (b) Degrading compression ratios when increasing the granularity of loss-less compression. 89
5.13 MEDICS processor architecture. The figure on the bottom-right conceptually illustrates the design of the entire chip. The figure on the left shows the architecture of an individual PE. The figure on the top-right shows the arithmetic execution pipeline. 90
5.14 MEDICS specifications. Overall per-PE specifications and power breakdown of individual components. 91
5.15 (a) *(Modified Fig. 5.1)* Suitability of MEDICS for the performance and power characteristics of the domain (b) Theoretical and realized (bandwidth-limited) run-times for advanced reconstruction algorithms. 93
6.1 Peak performance and power characteristics of several high-performance commercial processors and GPUs are provided: ARM Cortex-A8, Intel Pentium M, Core 2, and Core i7; IBM Cell; Nvidia GTX 280, Tesla S1070, and Tesla C2050; and AMD/ATI Radeon 5850 and 6850. 98
6.2 Static instruction type breakdown showing the % of instructions used for floating-point operations (FPU), loads and stores (Mem), address generation (AGU), special math library functions (SFU), control-flow (CF) and other instructions such as integer math operations and loads from constant memory (Other). 103
6.3 Benchmark utilization on a GTX 285 model. Utilization is measured as a percentage of peak FLOPs and is indicated as a number above each bar. The components of the bar represent a different source of stall cycles in the GPU. The mean utilization is 45%. 104
6.4 PEPSC architecture template. The shaded components are part of conventional SIMD datapaths. 106
6.5 Effect of length and width on the power efficiency of a 2D SIMD datapath, normalized to the 8-wide, 1-long case. 108
6.6 Chained FPU datapath with support for multiple subgraph execution. . . . 110
6.7 (a) Varying prefetcher degrees in the `lps` benchmark (b) Varying weighted prefetcher degrees in different benchmarks. 112
6.8 Dynamic-degree prefetcher. 113
6.9 An example diverging code fragment and the associated control-flow graph. The black and white arrows indicate which SIMD lanes are used and unused, respectively, when using an immediate-post-dominator-reconvergence strategy. 116

6.10 (a) Datapath latency, power and area effects of varying the number of FPUs when increasing FPU chain length. (b) Speedup with increasing chain length. The chain lengths are indicated in bold numbers below each bar; the dark portion above the "4" and "5" bars indicate the additional performance improvement from allowing multiple subgraphs to execute concurrently. 119
6.11 Comparison of DDP and degree-1 prefetching. 123
6.12 Comparison of prefetching from mem to L2 and mem to L2 to L1. 123
6.13 Speedup using various serialization-mitigation techniques. 124
6.14 Reduction in GPU overheads after cumulatively applying various techniques. "B" is the baseline bar, "D" is after adding the chained FPU datapath, "C" is after adding divergence-folding to mitigate control overhead and "M" is after adding the prefetcher to reduce memory latency. 124
6.15 Overall per-core specifications and power breakdown of individual components. 125
6.16 *(Modified Fig. 6.1)* Power-efficiency of PEPSC. Black points are peak performances, and the gray and white points represent different points of underutilization. 126
7.1 Performance vs. power requirements for various mobile computing applications. 131
7.2 A portion of the `rc4` benchmark. 134
7.3 A portion of the `crc` benchmark. 135
7.4 Fraction of cycles spent outside of inner-most loops. 136
7.5 Baseline SIMD+Scalar Processor. 137
7.6 SIMD-Morph Modifications. 137
7.7 Graphical representation of a portion of the `crc` benchmark on SIMD-Morph. 138
7.8 Distribution of subgraphs consisting of various numbers of operations. . . . 142
7.9 % Cycles saved by subgraphs consisting of various numbers of operations. 142
7.10 Distribution of subgraphs consisting of various depths of operations. . . . 143
7.11 % Cycles saved by subgraphs consisting of various depths of operations. . 143
7.12 Distribution of subgraphs consisting of various numbers of memory operations. 144
7.13 Performance impact of varying the number of register inputs. 146
7.14 Register file power with varying number of register inputs, normalized to the 4-port baseline. 146
7.15 Impact of varying the number of memory units on speedup. 148
7.16 Normalized power impact of varying the number of memory units relative to the 4-port baseline. Note that the y-axis does not start at 0. 148
7.17 Impact of varying the interconnect topology on speedup. 149
7.18 Normalized power impact of varying the topology relative to the 1x16 baseline. Note that the y-axis does not start at 0. 149
7.19 Impact of varying the number of register outputs on speedup. 150
7.20 Register file power with varying number of register outputs, normalized to the 2-port baseline. 150

7.21 Average performance improvements with varying configurations. 151
7.22 Average performance/power efficiency improvement of SIMD-Morph with the different datapath configurations. 152
7.23 Speedup from using SIMD-Morph using the optimal configuration. "Average media" refers to the average speedup obtained from accelerating the outer loops of Mediabench applications while "average other" refers to the average speedup obtained from accelerating the entirety of the other applications. The speedup seen in `pc1` is 9.5X but is capped at 4X in this graph. 153

LIST OF TABLES

Table

4.1 Medical application characteristics. 40
4.2 Characteristics of the individual accelerators for each benchmark. 50
4.3 GPU hardware characteristics. 55
5.1 Medical imaging application characteristics. 66
6.1 GPGPU-SIM configuration, matching the Nvidia GTX 285 as closely as possible. 104
6.2 Scientific application FPU operation-depth characteristics. 109
7.1 Configurations used in design-space exploration. Entries in bold indicate deviations from the baseline. 145

ABSTRACT

Power-Efficient Accelerators
for High-Performance Applications

by

Ganesh Suryanarayan Dasika

Chair: Scott Mahlke

Computers, regardless of their function, are always better if they can operate more quickly. The addition of computation resources allows for improved response times, greater functionality and more flexibility. The drawback with improving a computer's performance, however, is that it often comes at the cost of power and energy consumption. For many platforms, this is not an issue – desktop computers, for instance, have been consuming an increasing amount of power yet because they comprise a small fraction of a household's overall power consumption, this increase is tolerated.

Power increases are not tolerated, however, in modern mobile devices. These devices do more and more each generation and are deployed in increasingly interesting environments – whether it is the common smart phone, a portable medical imaging device or a complex sensor-processor used for disaster relief. Even in less portable circumstances,

such as servers used for high-throughput computation, reducing power consumption is an important way to reduce the cost of operating the server.

This thesis studies various applications from these and other domains that require ever-increasing compute ability but cannot tolerate a similar increase in power consumption. These domains include wireless signal-processing, portable medical imaging and scientific computing. These applications are analyzed in detail to ascertain the most appropriate hardware substrates and microarchitectural structures to help deliver the best performance-per-power efficiency.

The architectures evaluated included fixed-function and flexible loop accelerators, SIMD data-engines, general-purpose graphics-processing units and general-purpose processors. These architectures were improved in ways that increased programmability, instruction-level parallelism, memory bandwidth, and average throughput; and reduced control-divergence overhead, effective memory latency, and data-dependent stalls. These modifications led to solutions with 50 times the performance-per-power efficiency as current designs.

CHAPTER I

Introduction

Power and energy efficiency is a first-order design concern for all modern compute systems. Traditionally, the focus of power and energy optimizations have been on cellular phones and PDAs, optimized for improved battery life, or on laptop computers, optimized for improved battery life and for reduced surface temperatures. Lately, this has expanded to include desktop systems, servers and super computers for scientific and numeric computations. The motivation for improving their power efficiency is quite varied as well, including long-term reliability, reducing the monetary cost of wall-power consumption, reducing the cost of cooling systems, etc. The focus of this work is to improve the efficiency of processors, particularly those used to execute applications that have stringent performance requirements.

Three broad domains of applications were studied – untethered computing for wireless communication; portable and low-power medical imaging; and scientific, throughput-oriented computing. These domains are representative of compute-intensive applications and help focus our efforts.

Untethered computing for wireless communication: In the coming years, the deployment of untethered computers will continue to increase rapidly. The prime example today is the cell phone, but in the near future we expect to see the emergence of new classes of such devices. These devices will improve on the mobile phone by incorporating advanced functionality such as high-bandwidth Internet access, human-centric interfaces with voice recognition, high-definition video coding, and interactive conferencing. At the same time, we also expect the emergence of relatively simple, disposable devices that support the mobile computing infrastructure. The requirements of these low-end devices will grow in a manner similar to those for the high-end devices.

Untethered devices perform signal processing as one of their primary computational activities due to their heavy usage of wireless communication as well as rendering of audio and video signals. Fourth generation wireless technology (4G) has been proposed by the International Telecommunications Union to increase the bandwidth to maximum data rates of 100 Mbps for high mobility situations and 1 Gbps for stationary and low mobility scenarios like internet hot spots [77]. This translates to an increase of 10-1,000X in computational requirements over previous third generation wireless technologies (3G) with a power envelope that is increasing only 2-5x [109]. Other forms of signal processing, such as high-definition video, are also 10-100x more compute intensive than current mobile video.

Portable and low-power medical imaging: Medical imaging is one of the most effective tools used in modern medicine to aid physicians in diagnosing and analyzing ailments. Computed tomography, or CT, employs geometry processing to generate a three-

dimensional image of the inside of an object from a large series of two-dimensional x-ray images taken around a single axis of rotation. More than 62 million scans are ordered each year to detect ailments ranging from brain tumors and lung disease to guiding the passage of a needle into the body to obtain tissue samples [28]. Compared to traditional 2D x-rays, CT has inherent high-contrast resolution to accurately detect differences in tissue density of less than 1%. Further, the data is highly flexible in that it can be aggregated and viewed in all three planes depending on the diagnostic task. Other popular medical imaging techniques use varying methods to acquire images, e.g., other forms of radiation or radio frequency signals, including single photon emission computed tomography (SPECT), positron emission tomography (PET), and magnetic resonance imaging (MRI).

From a computer architecture perspective, the challenging aspect of medical imaging is the vast amount of computation that it requires. This computation is floating-point-intensive and utilizes a large amount of data. Portable image reconstruction requires an order of magnitude less performance than non-portable reconstruction, but also has a substantially lower power budget to operate in a less tethered environment.

Scientific computing: Scientists and mathematicians are increasingly realizing the computational benefits of using modern, multi-core architectures. In response to this, manufacturers of traditional desktop graphics-processing units (GPUs) have evolved their architectures to create desktop and server GPGPUs (General Purpose Graphics Processing Units). These GPGPUs are quickly becoming the platform of choice for many high-performance, highly parallel applications. GPGPUs are also commodity hardware products commonly available in many desktop and laptop computers, making them rather inexpensive. The

tools to program them are easily available as well; Nvidia's Compute Unified Device Architecture (CUDA) package, for example, provides a small set of extensions to the C programming language, allowing for straightforward implementation of parallel algorithms on GPGPUs. Individual cores in Intel's many-core designs implement the ubiquitous x86 ISA, allowing users to use a host of already-existing development tools to port their applications to it. Server products like the Nvidia Tesla S1070 with even more compute power are also available.

Applications from a wide variety of domains such as medical imaging, electronic design automation, physics simulations, and stock pricing models observe remarkable speed-ups on GPUs – at times, over 300X. Based on these dramatic performance increases, GPGPUs seem like an ideal computing substrate for high-performance, scientific computing. However, there are two major problems with GPGPUs – power consumption and an unbalanced ratio of compute ability to memory bandwidth.

While power is not necessarily a significant drawback for video game graphics acceleration, requiring powerful cooling systems is a significant impediment for more portable platforms. Some of the applications that are sped-up with GPUs use them to accelerate underlying algorithms that are often deployed in systems where portability or power consumptions is a critical issue. For instance, polynomial multiplication is used in very advanced cryptographic systems, real-time FFT solving is required for complex GPS receivers, and low-density parity-check error correcting codes are used in WiMAX and WiFi. Monte Carlo Recycling algorithms for computational finance, are often deployed in dense urban areas like Manhattan where, while portability is not an issue, power and cooling certainly are important concerns.

This thesis proposes various solutions to reduce the power consumption of devices used for high-throughput computing. Chapter III proposes a fixed-function solution to reduce the power consumption of non-programmable accelerators which can be used for efficient wireless signal processing [24].

Chapter IV looks at medical imaging – specifically CT and MRI reconstruction – and proposes a programmable accelerator solution which retains much of the efficiency of a fixed-function accelerator while still being programmable [25].

Chapter V proposes a fully programmable system for this domain which exploits the inherent data-level parallelism available, and also addresses the memory bandwidth and latency constraints of the domain [27].

Chapter VI analyzes the more general compute problems faced by scientific and numeric computing and proposes an architecture that is an order of magnitude more power efficient than existing solutions [26].

Data-level parallelism is often in abundance in the applications studied in this work. Chapter VII explores how non-data-parallel code in traditionally data-parallel domains can be accelerated using a SIMD engine that can also extract instruction-level parallelism.

Chapter VIII explores some future directions of this work. These include a power-efficient solution for throughput-oriented scientific computing and a low-power processor for extremely portable medical ultrasound devices.

CHAPTER II

Background and Related Work

2.1 Architecture Styles

A wide range of architectures have been designed to address the problem of providing high performance computation in a power-efficient manner. These solutions maintain or sacrifice programmability to various degrees depending on the domain they target. Figure 2.1 shows the performance (on the y-axis) and programmability (on the x-axis) expectations from various architecture styles. The numbers next to each of the ovals shows the approximate performance-power ratio offered by each of these solutions.

General purpose processors (GPPs) which fall on the lower right corner of the figure, are highly programmable solutions but are limited in terms of the peak performance they can achieve. Further, structures like instruction decoders and caches that are needed to support programmability consume energy. This results in a very low computational efficiency of about 1 MIPS-per-mW, for example, for the Intel Pentium-M processor.

On the other end of the spectrum are Application-specific Integrated Circuits (ASICs). ASICs are custom-designed specifically for a particular problem, without extraneous hard-

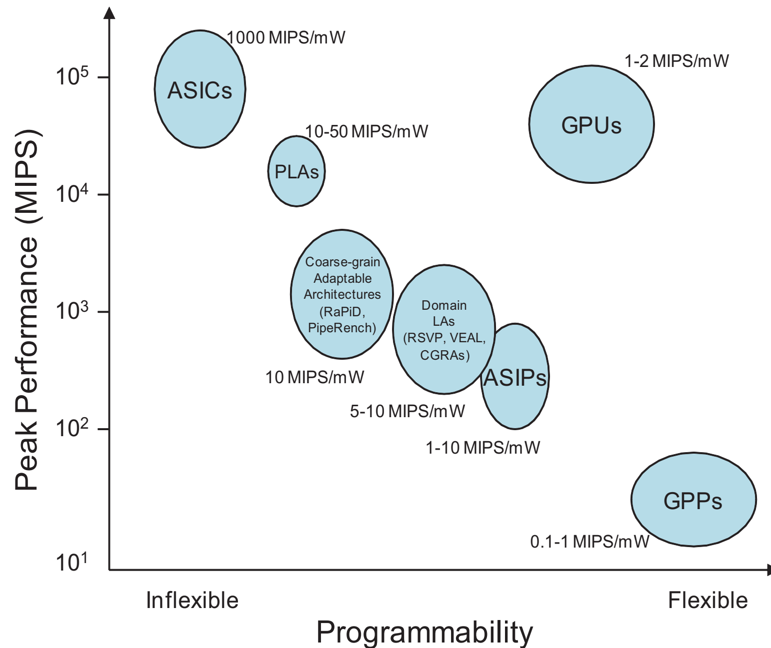


Figure 2.1: *Comparison of peak performance, power efficiency, and programmability of different architecture design styles.*

ware structures. Thus, ASICs have a high computational density with hardwired control, resulting in high computation efficiency up to 1,000 to 10,000 times more than that of GPPs. The space between these two extremes is populated by different solutions that have varying degrees of programmability.

Digital signal processors [97, 98, 62] (DSPs) increase computation efficiency by providing specialized features that optimize the execution of signal-processing algorithms. These features include special arithmetic operations like multiply-accumulate and bit manipulation operations, hardware modulo addressing, limited data/control-flow hazard detection in hardware and memory architecture optimized for streaming data. A wide range of DSP algorithms can be executed efficiently on these processors efficiently. DSPs typically offer an order of magnitude increase in power efficiency.

Single-instruction multiple-data (SIMD) accelerators are commonly used in microprocessors to accelerate the execution of media applications. These accelerators perform the same computation on multiple data items using a single instruction. To utilize these accelerators, the baseline instruction set of a processor is extended with a set of SIMD instructions to invoke the hardware. Intels MMX and SSE extensions are examples of two generations of such instructions for the x86 instruction set architecture (ISA). SIMD accelerators are popular across desktop and embedded processor families, providing large performance gains at low cost and energy overheads.

Application specific instruction-set processors (ASIPs) are processors with custom extensions for a particular application or application-domain. They can be quite efficient when running the applications for which they are designed, and they are also capable of running any other application, though with reduced efficiency. Examples include processors from Tensilica [96] and ARC [2], transport triggered architectures [21] and custom-fit processors [34].

Domain loop accelerators are designed to execute computation intensive loops present in media and signal processing domains. Their design is close to that of VLIW processors, but with a much higher number of function units (FUs), and consequently, a higher peak performance. Very long instruction words in a control memory direct all FUs every cycle. However, domain loop accelerators (LAs) have less flexibility than GPPs because only highly computationally-intensive loops map well to them. Some examples of architectures in this design space are VEAL [17], RSVP [13] and CGRAs [59].

FPGAs have fine grain logic blocks that can be reconfigured to perform various bit level logic and arithmetic functions. The fine grain reconfigurability allows FPGAs to be

very flexible. Bit parallel computations in domains like encryption can be performed very efficiently. However, complex integer and floating point operations do not map well on to FPGAs as not all FPGAs have dedicated floating-point units. Thus, for some domains, FPGAs are very flexible and highly efficient.

Coarse-grain adaptable architectures have coarser-grain building blocks compared to FPGAs, but, like FPGAs, still maintain bit-level reconfigurability. The coarser reconfiguration granularity improves the computation efficiency of these solutions. However, non-standard tools are needed to map computations onto them and their success have been limited to the multimedia domain. CGRAs [59], PipeRench [39], RaPiD [31] are some examples of coarse-grain adaptable architectures.

GPUs have many appealing hardware features. Firstly, they lend themselves very well to both thread-level and data-level parallelism. Thread-level parallelism (TLP) is exploited by having a large number of independent processing elements (PEs) on the GPU, each with its own set of FUs and local storage. Individual threads can quite cleanly be assigned, either statically by the programmer or dynamically by the hardware, to each of these PEs and inter-thread communication is made possible by some form of interconnect fabric or through local storage such as caches. Programs with a large amount of data-level parallelism (DLP) can make use of vector-SIMD units in these PEs which allow a single instruction to perform an operation on several data at the same time. DLP can also be extracted in programs with compute-intensive loops that have little or no inter-iteration dependencies by executing operations from different iterations within a single SIMD instruction.

2.2 Application Domains of Interest

These architectures were evaluated by their performance-per-power efficiency while executing applications from a number of emerging domains such as medical imaging, scientific computing and applications relevant for modern smart-phones.

Portable medical imaging is an area of growing interest in the medical community for a variety of reasons. Recent medical studies have shown a marked improvement in patient health when using portable CT scanners and MRI machines, especially in emergency and critical cases. Conventional CT scanners require a very large amount of power to operate, the majority of which is for the x-ray emitters themselves which consume several kilowatts of power. However, due to the ever increasing number of CT scans performed on patients, there is growing concern about the effects of elevated radiation exposure and, consequently, increased interest in reducing the intensity and power of the x-rays. Using low-power and low doses of x-rays, however, requires more compute-intensive, iterative techniques to compensate for the associated artifacts.

Modern medical imaging computation is done through a combination of server-class processors and ASICs. This application domain is generally characterized by heavy floating-point computation, high bandwidth requirements, and large amounts of data-level parallelism. An emerging class of image reconstruction applications require orders of magnitude more performance and, as such, are generally targeted towards GPGPUs. In a more power-constrained environment, GPGPUs are generally underutilized, inefficient and, ultimately, unsuitable. Compute systems to best execute this class of applications, based on SIMD architectures and flexible loop accelerators are studied in this work.

Similar to medical imaging, but more general, is the broader class of scientific and numeric computation – another domain where GPGPUs have made significant progress. Power is not necessarily a significant drawback for video game but is a significant concern for more portable, or more dense, platforms. Some of the algorithms that are commonly accelerated by GPUs are often deployed in systems where portability or power consumption is a critical issue. Many aspects of GPGPUs and other data-parallel architectures that make them unsuitable for scientific computation, especially in power-constrained environments. For instance, the applications are memory intensive, but GPGPUs tend to employ inefficient means to hide memory latency; the applications often exhibit control divergence, but data-parallel architectures are least efficient when executing non-straight-line code. Architectural and microarchitectural modifications to current SIMD and GPGPU architectures are studied to help accelerate scientific applications in a more power-efficient manner.

The design of the next generation hand-held mobile platforms must address three critical issues: efficiency, programmability and adaptivity. The inherent computational efficiency of 3G solutions is insufficient and must be increased by at least an order of magnitude. Straight-forward scaling of 3G solutions by techniques such as increasing the number of cores is part of the solution, but is not enough on its own. Programmability provides the opportunity for a single platform to support multiple applications and even multiple standards within each application domain. Further, programmability provides faster time to market as hardware and software development can proceed in parallel, the ability to fix bugs and add features after manufacturing, and higher chip volumes as a single platform can support a family of mobile devices. Lastly, hardware adaptivity is necessary to maintain efficiency as the core computational characteristics of the applications change. 3G

solutions rely heavily on the vast amounts of vector parallelism in wireless signal processing algorithms, but lose most of their efficiency when vector parallelism is unavailable or constrained. This thesis explores the use of SIMD architectures and ASICs to help accelerate the mobile applications of tomorrow while allowing a phone battery to last much longer.

CHAPTER III

Voltage and Frequency Scaling for Loop-Accelerators

3.1 Introduction

As silicon technologies enter deep sub-micron realms, circuit level techniques are increasingly employed in conjunction with architectural techniques to achieve the stringent performance and power goals of embedded applications. Dynamic voltage and frequency scaling (DVFS) are widely used to reduce the overall energy consumption of a computer system, particularly for workloads with high variation in processing requirements. DVFS can either be used to push the operating conditions of a circuit beyond the nominal operating conditions assumed during design time to achieve improved clock frequency, or to reduce energy consumption at times when the full capabilities of the hardware are not required.

A critical issue for DVFS-enabled computer systems is determining the safe operating voltage at which maximum execution efficiency is achieved, while still guaranteeing correct operation of all components. Traditional techniques for DVFS utilize a delay chain or a lookup table to determine the minimum voltage necessary to guarantee error-free operation

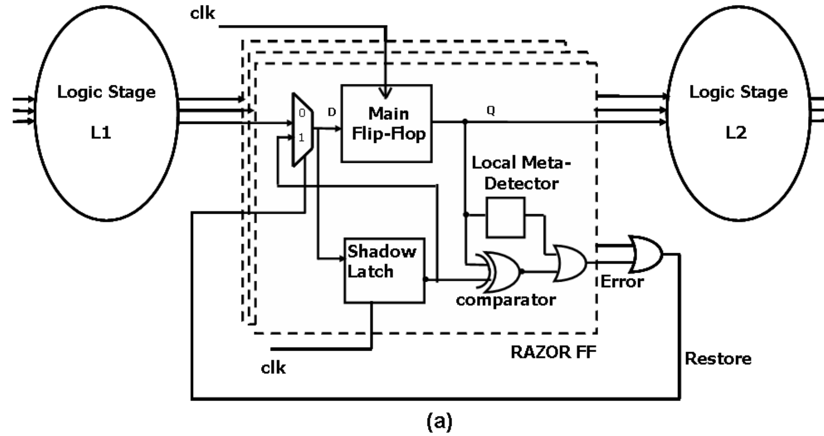


Figure 3.1: *Abstract view of the Razor flip-flop.*

at a particular frequency [10]. Design-time characterization of the critical paths determines the margins that need to be added in order to ensure that the synthetic delay path is guaranteed to fail before the actual paths in the presence of worst-case operating conditions, process variation, temperature hot spots, and supply voltage uncertainties.

The Razor [23, 7] flip-flop, shown in Figure 3.1, is a cost-effective, circuit-level timing speculation technique that allows detection and correction of speed-path failures. In the traditional worst-case design technique, safety margins are added during design time to ensure computation correctness even under the worst-case combination of input vectors and operating conditions (process, voltage and temperature conditions). Razor is able to exploit these margins when operating under more typical conditions by aggressively scaling voltage and frequency and relying on a combination of in-situ architectural and circuit-level techniques to suitably flag any resultant timing errors and recover from them.

In large designs, the Razor shadow latch cannot be used if complex control signals are on the critical path. For example, if critical clock-enable signals going into clock-gating cells are too slow, the affected flip-flops will not be able to restore to their correct

state as they will have been clocked incorrectly and their shadow latches will not have the correct data either. In situations like this, the most suitable way to recover from Razor errors is to add extra storage elements for checkpointing the microarchitectural state from which the processor can be restored to a point before the error. Leaving data in flight in this manner complicates a processor's control and forwarding logic which is often on the critical path. Checkpointing for Razor error recovery requires techniques specific to the microarchitecture under consideration. This technique is invasive and, as such, requires careful analysis of individual microarchitectures before Razor can be employed.

Another difficulty of deploying this technology is that the Razor flip-flop's hold-time is much larger than that of a conventional flip-flop. Any fluctuations shortly after the rising clock edge are treated as timing errors. As a result, short paths to a Razor flip-flop must be lengthened by adding buffers to ensure the hold-time constraint is met on all paths. This can be costly in terms of area and energy consumption, thereby impacting the gains achieved through DVFS.

In essence, Razor would be more easily deployed in architectures that have predictable control logic to quickly compute a restore point; storage structures for checkpointing where data is guaranteed to not be overwritten for several cycles; and a regular microarchitecture with predictable path lengths to enable the easy application of Razor. Further, it is important to have an automated system to insert Razor flip-flops and the supporting hardware in order to reduce design time.

In essence, Razor would be more easily deployed in architectures that have the following characteristics:

1. Predictable control logic, to quickly compute a restore point.

2. Storage structures for checkpointing where data is guaranteed to not be overwritten for several cycles.

3. A regular microarchitecture with predictable path lengths to enable the easy application of Razor.

Further, it is important to have an automated system to insert Razor flip-flops and the supporting hardware in order to reduce the design time for a processor.

Razor's functionality is important in more than just general purpose processors. The market for portable devices that include cellphones, cameras, and PDAs is growing explosively. Convergence of various functionalities like audio, video, and wireless processing onto these portable devices requires very high performance computing systems that must operate under strict power budgets to ensure adequate battery lifetime. Specialized hardware in the form of loop accelerators (LAs) are commonly used along side general purpose processors for the compute-intensive portions of applications for which software implementations on a processor are too inefficient. Low power and high performance design, systematic verification, and short time-to-market are crucial objectives for designing these accelerators. Automated synthesis of accelerators from high level algorithmic descriptions has the potential to meet these objectives.

The focus of this paper is an automated system to synthesize Razor-enabled loop accelerators (LAs) from high-level specifications. LAs are ideal for applying Razor technology due to their regularity, queue-based storage structures, and simple control. BLADES (Better-than-worst-case Loop Accelerator Design) is an energy-efficient, application-specific

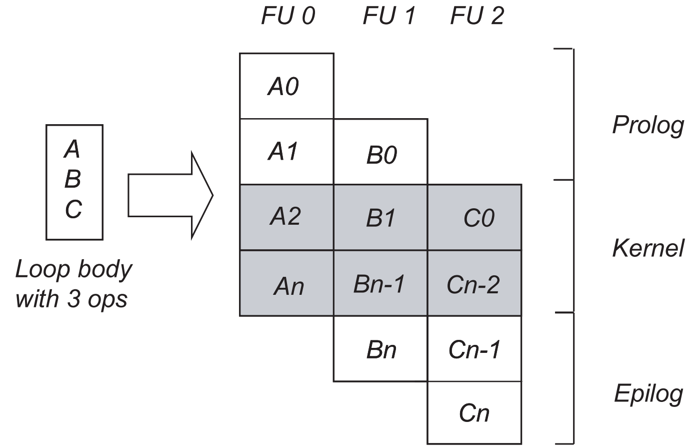


Figure 3.2: *An example modulo-scheduled loop.*

solution that adapts its operation to the environmental conditions, including silicon variation, temperature, and data precision. The inputs to BLADES are the target application expressed in C and the desired performance. Compiler analyses and scheduling are used to synthesize a minimum cost LA for the application to meet the given performance target [33].

We extend a baseline LA system with an application specific error recovery mechanism that is automatically derived. The LA is augmented with additional registers to enable efficient rollback and re-execution when a timing violation is detected. Further, we augment the system to automatically chain primitive computation operations together to reduce the overhead of ensuring that flip-flop hold-time constraints are met.

Across a large set of media and signal processing loops, LAs with Razor technology achieved an average energy savings of 32% with voltage scaling.

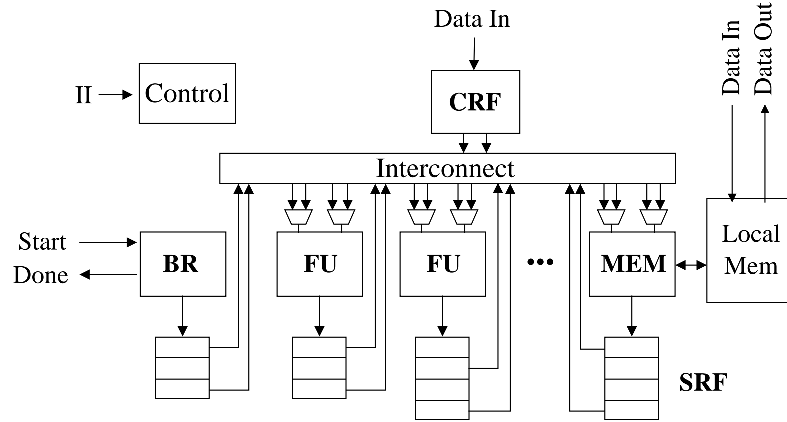


Figure 3.3: *Hardware schema of a loop accelerator.*

3.2 The BLADES Architecture

3.2.1 Loop Accelerator Architecture

The loop accelerator is a hardware realization of a modulo-scheduled loop. Iterative modulo scheduling (IMS) [76] is a software pipelining technique that interleaves successive iterations of a loop. The goal is to find a valid schedule for an innermost loop that can be overlapped with itself multiple times so that a constant interval between successive iterations (Initiation Interval or II) is minimized. The modulo schedule contains a *kernel* which repeats every II cycles and may include operations from multiple loop iterations as shown in Figure 3.2. The pipeline is initially filled in the prolog region and drained in the epilog region of the schedule. The central benefit of modulo scheduling is that the desired throughput (II) can be specified as an input, and then the compiler scheduler attempts to find a valid schedule at that II value.

The LA used in this work is a hardware realization of a modulo-scheduled loop. Modulo-scheduling is a software pipelining technique that achieves high levels of parallelism by overlapping successive iterations of a loop [76]. The template for the baseline LA archi-

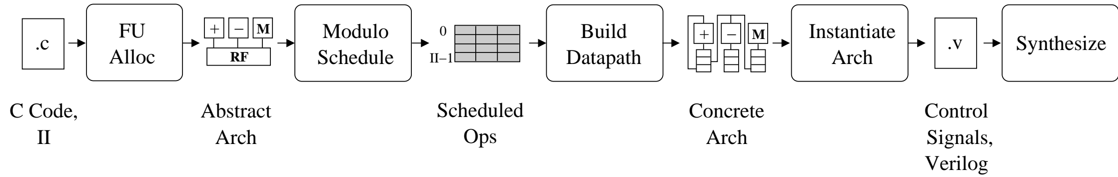


Figure 3.4: *Loop accelerator design flow.*

tecture is shown in Figure 3.3 [33]. The LA is designed to exploit the high degree of parallelism available in modulo-scheduled loops with a large number of functional units (FUs). Each FU writes to a dedicated shift register file (SRF); in each cycle, the contents of the registers shift downwards to the next register. The entries in a SRF therefore contain the values produced by the corresponding FU in the order they were computed. In addition, a central register file (CRF) holds static live-in (live-out) register values received from (sent to) the host processor which cannot be stored in the SRFs. Multiple registers may be connected to each FU input in which case a multiplexer (MUX) is used to select the appropriate one. Since the operations executing in a modulo scheduled loop are periodic, the selector for this MUX is simply a modulo counter.

The synthesis system for this architecture takes an application loop in C and generates behavioral Verilog for a hardware accelerator that implements the loop within a given performance constraint. The performance requirement is specified as an Initiation Interval (II), the number of cycles between the start of execution of subsequent iterations of the loop.

3.2.2 Applying Razor

Razor flip-flops are employed in LAs to facilitate aggressive DVFS beyond the point of erroneous operation. To ensure proper operation, errors must be dynamically detected and the erroneous operations must be re-executed. Razor is supported in the LA architecture

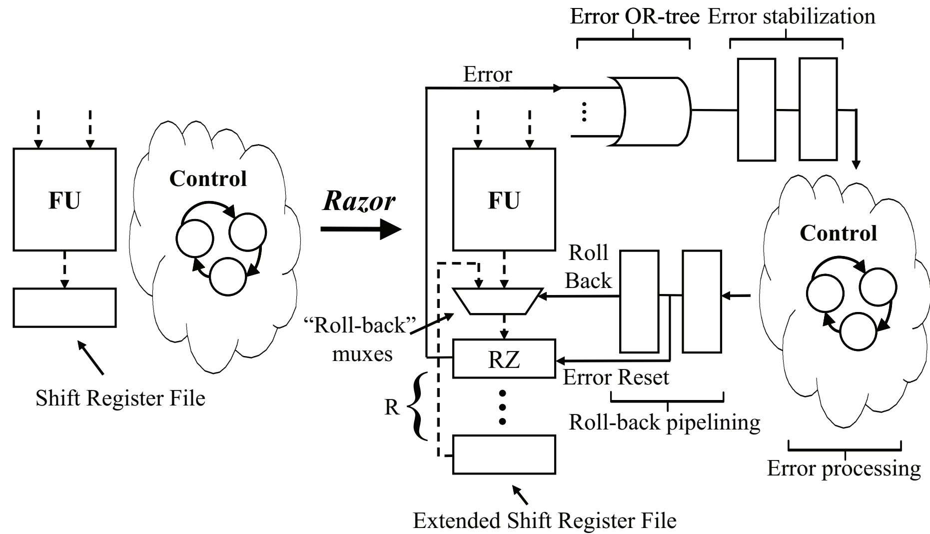


Figure 3.5: *Structural modifications to a loop accelerator to support Razor. Data signals are dotted lines and control signals are solid lines.*

with three important changes as illustrated in Figure 3.5: extending the shift register files, adding roll-back multiplexers and incorporating an error-cognizant controller.

First, all SRFs are extended to keep data values live for more cycles. Each SRF need only be extended as per the roll-back penalty, *R*, of the design – the number of clock cycles spent detecting timing errors and restoring all the registers to a previously-known state.

The second extension is a set of "roll-back multiplexers" that are added in order to support restoration of prior state. When an error occurs, an SRF entry is restored to its value stored *R* cycles earlier using the additional entries in the SRF. In this manner, all FUs are re-executed with their inputs from *R* prior cycles. Similarly, the memory units in the BLADES architecture write their outputs to store queues first and these values are only committed to main memory after *R* cycles, when the addresses and values are known to be error-free. In the event of a Razor error, the pipeline state in the FUs is disregarded and execution resumes from an older state.

The final extension for a Razor LA is in the controller and its supporting logic. As stated earlier, the LA is a hardware implementation of a modulo-scheduled loop and is, as such, statically scheduled and control signal values for all cycles of executions are all statically determined. This makes reverting back to a prior state simple – if an error was detected while the LA is executing, the controller reverts to a state *R* cycles earlier.

The different aspects of the design that determine the value of *R* are illustrated in Figure 3.5:

1. FU pipeline: Values in pipeline registers are not saved. Therefore, on an error, all data in a FU's pipeline registers have to be discarded, and the corresponding operations re-executed. In this work, the maximum pipeline depth for any FU is 2 cycles, but this number may vary between designs.

2. Error OR-tree: The error signals from all the Razor flip-flops across the design are OR'd together to create one unified error signal. If the design has a very large number of Razor flip-flops, the fan-in to this OR gate can be quite large, requiring the gate to be pipelined. This is never the case in this work.

3. Error stabilization: In some corner cases, it is possible for the OR'd error signal to be metastable. To overcome this, the signal is passed through 2 flip-flops.

4. Error processing and roll-back generation: The control sees the error signal, prepares to revert to a previous state and sends a "roll-back" signal to the SRFs.

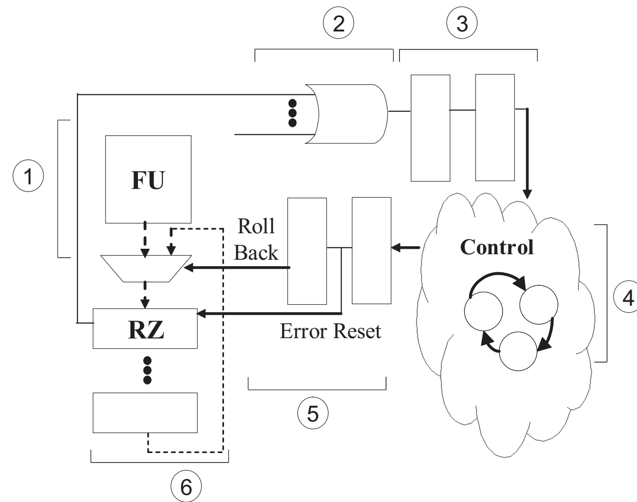


Figure 3.6: *Life-cycle of an error*

5. Roll-back pipeline: The “roll-back“ signal is pipelined in order to prevent the recovery process from being timing critical. A cycle after it is generated by the controller, all the error flags in the Razor flip-flops are reset.

6. Roll-back: A cycle after the error flags are reset, the old data is restored. If the restoration process fails, error flags are set again and the process restarts.

As stated earlier, the value of R is the time it takes to detect and error and then revert the state of the LA to a prior, known state. The different aspects of the design that govern this value are illustrated in Figure 3.6:

1. On an error, all data in an FU’s pipeline registers have to be discarded, and the corresponding operations re-executed, since individual pipeline registers are not saved over time in the same manner that data in SRF registers are. Therefore, increasing the level of pipelining in FUs increases R. In this work, the maximum pipeline depth for any FU is 2 cycles.

2. The error signals from all the Razor flip-flops across the design are OR'd together to create one unified error signal. If the design has a very large number of Razor flip-flops, the fan-in to this OR gate can be quite large, requiring the gate to be pipelined. This is never the case in this work.

3. In some corner cases, it is possible for the OR'd error signal to be metastable. To overcome this, the signal is passed through 2 flip-flops.

4. The control sees the error signal, prepares to revert to a previous state and sends a "roll-back" signal to the SRFs.

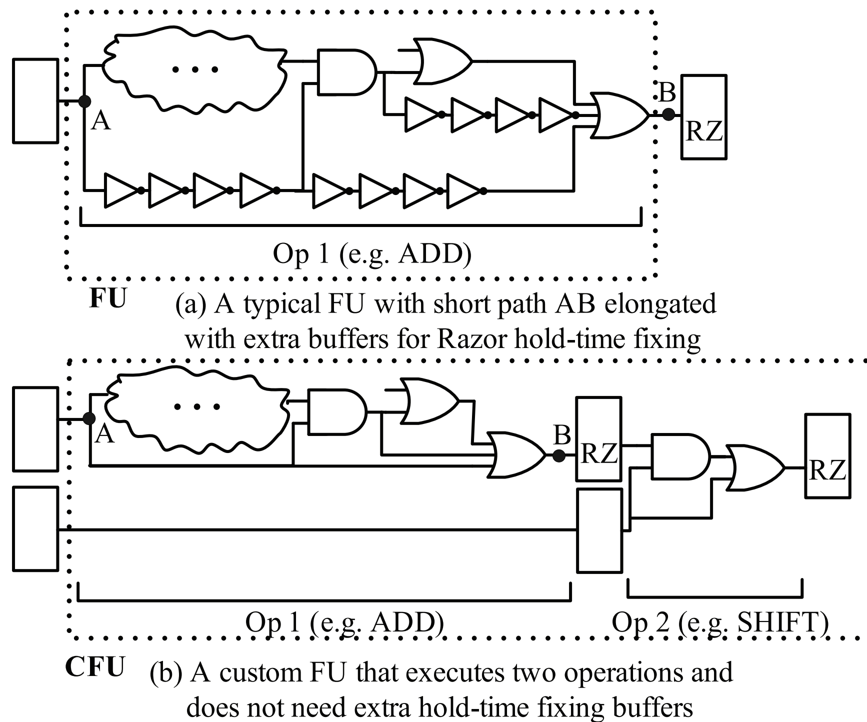
5. The "roll-back" signal is pipelined in order to prevent the recovery process from being timing critical. A cycle after it is generated by the controller, all the error flags in the Razor flip-flops are reset.

6. A cycle after the error flags are reset, the old data is restored. If the restoration process fails, error flags are set again and the process restarts.

In the designs presented in this work, *R* is 6 cycles (2 cycle-deep FU pipeline + 2 cycle error stabilization + 2 cycle error reset and roll-back).

3.2.3 Overheads of Using Razor

One of the main challenges to overcome when deploying Razor is the increased hold-time constraint. During the speculation window (the period of time after the rising clock edge where a change in the input signal is treated as a timing error), if the outputs of short paths in the design change correctly due to changing inputs, their change in value might



(a) A typical FU with short path AB elongated with extra buffers for Razor hold-time fixing

(b) A custom FU that executes two operations and does not need extra hold-time fixing buffers

Figure 3.7: *The effect of using CFUs*

be misinterpreted as a timing error. To avoid this, the shortest path in the design should be longer than the speculation window. To ensure this, extra buffers are inserted by the synthesis and place and route tools along the short paths.

For example, in Figure 3.7(a), the different paths connecting point A to point B would have several buffers inserted on them since there are very few gates on these paths. Based on the size of the design and the number of short paths, the energy overhead can be quite large; over 20% if the speculation window is 40% of the clock period.

There are a few different ways to circumvent the Razor hold-fixing problem. One way is to not issue back-to-back operations on the same FU. This would prevent the inputs to the FU from changing every cycle provided each FU's inputs are gated. However, this would halve the performance and would not be an appropriate solution in most situations. The performance need not be hurt if the number of FUs was doubled, each executing a

new operation every 2 cycles. This would preserve the throughput of the LA. This method would not double the area since only the number of FUs would have to double and not the controller or the number of SRFs and CRFs, but this technique would certainly have a noticeable area cost for reducing the dynamic energy consumption. Another technique is a hybrid solution – identifying opportunities to create multiple-operation FUs that require no hold fixing latches; and, when that is not possible, resorting to using buffers to fix the increased hold time.

3.2.4 Shaving the Overheads of Razor

To address the hold-fixing problem, we propose to combine multiple FUs into 2-cycle "custom functional units" (CFUs). A CFU uses 2 cycles to execute 2 or more operations back-to-back. Its inputs are only changed every 2 cycles and the values computed after 1 cycle are stored in a register. This way, the values that fan-in to Razor flip-flops are guaranteed not to change during the speculation window unless there is a timing error, thereby eliminating the need to insert extra buffers, as shown in Figure 3.7(b). A 2-operation CFU is illustrated in Figure 3.7(b), but a third operation could also be executed, in parallel to op 1 and feeding op 2.

These CFUs are identified by the compiler after analyzing the dataflow between different instructions. The input to the compiler are custom operation (COP) graphs of different dataflow patterns and opcodes; each node in the graph represents an input, an output or an opcode and each edge represents dataflow. The compiler finds subgraphs in the loop kernel that are isomorphic to the input CFU graphs and converts these subgraphs to COPs.

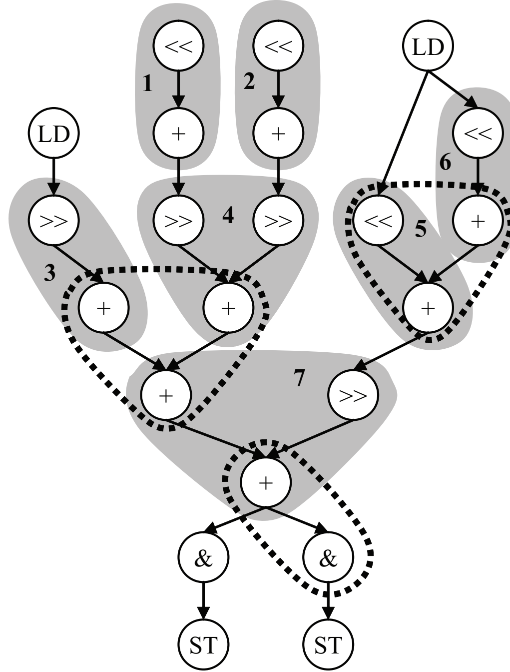


Figure 3.8: *Dataflow graph for a portion of the* `fsed` *loop kernel. Operations chained to form COPs are in the numbered, shaded regions. Some operations that were not selected to be combined are shown within dotted lines.*

When the hardware for these different CFUs is generated, CFUs that implement different opcodes, but have the same COP graph, can be combined into a single FU. For example, the operations

```
r0 = (r1 * r2) - (r3 + r4)
r5 = (r6 + r7) - (r8 + r9)
```

can be executed on the same FU. Adding an extra adder and some simple control logic can extend an MPY/ADD/SUB CFU to also execute ADD/ADD/SUB operations. Overlapping similar COPs in this manner reduces the overall number of FUs and therefore reduces the number of SRFs in the design at the cost of a few more muxes. For the purpose of this paper, the opcodes supported by each COP are limited to binary arithmetic operations since these make up the vast majority of opcodes used in loop kernels.

Figure 3.8 shows an example of the COPs identified in a portion of the `fsed` benchmark's inner-loop. The operations in shaded areas 1, 2, 3, 5 and 6 are combined to form 3-operand, 2-operation CFUs and the ones in shaded areas 4 and 7 are combined to form 4-operand, 3-operation CFUs. Depending on the schedule, COPs 4 and 7 could potentially be assigned to the same CFU due to their dataflow and functional similarities. Similarly, COPs 1, 2, 3, 5, and 6 could also be combined.

A loop kernel may have several different COPs that are not chosen to be implemented as a CFU. Examples of some of these are shown inside dotted lines in Figure 3.8. The selection algorithm used in this paper is a greedy algorithm that selects the largest possible subgraphs searching upwards from the bottom of the loop. The final goal is to try to encompass as many operations into CFUs as possible while minimizing the number of unique individual opcodes, giving priority to fewer, larger CFUs over many, smaller CFUs.

COP 5, for example, could have been expanded to include the addition operation in COP 6. However, this would leave the left-shift operation in COP 6 alone and a separate left-shift FU would be required to implement it. The ADD/AND COP circled at the bottom of the figure could be selected but priority would be given to the larger COP 7. Further, since an AND FU would still be required to implement the one remaining AND operation, creating an ADD/AND CFU would not necessarily have any benefit.

An optimal branch-and-bound-based algorithm was also implemented to select subgraphs, but was not used because it resulted in no noticeable difference in quality-of-result (QoR) but required significantly more time to execute.

Kernel	#Ops	#COPs	Domain
dequant	46	5	Video processing
idct	123	24	
fsed	37	7	Image processing
sharp	51	4	
sobel	40	2	
systolic_dct	70	16	
heat	20	1	DSP
rls	54	8	
turbo_decoder	142	12	Error correction
viterbi	111	5	

Figure 3.9: *Benchmarks used in this work.*

3.3 Experiments

3.3.1 Setup

The designs in this paper were synthesized at their maximum possible frequencies (i.e at the point where increasing the target frequency did not lead to improved QoR) with Design Compiler 2007-03, and Galaxy ICC 2007-03 using a 65nm process with a nominal supply voltage of 1 Volt. Simulations to observe error rates were performed in Nanosim 2005-09. Energy numbers were obtained using Nanosim and PrimeTime-PX 2006-12. The synthesis target assumed slow conditions (0.9V, slow silicon, 125°C) and simulations assumed typical conditions (1.0V, typical silicon, 25°C).

Figure 3.9 lists the benchmarks and several important characteristics of each: the number of operations (column 2), the number of custom operations identified (column 3) and the domain they come from (column 4).

The LA synthesis system is based on the Trimaran [104] compiler infrastructure, augmented with a synthesis-aware modulo scheduler and a backend for constructing the LA datapath, inserting Razor hardware, and emitting RTL for synthesis and simulation. Three

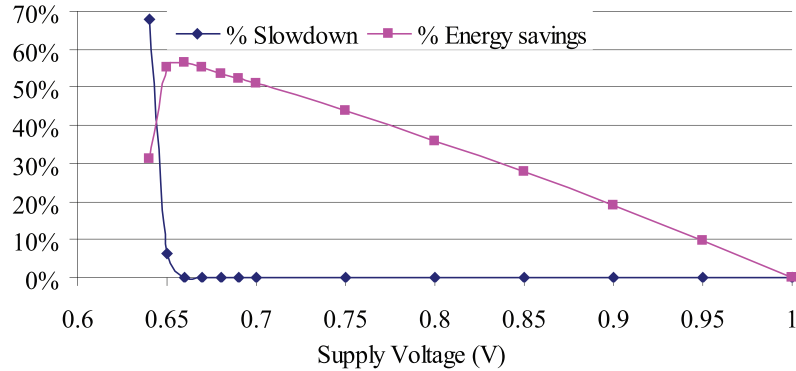


Figure 3.10: *Energy savings and slowdown with voltage-scaling for the* `sobel` *benchmark.*

different LAs were created for each benchmark; one without any Razor flip-flops, one with Razor flip-flops and one with Razor flip-flops and CFUs.

LA hardware was synthesized for ten compute-intensive loop kernels from various application domains. In the designs presented in this paper, every FU was followed by a Razor flip-flop - i.e. all the bits in all the top entries in SRFs were Razor flip-flops and all the others were normal D-flip-flops. This would likely be too conservative at mature process nodes, but in the worst-case scenario where process variation is a significant problem, it is reasonable. In a mature process, only the the most critical paths need to terminate in Razor flip-flops.

3.3.2 Proof of Concept

A simple experiment illustrates that an LA's behavior matches that of a general purpose processor when applying DVFS with Razor. An LA for `sobel` – an edge detection kernel often seen in image processing applications – was run at various voltage levels and at the nominal clock frequency. Whenever an error occurred, the LA was rolled-back 6 cycles, the clock frequency was halved, and the offending operation was re-executed. The

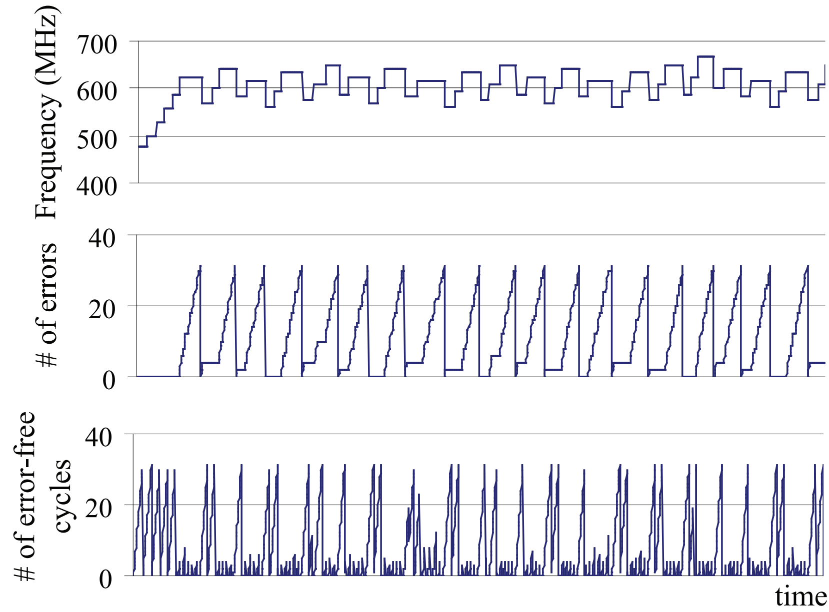


Figure 3.11: *Dynamic execution of a* `sobel` *BLADES processor.*

frequency was restored after the offending instruction completed. Figure 3.10 shows the energy savings obtained using Razor and the slowdown from re-executing operations that cause an error. Energy savings continue to increase until approximately 70% of the nominal operating voltage. The similarity of this graph to data presented in prior Razor work [23, 7] that used general purpose processors verifies that it is just as effective when applied to LAs.

3.3.3 Dynamic Frequency Scaling

A 500 MHz BLADES `sobel` processor (Figure 3.12) was fabricated alongside an ARM1176 in a 65nm process. The LA communicates with the ARM1176 via an ARM Peripheral Bus (APB) interface. The total die area of the SoC is $4mm^2$, of which the pictured LA uses $0.21mm^2$, or approximately 5% of the total area.

To simulate behavior under reduced voltages, the LA was run with dynamically changing frequencies, shown in Figure 3.11. When several cycles of error-free operation are

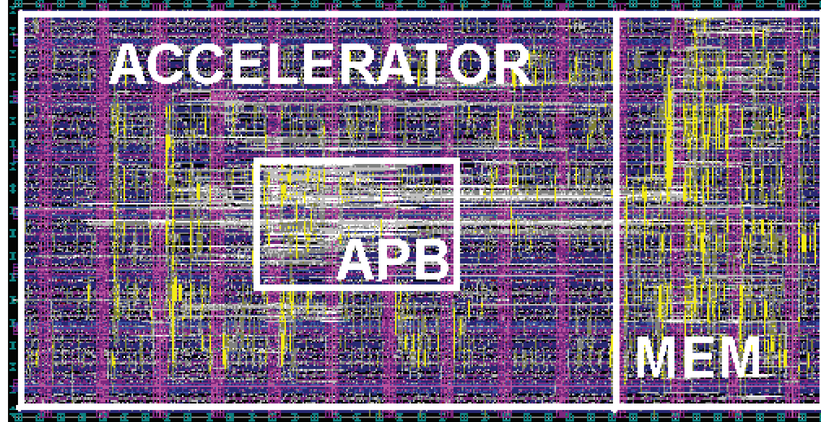


Figure 3.12: `sobel` *BLADES processor with 256 bytes of memory at 65nm. The accelerator communicates with an ARM1176 via an APB interface.*

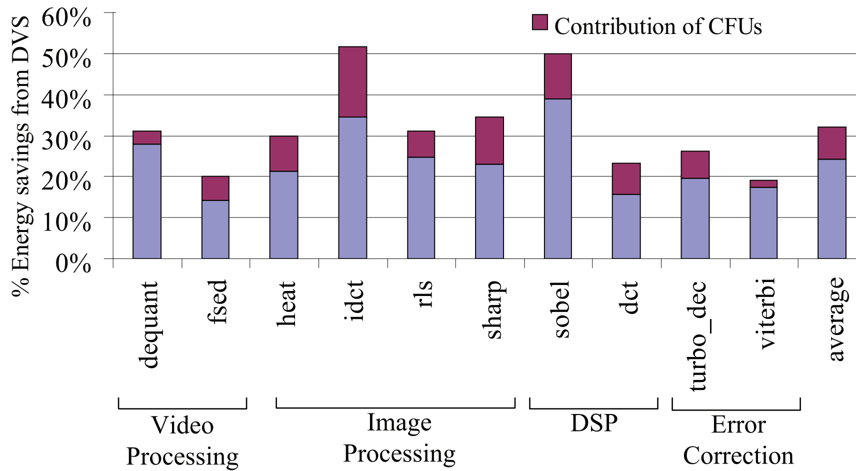


Figure 3.13: *Dynamic energy savings when using DVS. The top portion is the contribution to this number by using CFUs.*

observed, the clock period is gradually decreased. When a timing error does occur, the supplied clock frequency remains unchanged during the error recovery process but every 2 clock cycles are gated, effectively halving the frequency in the LA. The frequency is only reduced after several consecutive errors are observed.

3.3.4 Effect of CFUs

Figure 3.13 shows the savings in energy from using DVS across four application domains. The contributions of the CFU technique presented in this paper make up the top portion of each bar. On average, there is a 32% savings in energy, of which 8% was due to the CFU technique.

The benchmarks `idct`, `sharp` and `dct` had not only multiple CFUs, but also a reduction in the total number of FUs in the design since some FU types were entirely subsumed into CFUs. In `idct`, for example, none of the multiply units required extra hold-fixing since they were all within CFUs. The same was true in `sharp`, but for subtract units. The CFUs contributed to approximately 33% of the energy savings observed in these benchmarks.

However, the benchmarks `viterbi` and `dequant` did not do as well; the CFUs contributed to only 9% and 10% to the overall energy savings, respectively. The primary reason for this is that both of these benchmarks have a significant amount of control flow within the inner-loop. These results can be improved by adding better predicate support to the CFUs. Currently, all the operations within a CFU are guarded by the same predicate; allowing different operations within a CFU to execute independently would add some complexity to the replacement algorithm and the hardware generated but would result in more energy savings.

3.4 Related Work

Frequency scaling applied to ASICs has been researched by different groups. Dhar et al. [29] proposed an adaptive voltage scaling scheme that utilized several blocks of logic around the main ASIC to dynamically change the voltage as is needed for the desired system speed. Their work, however, is purely external to the processor's design and is not interacting with it in any way unlike the scheme proposed in this paper. In [105], the authors propose several modifications built in to the main logic of the design for the purposes of scaling the voltage. Their technique, however, utilizes no knowledge of the application itself - what the constraints are on data, when specific values will actually be used, etc., and as such have a limited field of view.

Razor-style flip-flops have been used in the past for different reasons, the most significant of these [23] being for the purpose of reducing the supply voltage of a processor and in turn reducing the overall energy consumption. Razor flip-flops, or more generically, flip-flops which are able to detect changes in the input data near the clock edge, have also been used for reliability reasons in [63].

Opcode chaining was previously used in [90, 46, 83]. However, prior work focuses on chaining multiple simple operations into a single cycle to improve performance by performing a single complex computation instead of multiple simple ones.

3.5 Conclusion

In this paper, we propose the automatic design of adaptive loop accelerators. The designs leverage Razor technology that consists of a delay-error tolerant flip-flop placed on

critical paths to identify timing errors in circuits. Razor enables a design to dynamically adjust its supply voltage and operating frequency to an optimal point based on the environmental conditions and characteristics of the computation. Adaptive loop accelerators consist of strategically deployed Razor flip-flops at the vulnerable locations of the loop accelerator as well as additional flip-flops to enable rollback and re-execution in the event of a timing error. Our loop accelerators utilize a stylized architecture template that allows highly customized designs to be automatically adapted to detect and recover from errors. These accelerators can be used to reduce energy consumption, increase performance, or provide adaptability in the presence of process variation. For our experiments, we show the clock frequency and voltage can easily be scaled in accelerators by adding a small number of Razor flip-flops. Specifically, voltage can be reduced to below 70% of the nominal supply voltage resulting in over 50% energy savings with negligible performance penalty, 32% on average. Further, by fusing primitive operations into custom operations, the hold-fixing requirement of Razor is relaxed and the associated energy overhead is reduced by as much as 33%, 24% on average.

CHAPTER IV

Flexible Loop-Accelerator Systems

4.1 Introduction

The advent of programmable graphics processing units, or GPUs, for general-purpose computing is one of the major steps taken in computing over the last few years. These GPGPUs which, in the past, have been predominantly used for gaming and advanced image and video editing, are now being used by many developers to accelerate inherently parallel programs in several other domains. Indeed, considerable amounts time and engineering effort are often spent in order to modify programs so that they may run effectively on GPUs.

Several different application domains observe remarkable speedups when using GPUs, including the following [68]:

- 4X to 100X speedup on medical applications, such as biomedical image analysis, 3D reconstruction of tissue structures for large sets of microscopic images and accelerating MRI reconstructions.

- 8X to 260X speedup on electronic design automation, such as power grid analysis and statistical static timing analysis.
- 4X to 327X speedup on physics applications, such as weather prediction and astrophysics.
- 11X to 100X speed up financial applications such as instrument pricing using Monte-Carlo methods.

4.1.1 The Advantages of GPUs

GPUs have many appealing hardware features as mentioned in Chapter II. Further, GDDR RAM and its increasingly fast successors have allowed for GPUs to have access to an immense amount of memory bandwidth. The AMD Radeon HD 4870 - the first GPU to support GDDR5 memory - has a memory bandwidth of up to 115 GB/s.

Above all, GPUs are commodity hardware products commonly available as a part of many desktop and laptop computers. The tools to program them are also easily available; NVIDIA's Compute Unified Device Architecture (CUDA) package, for example, is free to download from their website [64]. CUDA is a general purpose parallel computing architecture which consists of the CUDA instruction set and the compute engine in the GPU. It provides a small set of extensions to the C programming language, which enables straightforward implementation of parallel algorithms on the GPU. CUDA also supports scheduling the computation between CPU and GPU, such that serial portions of applications run on the CPU and parallel portions are mapped to the GPU.

Individual cores in Intel's up-and-coming "Larrabee" processor and its successors implement the ubiquitous x86 ISA [88], allowing users to use a host of already-existing development tools to port their applications to it. Server products like Tesla [67] with even more compute power are also available.

4.1.2 The Disadvantages of GPUs

One of the main bottlenecks in applications running on GPUs is the gap between their computational ability and their memory bandwidth. While the absolute memory bandwidth available to GPUs is quite high, it is not growing at the same rate as their theoretical peak performance. Currently, even one of the latest generation of NVIDIA graphics processors, the GeForce GTX 280 [66] can only transfer up to 142 GB/s of data despite having a peak performance of 933 GFLOPS [20]. This works out to approximately 0.15 bytes of data, on average, per floating-point operation.

Further, the total power consumption of these GPUs, is enormous. The GeForce GTX 280, for example, while having a tremendous amount of compute ability, consumes 236W of power resulting in a very low MIPS-per-mW power-efficiency ratio at peak performance. Inside GPU cores, the main power-consuming components are general storage elements such as register files and scratchpad memories. This is not necessarily a drawback with GPUs, specifically, but rather with most general-purpose programming devices. The interconnect required to access random elements in register files is quite power-hungry, and the more read and write ports a register file has, the higher its power consumption. Consequently, register files that feed vector-SIMD FUs and must output several elements at once consume significant amounts of power.

Though power is not necessarily a significant drawback for users interested in using these devices for video game graphics acceleration, it is not a desirable choice when on a limited power source, in an embedded system or in a high-temperature environment. To address these disadvantages, many designers turn to customized and domain-specific hardware.

4.1.3 Programmable Loop Accelerators

The programmable solutions shown in Figure 2.1 are all "universally" programmable, allowing any loop to be mapped on to them, although at varying degrees of efficiency. There is a wide gap between the efficiency that can be achieved by ASICs and the efficiency that can be achieved by these programmable solutions. There are, for example, instances where there is a narrow requirement of flexibility. Using any of these above solutions is overkill for these instances as these solutions sacrifice too much efficiency for the needed flexibility. Further, most of the middle-ground solutions listed above do not offer any support for fast floating-point computation, which limits the number of applications that they can be used for.

This work advocates an open area in the design space where a non-trivial amount of programmability is provided in terms of intra-processor communication, functionality and storage, but the application and domain-specific design, as a whole, resembles an ASIC more than a processor. The design point is labeled Programmable Loop Accelerator, or PLA (not to be confused with programmable logic array). The specifics of the PLA are described in Section 4.3.

4.2 Targeting Medical Applications

Medical imaging devices are generally large, bulky and expensive machines that have very limited portability and consume large amounts of power. There is an increasing focus on reducing the power of these medical imaging devices [78]. In order to address this issue, this work focuses on principle components of Computed Tomography (CT) and Magnetic Resonance Imaging (MRI) image processing and reconstruction. The applications considered here are explored in more detail in Chapter V.

A CT scan involves capturing a composite image from a series of X-Ray images taken from various angles around a subject. It produces a very large amount of data that can be manipulated using a variety of techniques to best arrive at a diagnosis. Oftentimes, this involves separating different layers of the captured image based on their radio-densities. A common way of accomplishing this is by using a well-known image-processing algorithm known as "image segmentation".

In essence, image segmentation allows one to partition a given image into multiple regions based on any of a number of different criteria such as edges, colors, textures, etc. Being able to partition images in this manner allows for studying of isolated sections of the image rather than of all the information that was captured.

The segmentation algorithm used in this work has three main floating-point-intensive components: Graph segmenting (`CT.segment`), Laplacian filtering (`CT.laplace`) and Gaussian convolution
(`CT.gauss`).

Benchmark	#instrs	%FP	Data Req'd $\frac{bytes}{instr}$
MRI.FH	38	42	0.95
MRI.Q	34	35	1.06
CT.segment	26	42	1.38
CT.laplace	20	30	1.20
CT.gauss	22	32	1.09

Table 4.1: *Medical application characteristics.*

Laplacian filtering highlights portions of the image that exhibit a rapid change of intensity and is used in the segmentation algorithm for edge detection. Gaussian convolution is used to smooth textures in an image to allow for better partitioning of the image into different regions.

An MRI scan, instead of using X-Rays, uses a strong magnetic and radio frequency fields to align, and alter the alignment of, hydrogen atoms in the body. The hydrogen atoms then produce a rotating magnetic field that can be detected by the MRI scanner and converted to an image. The main computational component of reconstructing an MRI image is calculating the value of two different vectors, known here as MRI.FH and MRI.Q, respectively (explained in more detail in [58, 92]).

Table 4.1 shows some characteristics of the benchmarks in consideration. All of these benchmarks are floating-point-intensive and require large amounts of data for the computation they perform, especially when compared to the 0.15 bytes/instruction supported by the GTX 280 GPU mentioned earlier. The main loops in these benchmarks are "do-all" loops - there are no inter-iteration dependences.

Prior work in this field has predominantly focused on using commercial products to accelerate medical imaging. For instance, in [45], the authors port "large-scale, biomed-

ical image analysis" applications to multi-core CPUs and GPUs, and compare different implementation strategies with each other. In [80], the authors study image registration and segmentation and accelerate those applications by using CUDA on a GPU. In [92], the authors use both the hardware parallelism and the special function units available on an NVIDIA GPU to dramatically improve the performance of an advanced MRI reconstruction algorithm. There are several other such examples of novel work in this field.

In contrast with such research, this work focuses on designing a new, highly efficient, microarchitecture and architecture with the specific hardware requirements of medical imaging in consideration.

4.3 PUMA

PUMA, ***P**arallel **micro**-architecture for **M**edical **A**pplications*, is a tiled architecture as shown in Figure 4.1. It is specifically designed to maximize power-efficiency when executing medical imaging applications while still retaining full programmability. Each tile in PUMA is an instance of a specialized PLA - a generalized loop accelerator. The PLA tiles are connected to their neighboring tiles and to the external interface through a high-bandwidth mesh of on-chip routers.

4.3.1 Background

Figure 4.2 shows the hardware schema for the single-function loop accelerator [33, 24, 87]. The LA is designed to efficiently execute a modulo scheduled loop [76] in hardware. The length of the schedule, and the corresponding run-time of the loop, are determined

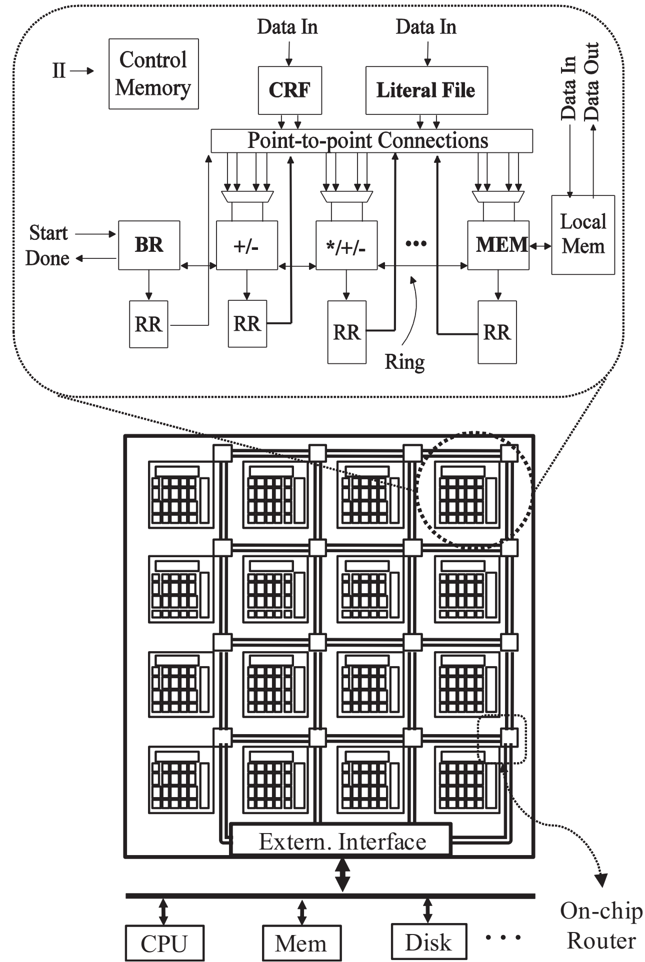


Figure 4.1: *PUMA. Each tile comprises of a programmable loop accelerator (template pictured) and the control and data memories required for its operation. On-chip routers transfer data between each tile and the external interface.*

by the *initiation interval* (II) - the number of cycles between the beginnings of successive iterations of the loop. Thus, a lower II corresponds to a shorter schedule and increased performance. The modulo schedule contains a *kernel* that repeats every II cycles and may include operations from multiple loop iterations.

The LA is designed to exploit the high degree of parallelism available in modulo scheduled loops with a large number of function units (FUs). Each FU performs a specific set of functions that is tailored for the particular loop. Each FU writes to a dedicated shift register file (SRF); in each cycle, the contents of the registers shift downwards to the next register.

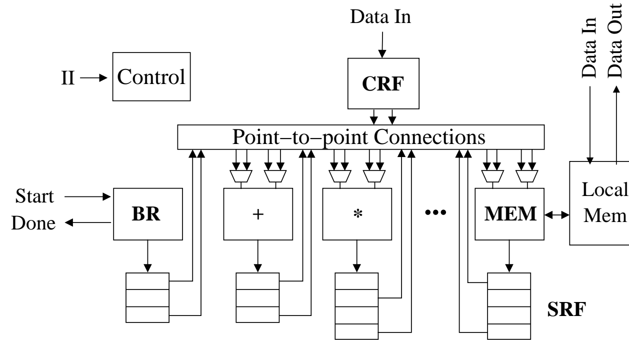


Figure 4.2: *Template for single-function loop accelerator.*

Point-to-point wires from the registers to FU inputs allow data transfer from producers directly to consumers. Multiple registers may be connected to each FU input; a multiplexer (MUX) is used to select the appropriate one. Since the operations executing in a modulo scheduled loop are periodic, the selector for this MUX is essentially a modulo counter. In addition, a central register file (CRF) holds static live-in register values that cannot be stored in the SRFs.

The schema described is a template that is customized for the particular loop being accelerated. The number, types, and widths of the FUs, the widths and depths of the SRFs, and the connections from the SRFs to the FUs are all determined from the loop. During synthesis, the loop is first modulo scheduled to meet a given performance requirement, and then the details of the LA datapath are determined from the communication patterns in the scheduled loop.

The control path for the single-function LA consists of a finite state machine with II states corresponding to each of time slots in the kernel of the modulo schedule. In each state, control signals direct the execution of FUs (for FUs capable of multiple operations) and control the MUXes at the FU inputs.

Finally, a Verilog HDL realization of the accelerator is generated by emitting modules with pre-defined behavioral Verilog descriptions that correspond to the datapath elements. A simulation environment is used to verify that the Verilog properly implements the loop. Finally, gate-level synthesis, placement, routing, power analysis and post-synthesis verification are performed on the design.

4.3.2 PUMA Architecture

4.3.2.1 Baseline PLA Design

A PLA is generalized loop accelerator, created by modifying the template datapath shown in Figure 4.2. A generic datapath template for the PLA is illustrated on the right side of Figure 4.1 The accelerator is designed for a specific loop at a specific throughput, but contains a more general datapath than the single-function LA to allow for different loops to be mapped onto the hardware [32]. These generalizations provide the LA with flexibility in functionality, storage, control and communication.

To provide functionality, simple modifications were made to FUs in order for them to support more operations; adders (both integer and floating-point) are generalized to adder/subtracter units, left-shift units are generalized to left/right rotators, every FU can execute an identity operation to act like a move instruction, etc. Even load-store units can be generalized to integer adder/subtracter units if they already had the functionality to compute indirect addresses. Further, the input-muxes to FUs are redesigned to allow for operand-swapping as well.

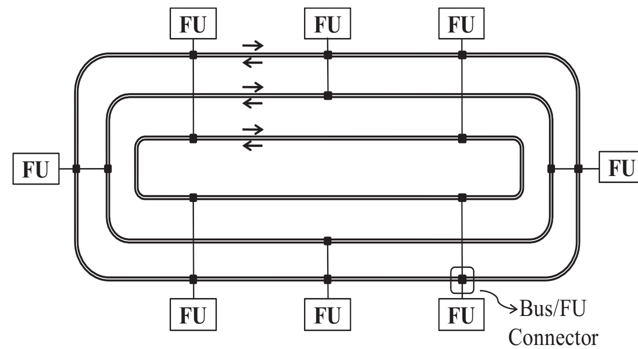


Figure 4.3: *The PUMA ring architecture. Two bus segments, moving data in opposite directions, connect each bus/FU connector. In this 8-FU example, the longest move latency between any two FUs is 2 cycles.*

The SRFs used in the LA have limited addressability and fixed life-times for variables. To overcome these constraints and provide more generality, these SRFs are replaced with rotating-register files (RRs).

To improve controllability, the LA's finite state machine is replaced with a control memory, the contents of which can be changed based on the loop that needs to be executed. Further, numerical constants which were hard-coded in the LA are instead stored in a literal register file.

To generalize communication, the PLA has a bus (in addition to the point-to-point connections) that connects all the RRs to all the FUs. To reduce the hardware cost of having a potentially long bus, its width is limited to one operand, but has a predictable latency of one cycle.

4.3.2.2 PUMA PLA

The PLA bus is not always a viable solution. One main disadvantage with the bus is that it is not a scalable solution for larger PLAs with many FUs. Further, the bus only carries a

Maximize:

$$\sum_{i \in T_\alpha} \sum_{j \in T_\beta} C_{ij} \qquad \forall\alpha\forall\beta : \alpha \neq \beta$$

Subject to:

$$\begin{array}{lcr}
(1) & \sum_{j=0}^{\#FUs} X_{ij} = 5 & i \in [0, \#FUs) \\
(2) & X_{ii} = 1 & i \in [0, \#FUs) \\
(3) & C_{ii} = 1 & i \in [0, \#FUs) \\
(4) & X_{ij} = X_{ji} & i, j \in [0, \#FUs) \\
(5) & C_{ij} = C_{ji} & i, j \in [0, \#FUs) \\
(6) & C_{ij} \leq X_{ij} + I_{ij} & i, j \in [0, \#FUs)
\end{array}$$

Figure 4.4: *ILP formulation for FU arrangement on the PUMA ring.*

single operand per cycle, limiting the amount of programmability available in the PLA and the sequences of opcodes that can be executed in parallel.

To overcome these limitations, the intra-PLA communication fabric in PUMA is changed to a ring. A ring allows for as many operands to be transferred as there are connections to FUs. It does have its limitations, however. The assumed single-cycle latency to transfer data between two arbitrary points in the PLA using the bus is no longer valid, as it takes one cycle to transfer an operand from one ring connection (or ring stop) to another. Also, the longer the distance an operand needs to travel on the ring, the more FUs that have to execute move instructions to propagate the operand along at each ring stop. These added instructions can potentially increase the loop's schedule length and reduce the accelerator's performance.

In PUMA, the ring architecture actually consists of six rings (three sets of two rings going in opposite directions) as shown in Figure 4.3. The first set of rings has a Bus/FU connector (or ring-stop) at every single FU. The second set of rings has a ring-stop at all

the odd-numbered FUs, and the third set of rings has a ring-stop at all the even-numbered FUs. This effectively connects an FU/RF pair to its two neighbors and also to its neighbors' neighbors; i.e. every FU can communicate with itself or with other FUs one or two positions on either side of it on the ring. With this configuration, the number of cycles required to transmit data between any two arbitrary FUs is no more than $\lceil \frac{\#FUs}{4} \rceil$, and regardless of the ordering of FUs on the ring, every possible producer-consumer pairing can be executed, provided sufficient time.

In the 8-FU example shown in Figure 4.3, the maximum latency to transfer an operand from one FU to another is 2 cycles. It can transfer up to 32 32-bit operands in any given cycle, providing an effective inter-FU bandwidth of 57.6 GB/s at a clock frequency of 450 MHz. In contrast, using a bus would have provided an inter-FU bandwidth of 1.8 GB/s.

In order to best maintain generality, we chose to arrange the FUs along the ring to allow maximum connectivity and to distribute the various types of FUs as evenly as possible. This was done by formulating the problem as an integer linear program (ILP) as shown in Figure 4.4.

In the objective function, T_α and T_β are two different sets of FUs, each set having all and only the FUs of a particular type. The subscripts i and j are FU IDs and C_{ij} is a binary variable that is 1 if a connection exists between FU i and FU j. Essentially, this objective function aims to maximize the number of connections between different types of FUs, subject to the following constraints:

- In constraint set (1), X_{ij} is a binary variable that is 1 if FU i is "positioned" adjacent to FU j, implying that they are connected by the ring. Every FU, therefore, is "po-

sitioned" next to 5 other FUs: itself, its two neighbors and the two additional FUs neighboring its neighbors.

- Constraint sets (2) and (3) specify that every FU is "positioned" next to and connected to itself.

- Constraint sets (4) and (5) specify that all added connections are bidirectional.

- In constraint set (6), I_{ij} is a binary number that is 1 if a connection between FU i and FU j has already been inserted by the synthesis tool. This constraint enforces the rule that a connection between FU i and FU j can only exist if they are either "positioned" next to each other or are already connected.

- A 7th set of constraints was initially used which specified that there must always be a path between any two FUs with exactly $\lceil \frac{\#FUs}{4} \rceil$ connections between them This constraint was used to prevent insular sets of 5 FUs or sets of 5 FUs connected linearly rather than in a ring (i.e. without a direct connection between the two ends). While this problem might occur in theory, the pre-existing connections put in place by the synthesis system prevent it from happening in practice and these constraints were removed to reduce the size of the ILP.

Once the optimal solution is obtained, the values of the X_{ij} variables provide a unique ring arrangement.

4.3.3 PUMA System Architecture

Tiled architectures have been used in several other projects, such as Raw [94], TRIPS [81], MorphoSys [56], Merrimac [22], etc. Such an architecture was chosen for PUMA (as shown in Figure 4.1) for a few different reasons. The replication of identical tiles means that the application need not be restricted to run on only a few specific portions of the processor, making compilation for PUMA easier. This is especially useful if the processor is used to execute a stream-like application. For example, in the CT scan benchmarks used here, image segmentation can be executed on one part of the image on some tiles, transfer the resultant data to other tiles for filtering, and perform segmentation on a different part of the image. Further, replication of a single PLA design simplifies the top-level system design and makes testing and verification easier.

The programming and execution model of PUMA closely follows that of modern-day general-purpose GPU processors. Like GPGPUs, PUMA is intended to be mainly used to accelerate compute-intensive, highly-parallel loops, but is able to execute all the other sections of the program as well, albeit at reduced performance.

PUMA is currently envisioned to be in one of two forms: either a discrete core on a PCI-Express (or similar) card external to the main processor core (as pictured in Figure 4.1), or on the same die as the main processor, connected either through memory or via an ultra-high bandwidth, on-chip bus. For the purposes of this work, we have assumed the former model, for more fair, direct comparisons to the current state of the art GPGPUs.

Benchmark	**Peak Perf.** $\frac{GFLOps}{sec}$	**Peak Perf.** $\frac{GIOps}{sec}$	**B/W** $\frac{GB}{sec}$	**#Tiles**
`MRI.FH`	7.2	5.4	16.2	9
`MRI.Q`	5.4	5.4	16.2	9
`CT.segment`	4.95	2.25	16.2	9
`CT.laplace`	2.7	3.15	10.8	14
`CT.gauss`	3.15	3.6	10.8	14

Table 4.2: *Characteristics of the individual accelerators for each benchmark.*

4.4 Experiments and Results

4.4.1 Setup

All the PLAs in this work were synthesized for (and run at) a frequency of 450 MHz. The logic synthesis was done using Synopsys Design Compiler 2006-06 and Synopsys Physical Compiler 2006-06, targeting a 65nm process technology with a nominal supply voltage of 0.9 Volts. Energy numbers were obtained using Synopsys PrimeTime-PX 2006-12. For the purposes of this study, we assume that a peak memory bandwidth of 142 GB/s is available to each PUMA system. This is the same amount of bandwidth afforded to the NVIDIA GTX 280 processor.

The synthesis target assumed worst-case performance conditions (slow silicon, reduced voltage and increased temperature) and energy and power measurements assumed typical power conditions (normal silicon, nominal voltage, room temperature).

4.4.2 PLA Characteristics

PUMA systems were built using PLAs for each of the five benchmarks in considerations (five systems, each composed entirely of multiple tiles of a single type of PLA). Table 4.2

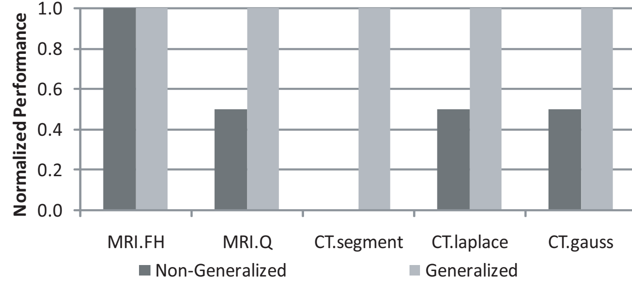


Figure 4.5: *Normalized performance of benchmarks on LA and PUMA PLA designed for* `MRI.FH`.

shows various characteristics of these accelerators. The "Peak Perf." columns show the throughput when executing floating-point operations and integer operations, respectively, in billions of operations per second. The next column shows the minimum bandwidth required by each application to prevent starvation. Finally, the last column shows the total number of tiles of each PLA that would be present in a PUMA system. The number of tiles was chosen to prevent data starvation, to make the most efficient use of the resources available. For example, the number of tiles in a system with `MRI.FH` tiles is $\lceil \frac{142}{16.2} \rceil$, or 9.

Figure 4.5 shows the normalized performance difference between the non-generalized and generalized loop accelerators across various benchmarks, to illustrate the effects of the modifications made to the baseline accelerator to increase programmability. Each of the different benchmarks were compiled for the `MRI.FH` accelerator.

The left column for each benchmark shows its normalized performance. The benchmarks `MRI.Q`, `CT.laplace` and `CT.gauss` suffered a 50% reduction in performance; i.e. their II values had to be doubled, from 1 to 2, in order for them to execute on the baseline loop-accelerator. The benchmark `CT.segment` could not be compiled for the `MRI.FH` accelerator at all.

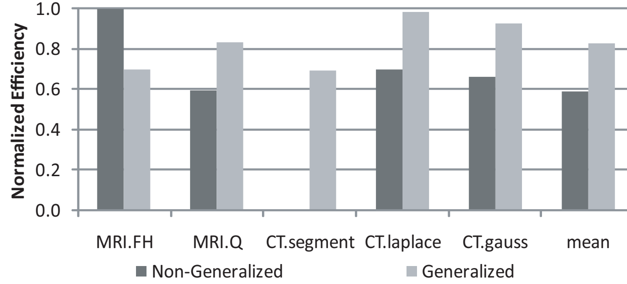


Figure 4.6: *Normalized* $\frac{Performance}{Power}$ *efficiency of benchmarks relative to* `MRI.FH`.

For each benchmark, the column on the right shows the achieved performance on the generalized accelerator, with the hardware modifications specified in section 4.3.2.1. As shown, these modifications allowed all the benchmarks to run at full performance, at minimum II.

Figure 4.6 shows a graph similar to that in Figure 4.5, but shows the normalized efficiency in terms of the accelerator's performance-to-power ratio. Due to the increase in overall performance provided by the generalizations, the benchmarks `MRI.Q`, `CT.laplace` and `CT.gauss` had a significant increase in efficiency despite the power overhead of the additions. The `MRI.FH` benchmark, however, which would not experience any improved performance from the generalizations loses efficiency due to the increase in the accelerator's power consumption. On average, the generalizations increased the accelerator's efficiency increased by approximately 40%.

4.4.3 System Characteristics

We evaluated five different PUMA system designs, one for each PLA design. Each system had a different number of tiles, based on the bandwidth requirement of each benchmark as specified in Table 4.2.

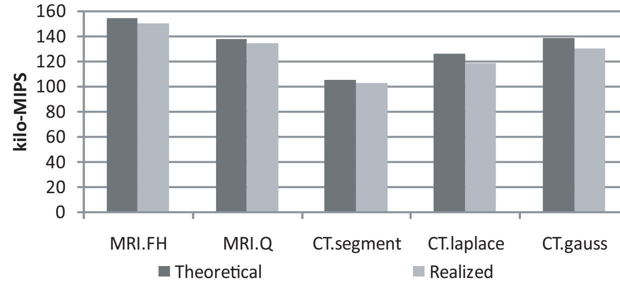


Figure 4.7: *Overall system performance for each of the PUMA systems.*

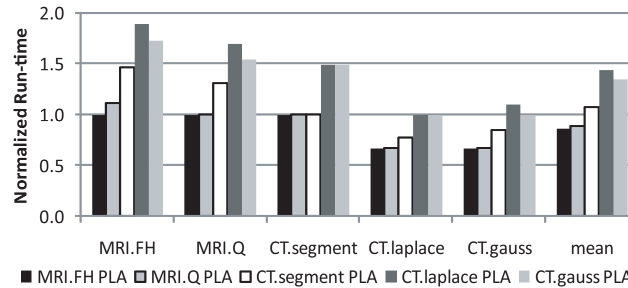


Figure 4.8: *Run-time of benchmarks running on different PLAs, normalized to that of the native benchmark.*

Figure 4.7 shows the total performance offered by the PUMA systems designed around each of the different PLAs, measured in thousands of MIPS. For each benchmark, the column on the left shows the peak raw performance available to all applications. The column on the right shows what the bandwidth-limited performance is while each system is running the benchmark that it was designed for. These theoretical and realized performances are quite close, differing on average by less than 4%.

Figure 4.8 shows a set of columns for each benchmark, where each column indicates the normalized run-time of the benchmark on different PLAs. These values are normalized to the run-time of the benchmark on a PLA designed for it. All of the benchmarks could be scheduled with an II of 1. Therefore, there are often considerable reductions in run-time when the smaller benchmarks are executed on PLAs designed for the larger benchmarks. The most dramatic example is the difference in the run-times of the `CT.gauss` benchmark

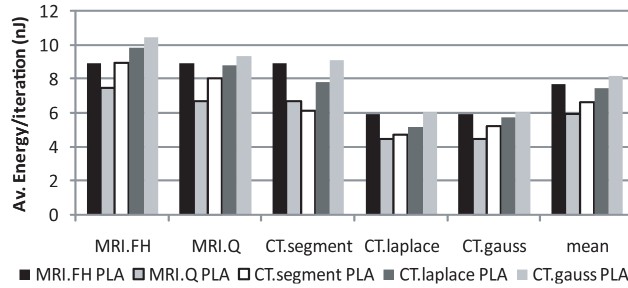


Figure 4.9: *Average energy consumed (per iteration) by each benchmark while running on PUMA systems designed around different PLAs.*

on the `CT.laplace` and `MRI.FH` systems. The opposite, of course, also holds: the larger benchmarks suffer a significant increase in run-time when executing on PLAs designed for smaller benchmarks. Of note is the difference in the run-times of the `MRI.FH` benchmark on the `CT.laplace` and `MRI.FH` systems.

Figure 4.9 shows a similar graph to that in Figure 4.8, but showing the average energy consumed per iteration by each benchmark while running on PLAs designed for other benchmarks. The energy consumption was primarily determined by the size of each benchmark, with the two MRI benchmarks consuming the most regardless of which PLA they ran on.

The most important thing to note on this graph is that the most energy-efficient system is the one designed for `MRI.Q`. The main reason for this is that of the five benchmarks in consideration, it is the one that is closest to being the "average benchmark". This is clear from the data presented in Table 4.2. Its data:compute ratio is quite close to the average among the benchmarks providing a good balance between the more compute-intensive benchmarks and the more data-intensive benchmarks. Its integer and floating-point throughput are identical, providing a balance between the more floating-point-intensive benchmarks and the more integer-intensive benchmarks.

GPU	#cores	Freq. (MHz)	TDP (W)	Peak Perf. $\frac{GFLOPs}{sec}$	B/W $\frac{GB}{sec}$
GTS 250	128	1,836	150	705	70.4
GTX 260	192	1,242	182	715	111.9
GTX 280	240	1,296	236	933	141.7
GTX 285	240	1,476	183	1,063	159.0
GTX 295	480	1,242	289	1,788	223.8

Table 4.3: *GPU hardware characteristics.*

4.4.4 Commodity GPGPU Comparison

While other architectures may certainly be used for this domain, GPGPUs are the solutions that are currently in use in many medical imaging applications and, therefore, the most suitable comparison point. For this reason, we assessed the performance and efficiency of five NVIDIA GPUs shown in Table 4.3.

Each of these GPUs are different implementations of the GTX 200 architecture [65]. In this context, "#cores", or "streaming processors" (SPs), are individual data paths with 3 floating-point FPUs each. Eight of these SPs, sharing an instruction unit and a local storage structure make up a "streaming multiprocessor" (SM). Three SMs make up a "thread/texture processing cluster" (TPC). Each GPU has two clock frequencies associated with it - a faster one for the processor and a slower one for the graphics-specific hardware. The clock frequency mentioned here is the former. The power numbers in the table are the peak TDP (thermal design power) of the core - the absolute worst-case power consumption. Under normal use, and in the experiments in this paper, the power is significantly less than the number listed here.

Figure 4.10 shows the result of direct performance comparisons between an `MRI.FH` PUMA system and the GPUs in consideration. The column on the left shows the total

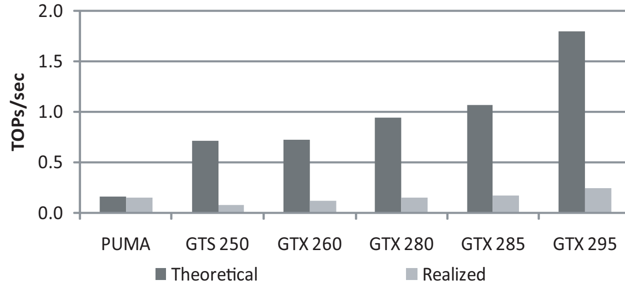


Figure 4.10: *Achieved performance of the MRI.FH benchmark (in trillions of operations) on the* `MRI.FH` *PUMA system and on various NVIDIA GPUs based on the GT200 architecture.*

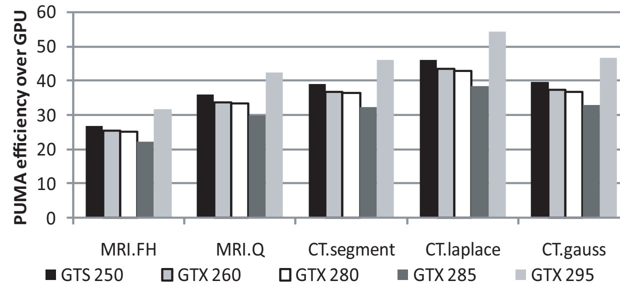


Figure 4.11: $\frac{Performance}{Power}$ *efficiency improvement (in multiples,* not *in percentages) of running benchmarks on PUMA systems rather than on commodity GPUs.*

compute capability of each of the processors. The column on the right shows the realized performance while executing the `MRI.FH` benchmark, accounting for bandwidth restrictions. PUMA achieves a very small fraction of the peak performance offered by the GPUs, between 8.6% of the dual-GPU GTX 295 and 21.8% of the GTS 250.

This gap changes dramatically, however, when accounting for the bandwidth-intensive nature of the application in question. PUMA delivers between 63% (on the dual-GPU GTX 295) and 2X the performance (on the GTS 250) of the GPUs.

The case for PUMA is further underscored by examining the GPUs' power efficiency, as shown in Figure 4.11. This graph shows how many *times* more efficient, in terms of number of operations per Watt, PUMA systems are relative to the GPUs in consideration. These

values range from 20X, for the most complex benchmark running on the most efficient GPU, to 54X, for the least complex benchmark running on the least efficient GPU.

4.5 Conclusion

The PUMA architecture is a power-efficient accelerator system designed specifically for efficient medical image reconstruction. It consists of tiles of programmable loop accelerators - ASICs with added hardware to support general-purpose computing - designed around the computation requirements of the image reconstruction domain. As applications in this domain are normally executed on very high-performance GPGPUs, the latest NVIDIA GPU architecture was used to gauge the performance and efficiency of PUMA. The results are very encouraging - PUMA achieves up to 2 times the performance of a modern GPU architecture and has up to 54 times the power efficiency.

CHAPTER V

Accelerator Systems for High-Throughput, Bandwidth-Constrained Applications

5.1 Introduction

Medical imaging is one of the most effective tools used in modern medicine to aid physicians in diagnosing and analyzing ailments. Computed tomography, or CT, employs geometry processing to generate a three-dimensional image of the inside of an object from a large series of two-dimensional x-ray images taken around a single axis of rotation. More than 62 million scans are ordered each year to detect ailments ranging from brain tumors and lung disease to guiding the passage of a needle into the body to obtain tissue samples [28]. Compared to traditional 2D x-rays, CT has inherent high-contrast resolution to accurately detect differences in tissue density of less than 1%. Further, the data is highly flexible in that it can be aggregated and viewed in all three planes depending on the diagnostic task. Other popular medical imaging techniques use varying methods to acquire images, e.g., other forms of radiation or radio frequency signals, including single photon

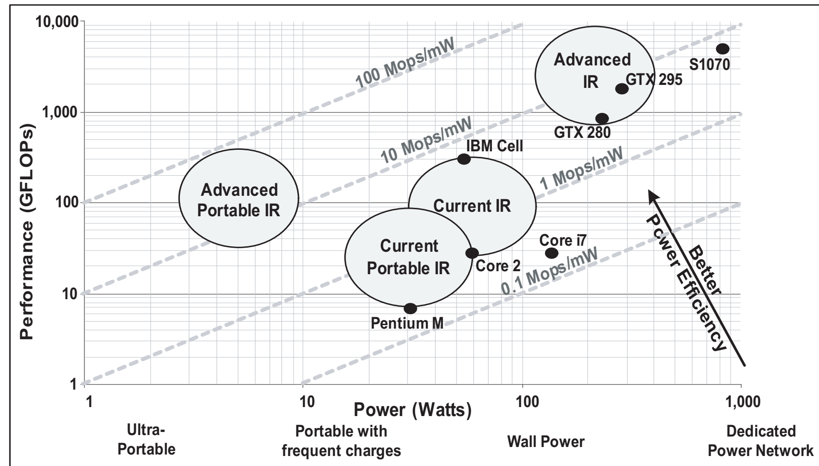


Figure 5.1: *Performance and power requirements for the domains of current and advanced image reconstruction techniques in both tethered and untethered environments. Diagonal lines indicate "Mops/mW" performance/power efficiency. For comparison, the peak performance and power of several commercial processors and GPGPUs are provided: Intel Pentium M, Intel Core 2, Intel i7, IBM Cell, Nvidia GTX 280, Nvidia GTX 295, and Nvidia Tesla S1070.*

emission computed tomography (SPECT), positron emission tomography (PET), and magnetic resonance imaging (MRI).

From a computer architecture perspective, the challenging aspect of medical imaging is the vast amount of computation that it requires. This computation is floating-point-intensive and utilizes a large amount of data. Figure 5.1 presents the performance requirements and power envelopes of large scale imaging systems as well as portable (bed-side or in-field) systems. Here, the term image reconstruction (IR) is used to encompass the key algorithms found in most variations of medical imaging. From the figure, the performance of advanced imaging used in non-portable systems ranges from 900 GFLOPs to nearly 10 TFLOPs. Portable IR requires an order of magnitude less performance, but also has a substantially lower power budget to operate in a less tethered environment. Figure 5.1 also presents the peak performance and power of several commodity processors for comparison including processors from Intel, IBM and Nvidia.

Current medical imaging compute substrates: Conventional CT scanners and MRI systems use a combination of general-purpose x86 processors, ASICs, and FPGAs to perform the necessary computation. The JXT6966 from Trenton systems [103] is a board consisting of multiple Core i7-class processors; Texas Instruments has a number of comprehensive solutions which use a combination of analog components to control the x-ray emitters and detectors, and fixed-point DSPs for image reconstruction [99]; and Nvidia GPGPUs have been used to accelerate MRI reconstruction [92]. These solutions all have their drawbacks. The TI solutions do not support floating-point computation. The x86-based solutions require turnaround times of many hours for the advanced IR algorithms that researchers propose [101]. As a result, many developers have turned to general-purpose graphics processing units, or GPGPUs, which are capable of delivering the requisite performance. GPGPUs, however, have a large disparity between their compute capabilities and memory bandwidths. For instance, the latest generation of Nvidia GPGPUs, the GTX285, has a peak performance of 1,063 GFLOPs but can transfer only up to 159 GB/s of data. This works out to an average 0.15 bytes of data per FP operation. For graphics applications, such a ratio is sustainable, but most IR algorithms require an order of magnitude more memory bandwidth due to the data-intensive nature of the computation. The resultant bandwidth-limited performance is considerably less than the reported peak performance. Power consumption, too, is an issue for many of the existing solutions.

Low-radiation and portable medical imaging: Portable medical imaging is an area of growing interest in the medical community for a variety of reasons. Recent medical studies [108, 42, 78] have shown a marked improvement in patient health when using portable CT scanners and MRI machines, especially in emergency and critical cases. The National

Institute for Neurological Disorders and Stroke recommend that patients displaying signs of a severe stroke undergo a CT scan to examine if they would be eligible for thrombolytic therapy – injection of a blood-clot dissolving medication. However, there is a short window of 3 hours for the treatment to be applied. In [108], the effective usage of portable CT scanners allowed patients to be examined more quickly after their arrival at a hospital and resulted in an 86% increase in the number of patients who were eligible for thrombolytic therapy.

In [42], the authors examined the number of medical and technical complications that occurred in medium and high-risk neurosurgery intensive-care unit patients while they were being transporting to CT scanning rooms. Using mobile CT scanners and bringing the scanners to the patients, rather than vice-versa, cut down not only the time required (between 40% and 55%) and the number of hospital personnel required to perform the scan but, most importantly, reduced the number of complications between 83% and 100%.

Conventional CT scanners require a very large amount of power to operate, the majority of which is for the x-ray emitters themselves which consume several kilowatts of power. However, due to the ever increasing number of CT scans performed on patients, there is growing concern about the effects of elevated radiation exposure [28] and, consequently, increased interest in reducing the intensity and power of the x-rays [69]. One approach is using carbon nanotube-based x-ray emitters which use just a few milliwatts of power [89, 116]. Using low-power and low doses of x-rays, however, requires more compute-intensive, iterative techniques to compensate for the associated artifacts [44, 101]. Therefore, reducing the supply voltage and clock frequency of high-performance processors is not an appropriate solution to reduce the power of the reconstruction engine as these

techniques reduce performance as well. Essentially, efforts made to reduce the power consumption of x-ray emission and detection are increasing the power requirements of the reconstruction engines, to the point that the computational devices used for image reconstruction have become the dominant power-consuming components.

A new domain-specific design: To overcome the problems of high power consumption and insufficient memory bandwidth, this work presents an architecture and system design that is targeted for portable medical imaging. Our design, named MEDICS (Medical Imaging Compute System), utilizes three critical technologies to achieve its objectives: (1) a 2D datapath comprised of a chained, wide-SIMD floating-point execution unit to efficiently support the commonly occuring computation subgraphs while minimizing accesses to the register-file; (2) image compression units to compress/decompress data as it is brought on-chip to maximize the available off-chip memory bandwidth; and (3) a memory system consisting of a 3D stacked DRAM and input/output streaming buffer to sustain high utilization of a wide-SIMD datapath. Power-efficiency is also garnered by using compile-time software pipelining to hide memory and datapath latencies, thereby eliminating the need for sustaining a large number of contexts found in some high-performance processors like GPGPUs. Overall, MEDICS achieves 128 GFLOPS while consuming as little as 1.6W on state-of-the-art CT reconstruction algorithms.

5.2 Computational Requirements of Image Reconstruction

The fundamental problem in tomographic image reconstruction is illustrated in Figure 5.2. Figure 5.2(a) sketches the general concept of how a CT scan occurs. The process

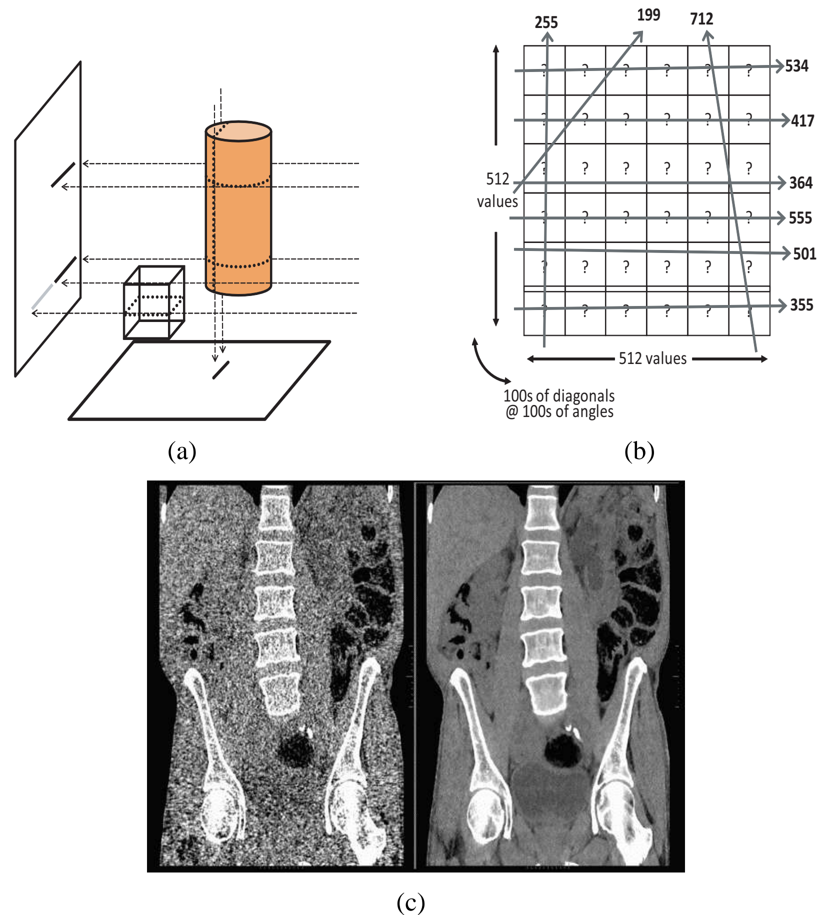


Figure 5.2: *Capturing and deciphering x-rays for tomographic image reconstruction. (a) Detected x-ray attenuations vary based on the density of the object; the opaque cylinder allows much fewer x-rays to pass through it than the transparent cube. (b) Internal density values must be computed using x-ray attenuations measured by the detector array. (c) A slice of 3D helical x-ray CT scan reconstructed by the conventional FBP method (left) and by a MBIR method (right). For these thin-slice images, the MBIR method exhibits much lower noise than the FBP images, enabling better diagnoses.*

begins with x-rays (dotted arrows) being shot from multiple directions at the object. The detector on the opposite side of the x-ray emitter can only measure x-ray attenuation, so it detects very few of the x-rays passing through the opaque cylinder (represented by the dark lines), but detects more of the x-rays passing through the transparent cube (represented by the light line).

Successive scans (or slices) all around the subject at various angles leads to the problem presented in Figure 5.2(b). Here, each arrow represents a path through the matrix and the number next to it is the sum of all the numbers in the elements that the arrow passes through. Using this information, the values in the matrix are populated. Similarly, in tomography, the matrix is the cross section of interest in the human body, each arrow represents one x-ray and the number next to the arrow represents the attenuation measured by the corresponding cell in the detector array. Using the x-ray positions, angles and attenuations, the reconstruction algorithms have to compute the densities and sizes of all the elements in the matrix in order to provide a two-dimensional image of the area of interest.

The total data per scan that reconstruction algorithms have to process is quite large. For instance, the raw data collected by a recent generation 3D multi-slice helical CT system has the following dimensions:

N_{s}	888	samples per detector row
N_{a}	984	projection views per rotation
N_{t}	64	detector rows
N_{r}	10	rotations (turns of helix)

A raw data set of the above dimensions is called a sinogram, and iterative reconstruction methods need three sinograms: the log transmission data, the statistical weighting associated with each measurement, and a working array where predicted measurements are synthesized based on the current estimate of the object at a given iteration. Each of these three sinograms are stored as single-precision, 4-byte floating-point (FP) values, so

the typical minimum memory needed for the data is

$$3 \cdot 4 \cdot N_\mathrm{s} \cdot N_\mathrm{t} \cdot N_\mathrm{a} \cdot N_\mathrm{r} \approx 6.4 \text{ GB}.$$

From that data, one reconstructs a 3D image volume that is a stack of $N_\mathrm{z} \approx 700$ slices, where each slice contains $N_\mathrm{x} \times N_\mathrm{y}$ pixels, where typically $N_\mathrm{x} = N_\mathrm{y} \approx 512$. These image values are also stored as single precision FP numbers, so the memory required for a single 3D image volume is

$$4 \cdot N_\mathrm{x} \cdot N_\mathrm{y} \cdot N_\mathrm{z} \approx 700 \text{ MB}.$$

Advanced iterative algorithms for image reconstruction require 2 to 5 arrays of that size, so several GB of RAM are needed for these 3D image volumes. Further, since the imaging data is iterated over several times, the time taken to access the data must be kept small.

Each iteration of an iterative image reconstruction algorithm requires many FP operations. The dominant computations are called the “forward projection” and the “back projection” operations; one of each is needed each iteration, and they require about the same amount of computation. The amount of computation (measured in number of FP operations) required for a forward projection operation is approximately

$$4 \cdot N_\mathrm{x} \cdot N_\mathrm{y} \cdot N_\mathrm{z} \cdot N_\mathrm{a} \cdot N_\mathrm{r} \approx 7.2 \text{ trillion FP operations}.$$

An iterative algorithm needs several iterations; the number of iterations depends on the algorithm. The algorithms that are most easily parallelized might need about 100 iterations, each of which needs a foward projection and a back-projection operation. If algorithm

Benchmark	#instrs	Data req'd	Registers req'd		#FPU chains		
		$B/instr$	Int.	FP	2-op	3-op	4-op
MBIR	17	0.94	12	14	1	0	1
Radon	70	0.80	12	18	1	3	2
Hough	21	0.95	12	8	1	0	0
Segment	86	1.26	16	11	0	6	6
Laplace	23	1.04	13	10	1	1	1
Gauss	25	0.80	14	9	0	1	0
MRI.FH	41	0.88	14	16	1	2	1
MRI.Q	37	0.97	14	14	3	0	1

Table 5.1: *Medical imaging application characteristics.*

advances could reduce the number of iterations to only 10 iterations, then the total operation count would be about $2 \cdot 10 \cdot 7.2 = 144$ trillion FP operations

Portable devices would require fewer helical rotations (N_{r}) as they would have a narrower region-of-interest. The total number of operations, therefore, would be between 14 and 30 trillion FP operations for 1 or 2 rotations.

5.2.1 Benchmark Overview

For the purposes of this work, a representative subset of different algorithms used in the reconstruction process were analyzed. Though MRI is a very commonly used imaging technique, generating an MRI image is not considered tomographic image reconstruction as it is not a product of multiple cross-sectional images. This work, however, still presents results using MRI-related benchmarks as they are computationally very similar to the tomographic benchmarks.

MBIR: Model-based iterative reconstruction[101] (MBIR) algorithms work by iteratively minimizing a cost function that captures the physics of an x-ray CT imaging system, the

statistics of x-ray CT data, and a priori knowledge about the object (patient) being scanned. By incorporating accurate models, MBIR methods are less sensitive to noise in the data, and can therefore provide equivalent image quality as present-day "filtered back-projection" (FBP) algorithms with much lower x-ray doses. Alternatively, they can provide improved image quality at comparable x-ray doses. Figure 5.2(c) [101] shows an example of a coronal reformatted slice of a 3D helical x-ray CT scan reconstructed by the conventional FBP method and by an MBIR method. For these thin-slice images, the MBIR method exhibits much lower noise than the FBP images, enabling better diagnoses. The benchmark used here is the most compute-intensive inner-loop in the algorithm.

The Radon Transform: The Radon transform of a continuous two-dimensional function g(x,y) is found by stacking or integrating values of g along slanted lines. Its primary function in computer image processing is the identification of curves with specific shapes.

The Hough Transform: The Hough transform, like the Radon transform, is used to identify specific curves and shapes. It does this by finding object imperfections within a certain type of shape through a voting procedure which is carried out in parameter space. The Hough transform was historically concerned with identifying lines in an image, but has later been used to identify the locations of circles and ellipses.

CT Segmentation: A CT scan involves capturing a composite image from a series of x-ray images taken from various angles around a subject. It produces a very large amount of data that can be manipulated using a variety of techniques to best arrive at a diagnosis. Oftentimes, this involves separating different layers of the captured image based on their radio-densities. A common way of accomplishing this is by using a well-known image-processing algorithm known as "image segmentation". In essence, image segmentation

allows one to partition a given image into multiple regions based on any of a number of different criteria such as edges, colors, textures, etc. The segmentation algorithm used in this work has three main FP-intensive components, Graph segmenting (`Segment`), Laplacian filtering (`Laplace`), and Gaussian convolution (`Gauss`).

Laplacian Filtering: Laplacian filtering highlights portions of the image that exhibit a rapid change of intensity and is used in the segmentation algorithm for edge detection.

Gaussian Convolution: Gaussian convolution is used to smooth textures in an image to allow for better partitioning of the image into different regions.

MRI Cartesian Scan Vectors: One of the main computational components of reconstructing an MRI image is calculating the value of two different vectors, which are referred to as `MRI.FH` and `MRI.Q` [58, 92] in this paper. These vectors are used to reconstruct an image using non-Cartesian sampling trajectories – a computationally less efficient but faster and less noisy than reconstructing using a Cartesian scan trajectory.

5.2.2 Benchmark Analysis

Table 5.1 shows some of the key characteristics of the benchmarks under consideration. The columns are defined as follows: “#instrs” specifies the number of assembly instructions in each of the benchmarks, “Data required” specifies the memory requirements in terms of average number of bytes required per instruction, “Registers required” specifies the number of entries required in integer and FP register files (RFs) so that the benchmark need not spill temporaries to memory, and “#FPU chains” specifies the number of FP computation chains that are 2, 3, and 4 operations deep. From the table, all of these benchmarks are FP-intensive and require a large amount of data for the computation they

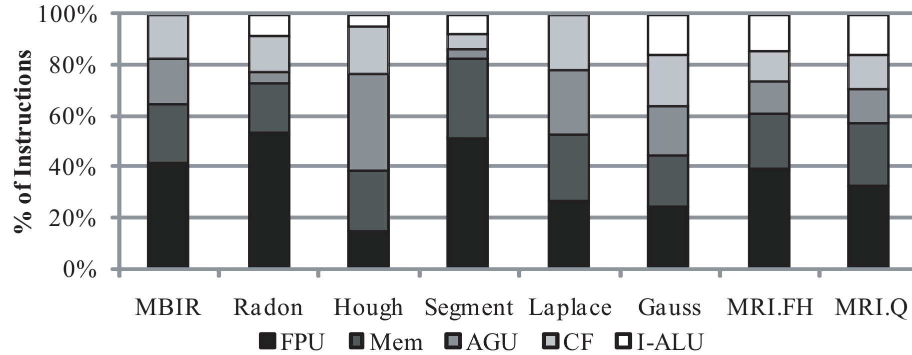


Figure 5.3: *Instruction type breakdown showing the % of instructions used for FP computation (FPU), loads and stores (Mem), address generation (AGU), control-flow (CF) and integer ALU (I-ALU).*

perform with memory-to-computation ratios ranging from 0.80 to 1.26, well in excess of the 0.15 bytes/instruction supported by the GTX 285 GPGPU mentioned earlier. The loops in these benchmarks are "do-all" loops – there are no inter-iteration dependences. However, each iteration is typically sequential as indicated by the relatively small number of registers that are required. FP computation tends to be organized as moderately deep chains of sequentially dependent computation instructions.

Figure 5.3 characterizes the type and frequency of instructions in each benchmark, showing the percentage of FP arithmetic, memory, address-generation, control-flow, and integer arithmetic instructions, respectively. As can be seen from this graph, the computation in these benchmarks is predominantly FP arithmetic, but there are some integer operations as well. Of the integer registers specified in Table 5.1, most of these registers are used for memory address generation, as the benchmarks often access elements from multiple arrays and several member variables of data structures. In most cases, the control-flow instructions are those to check the terminating condition of the loop. Benchmarks with a high % of control-flow instructions have if-else conditions within the loop kernel.

5.3 The MEDICS Processor Architecture

A computation system for portable medical imaging must meet several requirements in order to deliver high performance while still having low power utilization:

Low-latency FP computation: Most floating-point units (FPUs) have a latency of 3 to 4 cycles. The applications considered here often perform multiple consecutive FP operations on one piece of data before storing the result in memory. Chains of 4 or 5 dependent operations result in execution times of 12 to 15 cycles. Efforts must therefore be made to either reduce the latency of the FPU pipeline or implement low-power techniques to hide this latency.

Wide-SIMD FP pipeline: The algorithms in this domain are all FP-intensive; the representative benchmarks considered in this paper have, on average, 36% of FP instructions in the inner-most, most frequently-executed loops. Further, the loops are all do-all loops – all the iterations of the loops can execute in parallel as there are no inter-iteration dependencies. This property enables simple SIMD-ization of these loops, where subsequent iterations of the loop may be assigned to individial lanes in the SIMD datapath. Further, since SIMD replicates only the arithmetic units, there is comparatively less control overhead in a SIMD design compared to a single-issue design resulting in improved power efficiency. Many of these algorithms have some simple control flow in their inner-most loops, primarily limited to operations such as bounds-checking requiring some support for predication. The ability to broadcast a single value to all SIMD lanes is also required.

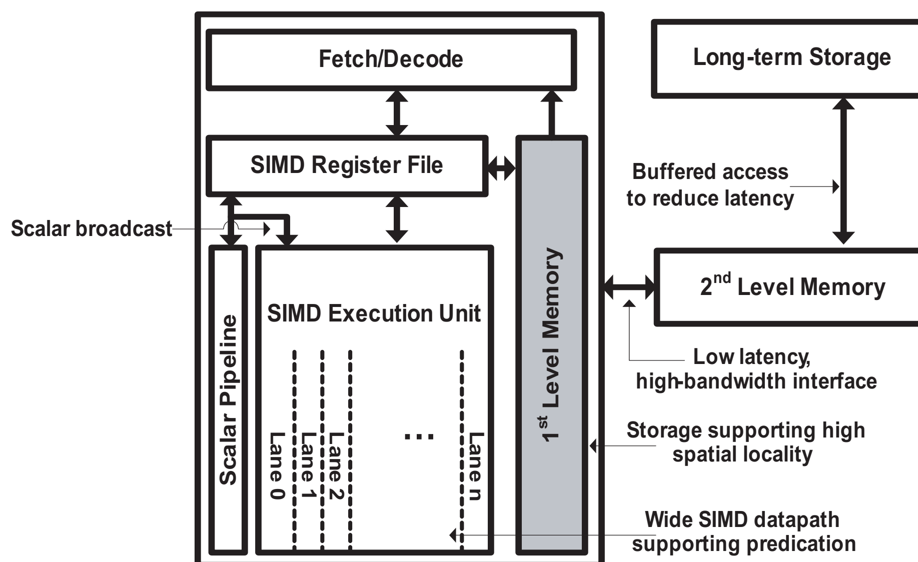


Figure 5.4: *Components required for a medical imaging compute system.*

High bandwidth, low-latency storage: Image reconstruction algorithms have a very high memory-to-compute ratios, requiring large amounts of data in a very short time. In order to support this behavior, a sufficient amount of memory has to be available on-chip. In addition to performing little compute per unit data, the total amount of data processed in reconstruction applications is also large. Several gigabytes of storage is therefore required, and should ideally placed as "close" to the processor as possible.

These requirements are combined to create a high-level system sketch for MEDICS shown in Figure 5.4 and explored in detail below.

5.3.1 FPU Pipeline

FP operations in reconstruction applications – especially those in frequently executed blocks of code – are FP adds, subtracts and multiplies. While the occassional divide is

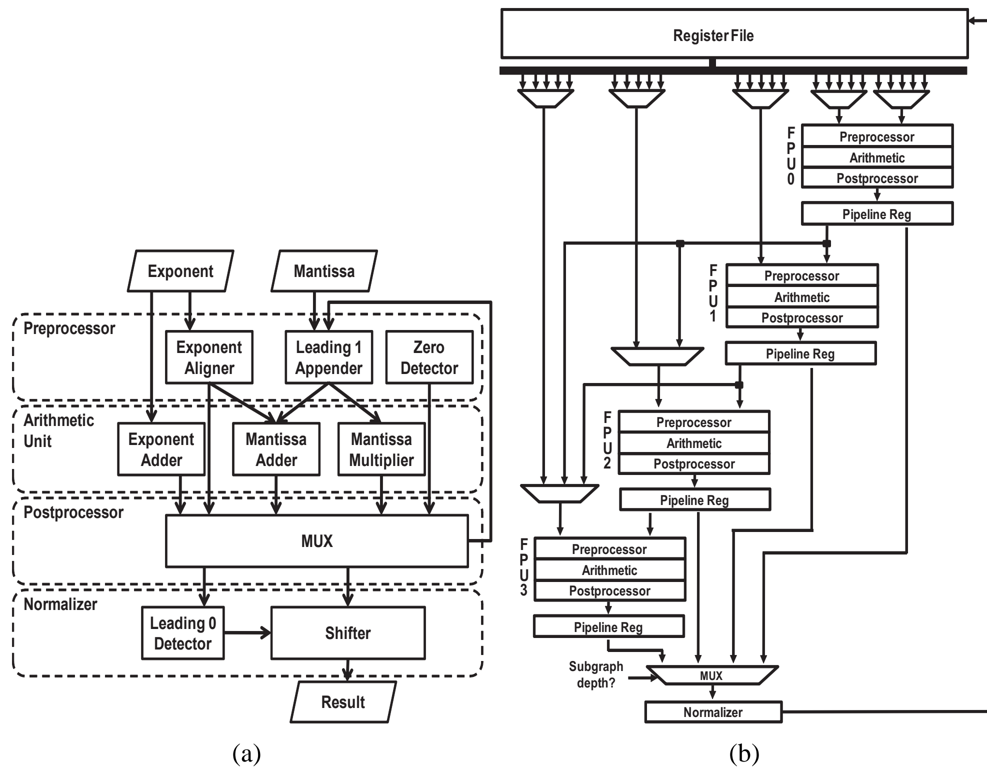
required, it is usually executed in software with the aid of a reciprocal operation. Efforts are therefore made in this work to optimize the main FPU pipeline, consisting of an FP adder/subtractor and a multiplier.

5.3.1.1 Reducing FPU Latency

For an FPU with a latency of 3 clock cycles, back-to-back dependent operations may only be issued every 3 cycles resulting in a significant loss in performance. Operation chaining in FPUs helps mitigate this latency, allowing some parallelism in the initial processing and formatting of FP numbers.

In addition to reducing data-dependent stalls, chaining operations has the advantage of reducing the overall number of register-file (RF) accesses; a sequence of two multiply instructions back-to-back, for example, normally requires a total of four read accesses and two write accesses. Chaining will reduce this to three read accesses and a single write access, though it will require an additional read port to perform all the reads simultaneously. The savings from reducing the overall number of accesses is often more than the added cost of a read port.

Reconstruction applications, however, typically have chains of dependent FP operations longer than two operations. This work explores the possibility of extending the principles applied in constructing conventional fused multiply-add FPUs further than is traditionally done allowing for one FP instruction to execute several FP operations in series to best accelerate long FP chains and to reduce RF access power. Further, these units have to be generalized to execute any sequence of FP operations in any order rather than just merely a fused multiply-add/subtract sequence.



(a) (b)

Figure 5.5: *FPU architecture. (a) Internal structure for FPU. (b) The Normalizer stage may be removed for all but the last in a chain of FPUs.*

Conventional FPU Architecture: A typical FP adder consists of a zero detector, an exponent aligner, a mantissa adder, a normalizer, and an overflow/underflow detector. The exponent aligner in the FP adder aligns the mantissa for the two operands so as to use the same exponent to compute the addition operation. Meanwhile, an FP multiplier generally consists of a zero detector, an exponent adder, a mantissa multiplier, a normalizer, and an overflow/underflow detector. Following the IEEE-754 standard, FP units also include rounders and flag generators. The rounder takes into account desired rounding modes, namely round to nearest, round toward 0, round toward positive infinity, and round toward negative infinity. The flag generator indicates whether the result is zero, not-a-number, generates an overflow, or generates an underflow. Figure 5.5(a) illustrates the interaction between these various components.

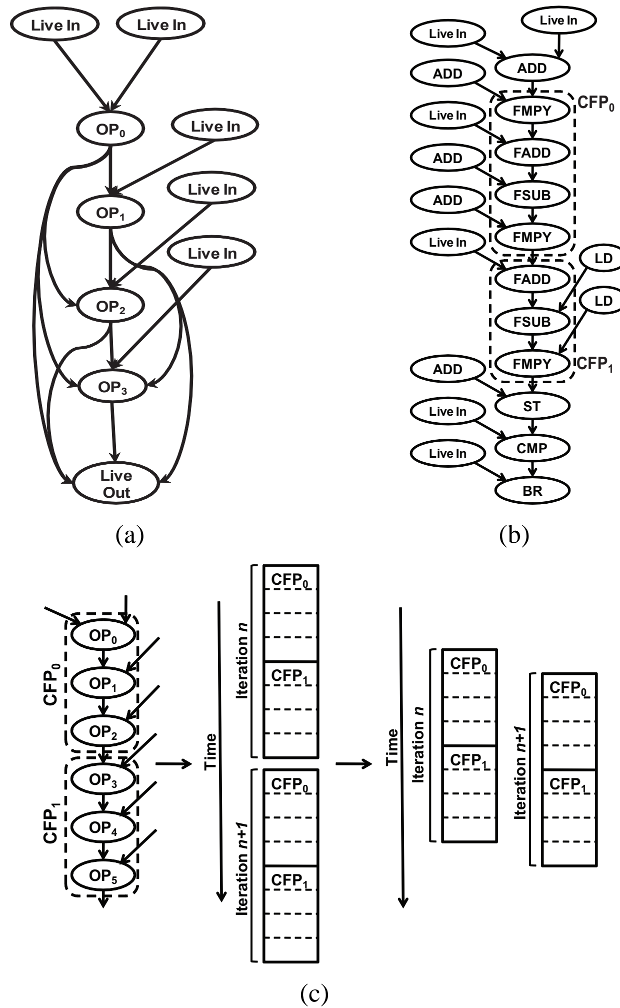


Figure 5.6: *Example using chained FPUs (CFPs) from the Radon benchmark. (a) Operation identification. (b) Latency hiding via software pipelining.*

Chained FPU Design: As depicted in Figure 5.5(a), the FPU implementation used in this work is divided into four conceptual stages: preprocessor, arithmetic unit, postprocessor and normalizer. Typically, all FPUs operate only on normalized FP numbers and this is enforced by the normalizer. In general terms, the earlier parts of the FPU pipeline consist of components that expand the IEEE standard-form input operands into intermediate representations suitable for the main arithmetic units, and the later parts of the FPU pipeline compact the results of the computation back into the IEEE representation. When

operations are performed back-to-back, the intermediate values are never committed to architected state and, as such, need not be represented in the standard form, saving time on the FPU critical path and reducing the required hardware.

When the normalizer takes the result from the postprocessor, it primarily detects leading zeroes and shifts them as necessary so as to conform to the IEEE-754 FP format. If multiple FPUs are chained together and the value computed in the postprocessor is only an intermediate value, and not one committed to architectural state, the normalizing step may be removed and the next stage in the FPU chain can treat this result as a denormalized value. The normalizer consumes a significant amount of computation time – close to 30% – so its removal results in marked performance improvements. More details about the achievable improvements are presented in Section 5.4.

Some extra logic that is necessary to make chaining work properly includes the truncation and shifter placed in the postprocessor to process varying result widths, resolve temporary overflows, and detect the true location of leading ones so that they may be correctly appended for the next stage of the computation. The 32-bit representation produced by the postprocessor simplifies the relay between one stage of FP computation to the next, and if deemed necessary, outputs can be pulled out at any FPU to be normalized immediately and its results obtained. The conceptual operation of the chained design is illustrated in Figure 5.5(b) where the normalization step is performed for only the final operation in the sequence. Additional control logic allows for earlier normalization of intermediate values if a sequence of instructions has fewer than the maximum number of operations. Operations can receive their inputs from either a predecessor or from the register file.

Identifying FP chains: Modifications made to the Trimaran [104] compiler are used to identify and select sequences of instructions for execution on the chained FPU. First, an abstract representation of the possible sequences of execution is created in a data-flow graph (DFG) form. In the DFG, nodes are used to represent each input, output and individual FPUs in the chain. Directed edges are used to represent all possible communication between these nodes. An abstract graph representation for the 4-Op chained FPU in Figure 5.5(b) is shown in Figure 5.6(a). This representation has 5 input nodes, 1 output node and 4 operation nodes. Two edges are drawn from the inputs to OP0 and one edge is drawn to one input of OP1, OP2 and OP3. Edges are also drawn from OP0 to OP1, OP2, and OP3; from OP1 to OP2, and OP3; from OP2 to OP3; and, finally, from all of the operation nodes to the output node.

The compiler receives the application source code and this abstract representation as input. It then draws a DFG for the compute-intensive inner-most loop of the benchmark. We the find subgraphs in the application's DFG that are isomorphic to the FPU's DFG, which provides us with the set of operations that can be executed as one instruction in the FPU chain. A greedy algorithm is then used to select the largest of these subgraphs; i.e. a single, 4-long sequence of operations is preferred over two 2-long sequences.

Figure 5.6(b) shows a subset of the DFG for the Radon transform benchmark's inner-most loop. Two chained FPUs are identified here - one four FP operations in length and the other three FP operations in length. The sub-graphs used in this work were sequences of dependent FP add, subtract, and multiply operations where intermediate values were not live-out of the DFG but were only consumed locally. The internal interconnection is illustrated in Figure 5.5(b).

5.3.1.2 Hiding FPU Latency

While the total time taken to execute several back-to-back FP operations may be reduced using FPU chaining, it significantly increases the total pipeline depth and, consequently, the latency of any FP instruction. Traditional architectures use hardware multithreading to hide various sources of latency – control latency, computation latency, memory latency, etc. While hardware multithreading helps increase the overall performance, it has a few drawbacks. Firstly, the control when using multithreading is significantly more complicated as each thread has its own PC, machine status register, execution trace, etc. In addition, each thread must be presented with the same architectural state. This work takes a compiler-directed approach of hiding the long FPU pipeline latency by software pipelining the inner-most loops [76] and overlapping independent successive iterations as shown in Figure 5.6(c). When using software pipelining, since more data is being processed in parallel, the RF must be increased in size to provide enough data for the required computation. Instantaneous power will also increase due to the increase in operations at any given time, but the overall energy of computation will decrease since the processor is spending more time doing useful work rather than idling and waiting to issue a new instruction.

5.3.2 Data Compression

5.3.2.1 Lossy Compression

One approach to narrow the bandwidth gap and make more effective use of the computing resources available is to reduce the total amount of data required by the application. Iterative CT reconstruction techniques [44, 101] are inherently error-tolerant since

they keep iterating until the number of artifacts in the image is within some tolerance. A technique used in MEDICS that exploits this in a power-reducing manner is the use of 16-bit floating-point computation mentioned in the IEEE 754-2008 (or IEEE 754r) standard. These “half-precision” floating-point values consist of a sign bit, a 5-bit exponent field and a 10-bit mantissa field as opposed to the 8-bit exponent field and 23-bit mantissa field used in the 32-bit standard.

The advantages of changing the floating-point width are two-fold. The first advantage is an effective doubling of bandwidth; i.e. twice as many operations can now execute using the same bandwidth as before. The authors of [57] explored the effects of using 16-bit floating point on CT image reconstruction. Not only did [57] conclude that the error produced by this reduced precision was well within tolerance but that the resulting increase in bandwidth has shown over a 50% increase in image reconstruction performance on Intel CPUs.

The second advantage is a reduction in the datapath hardware; our experiments showed a 58.5% reduction for a 2-input FPU. Further, the FP register file size can also be reduced to a 16-bit width. The effects of this change are discussed in section 5.4.

5.3.2.2 Lossless Compression

Several recent studies have demonstrated the effectiveness of compression in reducing the size of CT scan results. These studies have primarily been focused on long-term storage size, but similar principles are applied in this work to improve performance. In [3], the Lempel-Ziv and Huffman lossless compression algorithms are used to compress sinogram data from PET scans. In a similar manner, the JPEG-LS lossless compression algorithms

are applied to compress sinogram data from CT scans [5]. They explore using lossy compression algorithms as well and demonstrate that compression ratios up to 20:1 are possible with minimal artifacts in the final reconstructed image. Lossless compression of sinograms results in 10:1 and greater compression ratios. An important point to note is that the compression ratios when lossless techniques are applied to final images are considerably less – on the order of 2:1 and 3:1. Sinograms are generally much more compressible than images due to their highly correlated structure.

Further, rather than using software techniques for compression, this work takes the approach of using ASICs for performing compression and decompression. Using software-based compression provides greater flexibility in terms of the compression algorithm used and at what granularity data has to be compressed. However, it leads to a considerable loss of performance, up to 50% [84] in some cases, without accounting for the fact that the main application cannot execute on the core while the compression and decompression software is running. For this reason, this work focuses on a hardware-only approach to compression using ASIC implementations of the JPEG-LS compression algorithm [70, 71, 84]. For a 32-bit pixel, this hardware implementation has an area footprint of $0.07mm^2$, power consumption of 1.6mW, and can operate at a frequency of 500 MHz.

5.3.3 Memory System

In this work, a 3D die-stacked DRAM array is used for high-density, low latency storage. Where traditional designs incur a latency of several hundred cycles while accessing on-board DRAM storage, utilizing 3D stacking provides the storage density of DRAM but without incurring the wire delay penalty of going off-chip. The 3D DRAM cells are

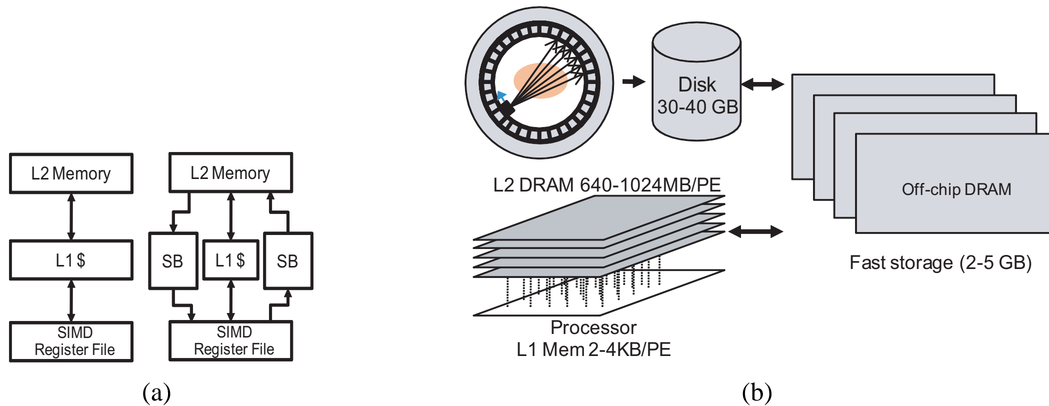


Figure 5.7: *Memory Interfaces. (a)Datapath-to-DRAM memory system. A large L1 or a smaller L1 with input and output streaming FIFOs may be used. (b)Off-chip memory system.*

accessed using through-silicon vias (TSVs). TSVs are essentially identical to the vias normally used to connect adjacent metal layers in silicon [55]. The specific implementation used in this paper is that provided by Tezzaron Corporation [100]. This implementation, at a 130nm technology, has a density of 10.6 MB/layer-per-mm^2, a DRAM-to-core latency of 10ns, a data throughput of 4 giga-transfers/sec, and supports 10,000 TSVs each mm^2; i.e., for a memory interface of area 0.1 mm^2, the DRAM-to-core memory bandwidth supported is up to 500 GB/s, considerably higher than any off-chip memory bandwidth currently offered, and much beyond the bandwidth requirements of most applications, including that of medical image reconstruction. Further, when deployed on a core running at 500 MHz, the 10ns delay results in a total latency of approximately 10 cycles to send the required memory address to the DRAM and receive the appropriate data. This latency is comparable, or faster, to modern L2 caches and, for this reason, the 3D-DRAM storage will henceforth be known as “L2 memory” in this paper.

5.3.3.1 On-chip Memory Organization

Two potential on-chip memory systems are considered, as depicted in Figure 5.7(a). The first, more traditional system, is to use an L1 SRAM cache between the SIMD RF and the L2 memory in order to hide the latency of an L2 memory access. The second system takes advantage of the streaming nature of these benchmarks by employing “streaming buffers” (SBs) to continuously fetch and store data at pre-specified offsets. These are two alternatives to the approach used in modern GPGPUs – utilizing a large number of thread contexts and some caching to keep the computation units busy while data is fetched from memory. All three techniques, as they apply to this domain, are explored in Section 5.4.

5.3.3.2 Off-Chip Memory Organization

Sinogram data generated in reconstruction systems is first stored in a “recon box” hard disk drive. The data on the drive is then processed and reconstructed using whichever processor is used in the machine. A similar design is envisioned for the system proposed in this work as shown in Figure 5.7(b). Additional steps are taken, however to further hide the long latency of a disk access. This is done by using a very large array of off-chip DRAM (2 to 5 GB). The on-board storage size selected is an appropriate match for repeated access to the large working set size required by advanced reconstruction algorithms, as mentioned in Section 5.2.

5.4 Experimental Evaluation

The major components of MEDICS were designed in Verilog and synthesized at 500 MHz on a 65nm process technology using the Synopsys Design Compiler and Physical Compiler. Power results were obtained via VCS and Primetime-PX, assuming 100% utilization. Area and power characteristics for regular memory structures like dual-ported RFs and caches were obtained through a 65nm Artisan memory compiler while RFs with more than 2 read ports were designed in Verilog and synthesized. The benchmarks were SIMD-ized by hand and compiled using the Trimaran compiler infrastructure [104]. A SIMD width of 64 was chosen as having a wider datapath led to the scalar broadcast interconnect being on the critical path and reducing the efficiency of the processor.

5.4.1 FPU Chaining

Table 5.1 indicates that a number of the applications in this domain have several long sequences of back-to-back FP operations. Based on this data, the FP datapath in MEDICS was designed with an FPU consisting of 4 back-to-back operations. Figures 5.8 and 5.9 show the effects of varying the number of operations in the FPU.

In Figure 5.8, the x-axis for all the graphs, "FP ops/instruction" is the number of successive, dependent FP operations executed in the chained FPU. The "latency" graph shows the time (in 2ns clock cycles) taken to execute an input subgraph. The baseline 3-cycle FPU takes 3 cycles for each operation and thus has a latency that increases by 3 cycles for every added operation. The removal of redundant hardware in the optimized FPU chain re-

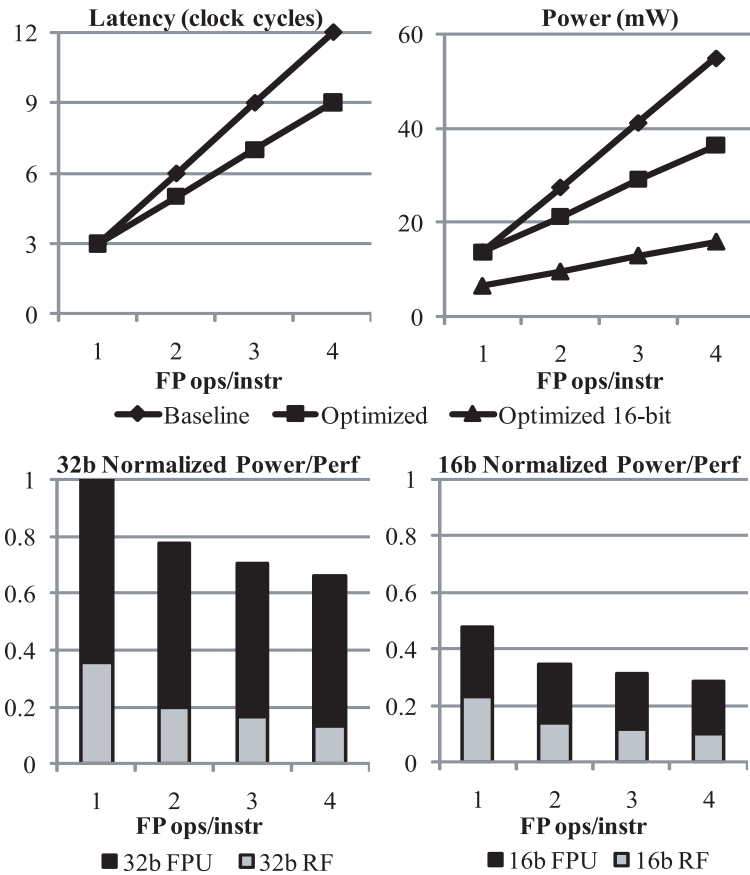


Figure 5.8: *Datapath latency, power and area effects of varying the number of FPUs when increasing FPU chain length.*

sults in significantly less overall latency – a savings of 3 cycles when 4 FPUs are connected back-to-back.

The "power" graph illustrates the power savings obtained from optimized normalizing. Here, the baseline is multiple un-modified FPUs executing operations back-to-back. In this graph, too, the gap between optimized and unoptimized FPUs widens quite dramatically as the number of operations per instruction increases. This gap widens further when the width of the datapath is reduced to 16 bits as mentioned in Section 5.3.2.1. The normalizing hardware removed as part of the FPU optimization process removes a higher percentage of the hardware in a 16-bit FPU than in a 32-bit FPU, resulting in reduced baseline and

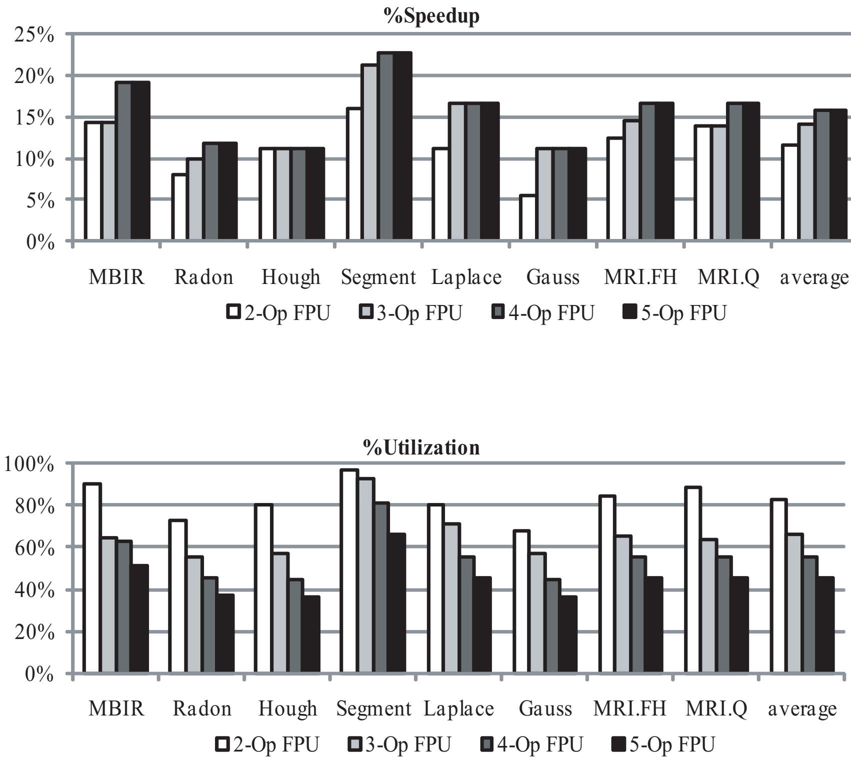


Figure 5.9: *Speedup and FPU utilization with increasing chain length.*

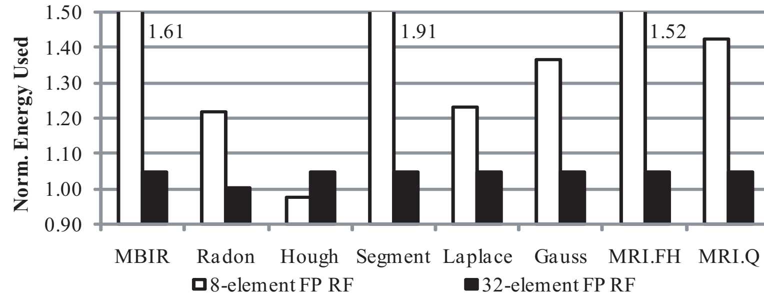
incremental power with each added FPU in the chain. The power measurement in this graph is the sum of the RF access power and the FPU execution power to better reflect the power penalty from increasing the number of RF read-ports.

The “power/perf” graphs address the efficiency of the different solutions. They show the normalized power consumed per operation to achieve an IPC of 1; i.e., with every stage of the FPU occupied and busy. Here, too, the power consumed for the amount of work done steadily reduces as the number of FP operations per instruction increases. While the access power for an RF increases for every added read port, the improvement in the efficiency of the RF shown in the graph indicates that this is amortized by the reduction in overall accesses and by the performance improvement achieved by chaining together FPUs.

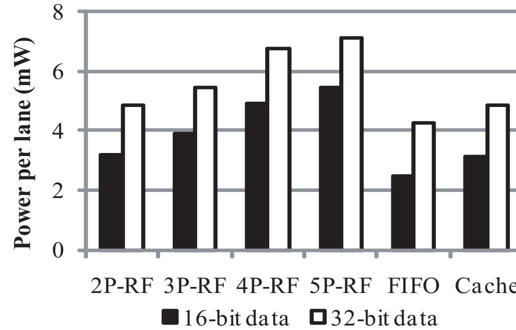
Figure 5.9 shows the speedup and FPU utilization observed for the different benchmarks when using 2-, 3-, 4-, and 5-operation chained FPU datapaths. The speedup varies based on how frequently the various chained FP operations occur in each benchmark (see Table 5.1) and the latency penalty incurred when issuing back-to-back dependent FP operations. The benchmarks that had the most to gain from chaining FPUs were the ones with the most number of 4-long chains of FP operations – `MBIR` and `Segment`, for example. On average, 12%, 14% and 16% speedups are observed when using 2-op, 3-op and 4-op chained FPUs, respectively. Speedup saturates at 4 operations and adding a fifth operation in the chain only reduces the overall utilization of the FPU. Using a 4-op chain is, therefore, the best solution.

5.4.2 Local Storage

Two aspects of local storage were evaluated. The first is the number of entries in the main RF. A baseline RF size of 16 elements was chosen as this captures the register storage requirement for the majority of the benchmarks shown in Table 5.1. The larger an RF, the more power it consumes per access, and so if only the instantaneous power consumption is considered, using a smaller RF is the preferred solution. However, this does not account for the increase in the application run-time associated with spill instructions inserted due to a lack of available registers. Therefore, the metric chosen to evaluate the efficiency of various RF sizes was the energy consumed by individual iterations of the applications, accounting for RF access and execution energy. Figure 5.10(a) shows the energy consumption when using a 32-bit 8-entry RF and a 32-bit 32-entry RF, normalized to that of a 16-entry RF. The energy consumed when using an 8-entry RF is significantly higher than 1 – almost



(a)



(b)

Figure 5.10: *Local storage characteristics. (a) Energy consumption when using 8-entry and 32-entry RFs, normalized to energy consumption of 16-entry RF. (b) Power consumption of each additional RF context, using a FIFO streaming buffer and using an L1 cache to hide L2 memory latency. The FIFO and context-based solutions include an additional 16-byte/lane L1 cache for register spill.*

double in the case of the `Segment` benchmark – and is only less than 1 for the `Hough` benchmark, which requires only 8 registers. The energy overhead of using a 32-entry RF is quite small – less so for the `Radon` benchmark since it actually requires 18 registers to not have any spill code. Therefore, while a 16-entry RF is chosen for the MEDICS design, the added power and energy overhead of increasing this to a 32-entry RF is quite minimal.

The second aspect of local storage that was evaluated was the mechanism used to hide the latency of the L2 memory. Three different solutions were analyzed using an L1 cache, using streaming FIFO buffers, and replicating register contexts. To hide 10 cycles of L2 memory latency, sufficient buffering is required for 10 memory instructions, or 40 bytes

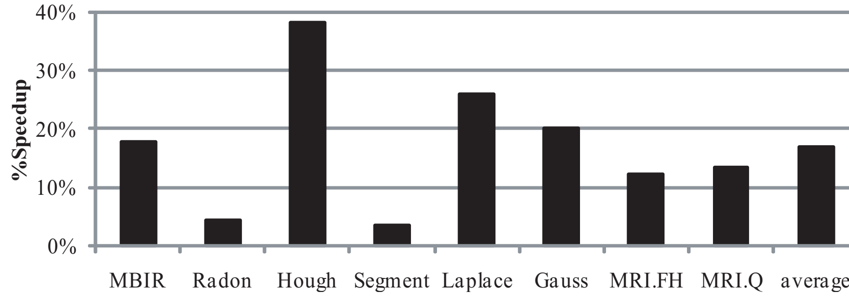


Figure 5.11: *Speedup using a FIFO streaming buffer instead of an L1 cache.*

for 32-bit data values. This number is increased to the nearest power of 2 to 64 bytes per SIMD lane. This additional storage may be used for miscellaneous data like register spill.

An L1 cache of 64 bytes/lane can be used, but this does not effectively exploit the spatial locality present in these applications. Since all of these algorithms process images sequentially, pixel-by-pixel, a 64 bytes/lane streaming data buffer FIFO and DMA engine coupled together to transfer multiple loop iterations worth of data in response to a single request is a better solution. This process reduces execution time since explicit address computation for loads and stores may be eliminated from the main datapath. There is added performance overhead of programming the FIFO, but this is amortized over the length of the loop. A small L1 cache of 16 bytes would still required for miscellaneous data as mentioned earlier. Another technique, used in modern GPGPUs, is to hide memory latency by simply using thousands of register contexts.

The power consumptions of these different techniques was evaluated, with both 16-bit and 32-bit datapaths, and the results are shown in Figures 5.10(b) (here, “*n*P-RF” is the power overhead per context when utilizing 16-entry RFs with *n* read ports). This power is multiplied for each context required, i.e. if a processor has 10,000 16-entry contexts, the power consumed by the dual-ported register files will be approximately 5 Watts. Due

to its high power consumption, the common technique of having several parallel contexts is the least economical. This is evident even though the L2 memory latency that has to be hidden is very low, compared to that of the off-chip DRAMs used in modern GPGPUs. The overhead naturally increases as read ports are added to RFs to support FPU chaining. The cache consumes approximately the same amount of power as an additional dual-ported RF. The streaming FIFO buffers, though, consume less power than either solution, primarily due to the lack of any addressability. The case for the streaming buffers is further underscored in Figure 5.11 which illustrates the speedup achieved from not having to always issue address-calculating instructions when using the streaming buffers – 17%, on average.

Based on the performance improvement observed and their low power consumption, FIFO buffers and a small L1 cache provide the best interface between the datapath and L2 memory. It is not necessary to add hardware to maintain coherence between these structures; the primary input and output data structures of the applications considered are accessed through the FIFO buffers and the L1 is only used for storing any intermediate values such as register spill. Therefore, as there is no overlap between data stored in the L1 and data stored in the FIFOs, area and power-expensive hardware coherence schemes need not be used in this design.

5.4.3 Compression

Two options for integrating hardware compression were considered: in-pipeline and compression at the memory bus as shown in Figure 5.12(a). In the first approach, the compression and decompression engines are placed directly in the main datapath. In this scheme, compressed data is stored on the disk, on the on-board, off-chip DRAM, and

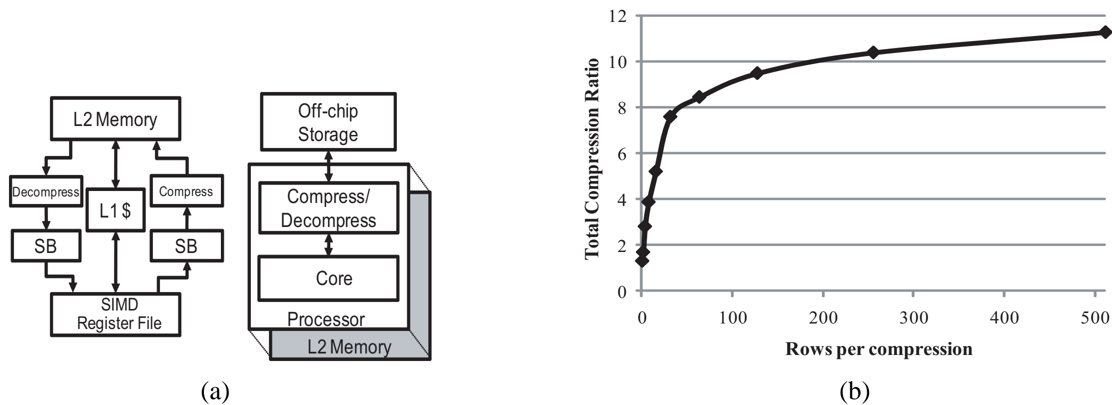


Figure 5.12: *Image compression. (a) Datapath-to-DRAM memory system with two different possible configurations for using compression and decompression engines. (b) Degrading compression ratios when increasing the granularity of loss-less compression.*

in the L2 memory. Compression and decompression hardware is placed between the L2 memory and the streaming buffers. The DMA engine responsible for filling and clearing the streaming buffer FIFOs triggers the decompression and compression, respectively. The decompressed data, however, must fit in the streaming buffer FIFO which is only a few bytes in size per lane. Considering each row is 2 kB in size, a small subset or fraction of rows in each sinogram must be compressed together rather than compressing an entire 512-row sinogram.

In the second approach, only the disk and on-board storage hold compressed data and data is decompressed before it is stored in the L2 memory. The main advantage with this approach is the improvement in compression ratios. Figure 5.12(b) shows the typical degradation of compression ratios as fewer and fewer rows of a sinogram are compressed. If an entire slice (512 rows) is compressed, its size shrinks from 1.05 MB to 93.2 kB (11.3:1 compression), whereas if the slice is compressed row-by-row, its overall size is 1.02 MB (1.3:1 compression) – a negligible reduction in size. Due to the improved compression achieved by decompressing data into the L2 memory, MEDICS's positioning of the com-

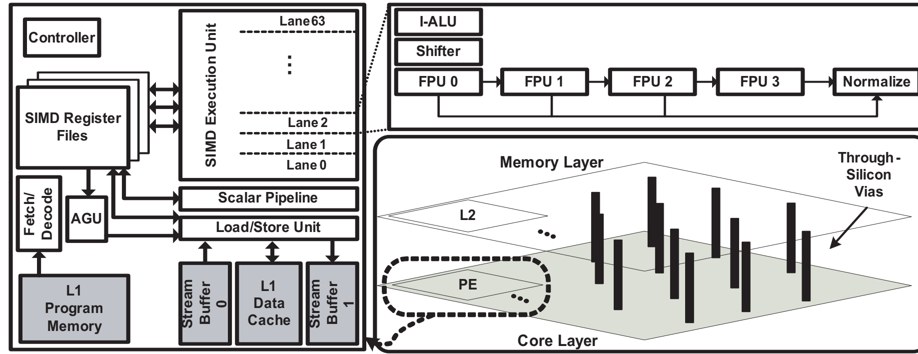


Figure 5.13: *MEDICS processor architecture. The figure on the bottom-right conceptually illustrates the design of the entire chip. The figure on the left shows the architecture of an individual PE. The figure on the top-right shows the arithmetic execution pipeline.*

pression/decompression engines on the boundary of on-chip and off-chip storage is the better of the two techniques.

Using compression engines on the memory bus interface allows increasing the total number of PEs on the processor, thereby increasing the total processing capability. The decompression engines used have a throughput of 1 pixel/cycle, resulting in a latency of 2^{18} cycles per 512x512 image. At this rate, using a 64-wide SIMD PE, at least 64 instructions need to execute per 4-byte pixel generated in order to not be limited by the decompression engines. While this is feasible for the benchmarks with lower memory footprints, the ones with higher memory footprints require almost eight times this throughput, or 8 decompression engines running in parallel, processing different chunks of data.

5.5 The MEDICS System

The MEDICS architecture is shown in Figure 5.13. The 3D-stacked DRAM interface is illustrated in the lower-right part of the figure. An individual PE is illustrated on the left, showing the separate scalar and vector pipelines. The top-right shows an individual lane

	16-bit	32-bit
Frequency	500 MHz	
SIMD Lanes	64	
Peak Performance	128 GFLOPs	
Peak Total Power	1.58W	3.05W
Total Area	36.3 mm^2	40.7 mm^2
On-chip DRAM	774 MB	867 MB
Efficiency	81.1 Mops/mW	42.9 Mops/mW

Component	16-bit Power	32-bit Power
4-op 16-bit FPU	10.30 mW/ln.	29.15 mW/ln.
16-element 16-bit RF	5.48 mW/ln.	7.14 mW/ln.
Local stream buffers	2.51 mW/ln.	4.25 mW/ln.
Etc. datapath (scalar pipe, AGU, control)	59 mW	
DRAM Power	350 mW	390 mW

Figure 5.14: *MEDICS specifications. Overall per-PE specifications and power breakdown of individual components.*

of the vector pipeline in more detail. A compound FPU which does 4 FP operations back to back is used in this design. The RF has 16 elements for each of the FP and integer RFs based on the data in Table 5.1 and FIFO stream buffers are used to transfer data between the datapath and the L2 memory.

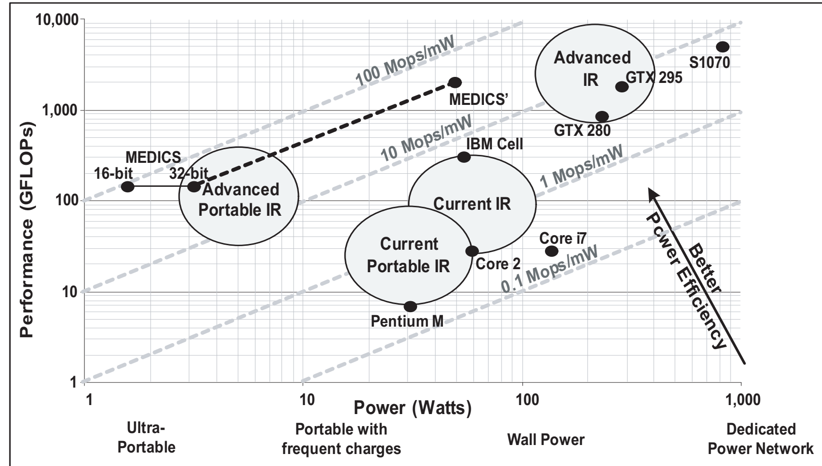
The MEDICS processor's design characteristics and power consumption breakdown are shown in Figure 5.14. The top part of Figure 5.14 shows the specifications of each individual PE. The 16-bit version consumes significantly less power than the 32-bit version. However, due to the reduced size of the FP datapath, the area is also reduced, leading to approximately 11% less on-chip stacked DRAM. This is not a problem, though, since the reduced bitwidth effectively leads to a doubleing of the DRAM's utilization. The bottom part of Figure 5.14 shows a component-by-component breakdown of the power consumed

in the MEDICS processor. The most significant power reduction from the 32-bit to the 16-bit datapath is seen in the 4-op, 5-input FPU which sees a 64% reduction.

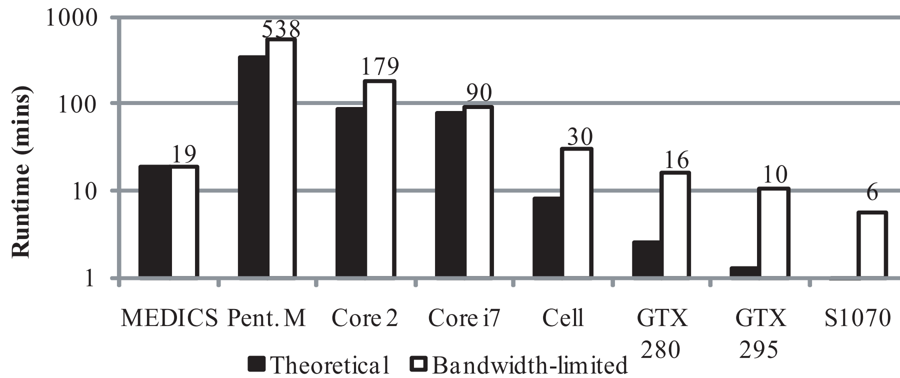
In addition are the area and power of the 8 sets of compression/decompression engines required for sufficient throughput, the power consumption of the processor changes very little with this addition and the area increases by 1.2mm^2. Using this compression mechanism, the effective bandwidth seen by the processor is 10X the nominal, allowing for an equivalent increase in the amount of processing power and improved scalability while maintaining performance/power efficiency.

Figure 5.15(a) shows a modified Figure 5.1. Extra data points have been added to show how MEDICS compares in performance/power efficiency to the other processors in consideration. The 32-bit MEDICS system has an efficiency improvement of 10.6X over the Nvidia GTX 280 and 6.8X over the Nvidia GTX 295, which are also fabricated on a 65nm process. The 16-bit MEDICS system has a 20.5X and 13.1X improvement, respectively. Its performance and power consumption makes it an excellent choice for advanced, low-power image reconstruction. If more PEs are used, and the necessary compression engines and on-chip communication network are added, the design scales to the denoted as MEDICS' on the plot. At this point, it delivers the *same* performance as the GTX 295 while consuming significantly less power.

For an off-chip bandwidth of 141 GB/s (the same as that of Nvidia GTX 280), MEDICS has a peak data consumption of 1.11 bytes/instr. Given the benchmarks in consideration, the only benchmark that would be bandwidth-limited is the `Segment` benchmark, which requires 1.26 bytes/instr. The peak consumption of the other processors in consideration, however, ranges between 0.08 bytes/instr for S1070 to 0.85 bytes/instr for the Core i7, all



(a)



(b)

Figure 5.15: *(a) (Modified Fig. 5.1) Suitability of MEDICS for the performance and power characteristics of the domain (b) Theoretical and realized (bandwidth-limited) run-times for advanced reconstruction algorithms.*

lower than what is required for this domain. The ramification of this disparity is best illustrated in Figure 5.15(b). This graph shows the overall run-time of the MBIR reconstruction application [101], assuming that the application requires the full 144 trillion operations specified in Section 5.2. The "theoretical" bar shows what the run-time would be if the listed peak performance rating were actually possible. The "realized" bar shows what is actually possible given the bandwidth constraints of the various processors under consideration.

While consuming 1 to 2 orders of magnitude less power than all the other existing solutions, MEDICS delivers reconstruction run-times that are matched only by high-end desktop GPGPUs. It is only significantly outperformed by the server-class S1070 which consumes over 100X the power while only reducing the run-time by two-thirds. The principle efficiency improvements come from:

- Optimized FPU design
- Fewer hardware contexts
- Increased bandwidth to off-core storage
- Improved on-chip latency-hiding
- Removing power-hungry application-specific hardware (e.g. texture units)

5.6 Related Work

There are multiple current hardware solutions for medical image reconstruction, based on DSPs [99], general-purpose processors [103] and GPGPUs [65]. These are unsuitable for the next generation of low-power image reconstruction systems for the reasons enumerated in Section 5.1, such as lack of floating-point supprt, insufficient performance and high power consumption.

There are several other examples of low-power, high-throughput SIMD architectures like SODA [110] for signal-processing but it, being a purely integer architecture, is unsuitable for this domain space. There are also high-throughput floating point architectures such as TRIPS [81], RAW [95] and Rigel [49], but these are more focused on general-purpose

computing and do not have the same power efficiency as MEDICS, nor do they address any bandwidth concerns. There has also been recent work on image-processing [41, 58] and physics simulations [113] targetting domains traditionally addressed by commercial GPGPUs but these, too, do not address bandwidth and power to the extent that this work does.

Some CRAY systems use a "chained FPU" design. However, these are essentially just forwarding paths across different SIMD lanes. While this connectivity reduces register acceses, the FPU itself was not redesigned the way the MEDICS FPU was.

Other prior work specifically targetting medical imaging has predominantly focused on using porting existing programs to the commercial products mentioned earlier. For instance, in [45], the authors port "large-scale, biomedical image analysis" applications to multi-core CPUs and GPUs, and compare different implementation strategies with each other. In [80], the authors study image registration and segmentation and accelerate those applications by using CUDA on a GPGPU. In [92], the authors use both the hardware parallelism and the special function units available on an Nvidia GPGPU to dramatically improve the performance of an advanced MRI reconstruction algorithm.

5.7 Conclusion

The MEDICS architecture is a power-efficient system designed for efficient medical image reconstruction. It consists of PEs of wide SIMD floating-point engines designed around the computation requirements of the image reconstruction domain. Each PE achieves a high performance-per-power efficiency by using techniques such as FPU-chaining, streaming

buffers and compression hardware. As applications in this domain are normally executed on high-performance, general-purpose processors and GPGPUs, these architectures were used to gauge the performance and efficiency of MEDICS. The results are very encouraging, with MEDICS achieving over 20X the power efficiency. The design is also bandwidth-balanced so that all of the performance available on the processor may be effectively used for computation.

CHAPTER VI

General Accelerators for High-Throughput, Data-Parallel Applications

6.1 Introduction

Scientists have traditionally relied on large-scale supercomputers to deliver the computational horsepower to solve their problems. This landscape is rapidly changing as relatively cheap computer systems that deliver supercomputer-level performance can be assembled from commodity multicore chips available from Intel, AMD, and Nvidia. For example, the Intel Xeon X7560, which uses the Nehalem microarchitecture, has a peak performance of 144 GFLOPs (8 cores, each with a 4-wide SSE unit, running at 2.266 GHz) with a total power dissipation of 130 Watts. The AMD Radeon 6870 graphics processing unit (GPU) can deliver a peak performance of nearly 2 TFLOPs (960 stream processor cores running at 850 MHz) with a total power dissipation of 256 Watts. For some applications, including medical imaging, electronic design automation, physics simulations, and stock pricing models, GPUs present a more attractive option in terms of performance, with speedups of up to 300X over conventional x86 processors (CPUs) [61, 75, 102, 86, 79].

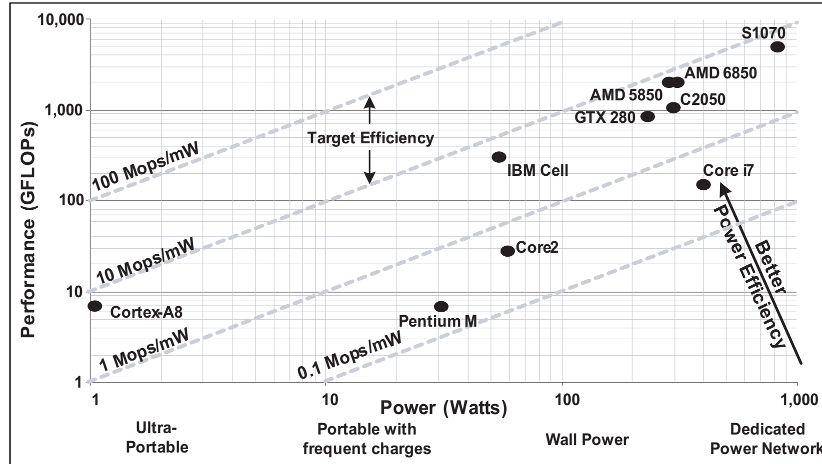


Figure 6.1: *Peak performance and power characteristics of several high-performance commercial processors and GPUs are provided: ARM Cortex-A8, Intel Pentium M, Core 2, and Core i7; IBM Cell; Nvidia GTX 280, Tesla S1070, and Tesla C2050; and AMD/ATI Radeon 5850 and 6850.*

However, these speedups are not universal as they depend heavily on both the nature of the application as well as the performance optimizations applied by the programmer [53]. But, due to their peak performance benefits, GPUs have emerged as the computing substrate of choice for many scientific applications.

A natural question is, "what is the proper substrate for scientific computing – CPUs or GPUs?" This paper takes the position that the answer to this question is *neither*. CPUs are more focused on scalar program performance and do not have sufficient raw floating-point computation resources. Conversely, GPUs suffer from two major problems that limit scalability as well as the ability to deliver the promised throughputs for a wide range of applications: high power consumption and long memory access latencies. Rather, a new solution is required that offers high raw performance, power efficiency, and a tenable memory system.

Though power is not necessarily a significant drawback for video game graphics acceleration, requiring powerful cooling systems is a significant impediment for more portable

platforms. Some of the algorithms that are commonly accelerated by GPUs are often deployed in systems where portability or power consumption is a critical issue. For instance, polynomial multiplication is used in very advanced cryptographic systems, real-time FFT solving is required for complex GPS receivers, and low-density parity-check error correcting codes are used in WiMAX and WiFi. Monte Carlo Recycling or options-pricing algorithms for computational finance, are often deployed in dense urban areas where, while portability is not an issue, power and cooling certainly is an important cost concern. Even if the power consumption of a single GPU is not a concern, combining multiple of these chips together to produce higher performance systems is untenable beyond a modest number (e.g., 1,000 Nvidia GTX 280s to create a PetaFLOP system would require 200 kW).

To understand the trade-offs more clearly, Figure 6.1 presents performance-vs-power trade-offs for a variety of CPUs and GPUs. The GTX280 consumes over 200W of power while achieving a peak performance of 933 GFLOPs, resulting in a relative low power-efficiency of less than 4 Mops/mW. Other solutions that push performance higher, like the Tesla S1070, can consume over 1kW. Other general-purpose solutions from IBM, Intel, and ARM, while consuming significantly less power, have similar or worse performance-per-Watt efficiency. The Core-2 and Core-i7, while consuming less power than the GPU solutions also fall behind in terms of peak performance even when using the single-instruction multiple-data (SIMD) SSE instructions, leading to an overall loss of efficiency.

To overcome the limitations of CPUs and GPUs, we take the approach of designing a processor customized for the scientific computing domain from the ground up. We focus on dense matrix scientific applications that are generally data parallel. The *PEPSC* processor is designed with three guiding principles: power efficiency, maximizing hard-

ware utilization, and efficient handling of large memory latencies. Our goal is one TFLOP performance at a power level of 10's of Watts at current technology nodes. As shown in Figure 6.1, this requires increasing the energy efficiency of modern CPU and GPU solutions by an order of magnitude to approximately 20-100 Mops/mW. One solution would be to develop ASICs or hardwired accelerators for common computations [107]. However, we believe this approach is orthogonal and rather focus on a fully programmable SIMD floating point datapath as the starting point of our design. While CPUs and GPUs have economies of scale that make the cost of their chips lower than PEPSC could ever achieve, the design of PEPSC is useful to understand the limitations of current GPU solutions and to provide microarchitectural ideas that can be incorporated into future CPUs or GPUs.

This paper offers the following contributions:

- An analysis of the performance/efficiency bottlenecks of current GPUs on dense matrix scientific applications (Section 6.2).

- Three microarchitectual mechanisms to overcome GPU inefficiencies due to datapath execution, memory stalls, and control divergence (Sections 6.3.1, 6.3.2, and 6.3.3, respectively):

    1. An optimized two-dimensional SIMD datapath design which leverages the power-efficiency of data-parallel architectures and aggressively fused floating-point units.

    2. Divergence-folding integrated into the FP datapath to reduce the cost of control divergence in wide SIMD datapaths and exploit "branch-level" parallelism.

3. Dynamic degree prefetching to more efficiently hide large memory latency while not exacerbating the memory bandwidth requirements of the application.

- An analysis of the performance and power-efficiency of the PEPSC architecture across a range of scientific applications.

6.2 Analysis of Scientific Applications on GPUs

GPUs are currently the preferred solution for scientific computing, but they have their own set of inefficiencies. In order to motivate an improved architecture, we first analyze the efficiency of GPUs on various scientific and numerical applications. While the specific domains that these applications belong to vary widely, the set of applications used here, encompassing several classes of the Berkeley "dwarf" taxonomy is representative of non-graphics applications executed on GPUs.

6.2.1 Application Analysis

Ten benchmarks were analyzed. The source code for these applications is derived from a variety of sources, including the Nvidia CUDA software development kit, the GPGPU-SIM [6] benchmark suite, the Rodinia [11] benchmark suite, the Parboil benchmark suite, and the Nvidia CUDA Zone:

- binomialOptions (`binOpt`) The binomial option pricing method is a numerical method used for valuing stock options.

- BlackScholes (`black`) A pricing model used principally for European-style options that uses partial differential equations to calculate prices.

- Fast Fourier Transform (`fft`) A high-performance implementation of the discrete Fourier Transform, used for converting a function from the time domain to the frequency domain.
- Fast Walsh Transform (`fwt`) The matrix product of a square set of data and a matrix of basis vectors that are used for signal/image processing and image compression.
- Laplace Transform (`lps`) An integral transform for solving differential and integral equations.
- LU Decomposition (`lu`) A matrix decomposition for solving linear equations or calculating determinants.
- Monte-Carlo (`mc`) A method used to value and analyze financial instruments by simulating various sources of uncertainty.
- Needleman-Wunsch (`nw`) A bioinformatics algorithm used to align protein and nucleotide sequences.
- Stochastic Different Equation Solver (`sde`) Numerical integration of stochastic differential equations.
- Speckle-Reducing Anisotropic Diffusion (`srad`) A diffusion algorithm based on partial differential equations that is used for removing the speckles in an image for ultrasonic and radar imaging.

Figure 6.2 characterizes the type and frequency of instructions in each benchmark, showing the percentage of FP arithmetic instructions, load/store instructions, address-generation

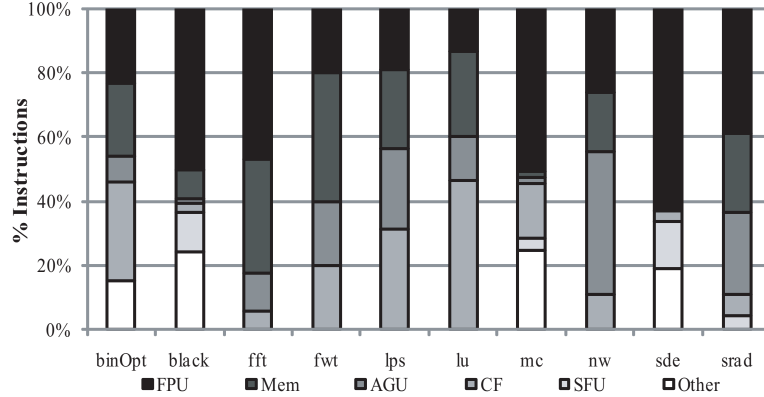


Figure 6.2: *Static instruction type breakdown showing the % of instructions used for floating-point operations (FPU), loads and stores (Mem), address generation (AGU), special math library functions (SFU), control-flow (CF) and other instructions such as integer math operations and loads from constant memory (Other).*

instructions, special-function floating-point library instructions (e.g. logarithms, trigonometry), control-flow instructions, respectively.

The computation in these benchmarks is predominantly FP arithmetic. Other than that, the instruction breakdown of these benchmarks varies widely. Some benchmarks which access a number of different arrays have a higher number of integer arithmetic operations in order to perform address calculations. While most control-flow instructions in these applications are used for checking loop terminating conditions, a few benchmarks are quit control-flow intensive. These applications are all comprised primarily of parallelizable loops that will run efficiently on GPU-style architectures.

6.2.2 GPU Utilization

We analyze the benchmarks' behavior on GPUs using the GPGPU-SIM simulator [6]. The configuration used closely matches the Nvidia FX5800 configuration used in [6] and provided by their simulator. Modifications were made to make the simulated designed very

Parameter	Value
Number of Shader Cores	30
Process clock frequency	1,476 MHz
Warp Size	32
SIMD Pipeline Width	8
Number of Threads/Core	1,024
Number of CTAs/Core	8
Max number of registers/core	16,384
Total memory bandwidth	159 GB/sec

Table 6.1: *GPGPU-SIM configuration, matching the Nvidia GTX 285 as closely as possible.*

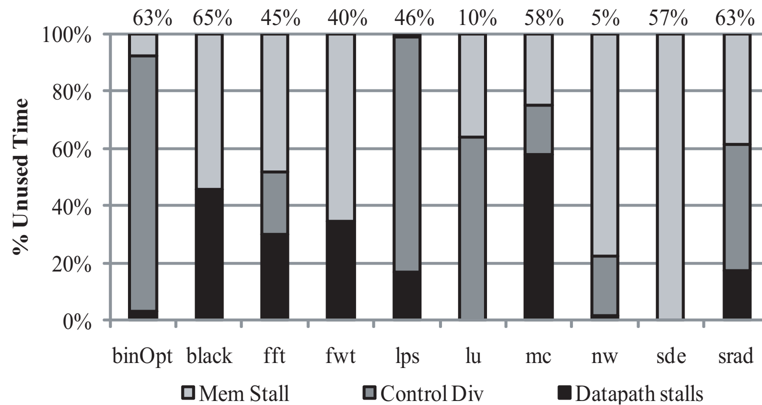


Figure 6.3: *Benchmark utilization on a GTX 285 model. Utilization is measured as a percentage of peak FLOPs and is indicated as a number above each bar. The components of the bar represent a different source of stall cycles in the GPU. The mean utilization is 45%.*

similar to the GTX 285 GPU, the most recent GPU that uses the FX5800's microarchitecture.

The configuration of the GPU is listed in Table 6.1.

Figure 6.3 illustrates the performance of each of our benchmarks and the sources of underutilization. "Utilization" here is the percentage of the theoretical peak performance of the simulated architecture actually achieved by each of these benchmarks. Idle times between kernel executions when data was being transferred between the GPU and CPU were not considered.

On average, these applications make use of around 45% of the general-purpose compute power of the GPU, with the extremes being BlackScholes, with a 65% utilization, and Needleman-Wunsch, with a 5% utilization. It is important to note here that the utilizations of individual benchmarks vary widely. Further, the extent to which specific causes lead to underutilization also varies from one application to another.

Figure 6.3 illustrates four principle reasons for under utilization of GPUs:

- Memory stalls: The benefit from having numerous thread contexts in GPUs is the ability to hide memory latency by issuing a different thread-warp from the same block of instructions when one warp is waiting on memory. This is not always enough, however, and this portion of the graph indicates the amount of time that all available warps are waiting for data from memory. To check whether it is memory bandwidth or memory latency that is the limiting factor, a separate study was done using a machine configuration that had double the bandwidth of the GTX 285. The change in utilization was negligible.

- Datapath stalls: This portion of the graph indicates the amount of time the shader core datapath itself is stalled for reasons such as read-after-write hazards. This is especially of concern on GPUs, which tend to have very deep floating-point pipelines.

- Serialization: The CUDA SIMT system collects 32 individual threads into a "warp" and executes them in a manner similar to a 32-wide SIMD machine. In sequential code, threads in a warp all execute the same series of instructions. However, in the event that some threads take a different execution path, the warp is split into the taken and not-taken portions and these two newly-formed warps are executed back-

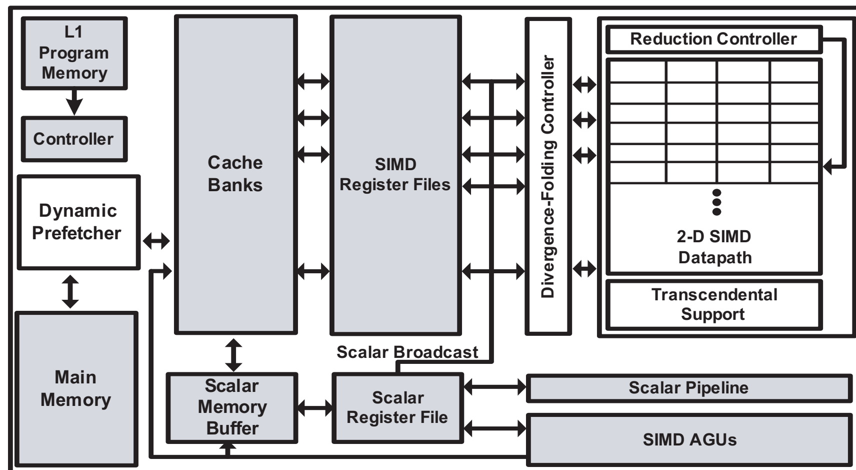


Figure 6.4: *PEPSC architecture template. The shaded components are part of conventional SIMD datapaths.*

to-back rather than concurrently, reducing the overall utilization of the processor. The "control divergence" portion of the graph indicates the amount of time that is spent executing fractions of a warp rather than an entire warp at a given time.

Although a newer generation of Nvidia GPUs than the GTX 285 has been released, the major changes made to design – such as allowing multiple kernels to execute concurrently – do not affect the observations made in this analysis.

6.3 The PEPSC Architecture

An overview of the PEPSC architecture is presented in Figure 6.4. It has the following architectural components to fulfill the basic computational requirements for data-parallel scientific applications:

1. A wide SIMD machine to effectively exploit data-level parallelism.

2. A scalar pipeline for non-SIMD operations such as non-kernel code and incrementing loop-induction variables.

3. A dedicated address generation unit (AGU).

4. Special function units for math library functions such as sine, cosine and divide.

PEPSC employs several additional techniques to improve the efficiency of scientific computing, each addressing a different source of the current reduced utilization. These are:

- A 2-dimensional design that extracts power efficiency from both the width and the depth of a SIMD datapath.
- Fine-grain control of the SIMD datapath to mitigate the cost of control divergence.
- A dynamically adjusting prefetcher to mitigate memory latency.
- An integrated reduction floating-point adder tree for fast, parallel accumulation with very low hardware overhead.

These features are explained in more detail in the following sections.

6.3.1 A Two-Dimensional SIMD Datapath

The first dimension of any SIMD datapath is the number of SIMD lanes. The optimal number of lanes in a domain-specific architecture is generally decided by the amount of data parallelism available in the domain. In massively-parallel scientific computing applications, however, the prevalence of "DOALL" loops allows for an effectively infinite amount of data parallelism. The deciding metric in such situations is the change in power efficiency of a SIMD datapath with increasing width due to factors such as control and memory divergence.

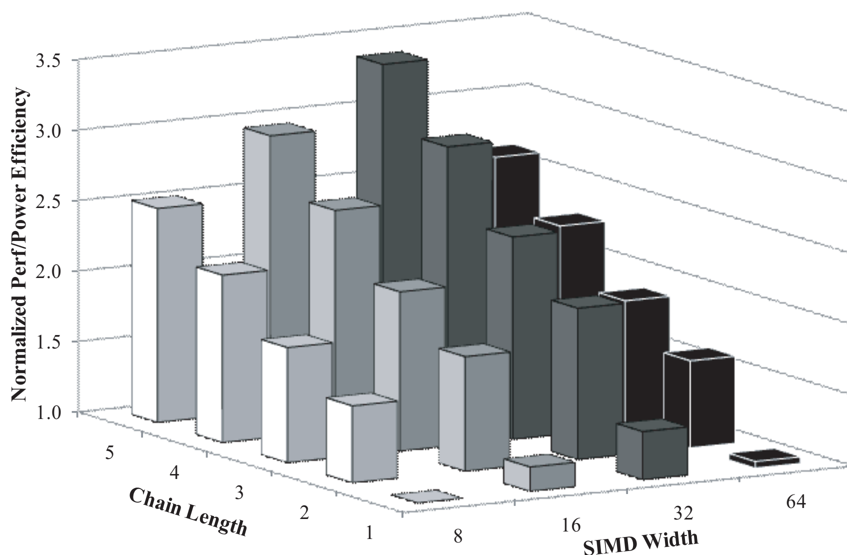


Figure 6.5: *Effect of length and width on the power efficiency of a 2D SIMD datapath, normalized to the 8-wide, 1-long case.*

The PEPSC datapath introduces a second dimension with a novel operation chaining technique. The basic concept of operation chaining to efficiently execute back-to-back, dependent operations dates back to the CRAY vector architectures. The architecture had separate pipelines for different opcodes allowing, for example, forwarding the result of an add instruction to an adjacent multiply unit to execute an add-multiply operation. In older, memory-to-memory datapaths with few registers, chaining was seen as an effective method to eliminate the need to write a value to a high-latency main memory only to read it back again.

The PEPSC architecture uses a *different* style of chaining. It allows for several back-to-back, dependent floating-point operations to be executed on a novel, deeply-pipelined fusion of multiple full-function FPUs. This results in performance improvement due to fewer read-after-write stalls and power savings from fewer accesses to the register file.

Benchmark	#FPU instrs	2-deep FPU	3-deep FPU		4-deep FPU			5-deep FPU			
		2-op	2-op	3-op	2-op	3-op	4-op	2-op	3-op	4-op	5-op
binOpt	3	1	1	0	1	0	0	1	0	0	0
black	37	13	6	4	5	1	3	5	1	1	2
fft	16	4	4	0	4	0	0	4	0	0	0
fwt	2	0	0	0	0	0	0	0	0	0	0
lps	6	2	1	1	0	0	1	0	0	0	1
lu	2	1	1	0	1	0	0	1	0	0	0
mc	27	10	5	3	4	1	2	5	0	0	2
nw	14	6	6	0	6	0	0	6	0	0	0
sde	43	13	9	4	3	3	3	3	3	3	0
srad	36	12	5	5	2	3	3	3	0	2	2

Table 6.2: *Scientific application FPU operation-depth characteristics.*

Figure 6.5 illustrates the power efficiency of varying SIMD widths between 8 lanes and 64 lanes when compounded on the power efficiency of 1- to 5-op FPU chaining. The efficiency is normalized to that of an 8-wide, 1-op FPU design, averaged across all benchmarks. These results indicate that a SIMD width of 32 lanes using a 5-op-deep chained FPU provides an optimal point, balancing the increasing efficiency of executing more operations per instruction with efficiency-reducing divergences. This 32x5 SIMD datapath has 3.4X the power efficiency of the 8x1 datapath.

The following sections explain the FPU microarchitecture in more detail.

6.3.1.1 Reducing FPU Latency

Scientific applications typically have chains of dependent FP operations longer than two operations. In the benchmarks studied in this work, nearly all have instances of 3 back-to-back floating point operations and some have instances of 5 back-to-back operations.

Table 6.2 illustrates the frequency of such chains. Each set of columns shows how many operations of a given length occur when hardware of a given length is provided. For

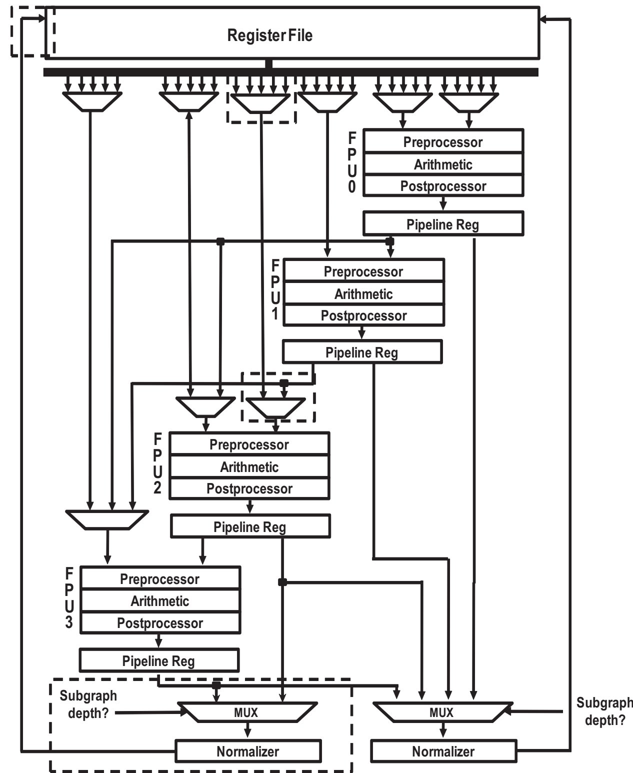


Figure 6.6: *Chained FPU datapath with support for multiple subgraph execution.*

example, in the benchmark `lps`, there is a sequence of 5 dependent FPU operations. When a 2-deep FPU is made available, these five FPU operations are executed as two successive 2-operation chains, followed by one 1-operation chain. If a 5-deep FPU is made available, these five FPU operations can be executed in a single 5-operation chain.

This work extends the generalized chains of FPUs introduced in Chapter V with "chained coalescence".

Using a 5-deep chained FP unit as suggested by Figure 6.5 will obviously result in under-utilization in some benchmarks. To combat this, a few hardware extensions to the chained FPU design allow for multiple independent 2- or 3-long FPU chains to execute in parallel on the FPU datapath. These extensions are indicated by dotted lines on Figure **??**. To allow for this, the register file has to have an additional read port and an additional write

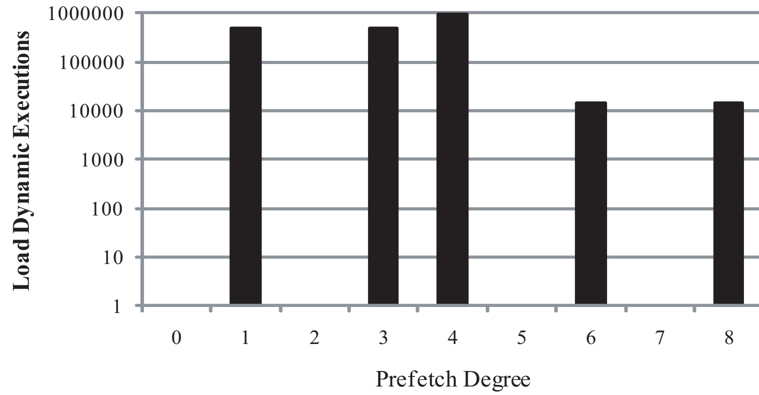
port. The FPU datapath will require an additional mux to allow for one of the FPUs to select between receiving its inputs from either a previous FPU or from the register file. A second normalizer stage will also have to be added since there will now be two two possible exit points from the chained datapath.

6.3.2 Reducing Memory Stalls

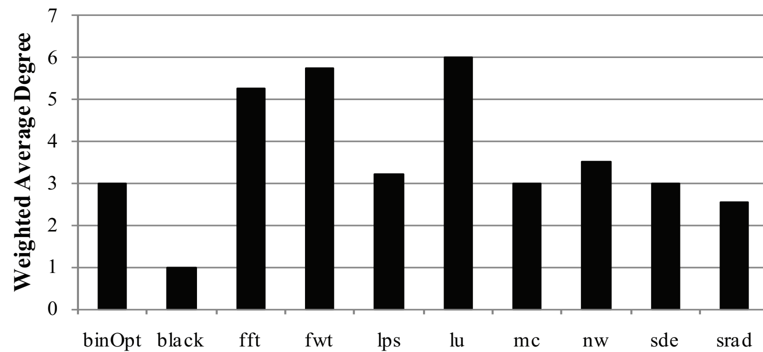
There are a few different alternatives when trying to mitigate problems with off-chip memory latency. Large caches offer a dense, lower-power alternative to register contexts, either to store the data required in future iterations of the program kernel or in order to cache infrequently-accessed register thread contexts. Even though modern GPUs have very large caches, these are often in the form of graphics-specific texture caches, and not easily used for other applications. Further, many scientific computing benchmarks access data in a streaming manner – values that are loaded are located in contiguous, or fixed-offset, memory locations and computed results are also stored in contiguous locations and are rarely ever re-used. This allows for creating a memory system that can easily predict what data is required when.

6.3.2.1 Stride Prefetcher

A conventional stride prefetcher [12, 36, 30, 91, 117] consists of the "prefetch table" – a table to store the miss address of a load instruction; the confidence of prefetching, the access stride. The PC of the load instruction is used as a unique identifier to index into the prefetch table.



(a)



(b)

Figure 6.7: *(a) Varying prefetcher degrees in the* `lps` *benchmark (b) Varying weighted prefetcher degrees in different benchmarks.*

6.3.2.2 Dynamic Degree Prefetcher

Stride prefetchers often have a notion of degree associated with them, indicating how early data should be prefetched. In cyclic code, it is the difference between the current loop iteration number and the iteration number for which data is being prefetched. A traditional stride prefetcher, as described in Section 6.3.2.1, uses a degree of one for all the entries in the prefetch table. With large loop bodies, degree-one prefetchers perform well as the time required for prefetching data is hidden by the time taken to execute one iteration of

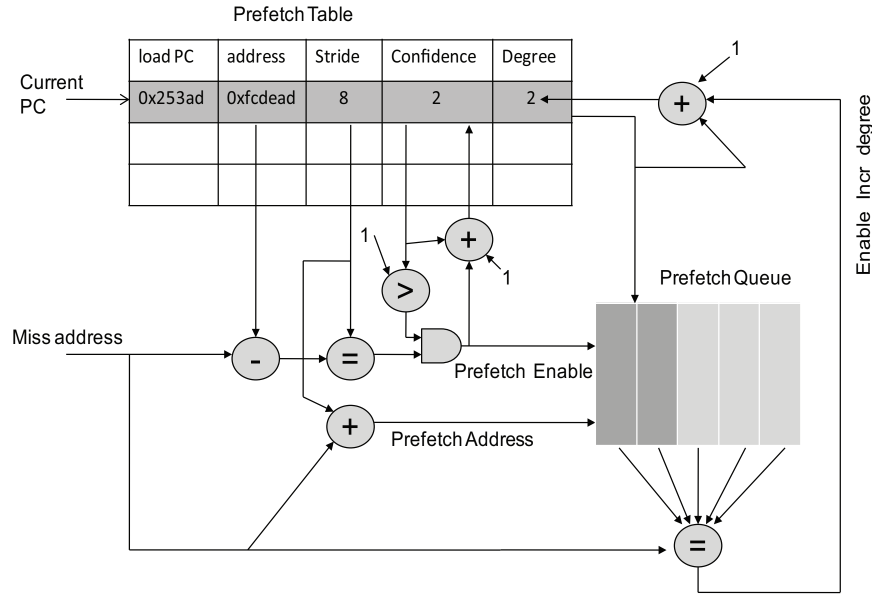


Figure 6.8: *Dynamic-degree prefetcher.*

the loop. However, if the time taken to execute a single iteration of the loop is less than the time required for the prefetcher to get the next working data, the processor will stall.

Figure 6.7(a) shows the number of times all the loads of a particular degree are executed in one of our benchmarks, `lps`. In this figure, the degree of prefetching in different loops varies between 1 and 8. An experiment was conducted to determine the variance of degrees across all benchmarks as shown in Figure 6.7(b). This figure demonstrates that the prefetch degree in scientific applications is a characteristic of the benchmark itself. From Figures 6.7(a) and 6.7(b), it can be concluded that there is enough variance in degree within and across benchmarks that would offset the advantages of presetting a fixed degree. Having a lower degree leads to a performance penalty as the processor would have to stall for data. But if the degree is too large, more data will be fetched than is necessary. This leads to an over-utilization of bandwidth as more data is fetched than necessary. Furthermore, this new data may evict useful data. All these factors warrant the requirement of a dynamic

solution. Loop-unrolling can decrease the degree of a load instruction and different amount of unrolling can be applied to the loops to change their degree to a fixed number. However, the performance benefits of this technique would vary between systems with different memory latencies.

This work presents a dynamic degree prefetcher (DDP) which varies the prefetch degree based on application behavior; higher degrees are assigned to cyclic code with shorter iteration lengths and lower degrees are assigned to code with longer iteration lengths. The computation of the degree is done at run-time, based on the miss patterns of the application. The DDP is illustrated in Figure 6.8. In the DDP, the initial degree of prefetching is set to 1. When an address is requested either by a load instruction or by a prefetch request, the prefetch queue is first checked to see if a prefetch request for data at that address has already been made. If so, it indicates that the prefetching for that load instruction is short-sighted. In response to this, the degree for the corresponding PC is incremented. By increasing the degree of the prefetcher, data is then prefetched from further ahead and, hence, by the time later iterations are executed, the required data will be present in the cache. As the degree is not fixed but dependent on the characteristics of the loop itself, different loops settle at different degrees.

6.3.3 Reducing Serialization

SIMD architectures are normally programmed by explicitly specifying SIMD operations in high-level languages like C or C++ using intrinsic operations. In order to specify that only specific lanes in a SIMD operation should commit, a programmer has to express this explicitly using SIMD mask registers. Many SIMD architectures employ a “write

mask" to specify whether or not a given lane commits its value. This write mask is specified as an additional operand to the source and destination data operands of an instruction. Destination registers for individual lanes are only over-written by the values computed in the corresponding lane if the write mask for that lane is a '1'. While this gives the programmer control over the exact implementation of the code, handling such a low-level issue can become tedious.

The CUDA environment circumvents the issue of divergence in SIMD architectures by abstracting out the underlying SIMD hardware and allowing a programmer to essentially write scalar code and have the hardware assemble threads of scalar code into SIMD code. However, this now means that the hardware has to handle all control divergences. Figure 6.9 shows an example of divergent code. Divergence in Nvidia shader cores is handled using a technique called "immediate post-dominator reconvergence", shown in Figure 6.9 which illustrates the way that post-dominator reconvergence works for a simple 4-wide SIMD [37]. Here, different lanes execute code on the "then" and "else" paths. The execution of the "then" and "else" basic-blocks is serialized, and no computation is done on the unused lanes for each path, as shown by the white arrows, reducing the overall utilization of the datapath.

The AMD/ATI shader core architecture differs from the Nvidia one in its ability to extract intra-thread instruction-level parallelism using 5-wide VLIW "thread processors". These thread processors are in-turn clustered into 16-wide SIMD arrays with each element executing the same VLIW instruction word at any given time so this configuration, too, is vulnerable to reduced efficiency from divergence.

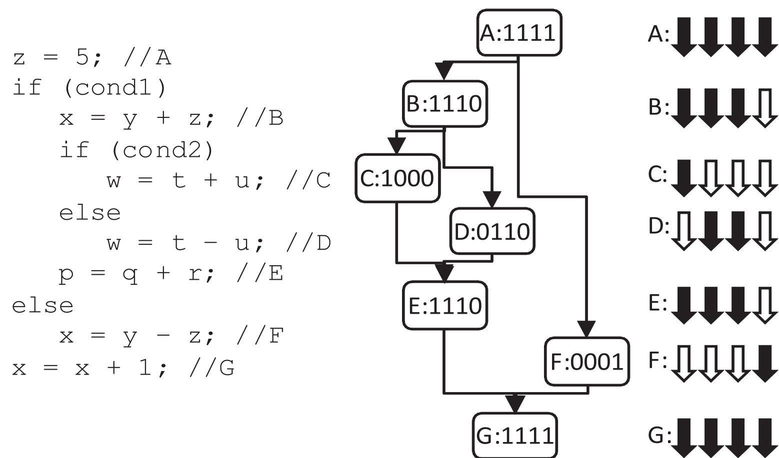


Figure 6.9: *An example diverging code fragment and the associated control-flow graph. The black and white arrows indicate which SIMD lanes are used and unused, respectively, when using an immediate-post-dominator-reconvergence strategy.*

Solutions such as dynamic warp formation (DWF) [37] address the general problem of control divergence in SIMD architectures by grouping together CUDA SIMT threads that execute the same side of a branch. Such techniques, while effective at reducing divergence, are often limited by other effects; migrated threads must still execute in the same SIMD lane from which they originate, and mixing-and-matching threads from different warps reduces memory coalescence, reducing the effectiveness of the memory system.

6.3.3.1 Divergence-Folding

All lanes in a non-multithreaded SIMD architecture like PEPSC must apply the traditional technique of SIMD predicate masks and execute both sides of a branch for every lane. The per-lane masks are then used to decide whether the "then" or the "else" path of the branch for a given lane is finally committed to the register file. The performance penalty incurred by executing both sides of a branch due to the SIMD nature of the architecture can be reduced by the effective mapping of instructions on PEPSC's chained datapath.

Operations with complementary predicates can be executed simultaneously if two FUs are available. The 2D SIMD datapath allows such execution as there are multiple FPUs in each lane. The modified, dual-output chained FPU has sufficient register inputs and outputs to execute and commit two FPU chains concurrently. Therefore, independent FP operations on opposite sides of a branch can also be executed concurrently. A few modifications are made to the infrastructure to support this. First, the compiler is modified to allow FPU operations guarded by complementary predicates to be executed on the FPU datapath at the same time. Second, the control logic for the output selector muxes of the FPU datapath is modified to select which of the two concurrently executing subgraphs should write their values to the register file based on a given SIMD lane's predicate mask bit. Since support for writing to two different registers already exists, this technique still works even if the two sides of a branch are writing to different registers.

There are a few caveats to this technique – the "then" and "else" paths should both be comprised of floating-point operations and the operation chain length should be limited to 2- or 3-long chains for these paths at most in order for both sides to fit on the FPU datapath.

Where the traditional predicate mask technique would require sequential execution of both sides of the branch, this divergence-folding technique exploits "branch-level" parallelism and executes both sides of the branch in parallel.

6.3.3.2 Reduction Tree

Another source of serialization are "reduction" operations such as an accumulation of all the values computed so far. Since such operations only involve addition to a single value, it is inherently serial and, as such, not SIMD-izable via a traditional SIMD machine.

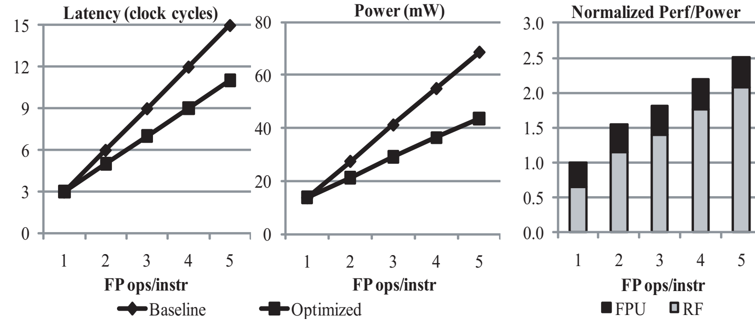
In order to minimize the amount of time spent performing these serial reduction operations, an "adder tree" can be used to sum up all the values in a SIMD register in logarithmic time. For a 32-wide SIMD machine, this requires the use of a total of 31 floating-point adders, arranged 5 adders deep.

Normally, the hardware cost of such an addition would be quite substantial – on the order of doubling the number of FUs. However, due to the FPU chaining techniques presented in Section 6.3.1.1, this hardware cost can be reduced. Essentially, adding some interconnect logic to half of the two-dimensional datapath creates an FPU tree. For a SIMD width of n, the first level of the FPU chain (FPU0 in Figure **??**) sums together $\frac{n}{2}$ pairs of adjacent elements, the second level (FPU1) sums together the resulting $\frac{n}{4}$ pairs, etc. until the last element adds one pair. For a 5-deep FPU chain, this technique can only be used to add together 2^5, or 32, values.

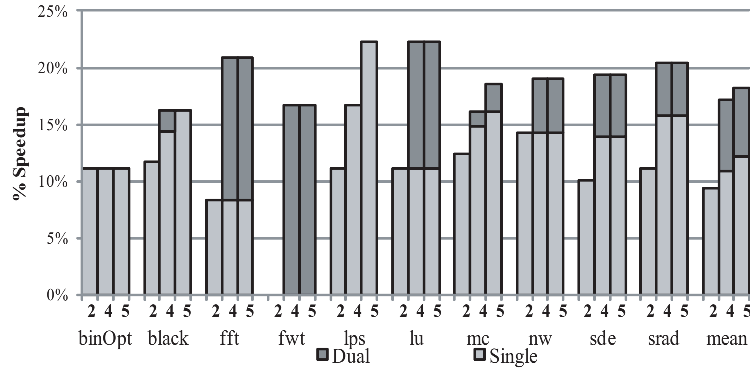
6.4 Results

The major components of PEPSC were designed in Verilog and synthesized using the Synopsys Design Compiler and Physical Compiler tools. Power results were obtained via VCS and Primetime-PX, assuming 100% utilization. Power characteristics for regular memory structures like dual-ported RFs and caches were obtained through an Artisan memory compiler while RFs with more than 2 read ports were designed in Verilog and synthesized.

The M5 simulator system was used to study the cache design and the DDP. The main memory latency was set at 200 cycles. The L2 size was 4kB per SIMD lane with 4-way



(a)



(b)

Figure 6.10: *(a) Datapath latency, power and area effects of varying the number of FPUs when increasing FPU chain length. (b) Speedup with increasing chain length. The chain lengths are indicated in bold numbers below each bar; the dark portion above the "4" and "5" bars indicate the additional performance improvement from allowing multiple subgraphs to execute concurrently.*

associativity and a delay of 20 cycles. The L1 size was 512B per SIMD lane with 4-way associativity and a delay of 1 cycle.

Sections 6.4.1, 6.4.2 and 6.4.3 present speedup information for the specific problem solved in isolation. Combined results showing the aggregate affect of all the modifications are presented in Sections 6.4.4 and 6.4.5.

6.4.1 Datapath optimizations

Figure 6.5 indicates that a number of the applications in this domain have several long sequences of back-to-back FP operations. Based on this data, the FP datapath in PEPSC

was designed with an FPU consisting of 5 back-to-back operations. Figure 6.10 shows the effects of varying the number of operations in the FPU.

In Figure 6.10(a), the x-axis for all the graphs, “FP ops/instruction” is the number of successive, dependent FP operations executed in the chained FPU. The “latency” graph shows the time (in clock cycles) taken to execute an input subgraph. The baseline 3-cycle FPU takes 3 cycles for each operation and thus has a latency that increases by 3 cycles for every added operation. The removal of redundant hardware in the optimized FPU chain results in significantly less overall latency – a savings of 4 cycles when 5 FPUs are connected back-to-back.

The “power” graph in Figure 6.10(a) illustrates the power savings obtained from optimized normalizing. Here, the baseline is multiple un-modified FPUs executing operations back-to-back. In this graph, too, the gap between optimized and unoptimized FPUs widens quite dramatically as the number of operations per instruction increases. The power measurement in this graph is the sum of the RF access power and the FPU execution power to better reflect the power penalty from increasing the number of RF read-ports.

The “normalized perf/power” graph in Figure 6.10(a) addresses the efficiency of the different solutions. Here, too, the power consumed for the amount of work done steadily reduces as the number of FP operations per instruction increases. While the access power for an RF increases for every added read port, the improvement in the efficiency of the RF shown in the graph indicates that this is amortized by the reduction in overall accesses and by the performance improvement achieved by chaining together FPUs.

Figure 6.10(b) shows the reduction of stall cycles observed for the different benchmarks when using 2-, 4-, and 5-operation chained FPU datapaths. The speedup varies based on

how frequently the various chained FP operations occur in each benchmark and the latency penalty incurred when issuing back-to-back dependent FP operations. The benchmarks that had the most to gain from chaining FPUs were, in general, the ones with the most number of 5-long chains of FP operations – `black`, and `lps`, for example. The `fwt` benchmark had no operation patterns that could be accelerated using the chained FPU and, therefore, had no performance improvement. The "dual" bars indicate the additional performance improvement from allowing multiple independent subgraphs from executing on the FPU datapath. Significant performance improvement is observed in the majority of benchmarks, the most notable being `fwt` which previously could not exploit the improved datapath design. On average, the speedup for a 5-long chain increased from 12% to 18%.

Speedup saturates at 5 operations and adding a sixth operation in the chain only reduces the overall utilization of the FPU. Using a 5-op chain is, therefore, the best solution, from both a performance and an efficiency perspective.

6.4.2 Memory System

To ascertain the effectiveness of our prefetching techniques, we measured the performance of a baseline no-prefetcher system, the performance with an added degree-1 prefetcher and the performance with the DDP. These results are shown in Figure 6.11. The baseline degree-1 provides, on average, a speedup of 2.3X, resultant from a 62.2% reduction of average D-cache miss latency. The more sophisticated and adaptive dynamic degree prefetcher shown in Figure 6.8 provides a speedup of 3.4X on average, due to an additional 53.1% reduction in average D-cache miss latency over the degree-1 prefetcher. This amounts to a speedup of 1.5X speedup over the traditional degree-1 prefetcher. The

`srad` benchmark presents a pathological case; the main inner-loop has 5 separate, equally-spaced loads from the same array, leading to higher-than-expected conflict and pollution in one particular set in our 4-way associative cache. To confirm this, we repeated the experiment but with an 8kB, 8-way associative L2; for this configuration, the performance of the DDP was better than that of the degree-1 prefetcher.

The memory system used for the results in Figure 6.11 prefetched data from main memory into the L2 cache, but not from the L2 cache into the L1 cache. The effect of adding an additional prefetcher from the L2 cache to the L1 cache was studied and the results are presented in Figure 6.12. While there was a performance improvement in some benchmarks by the addition of a prefetcher from the L2 cache to the L1 cache, this is offset by the performance loss observed in others, resulting in a negligible average performance difference between the two cases. The performance reduction observed from adding the L2-to-L1 prefetcher was primarily caused by the pollution of the L1 cache by data prefetched too soon, resulting in the eviction of needed data such as local variables on the stack.

A common concern when using prefetchers is the transfer of unnecessary data. The total amount of data transferred from main memory to L2 was analyzed for the no-prefetcher, degree-1 and dynamic degree prefetchers. The degree-1 prefetcher transferred, on average, only 0.8% more data than the no-prefetcher case, and DDP transferred 1.3% more than the no-prefetcher case.

6.4.3 Serialization

Four of the benchmarks studied demonstrated speedup using the control-flow techniques discussed in Section 6.3.3: `lps`, `lu`, `mc` and `srad`. The speedups obtainable in

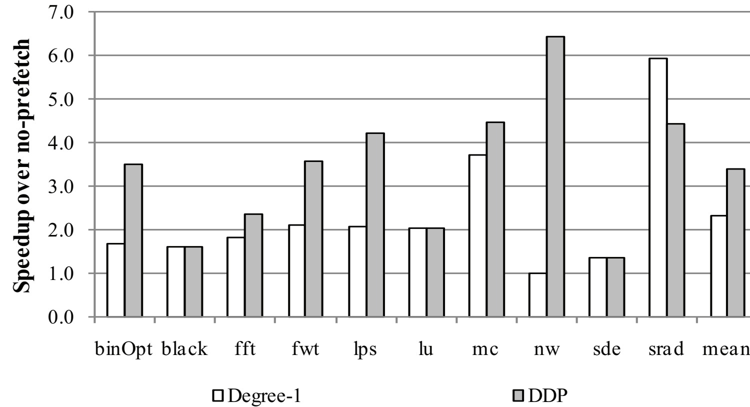


Figure 6.11: *Comparison of DDP and degree-1 prefetching.*

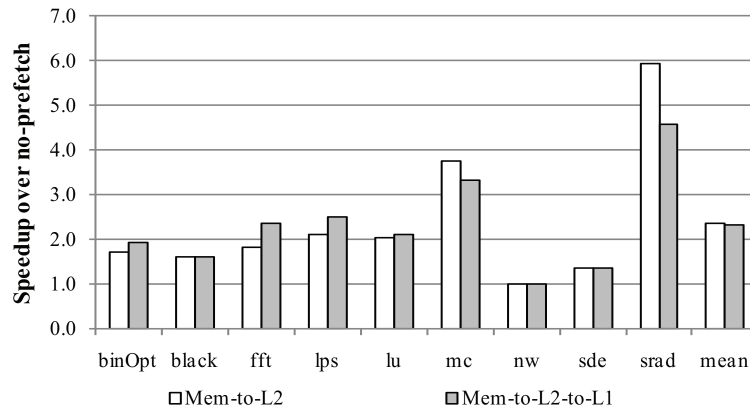


Figure 6.12: *Comparison of prefetching from mem to L2 and mem to L2 to L1.*

these benchmarks is shown in Figure 6.13. The source of the improvement between them varies.

The benchmark `lps`, `lu` and `srad` benefit primarily from the use of the divergence-folding technique. All these benchmarks are control-flow intensive, and have predominantly symmetrical control-flow graphs, performing similar, short computation on either side of a divergence – the perfect use-case for the technique. These benchmarks saw reductions of 38%, 30% and 50% in control-flow overhead, respectively.

The `mc` benchmark employs reduction operations at the end of its loops which are not amenable to SIMD-ization. Making use of the reduction adder tree proposed in Sec-

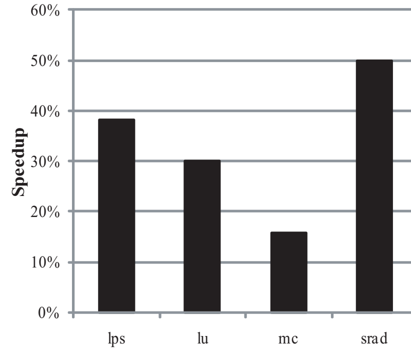


Figure 6.13: *Speedup using various serialization-mitigation techniques.*

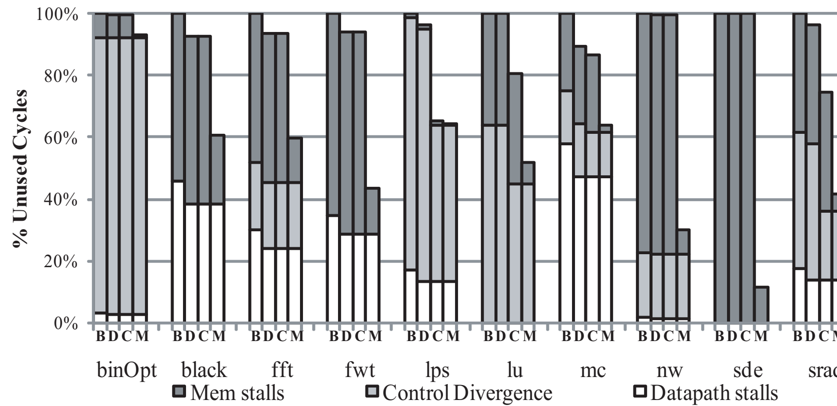


Figure 6.14: *Reduction in GPU overheads after cumulatively applying various techniques. "B" is the baseline bar, "D" is after adding the chained FPU datapath, "C" is after adding divergence-folding to mitigate control overhead and "M" is after adding the prefetcher to reduce memory latency.*

tion 6.3.3.2 rather than serially summing values provides a 16% reduction of control-flow overhead.

6.4.4 Application to GPU

Rather than creating an entirely new architecture, another option would be to augment existing GPUs with the hardware introduced in this work. Our estimates for the impact this would have are illustrated in Figure 6.14. The first bar for each benchmark is the baseline underutilization breakdown from Figure 6.3 and each subsequent bar is the revised breakdown with cumulatively adding the FPU chaining, divergence folding and a reduction

Frequency	750 MHz
SIMD Lanes	32
Peak Performance	120 GFLOPs
Peak Total Power	2.10W
Efficiency	56.9 Mops/mW
Technology Node	45nm

Component	**Power**
5-op 32-bit FPU with reduction + divergence-folding	1.18W (~37mW/ln)
16-element 32-bit RF	9.28 mW/ln.
Memory system	580mW
Etc. datapath (scalar pipe, AGU, control)	59 mW

Figure 6.15: *Overall per-core specifications and power breakdown of individual components.*

tree, and the dynamic-degree prefetcher. This average utilization increases to around 73% – a significant improvement over the 45% utilization baseline.

6.4.5 PEPSC Specifications

The PEPSC processor's design characteristics and power consumption breakdown are shown in Figure 6.15. The top table in Figure 6.15 shows the specifications of each individual core. The efficiency of 56.9 Mops/mW is approximately 10X that of the Nvidia GTX 295. The bottom table in Figure 6.15 shows a component-by-component breakdown of the power consumed in the PEPSC processor. The FPU power includes the power consumed by the additional control logic required by the op-selection scheme presented in Section 6.3.3.1 and also the extra routing and FP adder required by the reduction tree presented in Section 6.3.3.2. Since there are components to this FPU datapath that go across lanes, the per-lane power consumption number is an approximation. The "Memory system"

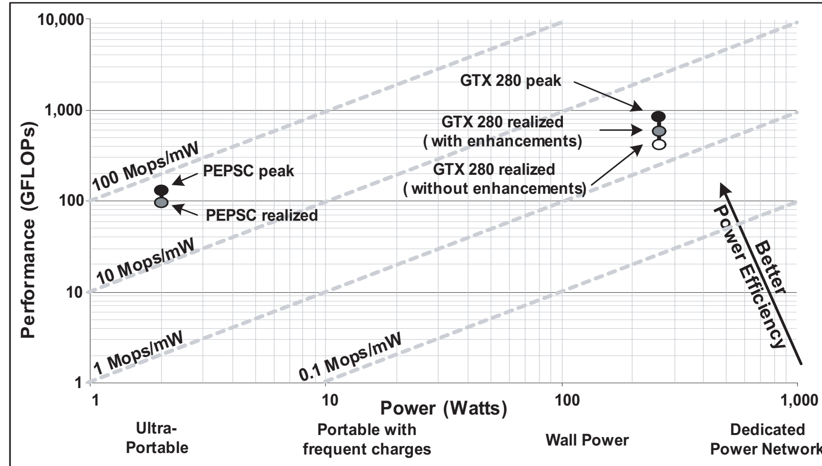


Figure 6.16: *(Modified Fig. 6.1) Power-efficiency of PEPSC. Black points are peak performances, and the gray and white points represent different points of underutilization.*

power includes the power consumed by the two levels of caches and the dynamic-degree prefetching engine.

Figure 6.16 is a modified version of Figure 6.1. It illustrates the power consumption and performance of PEPSC relative to other current designs. The white dot under the GTX 280 dot is the average realized performance for the baseline design. The gray dots below the PEPSC and GTX 280 dots indicated the average realized performance, as opposed to the peak performance.

For both peak and average use-cases, the PEPSC design is over 10X as power-efficient as modern GPUs.

6.5 Related Work

There are several other examples of low-power, high-throughput SIMD architectures like SODA [110] for signal-processing but it, being a purely integer architecture, is unsuitable for this domain space. VTA [51] has a control processor with a vector of virtual

processors, which are far more complex and power-consuming than the SIMD lanes of PEPSC. Thera are also high-throughput floating point architectures such as Merrimac [22], TRIPS [81] and RAW [95] but these are more focused on general-purpose computing and do not have the same power efficiency as PEPSC. The streaming buffers in Merrimac are a potential alternative to the memory system employed in this paper, but is less general than using a cache hierarchy. There has also been recent work on image-processing [41] and physics simulations [113], targeting domains traditionally addressed by commercial GPUs but these, too, do not address power to the extent that this work does. Parallax [113] is a heterogenous domain-specific architecture for real-time physics with coarse-grain and fine-grain cores coupled together to distribute the workload to achieve high performance.

Some CRAY systems use a "chained FPU" design. However, these are essentially just forwarding paths across different FUs. While this connectivity reduces register accesses, the FPU itself was not redesigned the way the PEPSC FPU is. In [114], the authors create a compound functional unit used to accelerate a variety of different applications but do not optimize the functional unit by removing overlapping hardware the way that this work's FPU chaining implementation does; further, it provides a solution that is more amenable for use as a co-processor rather than a within-datapath FU.

An alternative technique to mitigate the serialization problem of modern GPUs was presented in [37]. This "dynamic warp formation" technique collects warps of SIMT threads in Nvidia GPUs that branch in the same direction. This technique is quite effective but is limited in that the relative position of a thread within a warp can never change in order to maintain program correctness.

Hardware based stride prefetchers have been in existence for some time [12, 36]. Even though the current trend in data prefetching [30] is to look for correlation in data accesses, we feel that a simple strategy is more effective for scientific applications which have more regular access patterns. Variation of degree has been studied by [91]. They take periodic samples of various parameters of prefetching and change the aggressiveness of prefetching at the end of every sampling interval. Furthermore, they change the aggressiveness of the prefetcher in quanta, rather than reaching a general sweet spot. Increasing degree one at a time is more effective for scientific applications as the application execution times are quite high and the initial delay in ramping up the stride degree is only a small factor. In [91] they fix the aggressiveness of the prefetcher for a region which is effective for the specINT benchmarks, whereas in the benchmarks that we have used, precise prefetching on a per-stream basis is possible because of the regularity in memory accesses. TAS [117] also increments the degree of prefetching but in that work interval state is computed from global time and on its basis the degree is increased in discrete preset quanta.

6.6 Conclusion

The PEPSC architecture – a power-efficient architecture for scientific computing – is presented in this work. When running modern scientific, throughput-intensive applications, it demonstrates a significant improvement in performance/power efficiency over existing off-the-shelf processors. This architecture was designed by addressing specific sources of inefficiencies in the current state-of-the-art processors. Architectural improvements include an efficient chained-operation datapath to reduce register file accesses and computation

latency, an intelligent data prefetching mechanism to mitigate memory access penalties, and a finely controllable SIMD datapath to exploit data-level parallelism while mitigating any control divergence penalty. The PEPSC processor provides a performance/power efficiency improvement of more than 10X over modern GPUs. While a single domain-specific design is presented here, many of the architectural techniques introduced are general enough to be deployed on current designs.

CHAPTER VII

Data-Parallel Accelerators to Extract Instruction-Level Parallelism

7.1 Introduction

In the coming years, the deployment of mobile computers will continue to skyrocket. The prime example today is the smart-phone, but in the near future we expect to see the emergence of new classes of such devices. These devices will improve on the smart-phone by incorporating advanced functionality such as high-bandwidth Internet access, human-centric interfaces with voice recognition, high-definition video coding, and interactive conferencing. While integrating new capabilities is important for attracting customers, battery lifetime and power dissipation remains paramount.

Untethered devices perform signal processing as one of their primary computational activities due to their heavy usage of wireless communication as well as rendering of audio and video signals. Fourth generation wireless technology (4G) has been proposed by the International Telecommunications Union to increase the bandwidth to maximum data rates of 100 Mbps for high mobility situations and 1 Gbps for stationary and low mobility

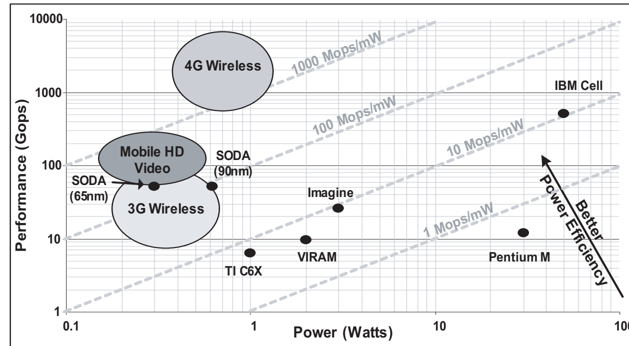


Figure 7.1: *Performance vs. power requirements for various mobile computing applications.*

scenarios like Internet hot spots [77]. This translates to an increase in the computational requirements of 10-1000x over previous third generation wireless technologies (3G), with a power envelope that is limited to increasing only 2-5x [109].

Figure 7.1 presents the demands of 3G and 4G computing in terms of their peak processing throughput requirements and their power budgets. Conventional processors cannot meet these requirements; low-power laptop processors, such as the Pentium M, operate below 1 Mop/mW, while digital signal processors, such as the TI C6x, operate around 10 Mops/mW. Conversely, 3G wireless protocols, such as W-CDMA and 802.11a, require approximately 100 Mops/mW. The IBM Cell system can provide excellent throughput, but its power consumption makes it infeasible for mobile devices [73]. Research solutions, such as VIRAM [50] and Imagine [1], can achieve the performance requirements for 3G, but exceed the power budgets of mobile terminals. SODA improves upon these solutions and delivers a solution for 3G wireless [54]. Companies such as Phillips [106], Infineon [8], ARM [111], and Sandbridge [38] also propose domain-specific systems that can support 3G.

The design of the next generation mobile platforms must address three critical issues: efficiency, programmability and adaptivity. The inherent computational efficiency of 3G

solutions is insufficient and must be increased by at least an order of magnitude as shown in Figure 7.1. Straight-forward scaling of 3G solutions by techniques such as increasing the number of cores is part of the solution, but is not enough on its own. Programmability provides the opportunity for a single platform to support multiple applications and even multiple standards within each application domain. Further, programmability provides faster time to market as hardware and software development can proceed in parallel, the ability to fix bugs and add features after manufacturing, and higher chip volumes as a single platform can support a family of mobile devices. Lastly, hardware adaptivity is necessary to maintain efficiency as the core computational characteristics of the applications change. 3G solutions rely heavily on the vast amounts of vector parallelism in wireless signal processing algorithms, but lose most of their efficiency when vector parallelism is unavailable or constrained.

The AnySP architecture tackles the first two issues using a combination of wide-SIMD execution (64 lanes), efficient data shuffling, and support for common intrinsic functions [112]. Efficiency and programmability are simultaneously garnered by pushing SIMD execution to new levels, while applying domain specific customizations to the datapath. Adaptivity within vectorizable code is also supported by allowing neighboring lanes to conjoin, creating 32 lanes, each supporting a computation depth of two, or overlaying multiple narrow vector computation threads using independent address generation units. Researchers at ST Microelectronics apply similar generalizations to SIMD datapaths [72]. However, all of these solutions ignore non-SIMD-izable computation. Such code is relegated to execute on a low performance scalar pipeline provided in the design. For inherently scalar applications such as compression or encryption, high computational rates cannot be sustained. Even for

highly vectorizable codes, Amdahl's Law will expose the non-vectorizable portions of the code as the eventual bottleneck.

One approach to accelerating the scalar code is to enhance the capabilities of the scalar pipeline. Integrating specialized functional units which more efficiently execute critical portions of an application's dataflow graph with instruction set extensions to utilize the new hardware is a popular approach for designing application specific instruction processors, or ASIPs. Several commercial tool chains design ASIPs with instruction set extensions, including ARM OptimoDE, and ARC Architect. However, this approach fails to take advantage of the vast hardware resources already present in the datapath, namely the SIMD execution units, which are mostly idle when scalar code is executing.

In this paper, we propose a dynamically changeable SIMD datapath, referred to as *SIMD-Morph*, that in the base mode can execute data-parallel code across all the SIMD lanes. However, for scalar code, the datapath is morphed into a subgraph accelerator similar to the configurable compute accelerator (CCA) [15]. A CCA consists of an array of simple functional units interconnected in a feed-forward manner. The CCA executes dataflow subgraphs identified by the compiler as atomic units. Acceleration is provided in two ways. First, instruction-level parallelism is exploited by concurrently executing independent operations in the subgraphs (horizontal compression). Second, operation chaining executes dependent operations in a single cycle by exploiting slack in the pipeline and eliminating conventional register forwarding (vertical compression).

The challenges of SIMD-Morph that lead to the central contributions of this work are as follows:

```
1:  for( i = 0; i < length; i++ )
2:  {
3:      x = (unsigned char) ( x + 1 );
4:      a = m[x];
5:      y = (unsigned char) ( y + a );
6:      m[x] = b = m[y];
7:      m[y] = a;
8:      data[i] ^=
9:        m[(unsigned char) ( a + b )];
10:  }
```

Figure 7.2: *A portion of the* `rc4` *benchmark.*

- The extensions to the SIMD datapath to enable effective execution of a scalar subgraph where both horizontal and vertical compression are achieved.
- A design space exploration to define the organization, needed capabilities, and connectivity of the subgraph accelerator to maximize utilization of the available functional resources in the SIMD datapath.

7.2 Motivation

As embedded devices become more and more pervasive, the types of computation they are required to perform becomes more varied. SIMD-ization is one of the most common architectural techniques used to improve the efficiency, but it is not effective for many styles of applications, e.g., some types of data encryption and compression, and many forms of error detection and correction.

```
1:  for (j = 0;  j < data_blk_size;  j++)
2:  {
3:    a = crc_accum >> 24;
4:    b = *data_blk_ptr;
5:    c = a^b;
6:    i = c&0xff;
7:    d = crc_table[i];
8:    e = crc_accum << 8;
9:    crc_accum = e ^ d;
10:   data_blk_ptr++;
11: }
```

Figure 7.3: *A portion of the* `crc` *benchmark.*

To give a specific example, Figure 7.2 is the critical loop for the RC4 encryption algorithm, the basis of Secure Sockets Layer (SSL) and many other popular streaming, secure protocols.

Note that while the data being encrypted is accessed in a sequential order (line 8), the value being XORed with the data is the result of multiple indirect memory accesses to `m[y]`. This type of pointer-chasing is difficult to perform efficiently on SIMD accelerators, which are typically designed for contiguous accesses.

As another example, Figure 7.3 is the main loop from a cyclic redundancy check (CRC) algorithm, commonly used to detect errors in signal transmissions.

Similar to RC4, CRC has non-contiguous, pointer-chasing memory accesses (line 7). An additional challenge is that CRC also has a dependence, `crc_accum`, that crosses loop iterations thus preventing any significant form of data-level parallelism. Like RC4, CRC would garner little if any benefit from traditional SIMD accelerators. Both of these applications are very important to many embedded domains, and augmenting a SIMD datapath to better support them would prove very beneficial.

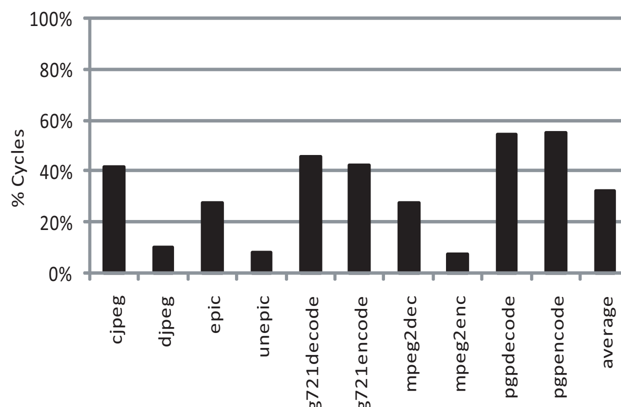


Figure 7.4: *Fraction of cycles spent outside of inner-most loops.*

Even in applications that are amenable to SIMD acceleration, often times only their inner-most loops are data-parallel and can be SIMD-ized. SIMD datapaths are continually getting wider with Moore's law [16] but as this happens, the non-SIMD-ized portions of the application will begin to dominate execution time as per Amdahl's law.

Figure 7.4 illustrates the importance of this trend. This figure shows the percentage of time spent outside inner-most loops, i.e. executing non-SIMD-izable code for several SIMD-izable benchmarks from the MediaBench Suite [52]. From this figure we can deduce that even in an ideal system with an infinitely-wide SIMD machine, over 30% of the application's execution time is spent on non-SIMD-ized code. This limits the speedup of these applications to $\approx$3x in the best case, and clearly represents an important target for acceleration.

Several important application domains are not amenable to SIMD-ization, and even those that are have a significant fraction of non-SIMD-izable code that is important to accelerate. For these reasons, this paper answers the question, “How can the standard narrow-

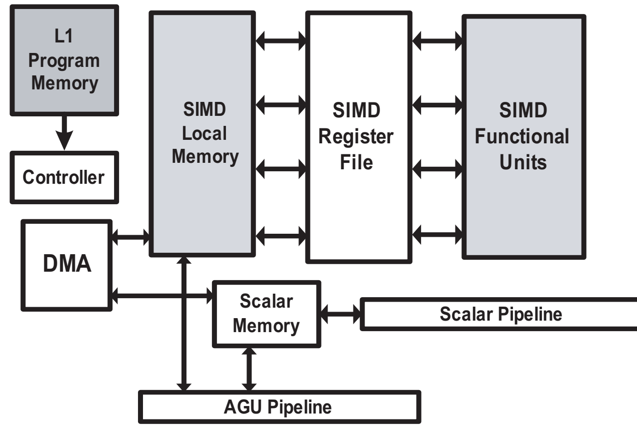


Figure 7.5: *Baseline SIMD+Scalar Processor.*

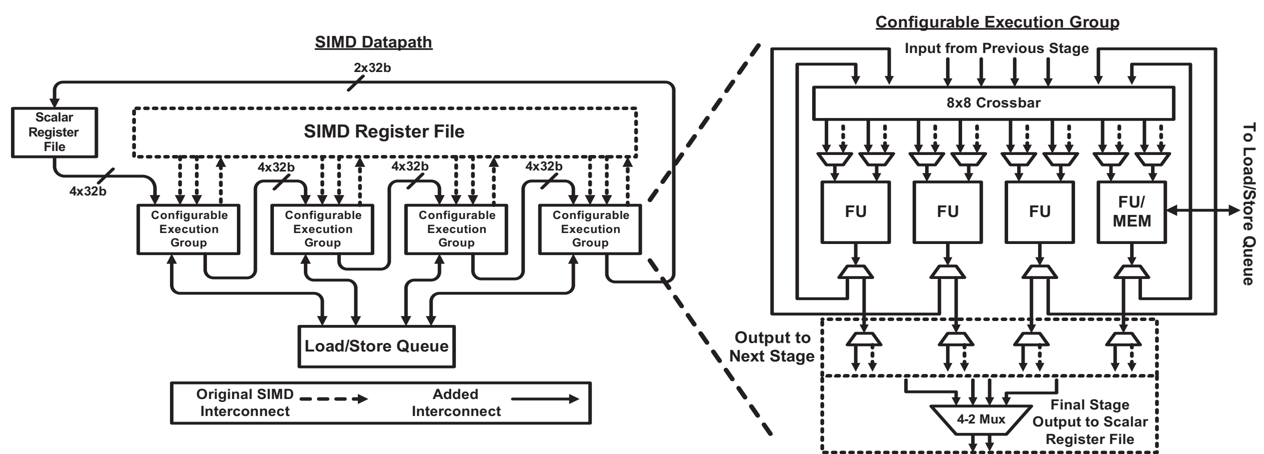


Figure 7.6: *SIMD-Morph Modifications.*

scalar-pipeline-with-wide-SIMD-pipeline architectures be modified to better accelerate a wider variety of applications?”

7.3 SIMD-Morph

7.3.1 Hardware

The baseline SIMD architecture used is shown in Figure 7.5. This baseline is modified in the manner shown in Figure 7.6 to create the SIMD-Morph architecture. The 16 SIMD lanes are grouped in 4 Configurable Execution Groups (CEGs) of 4 elements each, named

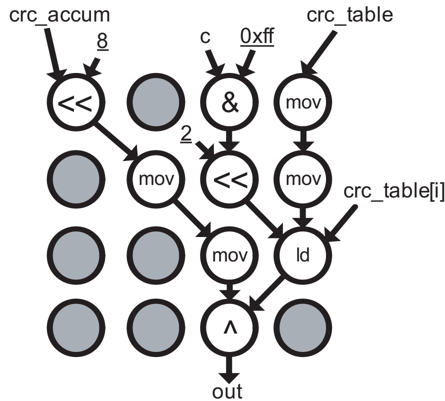


Figure 7.7: *Graphical representation of a portion of the* `crc` *benchmark on SIMD-Morph.*

CEG0 (lanes 0 to 3) through CEG3 (lanes 12 to 15). The FUs in all the lanes are capable of executing the same operations as before but now each CEG has an added memory unit capable of executing scalar loads and stores. Each FU may receive its inputs from 8 possible sources: the 4 outputs of the previous CEG, any of the other 3 FUs in its row or from an 8-bit constant register (not pictured). In addition, the memory FUs may receive loaded data from memory.

Register data is transferred into the SIMD datapath via CEG0's 4 inputs directly from the scalar register file. Register data is transferred out of the SIMD datapath via CEG3's outputs. Despite the increased complexity of adding connections from the scalar register file to the SIMD datapath, this method is a better solution than adding extra ports to the SIMD register file. This is because the code that will require this functionality is not SIMD-parallel and would normally be executed on the scalar pipeline so adding these connections to the SIMD register file will require extra cycles to copy data between the two register files, thereby reducing performance.

Figure 7.7 shows a representation of lines 6-9 of the `crc` benchmark shown in Figure 7.3 on the SIMD-Morph hardware. The live-in values in this case are the variables

`crc_accum`, `c` and the base pointer `crc_table`. This pointer is used to issue the load from `crc_table[i]` (line 7 in the code). The shaded nodes are idle and are not used for any computation. The live-out value is the updated value of `crc_accum`. The `mov` operations are used to transfer operands from where they are generated to where they are read.

7.3.2 Configuration

A control memory is required in order to store the various configuration bits required for SIMD-Morph. The requirements of the different components are broken down as follows:

- To specify opcodes, each element requires 5 bits to support the various arithmetic and logic operations. The memory units require an additional bit to support various load and store instructions. This is a total of 21 bits per CEG, or 84 bits total.

- Each FU has 2 ports, each of which can receive inputs from 1 of 7 possible sources (requiring a 3-bit mux per port). Further, 1 global bit is required to specify whether the FU receives its inputs in "SIMD-Morph mode" (from other FUs and the scalar register file) or in "normal mode" (from the SIMD register file). Each FU can also receive an 8-bit literal value as input. The total bits required to specify inputs is, therefore 225 bits (1 + 8*16 + 3*2*16).

- The very last row outputs back to the 2 write ports in the scalar register file. Each of these ports may receive its inputs from any of the 4 elements in CEG3, requiring 4 bits of control.

The total number of bits required to configure SIMD-Morph is, therefore, 225 + 84 + 4, or 313 bits.

7.3.3 Compilation

Effectively utilizing accelerators such as SIMD-Morph requires tool chain support, and so it is important to introduce the compilation strategy used during design space exploration. Compiling an application to make use of computation accelerators boils down to two steps: *enumerating* portions of the application's dataflow graph (DFG) that can be executed on the accelerator, and *selecting* which portions to accelerate.

"Enumeration" consists of generating a set of subgraphs from a given DFG, and determining if they can run on an accelerator. Generating a set of subgraphs is difficult because the number of possible subgraphs grows exponentially with the size of the DFG. Determining if the subgraphs can run on an accelerator, i.e., determining if they perform the same computation, is essentially equivalence checking, which is NP-complete. The problem is further complicated if the accelerators perform a superset of the desired computation (e.g., an accelerator for dot-products could also accelerate multiply-accumulates in an application).

"Selecting" which subgraphs to accelerate is also difficult. Typically, the selection problem is formulated to push as much computation as possible onto the accelerators, while ensuring that there is no overlap between subgraphs. That is, given a set of enumerated subgraphs, find the group that covers the largest portion of the DFG while minimizing the number of nodes appearing in multiple subgraphs. This problem is also NP-complete and

is quite similar to the well known technology mapping problem in VLSI design. Clearly, mapping applications to subgraphs is a challenging compilation problem.

Previous work has shown that greedy solutions work poorly, particularly when the accelerator is large, like SIMD-Morph is [18]. For that reason, we leveraged a more thorough compilation approach very similar to previous work [18]. Essentially, the compiler performs an exhaustive search of the design space to enumerate and select the best possible set of subgraphs for acceleration. Several pruning heuristics keep the compilation time reasonable for the vast majority of cases, and timeouts prevent corner cases from taking an intractable amount of time. This more thorough compilation strategy ensures that the design space exploration is as accurate as possible.

7.3.4 Baseline Observations

In this work, benchmarks are generally classified into two categories. The "media" benchmarks are the same as those in Figure 7.4 and are from the Mediabench benchmark suite. The rest are from the MiBench [43], SPECINT2000 and NetBench [60] suites. These benchmarks were chosen as they were representative of applications that are normally run on mobile devices.

For the media benchmarks, the SIMD-izable inner loops are not considered as they are assumed to execute on a normally configured SIMD datapath. The other benchmarks demonstrate limited data-level parallelism and, therefore, the entirety of the benchmarks are considered for execution under SIMD-Morph.

Figure 7.8 shows the distribution of various sizes of subgraphs in each of the applications in increments of 4. The overwhelming majority of subgraphs, 83%, are between 1

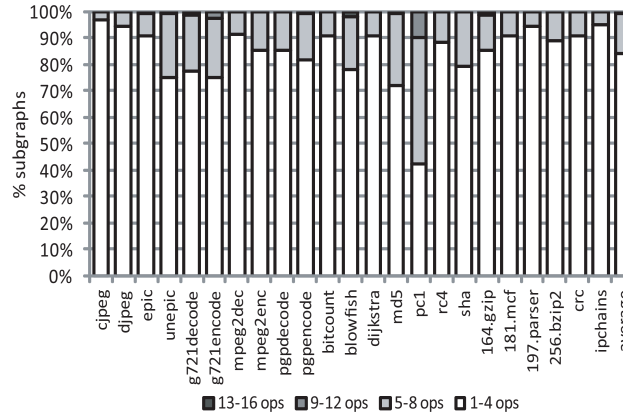


Figure 7.8: *Distribution of subgraphs consisting of various numbers of operations.*

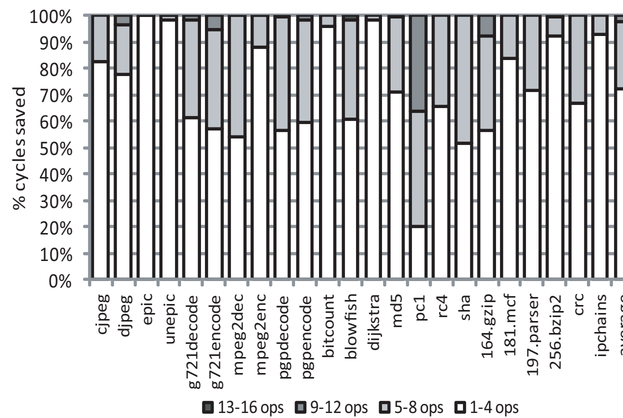


Figure 7.9: *% Cycles saved by subgraphs consisting of various numbers of operations.*

and 4 operations. Approximately 16% of subgraphs have between 5 and 8 ops and just over 1% of subgraphs have more than 8 operations.

Factoring in the speedup contributed by each of these subgraphs presents only a slightly different story. In Figure 7.9, each segment shows the % of overall cycles saved came from subgraphs of various sizes. The speedup contribution of 1 to 4-operation subgraphs is 71.5% and that of 5 to 6-operation subgraphs is 26%. This indicates that while there while there are few subgraphs larger than 4 operations in size, their speedup contribution is significant.

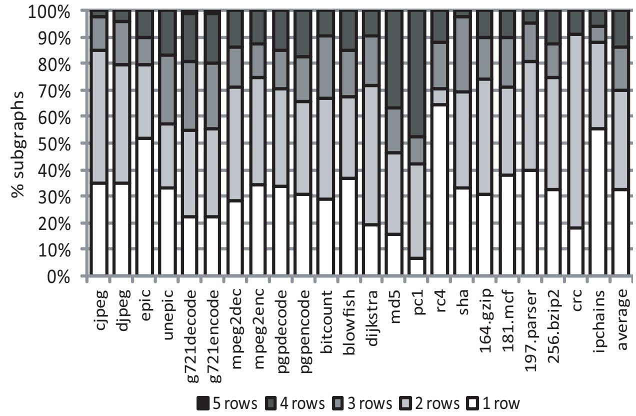


Figure 7.10: *Distribution of subgraphs consisting of various depths of operations.*

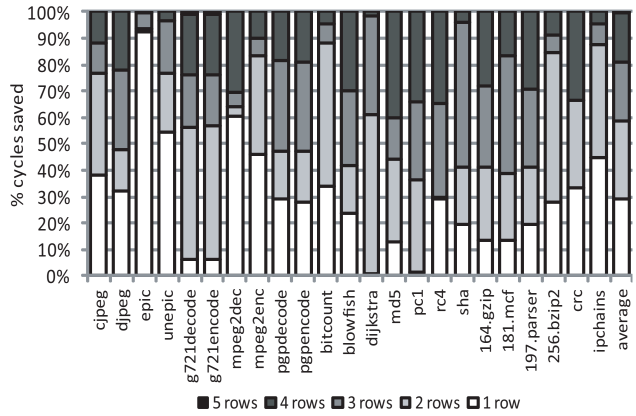


Figure 7.11: *% Cycles saved by subgraphs consisting of various depths of operations.*

Figures 7.10 and 7.11 illustrate the distribution of maximum depths of subgraphs and the speedups obtained by varying the subgraph depths. The subgraph depths for the benchmarks in consideration ranged from 1 operation to 5, although the execution frequency of the 5-deep subgraphs (there is one in each of `g721encode` and `g721decode`) was so small that it is not noticeable in these graphs.

On average, approximately 85% of the subgraphs have depths of 3 operations or fewer. The performance contribution paints a similar picture as well, with 80% of the performance improvement coming from subgraphs with depths of 3 operations or fewer.

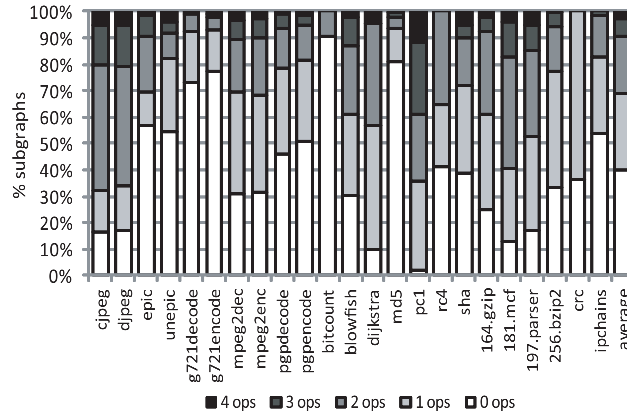


Figure 7.12: *Distribution of subgraphs consisting of various numbers of memory operations.*

Finally, Figure 7.12 shows the distribution of memory operations in the subgraphs. Over 90% of the subgraphs use only 2 of the 4 available memory units. However, over 60% of subgraphs have at least one memory operation, indicating that support for memory operations is still important, although it is unlikely that reducing the number of operations supported will have a significant impact on performance.

There is one salient point from all of these results: the baseline design is grossly over-provisioned. Further, the following may be considered:

- The maximum number of operations can be reduced, reducing the overall latency of SIMD-Morph
- Reducing the number of operations will also reduce the complexity of the interconnect network, further reducing latency and power
- Reducing the maximum depth of subgraphs by eliminating connections between some groups will prevent some subgraphs from executing but could potentially allow more, shallower subgraphs to execute. Increasing the number of subgraphs, however, could require increasing the number of live-in and live-out values

Config	#inputs	#outputs	#Mem	#Ops	Topology	Control bits req'd
`baseline`	4	2	4	16	Default	313
`config2`	**2**	2	4	16	Default	313
`config3`	**3**	2	4	16	Default	313
`config4`	**5**	2	4	16	Default	313
`config5`	**6**	2	4	16	Default	317
`config6`	4	2	**0**	16	Default	309
`config7`	4	2	**1**	16	Default	310
`config8`	4	2	**2**	16	Default	311
`config9`	4	2	4	16	**2 groups of 8**	313
`config10`	4	2	**2**	**8**	**1 group of 8**	157
`config11`	4	2	**1**	**4**	**1 group of 4**	79
`config12`	4	**1**	4	16	Default	311
`config13`	4	**3**	4	16	Default	315
`config14`	4	**4**	4	16	Default	317

Table 7.1: *Configurations used in design-space exploration. Entries in bold indicate deviations from the baseline.*

- Reducing the number of memory units will reduce the complexity of SIMD-Morph but will potentially severely limit the number of subgraphs that can be executed.

7.4 Design-Space Exploration and Results

The overall design-space was explored in order to best understand the bottlenecks of the design. The features that were varied include:

- The number of register inputs
- The number of register outputs
- The number of memory units in SIMD-Morph
- The interconnect topology

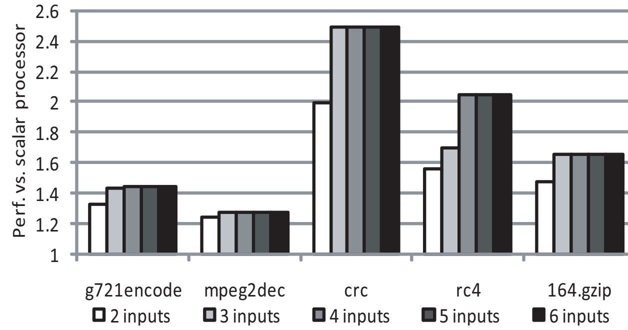


Figure 7.13: *Performance impact of varying the number of register inputs.*

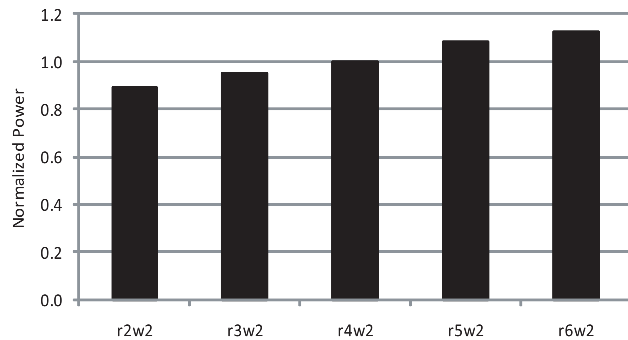


Figure 7.14: *Register file power with varying number of register inputs, normalized to the 4-port baseline.*

For the purposes of design-space exploration, a limited number of benchmarks were selected which were representative of the various benchmark suites in consideration: The Mediabench benchmarks `g721encode` and `mpeg2dec`, the Netbench benchmark `crc`, the encryption benchmark `rc4`, and the SPECINT2000 benchmark `164.gzip`. Each variation is compared to the baseline results shown in Section 7.3.

7.4.1 Varying Register Inputs

For these experiments, the number of register inputs was varied between 2, 3, 4 (the baseline), 5 and 6 as indicated by the configurations `config2`, `config3`, `config4` and `config5`. Every extra read port on the scalar register file contributes muxing cost and also

increases the encoding space required in the configuration memory. However, increasing the number of inputs also increases the number of parallel, independent operations that can be executed on SIMD-Morph and provide improved utilization and speedup.

Figure 7.13 shows the results of this exploration, with the second column showing the data for the baseline 4-input configuration. While there is significant improvement for `rc4`, most of the benchmarks show a very limited increase in performance; the principle reason for this being that the memory ports allow data to be "live-in" from memory arrays, reducing the sensitivity to the amount of live-in register data. The difference observed between the 2-input and 3-input performance of all the benchmarks shown, however, justifies the added cost of adding at least one extra port to a conventional 2-read-port register file. However, because performance saturates when using a 4-read-port register file, this is the preferred configuration. In our experiments, the power consumed by the scalar register file increases approximately linearly with the number of read ports, as shown in Figure 7.14.

7.4.2 Varying Number of Memory Units

The memory units in SIMD-Morph access memory via a multi-ported load/store queue. This system helps prevent inter-lock and overlapping accesses in the event of memory aliasing when executing pointer-intensive code. Increasing the number of units therefore requires appropriate modifications to the memory system. Further, address generation support needs to be added to elements in the SIMD datapath in order to support base+displacement memory operations. For these experiments, the number of memory units in SIMD-Morph was varied between 0, 1, 2 and 4 (the baseline) units as indicated by the configurations `config6`, `config7`, and `config8`.

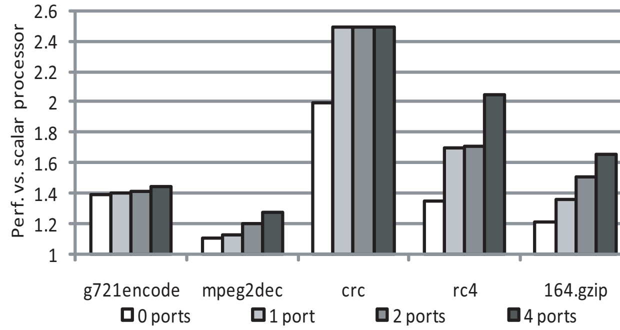


Figure 7.15: *Impact of varying the number of memory units on speedup.*

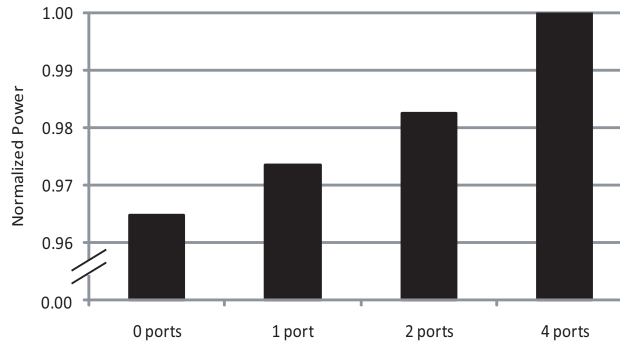


Figure 7.16: *Normalized power impact of varying the number of memory units relative to the 4-port baseline. Note that the y-axis does not start at 0.*

Figure 7.15 shows the results of this exploration, with the 4th column showing the data for the baseline 4-memory unit configuration. There is obviously a noticeable correlation between the performance impact of changing the number of memory units and the % of subgraphs that have the corresponding number of memory operations, shown in Figure 7.12. For instance, `164.gzip` is most sensitive to the number of units and it also has the most number of subgraphs that have at least 1 memory operation in them. Similarly, `g721encode` is the least sensitive and fewer than 30% of its subgraphs have memory operations. The power impact of varying the number of memory ports is quite negligible as shown in Figure 7.16 so there is very little incentive to reduce the total number of memory operations in SIMD-Morph.

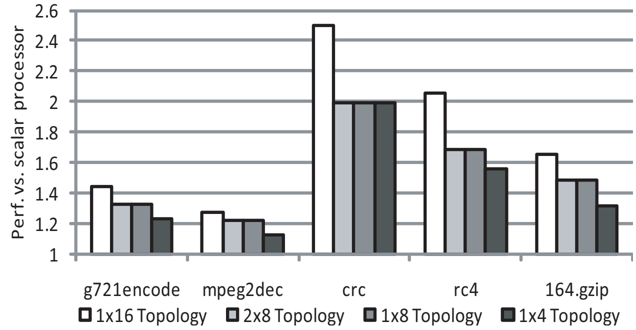


Figure 7.17: *Impact of varying the interconnect topology on speedup.*

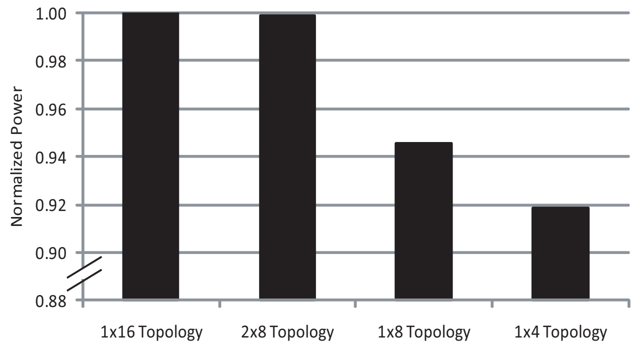


Figure 7.18: *Normalized power impact of varying the topology relative to the 1x16 baseline. Note that the y-axis does not start at 0.*

7.4.3 Varying Interconnect Topology

The data from Section 7.3 indicates that the performance impact of subgraphs with a large number of operations and with long chains of operation is, at best, minimal. In response to this, different SIMD-Morph topologies were explored, varying the interconnection network between the elements and also varying the total number of elements. These are indicated by configurations `config9`, `config10` and `config11`. In `config9`, each group of 8 elements does not communicate with the other, but one 4-element CEG within each group still feeds another. The configurations `config10` and `config11` explore the notion of not using all 16 lanes of the SIMD datapath but using 8 or 4, respectively, instead. The performance impact of these configurations is shown in Figure 7.17. The most impor-

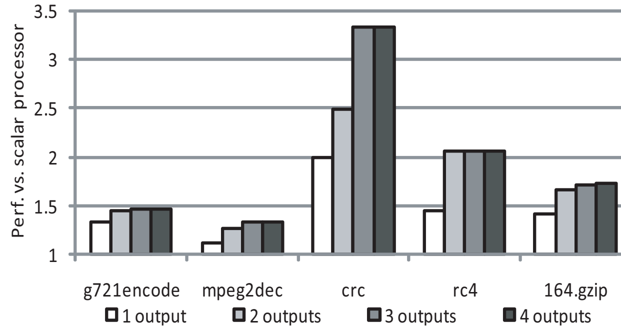


Figure 7.19: *Impact of varying the number of register outputs on speedup.*

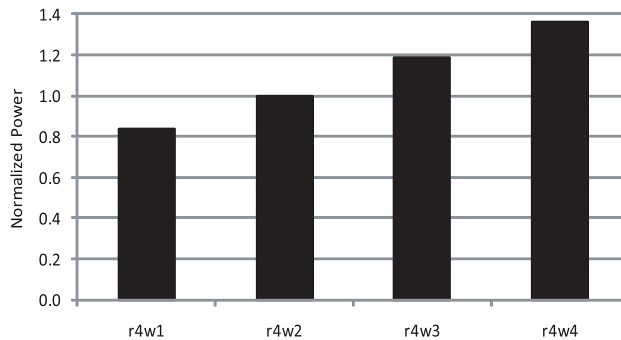


Figure 7.20: *Register file power with varying number of register outputs, normalized to the 2-port baseline.*

tant observation from this graph is that there is virtually no difference between the 2x8 and 1x8 configuration; i.e. once the maximum depth of subgraphs is halved, having extra units has no added benefit, so one may as well configure 8 elements in the SIMD-Morph style in order to save on overheads. The power savings from moving to a 1x16 configuration to a 2x8 configuration is also negligible as seen in Figure 7.18.

7.4.4 Varying Register Outputs

For these experiments, the number of register outputs was varied between 1, 2 (the baseline), 3 and 4. The modified configurations are indicated by configurations `config12`, `config13` and `config14`. Every extra write port on the scalar register file contributes

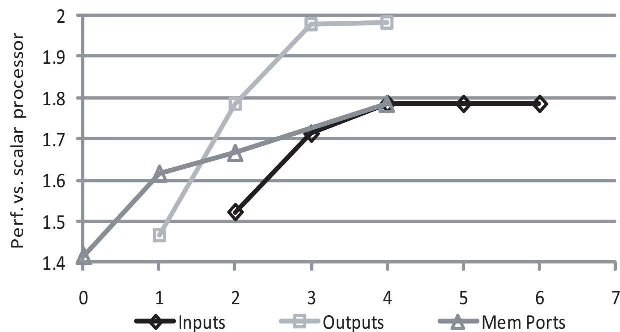


Figure 7.21: *Average performance improvements with varying configurations.*

muxing cost and also increases the encoding space required in the configuration memory. There is also extra overhead associated with forwarding values from the write ports to the read ports. However, much like with increasing the number of inputs, increasing the number of outputs also increases the number of parallel, independent operation chains that can be executed on SIMD-Morph, providing improved utilization and speedup, as shown in Figure 7.19.

Memory access support in SIMD-Morph allows for reduced use of register live-outs especially in situations where final values are written to memory arrays. For this reason, the benchmarks here are, to a point, insensitive to the number of outputs. A minimum of 1 or 2 is required to support incrementing loop counter or pointer values. A notable exception to this is, of course, `crc`, where few hot loops store values out to memory but values are live-out from one loop iteration to the next. Figure 7.20 shows the normalized power consumption of varying the number of write ports to the register file, indicating an approximately linear relationship between the number of ports and power consumed.

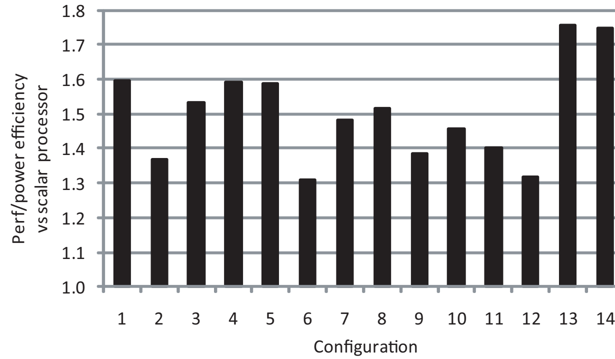


Figure 7.22: *Average performance/power efficiency improvement of SIMD-Morph with the different datapath configurations.*

7.5 Results

The simplest configuration that allows for the maximum performance from SIMD-Morph has 4 input values, 3 output values and 4 memory ports, using the baseline 1x16 interconnect topology. Performance saturation from using more complex configurations is illustrated in Figure 7.21, which shows no change in performance when using more than 4 input values and a negligible performance improvement when using more than 3 output values. This configuration is `config13` in Table 7.1. Figure 7.22 illustrates the performance:power efficiency of each of the configurations used in the design-space exploration . This figure, too, illustrates the optimal efficiency of `config13`.

The overall performance improvement when using the optimal-efficiency configuration of SIMD-Morph is shown in Figure 7.23. For the benchmarks from the Mediabench suite, only the non-SIMD-ized portions of the code are considered for acceleration on SIMD-Morph, while for the other benchmarks, the entire application is considered. For this reason, the averages of the two categories are displayed separately. The average speedup for the media applications is 1.4X, while the average speedup for the other applications is 2.6X.

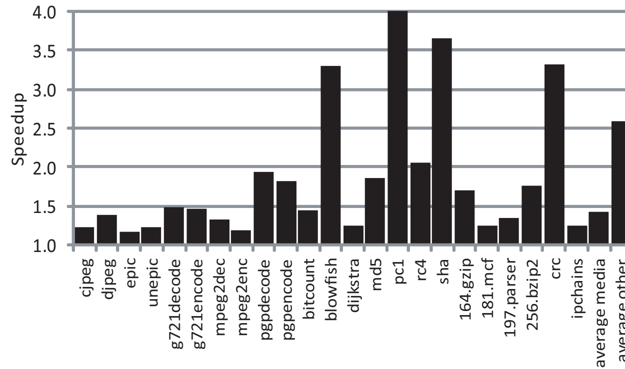


Figure 7.23: *Speedup from using SIMD-Morph using the optimal configuration. "Average media" refers to the average speedup obtained from accelerating the outer loops of Mediabench applications while "average other" refers to the average speedup obtained from accelerating the entirety of the other applications. The speedup seen in* `pc1` *is 9.5X but is capped at 4X in this graph.*

SIMD-Morph was synthesized on a 90nm technology with a targeted clock frequency of 200 MHz using Synopsys Design Compiler and Synopsys Physical Compiler. Power consumption was measured using Synopsys Primetime-PX. The chosen baseline assumes a processor with 16-wide SIMD datapath and a 16-entry scalar register file with three read ports and two write ports, the power consumption of which is 63.48mW. The baseline SIMD-Morph configuration, illustrated in Figure 7.6 is connected to a scalar register file with four read ports and two write ports, the power consumption of which is 71.01mW – a power overhead of 11.9% for the modified components. The efficiency-optimal configuration, `config13`, with three write ports instead of two in the scalar register file has a power consumption of 71.53mW – a power overhead of 12.7% for the modified components. The power consumption of the scalar datapath and the SIMD register files is not accounted for here as they are not modified in this work; accounting for the power consumption of these components will reduce this power overhead. The average performance/power efficiency improvement of using the optimal SIMD-Morph configuration is 1.75X.

7.6 Related Work

Utilizing instruction set extensions to improve the computational efficiency of applications is a well studied field. Examples of industry standard domain specific instruction set extensions such as Intel's SSE or AMD's 3DNow! multimedia instructions are commonplace in modern systems. Techniques for generating domain specific extensions are typically ad-hoc, where an architect examines a family of target applications and determines what extensions can be expected to provide increased performance.

In contrast to domain specific extensions, many techniques for generating application specific instruction set extensions have been proposed [4, 9, 19, 40, 47, 93]. Each of these algorithms provide either exact formulations or heuristics to effectively identify those portions of an application's dataflow graph that can efficiently be implemented in hardware. These techniques are not directly applicable to this work, because they do not take into account the underlying structure of the execution hardware.

Much attention has been given to the structure of a CCA (configurable compute accelerator) design for accelerating dataflow subgraphs. The research in [115] proposed using a fine granularity CCA based on slightly specialized FPGA-like elements. Restricting the interconnect of the FPGA-like elements reduces the delay of a CCA without radically affecting the number of subgraphs that can be mapped onto the accelerator. While the flexibility to map many subgraphs onto configurable hardware is appealing, there are significant drawbacks of a large number of control bits and the substantial delay of FPGA-like elements. A key observation is that the flexibility of an FPGA is generally more than is necessary for dataflow graph acceleration.

Once a CCA execution engine is developed, techniques are needed to map dataflow subgraphs onto the execute engines. Many hardware based frameworks exist for this process. Most of these arose from the observation that in systems with a trace cache, the latency of the fill unit has a negligible performance impact until it becomes very large (on the order of 10,000 cycles [35]). That is, once instructions retire from the pipeline and a trace is constructed, there is ample time before that trace will be needed again. Two recently proposed schemes [14, 82] used this latency to perform the mapping of dataflow subgraphs onto specialized execution hardware. While the trace cache provides an excellent place for mapping subgraphs into the instruction stream, it is far too large and inefficient for embedded domains.

A simplified dynamic subgraph mapping system was described in [48, 85]. These papers used the design proposed in [74] as the baseline of their system, which greatly simplifies the mapping problem. Because our goal was to allow for more flexibility than their CCA design allowed for, our presented identification algorithm is much more complex.

The key difference between this paper and prior work is that instead of proposing a CCA *architecture*, we propose an *architecture framework* into which many CCA architectures fit. This framework provides a clean interface between a processor pipeline and a CCA, enabling easy customization of a CCA for the expected system workload. We demonstrate how the framework can process a dataflow subgraph to generate CCA instruction on the fly, without the costs associated with a trace cache. Beyond the architecture framework, we describe the compilation process, by which subgraphs are identified in applications and communicated to the architecture framework.

7.7 Conclusion

Modern wireless devices are often equipped with wide SIMD processors in order to exploit data-level parallelism that is very often present in the media applications that these devices need to execute. However, not all portions of these applications is amenable to SIMD-izing. Further, several other, non-media applications are often run on these devices as well and during their executing only the scalar portion of the processor is used while the SIMD datapath is left idle. SIMD-Morph is a design which allows the SIMD datapath to be used in these situations as well. This design modifies a standard 32-bit, 16-wide SIMD datapath by adding connectivity between the different lanes in order to exploit instruction-level parallelism in sections of code that would normally be executed on the scalar pipeline of a SIMD processor. The performance benefit of SIMD-Morph is evaluated in two ways: speedup obtained from executing outer loops of applications when the inner loops are easily SIMD-ized and also the speedup obtained from executing purely sequential code on a substrate that is normally used to execute code with data-level parallelism. The performance impact for these two scenarios is a very impressive 1.4X and 2.6X, respectively.

CHAPTER VIII

Future Work

8.1 Efficient CPU-GPU Fusion

GPUs have evolved considerably to become the driving force behind "desktop supercomputers". Many data-parallel applications show tremendous acceleration when ported to GPUs – something which is only becoming more popular as GPUs are increasingly more available and are increasingly easier to program. To provide such tremendous compute power, GPU architectures are considered to be a case of extreme design which leads to enormous power consumption.

It is this power consumption that is limiting the pervasive use of GPUs as computation work horses in mobile devices. Solutions such as PEPSC (VI) have been proposed to provide a low power scientific compute engines which can be used on embedded platforms. However, the absence of any graphics capability is a severe limitation of these engines for mass-market mobile devices like smart-phones. An extension of the work presented in this thesis is a unified graphics/SIMD architecture that can readily alternate between general-purpose throughput-computing and also high-performance graphics processing. This is in

contrast to existing GPGPU architectures which can only operate either in “graphics” or in “general purpose” mode at any given time.

The application domain most helped by such work is that of augmented reality and computer vision; applications where a user’s mobile device understands the user’s environment through visual, location or other input, supplements it with rich content and recreates, or “augments”, the environment according to the user’s requirements. Such application require graphics performance in order to render visual information in a timely manner, and require high-throughput computation in order to quickly understand and process the information in the user’s environment.

Unified graphics/SIMD architectures also help provide an overall improvement in power and area efficiency. In applications where either the graphics functionality or data-parallel processing are required exclusively, unifying the hardware required by both, like caches and floating-point datapaths, can help reduce both the power and area foot print of a system, thereby reducing cost. Unifying caches can also help improve performance as it mitigates the latency of data-transfer between the graphics and general-purpose processing units.

Extending this thesis to unify throughput and graphics computing will unlock considerable performance potential and power efficiency in mobile devices, enabling the applications discussed here and many more.

8.2 System Design for Portable, Low-Power Medical Ultrasound

It is envisioned that ten years from now, every physician will be equipped with a hand-held ultrasound capable of generating real-time 3D medical images superior to anything

currently published. This “Stethoscope of the Future”, is a must-have device for diagnostic medicine that should be as easy to use and maintain as stethoscopes and thermometers.

Over the years, ultrasound has increased in both capability and portability. Current systems can now image anatomic structure, provide qualitative and quantitative blood flow information and have advanced from 2D to 3D imaging. The real-time update rate (e.g. >100 frames/sec for cardiac imaging) of ultrasound imaging still exceeds that of MRI and CT, requires far lower equipment investment, and does not use ionizing radiation. Most major ultrasound vendors now have portable offerings, and several firms specialize exclusively in portable systems. The features (e.g., color and pulse Doppler) and image quality of portable systems vary, in part due to hardware limitations and market considerations. For example, some portable systems limit the number of processing channels and array elements used to produce images. While portable-system image quality has improved steadily, it still falls short of advanced (non- portable) ultrasound systems. Even high-end research systems are limited by circuit, processing density, and power constraints.

Until recently, ultrasound images have been 2D views (e.g., a real time image slice through an organ) and have relied on highly-trained specialists to properly orient the probe and select the position within the body to image with only few and complex visual cues. The advent of realtime 3D (and animated 4D) ultrasound drastically improves system ease-of-use and has already demonstrated clinical utility, including substantial improvements in diagnostic efficiency and patient throughput. Our goal is to develop a battery-operated hand-held ultrasound platform that can generate volumetric imaging of size 6x8x14cm with least 1 imaging volume/sec while exceeding the best-available image quality of current ultrasound systems. Such a system will revolutionize the accessibility and usability

of advanced medical imaging, and the battery operated form-factor will ease deploying ultrasound in parts of the developing world where advanced medical imaging is not readily available today. Moreover, it will be an enabling technology for medical research by providing the first comprehensive access to raw data from every element in a fully sampled 2D array for algorithm and application testing.

To be pursued are two fundamental advances in ultrasound imaging technology that are enabled by an ultra-dense embedded imaging platform: 1) hand-held 3D ultrasound imaging with breakthrough resolution and 2) aberration correction.

There are significant challenges to achieving these ambitious goals. To image a 3D volume with acceptable resolution for diagnosis requires a 128x128 element transducer array and, at 12-bit digitization, the capability to process 14 terabytes/sec. In todays technology, we estimate the power consumption of such a device at around 120W (10W for transmit circuitry, 78W for analog-to-digital conversion (ADC) and 32W for the DSP processors). We expect to leverage device scaling to reduce power requirements to under 12W in 5-8 years. Over the past decade, ADC energy efficiency has been improving at a rate of 18X per decade. Over the next decade this trend is expected to continue through process scaling, the use of new ADC architectures, and the use of more complex calibration by migrating complexity into the digital domain. However, for a battery-operated device, our target power budget is only 5W. Furthermore, the small footprint of this device demands that the analog circuits for data acquisition and digital circuits for processing such large data volumes must be stacked.

The variability of the speed of sound in tissue poses a fundamental barrier to achieving high ultrasound image quality. This phenomenon causes image aberration that must

be corrected to further advance the state-of-the-art. Only by having arrays with elements distributed in 2D can the appropriate correction be made. Development of signal processors capable of aberration correction will benefit all ultrasound systems, regardless of form factor.

Such a processor should be of a low-power, programmable architecture that performs delay-and-sum (DAS) beamforming and simple phase aberration correction. For DAS, since operations in a channel are independent of each other, a wide SIMD machine, where each lane of the machine processes the data in a channel, would be highly suitable. For phase aberration, windowing operations are performed on data in neighboring channels, so the proposed architecture has to support operations on localized data as well. Building on previous work on SIMD architectures, a low-power, wide-SIMD architecture can be developed for beamforming and phase aberration correction. Since the number of channels is large, a multi-core system, where each core is a wide-SIMD processing engine will be required. The cores will be organized into clusters, and each cluster will be equipped with a large memory and a fast interconnection bus. Operations on neighboring data can be supported by using shuffle and swizzle networks attached to the local memory of each core.

This processor may also be adapted for more advanced signal processing techniques such as phase aberration algorithms based on generalized coherence factors, more complex beamforming based on delay focus FIR and spatial matched filter and coded excitation. To account for the increased amount of computation that will be required and the increase in storage requirement, more cores and 3D-stacked memory may be used.

CHAPTER IX

Conclusions

Modern general-purpose processors are inadequate for a range of new application domains with very stringent performance and power requirements. Desktop processors are designed to execute a very wide variety of programs but are not designed with any specific domain in mind and, as such, operate inefficiently in terms of power and energy usage and, quite often, cannot execute applications quickly enough or with sufficient throughput. General-purpose GPUs are becoming an increasingly popular platform to run applications that require a high computation throughput. They are limited, however, by memory bandwidth and power and, as such, cannot always achieve their full potential.

This work presented a number of solutions applicable to popular high-throughput computing domains, namely computation for wireless communication devices, computation for image reconstruction and computation for scientific and numerical applications, each of which in turn require have slightly different requirements from the hardware on which they execute.

For fixed-function performance, where the applications are stable, well-understood, and not routinely changing, the BLADES system provides the necessary performance while

incorporating aggressive dynamic voltage and frequency scaling to effectively reduce the power and energy consumption.

For portable medical imaging applications where the performance and power-efficiency of an ASIC is required but some programmability is also necessary in order to accomodate changing reconstruction algorithms, the PUMA system is a tiled architecture that makes use of programmable loop accelerators. This sytem can match the performance of modern GPUs in this domain with a mere fraction of the power consumed.

MEDICS and SIMD-Morph are solutions for applications where full generality is required but power-efficiency is still extremely important. MEDICS provides an architecture that can effectively execute medical image reconstruction applications that have a high amount of data-level parallelism and has a memory system that can effectively feed these data-hungry applications without wasting processor resourced. SIMD-Morph also allows for data-level parallelism using a SIMD datapath but, in addition, provides a communication network between the SIMD functional units that allow it to exploit instruction-level parallelism present in most scalar applications as well. Such an architecture is very suitable in wireless communication devices which may be required to execute both parallel and sequential applications.

PEPSC extends the functionality provided by MEDICS into the broad scientific and numerical computing domain. It tackles specific performance-limiting aspects of modern GPUs with novel, easy-to-use microarchitectural modifications. These modifications dramatically improve the utilization of the PEPSC SIMD datapath and provide considerable improvement in power efficiency – 10X that of modern GPUs. The microarchitectural

modifications are also general enough to be applied to modern GPU architectures, providing them with a 70% improvement in utilization.

Power consumption is a first-order design constraint for modern processors. However, with increasing performance requirements, power consumption can only be reduced by rapidly improving a processor's power efficiency. The techniques presented in this thesis are the path to significant efficiency improvements and a gateway to new computing paradigms.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] J. H. Ahn et al. Evaluating the Imagine stream architecture. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, pages 14–25, June 2004.

[2] ARC International. Arctangent processor. http://www.arc.com.

[3] E. Asma, D. Shattuck, and R. Leahy. Lossless compression of dynamic PET data. *IEEE Transactions on Nuclear Science*, 50(1):9–16, Feb. 2003.

[4] K. Atasu, L. Pozzi, and P. Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *Proc. of the 40th Design Automation Conference*, pages 256–261, June 2003.

[5] K. Bae and B. Whiting. CT data storage reduction by means of compressing projection data instead of images: Feasibility study. *Radiology*, 219(3):850–855, June 2001.

[6] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing cuda workloads using a detailed GPU simulator. In *Proc. of the 2009 IEEE Symposium on Performance Analysis of Systems and Software*, pages 163–174, Apr. 2009.

[7] D. Blaauw, S. Kalaiselvan, K. Lai, W.-H. Ma, S. Pant, C. Tokunaga, S. Das, and D. Bull. Razor II: In-situ error detection and correction for PVT and SER tolerance. In *IEEE International Solid-State Circuits Conference*, Feb. 2008.

[8] H.-M. Bluethgen, C. Grassmann, W. Raab, and U. Ramacher. A programmable platform for software-defined radio. *International Symposium on System-on-Chip*, pages 15–, Nov. 2003.

[9] P. Brisk et al. Instruction generation and regularity extraction for reconfigurable processors. In *Proc. of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 262–269, 2002.

[10] T. Burd, T. Pering, A. Stratakos, and R. Brodersen. A dynamic voltage scaled microprocessor system. *Journal of Solid State Circuits*, 35(11):1571–1580, Nov. 2000.

[11] S. Che, M. Boyer, J. Meng, D. Tarjan, , J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. of the IEEE Symposium on Workload Characterization*, pages 44–54, 2009.

[12] T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5):609–623, 1995.

[13] S. Ciricescu et al. The reconfigurable streaming vector processor (RSVP). In *Proc. of the 36th Annual International Symposium on Microarchitecture*, pages 141–150, 2003.

[14] N. Clark et al. Application-specific processing on a general-purpose core via transparent instruction set customization. In *Proc. of the 37th Annual International Symposium on Microarchitecture*, pages 30–40, Dec. 2004.

[15] N. Clark et al. An architecture framework for transparent instruction set customization in embedded processors. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, pages 272–283, June 2005.

[16] N. Clark et al. Liquid SIMD: Abstracting SIMD hardware using lightweight dynamic mapping. In *Proc. of the 13th International Symposium on High-Performance Computer Architecture*, pages 216–227, 2007.

[17] N. Clark, A. Hormati, and S. Mahlke. VEAL: Virtualized execution accelerator for loops. In *Proc. of the 35th Annual International Symposium on Computer Architecture*, pages 389–400, June 2008.

[18] N. Clark, A. Hormati, S. Mahlke, and S. Yehia. Scalable subgraph mapping for acyclic computation accelerators. In *Proc. of the 2006 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 147–157, Oct. 2006.

[19] N. Clark, H. Zhong, and S. Mahlke. Processor acceleration through automated instruction set customization. In *Proc. of the 36th Annual International Symposium on Microarchitecture*, pages 129–140, Dec. 2003.

[20] CNET. The Gizmo Report: NVIDIA's GeForce GTX 280 GPU – introduction, 2008. http://news.cnet.com/8301-13512_3-9969234-23.html.

[21] H. Corporaal. TTAs: Missing the ILP complexity wall. *Journal of System Architecture*, 45(1):949–973, 1999.

[22] W. Dally et al. Merrimac: Supercomputing with streams. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, pages 35–42, 2003.

[23] S. Das et al. A self-tuning DVS processor using delay-error detection and correction. *Journal of Solid State Circuits*, 41(4):792–804, 2006.

[24] G. Dasika, S. Das, K. Fan, S. Mahlke, and D. Bull. DVFS in loop accelerators using BLADES. In *Proc. of the 45th Design Automation Conference*, pages 894–897, June 2008.

[25] G. Dasika, K. Fan, and S. Mahlke. Power-efficient medical image processing using puma. In *Proceedings of the IEEE 7th Symposium on Application Specific Processors*, pages 29–34, 2009.

[26] G. Dasika, A. Sethia, T. Mudge, and S. Mahlke. Low-power scientific computing. In *Proc. of the 2009 Workshop on New Directions in Computer Architecture*, pages 21–22, 2009.

[27] G. Dasika, M. Woh, S. Seo, N. Clark, T. Mudge, and S. Mahlke. Mighty morphing power-simd. In *Proc. of the 2010 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 67–76, 2010.

[28] A. B. de Gonzalez et al. Projected cancer risks from computed tomographic scans performed in the United States in 2007. *Archives of Internal Medicine*, 169(22):2071–2077, 2009.

[29] S. Dhar, D. Maksimovic, and B. Kranzen. Closed-loop adaptive voltage scaling controller for standard-cell ASICs. In *Proc. of the 2002 International Symposium on Low Power Electronics and Design*, pages 103–107, Aug. 2003.

[30] P. Diaz and M. Cintra. Stream chaining: exploiting multiple levels of correlation in data prefetching. In *Proc. of the 36th Annual International Symposium on Computer Architecture*, pages 81–92, 2009.

[31] C. Ebeling et al. Mapping applications to the RaPiD configurable architecture. In *Proc. of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 106–115, Apr. 1997.

[32] K. Fan et al. Modulo scheduling for highly customized datapaths to increase hardware reusability. In *Proc. of the 2008 International Symposium on Code Generation and Optimization*, pages 124–133, Apr. 2008.

[33] K. Fan, M. Kudlur, H. Park, and S. Mahlke. Compiler-directed synthesis of multifunction loop accelerators. In *Proc. of the 2005 Workshop on Application Specific Processors*, pages 91–98, Sept. 2005.

[34] J. A. Fisher et al. Custom-fit processors: Letting applications define architectures. In *Proc. of the 29th Annual International Symposium on Microarchitecture*, pages 324–335, Dec. 1996.

[35] D. Friendly, S. Patel, and Y. Patt. Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors. In *Proc. of the 25th Annual International Symposium on Computer Architecture*, pages 173–181, June 1998.

[36] J. W. C. Fu, J. H. Patel, and B. L. Janssens. Stride directed prefetching in scalar processors. In *Proc. of the 25th Annual International Symposium on Microarchitecture*, pages 102–110, 1992.

[37] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proc. of the 40th Annual International Symposium on Microarchitecture*, pages 407–420, 2007.

[38] J. Glossner, E. Hokenek, and M. Moudgill. The Sandbridge Sandblaster Communications Processor. In *3rd Workshop on Application Specific Processors*, pages 53–58, Sept. 2004.

[39] S. Goldstein et al. PipeRench: A coprocessor for streaming multimedia acceleration. In *Proc. of the 26th Annual International Symposium on Computer Architecture*, pages 28–39, June 1999.

[40] D. Goodwin and D. Petkov. Automatic generation of application specific processors. In *Proc. of the 2003 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 137–147, 2003.

[41] V. Govindaraju et al. Toward a multicore architecture for real-time ray-tracing. In *Proc. of the 41st Annual International Symposium on Microarchitecture*, pages 176–197, Dec. 2008.

[42] T. Gunnarsson et al. Mobile computerized tomography scanning in the neurosurgery intensive care unit: increase in patient safety and reduction of staff workload. *Journal of Neurosurgery*, 93(3):432–436, Sept. 2000.

[43] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. of the 4th IEEE Workshop on Workload Characterization*, pages 10–22, Dec. 2001.

[44] A. K. Hara et al. Iterative reconstruction technique for reducing body radiation dose at CT: Feasibility study. *American Journal of Roentgenology*, 193(3):764–771, Sept. 2009.

[45] T. D. Hartley et al. Biomedical image analysis on a cooperative cluster of GPUs and multicores. In *Proc. of the 2008 International Conference on Supercomputing*, pages 15–25, 2008.

[46] S. Hu and J. E. Smith. Using dynamic binary translation to fuse dependent instructions. In *Proc. of the 2004 International Symposium on Code Generation and Optimization*, pages 213–226, 2004.

[47] I. Huang. *Co-Synthesis of Instruction Sets and Microarchitectures*. PhD thesis, University of Southern California, 1994.

[48] Q. Jacobson and J. E. Smith. Instruction pre-processing in trace processors. In *Proc. of the 5th International Symposium on High-Performance Computer Architecture*, pages 125–133, 1999.

[49] J. Kelm et al. Rigel: An architecture and scalable programming interface for a 1000-core accelerator. In *Proc. of the 36th Annual International Symposium on Computer Architecture*, pages 140–151, June 2009.

[50] C. Kozyrakis and C. Patterson. Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks. In *Proc. of the 35th Intl. Symposium on Microarchitecture*, pages 283–293, Nov. 2002.

[51] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, , and K. Asanovic. The vector-thread architecture. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, pages 52–63, 2004.

[52] C. Lee, M. Potkonjak, and W. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proc. of the 30th Annual International Symposium on Microarchitecture*, pages 330–335, 1997.

[53] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *Proc. of the 37th Annual International Symposium on Computer Architecture*, pages 451–460, 2010.

[54] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, and C. Chakrabarti. SODA: A low-power architecture for software radio. In *In Proc. of the 33rd Annual International Symposium on Computer Architecture*, pages 89–101, 2006.

[55] G. H. Loh. 3D-Stacked memory architectures for multi-core processors. In *Proc. of the 35th Annual International Symposium on Computer Architecture*, pages 453–464, 2008.

[56] G. Lu et al. The MorphoSys parallel reconfigurable system. In *Proc. of the 5th International Euro-Par Conference*, pages 727–734, 1999.

[57] C. Maass et al. CT image reconstruction with half precision floating point values, 2009.

[58] A. Mahesri et al. Tradeoffs in designing accelerator architectures for visual computing. In *Proc. of the 41st Annual International Symposium on Microarchitecture*, pages 164–175, Nov. 2008.

[59] B. Mei et al. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *Proc. of the 2003 Design, Automation and Test in Europe*, pages 296–301, Mar. 2003.

[60] G. Memik, W. H. Mangione-Smith, and W. Hu. NetBench: A benchmarking suite for network processors. In *Proc. of the 2001 International Conference on Computer Aided Design*, pages 39–42, 2001.

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

[62] Motorola. *CPU12 Reference Manual*, June 2003. http://e-www.motorola.com/brdata/PDFDB/docs/CPU12RM.pdf.

[63] M. Nicolaidis. Time redundancy based soft-error tolerance to rescue nanometer technologies. In *Proc. of the 1999 IEEE VLSI Test Symposium*, pages 86–94, 1999.

[64] NVIDIA. *CUDA Programming Guide*, June 2007. http://developer.download.nvidia.com/compute/cuda.

[65] NVIDIA. GeForce GTX 200 GPU architectural overview, 2008. http://www.nvidia.com/docs/IO/55506/GeForce_GTX_200_GPU_Technical_Brief.pdf.

[66] NVIDIA. GeForce GTX 280, 2008. http://www.nvidia.com/object/product_geforce_gtx_280_us.html.

[67] NVIDIA. NVIDIA Tesla S1070, 2008. http://www.nvidia.com/object/ product_tesla_s1070_us.html.

[68] Nvidia. Cuda Zone, 2009. http://www.nvidia.com/object/cuda_home.html.

[69] J. Orchard and J. T. Yeow. Toward a flexible and portable CT scanner. In *Medical Image Computing and Computer-Assisted Intervention 2008*, pages 188–195, 2008.

[70] M. Papadonikolakis et al. Efficient high-performance ASIC implementation of JPEG-LS encoder. In *Proc. of the 2007 Design, Automation and Test in Europe*, pages 159–164, Apr. 2007.

[71] M. Papadonikolakis et al. Efficient high-performance implementation of JPEG-LS encoder. *Journal of Real-Time Image Processing*, 3(4):303–310, Dec. 2008.

[72] P. Paulin. Real-life challenges on mapping high-end video to mp-soc, 2009. 9th International Forum on Embedded MPSoC and Multicore.

[73] D. Pham et al. The design and implementation of a first generation CELL processor. In *IEEE Intl. Solid State Circuits Symposium*, Feb. 2005.

[74] J. Phillips and S. Vassiliadis. High-performance 3-1 interlock collapsing ALU's. *IEEE Transactions on Computers*, 43(3):257–268, 1994.

[75] F. Pratas, P. Trancoso, A. Stamatakis, and L. Sousa. Fine-grain parallelism using multi-core, Cell/BE, and GPU systems: Accelerating the phylogenetic likelihood function. In *Proc. of the 2009 International Conference on Parallel Processing*, pages 9–17, 2009.

[76] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, Nov. 1994.

[77] I. T. U. M. Recommendation. *Framework and overall objectives of the future development of IMT-2000 and systems beyond IMT-2000"*. http://www.ieee802.org/secmail/pdf00204.pdf.

[78] M. T. Review. Cheap, Portable MRI, 2006. http://www.technology-review.com/computing/17499/?a=f.

[79] D. Rivera, D. Schaa, M. Moffie, and D. Kaeli. Exploring novel parallelization technologies for 3-D imaging applications. In *Symposium on Computer Architecture and High Performance Computing*, pages 26–33, 2007.

[80] A. Ruiz, M. Ujaldon, L. Cooper, and K. Huang. Non-rigid registration for large sets of microscopic images on graphics processors. *Springer Journal of Signal Processing*, May 2008.

[81] K. Sankaralingam et al. Exploiting ILP, TLP, and DLP using polymorphism in the TRIPS architecture. In *Proc. of the 30th Annual International Symposium on Computer Architecture*, pages 422–433, June 2003.

[82] P. Sassone and D. S. Wills. Dynamic strands: Collapsing speculative dependence chains for reducing pipeline communication. In *Proc. of the 37th Annual International Symposium on Microarchitecture*, pages 7–17, Dec. 2004.

[83] P. Sassone, D. S. Wills, and G. Loh. Static strands: safely collapsing dependence chains for increasing embedded power efficiency. In *Proc. of the 2005 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 127–136, June 2005.

[84] A. Savakis and M. Piorun. Benchmarking and hardware implementation of JPEG-LS. In *2002 International Conference on Image Processing*, pages 949–952, Sept. 2002.

[85] Y. Sazeides, S. Vassiliadis, and J. E. Smith. The performance potential of data dependence speculation & collapsing. In *Proc. of the 29th Annual International Symposium on Microarchitecture*, pages 238–247. IEEE Computer Society, 1996.

[86] D. Schaa and D. Kaeli. Exploring the multiple-gpu design space. In *2009 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–12, 2009.

[87] R. Schreiber et al. PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators. *Journal of VLSI Signal Processing*, 31(2):127–142, 2002.

[88] L. Seiler et al. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, 27(3):1–15, 2008.

[89] N. Sinha and J. Yeow. Carbon nanotubes for biomedical applications. *IEEE Transactions on Nanobioscience*, 4(2):180–195, June 2005.

[90] M. Sivaraman and S. Aditya. Cycle-time aware architecture synthesis of custom hardware accelerators. In *Proc. of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 35–42, 2002.

[91] S. Srinath et al. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *Proc. of the 13th International Symposium on High-Performance Computer Architecture*, pages 63–74, Feb. 2007.

[92] S. S. Stone et al. Accelerating advanced MRI reconstructions on GPUs. In *2008 Symposium on Computing Frontiers*, pages 261–272, 2008.

[93] F. Sun et al. Synthesis of custom processors based on extensible platforms. In *Proc. of the 2002 International Conference on Computer Aided Design*, pages 641–648, Nov. 2002.

[94] M. Taylor et al. Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, pages 2–13, June 2004.

[95] M. B. Taylor et al. The Raw microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[96] Tensilica Inc. *Diamond Standard Processor Core Family Architecture*, July 2007. http://www.tensilica.com/pdf/Diamond WP.pdf.

[97] Texas Instruments. *TMS320C54X DSP Reference Set*, Mar. 2001. http://www-s.ti.com/sc/psheets/spru131g/spru131g.pdf.

[98] Texas Instruments. *TMS320C6000 CPU and Instruction Set Reference Guide*, July 2006. http://focus.ti.com/lit/ug/spru189g/spru189g.pdf.

[99] Texas Instruments. CT Scanner: Medical Imaging Solutions from Texas Instruments, 2009. http://focus.ti.com/docs/solution/folders/print/529.html.

[100] Tezzaron Semiconductor. FaStack Memory, 2009. http://www.tezzaron.com/memory/FaStack_memory.html.

[101] J.-B. Thibault, K. Sauer, C. Bouman, and J. Hsieh. A three-dimensional statistical approach to improved image quality for multi-slice helical CT. *Medical Physics*, 34(11):4526–4544, Nov. 2007.

[102] P. Trancoso, D. Othonos, and A. Artemiou. Data parallel acceleration of decision support queries using Cell/BE and GPUs. In *2009 Symposium on Computing Frontiers*, pages 117–126, 2009.

[103] Trenton Systems. Trenton JXT6966 News Release, 2010. http://www.trentontechnology.com/.

[104] Trimaran. An infrastructure for research in ILP, 2000. http://www.trimaran.org/.

[105] K. Usami et al. Design methodology of ultra low-power MPEG4 codec core exploiting voltage scaling techniques. In *Proc. of the 35th Design Automation Conference*, pages 483–488, 1998.

[106] K. van Berkel, F. Heinle, P. P. E. Meuwissen, K. Moerman, and M. Weiss. Vector processing as an enabler for software-defined radio in handheld devices. *EURASIP J. Appl. Signal Process.*, 2005(1):2613–2625, 2005.

[107] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation cores: reducing the energy of mature computations. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 205–218, 2010.

[108] D. Weinreb and J. Stahl. The introduction of a portable head/neck CT scanner may be associated with an 86% increase in the predicted percent of acute stroke patients treatable with thrombolytic therapy, 2008. Radiological Society of North America.

[109] M. Woh et al. The next generation challenge for software defined radio. In *Proc. of the* 7th*International Symposium on Systems, Architectures, Modeling, and Simulation*, pages 343–354, July 2007.

[110] M. Woh et al. From SODA to scotch: The evolution of a wireless baseband processor. In *Proc. of the 41st Annual International Symposium on Microarchitecture*, pages 152–163, Nov. 2008.

[111] M. Woh, Y. Lin, S. Seo, S. Mahlke, T. Mudge, C. Chakrabarti, R. Bruce, D. Kershaw, A. Reid, M. Wilder, and K. Flautner. From soda to scotch: The evolution of a

wireless baseband processor. *Proceedings. 41th Annual IEEE/ACM International Symposium on Microarchitecture, 2008. MICRO-41.*, pages 152–163, Nov. 2008.

[112] M. Woh, S. Seo, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. AnySP: Anytime Anywhere Anyway Signal Processing. In *Proc. of the 36th Annual International Symposium on Computer Architecture*, pages 128–139, June 2009.

[113] T. Yeh et al. ParallAX: an architecture for real-time physics. In *Proc. of the 34th Annual International Symposium on Computer Architecture*, pages 232–243, June 2007.

[114] S. Yehia, S. Girbal, H. Berry, and O. Temam. Reconciling specialization and flexibility through compound circuits. In *Proc. of the 15th International Symposium on High-Performance Computer Architecture*, pages 277–288, 2009.

[115] S. Yehia and O. Temam. From sequences of dependent instructions to functions: An approach for improving performance without ILP or speculation. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, pages 238–249, June 2004.

[116] J. Zhang et al. Stationary scanning x-ray source based on carbon nanotube field emitters. *Applied Physics Letters*, 86(18):184104, 2005.

[117] H. Zhu, Y. Chen, and X.-H. Sun. Timing local streams: improving timeliness in data prefetching. In *Proc. of the 2010 International Conference on Supercomputing*, pages 169–178, 2010.