

APPLICATION OPTIMIZED COMPUTING

A DISSERTATION  
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL  
ENGINEERING  
AND THE COMMITTEE ON GRADUATE STUDIES  
OF STANFORD UNIVERSITY  
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS  
FOR THE DEGREE OF  
DOCTOR OF PHILOSOPHY

Wajahat Qadeer

December 2013

© 2013 by Wajahat Qadeer. All Rights Reserved.

Re-distributed by Stanford University under license with the author.



This work is licensed under a Creative Commons Attribution-Noncommercial 3.0 United States License.

<http://creativecommons.org/licenses/by-nc/3.0/us/>

This dissertation is online at: <http://purl.stanford.edu/xz888kk6027>

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

**Mark Horowitz, Primary Adviser**

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

**Christoforos Kozyrakis**

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

**Stephen Richardson**

Approved for the Stanford University Committee on Graduate Studies.

**Patricia J. Gumport, Vice Provost for Graduate Education**

*This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.*

# Abstract

All computing systems are power limited, whether it is the 1W limit of a cell phone, or the 100W limit of a server. Triggered by the end of voltage scaling, the restricted power budgets are the result of energy per transistor switch scaling slower than the number of transistors on the chip. Since technology scaling no longer provides the energy savings it once did, to scale performance (operations/sec), we must improve the energy per operation by reducing the number of transistors involved in each operation. This fundamental change is forcing the design community to find new approaches to energy efficient computing.

Most designs use pre-designed processor based solutions because of their flexibility and availability. However, they are usually not the most energy efficient solutions. The need for better energy efficiency has pushed these systems to become multi-core. By reducing the peak performance of a processing core, the energy required per instruction can be reduced, either because the processor can be operated at a lower voltage, or because simpler core architectures can be used. Unfortunately, the efficiency gains provided by parallelism are finite and further gains will be difficult.

To better understand the potential of producing general-purpose chips with better efficiency, this thesis tries to analyze in detail the types of inefficiencies that exist in general-purpose systems — designs that can be outclassed by up to 3 orders of magnitude in both performance and energy-efficiency by ASIC designs. To collect this data, we classify applications using the dominant sources of energy: compute, control and memory. For compute and control bound applications we gather this data by first identifying the types and magnitudes of energy overheads that exist in a general-purpose Tensilica based extensible RISC chip multiprocessor (CMP) system and then



by exploring the architectural support and customizations needed to transform a general-purpose system to have the same energy efficiency as an ASIC.

Because the fundamental operations in compute bound applications are generally very low-power, amortization of overheads introduced by programmability requires execution of hundreds of these operations in one cycle. Interestingly, a high percentage of compute bound applications share common data-flow characteristics, which we exploit to create a flexible yet efficient domain specific processor, called the Convolution Engine. Although, control bound applications also operate on low-power control flow operations, sequential dependencies restrict the number of control flow operations fuseable into one instruction to between ten and fifteen. This restriction also defines the extent of achievable efficiency for control bound applications.

Unlike the low-power operations abundant in compute and control bound applications, the fundamental cost of a memory fetch is considerable. Improving the system efficiency of memory bound applications not only requires improving the efficiency of the processing elements, but also requires substantially increasing reuse in data fetches.

# Acknowledgement

I owe where I am today, in large part, to the help and encouragement of some of the marvelous people I have come to know at Stanford. First and foremost, I would like to thank Professor Mark Horowitz for his guidance, patience and encouragement. Not only is Mark a great advisor, more importantly, he is an amazing person and I cannot thank him enough for his support throughout my doctoral work.

I would like to extend my gratitude to my co-advisor Professor Christos Kozyrakis for teaching some of the most wonderful processor design courses, for sharing his vast knowledge with me and for always providing great feedback on my research. I would also like to thank Doctor Stephen Richardson for not only serving on my defense and reading committees, but also for being one of the most helpful and amazing people I know. I am also grateful to Professor Sanjay Lall for being the chair of my defense committee and Professor Subhashish Mitra for serving on my defense committee.

It's impossible to mention everyone that has helped me over the years, but I would like to extend a special thanks to my friend and colleague Rehan Hameed for his help and support throughout my stay at Stanford. Many thanks to Alex Solomatnikov, Amin Firoozshahian, Ofer Shacham, Omid Azizi, Megan Wachs, Andrew Danowitz, John Brunhaver and other members of the VLSI group for their support, guidance and assistance. I would also like to acknowledge the administrative staff, especially Teresa Lynn and Mary Jane Swenson, for their assistance with numerous miscellaneous issues.

My stay at Stanford would have been incomplete without my amazing friends at and near Stanford. I would like to thank them for always trying to keep me on track, for the long squash sessions and for the amazing photography trips.

Finally, I would like to add that I am eternally grateful to my parents, S. A. Qadeer and Rizwana Qadeer. If it had not been for their sacrifices, support and love, I would never have dreamed of being here. My greatest thanks to my wife, Tehmina, and my daughter, Zahra, for their love, support and encouragement and for always standing by me even in the most difficult of situations.

# Contents

<b>Abstract</b>	<b>iv</b>
<b>Acknowledgement</b>	<b>vi</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Background</b>	<b>7</b>
2.1 Embedded Low-Power Multiprocessor (ELM) . . . . .	9
2.2 SIMD Architectures and Vector Machines . . . . .	10
2.3 Extensible Processors . . . . .	11
2.4 Application Classes . . . . .	12
<b>3 Understanding Sources of Inefficiency in Compute and Control</b>	
<b>Bound Applications</b>	<b>13</b>
3.1 H.264 . . . . .	15
3.1.1 Motion Estimation . . . . .	16
3.1.2 Intra-Prediction, Discrete Cosine Transform and Quantization . . . . .	21
3.1.3 CABAC . . . . .	21
3.2 Experimental Methodology . . . . .	25
3.3 SIMD and Operation Fusion . . . . .	29
3.4 Custom Instructions for Compute Bound Applications . . . . .	32
3.4.1 IME Strategy . . . . .	33

3.4.2	FME Strategy . . . . .	34
3.5	Custom Instructions for Control Bound	
	Applications . . . . .	36
3.5.1	CABAC Strategy . . . . .	37
3.6	Custom Instructions Area Comparison . . . . .	39
3.7	Custom Instructions Summary . . . . .	40
3.8	Conclusion . . . . .	41
<b>4</b>	<b>Convolution Engine: Balancing Efficiency and Flexibility in Specialized Computing</b>	<b>43</b>
4.1	Computational Models . . . . .	45
4.2	Applications . . . . .	47
4.2.1	Motion Estimation . . . . .	47
4.2.2	Scale Invariant Feature Transform (SIFT) . . . . .	48
4.2.3	Demosaic . . . . .	51
4.2.4	Mapping to “Map” and “Reduce” Abstraction . . . . .	54
4.3	Convolution Engine . . . . .	55
4.3.1	Register Files and the Load/Store Unit . . . . .	60
4.3.2	Map & Reduce Logic . . . . .	61
4.3.3	Instruction Graph Fusion . . . . .	62
4.3.4	Lightweight SIMD . . . . .	65
4.3.5	Custom Functional Units . . . . .	66
4.3.6	A 2-D Filter Example . . . . .	66
4.3.7	Resource Sizing . . . . .	67
4.3.8	Convolution Engine CMP . . . . .	69
4.3.9	Programming the Convolution Engine . . . . .	70
4.4	Evaluation Methodology . . . . .	73
4.4.1	H.264 Motion Estimation . . . . .	75
4.4.2	SIFT . . . . .	75
4.4.3	Demosaic . . . . .	76
4.5	Results . . . . .	76

4.5.1	Generating Instances with Varying Degrees of Flexibility . . . . .	79
4.6	Conclusion . . . . .	82
<b>5</b>	<b>Analysis of Memory Bound Applications</b>	<b>84</b>
5.1	Speech Recognition . . . . .	85
5.1.1	CMU Sphinx Speech Recognition . . . . .	86
5.2	Baseline Speech Recognition System . . . . .	89
5.2.1	Pruning Stage . . . . .	90
5.2.2	Patch List . . . . .	90
5.2.3	CMP System . . . . .	90
5.3	Experimental Methodology . . . . .	92
5.3.1	Processor Optimization Strategy . . . . .	93
5.3.2	Memory Optimization Strategy . . . . .	93
5.4	Results . . . . .	95
5.4.1	Processor Optimization Results . . . . .	96
5.4.2	Memory Optimization Results . . . . .	98
5.5	Conclusion . . . . .	102
<b>6</b>	<b>Conclusion</b>	<b>104</b>
	<b>Bibliography</b>	<b>107</b>

# List of Tables

3.1	Description of VLIW and SIMD resources employed for each sub-algorithm. . . . .	29
4.1	Mapping kernels to map and reduce abstraction. . . . .	55
4.2	Sizes for various resources in the convolution engine. . . . .	68
4.3	Energy required for filtering instructions using 32, 64 and 128 ALUs. . . . .	68
4.4	Convolution engine instructions. . . . .	71
5.1	Memory energy consumption at various levels of hierarchy. . . . .	84
5.2	Last level cache misses for the major consumers of memory energy before memory optimization. . . . .	94
5.3	Last level cache misses for the major consumers of memory energy after memory optimizations. . . . .	101

# List of Figures

1.1	Historic microprocessor power consumption statistics. . . . .	2
1.2	Historic microprocessor power per SPEC (Standard Performance Evaluation Corp.) benchmark score vs. performance statistics. . . . .	3
2.1	Energy breakdown for a high definition H.264 encoder using a Tensilica RISC processor. . . . .	8
3.1	Comparison of functional unit energy with that of a typical RISC instruction in 90nm. . . . .	14
3.2	Supported macro-blocks in H.264. . . . .	16
3.3	H.264 Integer Motion Estimation Search Procedure. . . . .	17
3.4	FME search locations. . . . .	19
3.5	Luma intra-prediction modes for a 4x4 block. . . . .	20
3.6	Illustration of binarization procedure for CABAC. . . . .	22
3.7	CABAC context modeling example. . . . .	23
3.8	Binary arithmetic coder encoding flow. . . . .	24
3.9	Four stage macroblock partition of H.264. . . . .	25
3.10	Performance and energy gap of base CMP when compared to an equivalent ASIC. . . . .	26
3.11	H.264 energy distribution over different sub-algorithms. . . . .	27
3.12	H.264 processor energy breakdown for base CMP. . . . .	28
3.13	Comparison of energy consumption at different levels of optimization for each sub-algorithm of H.264. . . . .	29



3.14	Speedup comparison at different levels of optimization for each sub-algorithm of H.264. . . . .	30
3.15	Processor energy breakdown for H.264 at each level of optimization. .	31
3.16	Custom storage and compute for IME 4x4 SAD. . . . .	33
3.17	FME upsampling unit. . . . .	35
3.18	A simplified version of a control bound loop from BAC. . . . .	37
3.19	CABAC Arithmetic Encoding Loop. . . . .	38
3.20	Area comparison of “magic” instructions with the simple Tensilica RISC processor for IME, FME, IP and CABAC. . . . .	39
3.21	Comparison of area efficiency ( $FPS/mm^2$ ) for IME, FME IP and CABAC. . . . .	40
4.1	Constraints a flexible engine must satisfy to remain efficient. . . . .	44
4.2	Fusion of individual operations in a super instruction by the generalized reduction stage. . . . .	47
4.3	SIFT difference of Gaussian. . . . .	49
4.4	SIFT Extrema detection. . . . .	50
4.5	Luminance interpolation in Demosaic. . . . .	52
4.6	Chromiance interpolation in Demosaic. . . . .	53
4.7	SIMD implementation of generic 1D convolution. . . . .	55
4.8	128-bit SIMD implementation of a 16-tap 1D horizontal convolution. .	56
4.9	Shift register based implementation of 16-tap 1D horizontal convolution	57
4.10	GPU energy breakdown for SAD. . . . .	58
4.11	Block diagram of the convolution engine. . . . .	59
4.12	Complex graph fusion unit. . . . .	63
4.13	Instruction graph fusion unit. . . . .	64
4.14	Execution of a 4x4 2D Filter on the convolution engine. . . . .	66
4.15	Communication connections among components of convolution engine CMP. . . . .	69
4.16	Mapping of applications to convolution engine CMP. . . . .	74

4.17	Energy consumption comparison among convolution engine, custom cores and SIMD. . . . .	77
4.18	Comparison of operations per $mm^2$ among convolution engine, custom cores and SIMD. . . . .	78
4.19	Change in energy consumption as programmability is incrementally added to the core. . . . .	82
4.20	Increase in area as programmability is incrementally added to the core.	82
5.1	Four state HMM model used for representing tri-phones in Sphinx 3.0.	86
5.2	Four stage HMM level partition of Sphinx 3.0. . . . .	91
5.3	Energy distribution between the processing elements and the memory for the base CMP system. . . . .	92
5.4	Improvement in performance over the base system that was operating at real-time after the application of custom instructions. . . . .	96
5.5	Improvement in energy per frame over the base system after the application of custom instructions. . . . .	96
5.6	Energy distribution between the processing elements and the memory after processor customizations. . . . .	97
5.7	The plot captures the total size of the HMM data structure in the current frame on the x-axis and the relative frequency a structure of this size is likely to occur in a given frame on the y-axis. . . . .	98
5.8	Relative frequency of a given amount of memory needed to store Patch List entries in the current frame. . . . .	99
5.9	Relative frequency of a given amount of memory needed to store Tri-gram word candidates in the current frame. . . . .	100
5.10	Relative frequency of a given amount of memory needed to store Bi-gram word candidates in the current frame. . . . .	101
5.11	Energy distribution between the processing elements and the memory for the optimized CMP system. . . . .	102

# Chapter 1

## Introduction

Over the past two decades, chip designers have leveraged technology scaling and rising power budgets to rapidly scale performance. Aided by Dennard's constant field scaling theory [1] chipmakers achieved a triple benefit: each generation supplied more gates per  $mm^2$ , gate delay decreased, and energy per gate switch decreased. Thus scaling alone brought about a significant growth in computing performance while maintaining a constant power profile. However, as shown in Figure 1.1, power and power density continued to increase, which caused a dramatic increase in processor power over the past 20 years. The reason was a combination of designers not following constant field scaling exactly and creating more aggressive designs, thereby increasing performance more quickly than Dennard predicted.

Unfortunately, rising costs of cooling high performance designs exacerbated by the changes to technology scaling beyond 90nm severely compromised our ability to keep power in check rendering most chips power limited. Consequently, almost all systems designed today, from high-performance servers to wireless sensors, are becoming energy constrained. In this power-constrained, post-Dennard era, creating energy-efficient designs is critical. Continually increasing performance in this new era requires lower energy per operation, because the product of operations per second (performance) and energy per operation is power, which is constrained.

Years of research have taught us that the best — and perhaps only — way to save energy is to cut waste. Clearly, the first step is to reduce waste in the design. Clock

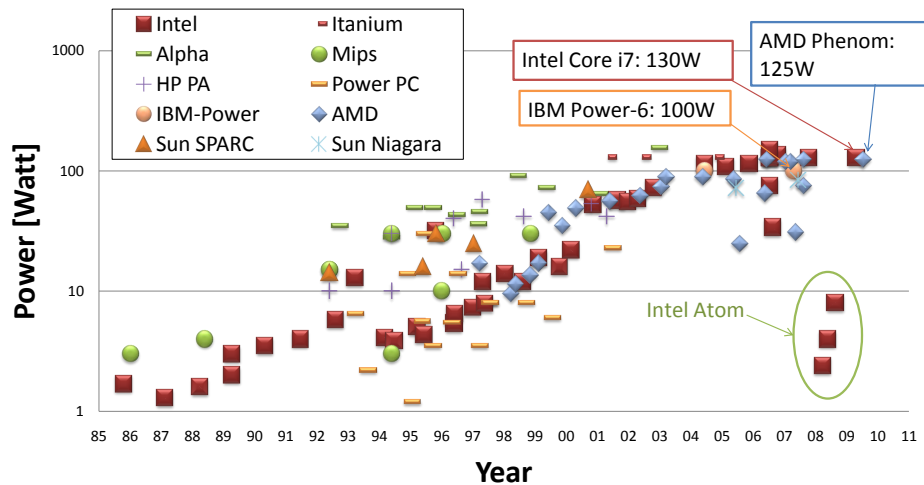


Figure 1.1: Historic microprocessor power consumption statistics. Power consumption has increased by over two orders of magnitude in the past two decades. However, as evidenced by the Intel Atom, there is a recent trend of lower power processors for the growing market of battery-operated devices [2]. (PA: Precision Architecture.)

gating prevents a logic block’s gates from switching during cycles when their output isn’t used, reducing dynamic energy with virtually no performance loss. Power gating goes even further by shutting off power to an entire block when it’s unused for longer periods of time, reducing idle leakage power, again at low performance costs.

However, power is also wasted indirectly when we waste performance. Higher performance requirements necessitate higher-energy operations, so removing performance waste reduces energy per operation. Using multiple simpler units rather than a single aggressive one, therefore, saves energy when processing parallel tasks. At the system level, this observation is driving the recent push for parallel computing. Backing off from peak performance to create simpler cores enables considerable reductions in energy per operation as shown in Figure 1.2. Although this also harms performance, we can reclaim this lost performance through additional cores at a far lower energy cost. Of course, this approach sacrifices single-threaded performance, and it also assumes that the application is parallel, which isn’t always true. However, given the power constraints, this move to parallelization was a trade-off that industry had to make.

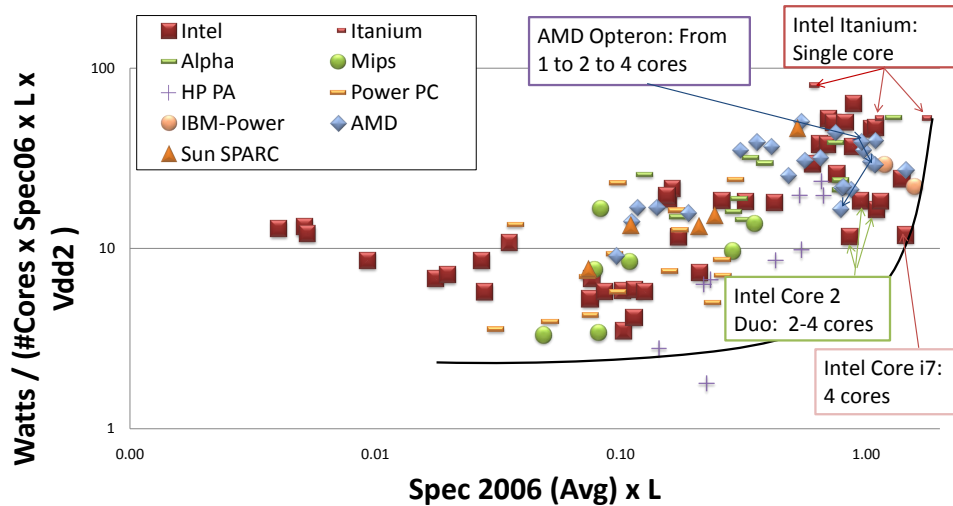


Figure 1.2: Historic microprocessor power per SPEC (Standard Performance Evaluation Corp.) benchmark score vs. performance statistics. Performance numbers (x-axis) are the average of the single-threaded SPECint2006 and SPECfp2006 results [3]. To account for technology generation, we normalized the numbers according to feature size  $L$ , which is inversely proportional to the inherent technology speed. The y-axis shows power per SPEC score, which is a measure of energy per operation. Energy numbers are normalized to the number of cores per chip and to the technology generation; since  $E = CV^2$  (where  $E$  is energy, and  $C$  is the capacitance),  $E$  is proportional to  $LV^2$ . Note how the move to multicore architectures typically sacrifices single-threaded performance, backing off from the steep part of the curve [2].

Unfortunately, though, we can't rely on parallelism to save us in the long term, for two reasons. First, as Amdahl noted in 1965, with extensive parallelization, serial code and communication bottlenecks rapidly begin to dominate execution time. Thus, the marginal energy cost of increasing performance through parallelism increases with the number of processors, and will start increasing the overall energy per operation. The second issue is that parallelism itself doesn't intrinsically lower energy per operation; lower energy is possible only if sacrificing performance also yields a lower energy design. Unfortunately, this trade-off follows the law of diminishing returns. After we back away from high-power designs, the remaining savings are modest.

Looking forward, the best tool in our power-saving arsenal is customization, because the most effective way to reduce waste is to find a solution that accomplishes the same task with less work. By tailoring hardware to a specific application, customization not only results in energy savings by requiring less work but also improves performance, allowing an even greater reduction of the required energy. The idea of specialization is well-known, and is already applied in varying degrees today. The use of single instruction, multiple data (SIMD) units (such as Intel's streaming SIMD extension (SSE), vector machines, or graphics processing units) as accelerators is an example of using special-purpose units to achieve higher performance and lower energy [4]. To estimate the potential gain possible through customization we execute a single video compression application, a 720p high definition H.264 encoder, on a Tensilica RISC processor. A staggering 500x difference in energy dissipation per frame is observed between the general-purpose (GP) software based implementation compared to a typical ASIC designed for real-time 720p H.264 encoding [5]. The huge difference is the consequence of the efficiency overheads that exist in a general-purpose (GP) processor due to flexible instructions and data fetch.

However, despite the clear energy efficiency advantage of ASICs, the number of new ASICs built today is not skyrocketing but decreasing, because designing them is too expensive. The design and verification cost for a state-of-the-art ASIC today is well over \$20 million, and the total NRE costs are more than twice that, owing to the custom software required for these custom chips [6][7]. Interestingly, fabrication costs, though very high, account for only roughly 10 percent of the total cost today [6]. That means high design, verification, and software costs are the primary reasons why the number of ASICs being produced today is actually decreasing [7], even though they're the most energy-efficient solution.

Thus, despite the fact that multiprocessors lag far behind ASICs in terms of efficiency, their flexibility enables them to amortize high NRE costs across many applications. However, the desire to achieve ASIC-like compute efficiencies with multiprocessor-like application development costs remains strong and is pushing designers to explore new design methodologies and programmable development platforms. But to fully realize the potential of producing general-purpose chips with

better efficiency, we need to fundamentally understand why specialized hardware is more efficient compared to general-purpose systems and how we can leverage that knowledge to build efficient programmable units. Additionally, we need to identify if the huge efficiency gap between ASICs and programmable units is limited to a few applications. If not then what sets the limit? And finally we need to determine how general we can make the flexible core while still maintaining a high level of efficiency.

Identification of sources of inefficiency in general-purpose systems requires first understanding the limitations of existing system optimization techniques across a broad spectrum of applications and determining how well they compare to custom hardware. We review the limitations of the existing low-power architectural techniques in Chapter 2 and present our strategy for detecting the fundamental shortcomings of current programmable systems. Before introducing our energy optimization techniques for the applications that we use in our study, we describe in the same chapter how we classify applications using the dominant consumers of energy: compute, control and memory.

Chapter 3 then looks in more depth at understanding the sources of inefficiency in compute and control bound applications with the help of a 720p HD H.264 encoder. We choose H.264 because it demonstrates the large energy advantage of ASIC solutions (500x) and because there exist commercial ASICs that can serve as a benchmark. Moreover, H.264 contains a variety of computational motifs, from highly compute bound (motion estimation) to control intensive ones (Context Adaptive Binary Arithmetic Coding [CABAC]). Aided by the benchmark ASIC, we analyze the source of the energy overhead in H.264's compute and control bound algorithms, and the rough strategy that we need to use to reduce it.

For compute bound applications, the problem is that the very low per-cycle energy cost of the fundamental operations means that the relative overhead from other sources such as instruction fetch becomes prohibitively large. Since it is generally not possible to reduce these overheads, the only way to get high efficiency is to perform 100s of fundamental operations in each cycle. This then makes the amortized overhead per operation small enough to yield efficient computation. Although, control bound applications also benefit from the aggregation of low-power control flow instructions

into a single instruction, it's generally not possible to fuse together more than 10–20 operations; therefore, the efficiency gains are limited.

Because compute bound applications operate on short data and generally possess enough data-parallelism to enable 100s of operations per cycle, our efficiency analysis reveals that they offer the highest efficiency gains. Fortunately, compute bound applications must share a number of characteristics to be compute bound and in Chapter 4 we exploit those features to develop a specialized processor that offers high efficiency while still retaining sufficient programmability. This flexible engine, which we refer to as the Convolution Engine, works for a reasonable percentage of compute bound applications within the domain of imaging and video systems.

After studying efficiency trade-offs between flexibility and customization for compute and control intensive applications, we turn our attention to memory bound applications. In this regard, we analyze Speech Recognition and present our strategies for improving energy efficiency in Chapter 5. We learn that the situation for memory bound applications is fundamentally different, since the energy cost of a memory fetch is significant. While efficiency can be gained by reducing non-memory overhead through parallelism, this will only give a small (3x) energy saving. Instead, efficient hardware solutions use both parallelism and other compute optimization to reduce the compute power, along with exploitation of application characteristics to increase locality in the data fetches. This increased reuse often enables the use of better utilization of smaller and closer on-chip memories, which have much lower energy costs compared to DRAM.



# Chapter 2

## Background

The main benefit that general-purpose (GP) processors possess over ASICs is programmability, but therein lie most of the overheads. Identification of processor inefficiencies stemming from these overheads requires determination of the major consumers of energy in a processor pipeline. With the help of a simple RISC pipeline, shown in Figure 2.1, this chapter highlights major consumers of energy in a processor pipeline, and then provides a brief overview of prior optimization work.

Figure 2.1 depicts the inner workings of a simple RISC pipeline. The pipeline works by fetching an instruction from the instruction cache, which is then transferred to the instruction decode unit. The register file is accessed in parallel with the decode unit and the fetched data is then processed by the arithmetic execution stage. The memory execution stage follows the arithmetic unit and if the fetched instruction requires a memory access, an interaction with the data cache takes place here. And in the end we have the write-back stage.

Execution of a high definition 720p H.264 encoder on a Tensilica RISC processor with a 7-stage pipeline not much different from the one described above, we get an energy breakdown presented in the pie-chart in Figure 2.1. The pie chart reveals that the instruction fetch and decode units (IF) consume the most energy. The energy dissipated in the data cache (D-\$) comes in second followed by the pipeline registers. Interestingly, the arithmetic unit that performs the useful work takes just 6% of the total energy meaning that the rest is all overhead. Yet there must be more to this

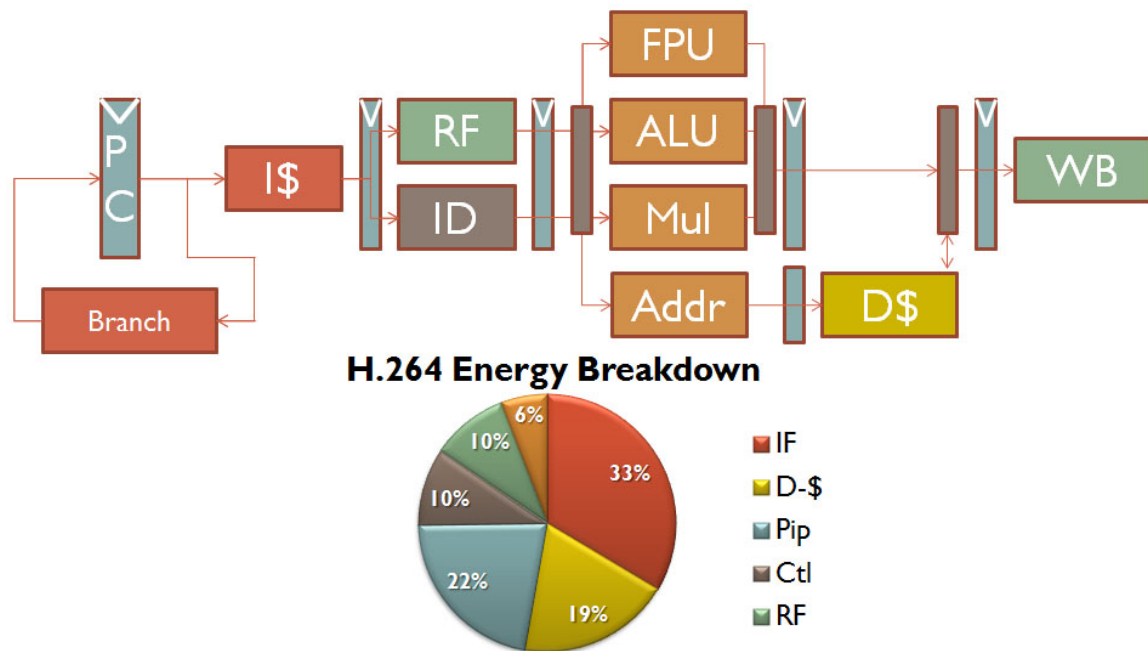


Figure 2.1: A simple RISC pipeline is presented at the top and the pie-chart represents the energy breakdown obtained after the execution of a high definition H.264 encoder on a Tensilica core with a 7-stage pipeline. In the pie-chart IF represents instruction fetch and decode energy; D-\$ stands for data-cache energy; Pip stands for energy dissipated in the pipeline registers; energy consumed by the control logic is represented by Ctl; and RF represents the energy dissipated in the register file.

story because if we optimize away the whole overhead we get an improvement of just 20x, not the 500x mentioned in Chapter 1.

Optimization strategies commonly engaged to improve the efficiency of GP processors attempt to reduce or amortize overheads associated with programmability over a larger number of operations. In the first group lie the architectural techniques ranging from instruction registers to hierarchical register files which attempt to reduce the energy consumption of the major overheads. Traditional Vector Processors are a good example of the second group: they use a single instruction to operate on multiple operands [8].

The other approach to efficient computing drops the processor framework, or relegates the processor to handling the UI, and some control, and builds a customized

engine for each application. However, as explained in Chapter 1, this requires an enormous design effort, and is becoming hard to justify except for very high volume applications. Somewhere between these two extremes lie extensible processors, which we describe in Section 2.3. These processors provide the framework of a CPU, but allow the user to add his/her own customized instructions. This thesis relies on the Tensilica extensible processor system to generate the data for the experiments presented in later chapters. The next section looks at these optimization techniques in detail, and briefly reviews the contemporary implementations of these ideas.

## 2.1 Embedded Low-Power Multiprocessor (ELM)

The low-power ELM processor [9][10] aims to bridge the enormous efficiency gap between ASICs and GP processors. The design is influenced by the observation that computation doesn't limit the efficiency of embedded processors, instead the burden falls on the inefficient delivery of instructions and data to the functional units. ELM features several innovative mechanisms for improving locality through carefully sized and strategically placed storage structures resulting in enhanced instruction issue and data supply efficiency [9][10].

As a part of the strategy to improve locality, ELM introduces software managed instruction registers placed in close proximity to the functional units. These registers eliminate costly instruction cache accesses by storing the critical code close to the functional units to reduce the fetch energy. In addition to the instruction registers, small operand registers are also added close to the functional units. These registers capture transient data, eliminating the penalty for accessing the big monolithic register file. The big register file also gets its share of enhancements in the form of indexed registers. With the help of register pointers, this scheme eliminates overhead instructions prevalent in compute-intensive applications that benefit from data-reuse in the register files. All these optimizations combine to substantially simplify the design and deliver extremely high efficiency. They estimate an energy per operation of 10pJ in a 45nm technology for short integer data. While this is 10x better than a RISC processor, it is comparable to what SIMD machines can achieve, and still about

10x worse than ASICs.

## 2.2 SIMD Architectures and Vector Machines

Single Instruction Multiple Data (SIMD) is frequently employed in parallel processors to control multiple functional units all performing the same operation on different data. Because SIMD units operate on multiple operands in parallel, appropriately designed SIMD units have a profound effect on the aggregate throughput and efficiency of data-parallel applications. The improvement in efficiency is caused by the amortization of overheads inherent in instruction fetch, decode and pipeline registers over a large number of functional operands. However, the usefulness of SIMD units wanes if the application possesses limited parallelism or is not able to aggregate enough parallel operands to fully utilize the SIMD array. The need to match the SIMD width limits the usefulness of a large SIMD array for many applications. While too wide a width results in waste especially in terms of area and leakage, too small a width entails inefficient exploitation of parallelism.

Large SIMD and vector processors are effective for applications with large data-parallelism [11][12]. To bring the efficiency and performance benefits of SIMD processing to mainstream applications, many contemporary processors have added smaller SIMD units to their pipelines. One of the well-known examples is Intel's streaming SIMD extension (SSE) [13][14] which in addition to operating on multiple parallel data elements, also offer high efficiency complex instructions. The complex instructions are essentially fused data-flow graphs consisting of multiple instructions fused into one operation, further improving the efficiency of SIMD units. As we will see in the next chapter, leveraging these units can also improve energy efficiency for data-parallel applications by upto 10x.

GPUs are generally considered to be the most successful SIMD machines today [11]. They contain a large number of SIMD cores, each running 8–32 data operations in parallel and are optimized for massively data parallel graphics applications. They employ a large number of low control overhead cores, sacrificing single thread performance in favor of very high throughput. To further improve throughput they

host a large numbers of threads on each core and threads switch on high memory latency operations. By specializing for their workload characteristic, GPUs are able to get much higher throughput performance and lower energy per operation than general purpose processors. However, these machines are designed for performance and not energy efficiency and many of the features especially in the memory system are not low-energy. To support multiple independent threads, the memory system is designed with a minimal focus on locality resulting in high energy dissipation in the register file and memory system making GPUs unfeasible for our applications. Thus, the resulting energy efficiency is only modestly better than a CPU [15].

## 2.3 Extensible Processors

Extensible processors such as Tensilica’s Xtensa provide a base design and instruction set which the designer can extend with custom instructions and datapath units, based on application requirements [16]. Extending the ISA for a given application can be done either manually or using automated tools. Tensilica provides an automated ISA extension tool [17] which has been shown to achieve speedups of 1.2x to 30x for EEMBC benchmarks [18] and signal processing algorithms [19]. Other tools have similarly demonstrated significant gains from automated ISA extension [20][21]. While automatic ISA extension can be very effective, manual creation of custom ISA extensions allows for even higher gains. In various app notes [22][23][24], Tensilica reports speedups of 40x to 300x for kernels such as FFT, AES encryption and DES encryption. We will use this system for the experimental data presented later in the thesis.

To determine the nature and degree of customization which should be supported by future programmable systems, we examine the sources of inefficiency present in a programmable CMP platform and explore which aspects of customization are critical for performance enhancements. We use the approach of customizable processors to incrementally transform a generic CMP into a specialized energy efficient design, to better understand the cost/benefit of different specializations.

## 2.4 Application Classes

The dominant energy consumer in an application governs that application’s optimization strategy. If the leading energy consumers are arithmetic operations, the optimization strategy needs to facilitate parallel operation of functional units with minimal data-memory fetches; however, if fetches to memory are abundant, enhancing data localization may present itself as the main challenge. Identification of these consumers requires sifting through an application’s instruction energy profile, pruning non-critical operations and grouping the remaining operations as being compute, control or memory. In this thesis we perform this step with the help of Tensilica’s XEnergy simulation environment that uses heuristics based upon a database of empirical data and the energy estimates are reported to be within 30% of synthesized RTL [25]. Aided by these energy simulations, we identify leading energy consumers within an application and classify that application as being compute bound, memory bound, or control bound. Despite the simplicity of the process, complications arise when an application exhibits multiple dominant consumers. In this case, we draw inspiration from Amdahl’s law and associate the application with the energy consumer that presents itself as the bottleneck in the optimization process.

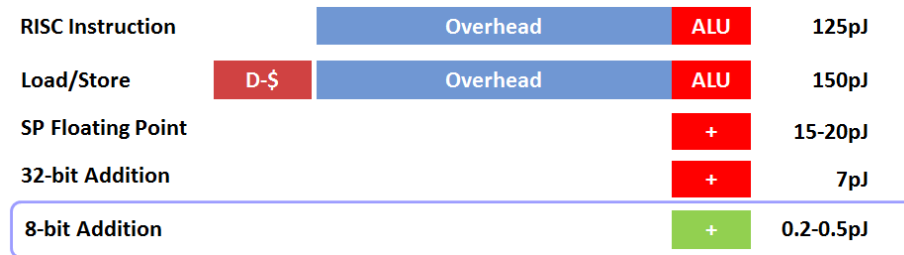
This classification into compute, control or memory bound applications permits us to explore optimization strategies specifically beneficial for each class of applications, which appreciably enhances their impact. Furthermore, this categorization also enhances our ability to identify algorithmic patterns common across a wide range of applications within each domain, which we can exploit to build efficient yet programmable hardware. We explore this issue further in the next two chapters.

## Chapter 3

# Understanding Sources of Inefficiency in Compute and Control Bound Applications

Compute bound applications are highly arithmetic intensive and perform a considerable amount of arithmetic operations. Unaffected by sequential dependencies, these applications possess a high degree of data-level parallelism permitting them to operate on hundreds of data elements in parallel. A high level of data locality is vital to keep the compute units occupied without frequently accessing the memory, otherwise the application risks becoming memory bound. Thus, for an application to remain compute bound and still meet the data supply needs of the functional units, it must generate little memory traffic and perform a substantial number of operations per memory fetch. Some typical examples of compute intensive imaging applications include H.264 motion estimation, Scale Invariant Feature Transform (SIFT)[26], Demosaic[27], etc.

Performance of compute bound applications responds well to multiprocessor optimization techniques such as single processor multiple data (SPMD). However, energy



**To get more than two orders of magnitude gain in efficiency**



Figure 3.1: Comparison of functional unit energy with that of a typical RISC instruction in 90nm. Strategy for amortizing processor overheads includes executing hundreds of low-power operations per instruction.

gains of general multiprocessing are modest, especially compared with other generally used data-parallel optimization techniques such as SIMD. Widely regarded as the most efficient general-purpose optimization for compute intensive applications, SIMD units are commonly added as datapath extensions to processing cores and their sizes range from ten to twenty elements. Because SIMD units can simultaneously execute tens of data operands in a single cycle, they are able to amortize processor overheads across multiple data elements resulting in an order of magnitude better efficiency compared to SPMD cores. However, the resulting efficiency is still 50x off our goal.

Although, scalability issues of SIMD units restrict their efficiency gains, the reason for the lower efficiency can be explained with the help of Figure 3.1, which compares the energy dissipation of various arithmetic operations with the instruction energy of an extremely simple RISC processor. The energy dissipation of arithmetic operations that perform the useful work in a computation remains much lower than the energy wasted in the instruction overheads. The crux of the problem is that the compute intensive applications such as H.264 typically operate on short data requiring just 0.2-0.5pJ (90nm) of energy per operation, which is even an order of magnitude lower than that of a 32-bit RISC ALU whose energy dissipation stands at 7pJ (90nm) shown in Figure 3.1.

Because ASICs perform just the basic low-energy operations and throw away



all the processor overheads including those that reside inside the ALU, they can eclipse general-purpose processors by as much as three orders of magnitude in efficiency. Thus, to achieve higher efficiency, a general-purpose processor not only needs to amortize overheads that exist inside a processor by performing hundreds of low-energy operations in a cycle, it must also waste almost no energy on memory accesses which stands at 25pJ (90nm) per memory fetch. However, the situation changes dramatically if the basic arithmetic operation in an application requires higher energy such as a floating point operation. In such a case, amortization of processor overheads can be achieved even by executing tens of operations in parallel instead of hundreds, though the resulting efficiency also remains limited to an order of magnitude.

To better understand the potential of creating general-purpose (GP) solutions for compute bound applications, we investigate the sources of overhead in GP systems using a single video compression application, 720p HD H.264 video encode. To build this understanding, we start with a simple RISC chip multiprocessor (CMP) and incrementally transform it to an optimized multiprocessor with specialized hardware units. On this task, a general-purpose software solution takes 500x more energy per frame and 500x more area than an ASIC [5] to reach the same performance. While memory is never an issue in this application, one component, Context Adaptive Binary Arithmetic Coding (CABAC), is control dominated, and after the compute has been optimized, limits both the energy and the performance of the application. We then explore how to reduce the overhead of control bound applications.

### 3.1 H.264

To understand how we customize a generic CMP to efficiently implement H.264, we must first understand the basic components of the H.264 algorithm. Three major functions comprise more than 99% of the total execution time in our base CMP implementation: *a*) Motion Estimation comprised of Integer Motion Estimation (IME) and Fractional Motion Estimation (FME) *b*) Intra Prediction (IP), Discrete Cosine Transform and Quantization (DCT/Quant) and *c*) Context Adaptive Binary Arithmetic Coding (CABAC). We implement the H.264 baseline profile at level 3.1; however, we

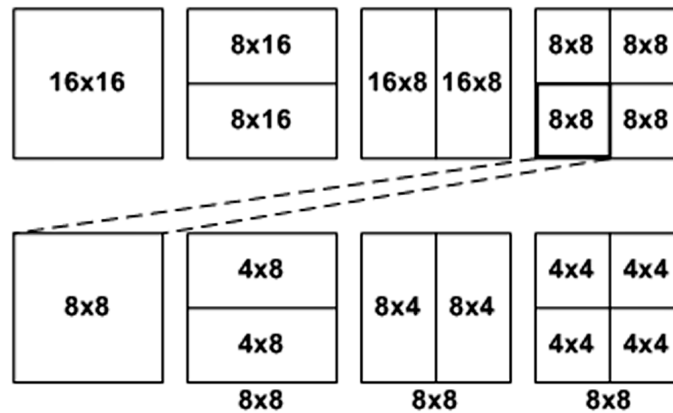


Figure 3.2: H.264 supports forty one different macro-block types starting from 16x16 going all the way down to 4x4 and motion estimation is performed separately for each one of these blocks.

use CABAC in place of CAVLC because CABAC is more complex and more challenging to improve [28][29]. CABAC is also more representative of advanced coding steps in other applications

### 3.1.1 Motion Estimation

A digital video stream consists of a sequence of frames that are closely spaced in time. Because of this proximity, the frames contain a significant amount of redundant information which if not removed, wastes storage as well as precious bandwidth when the stream is transmitted over a network. Motion estimation (ME) eliminates this redundancy by identifying the regions of least change between the current frame and its neighbors using a process known as *block matching*. This procedure involves comparing blocks of pixels in the current frame called *macro-blocks* against expanded regions of pixels in the adjacent frames called *search windows* to find a match. These matched blocks enable the transmission of only the difference between frames; thus, conserving storage as well as network bandwidth.

ME is widely regarded as a key component of many modern day video codecs such as H.264 because it sets the compression efficiency of the video stream. In H.264 ME facilitates high compression efficiency by sub-dividing the current macro-block, a two

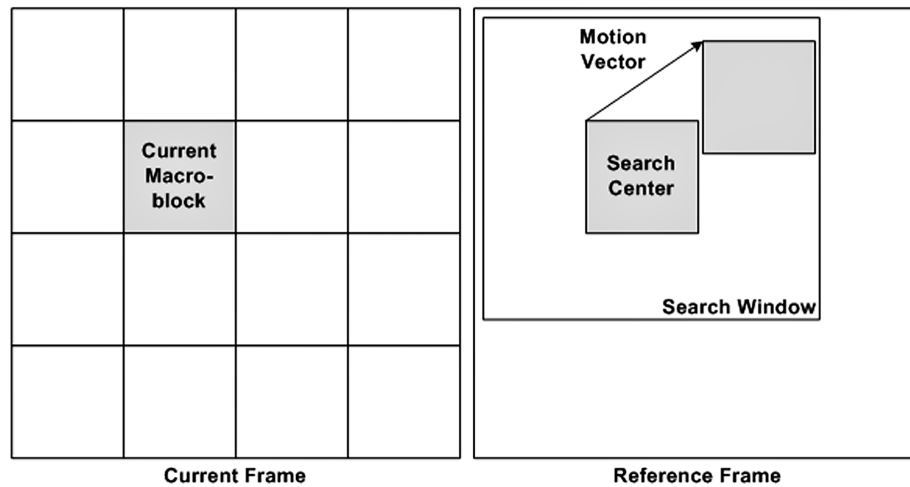


Figure 3.3: H.264 Integer Motion Estimation Search Procedure.

dimensional block of 16x16 pixels, into smaller sized regions ranging from 4x4 pixels to 16x16 pixels as shown in Figure 3.2. An improved matching accuracy ensues when a match is sought for each one of these blocks of pixels within the search window. However, this improvement in matching accuracy comes at the cost of computational intensity which increases many fold. Thus, it should come as no surprise that ME accounts for 90% of the execution time when a high definition H.264 encoder is executed in software using JM8.6 reference code [30]. For this implementation we use the Fast Full Search, which will be explained later.

While ME is highly computationally intensive, a small ASIC (in  $mm^2$ ) can easily satisfy the stringent performance and energy requirements of motion estimation; however, it poses a significant challenge for general purpose processors. The energy efficiency of general-purpose processors for H.264 motion estimation tends to be over two orders of magnitude worse compared to an ASIC. To determine the reason behind this dismal display of efficiency from general purpose hardware, we must first understand how ME works. The computation of ME takes place in two steps: it starts with integer motion estimation (IME), which is responsible for finding the match between the current frame and its neighbors, and ends with fractional motion estimation which takes the match from IME and further refines it by searching at sub-pixel granularity.

### Integer Motion Estimation (IME)

As has been stated above, IME forms the first step in the block matching process. The algorithm that yields the best search results in H.264 is called Full Search (FS). In FS the current frame is divided into 16x16 sized macro-blocks and for each macro-block and its sub-blocks presented in Figure 3.2, a block match is obtained in the reference frame using the process illustrated in Figure 3.3. Aided by motion displacements of already matched macro-blocks, a prediction of the location of the search window is made inside the reference frame. All the pixel locations within the search window are evaluated for a match using Sum of Absolute Difference (SAD) as the cost function shown in Equation (3.1). Motion Vectors (MV) are used to depict the motion between the matching blocks. These motion vectors are transmitted to the decoder along with the residual difference between the matched blocks.

$$SAD = \sum \sum abs(reference - current) \quad (3.1)$$

Although, FS yields the best search results, it performs a significant amount of redundant calculations by searching the whole window for each sub-block. Fast Full Search (FFS) eliminates these excess calculations by reusing SAD results of smaller blocks to form matches for their larger counterparts in addition to introducing early termination of unfeasible search locations. Despite these modifications that aid in reducing the computational complexity of FS, IME still takes up 56% of the total H.264 encoder execution time and 52% of total energy. Although, numerous fast variants of block matching have been proposed which require less computation[31][32][33], they are sub-optimal and sacrifice search quality by searching fewer locations. Furthermore, some of these algorithms are less suitable for a hardware implementation because they contain a non-deterministic number of pixel matching iterations, irregular workload distribution between iterations and the presence of branch instructions.

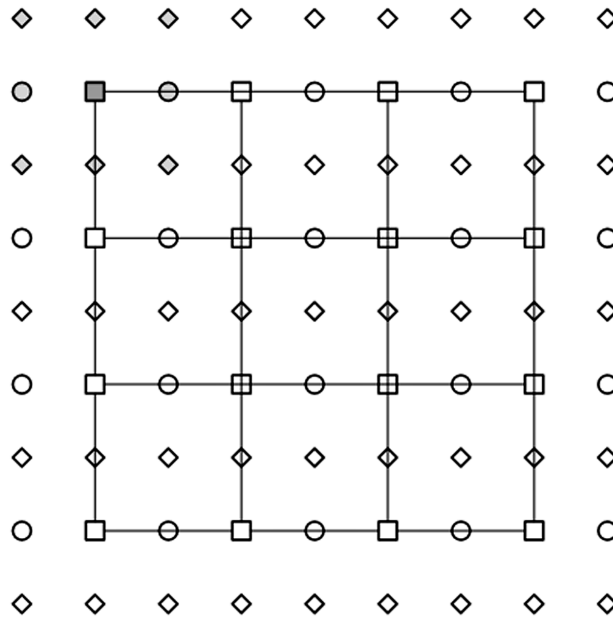


Figure 3.4: The squares represent IME’s best match for a 4x4 block. The greyed out locations at the top left represent the nine locations searched in FME to match the top left pixel.

### Fractional Motion Estimation (FME)

While IME performs motion estimation at integer pixel granularity, it is possible that motion may be better represented at the sub-pixel level. The task of Fractional Motion Estimation (FME) is to refine the match obtained by IME to finer granularities. Consider Figure 3.4 which illustrates the first step of the process. In this step, the pixels in the reference frame adjacent to the IME match are up-sampled to a half pixel granularity using Equation (3.2). Similar to IME, FME also subtracts the upsampled reference pixels from the pixels of the current macro-block; but, to better approximate the processing of the preceding H.264 pipeline, FME performs matching using a modified SAD algorithm called Sum of Absolute Transformed Difference (SATD). In this algorithm a Hadamard Transform is performed on the residues to approximate the effects of the Discrete Cosine Transform (DCT) which resides further down the pipeline — DCT is tasked with zeroing out as many residues as possible without compromising fidelity. Using SATD as the cost metric, nine locations including the

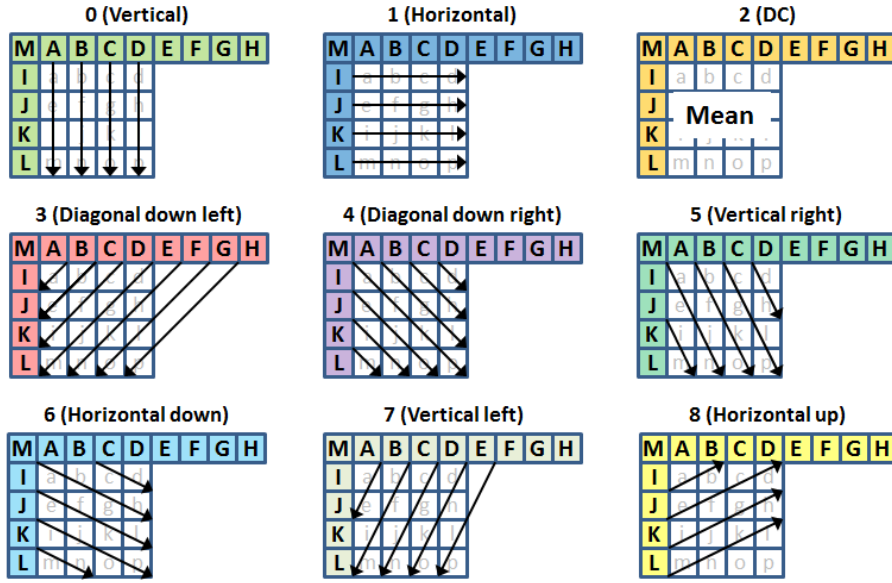


Figure 3.5: Luma intra-prediction modes for a 4x4 block [28][34][35].

original match are searched. In the second step of the process, the half-pixel search is further refined by searching eight locations around the half-pixel match at a quarter pixel granularity providing us with the final motion vector (MV). The algorithm used for quarter pixel up-sampling is a simple bi-linear filter.

$$x_{upsampled} = x_{-2} - 5x_{-1} + 20x_0 + 20x_1 - 5x_2 + x_3 \quad (3.2)$$

FME is also data parallel like IME, but it has some sequential dependencies and a more complex computation kernel which makes it more challenging to parallelize. FME takes up 36% of the total execution time and 40% of total energy.

### 3.1.2 Intra-Prediction, Discrete Cosine Transform and Quantization

Unlike IME and FME, intra-Prediction uses previously encoded neighboring image blocks from the current frame to form an alternate prediction for the current image-block. To maintain a high prediction accuracy, intra-prediction supports a large number of luminance and chrominance prediction modes: nine luminance modes for 4x4 blocks, four luminance modes for 16x16 blocks and four chrominance modes. Generation of the prediction for each block requires intra-prediction to compare the prediction results of various linear combinations of previously encoded and reconstructed neighboring blocks from the left and the top. Figure 3.5 presents the formation of nine luminance prediction modes employed for predicting 4x4 blocks. Although, the algorithm is dominated by arithmetic operations, the computation is much less regular than the motion estimation algorithms and the performance is marred by sequential dependencies between current blocks and their neighbors.

The residual data obtained by subtracting the predicted image block from the current image block is then transformed using a 4x4 integer approximation to the Discrete Cosine Transform (DCT) [36]. A set of coefficients are produced by the transform, which are then quantized (quant) and sent to CABAC. The precision of the quantization process is governed by the quantization parameter (QP), which is typically chosen to strike a balance between the quality of the output bit-stream and the compression ratio. Although, the basic function of DCT and Quant is relatively simple and data parallel, it is invoked a large number of times for each 16x16 image block, which calls for an efficient implementation. For the rest of this paper, we merge the operations of intra-prediction, DCT and Quant into the IP stage. The combined operation accounts for 7% of the total execution time and 6% of total energy.

### 3.1.3 CABAC

The coding efficiency of H.264 is significantly enhanced by the use of Context Adaptive Binary Arithmetic Coding (CABAC). It is extensively utilized to deliver high quality video at low-bitrates. While it takes less than 2% of the execution time and

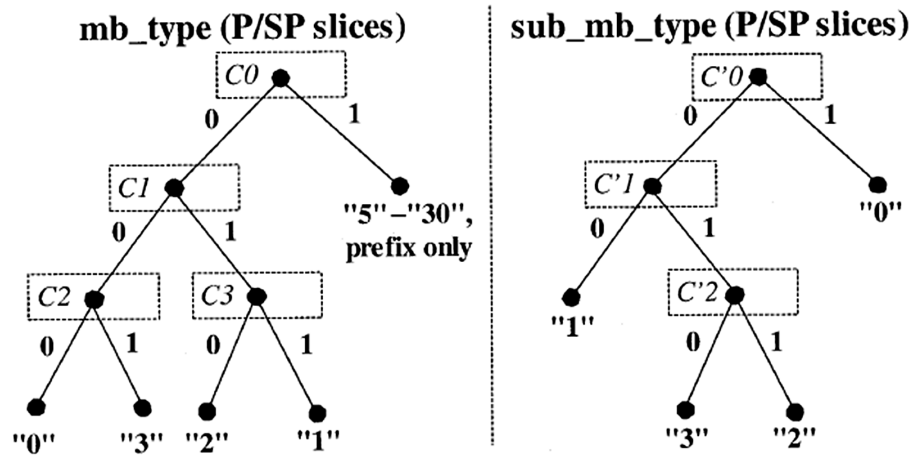


Figure 3.6: Illustration of the binarization for (a) macro-block type (`mb_type`) and (b) sub-macro-block type (`sb_mb_type`) for two different slice types [37].

total energy in a high definition H.264 encoder, CABAC often becomes the bottleneck in parallel systems due to its sequential nature. To entropy encode the H.264 bit-stream CABAC employs a three step process: binarization of non-binary input syntax elements (SE), context modeling of binarized bin strings and binary arithmetic coding [37].

### Binarization

In CABAC binarization is regarded as the “pre-processing” step to the subsequent stages of context modeling and arithmetic coding. The main task of binarization is to reduce the alphabet size of the non-binary syntax elements such as macro-block type and motion vector difference for encoding by converting the input syntax elements into binary codes while binary valued syntax elements bypass this stage. As shown in Figure 3.6, the input syntax elements are uniquely mapped to sequences of binary decisions called bins, interpretable in the context of a binary code tree [38]. Because of the reduction in the alphabet size, less complex binary arithmetic coders become available, substantially reducing the compute requirements. The conversion also allows maintenance of statistical properties at the level of individual bins; thus, improving coding efficiency.



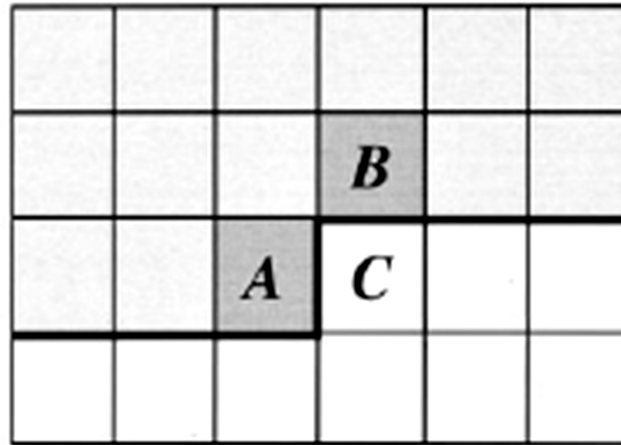


Figure 3.7: Illustration of a context template consisting of two neighboring syntax elements *A* and *B* to the left and on top of the current syntax element *C*[37].

### Context Modeling

The context modeling (CM) is tasked with assigning probability models to select “bins”. CM contains 399 context models with each model being 7-bits wide. Out of the 7-bits, 6-bits represent the probability associated with the bin and the remaining 1-bit indicates the most probable symbol (MPS) of the context bin. The probabilities are updated in the arithmetic coding stage.

Because the coding efficiency of binarized symbols is determined by a Context Model, selecting a model that accurately captures the statistical dependencies of the bins and keeping it up to date during the encoding process is essential [37]. Probability models assigned to selected bins of the binarized symbols are used in the subsequent arithmetic coding stage to drive the output bit stream. Although, the modeling process starts by passing each bin through a coding mode decision, not every bin is assigned a context model. Modeling is skipped for less frequently observed bins and bins thought to have an even probability distribution. However, depending upon the symbol type, the bins that are more frequently observed can choose among four different types of context models. As shown in Figure 3.7, the first type of context model involves assigning probability models to a syntax element depending upon the context of the two neighbors in the past of the current syntax element.

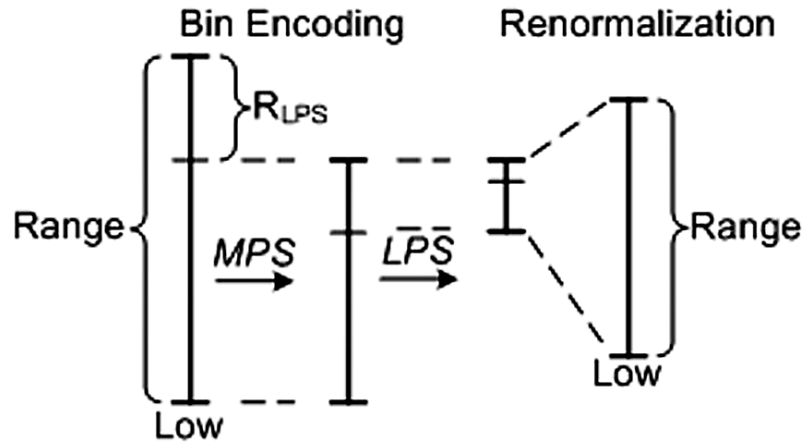


Figure 3.8: Bin encoding flow, along with the re-normalization step for when the Range of the encoding interval falls below a certain threshold.

The second type, which is only available for syntax elements belonging to macroblock types, takes into account the values of the previously coded bins ( $b_0, b_1, b_2, \dots, b_{i-1}$ ) to decide the context model. In contrast, the third and the fourth types, which are available for residual data, depend upon the position or level of the bin in the scanning path[39]. All of the 399 context models available in CABAC are divided into these four categories.

### Binary Arithmetic Coding

Binary arithmetic coding (BAC) stage employs a recursive interval sub-division technique to produce the output bit stream. As explained in [37] BAC works by repeatedly sub-dividing the encoding interval based upon *Least Probable Symbol* (LPS) or the *Most Probable Symbol* (MPS) obtained from the context modeling stage. To illustrate this further let's consider Figure 3.8. In the figure the binary encoding interval is represented by "Low" as its lower bound and "Range" as its width. Assuming that  $p_{LPS}$  represents the probability of LPS, BAC works by sub-dividing the encoding interval into two sub-ranges:  $R_{LPS} = R * p_{LPS}$  representing the Range associated with LPS and  $R - R_{LPS}$  representing the Range associated with MPS. The observed binary decision of the bin, LPS or MPS, identifies which sub-interval,  $R_{LPS}$  or  $R_{MPS}$ , is

used for further processing. Because “Range” and “Low” are represented by a finite number of bits, to ensure that the precision of “Range” is representable in hardware, a renormalization step is undertaken when the “Range” falls below a certain limit. Because by design renormalization takes a variable number of iterations, it presents a serious challenge for any hardware implementation.

## 3.2 Experimental Methodology

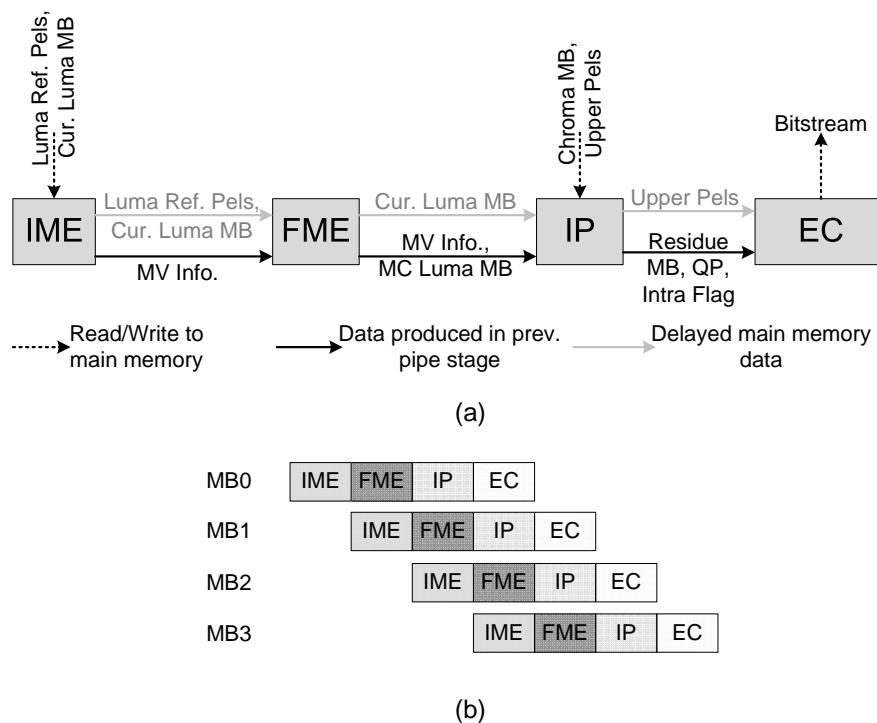


Figure 3.9: Four stage macroblock partition of H.264. (a) Data-flow between stages. (b) How the pipeline works on different macroblocks. IP stands for Intra-Prediction. EC is CABAC [40].

To understand what is needed to gain ASIC level efficiency, we use existing H.264 partitioning techniques adopted by ASIC implementations [5], and modify the H.264 encoder reference code JM 8.6 [30] to remove dependencies in IME motion vector

prediction with minimal loss in quality and allow mapping of the three major algorithmic blocks to the four-stage macro-block (MB) pipeline shown in Figure 3.9. This mapping exploits task level parallelism at the macro-block level and significantly reduces the inter-processor communication bandwidth requirements by sharing data between pipeline stages.

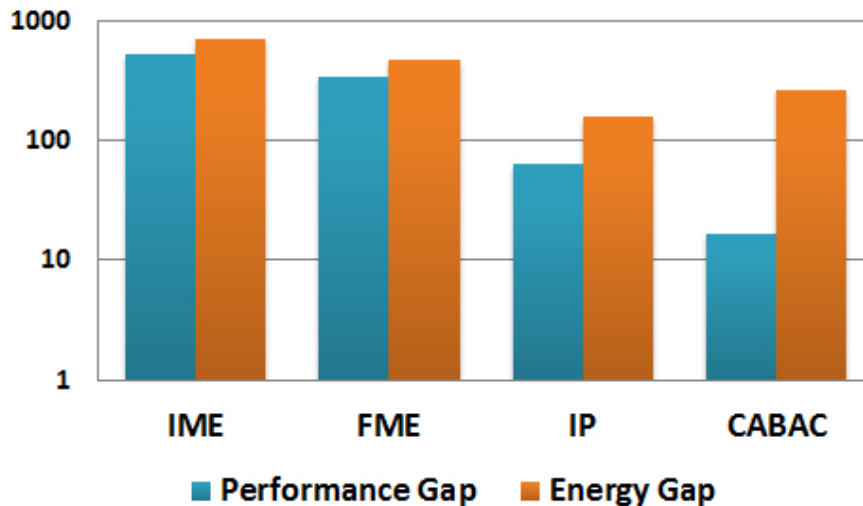


Figure 3.10: The performance and energy gap for base CMP implementation when compared to an equivalent ASIC [40].

In the base system, we map this four-stage macro-block partition to a four-processor Tensilica CMP system where each processor has 16KB 2-way set associative instruction and data caches. Figure 3.10 highlights the large efficiency gap between our base CMP and the reference ASIC for individual 720p HD H.264 sub-algorithms. The energy required for each RISC instruction is similar and as a result, the energy required for each task (shown in Figure 3.11) is related to the cycles spent on that task. The RISC implementation of IME, which is the major contributor to performance and energy consumption, has a performance gap of 525x and an energy gap of over 700x compared to the ASIC. IME and FME dominate the overall energy and thus need to be aggressively optimized. However, we also note that while IP and CABAC are much smaller parts of the total energy delay, even they need about 100x energy improvement to reach ASIC levels.

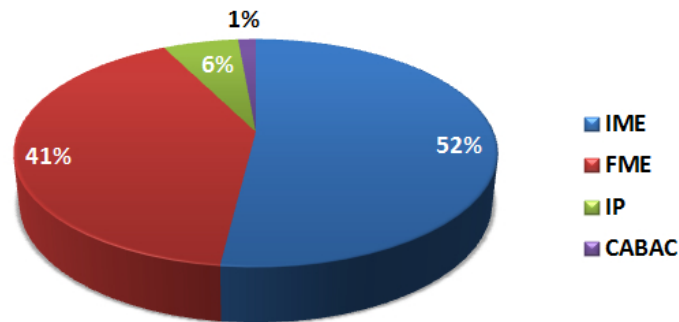


Figure 3.11: Processor energy breakdown for base implementation, over the different H.264 sub-algorithms. In the chart, IME takes 52% of the total energy followed by FME that accounts for 41% of the total. IP consumes 6% of the total while CABAC takes just 1% of the total [40].

Although, Figure 3.10 presents performance comparison for each processor independently, when functioning in a macro-block level pipeline, IME becomes the bottleneck restricting system performance to only 0.06 FPS. Allocation of more computing resources to IME and FME computationally balances the pipeline, but it has a negligible impact on energy efficiency and the improvement in performance remains below 2x. Needless to say, this imbalance has an insignificant effect on the two to three orders of magnitude performance and energy gap between the base CMP system and the ASIC that we are looking to close.

At approximately 8.6B instructions to process 1 frame, our base system consumes about 140 pJ per instruction; a reasonable value for a simple 90nm RISC processor running at 1.1V. To further analyze the energy efficiency of this base CMP implementation we break the processor's energy into different functional units as shown in Figure 3.12. This data makes it clear how far we need to go to approach ASIC efficiency. The energy spent in instruction fetch (IF) is an overhead due to the programmable nature of the processors and is absent in a custom hardware state machine, but eliminating all this overhead only increases the energy efficiency by less than one third. Even if we eliminate everything but the functional unit energy, we still end up with energy savings of only 20x — not nearly enough to reach ASIC levels.

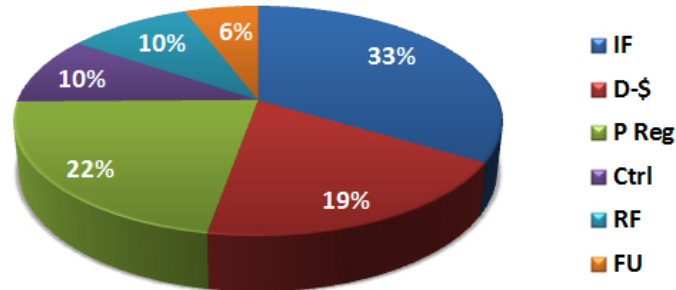


Figure 3.12: Processor energy breakdown for base implementation. IF (33%) is instruction fetch/decode. D-\$ (19%) is data cache. P Reg (22%) includes the pipeline registers, buses, and clocking. Ctrl (10%) is miscellaneous control. RF (10%) is register file. FU (6%) is the functional units [40].

Inspired by GPUs and Intel’s SSE instructions, we first apply datapath extensions that are relatively general-purpose data-parallel optimizations and consist of single instruction, multiple data (SIMD) and multiple instruction issue per cycle (we use long instruction word, or VLIW), with a limited degree of algorithm-specific customization coming in the form of operation fusion — the creation of new instructions that combine frequently occurring sequences of instructions. This step represents the datapaths in current state-of-the-art optimized CPUs or simple extensions to them. In the next step, we replace these generic datapaths by custom units, and allow unrestricted tailoring of the datapath by introducing new compute operations as well as new register file structures.

The results of these customizations are shown in Figures 3.13 through 3.15. The next three sections describe these results in detail and evaluate the effectiveness of these three customization strategies. Collectively, these results describe how efficiencies improve by 170x over the baseline CMP.

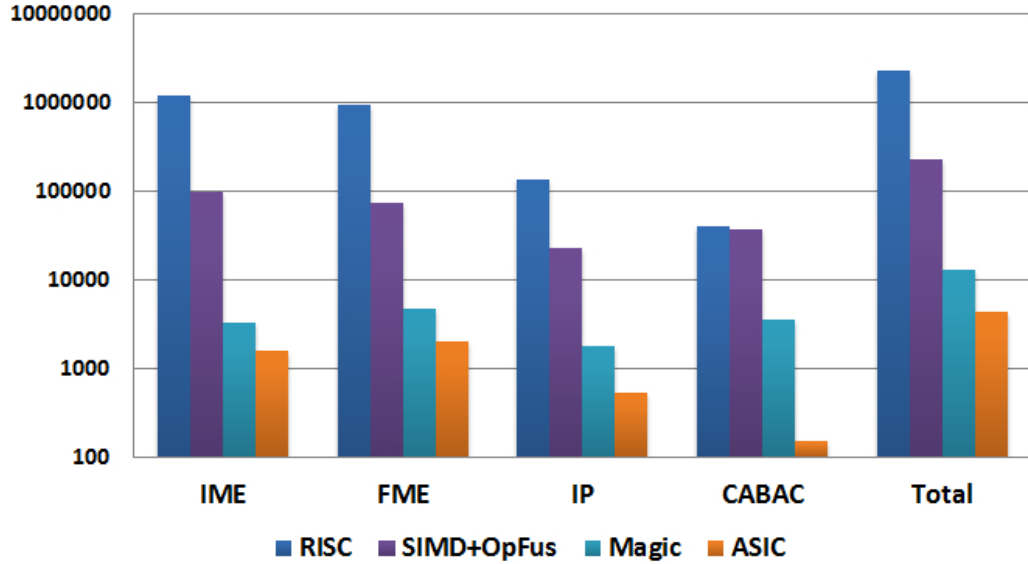


Figure 3.13: Each set of bar graphs represents energy consumption ( $\mu\text{J}$ ) at each stage of optimization for IME, FME, IP and CABAC respectively. The first bar in each set represents base RISC energy; followed by RISC augmented with SIMD and operation fusion; and then RISC augmented with “magic” instructions. The last bar in each group indicates energy consumption by the ASIC [40].

	VLIW Slots	Register File Size	SIMD Width	Load/Store Width
IME	2	16 rows x 16 cols x 8-bit	16	128-bit
FME	2	32 rows x 18 cols x 9-bit	18	128-bit
IP	2	16 rows x 8 cols x 8-bit	8	64-bit

Table 3.1: Description of VLIW and SIMD resources employed for each sub-algorithm.

### 3.3 SIMD and Operation Fusion

Using Tensilica’s TIE extensions we add VLIW instructions and SIMD execution units with vector register files of custom depths and widths as described in Table 3.1. A single SIMD instruction performs multiple operations (8 for IP, 16 for IME, and 18 for FME), reducing the number of instructions and consequently reducing IF energy. VLIW instructions execute 2 operations per cycle, further reducing cycle count. As expected, DLP algorithms using SIMD units show a large decrease in processor

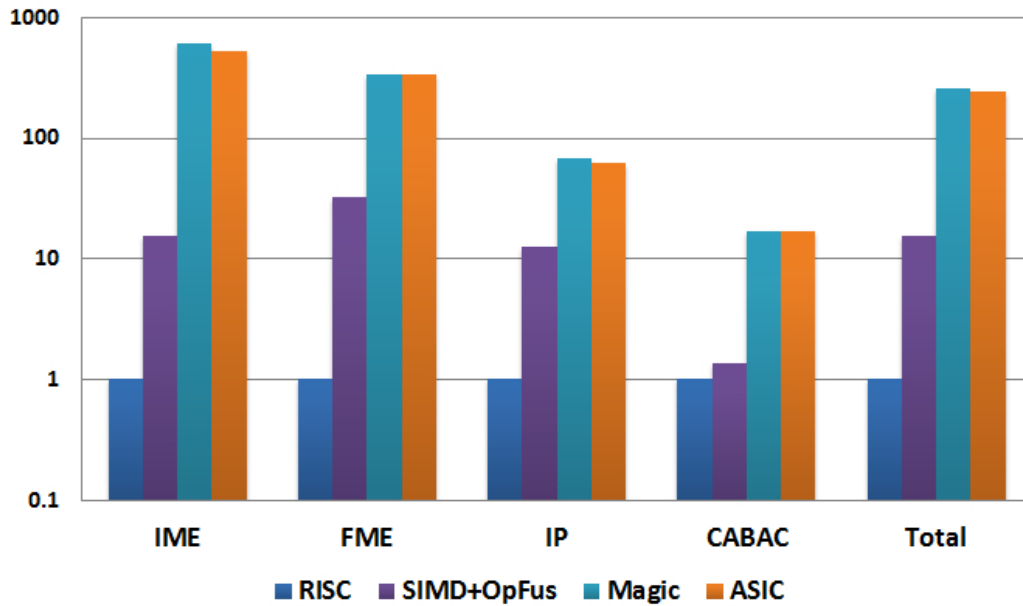


Figure 3.14: Speedup at each stage of optimization for IME, FME, IP and CABAC [40].

energy; speedup increases as the number of instructions executed decreases. IME and FME use 16 and 18-way SIMD datapaths and achieve speedups of 10x and 14x. Intra/DCT/Quant using an 8-way SIMD datapath achieves a speedup of 6x. The SIMD units use custom-width functional units instead of standard 32-bit versions to enable more efficient computation, and generally run between 8 and 16 bits. Moreover, SIMD operations perform wider register file and data cache accesses which are more energy efficient compared to narrower accesses. Therefore all components of instruction energy depicted in Figure 3.12 get a reduction through the use of these enhancements. Unfortunately, even performing sixteen concurrent operations barely increases the percentage energy used by the functional units, which still comprise around 10% of the total.

We further augment these enhancements with operation fusion, in which we fuse together frequently occurring complex instruction sub-graphs for both RISC and SIMD instructions. To prevent the register file ports from increasing, the fused instructions are restricted to use up to two input operands and can produce only one



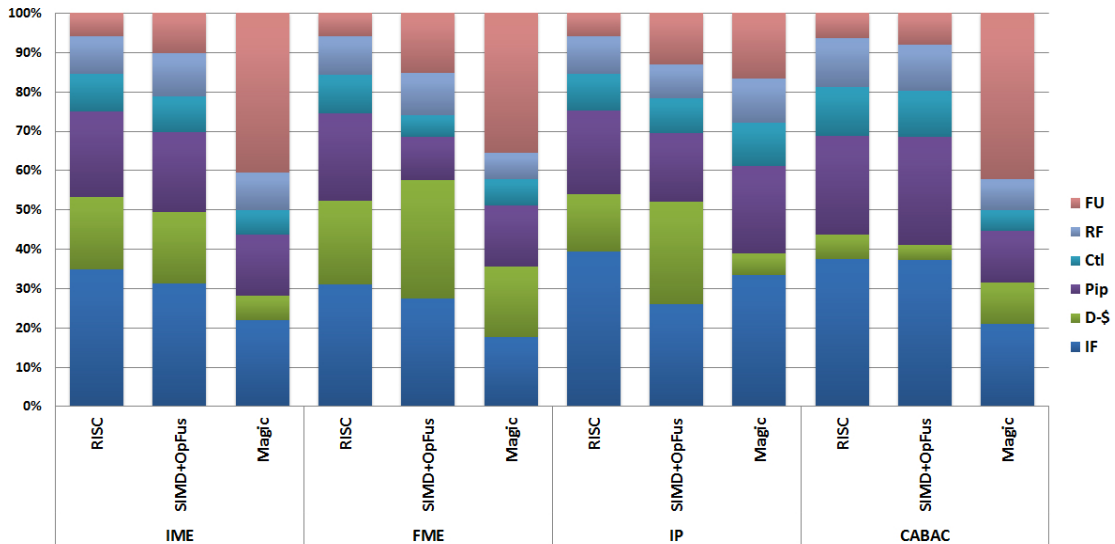


Figure 3.15: Processor energy breakdown for H.264. IF is instruction fetch/decode. D-\$ is data cache. Pip is the pipeline registers, buses, and clocking. Ctl is random control. RF is the register file. FU is the functional elements. Only the top bar or two (FU, RF) contribute useful work in the processor. For this application it is hard to achieve much more than 10% of the power in the FU without adding custom hardware units [40].

output. One example of operation fusion is the sum-of-absolute-difference operator used in IME in which we fuse together difference, absolute and sum operations to create a highly optimized two input, one output SAD instruction. Operation fusion improves energy efficiency by reducing the number of instructions and also by reducing the number of register file accesses by internally consuming short-lived intermediate data. Additionally, fusion gives us the ability to create more energy-efficient hardware implementations of the fused operations such as multiplication implemented using shifts and adds. The reductions due to operation fusion are less than 2x in energy and less than 2.5x in performance.

With SIMD, VLIW and Op Fusion support, IME, FME and IP processors achieve speedups of around 15x, 30x and 10x, respectively. CABAC is not data parallel and benefits only from VLIW and op fusion with a speedup of merely 1.1x and almost no change in energy per operation. Overall, the application gets an energy efficiency

gain of almost 10x, but still uses greater than 50x more energy than an ASIC. To reach ASIC levels of efficiency, we need a different approach.

### 3.4 Custom Instructions for Compute Bound Applications

The root cause of the large energy difference between GP and ASIC is that the basic operations in H.264 are very simple and low energy. They only require 8–16 bit integer operations, so the fundamental energy per operation is on the order of hundreds of femto-joules in a 90 nm process. All other costs in a processor — IF, register fetch, data fetch, control, and pipeline registers — are much larger (140 pJ) and dominate overall power. Standard SIMD and simple fused instructions can only go so far to improve the performance and energy efficiency. It is hard to aggregate more than 10–20 operations into an instruction without incurring growing inefficiencies, and executing tens of low energy 8–16 bit operations per cycle is not enough to overcome large processor overheads and we still end up with a machine where around 90% of the energy is going into overhead functions; see Figure 3.15. It is now easy to see how an ASIC can be 2–3 orders of magnitude lower energy than a processor. For computationally limited applications with low-energy operations, an ASIC can implement hardware which both has low overheads, and is a perfect structural match to the application. These features allow it to exploit large amounts of parallelism efficiently. To match these results in a processor we must amortize the per-instruction energy overheads over hundreds of these simple operations. To create instructions with this level of parallelism requires matching the hardware to the data-flow of the algorithm by building custom storage structures with algorithm-specific communication links to directly feed large amounts of data to custom functional units without explicit register accesses. These structures also substantially increase data-reuse in the datapath and reduce communication bandwidth and power at all levels of the memory hierarchy (register, cache, and memory).

Derived directly from the data-flow of the algorithm, “magic” instructions can

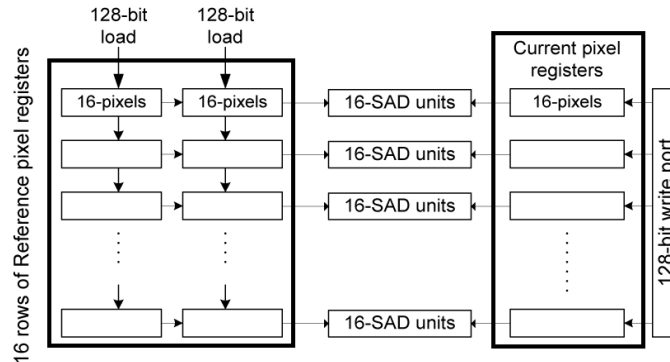


Figure 3.16: Custom storage and compute for IME 4x4 SAD. Current and ref-pixel register files feed all pixels to the 16x16 SAD array (16-SAD units) in parallel. Each bus is 128-bits wide. Also, the ref-pixel register file allows horizontal and vertical shifts [40].

have a large effect on both the energy and performance of an application. Yet they are often difficult to derive directly from the code. Such instructions typically require an understanding of the underlying algorithms, as well as the capabilities and limitations of existing hardware resources, thus requiring someone with an understanding of the algorithm and the hardware to create them.

### 3.4.1 IME Strategy

To demonstrate the nature and benefit of magic instructions we first look at IME, which determines the best alignment for two image blocks. The best match is defined by the smallest sum-of-absolute-differences (SAD) of all of the pixel values. Since finding the best match requires scanning one image block over a larger piece of the image, one can easily see that while this requires a large number of calculations, it also has very high data locality. Figure 3.16 shows the custom datapath elements added to the IME processor to accelerate this function. At the core is a 16 x 16 SAD array, which can perform 256 SAD operations in 1 cycle. Since our standard vector register files cannot feed enough data to this unit per cycle, the SAD unit is fed by a custom register structure, which allows parallel access to all 16-pixel rows and enables this datapath to perform one 256-pixel computation per cycle. In addition,

the intermediate results of the pixel operations need not be stored since they can be reduced in place (summed) to create the single desired output. Furthermore, because we need to check many possible alignments, the custom storage structure has support for parallel shifts in all four directions, thus allowing one to shift the entire comparison image in only one cycle. This feature drastically reduces the instructions wasted on loads, shifts, and pointer arithmetic operations as well as data cache accesses. “Magic” instructions and storage elements are also created for other major algorithmic functions in IME to achieve similar gains.

Thus, by reducing instruction overheads and by amortizing the remaining overheads over larger datapath widths, this functional unit finally consumes around 40% of the total instruction energy. The performance and energy efficiency improve by 200–300x over the base implementation, match the ASIC’s performance and come within 3x of ASIC energy. This customized solution is 20–30x better than the results using only generic data-parallel techniques and no longer uses the SIMD data structures.

### 3.4.2 FME Strategy

FME improves the IME match by refining the alignment to a quarter pixel granularity. Fractional alignment necessitates interpolation of the reference image to first a half and then to a quarter pixel granularity followed by a search stage focused on locating a better estimate of the current image block. A six tap 2D separable filter is used to perform the half pixel up-sampling step and is comprised of separate horizontal and vertical interpolation stages. Because FIR filters possess the unique advantage that each filtered output reuses nearly all inputs of the previous output, the data locality is extremely high. To exploit this high data locality we maintain local state inside the processor, which allows us to significantly cut back on the number of memory operations; thus, increasing efficiency. We further exploit the local state to perform hundreds of low-power arithmetic operations in parallel by generating and supplying multiple shifted versions of the input data to the functional units generating multiple filtered outputs. Such a large number of arithmetic operations executing in parallel

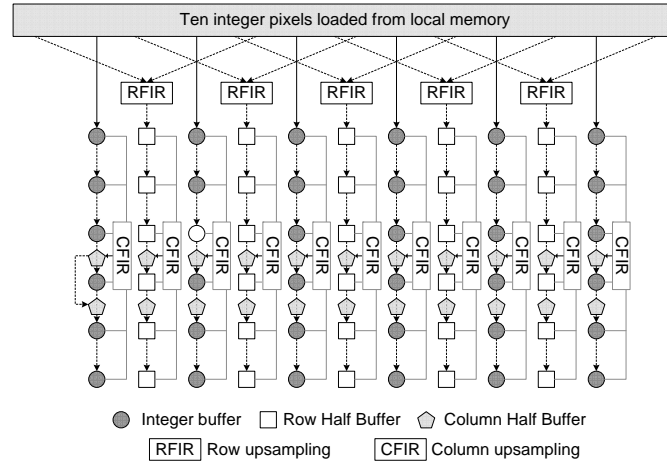


Figure 3.17: FME upsampling unit. Customized shift registers, directly wired to function logic, result in efficient upsampling. Ten integer pixels from local memory are used for row upsampling in RFIR blocks. Half upsampled pixels along with appropriate integer pixels are loaded into shift registers. CFIR accesses six shift registers in each column simultaneously to perform column upsampling [40].

have a profound effect on the efficiency, which improves dramatically.

To maintain the local state and to reduce the ill effects of IF and register file overheads, we augment the processor register file with a custom 8 bit wide, 6 entry shift register structure which works like a FIFO: every time a new 8 bit value is loaded, all elements are shifted. This eliminates the use of expensive register file accesses for either data shifting or operand fetch, which are now both handled by short local wires. All six entries can now be accessed in parallel and we create a six input multiplier/adder which uses a carry-save data format until the final addition to use less energy than a composition of normal 2-input adders. Finally, since we need to perform the up-sampling in 2-D, we build a shift register structure that stores the horizontally up-sampled data, and feeds its outputs to a number of vertical up-sampling units (Figure 3.17). This transformation yields large savings even beyond the savings in IF energy. From a pure datapath perspective (register file, pipeline registers, and functional units), this approach dissipates less than 1/30th the energy of a traditional approach.

A look at the FME SIMD code implementation highlights the advantages of this

custom hardware approach versus the use of larger SIMD arrays. The SIMD implementation suffers from code replication and excessive local memory and register file accesses, in addition to not having the most efficient functional units. FME contains seven different sub-block sizes ranging from 16 x 16 pixel blocks to 4 x 4 blocks, and not all of them can fully exploit the 18-way SIMD datapath. Additionally, to use the 18-way SIMD datapath, each sub-block requires a slightly different code sequence, which results in code replication and more I-fetch power because of the larger I-cache. To avoid these issues, the custom hardware upsampler processes 4 x 4 pixels. This allows it to reuse the same computation loop repeatedly without any code replication, which, in turn, lets us reduce the I-cache from a 16KB 4-way cache to a 2KB direct-mapped cache. Due to the abundance of short-lived data, we remove the vector register files and replace them with custom storage buffers. The “magic” instruction reduces the instruction cache energy by 54x and processor fetch and decode energy by 14x. Finally, as Figure 3.15 shows, 35% of the energy is now going into the functional units, and again the energy efficiency of this unit is close to an ASIC.

### 3.5 Custom Instructions for Control Bound Applications

Control bound applications such as H.264’s CABAC are highly sequential with small instruction blocks separated by dependent branches. As shown in Figure 3.18, the computation performed by a control bound application is usually not expensive, but it takes many instructions to compute. These algorithms benefit very little from data-parallel execution units and require innovative instruction level parallelism (ILP) techniques such as out-of-order (OOO) processing and speculative execution to improve execution times; however, the complexity of the ensuing hardware keeps energy consumption high. Absence of efficient general-purpose optimizations force control bound applications to rely on algorithm specific techniques to improve efficiency. The improvement in energy consumption for custom hardware is solely governed by the height and the achievable compaction of the instruction based directed acyclic graphs

```

while (range < QUARTER) {
  if (low >= HALF) { -----
    outstanding += 1;
    low -= HALF;
  } else if (low < QUARTER) {
    Outstanding -= 1;
  } else {
    Bitstofollow++;
    low -= QUARTER;
  }
  low <<= 1;
  range <<= 1; -----
}

```




Figure 3.18: A simplified version of a control bound loop from BAC.

(DAG) like the one shown in Figure 3.18. The DAGs are mainly comprised of basic blocks and control flow instructions. The greater the height of the DAG and higher the compaction, the greater the number of operations that can be executed per cycle and more effective is the amortization of processor overheads.

In our experience it’s hard to increase the overall efficiency for a control bound application by more than an order of magnitude. The lower overall gains compared to compute bound applications are the consequence of the dependent branches that restrict the fusion of DAGs to a maximum of 2–3 basic blocks. Because each basic block is typically 4–5 instructions long, fusing together more than 10–15 instructions efficiently becomes increasingly difficult, thus setting a hard limit on the expected efficiency.

### 3.5.1 CABAC Strategy

CABAC originally consumed less than 2% of the total energy, but after data-parallel components are accelerated by “magic” instructions, CABAC dominates the total energy and becomes the tall pole. However, it requires a different set of optimizations because it is control oriented and not data parallel. Thus, for CABAC, we are more interested in control fusion than operation fusion.

A critical part of CABAC is the arithmetic encoding stage, which is a serial process

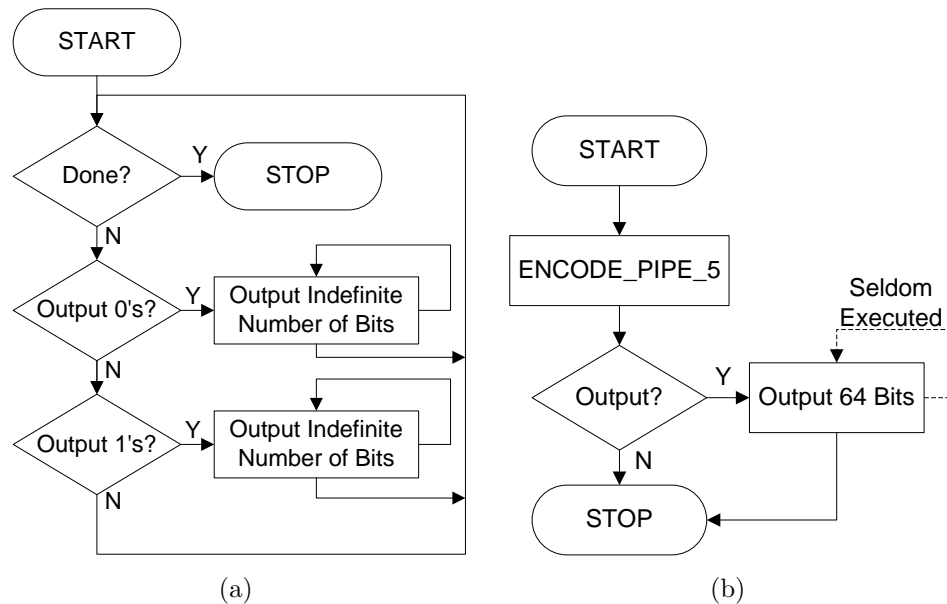


Figure 3.19: CABAC Arithmetic Encoding Loop 3.19(a) H.264 reference code. 3.19(b) After insertion of “magic” instructions. Much of the control logic in the main loop has been reduced to one constant time instruction ENCODE\_PIPE\_5 [40].

with small amounts of computation, but lots of control-flow decisions. Despite the low-complexity of the encoding stage, the abundance of control-flow decisions force the RISC based implementation to be comprised of tens of instructions resulting in low efficiency. Therefore, to reduce the control-flow overhead and to compress this large instruction graph, we break the arithmetic coding down into a simple pipeline and drastically change it from the reference code implementation, reducing the binary encoding of each symbol to five instructions. While there are several if-then-else conditionals reduced to single instructions (or with several compressed into one), the most significant reduction came in the encoding loop, as shown in Figure 3.19(a). Each iteration of this loop may or may not trigger execution of an internal loop that outputs an indefinite number of encoded bits. By fundamentally changing the algorithm, the while loop was reduced to a single constant time instruction (ENCODE\_PIPE\_5) and a rarely executed while loop, as shown in Figure 3.19(b).

The other critical part of CABAC is the conversion of non-binary-valued DCT coefficients to binary codes in the binarization stage. To improve the efficiency of



this step, we create a 16-entry LIFO structure to store DCT coefficients. To each LIFO entry, we add a single-bit flag to identify zero-valued DCT coefficients. These structures, along with their corresponding logic, reduce register file energy by bringing the most frequently used values out of the register file and into custom storage buffers. Using “magic” instructions we produce Unary and Exponential-Golomb codes using simple operations, which also helps reduce datapath energy. These modifications are inspired by the ASIC implementation described in Shojania and Sudharsanan [41]. CABAC is optimized to achieve the bit rate required for H.264 level 3.1 at 720p video resolution.

### 3.6 Custom Instructions Area Comparison

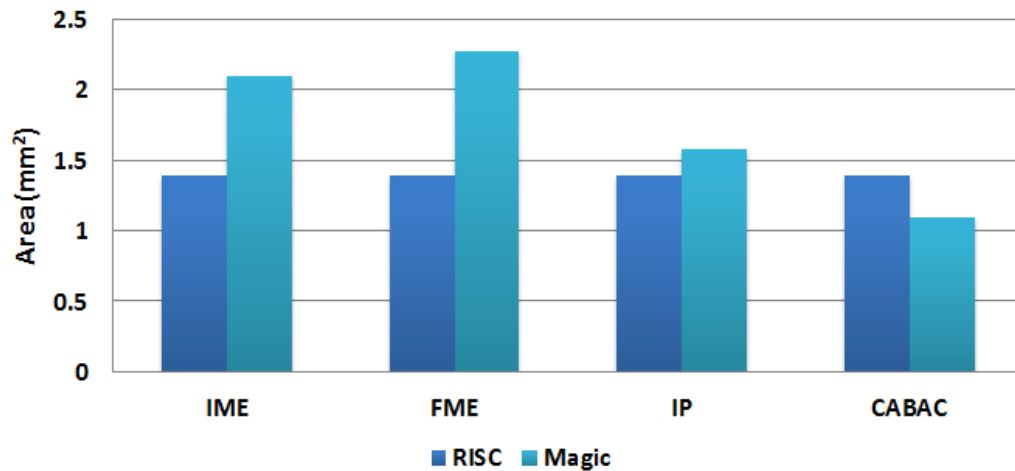


Figure 3.20: Area comparison of “magic” instructions with the base processor for IME, FME, IP and CABAC. Each area bar includes the area for Tensilica RISC core, instruction and data L1 caches and custom units (applicable only in “magic” instructions). Although, the non-optimized base processor for each algorithm has 16KB 2-way set associative L1 caches, the cache sizes are customized for “magic” instructions according to the requirements of each algorithm. Because CABAC’s code size reduces appreciably after the application of “magic” instructions, CABAC’s area for the customized case is smaller than that of the non-optimized base configuration.

The area comparisons for magic instructions for IME, FME, IP and CABAC are

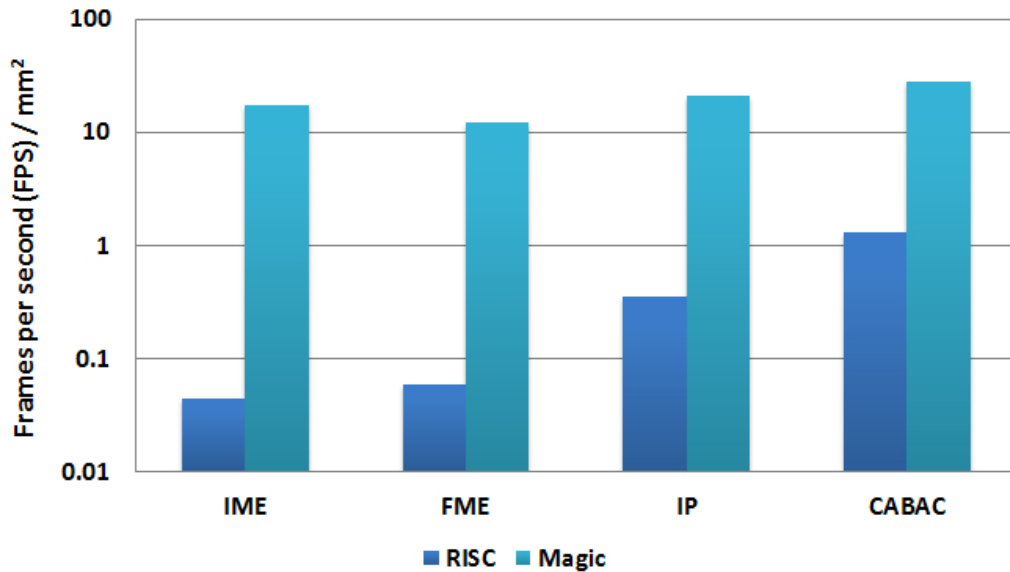


Figure 3.21: Comparison of area efficiency ( $FPS/mm^2$ ) for IME, FME IP and CABAC.

presented in Figure 3.20. Although, magic instruction based datapaths perform algorithm specific customization of L1 cache sizes reducing the memory area appreciably, the addition of custom storage structures and datapath extensions considerably increases the area of the processing elements. It's especially evident in IME and FME, which add substantially large storage structures and datapath extension units to the Tensilica RISC processors. However, as shown in Figure 3.21, IME and FME also report the highest gains in area efficiency ( $FPS/mm^2$ ), recording up to two orders of magnitude improvement in area efficiency over the non-optimized Tensilica core. Because of simpler computation, IP and CABAC require relatively smaller datapath extensions but they also report at least an order of magnitude boost in area efficiency.

### 3.7 Custom Instructions Summary

To summarize, the magic instructions for data-parallel algorithms perform up to hundreds of operations each time they are executed, so the overhead of the instruction

is better balanced by the work performed. Of course this is hard to do in a general way, since bandwidth requirements and utilization of a larger SIMD array would be problematic. Therefore we solved this problem by building storage units tailored to match the data-flow of the application, and then directly connecting the necessary functional units to these storage units. These data-flow optimized storage units greatly amplified the register fetch bandwidth, since data in the storage units is used for many different computations. In addition, since the intra-storage and functional unit communications were fixed and local, they can be managed at ASIC-like energy costs. After this effort, the processors optimized for data-parallel algorithms have a total speedup of up to 600x and an energy reduction of 60–350x compared to our base CMP as shown in Figure 3.15. The efficiencies found in these data-flow optimized datapaths are impressive. Because they perform hundreds of operations in parallel by taking advantage of data sharing patterns and by creating very efficient multiple-input operations, efficiency gains of up to three orders of magnitude are realizable over general-purpose processors.

Unlike data-parallel algorithms, efficiency gains for control intensive applications such as CABAC remain relatively low. We observe that for CABAC the performance improves by 17x while efficiency goes up by 8x. While impressive, these results demonstrate the challenge of improving the efficiency of control intensive applications.

### 3.8 Conclusion

It is important to remember that the “overhead” of using a processor depends on the energy required for the desired operation. Floating point (FP) energy costs are about 10x the small integer operations we have explored in this thesis, so machines with ten wide FP units will not be far from the maximum efficiency possible for that class of applications. Unfortunately for the very simple operations, a gain of 10x offered by SIMD units is not enough to bridge the wide efficiency gap between general-purpose optimizations and ASIC. Here we need to execute hundreds of operations per cycle, and thus need to couple storage with the compute hardware. Interestingly, for these types of gains to be possible the operations must be simple and the memory accesses

must be very limited. This greatly restricts the type of computation that can achieve these high gains, and opens the possibility of constructing a more general, highly efficient engine. The design of this engine will be described in the next chapter.

For control bound applications the solution is to restructure the application, and then compact the instructions required for control into a small number of complex instructions. Thus, without incorporating custom instructions in a control bound application the efficiency limit cannot be achieved.

## Chapter 4

# Convolution Engine: Balancing Efficiency and Flexibility in Specialized Computing

In this chapter we explore the trade-offs between flexibility and efficiency in specialized computing with the help of simple function limited applications. We have already identified that compute bound applications limited by extremely low-power arithmetic operations offer the highest efficiency gains. These gains in efficiency, which can be as high as three orders of magnitude, are achieved with the help of specialized units tuned to the data storage and compute structures and their connectivity to the data-flow and data locality patterns of the algorithms. This tuning eliminates redundant data transfers and facilitates creation of closely coupled datapaths and storage structures allowing hundreds of low-energy operations to be performed for each instruction and data fetched. Hence, if we identify data-flow and data locality patterns that are common to a wide range of kernels and perform a substantial amount of compute using short data with minimal reliance on the memory, we can create specialized units that are highly energy efficient and yet can be programmed and reused across a wide range of applications.

Thus, to achieve high efficiency while still retaining flexibility the engine must target a data-flow that enables execution of a substantial amount of compute with

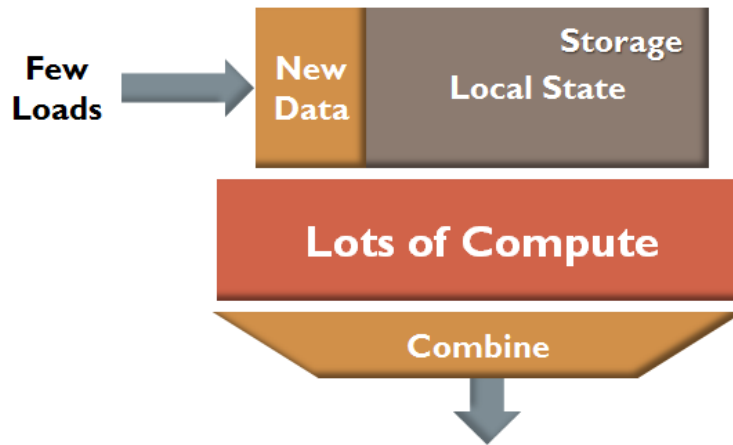


Figure 4.1: This diagram represents the constraints that a flexible processing element must satisfy to maintain high efficiency.

limited to no interaction with the memory. Although, these constraints significantly reduce energy waste and enable effective amortization of processor overheads, they profoundly influence the design of the flexible engine. In the absence of abundant memory traffic, the engine must now maintain local state to keep the compute units busy. A multipurpose storage structure is needed to store the local state and merge it with the incoming memory data. This structure is also responsible for supplying data to the compute units. Because the number of compute units is large, integration of a combining tree with the functional units is desired to reduce the number of outputs to a manageable size before storing them to the memory. An abstract model satisfying the constraints described above is presented in Figure 4.1.

While the data-flow is very restrictive, there are a number of commonly used computational models that satisfy the constraints we have identified. They all fall into the class of Convolution like applications. This computational model is widely employed in computational photography, image processing and video processing applications, which are quite popular on mobile systems. Convolution-like data-flow is a common motif among these applications and involves applying a function to a stencil of the

data, then performing a generalized fusion/reduction, then shifting the stencil to include a small amount of new data, and then repeating. Some prominent examples include demosaic, feature extraction and mapping in scale-invariant-feature-transform (SIFT), windowed histograms, median filtering, motion estimation for H.264 video processing and many more.

For imaging and video applications, mobile systems typically employ hardware accelerators optimized for a single kernel, and if configurable, are configured by experts in firmware. They offer the highest efficiency but little to no programmability. In contrast programmable accelerators prevalent in embedded and desktop processors such as SIMD units target data-parallel algorithms and remain one to two orders of magnitude less efficient compared to algorithm-specific custom units. In this section we show that by targeting the data-flow described above it is possible to build more efficient programmable engines that offer efficiencies comparable to custom hardware while still retaining flexibility. Because our solution is optimized for the convolution like data-flow, we refer to our engine as the Convolution Engine (CE).

## 4.1 Computational Models

Convolution is the fundamental building block of many scientific and image processing algorithms. Equation 4.1 and 4.2 provide the definition of standard discrete 1-D and 2-D convolutions. When dealing with images,  $Img$  is a function from image location to pixel value, while  $f$  is the filter applied to the image. Practical kernels reduce computation (at a small cost of accuracy) by making the filter size small, typically in the order of 3x3 to 8x8 for 2-D convolution.

$$(Img * f)[n] \stackrel{\text{def}}{=} \sum_{k=-\infty}^{\infty} Img[k] \cdot f[n - k] \quad (4.1)$$

$$(Img * f)[n, m] \stackrel{\text{def}}{=} \sum_{l=-\infty}^{\infty} \sum_{k=-\infty}^{\infty} Img[k] \cdot f[n - k, m - l] \quad (4.2)$$

We generalize the concept of convolution by identifying two components of the

convolution: a *map* operation and a *reduce* operation. In Equation 4.1 and 4.2, the map operation is multiplication that is done on pairs of pixel and tap coefficient, and the reduce operation is the summation of all these pairs to a single value at location  $[n,m]$ . Replacing the map operation in Equation 4.2 from  $x \cdot y$  to  $|x - y|$  while leaving the reduce operation as summation, yields a sum of absolute numbers (SAD) function which is used for H.264's motion estimation. Further replacing the reduce operation from  $\sum$  to *max* will yield a max of absolute differences operation. Equation 4.3 generalizes the standard definition of convolution, to a programmable form. We refer to it as a convolution engine, where  $f$ , *Map* and *Reduce* (' $R$ ' in Equation 4.3) are the pseudo instructions, and  $c$  is the size of the convolution.

$$(Img \overset{CE}{*} f)[n, m] \stackrel{\text{def}}{=} R_{|l|<c}\{R_{|k|<c}\{Map(Img[k], f[n - k, m - l])\}\} \quad (4.3)$$

The convolution like data-flow works for many applications, but is limited by the need to have a single associative operation in the reduction. There are a number of applications that have good data locality, but need to combine results through a specific graph of operations extractable from the original algorithm. By increasing the complexity of the *Reduce* operator to enable non-commutative functions, a generalized combining network can be formulated. Powered by a diverse set of operators, we can now extend the "reduction" stage to create a structure that can input a large number of values and then compute a small number of outputs through effectively a fused super instruction as shown in Figure 4.2.

The down side of this extension is that the placement of input into the combining tree is now significant; thus, to realize the full potential of the new generalized *reduce* operator, a high level of flexibility is required in the data supply network to move the needed data to the right position. This is achieved by extending the definition of the *map* operator to also support a data permutation network in addition to the already supported set of compute operators. These new enhancements to the *map* and *reduce* operators substantially boost their generalizability and applicability; thus, increasing the space of supported applications even further.



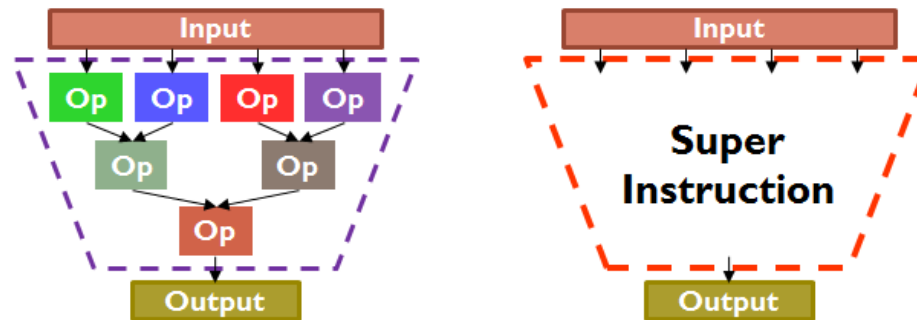


Figure 4.2: This diagram represents how the generalized reduction unit fuses the individual operations into a super instruction.

## 4.2 Applications

In this section we present our test applications and explain how we map their kernels onto the generalized *map-and-reduce* framework. Because we have already described H.264 Motion Estimation in detail in the previous chapter, here we include just a brief overview.

### 4.2.1 Motion Estimation

Motion estimation is a key component of many video codecs including H.264 in which it is computed in two steps: IME and FME.

#### Integer Motion Estimation (IME)

IME searches for an image-block's closest match from a reference image. The search is performed at each location within a two dimensional search window, using sum of absolute differences (SAD) as the cost function. IME operates on multiple scales with various blocks sizes from 4x4 to 16x16, though all of the larger block results can be derived from the 4x4 SAD results. SAD fits quite naturally to a convolution engine abstraction: the *map* function is absolute difference and the *reduce* function is summation.

### Motion Estimation (FME)

FME refines the initial match obtained at the IME step to a quarter-pixel resolution. FME first up-samples the block selected by IME, and then performs a slightly modified variant of the aforementioned SAD. Up-sampling also fits nicely to the convolution abstraction and actually includes two convolution operations: First the image block is up-sampled by two using a six-tap separable 2D filter. This part is purely convolution. The resulting image is up-sampled by another factor of two by interpolating adjacent pixels, which can be defined as a *map* operator (to generate the new pixels) with no *reduce*.

## 4.2.2 Scale Invariant Feature Transform (SIFT)

Scale Invariant Feature Transform (SIFT) is widely employed for object recognition, stereo matching and motion tracking. As explained in [26] SIFT works on images to extract features that are invariant to translations, rotations and scaling and are relatively stable against changes in illumination, image noise and camera viewpoint. The extraction process consists of four stages which work as a “cascade of filters”. They are scale-space extrema detection; key-point localization; orientation assignment; and key-point descriptor. Let’s look at the first two stages in detail:

### Scale-Space Extrema Detection

In the first stage all the image points are searched to isolate points of interest invariant to scale and rotation. As explained in [42], this process is referred to as blob detection in Computer Vision and represents mathematical techniques used to detect areas in an image that differ from their surroundings. In SIFT the scale-space-invariant blobs are detected with the help of a multi-resolution difference-of-Gaussian (DOG) pyramid in which the blobs correspond to the extrema in the DOG values. The DOG based detection procedure is an approximation to the local scale-space-extrema of the scale-normalized Laplacian of Gaussian operator which is generally employed to identify blobs. To understand this further let us consider an image  $f(x, y)$  smoothed by convolving with the Gaussian operator  $G(x, y; s) = \frac{1}{2\pi s} e^{-\frac{(x^2+y^2)}{2s}}$  at scale  $s$  to obtain

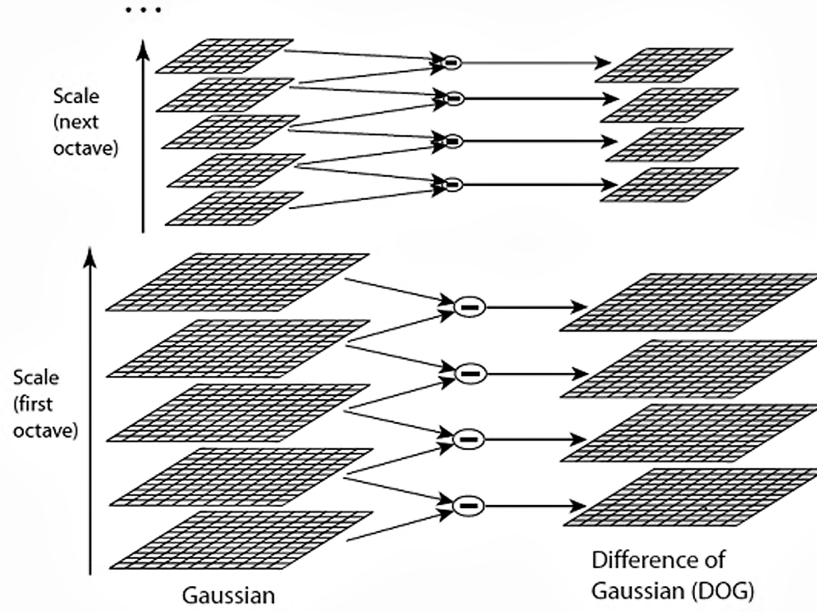


Figure 4.3: This diagram represents difference-of-Gaussian (DOG). Each Octave represents doubling of the value of  $s$  [26].

a scale-space representation:  $L(x, y; s) = G(x, y; s) * f(x, y)$ . Now, the points of interest are represented by the local extrema in the appropriate scale space identified by applying scale-normalized Laplacian operator to the blurred image as shown in Equation (4.4)[43]:

$$\nabla_{norm}^2 L(x, y; s) = s(L_{xx} + L_{yy}) = s\left(\frac{\partial^2 L}{\partial x^2} + \frac{\partial^2 L}{\partial y^2}\right) = s \nabla^2 (G(x, y; s) * f(x, y)) \quad (4.4)$$

As we alluded to earlier, SIFT approximates the scale-normalized Laplacian operator with the help of difference of Gaussians (DOG) as shown in Equation (4.5). This approximation allows SIFT to substantially reduce the computational intensity of the process for identification of points of interest.

$$\begin{aligned} DOG(x, y; s) &= L(x, y; s + ks) - L(x, y; s) \approx \frac{k}{2} \nabla^2 L(x, y; s) \\ DOG(x, y; s) &\approx \frac{(k^2 - 1)}{2} \nabla_{norm}^2 L(x, y; s) \end{aligned} \quad (4.5)$$

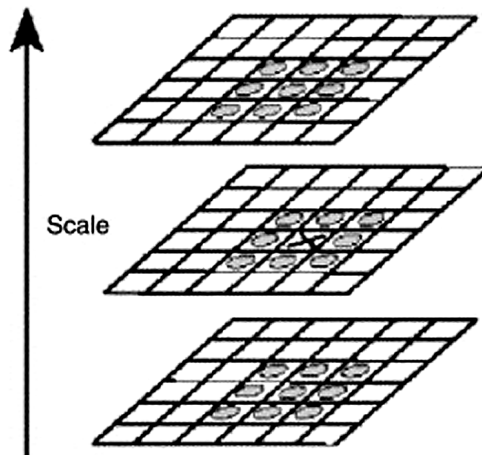


Figure 4.4: SIFT Extrema detection [26].

Since  $DOG(x, y; s)$  is the difference between Gaussian blurred images at scales  $s$  and  $ks$  where  $k$  is small, the first step in SIFT consists of generation of these images by convolving  $f(x, y)$  and  $G(x, y; s) = \frac{1}{2\pi s} e^{-\frac{(x^2+y^2)}{2s}}$  to obtain  $L(x, y; s) = G(x, y; s) * f(x, y)$  at different scales as illustrated in Figure 4.3[44]. To reduce the computational intensity further, the blurred images are down-sampled by a half before starting the next octave. Identification of key-points requires determination of the local extrema within DOGs. As shown in Figure 4.4, comparing each sample point within a DOG with its eight surrounding neighbors and nine neighbors in the scales above and below allows us to identify the local extrema. This requires 26 comparisons in total, but most of the points are eliminated quite quickly.

Even though finding scale-space extrema is a 3D stencil computation, we can convert the problem into a 2D stencil operation by interleaving rows from different images into a single buffer. The extrema operation is mapped to convolution using compare as a *map* operator and logical AND as the *reduce* operator.

### Key-Point Localization

The first SIFT stage is followed by a pruning stage where the number of selected key-points is trimmed down. The points that are on the edges or possess low-contrast are rejected because their stability is compromised. The pruning process requires

characterization of each key-point to determine its location, scale and ratio of principal curvatures. These properties are determined with the help of a quadratic 3D curve fitting method applied to the interpolated data nearby a key-point to locate the extrema. While low-contrast points are detected by thresholding the magnitude of the extremum value obtained by the 3D curve fitting method, edge detection requires solving for the ratio of eigenvalues of a 2x2 Hessian Matrix computed at the position and scale of the key-point as shown in Equation (4.6). The ratio between the determinant and the trace of the Hessian matrix generates the ratio between the principle of curvatures, which is thresholded to determine if the key-point lies on edge or not.

$$HL = \begin{bmatrix} L_{xx} & L_{xy} \\ L_{xy} & L_{yy} \end{bmatrix} \quad (4.6)$$

Because SIFT performs a substantial amount of Gaussian filtering, its execution time is dominated by two-dimensional convolution operations which match our architecture. Thus, both filtering and extrema detection can be mapped to the convolution engine.

### 4.2.3 Demosaic

Imaging sensors capture color information with the help of a color filter array (CFA) in which each location captures just one color. Bayer array is one such CFA in which the luminance information (green) is sampled at twice the rate of chrominance (red and blue). The association of green with luminance is derived from the luminance response of the eye which is highest for green. The missing colors are interpolated to get the full image through a procedure called Demosaicing. There are many ways of implementing Demosaicing and we will use the method based upon Adaptive Color Plane Interpolation (ACPI) [27]. ACPI is a multi-pass process that makes use of gradients and second derivative terms to interpolate the missing colors as described in more detail in [45]. ACPI starts by interpolating the missing luminance (green) values and then uses the existing and newly interpolated luminance values to formulate the missing chrominance (red/blue) values.

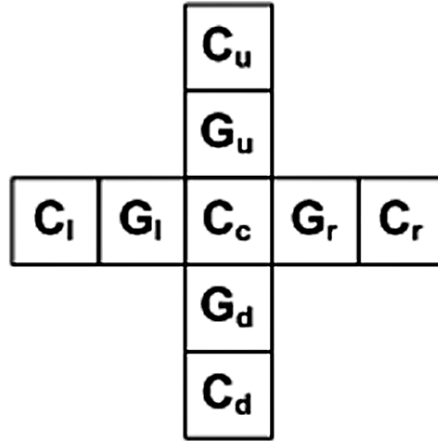


Figure 4.5: Pixels considered for the interpolation of missing luminance values. [45]

**Luminance Interpolation** Interpolation of luminance values constitutes the first pass of the algorithm. Determination of the direction of the biggest change is needed to interpolate the missing green. Consider Figure 4.5 in which  $C_i$  represents chrominance values (red/blue) while  $G_i$  represents luminance or greens. The direction of biggest change is determined by comparing the vertical and the horizontal luminance gradients aided by chrominance second derivatives as shown in Equation (4.7). The goal is to interpolate in the direction of the smallest slope and derivative. Once determined, the direction of interpolation is used to compute an arithmetic average of the luminance weighted by chrominance second derivatives from the same direction as shown in Equation (4.8). If the horizontal and vertical directions display an equal change, the luminance values from all four directions are averaged with the appropriate chrominance second derivatives added in.

$$\begin{aligned}
 Interpolate_{horiz} &= abs(G_R - G_L) + abs(2C_C - C_R - C_L)/2 \\
 Interpolate_{vert} &= abs(G_U - G_D) + abs(2C_C - C_U - C_D)/2
 \end{aligned}
 \tag{4.7}$$

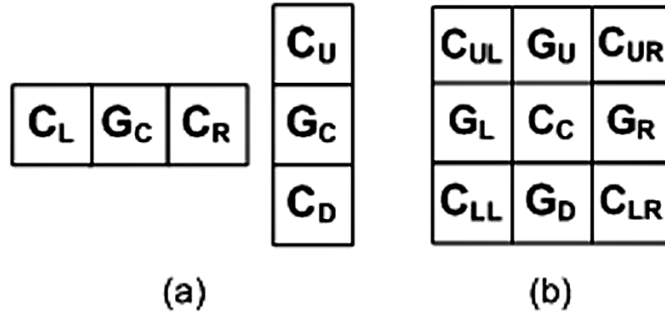


Figure 4.6: Pixels considered for the interpolation of missing chrominance values [45]. Figure a) represents the case when the missing red or blue information is required in the presence of an existing green while figure b) represents the case when the red/blue information is required in a place where either blue or red already exists.

```

if(Interpolatehoriz < Interpolatevert){
    //Interpolate horizontally
    GC = (GL + GR)/2
}elseif(Interpolatevert < Interpolatehoriz){
    //Interpolate vertically
    GC = (GU + GD)/2
}else{
    //Use both directions
    CAVG = (CU + CD + CL + CR)/4
    Ccorrection = CC - CAVG
    GC = (GL + GU + GR + GD)/4 + Ccorrection/2
}

```

(4.8)

**Chrominance Interpolation** In the second pass of the algorithm the missing chrominance information is estimated with the assistance of the newly computed and existing luminance pixels. Figure 4.6 (a) illustrates the two scenarios when the missing chrominance information is sought in the presence of existing greens. In this case an arithmetic average of the colors adjacent to the center green is obtained and

a weighting factor is applied comprised of the second derivative of the green in the center and the newly computed green values at the two locations adjacent to the location of interest. Now, consider Figure 4.6 (b) in which the red/blue information is required in a place where either blue or red already exists. The procedure followed for estimating chrominance in this particular case is similar to the one described for estimating the missing luminance values. The main difference is that the vertical and horizontal directions are replaced by the two diagonals. The direction of the biggest spatial change is determined with the help of the chrominance gradients along the two diagonals aided by the second derivatives of the newly formed greens. An arithmetic average of the chrominance is obtained along the selected diagonal with a weighting factor coming in the form of the second derivatives of the greens. Once again if the two diagonals display directions of an equal spatial change, all four chrominance values are averaged with an appropriate luminance based second derivative term added.

While Demosaic could use a conventional CE, it would first need to compute its gradients. Then it would need to compare the gradients to find out which direction was more stable, and finally using this information it could compute the needed output. But this solution would not be efficient since little computation is done in each step. Since all the information required is available from the original input data, and the total computation is not complex, we use a more complex combining tree to do the entire computation in one step.

#### 4.2.4 Mapping to “Map” and “Reduce” Abstraction

Table 4.1 summarizes the kernels we use and how they map to the map and reduce abstraction. It further describes each algorithm’s data-flow pattern and categorizes the computational model as convolution or fusion. Although, kernels within the table could have identical map and reduce operators and data-flow patterns, they may differ in the way they fetch, manipulate and store the data. This dissimilarity in data manipulation is best illustrated by FME up-sampling, which despite having the same map and reduce operators as regular filtering, produces four times the data of its input image and requires additional hardware support for interleaving the up-sampled



	Map	Reduce	Stencil Sizes	Data-Flow
<b>IME SAD</b>	Abs Diff	Add	4x4	2D Convolution
<b>FME 1/2 Pixel Upsampling</b>	Multiply	Add	6	1D Horizontal And Vertical Convolution
<b>FME 1/4 Pixel Upsampling</b>	Average	None	–	2D Matrix Operation
<b>SIFT Gaussian Blur</b>	Multiply	Add	9, 13, 15	1D Horizontal And Vertical Convolution
<b>SIFT DoG</b>	Subtract	None	–	2D Matrix Operation
<b>SIFT Extrema</b>	Compare	Logical AND	3	1D Horizontal And Vertical Convolution
<b>Demosaic Interpolation</b>	Permute	Fusion	–	2D Graph Fusion

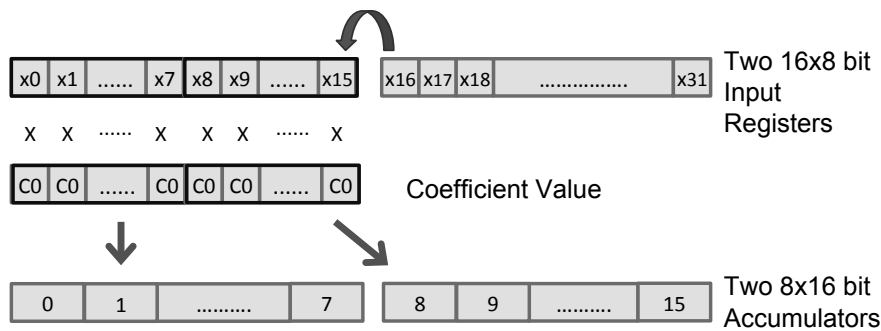
Table 4.1: Mapping kernels to map and reduce abstraction. Some kernels such as subtraction operate on single pixels and thus have no stencil size defined. We call these matrix operations. There is no reduce step for these operations [46].

values before sending them to memory. These special data manipulation requirements differentiate these algorithms from simple filtering and fusion operations and require additional hardware support which we will describe in later sections.

### 4.3 Convolution Engine

$$\begin{aligned}
 Y_0 &= x_0 * c_0 + x_1 * c_1 + x_2 * c_2 + x_3 * c_3 + \dots + x_n * c_n \\
 Y_1 &= x_1 * c_0 + x_2 * c_1 + x_3 * c_2 + x_4 * c_3 + \dots + x_{n+1} * c_n \\
 Y_2 &= x_2 * c_0 + x_3 * c_1 + x_4 * c_2 + x_5 * c_3 + \dots + x_{n+2} * c_n \\
 &\dots
 \end{aligned}$$

Figure 4.7: We use the n-tap 1D convolution presented here to explain our SIMD implementation. For SIMD the equation is parallelized across outputs and executed one column at a time [46].



Core Kernel:

```

Load input
Out0 =0; Out1 = 0;
For I = 1 ... 15
    Load coefficient i
    Out0 = Out0 + Input_Lo * Coeff_i
    Out1 = Out1 + Input_Hi * Coeff_i
    Shift Input Register 0
    Shift Input Register 1
End For
Normalize Output
Store to mem

```

Figure 4.8: 1D Horizontal 16-tap convolution on a 128-bit SIMD machine, similar to the optimized implementation described in [47]. 16 outputs are computed in parallel to maximize SIMD usage. Output is stored in two vector registers and two multiply-accumulate instruction are required at each step [46].

Convolution operators are highly compute-intensive, data-parallel operations particularly for large stencil sizes, and lend themselves to vector processing. However, existing SIMD units are limited in the extent to which they can exploit the inherent parallelism and locality of convolution due to the organization of their register files. Figure 4.7 presents equations for an  $n$ -tap 1D convolution that form the basis of a SIMD based convolution implementation presented in Figure 4.8. We demonstrate in Figure 4.8 the limitations of a SIMD based convolution implementation by executing a 16-tap convolution on a 128-bit SIMD datapath. This is a typical SIMD implementation similar to the one presented in [47], and the SIMD datapath is similar to ones found in many current processors. To enable the datapath to utilize the vector

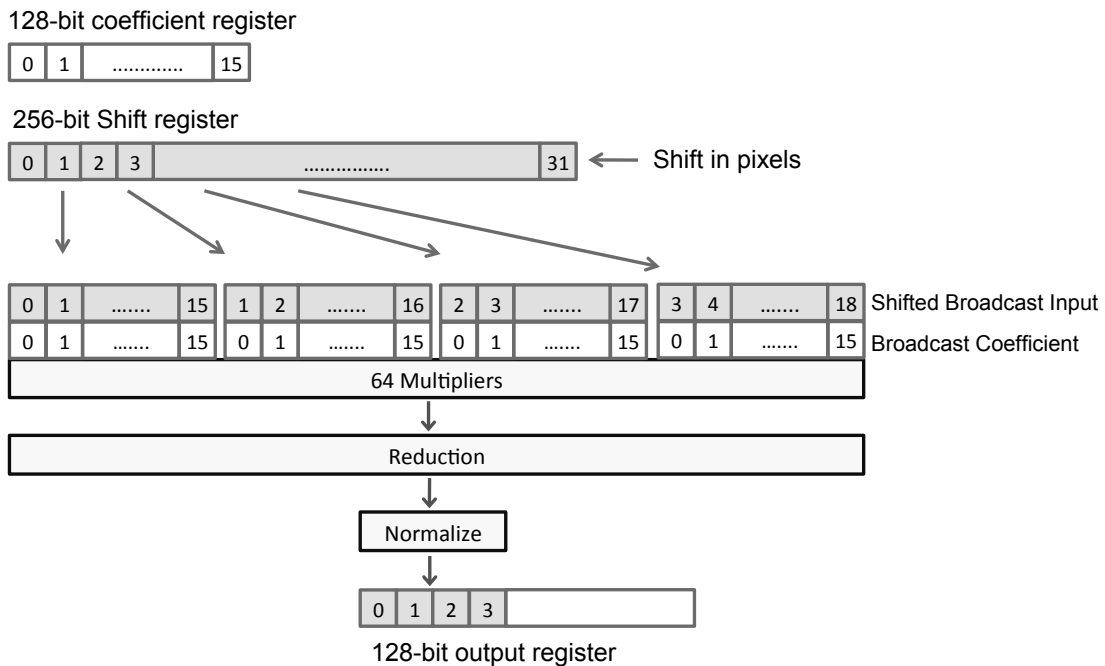


Figure 4.9: 1D Horizontal 16-tap convolution using a shifter register with shifted broadcast capability. Computes 4 output pixels per instruction [46].

registers completely irrespective of the filter size, the convolution operation is vectorized across output locations allowing the datapath to compute eight output values in parallel.

Given the short integer computation that is required, one needs a large amount of parallelism per instruction to be energy efficient [40]. While this application has the needed parallelism, scaling the datapath by eight times to perform sixty four 16-bit operations per cycle would prove extremely costly. It would require an eight times increase in the register file size, inflating it to 1024-bits, greatly increasing its energy and area. To make matters worse, as shown in Figure 4.8, the energy efficiency of the SIMD datapath is further degraded by the fact that a substantial percentage of instructions are used to perform data shuffles which consume instruction and register energy without doing any operations. Alternatively, one can reload shifted versions of vectors from the memory to avoid data shuffles; however, that also results in substantial energy waste due to excessive memory fetches. These data

motion overheads are worse for vertical and 2-D convolution.

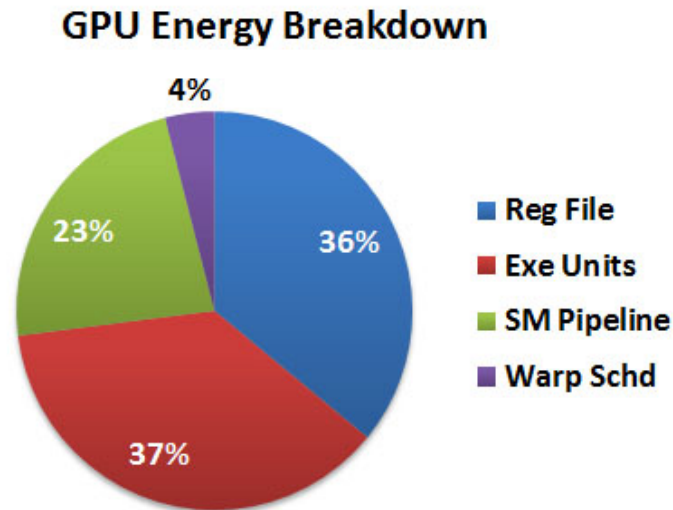


Figure 4.10: The pie chart presents the GPU energy breakdown. The energy consumed in the memory structures is not included. As can be seen, streaming multiprocessor (SM) pipeline (37%) and the register file (36%) consume the most amount of energy followed by execution units (23%) and the warp scheduler (4%).

GPUs target massively data parallel applications and can achieve much higher performance for convolution operations than SIMD. However, due to their large register file structures and 32-bit floating point units, we don't expect GPUs to have very low energy consumption. To evaluate this further we measure the performance and energy consumption of an optimized GPU implementation of H.264 SAD algorithm [48] running on an NVIDIA GTX480 [49] using GPGPU-Sim simulator [50] with GPUWatch energy model [51]. The GPU implementation runs forty times faster compared to an embedded 128-bit SIMD unit, but consumes thirty times higher energy. The energy breakdown is presented in Figure 4.10 and shows that streaming multiprocessor pipeline and the register files consume the most amount of energy. Even with a GPU customized for media applications we do not expect the energy consumption to be much better than the SIMD implementation as the GPU energy is dominated by register file, which is central to how GPUs achieve their high degree

of parallelism.

CE reduces most of the register file overheads described earlier with the help of a shift register file or a FIFO like storage structure. As shown in Figure 4.9, when such a storage structure is augmented with an ability to generate multiple shifted versions of the input data, it can not only facilitate execution of multiple simultaneous stencils, but can also eliminate most of the shortcomings of traditional vector register files. Aided by the ability to broadcast data, these multiple shifted versions can fill sixty four ALUs from a small 256-bit register file saving valuable register file access energy as well as area.

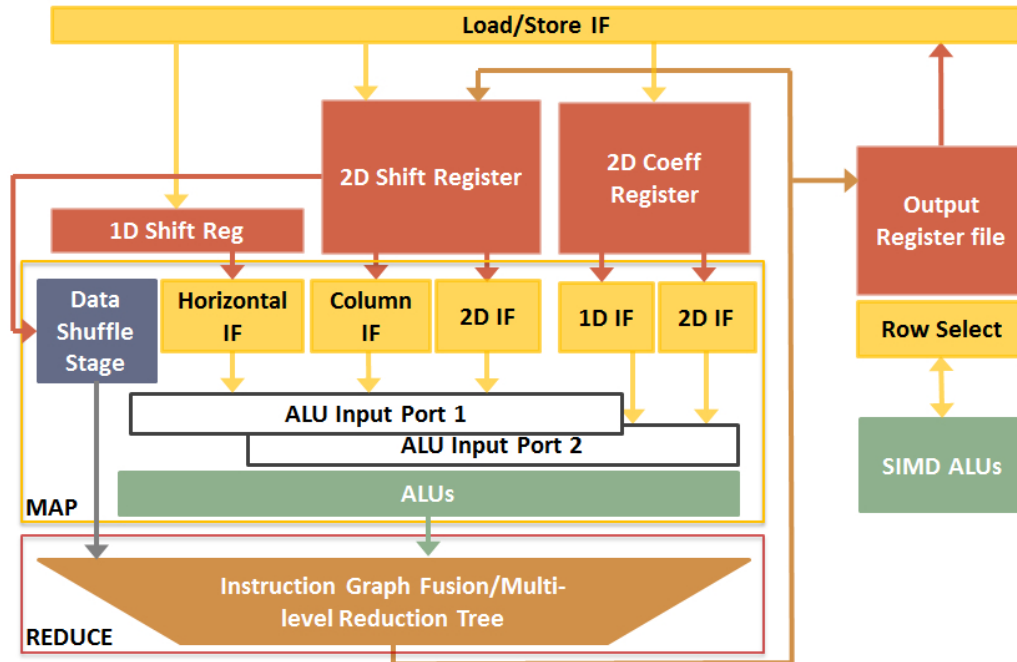


Figure 4.11: Block diagram of Convolution Engine. The interface units (IF) connect the register files to the functional units and provide shifted broadcast to facilitate convolution. Data shuffle (DS) stage combined with Instruction Graph Fusion (IGF) stage form the Complex Graph Fusion Unit. IGF is integrated into the reduction stage for greater flexibility [46].

Our CE facilitates further reductions in energy overheads by supporting more complex operation in the reduction tree, allowing multiple “instructions” to be fused

together. This fusion also offers the added benefit of eliminating temporary storage of intermediate data in big register files saving valuable register file energy. Furthermore, by changing the shift register to have two dimensions, and by allowing column accesses and two dimensional shifts, these shift registers possess the potential to extensively improve the energy efficiency of vertical and two dimensional filtering. As Figure 4.11 shows, these 1D and 2D shift registers sit at the heart of our Convolution Engine.

Using Tensilica's TIE language [52], the CE is developed as a domain specific hardware extension to Tensilica's extensible RISC cores [20]. To augment the CE with the ability to run multiple independent threads, user-defined hardware interfaces called TIE ports are added to allow multiple RISC cores to share the CE. The TIE ports enable routing of appropriate control signals to the CE from the RISC cores at the granularity of individual instructions. Since the number of cores interfacing with CE could be more than one, the TIE ports are muxed. The cores are also responsible for memory address generation, but the data is sent/return directly from the register files within CE. The next sections discuss the key blocks depicted in Figure 4.11.

### 4.3.1 Register Files and the Load/Store Unit

The Convolution Engine uses a 1D shift register to supply data for horizontal convolution flow. New image pixels are shifted horizontally into the 1D register as the 1D stencil moves over an image row. The 2D shift register is used for vertical and 2D convolution flows and supports vertical row shift: one new row of pixel data is shifted in as the 2D stencil moves vertically down into the image. The 2D register provides simultaneous access to all of its elements enabling the interface unit to feed any data element into the ALUs as needed.

The 2D Coefficient Register stores data that does not change as the stencil moves across the image. This can be filter coefficients, current image pixels in IME for performing SAD, or pixels at the center of Windowed Min/Max stencils. The results of filtering operations are either written back to the 2D Shift Register or the Output Register. The Output Register is designed to behave both as a 2D Shift register as well as a Vector Register file for the vector unit. The shift behavior is invoked when

the output of the stencil operation is written. This shift simplifies the register write logic and reduces the energy. This is especially useful when the stencil operation produces the data for just a few locations and the newly produced data needs to be merged with the existing data which results in a read modify write operation. The Vector Register file behavior is invoked when the Output Register file is interfaced with the vector unit shown in the Figure 4.11.

The data transfer between these register files and the main memory is handled by the load/store unit. To improve efficiency, it supports multiple memory access widths with the maximum memory access width being 256-bits and can handle unaligned accesses. These load/store operations can be coupled with the shift operations mentioned above.

### 4.3.2 Map & Reduce Logic

As described earlier we abstract convolution as a *map* and *reduce* step that transforms each input pixel into an output pixel. In our implementation interface units and ALUs work together to implement the *map* operation; the interface units arrange the data as needed for the particular map pattern and the functional units perform the arithmetic.

#### Interface Units

The Interface Units (IF) arrange data from the register files into a specific pattern needed by the map operation. Currently this includes providing shifted versions of 1D and 2D blocks, and column access to 2D register, though we are also exploring a more generalized permutation layer to support arbitrary maps. All of the functionality needed for generating multiple shifted versions of the data is encapsulated within the IFs. This allows us to shorten the wires by efficiently generating the needed data within one block while keeping the rest of the datapath simple and relatively free of control logic. Since the IFs are tasked to facilitate stencil based operations, the multiplexing logic remains simple and prevents the IFs from becoming the bottleneck.

The Horizontal Interface generates multiple shifted versions of the 1D data and

feeds them to the ALU units. The data arrangement changes depending on the size of the stencil so this unit supports multiple power of 2 stencil sizes and allows selecting between them. Column Interface simultaneously access the columns of the 2D Shift register to generate input data for multiple locations of a vertical 1D filtering kernel. The 2D interface behaves similarly to the Vertical interface and accesses multiple shifted 2D data blocks to generate data for multiple 2D stencil locations. Again multiple column sizes and 2D block sizes are supported and the appropriate one is selected by the convolution instruction.

### Functional Units

Since all data re-arrangement is handled by the interface unit, the functional units are just an array of short fixed point two-input arithmetic ALUs. In addition to multipliers, we support absolute difference to facilitate SAD and other typical arithmetic operations such as addition, subtraction, comparison. The output of the ALU is fed to the Reduce stage.

### Reduce Unit

The *reduce* part of the *map-reduce* operation is handled by a general purpose reduce stage. Based upon the needs of our applications, we currently support arithmetic and logical reduction stages. The degree of reduction is dependent on the kernel size, for example a 4x4 2D kernel requires a 16 to 1 reduction whereas 8 to 1 reduction is needed for an 8-tap 1D kernel. The reduction stage is implemented as a tree and outputs can be tapped out from multiple stages of the tree; allowing multiple outputs to be generated each cycle.

### 4.3.3 Instruction Graph Fusion

As described earlier, we increase the domain of applications which can effectively use the CE by creating a combining tree that is more powerful than what is required for true reductions. The ability to handle non-commutative operations enables us to merge many different convolution instructions into a single fused “super instruction”.



The fused instruction allows a small program to be executed for each pixel in one convolution instruction, which increases the computational efficiency proportionally to the reduction in required instructions.

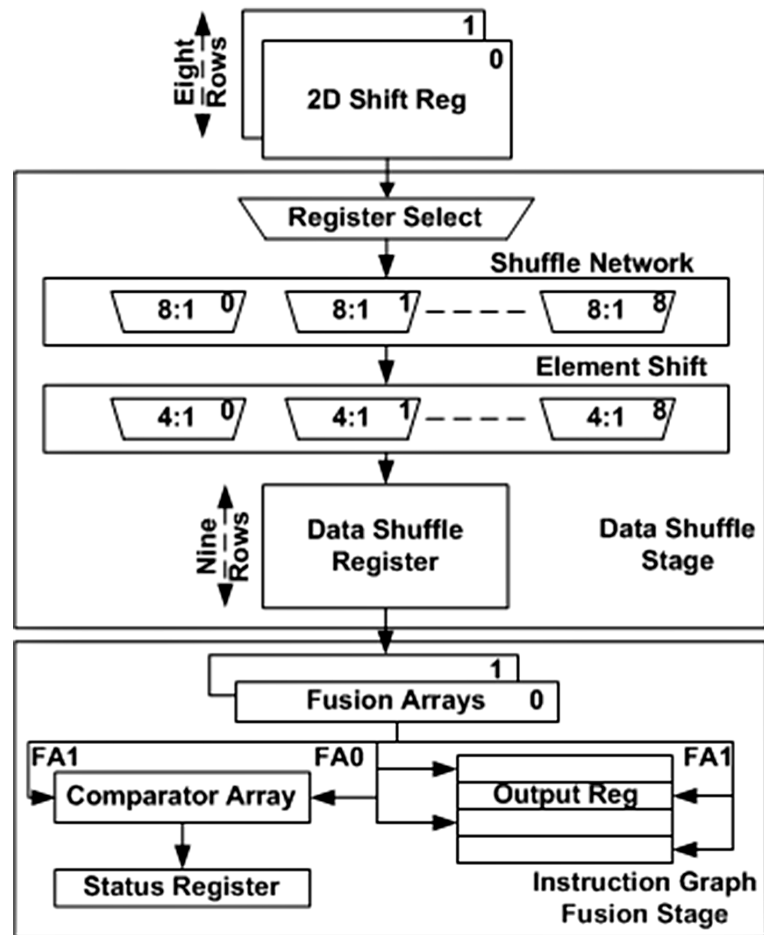


Figure 4.12: An overview of Complex Graph Fusion Unit. The “Shuffle Network” in the “Data Shuffle Stage” is responsible for selecting one out of eight rows of data from the register files. The selected data is shifted by “Element Shift” at the granularity of individual elements, and stored in the dedicated “Data Shuffle Register”. The “Fusion Arrays” in the “Instruction Graph Fusion” stage represent the generalized reduction stage and perform fusion. The “Comparator Array” updates the “Status Register” after comparing the outputs of fusion arrays [46].

In CE, this extra capability is provided by the programmable complex graph fusion unit (CGFU) presented in Figure 4.12. CGFU is comprised of a Data Shuffle Stage

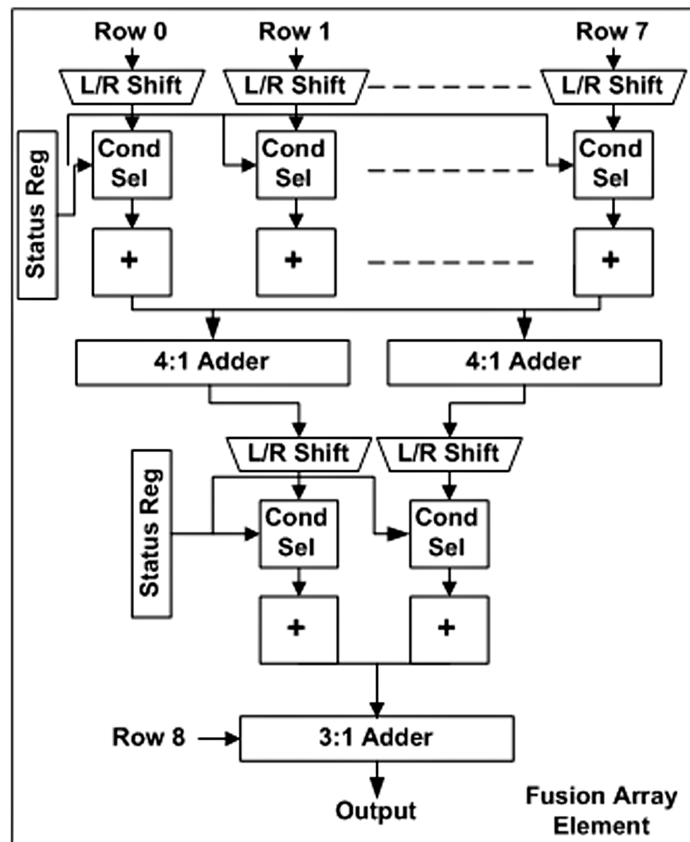


Figure 4.13: An overview of Instruction Graph Fusion (IGF) stage. The data from the dedicated “Data Shuffle Register” powers the IGF, which is organized as a generalized reduction network of functional units. Each FU is individually configurable and supports predicated execution [46].

which forms the *map* step followed by Instruction Graph Fusion Stage, which forms the *reduce* step. The CGFU has the ability to fuse together up to nine arithmetic instructions and obtains its input from both the input and the output registers.

### Data Shuffle

Since this more complex data combination is not commutative, the right data (output of the map operation) must be placed on each input to the combining network. Thus the CGFU includes a very flexible swizzle network that provides permutations of the input data and sends it to a shifter unit which takes the shuffled data to perform

element level shifts. These two units combine to form a highly flexible swizzle network that can reorder the data to support 1D horizontal, 1D vertical and even 2D window based fusions. However, this flexibility costs energy, so it is bypassed on standard convolution instructions. Often multiple fusion instructions use the data after it is shuffled, which we exploit to reduce the energy wasted in register file accesses by introducing a dedicated storage structure, called Data Shuffle Register file, between the data shuffle stage (DS) and the actual instruction graph fusion stage.

### Complex Graph Fusion

While the DS stage is tasked with data reordering, the Instruction Graph Fusion (IGF) stage is responsible for executing the more complex data combining to implement the fused instruction sub-graphs. The most critical parts of the IGF stage are the two fusion arrays which are shown in Figure 4.13. Each array supports a variety of arithmetic operations and can implement data dependent data-flow by using predicated execution. These units are pipelined, so bits of the Status Register which are set from computation early in the combining tree can be used later in the computation to generate the desired output. Like the normal reduction network, the outputs of the two arrays are also fed to a two dimensional output register where they are stored in pairs. The absorption of IGF into the reduction stage does entail higher energy costs for convolution operations, but our experiments indicate that the overheads remain less than 15%.

#### 4.3.4 Lightweight SIMD

To facilitate vector operations on the output data, we have added a 16-element SIMD unit that interfaces with the Output Register. This unit accesses the 2D Output Register as a Vector Register file to perform regular Vector operations. This is a lightweight unit which only supports basic vector add and subtract type operations and has no support for higher cost operations such as multiplications found in a typical SIMD engine.

### 4.3.5 Custom Functional Units

An application may perform computation that conforms neither to the convolution block nor to the vector unit, or may otherwise benefit from a fixed function implementation. If the designer wishes to build a customized unit for such computation, the Convolution Engine allows the fixed function block access to its Output Register File. This model is similar to a GPU where custom blocks are employed for rasterization and such, and that work alongside the shader cores. For these applications, we created three custom functional blocks to compute motion vector costs in IME and FME and the Hadamard Transform in FME.

### 4.3.6 A 2-D Filter Example

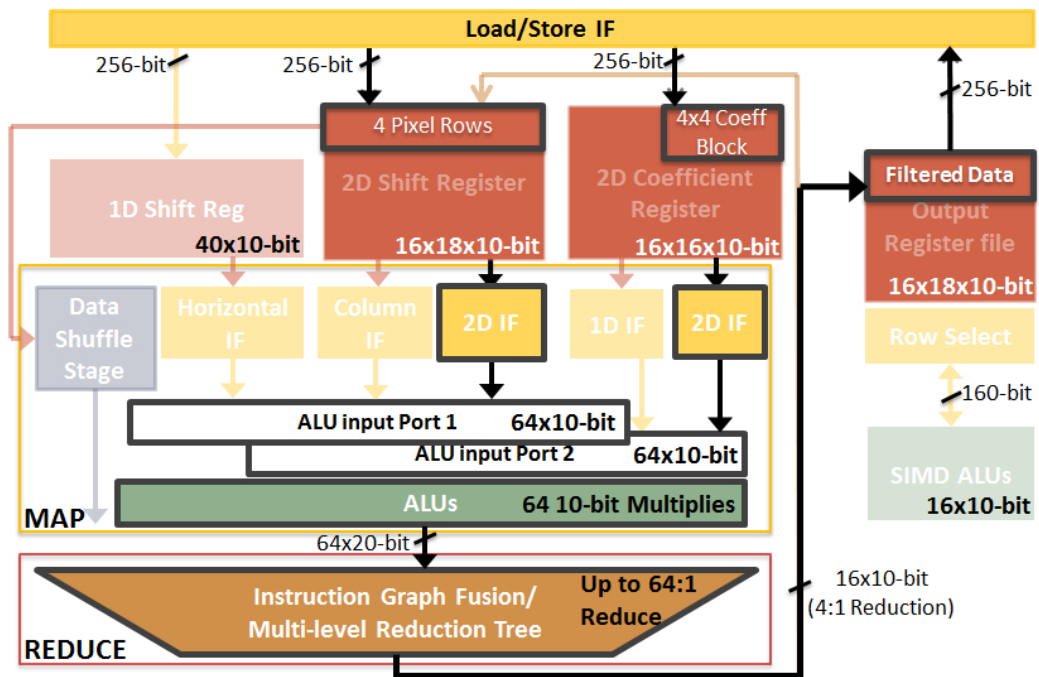


Figure 4.14: Executing a 4x4 2D Filter on CE. The boxes displayed in a lighter color represent units not used in the example. The sizes of all of the resources are defined which will be explained in a later section [46].

Figure 4.14 shows how a 4x4 2D filtering operation maps onto the convolution

engine. Filter coefficients reside in first four rows of the Coefficient Register. Four rows of image data are shifted into the first four rows of the 2D Shift register. In this example we have 64 functional units so we can perform filtering on up to four 4x4 2D locations in parallel. The 2D Interface Unit generates four shifted versions of 4x4 blocks, lays them out in 1D and feeds them to the ALUs. The Coefficient Register Interface Unit replicates the 4x4 input coefficients 4 times and send them to the other ALU port. The functional units perform an element-wise multiplication of each input pixel with corresponding coefficients and the output is fed to the Reduction stage. The degree of reduction to perform is determined by the filter size which in this case is 16:1. The four outputs of the reduction stage are normalized and written to the Output Register.

Since our registers contain data for sixteen filter locations, we continue to perform the same operation described above; however, the 2D Interface Unit now employs horizontal offset to skip over already processed locations and to get the new data while the rest of the operations execute as above. Once we have filtered sixteen locations, the existing rows are shifted down and a new row of data is brought in and we continue processing the data in the vertical direction. Once all the rows have been processed we start over from the first image row, processing next vertical stripe and continue execution until the whole input data has been filtered.

For symmetric kernels the interface units combine the symmetric data before coefficient multiplication (since the taps are the same), allowing it to use adders in place of multipliers. Since adders take 2–3x lower energy, this further reduces wasted energy. The load/store unit also provides interleaved access where data from a memory load is split and stored into two registers. An example use is in demosaic, which needs to split the input data into multiple color channels.

### 4.3.7 Resource Sizing

Energy efficiency and resource requirements of target applications drive the sizes of various resources within the CE. Energy overheads such as instruction fetch and decode affect the efficiency of programmable systems and can only be amortized

Table 4.2: Sizes for various resources in CE.

	<b>Resource Sizes</b>
<b>ALUs</b>	64 10-bit ALUs
<b>1D Shift Reg</b>	1 row x 40 cols x 10-bit
<b>2D Input Shift Reg</b>	16 rows x 18 cols x 10-bit
<b>2D Output Shift Register</b>	16 rows x 18 cols x 10-bit
<b>2D Coefficient Register</b>	16 rows x 16 cols x 10-bit
<b>Horizontal Interface</b>	4, 8, 16 kernel patterns
<b>Vertical Interface</b>	4, 8, 16 kernel patterns
<b>2D Interface</b>	4x4, 8x8 , 16x16 patterns
<b>Reduction Tree</b>	4:1, 8:1, 16:1, 32:1, 64:1

Table 4.3: Energy for filtering instructions implemented as processor extensions with 32, 64 or 128 ALUs in 90nm. Overhead is the energy for instruction fetch, decode and sequencing [46].

	<b>32 ALUs</b>	<b>64 ALUs</b>	<b>128 ALUs</b>
<b>Total Energy (pJ)</b>	468	939	1632
<b>Overhead Energy (pJ)</b>	111	117	132
<b>Percent Overhead</b>	24	12	8

by performing hundreds of arithmetic operations per instruction as shown in [40]. However, the authors in [40] studied small data such as 8-bit addition/subtraction, while convolution is typically dominated by multiplication that takes more energy per operation. To determine how to size the ALUs for the CE with the goal of keeping overheads as low as possible, we present the energy dissipated in executing filtering instructions using thirty-two, sixty-four and one-hundred-twenty-eight 10-bit ALUs (the precision required) in Table 4.3. In this table the total energy is comprised of the energy wasted in the processor overheads including fetch, decode and sequencing as well as the useful energy spent in performing the actual compute. As the number of ALUs increases, the overhead energy as a percentage of the total energy reduces. We choose sixty-four as the number of ALUs in the CE as a compromise between efficiency and flexibility because it is easier to chain small units. The rest of the resources are sized to keep sixty-four 10-bit ALUs busy. The size and capability of each resource is presented in Table 4.2. These resources support filter sizes of 4, 8

and 16 for 1D filtering and 4x4, 8x8 and 16x16 for 2D filtering. Notice that that the register file sizes deviate from power of 2; this departure allows us to handle boundary conditions common in convolution operations efficiently.

### 4.3.8 Convolution Engine CMP

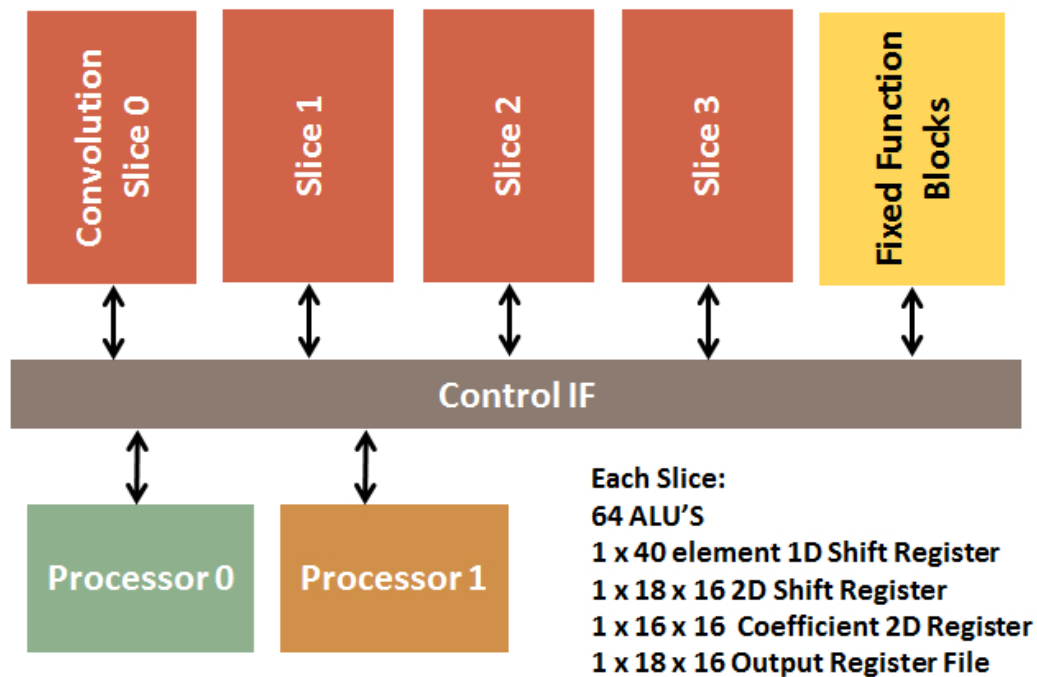


Figure 4.15: Communication connections among various components of the Convolution Engine CMP. The two Tensilica RISC processors use their TIE ports to provide control signals to the four CE slices and the fixed function blocks. The control interface (IF) acts as the arbiter between requests in the case of conflicts [46].

To meet the diverse performance and energy requirements of different applications effectively, we have developed a CE chip multiprocessor (CMP) shown in Figure 4.15. The CMP consists of four CEs and two Tensilica's extensible RISC processors communicating with the CEs through muxed TIE ports as described earlier in this section. The decision to support two independent threads of control in the form of two processors is influenced largely by the requirements of the applications of interest, but

also to a lesser extent by energy efficiency as smaller TIE port muxes keep energy wasted per instruction low. In the CMP, each instance of the CE is referred to as a slice and the slices possess the capability to either operate independent of other slices or in concatenation to perform an even larger number of operations per cycle. Dynamic concatenation of slices is especially desirable when the performance requirements of an algorithm cannot be satisfied by one slice or when the algorithm operates on small data requiring more than 64 operations per cycle to amortize overheads. When the slices are concatenated dynamically the register files and interface units of the interconnected slices are joined through short wires that run from one slice to another. Since the slices are laid out in close proximity to one another as shown in Figure 4.15, these wires waste very little energy and don't have a large effect on the energy efficiency of connected slices. In addition to connecting multiple slices together to form a bigger slice with wide registers and ALU arrays, it is also possible to shut off the ALUs in the additional slices and use their registers as additional independent storage structures. Although, all the slices offer the same functionality, slices 0 and 1 are also equipped with complex graph fusion units integrated into their reduction blocks. The side effect of this integration is the additional 10–15% cost incurred by convolution operations executed on these slices. The processors and slices are fed by dual-ported 16K instruction and 32K data caches. As has been discussed earlier, the processors are responsible for data address generation for the connected slices, but the flow of data into and out of the data cache is controlled by the slices themselves.

### 4.3.9 Programming the Convolution Engine

The Convolution Engine is implemented as a processor extension and adds a small set of instructions to the processor ISA. These CE instructions can be issued as needed in regular C code through compiler intrinsics. Table 4.4 lists the major instructions that CE adds to the ISA and Listing 4.1 presents a simplified example code which implements a 15-tap horizontal filter for a single image row. There are four types of instructions: configuration, memory, compute and permute. Configuration instructions set options which are expected to stay fixed for a kernel such as convolution size,



Table 4.4: Major instructions added to processor ISA [46].

	<b>Instructions</b>	<b>Description</b>
<b>Configuration</b>	<b>SET_CE_OPS</b>	Set arithmetic functions for MAP and REDUCE steps
	<b>SET_CE_OPSIZE</b>	Set convolution size
<b>Memory</b>	<b>LD_COEFF_REG_n</b>	Load n bits to specified row of 2D coeff register
	<b>LD_1D_REG_n</b>	Load n bits to 1D shift register. Optional shift left
	<b>LD_2D_REG_n</b>	Load n bits to top row of 2D shift register. Optional shift row down
	<b>ST_OUT_REG_n</b>	Store top row of 2D output register to memory
<b>Compute</b>	<b>CONVOLVE_1D_HOR</b>	1D convolution step - input from 1D shift register
	<b>CONVOLVE_1D_VER</b>	1D convolution step - column access to 2D shift register
	<b>CONVOLVE_2D</b>	2D Convolution step with 2D access to 2D shift register
	<b>EXE_FUSION</b>	Performs instruction graph fusion
	<b>SIMD_1D</b>	Execute SIMD operations
<b>Permutation</b>	<b>EXE_SHUFFLE</b>	Shuffles input data

ALU operation to use etc. Other options which can change on a per instruction basis are specified as instruction operands. Then there are memory operations to load and store data into appropriate registers as required. There is one load instruction for each input register type (1D input register, 2D input register, Coefficient register). Then there are the compute instructions, one for each of the 3 supported convolution flows — 1D horizontal, 1D vertical and 2D. For example the CONVOLVE\_2D instruction reads one set of values from 2D and coefficient registers, performs the convolution and writes the result into the row 0 of the 2D output register. The load,

store and compute instructions are issued repeatedly as needed to implement the required algorithm. Finally, there are the permutation instruction used to implement CGFU.

```

// Set Number of slices
SET_NUM_SLICES(2);

// Set MAP function = MULT, Reduce function = ADD
SET_CE_OPS (CE_MULT, CE_ADD);

// Set convolution size 16, mask out 16th element
SET_CE_OPSIZE(16, 0x7fff);

// Load 16 8-bit coefficients into Coeff Reg Row 0
LD_COEFF_REG_128(coeffPtr, 0);

// Load & shift 16 input pixels into 1D shift register
LD_1D_REG_128(inPtr, SHIFT_ENABLED);

// Filtering loop
for (x = 0; x < width - 16; x += 16) {
    // Load & Shift 16 more pixels
    LD_1D_REG_128(inPtr, SHIFT_ENABLED);

        //With a convolution size of 16, using 128 ALUs
        //8 positions can be done in parallel

    // Filter first 8 locations
    CONVOLVE_1D_HOR(IN_OFFSET_0, OUT_OFFSET_0);

    // Filter next 8 locations
    CONVOLVE_1D_HOR(IN_OFFSET_8, OUT_OFFSET_8);

    // Add 2 to row 0 of output register
    SIMD_ADD_CONST (0, 2);

    // Store 16 output pixels
    ST_OUT_REG_128(outPtr);

    inPtr += 16;
    outPtr += 16;
}

```

Listing 4.1: Example C code implements a 15-tap filter for one image row on a two slice configuration using 128 ALUs. A scalar value of 2 is added to each output before storing to memory [46].

The code example in Listing 4.1 brings together configuration, memory and compute. First the CE is set to perform multiplication at MAP stage and addition at reduce stage which are the required setting for filtering. The convolution size is set

which controls the pattern in which data is fed from the registers to the ALUs. Filter tap coefficients are then loaded into the coefficient register. Finally the main processing loop repeatedly loads new input pixels into the 1D register and issues 1D\_CONVOLVE operations to perform filtering. While 16 new pixels are read with every load, our 128-ALU CE configuration can only process eight 16-tap filters per operation. Therefore two 1D\_CONVOLVE operations are performed per iteration, where the second operation reads the input from an offset of 8 and writes its output at an offset of 8 in the output register. For illustration purposes we have added a SIMD instruction which adds 2 to the filtering output in row 0 of 2D output register. The results from output register are written back to memory.

It is important to note that unlike a stand-alone accelerator the sequence of operations in CE is completely controlled by the software which gives complete flexibility over the algorithm. Also CE code is freely mixed into the C code which gives added flexibility. For example in the filtering code above it is possible for the algorithm to produce one CE output to memory and then perform a number of non-CE operations on that output before invoking CE to produce another output.

The next section describes how we map different applications to a Convolution Engine based CMP and the experiments we perform to determine the impact of programmability on efficiency. By incrementally enabling these options on top of a fixed kernel core we can approach the fully programmable CE in small steps and assess the energy and area cost of each addition.

## 4.4 Evaluation Methodology

To evaluate the Convolution Engine approach, we map each computationally bound target application described in Section 4.2 on a CE based CMP. As already discussed, this system is fairly flexible and can easily accommodate algorithmic changes such as different motion estimation block sizes and different downsampling techniques. To quantify the performance and energy cost of such a programmable unit, we also built custom heterogeneous chip multiprocessors (CMP) for each of the three applications. These custom CMPs are based around application-specific cores, each of which is

highly specialized and only has resources to do a specific kernel required by the application. Both the CE and application-specific cores are built as a datapath extension to the processor cores using Tensilica’s TIE language [52]. Tensilica’s TIE compiler uses this description to generate simulation models and RTL as well as area estimates for each extended processor configuration.

To quickly simulate and evaluate the CMP configurations, we created a multi-processor simulation framework that employs Tensilica’s Xtensa Modeling Platform (XTMP) to perform cycle accurate simulation of the processors and caches. For energy estimation we use Tensilica’s energy explorer tool, which uses a program execution trace to give a detailed analysis of energy consumption in the processor core as well as the memory system. The estimated energy consumption is within 30% of actual energy dissipation. To account for interconnection energy, we created a floor plan for the CMP and estimated the wire energies from that. That interconnection energy was then added to energy estimates from Tensilica tools. The simulation results employ 90nm technology at 1.1V operating voltage with a target frequency of 450MHz. All units are pipelined appropriately to achieve the frequency target.



Figure 4.16: Mapping of applications to CE CMP [46].

Figure 4.16 presents how each application is mapped to our CE based CMP. This mapping is influenced by the application’s performance requirements. In this study we support HD 720P video at 30FPS. This translates to an input data rate of around 30 MPixels/s. For still images we want to support a similar data rate of around 40–50 MPixels/s which can be translated for example to processing 5MP images at 8–10FPS or 3MP images at a higher rate of 13–16FPS etc. H.264 motion estimation only deals with video data, whereas SIFT and Demosaic can be applied to both video and still images. However, when SIFT and Demosaic are applied to 720p HD video streams

the resolution drops from 5MP to 1MP increasing the frame rate substantially. More details on the mapping of each application are given below.

#### 4.4.1 H.264 Motion Estimation

Our mapping allocates one processor to the task of H.264 integer motion estimation. The 4x4 SAD computation is mapped to the convolution engine block, and the SIMD unit handles the task of combining these to form the larger SAD results. This requires a 16x32 2D shift register and 128 ABS-DIFF ALU units, so 2 slices are allocated to this processor. In addition a fixed function block is used to compute motion vector cost, which is a lookup-table based operation. Fractional motion estimation uses up only 64 ALU units, but requires multiple register files to handle the large amount of data produced by up-sampling, so it takes up 2 slices. The convolution engine handles up-sampling and SAD computation. A custom fixed function block handles the Hadamard transform.

#### 4.4.2 SIFT

Each level in the SIFT Gaussian pyramid requires five 2D Gaussian blur filtering operations, and then down-sampling is performed to go to the next level. The various Gaussian blurs, the difference operation and the down-sampling are all mapped to one of the processors, which uses one convolution engine slice. The Gaussian filtering kernel is a separable 2D filtering kernel so it is implemented as a horizontal filter followed by a vertical filter. The second processor handles extrema detection, which is a windowed min/max operation followed by thresholding to drop weak candidates. This processor uses 2 slices to implement the windowed min across 3 difference images and SIMD operations to perform the thresholding. SIFT generates a large amount of intermediate pyramid data, therefore 64x64 image blocking is used to minimize the intermediate data footprint in memory. The minima operation crosses block boundaries so buffering of some filtered image rows is required. Moreover, the processing is done in multiple passes, with each pass handling each level of the pyramid.

### 4.4.3 Demosaic

Demosaic generates a lot of new pixels and intermediate data and thus needs multiple 2D shift register files. It uses register resources from two slices and further uses register blocking to get multiple virtual registers from the same physical register. Demosaicing is the first step in a typical camera pipeline. In our current mapping, the second processor and remaining two slices are idle when demosaicing is in operation. However, these resources can be used to implement next steps in the imaging pipeline such as white balance, denoising, and sharpening which are also based on convolution-based kernels.

## 4.5 Results

Figures 4.17 and 4.18 compare the performance and energy dissipation of the proposed Convolution Engine against a 128-bit data-parallel (SIMD) engine and an application specific accelerator implementation for each of the five algorithms of interest. In most cases we used the SIMD engine as a 16-way 8-bit datapath, but in a few examples we created 8-way 16-bit datapaths. For our algorithms, making this unit wider did not change the energy efficiency appreciably.

The fixed function data points truly highlight the power of customization: for each application a customized accelerator required 8x–50x less energy compared to an optimized data-parallel engine. Performance per unit area improves a similar amount, 8x–30x higher than the SIMD implementation. Demosaic achieves the smallest improvement (8x) because it generates two new pixel values for every pixel that it loads from the memory. Therefore, after the customization of compute operations, loads/stores and address manipulation operations become the bottleneck and account for approximately 70% of the total instructions.

Note the biggest gains were in IME and SIFT extrema calculations. Both kernels rely on short integer *add/subtract* operations that are very low energy (relative to the *multiply* used in filtering and up-sampling). To be efficient when the cost of compute is low, either the data movement and control overhead should be very low, or more

operations must be performed to amortize these costs. In a SIMD implementation these overheads are still large relative to the amount of computation done. These kernels also use a 2D data flow which requires constant accesses and fetches from the register file. Custom hardware, on the other hand, achieves better performance at lower energy by supporting custom 2D data access patterns. Rather than a vector, it works on a matrix which is shifted every cycle. Having more data in flight enables a larger number arithmetic units to work in parallel, better amortizing instruction and data fetch.

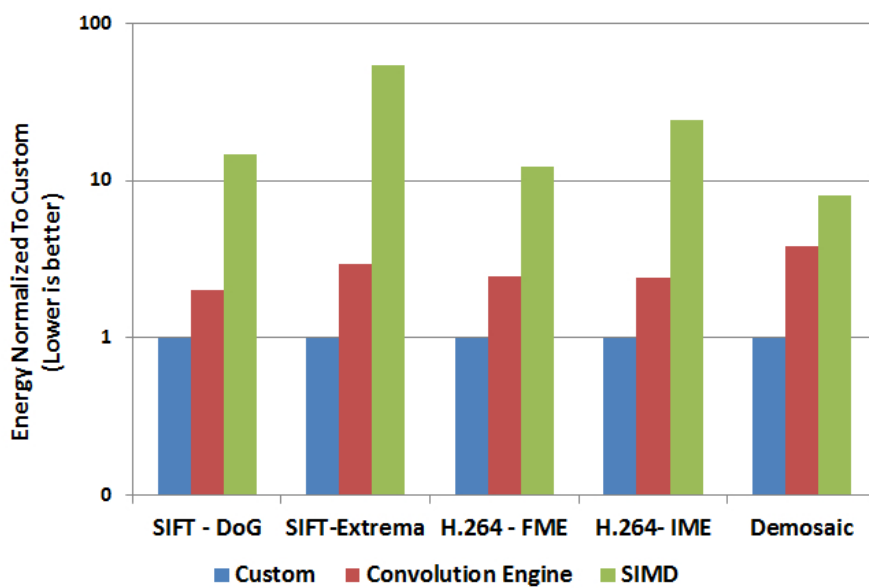


Figure 4.17: Energy consumption normalized to Custom implementation: Convolution Engine vs Custom Cores and SIMD [46].

With this analysis in mind, we can now better understand where a Convolution Engine stands. The architecture of the Convolution Engine is closely matched to the data-flow of convolution based algorithms, therefore the instruction stream difference between fixed function units and the Convolution Engine is very small. Compared to a SIMD implementation, the convolution engine requires 8x–15x less energy with the exception of Demosaic that shows an improvement of 4x while the performance to area ratio of CE is 5–6x better. Again Demosaic is at the low end of the gain as a consequence of the abundance of loads and stores. If we discount the effect of memory

operations from Demosaic, assuming its output is pipelined into another convolution like stage in the image pipeline, the CGFU based Demosaic implementation is approximately 7x better than SIMD and within 6x of custom accelerator. The higher energy ratio compared to a custom implementation points up the costs of the more flexible communication in CGFU compared to CE's blocks optimized for convolution.

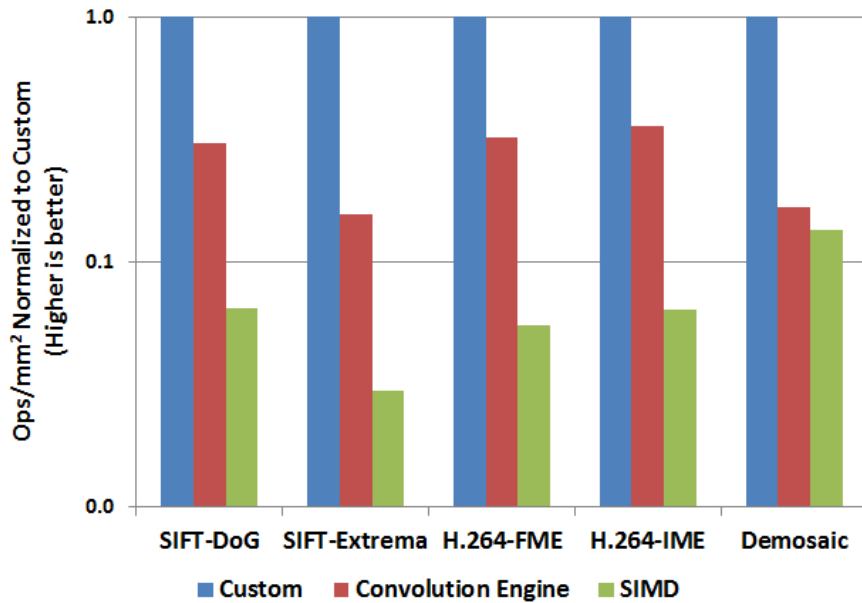


Figure 4.18: Ops/ $mm^2$  normalized to Custom implementation: Number of image blocks each core processes in one second, divided by the area of the core. For H.264 an image block is a 16x16 macroblock and for SIFT and demosaic it is a 64x64 image block [46].

The energy overhead of the CE implementation over application specific accelerator is modest (2–3x) for the other applications, and requires only twice the area. While these overheads are small, we explore the sources of these overheads in the next section.



### 4.5.1 Generating Instances with Varying Degrees of Flexibility

To study the impact of various programmability options, we have designed the CE in a highly parameterized way such that we can generate instances with varying degrees of flexibility ranging from fixed kernel to fully programmable CE. The results of this analysis are shown in Figures 4.19 and 4.20. The fixed 1-D convolution kernel instance shown in Figure 4.9 is used, the whole 2D register with its associated interface unit goes away. The 1D interface also goes away, replaced by the hardwired access pattern required for the particular kernel. The remaining registers are sized just large enough to handle the particular kernel, the flexible reduction tree is replaced by a fixed reduction and the ALU only supports the single arithmetic operation needed. The efficiency of this fixed kernel datapath should match custom cores. The programmability options that convolution engine has over this fixed kernel datapath can be grouped into three classes which build on top of each other:

#### Multiple kernel sizes

This includes adding all hardware resources to support multiple kernel sizes, such that we still support only a single kernel, but have more flexibility. The support for that primarily goes in interface units which become configurable. Register files have to be sized to efficiently support all supported kernel sizes instead of one. The reduction stage also becomes flexible.

#### Multiple flows

This step adds the remaining data access patterns not covered in previous step, such that all algorithm flows based on the same arithmetic operations and reduction type can be implemented. For example for a core supporting only 2D convolutions, this step will add vertical and 1D interfaces with full flexibility and also add any special access patterns not all already supported including offset accesses, interleaved writes and so on.

### Multiple arithmetic operations

This class adds multiple arithmetic and logical operations in the functional units, as well as multiple reduction types (summation versus logical reduction).

For SIFT's filtering stage, the first programmability class entails an increase in energy dissipation of just 25% which is relatively small. The fixed function hardware for SIFT already has a large enough 1D shift register to support a 16-tap 1D horizontal filter so adding support for smaller 4 and 8 tap 1D filters only requires adding a small number of multiplexing options in 1D horizontal IF unit and support for tapping the reduction tree at intermediate levels. However, the second programmability class incurs a bigger penalty because now a 2D shift register is added for vertical and 2D flows. The coefficient and output registers are also upgraded from 1D to 2D structures, and the ALU is now shared between Horizontal, Vertical and 2D operations. The result is a substantial increase in register access energy and ALU access energy. Moreover, the 2D register comes with support for multiple vertical and 2D kernel sizes as well as support for horizontal and vertical offsets and register blocking, so the area gets a big jump shown in Figure 4.20 and consequently the leakage energy increases as well. The final step of adding multiple compute units has a relatively negligible impact of 10%.

For SIFT extrema the cost of adding multiple kernel sizes is again only 1.3x. However, supporting additional access patterns adds another 2x on top of that bringing the total cost to roughly 2.5x over the fixed kernel version. Unlike filtering stage, SIFT extrema starts with 2D structures so the additional cost of adding the 1D horizontal operations is relatively low. However, the 2D and vertical IF units also become more complex to support various horizontal and vertical offsets into the 2D register. The cost of multiplexing to support these is very significant compared to the low energy map and reduce operations used in this algorithm. The result is a big relative jump in energy. The last step of supporting more arithmetic operations again has a relatively small incremental cost of around 1.2x. The final programmable version still takes roughly 12x less energy compared to the SIMD version.

Like SIFT extrema, IME also has a lightweight map step (absolute difference),

however, it has a more substantial reduction step (summation). So the relative cost of muxing needed to support multiple 2D access patterns is in between the high-energy-cost filtering operations and low-energy-cost extrema operations. The cost of supporting multiple kernel sizes and multiple arithmetic operations is still relatively small.

FME differs slightly from other algorithms in that it takes a big hit when going to multiple kernel sizes. The fixed function core supports 1D-Horizontal and 1D-Vertical filtering for a relatively small filter size of 8 taps. The storage structures are sized accordingly and consist of two small 2D input and two even smaller 2D output shift registers. Adding support for multiple kernel sizes requires making each of these registers larger. Thus multiple stencil sizes not only require additional area in the interface units, but the bigger storage structures also make the muxes substantially bigger, increasing the register access cost. This is further exacerbated by the increase in the leakage energy brought about by the bigger storage structures. Thus the first programmability class has the most impact on the energy efficiency of FME. The impact of the second programmability class is relatively modest as it only adds a 2D interface unit — most of the hardware has already been added by the first programmability class. The cost of supporting multiple arithmetic operations is once again small suggesting that this programmability class is the least expensive to add across all algorithms.

Our results show that the biggest impact on energy efficiency takes place when the needed communication paths become more complex. This overhead is more serious when the fundamental computation energy is small. In general the communication path complexity grows with the size of the storage structures, so over provisioning registers as is needed in a programmable unit hurts efficiency. This energy overhead is made worse since such structures not only require more logic in terms of routing and muxing, but also have a direct impact on the leakage energy. On the other hand, more flexible function units have small overheads, which provides flexibility at low cost.

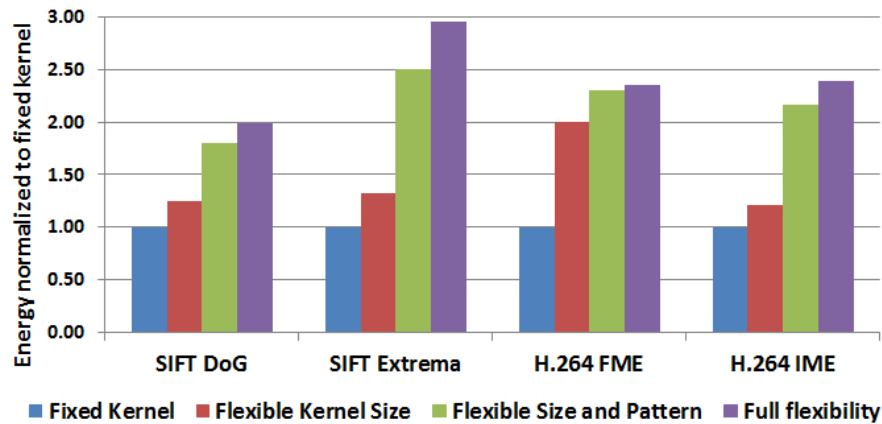


Figure 4.19: Change in energy consumption as programmability is incrementally added to the core [46].

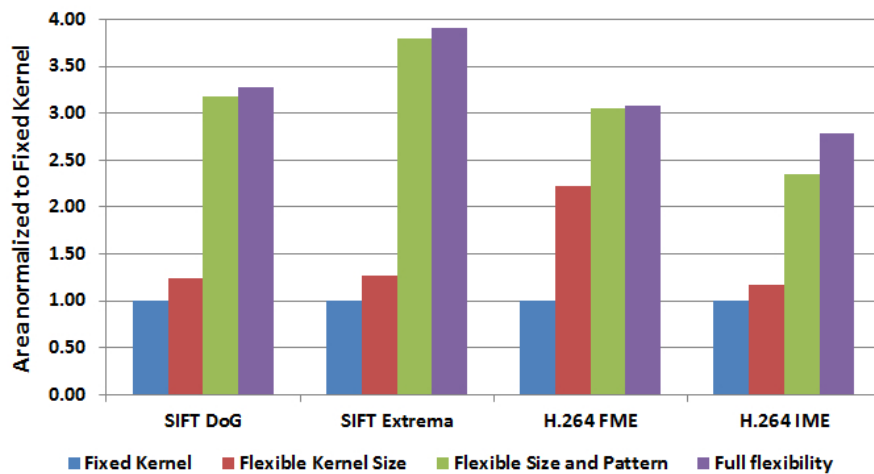


Figure 4.20: Increase in area as programmability is incrementally added to the core [46].

## 4.6 Conclusion

As specialization emerges as the main approach to addressing the energy limitations of current architectures, there is a strong desire to make maximal use of these specialized engines. This in turn argues for making them more flexible, and user accessible. While flexible specialized engines might sound like an oxymoron, we have found that

focusing on the key data-flow and data locality patterns within broad domains allows one to build a highly energy efficient engine, that is still user programmable. We presented the Convolution Engine which supports a number of different algorithms from computational photography, image processing and video processing, all based on convolution-like patterns. A single CE design supports applications with convolutions of various size, dimensions, and type of computation. To achieve energy efficiency, CE captures data reuse patterns, eliminates data transfer overheads, and enables a large number of operations per cycle. CE is within a factor of 2–3x of the energy and area efficiency of single-kernel accelerators and still provides an improvement of 8–15x over general-purpose cores with SIMD extensions for most applications.

# Chapter 5

## Analysis of Memory Bound Applications

Once the compute energy has been optimized away, the energy of loads and stores starts controlling the application energy rendering the application memory bound. Unlike compute bound applications where removing processor overheads is enough, the fundamental energy of a single memory access is expensive relative to the pipeline cost and requires innovative strategies for reducing its impact. Consider Table 5.1, which reveals that the cost of accessing memory increases by an order of magnitude with each level of hierarchy. Since accessing smaller memories that are closer to the processor takes much less energy than larger memories farther away from the core, the efficiency of a memory bound application such as Speech Recognition is predominantly determined by the data's proximity to the processor and reuse within that level of

Memory Hierarchy	Dynamic Energy per Access(pJ)
16KB 2-way L1 Cache	10
1MB 8-way L2 Cache	200
128MB DRAM	2000

Table 5.1: Memory energy consumption at various levels of hierarchy obtained using Cacti 6.5 [53].

hierarchy. The closer the data is to the processor and higher the reuse, the smaller is the memory energy and higher the application efficiency. Because of the high latency associated with accessing larger, farther memories this optimization is already done for performance reasons with the help of caches. Furthermore, the ability of the caches to capture data reuse within close proximity to the processor remains quite high so they also boost energy efficiency. Because of the efficiency benefits linked to performance optimizations, further reduction of memory energy becomes challenging and demands advanced strategies such as algorithmic restructuring and introduction of additional hierarchies primarily targeted at increasing efficiency. However, despite the gains introduced by memory centric techniques, improvement in efficiency remains modest unless a restructuring of the algorithm has a large change in memory locality.

## 5.1 Speech Recognition

Although, some applications start off as being memory bound with memory access energy taking a sizeable portion of the total system energy, many applications only become memory bound once the compute has been optimized away. Automatic speech recognition, which transforms human speech into text [54], is one such application. Speech recognition systems that offer high accuracy pose a significant challenge for general-purpose systems as they not only require significant compute power, but also need a substantial amount of memory bandwidth. One such speech recognition system is Sphinx 3.0, which has been developed at CMU. In contrast to recent speed optimized recognition systems, Sphinx 3.0 offers a high accuracy, but is marred by slow decoding speeds. The biggest obstacles in achieving high performance are high computational intensity and a large memory footprint. To achieve even reasonable decoding speeds using a 5000 word Wall Street Journal corpus [55], Sphinx 3.0 requires a powerful ILP optimized core with a significant amount of on-chip cache.

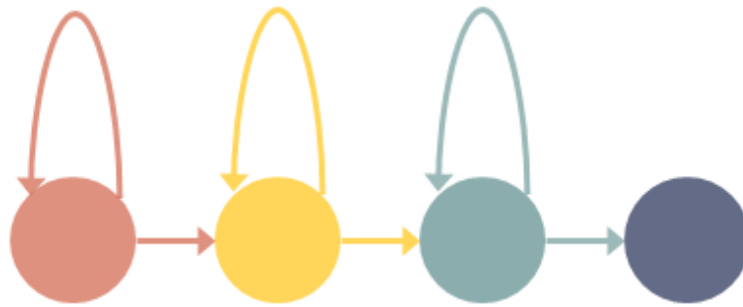


Figure 5.1: The figure presents the four state HMM model used for representing tri-phones in Sphinx 3.0.

### 5.1.1 CMU Sphinx Speech Recognition

Before devising ways to improve the decoding speed, let's look at how Sphinx 3.0 works. As explained in [54][56] the recognizer consists of three main stages: feature extraction, acoustic scoring and backend search. The first two stages or the frontend extract acoustic features from digitized input audio in 10msec frames; while the third stage or the backend uses the features extracted in the frontend to convert the input speech into text. To maintain a high accuracy, the backend operates in sub-word units called phones. Representation of speech at the phonetic level is performed using a four-state Hidden Markov Model (HMM) where the first three states correspond to a contextually dependent phone triplet called tri-phones[57] as shown in Figure 5.1. Aided by the acoustic features extracted in the frontend, the transition and observation probabilities of the HMM states govern within-word and cross-word transitions with the latter leading to recognized words. Let's look at each of these stages in detail.

#### Frontend

As defined earlier, the Frontend consists of two stages: feature extraction and acoustic scoring. Before being passed to the feature extraction stage, the input speech is digitized and organized into 10ms blocks called frames. These frames are converted by feature extraction into frequency domain components to identify and examine



the more distinctive speech features to help differentiate speech utterances. A 39-element feature vector is generated containing all the acoustic information for the frame and is passed to the acoustic scoring stage, which uses Gaussian Mixture Models (GMM) to match the features in each frame against a large collection of sounds or tri-phones to generate HMM state probabilities. Because the number of sounds can be enormous, modeling all sounds would require a huge amount of data; therefore, in Sphinx similar phones are combined together into senones[57] used to represent individual HMM states in place of phones. The backend search stage uses these sound probabilities to find the most probable word sequence by gluing together sounds using their probabilities.

While the number of features identified in the feature extraction stage is large, the execution time is dominated by the conversion of speech frames to the frequency domain. A 512 point Fast Fourier Transform (FFT) is typically employed to transform digitized speech into the frequency domain, which can be easily accelerated to run several times faster than real-time by an energy-efficient data parallel processor with support for bit-reverse addressing[54]. The acoustic scoring stage, however, is much more computationally demanding compared to feature extraction. Although, this computational intensity ensues from the use of GMMs, Sphinx 3.0 keeps the complexity in check by performing GMM computation in the log domain. Using some typical parameters, it is shown in [54] that the performance requirement of the acoustic scoring stage is approximately 500MFLOPs, which can be further reduced by exploring other optimization techniques such as quantization and bit-width reduction. The frontend consists of standard compute limited computation and can be easily handled using techniques described in the previous chapters.

## **Backend**

The main task of the backend search stage is to convert frontend senone state probabilities into word suggestions. To achieve a high accuracy, the search stage maintains a pool of active words whose probabilities are updated every frame using the incoming senone scores. Since each word can consist of multiple phones, the number of corresponding active HMMs can run into thousands and generally require off-chip

storage. The active HMMs are processed and updated by the backend stage with the help of the Viterbi Beam Search and the Language Model that are described below:

**Viterbi Beam Search** The primary task of the Viterbi algorithm is to update the state probabilities of active HMMs by combining the senone scores obtained from the frontend with the transition probabilities of the individual HMM states. This process is performed on a per-frame basis and at the end of each frame only the highest probabilities of the states are maintained highlighting the best path to the states. Although, Viterbi is guaranteed to find the most likely word sequence by continuously updating the state probabilities every frame, the search space becomes large especially for large vocabularies because of the addition of new HMMs to the system. This substantial increase in the space has a detrimental effect on the performance prompting beam pruning to curtail the list of active HMMs at the expense of some accuracy loss. Pruning involves removal of less probable states and their corresponding HMMs allowing only the more probable states to continue to the next frame. New HMMs are added to the system when the active state probabilities cross a certain threshold. The Language Model is invoked if a cross-word transition takes place otherwise within-word transitions are handled by an adjunct stage called the Transition stage. Furthermore, after a word is recognized, it is added to the word lattice which keeps track of detected words to facilitate the backtracking step employed in the end to determine the most likely sequence of detected words.

**Language Model** Sphinx 3.0 employs an n-gram Language Model (LM) where n equals 3, to identify word candidates most likely to follow those recently recognized. To activate new words after a cross-word transition, Sphinx 3.0 determines the relative probabilities of new words given preceding word sequences in the word lattice in the following order: word triples (trigrams) with probability  $P(w_3/(w_1, w_2))$  are considered first where  $w_3$  is the new word and  $w_1$  and  $w_2$  denote the two-word history; this is followed by word pairs (bigrams) with probability  $P(w_3/w_2)$ ; with single word probabilities (unigrams),  $P(w_3)$ , coming in the end. Because the memory requirements exponentially increase with the number of prior words considered, recognizers

usually stop at word triples or trigrams. Boosted by grammar knowledge and occurrence frequency of words and phrases in the training data, these n-gram probabilities substantially improve the decoding accuracy of Sphinx 3.0.

As mentioned earlier, recognition accuracy is achieved by the Backend stage by tracking numerous simultaneous words at the phonetic level. However, this comes at the expense of a substantial increase in the processing requirements. From our experiments we have determined that for a 5000 word Wall Street Journal corpus, the Backend stage accounts for 60% of the execution time of the speech recognition system. The percentage share rises to more than 70% for a 60,000 word vocabulary [58]. Because the backend search stage constitutes the most critical part of the speech recognition system, its irregular computation and high memory traffic presents the biggest obstacle to the improvement in performance and energy efficiency of speech recognition. Thus, this thesis focuses on devising strategies for improving the efficiency and performance of the backend search stage.

## 5.2 Baseline Speech Recognition System

Speech recognition's backend decoding path is quite long and suffers from sequential dependencies that restrict parallelism. Additionally, the decoder relies on a large probabilistic model built into the HMM data-structure that not only requires off-chip storage, but also needs frequent accesses resulting in high memory energy dissipation. To develop an energy efficient speech decoding system we need to reduce the energy wasted in the processor in addition to curtailing accesses to DRAM by boosting locality and data-reuse. Remarkably, the algorithmic changes required to expose task level parallelism by overcoming dependencies also carry the added advantage of limiting redundant memory accesses. While the number of modifications is large, the major changes are described below:

### 5.2.1 Pruning Stage

To maintain a high accuracy the decoder tracks numerous words simultaneously. Since these words are tracked at the phonetic level, the number of HMMs can run into thousands. To keep the number of HMMs from increasing exponentially Viterbi Beam Search stage prunes the HMMs whose probability falls below a certain threshold. This threshold is updated every frame and requires the Viterbi Stage to first update the state probabilities of all the active HMMs. A record is maintained of the highest probability which is then used to ascertain the threshold. The updated HMMs are read again and their probabilities compared with the threshold. Those that fall below the threshold are pruned away. Because of this dependency, the active HMMs are read twice which substantially increases the memory traffic. Using the technique presented in [59], we eliminate these redundant accesses by using the highest probability from the previous frame. This technique also allows us to eliminate sequential dependencies by enabling different HMMs to be processed by different stages in parallel.

### 5.2.2 Patch List

When the Language Model adds new words to the system after a cross-word transition, the HMM list is searched for matches. If an HMM corresponding to the first tri-phone of the new word is active, its state probability is updated if it's lower than that of the new word. Since the HMMs generally require off-chip storage and the number of new word candidates can be substantial, this process can require several DRAM accesses. As presented in [59], we eliminate the off-chip accesses by de-coupling the Language Model from the HMM list using an on-chip storage structure referred to as the Patch List. This allows the new word candidates to be merged with the HMMs in the next frame when they are fetched for processing by the Viterbi Stage.

### 5.2.3 CMP System

Removal of sequential dependencies in the decoding path facilitates partitioning of the backend search into a four stage HMM level pipeline shown in Figure 5.2. The first three stages are formed when the “Viterbi Beam Search” described in Section 5.1.1

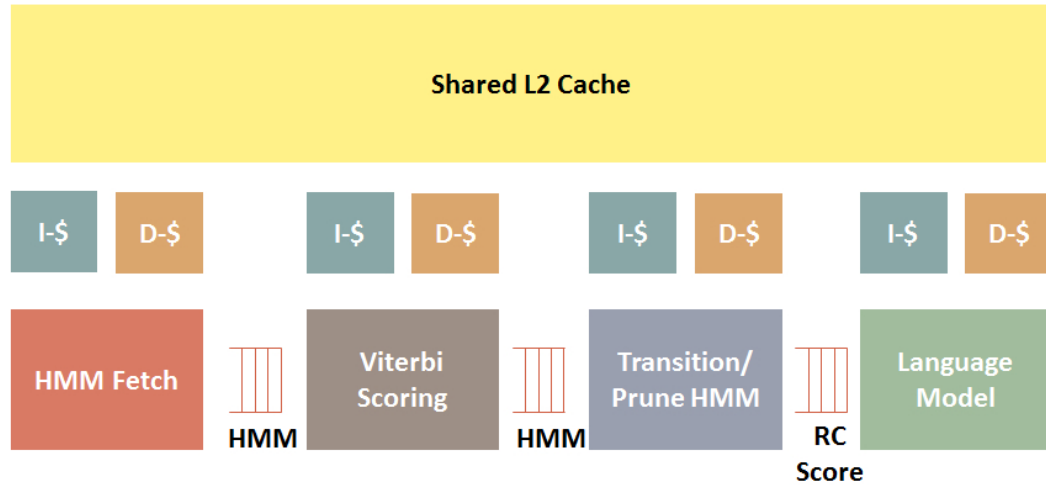


Figure 5.2: Four stage HMM level partition of Sphinx 3.0. Each processor contains private L1 caches connected through a shared L2. Queues are added between the first three processors to facilitate transfer of HMMs fetched by the “Fetch” stage. While the queue between “Transition/Prune” and the “Language Model” is used to store scores for all possible right-contexts (RC Scores) of the completed word instead of HMMs.

is split into three distinct phases called “Fetch”, “Viterbi” and “Transition/Prune”. The “Fetch” stage is responsible for fetching the HMMs from the memory and merging them with the words from the Patch List; the senone scores formulated in the front end are then employed by the “Viterbi” stage to update the state probabilities of the HMMs from the “Fetch” stage; while the “Transition/Prune” stage is tasked with tracking within and cross word transitions, pruning of non-performing HMMs and writing un-pruned HMMs back to the memory. The “Language Model” constitutes the fourth stage and is activated only after a cross-word transition is detected in the “Transition/Prune” stage. This division not only streamlines the data transfers between stages significantly, but also allows us to exploit task-level parallelism at the HMM level.

To build a base system, we map the four stage HMM level pipeline to a four-processor Tensilica based RISC CMP system designed to run at 400MHz in 90nm. Each processor is assigned private 16KB 2-way set associative instruction and data L1 caches connected through a 512KB 8-way set associative unified L2 cache with

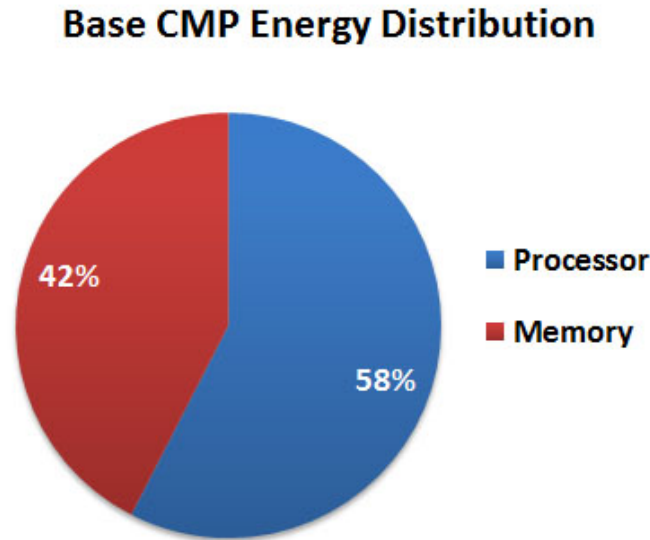


Figure 5.3: Energy distribution between the processing elements and the memory for the base CMP system.

a 32B line size. The memory energy is estimated using CACTI 6.5 [53]. To ensure a speedy transfer of HMMs and other data between stages, first-in-first-out (FIFO) queues are incorporated between processors. Assisted by the availability of significant task level parallelism, the CMP system accomplishes decoding of a 5000 word WSJ vocabulary in real-time. However, a quick look at Figure 5.3 reveals that memory is a major portion of the total system energy indicating that enhancement of the system efficiency also involves improving data locality in addition to eliminating superfluous loads and stores emanating from the processor.

### 5.3 Experimental Methodology

Because the data is too big to fit on-chip, memory energy consumes a substantial portion of the total system energy. As noted in [54] Sphinx 3.0 not only reports a much higher miss rate, 48% local miss rate for a unified 512KB L2 cache with a 128B line size, than other SPEC benchmarks for a 60K Broadcast News (HUB4)

task, but the memory footprint of 64MB also remains quite high. Although, in this thesis we use a 5000 word WSJ vocabulary task which is considerably smaller than 60K HUB4 vocabulary, the miss rate still remains quite high. Unfortunately, the processor instruction stream is also dominated by memory access operations such as loads and stores so a large improvement in processor energy dissipation observed in compute bound applications remains improbable without a considerable improvement in the data reuse inside the processor.

### 5.3.1 Processor Optimization Strategy

As alluded to earlier, processor energy represents a major portion of the total system energy. All four pipeline stages are load/store intensive with “Fetch”, “Transition/Prune” and “Language Model” also being control intensive. Other factors impeding processor efficiency gains are limited data level parallelism and irregular nature of the computation. Although, all these factors collaborate to render general purpose optimizations such as SIMD largely ineffective, speech recognition possesses a considerable degree of task level parallelism allowing multiple cores to operate in parallel; however, its effect on efficiency remains moderate at best. Thus, to improve efficiency as well as performance, we draw inspiration from the lessons learned in Chapter 3 and create highly customized fused instruction sub-graphs called DAGs. These fused instructions collapse the control-flow instructions and pack as much compute as possible in a single instruction permitting us to amortize processor overheads over many operations. As noted earlier, the efficiency improvements are determined by the height of fused DAGs and generally stay within an order of magnitude. After the processor energy efficiency is optimized, memory energy becomes the tall pole.

### 5.3.2 Memory Optimization Strategy

Due to a lack of significant locality, reducing memory energy is hard. The dynamic data is dominated by the large HMM data structure that is streamed in, updated and written back every frame. There is no intra-frame temporal locality in the HMM data structure and the data is too big to fit on-chip. Although, read-only data dominated

Major Consumers	Read Misses (M)	Write Misses (M)	%age Of Total
<b>HMM</b>	7.9	9.0	39.0
<b>Patch List</b>	8.0	5.6	31.6
<b>Language Model</b>	9.0	-	20.7
<b>Rest</b>	3.7	-	8.7
<b>Total</b>	28.6	14.6	100.0

Table 5.2: The table presents the LL cache misses for the major consumers of memory energy.

by the Language Model (LM) possesses moderate temporal locality, it also requires megabytes of storage. With all these factors colluding to increase the miss rate of the last level (LL) cache, memory energy goes up significantly. The cost of accessing the DRAM now represents 79% of the total memory energy. While we can improve performance with things like latency hiding and prefetch, they have no effect on memory energy.

We want to capture some reuse to prevent us from accessing memory that is further away in the memory hierarchy. Although, a relatively big LL cache can capture a considerable degree of data-locality, when it fails improving results further is difficult, especially with limited visibility into the behavior of individual data structures accessing the big cache. We address this issue by isolating the principal consumers of energy in the application, ascertaining trends in their access patterns and establishing the size of memory required by each consumer. While the principal consumers are recognized by studying LL cache misses, the latter two objectives are met by examining data access patterns across 600+ speech samples. Using the data patterns we identify statistically hot data and ascertain the size of memory required by each consumer to capture reuse in the common case while ignoring the tall tail behavior. Equipped with this information we determine the requisite cache size, establish guidelines on memory management of the important data structures and eliminate redundant cache misses.



## Principal Energy Consumers

After a careful analysis of the LL cache misses, we isolate the principal energy consumers, which are presented in Table 5.2. Together these data structures account for over 90% of the total LL cache misses.

- **Active HMMs:** As explained earlier, HMM is a probabilistic model employed for representing tri-phones of the language. In each frame of speech we test the active HMMs to check the probability of their occurrence in the prior and current frames. We always check some number of them across frames as long as their score remains high enough. Because each HMM entry requires 48B of storage and the number of active HMMs per frame runs into tens of thousands ( $> 30,000$ ), HMMs cause frequent cache misses and account for 39.0% of the total LL cache misses.
- **Patch List:** New words initiated by LM after a cross-word transition are stored in the Patch List. Each patch list entry is 16B wide, but with approximately 6500 words in the system Patch List requires in excess of 100KB of storage and accounts for 31.6% of the total LL cache misses.
- **Language Model Word Candidates:** Language model word candidates stand for the n-gram probabilities representing the relative probabilities of new words given preceding word sequences in the word lattice. Although, each entry is just 8B, there are millions of word candidates and require mega bytes of storage with relatively little locality. As a result, LM accounts for 20.7% of the total LL cache misses.

## 5.4 Results

We start looking at processor optimization because the energy consumed in the processing elements accounts for a slightly larger portion of the system energy. These improvements appreciably reduce the processor energy consumption leaving speech recognition memory bound. Since the memory optimization techniques consist of

reducing the energy dissipation of the principal consumers, we outline optimization strategies for each major consumer of energy describing the amount of memory required to store the statistically hot data and how to best manage the data-structure in a cache based system. Collectively, these results describe how system efficiency improves by 5x over the baseline CMP configuration.

### 5.4.1 Processor Optimization Results

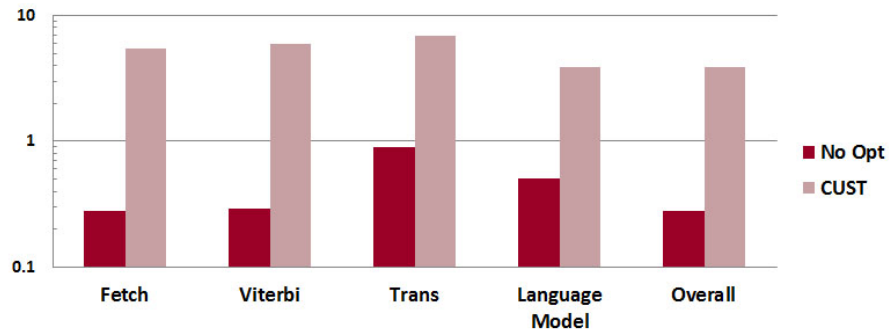


Figure 5.4: Improvement in performance over the base system that was operating at real-time after the application of custom instructions.

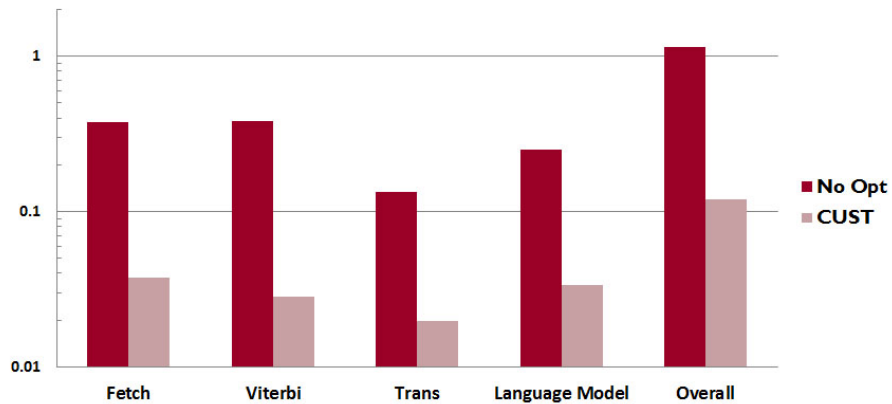


Figure 5.5: Improvement in energy per frame over the base system after the application of custom instructions.

Using Tensilica’s TIE language we create application specific instructions that collapse multiple control-flow graphs into single instructions. In addition to performing

### CMP Energy Distribution After Processor Optimization

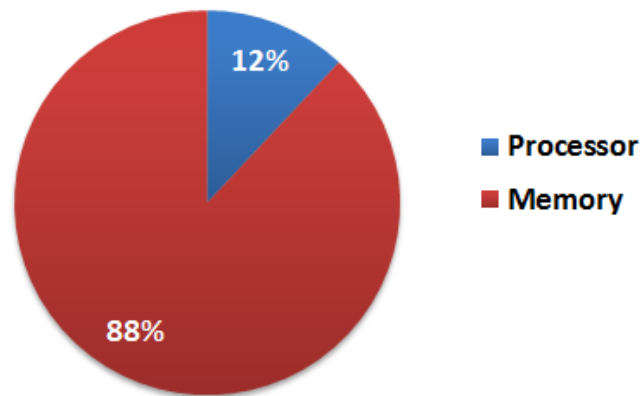


Figure 5.6: Energy distribution between the processing elements and the memory after processor customizations.

multiple operations per instruction, the fused instructions also facilitate the flow of short-lived intermediate data from one operation to another without accessing the register file providing a much needed energy efficiency improvement. However, as mentioned earlier, fusing together more than 2-3 basic blocks becomes increasingly difficult; thus, the efficiency gains remain restricted to an order of magnitude. The introduction of these operations enables the CMP to run at 15 times better than real-time and consume an order of magnitude less energy than the base configuration as shown in Figures 5.4 and 5.5 respectively. However, further gains are difficult to come by because random loads and stores start dominating the instruction stream of “Fetch”, “Viterbi” and “Transition/Prune” stages while the “Language Model”, which is also the bottleneck, gets limited by the energy wasted on branches. After the application of processor optimizations, speech recognition becomes memory bound because memory energy takes 88% of the total energy as shown in Figure 5.6.

## 5.4.2 Memory Optimization Results

As mentioned earlier “Active HMMs”, “Language Model Word Candidates” and “Patch List” account for more than 90% of the total LL cache misses. In this section we present optimization strategies for each data structure.

### Active HMMs

The first optimization involves reducing the memory footprint of active HMM data structure. With the help of non-standard precision, we can reduce the amount of storage required by each HMM entry from 48B to 32B without compromising data integrity. This shrinkage results in memory energy savings of 16% and cuts the number of HMM LL misses in half.

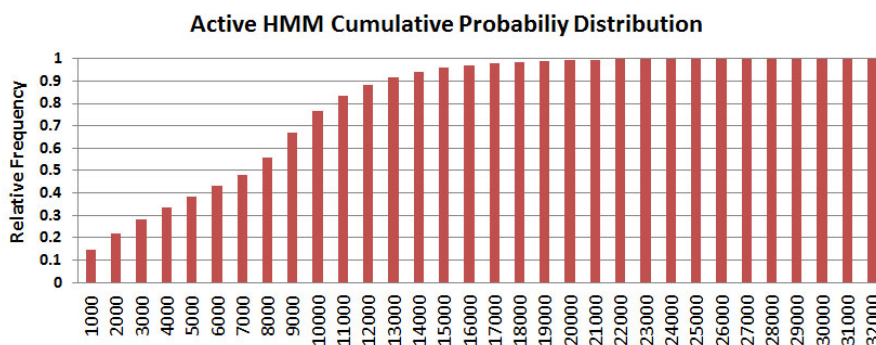


Figure 5.7: The plot captures the total size of the HMM data structure in the current frame on the x-axis and the relative frequency a structure of this size is likely to occur in a given frame on the y-axis.

Now, that we have reduced the size of the HMM data-structure, we turn our attention to enhancing the reuse of the HMM phoneme data across frames. With the help of 600+ speech samples, we determine the relative frequency of a certain size of the HMM data structure occurring in a given frame and plot it against the total size of the HMM data structure in the current frame as shown in Figure 5.7. The plot exposes the power-law curve in the probability distribution and reveals that the probability of the number of active HMMs staying below 13,000 is over 90%. Thus, with each HMM reduced to 32B, to capture reuse of the data in the common case

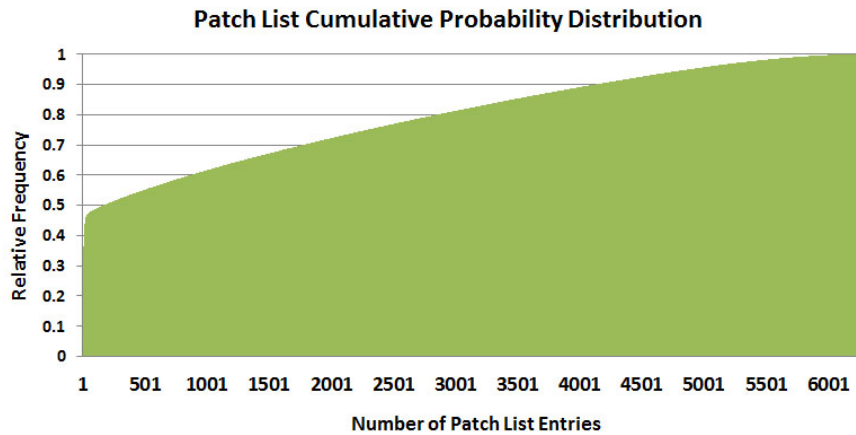


Figure 5.8: The plot captures the relative frequency of a given amount of memory needed to store Patch List entries in the current frame.

we need just 400KB of storage, which is significantly less than the 1MB of storage required in the worst case. However, to ensure that only memory locations within the recommended storage space are used, software management of the HMM storage space is imperative. Since the HMMs are constantly deleted and created, it's critical that the most recently deleted entry is reallocated. Using standard "malloc" was one cause of the high miss rates in the original application, a fact hard to find until we knew what the answer was supposed to be. In addition to the simple garbage collector, non-allocating stores are also needed to eliminate write misses emanating from output only data. Furthermore, you will notice that given the size of the recommended storage is 400KB, an LL cache size in excess of our base system's 512KB is required to leave room for other data structures.

### Patch List

Using the same technique that we employed in the last section, we determine that certain Patch List entries are updated much more often than others. Consider Figure 5.8, which presents the relative frequency of a given amount of memory needed by the Patch List entries in the current frame. The amount of memory is directly correlated with the number of valid Patch List entries in the current frame. From

the figure we ascertain that using just an 8KB memory we can capture more than 50% of the accesses going to the Patch List, which is much smaller than the 100KB required in the worst case. However, to substantially reduce memory energy we need to capture as much of the accessed data as possible and it turns out that placing all of newly activated/updated words on-chip gives us the greatest advantage in the trade-off of efficiency vs memory storage; thus, when we determine the new size of the L2 cache we consider the worst case storage requirement for Patch List entries.

### Language Model Word Candidates

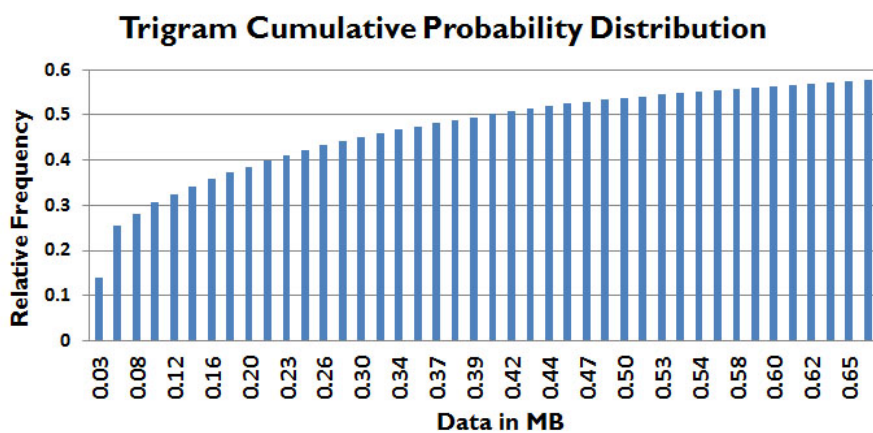


Figure 5.9: The plot captures the relative frequency of a given amount of memory needed to store Trigram word candidates in the current frame.

Unfortunately, Language Model word candidates require mega bytes of storage and incur frequent misses. However, a careful analysis of the access frequencies of the  $n$ -gram (Bigram and Trigram) probabilities reveals some interesting patterns. Firstly, we notice that accesses to the biggest LM data structure, Trigram Word Candidates, is uniformly distributed; however, two Trigrams are accessed much more frequently than others and account for 25% of the total Trigram traffic as shown in in Figure 5.9. Fortunately, Bigram word candidates as shown in Figure 5.10 exhibit substantially less randomness and up to 40% of the total Bigram traffic can be captured using just a couple of hundred kilobytes of storage, which is remarkably less than the few

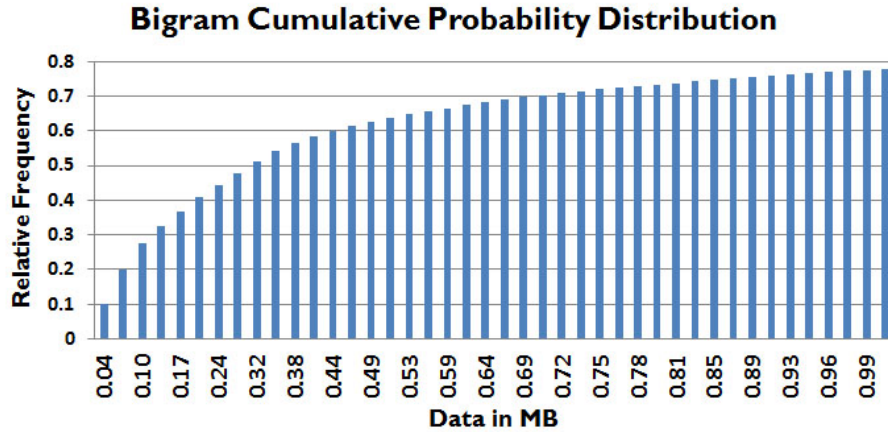


Figure 5.10: The plot captures the relative frequency of a given amount of memory needed to store Bigram word candidates in the current frame.

Major Consumers	Read Misses (M)	Write Misses (M)	%age Of Total
HMM	0.5	-	6.2
Patch List	-	-	0.0
Language Model	4.7	-	56.9
Rest	3.1	-	36.9
<b>Total</b>	<b>8.3</b>	<b>-</b>	<b>100.0</b>

Table 5.3: The table presents the LL cache misses for the major consumers of memory energy after memory optimizations.

megabytes required to store the whole Bigram data structure. However, in our quest to decrease the LL cache miss rate using usage statistics, we go one step further and rearrange Bigram and Trigram word candidates in memory placing frequently accessed word candidates in close proximity to each to improve spatial locality even more.

As we alluded to earlier, the amount of storage needed to capture statistically hot data is in excess of 768KB easily surpassing the 512KB LL cache used in our base CMP system. Taking into consideration requirements of other data structures in Sphinx 3.0, in our optimized system we use a 1MB LL cache. Furthermore, we augment our cache to perform non-allocating stores. The combination of bigger LL

### Custom CMP Energy Distribution

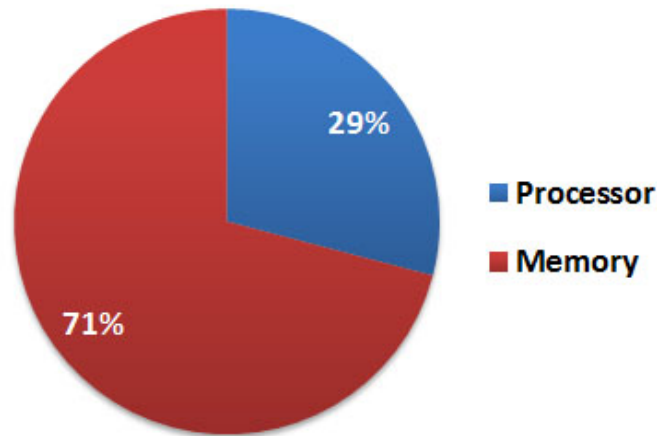


Figure 5.11: Energy distribution between the processing elements and the memory for the optimized CMP system.

cache size, software management of data structures to bring out locality and the addition of non-allocating stores, improves the memory energy by 3x. The LL miss breakdown is presented in Table 5.3 and the system energy breakdown is shown in Figure 5.11. From the figure we can determine that while the memory energy has decreased appreciably, the system is still memory bound.

## 5.5 Conclusion

Aided by processor customization and improved data reuse inside the cache, we managed to reduce the system energy by 5x. Despite insignificant spatial and temporal locality, the improvement in memory energy was made possible through the exploitation of application characteristics. Helped by the analysis of data access patterns of the principal consumers of memory energy, we were able to isolate and exploit statistically hot data that proved vital in improving memory energy. Thus finding locality somehow, in order to bring the data as close to the processing elements as



possible, is critical for improving memory energy. In our case locality was found by using probabilities in large models. However, the thing to note is that despite using extensive processor customization and memory enhancements, the gains were limited to an order of magnitude over the base CMP using a large LL cache. This indicates that caches generally work well for capturing locality and reuse in an application and significantly improving memory energy is difficult.

Because the memory energy now dominates the total energy, the system efficiency is limited by the fundamental cost of accessing the memory. Even an ideal ASIC that can eliminate all of the processor cost, would still only be a factor 1.3x more as we are fundamentally limited by the memory. Building an ASIC for this application makes little sense.

# Chapter 6

## Conclusion

When systems are power limited, improved performance requires decreasing the energy of each operation, but technology scaling is no longer providing the energy reduction required. Thus, providing this energy reduction requires tailoring systems to specific applications, and such customization is extremely expensive because of high non-recurring engineering (NRE) costs. To address this issue in this thesis we study the potential of creating efficient yet flexible general-purpose chips by analyzing the inefficiencies introduced by programmability. We gather this data across three classes of applications defined by their leading energy consumers: compute, control and memory. For each application class we determine the source of energy overhead and devise strategies to reduce them. We take applications that are employed in systems with stringent energy budgets like cellphones, video cameras, etc., but require custom hardware solutions to meet the energy and performance requirements effectively, since these provide examples of efficient hardware solutions we can analyze.

While studying the sources of inefficiency in compute bound applications, we learned that the scope of the problem depended upon the type of computation being performed. For many applications where ASICs have large gains like H.264 motion estimation, the critical operations are extremely low-power. For any compute bound application to achieve high efficiency, the critical operations must limit energy; however, the energy wasted in processor overheads such as instruction fetch and register file is far higher than that of the critical operations. To amortize these overheads,

we need to execute hundreds of basic operations per cycle which requires close coupling between storage and the functional units. Furthermore, interactions with the memory need to be curtailed otherwise compute intensive applications risk becoming memory bound.

Of course if the critical operation requires moderate energy such as floating point (FP), the overhead of using a processor is smaller, around 10x. This means that machines with ten wide FP units are not far from the maximum efficiency possible for that class of applications. Thus, CPUs with wide SIMD units or GPUs can be efficient programmable solutions for these applications. Unfortunately, this also means that energy gains are limited. Said differently, if we want ASIC-like energy efficiencies — 100x to 1000x more energy efficient than general-purpose CPUs — we will have to transform our algorithms to be dominated by the simple, low-energy operations we have been studying in this thesis.

Although, conditions for achieving high efficiency in compute bound applications seem overly restrictive at a first glance, there exists a large number of compute intensive applications which satisfy these requirements. These applications have a convolution like data-flow and we presented a flexible, efficient engine, called the Convolution Engine, which supports this flow. To achieve energy efficiency, CE captures data reuse patterns, eliminates data transfer overheads, and enables a large number of operations per cycle. CE is within a factor of 2–3x of the energy and area efficiency of single-kernel accelerators and still provides an improvement of 8–15x over general-purpose cores with SIMD extensions for most applications.

Unlike data parallel applications, efficiency gains for control intensive applications such as CABAC remain relatively low even after algorithmic restructuring. We observe that for CABAC even highly custom control-flow graph instructions are only able to improve the performance and efficiency by just an order of magnitude. The gains in control bound applications are limited by inter-dependent branches that restrict the number of fuseable basic blocks to 2–3. The pivotal role played by highly custom instructions in achieving the efficiency gains, makes us believe that for control intensive applications to achieve high efficiency custom instructions will need to be crafted for each important loop in the application. Looking forward, a generalized

control-flow graph fusion network that offers predicated execution and the ability to fuse together multiple simple control-flow operations in one instruction can offer efficiency gains superior to general-purpose processors while still retaining flexibility.

Of course after these data and control optimizations are done, the application is likely to become memory energy limited. While the answer to reducing energy is obvious, exploit locality, the easy locality has already been extracted by the caches in a modern processor. In these cases, it often requires a detailed look at each of the important data structures to understand the locality issues, and possible algorithmic restructuring to improve locality.

# Bibliography

- [1] R. Dennard, F. Gaensslen, H. Yu, V. Rideout, E. Bassous, and A. LeBlanc, “Design of Ion-Implanted MOSFET’s with Very Small Physical Dimensions,” *Proceedings of the IEEE (reprinted from IEEE Journal Of Solid-State Circuits, 1974)*, vol. 87, no. 4, pp. 668–678, 1999.
- [2] O. Shacham, O. Azizi, M. Wachs, W. Qadeer, Z. Asgar, K. Kelley, J. Stevenson, A. Solomatnikov, A. Firoozshahian, B. Lee, S. Richardson, and M. Horowitz, “Rethinking Digital Design: Why Design Must Change,” *IEEE Micro*, vol. 30, pp. 9–24, Nov. 2010.
- [3] Standard Performance Evaluation Corp., “SPEC CPU2006 Results.” <http://www.spec.org/cpu2006/results>, 2006.
- [4] P. Hanrahan, “Keynote: Why are Graphics Systems so Fast?,” *18th International Conference on Parallel Architectures and Compilation Techniques, 2009. PACT '09*, Sep 2009.
- [5] T.-C. Chen, S.-Y. Chien, Y.-W. Huang, C.-H. Tsai, C.-Y. Chen, T.-W. Chen, and L.-G. Chen, “Analysis and Architecture Design of an HDTV720p 30 Frames/sec H.264/AVC Encoder,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 16, no. 6, pp. 673–688, 2006.
- [6] R. E. Collett, “Executive Session: How to Address Today’s Growing System Complexity,” *DATE '10: Conference on Design, Automation and Test in Europe*, March 2010.

- [7] D. Grose, “Keynote: From Contract to Collaboration Delivering a New Approach to Foundry,” *DAC '10: Design Automation Conference*, June 2010.
- [8] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., fourth ed., 2006.
- [9] J. Balfour, W. Dally, D. Black-Schaffer, V. Parikh, and J. Park, “An Energy-Efficient Processor Architecture for Embedded Systems,” *Computer Architecture Letters*, vol. 7, no. 1, pp. 29–32, 2007.
- [10] J. D. Balfour, *Efficient Embedded Computing*. PhD thesis, Stanford University, 2010.
- [11] M. Garland and D. B. Kirk, “Understanding Throughput-Oriented Architectures,” *Communications of the ACM*, vol. 53, pp. 58–66, Nov 2010.
- [12] R. Espasa, M. Valero, and J. E. Smith, “Vector Architectures: Past, Present and Future,” *International Conference on Supercomputing*, p. 425432, 1998.
- [13] Intel Corporation, “Motion Estimation with Intel Streaming SIMD Extensions 4 (Intel SSE4),” 2008.
- [14] Intel Corporation, “Intel SSE4 Programming Reference.”  
<http://softwarecommunity.intel.com/isn/Downloads/Intel%20SSE4%20Programming%20Reference.pdf>.
- [15] M. Rofouei, T. Stathopoulos, S. Ryffel, W. Kaiser, and M. Sarrafzadeh, “Energy-Aware High Performance Computing with Graphic Processing Units,” *Workshop on Power Aware Computing and System*, Dec 2008.
- [16] C. Rowen and S. Leibson, “Flexible Architectures for Engineering Successful SOCs,” *Design Automation Conference, 2004. Proceedings. 41st*, pp. 692–697, 2004.

- [17] Tensilica Inc., “The What, Why, and How of Configurable Processors.” <http://www.tensilica.com/products/literature-docs/white-papers/configurable-processors.htm>.
- [18] Tensilica Inc., “Xtensa LX2 Benchmarks.” <http://www.tensilica.com/products/xtensa-customizable/xtensa-lx2/benchmarks.htm>.
- [19] P. Ienne and R. Leupers, *Customizable Embedded Processors: Design Technologies and Applications*. Morgan Kaufmann Publishers Inc., aug 2006.
- [20] R. Gonzalez, “Xtensa: A Configurable and Extensible Processor,” *Micro, IEEE*, vol. 20, pp. 60–70, Mar. 2000.
- [21] J. Cong, Y. Fan, G. Han, and Z. Zhang, “Application-Specific Instruction Generation for Configurable Processor Architectures,” *12th International Symposium on Field Programmable Gate Arrays*, pp. 183–189, 2004.
- [22] Tensilica Inc., “Implementing the Advanced Encryption Standard on Xtensa Processors.” <http://www.tensilica.com/products/literature-docs/application-notes/tie-application-notes/advanced-encryption-standard.htm>.
- [23] Tensilica Inc., “Implementing the Fast Fourier Transform (FFT).” <http://www.tensilica.com/products/literature-docs/application-notes/tie-application-notes/fast-fourier-transform-fft.htm>.
- [24] Tensilica Inc., “Xtensa Processor Extensions for Data Encryption Standard (DES).” <http://www.tensilica.com/products/literature-docs/application-notes/tie-application-notes/data-encryption-extensions.htm>.
- [25] Tensilica Inc., “Xtensa Energy Estimator (Xenergy) - Users Guide.”
- [26] D. G. Lowe, “Distinctive Image Features from Scale-Invariant Keypoints,” *International Journal of Computer Vision*, vol. 60, pp. 91–110, Nov 2004.

- [27] J. F. Hamilton and J. E. Adams, "Adaptive Color Plane Interpolation in Single Sensor Color Electronic Camera." US Patent US 5,629,734, 1997.
- [28] T. Wiegand, G. J. Sullivan, G. Bjntegaard, and A. Luthra, "Overview of the H.264/AVC Video Coding Standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, pp. 560–576, Jul 2003.
- [29] T. Wiegand and G. Sullivan, "Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification," *ITU-T Recommendation H.264 and ISO/IEC 14496-10 AVC*, May 2003.
- [30] ITU-T. Joint Video Team Reference Software JM8.6.
- [31] R. Li, B. Zeng, and M. L. Liou, "A New Three-Step Search Algorithm for Block Motion Estimation," *IEEE Transactions on Circuits and System for Video Technology*, vol. 4, pp. 438–442, Aug 1994.
- [32] J. R. Jain and A. K. Jain, "Displacement Measurement and Its Application in Interframe Image Coding," *IEEE Transactions on Communications*, vol. 29, pp. 1799–1808, Dec 1981.
- [33] S. Zhu and K.-K. Ma, "A New Diamond Search Algorithm for Fast Block Matching Motion Estimation," *IEEE Transactions on Image Processing*, vol. 9, pp. 287–290, Feb 2000.
- [34] G. J. Sullivan, P. Topiwala, and A. Luthra, "The H.264/AVC Advanced Video Coding Standard: Overview and Introduction to the Fidelity Range Extensions," *SPIE Conference on Applications of Digital Image Processing XXVII*, Nov 2004.
- [35] "IP Core for an H.264 Decoder SOC." <http://www.design-reuse.com/articles/15746/ip-core-for-an-h-264-decoder-soc.html>.
- [36] "Vcodex White Paper: An Overview of H.264 Advanced Video Coding." <http://www.vcodex.com/images/uploaded/469323879727520.pdf>.



- [37] D. Marpe, H. Schwarz, and T. Wiegand, "Context-Based Adaptive Binary Arithmetic Coding in the H.264/AVC Video Compression Standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, pp. 620–636, Jul 2003.
- [38] "Context-Based Adaptive Binary Arithmetic Coding (CABAC)." <http://www.hhi.fraunhofer.de/fields-of-competence/image-processing/research-groups/image-video-coding/statistical-modeling-coding/context-based-adaptive-binary-arithmetic-coding-cabac.html>.
- [39] D. Marpe, "Context-Based Adaptive Binary Arithmetic Coding (CABAC)." <http://iphome.hhi.de/marpe/cabac.html>.
- [40] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding Sources of Inefficiency in General-Purpose Chips," in *ISCA '10: Proc. 37th Annual International Symposium on Computer Architecture*, ACM, 2010.
- [41] H. Shojania and S. Sudharsanan, "A VLSI Architecture for High Performance CABAC Encoding," in *Visual Communications and Image Processing*, 2005.
- [42] [http://en.wikipedia.org/wiki/Blob\\_detection](http://en.wikipedia.org/wiki/Blob_detection).
- [43] [http://www.scholarpedia.org/article/Scale\\_Invariant\\_Feature\\_Transform](http://www.scholarpedia.org/article/Scale_Invariant_Feature_Transform).
- [44] F. Estrada, A. Jepson, and D. Fleet, "Local Features Tutorial." <http://www.cs.toronto.edu/~jepson/csc2503/tutSIFT04.pdf>, Nov 2004.
- [45] R. Ramanath, W. Snyder, G. Bilbro, and W. Sander, "Demosaicing methods for Bayer color arrays," *Journal of Electron Imaging*, vol. 11, pp. 306–315, Jul 2002.
- [46] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution Engine: Balancing Efficiency and Flexibility in

- Specialized Computing,” *Proceedings of the 40th Annual International Symposium on Computer Architecture*, vol. 41, pp. 24–35, Jun 2013.
- [47] Tensilica Inc., “ConnX Vectra LX DSP Engine.” [www.tensilica.com/uploads/119/Vectra-pdf](http://www.tensilica.com/uploads/119/Vectra-pdf).
- [48] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, vLi Wen Chang, N. Anssari, G. D. Liu, and W. mei W. Hwu, “IMPACT Technical Report,” in *IMPACT-12-01*, 2012.
- [49] NVIDIA Corporation, “GeForce GTX480 Specifications.” <http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-480/specifications>.
- [50] A. Bakhoda, G. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, “Analyzing CUDA Workloads Using a Detailed GPU Simulator,” in *ISPASS: IEEE International Symposium on Performance Analysis of Systems and Software*, 2009.
- [51] J. Leng, S. Gilani, T. Hetherington, A. E. Tantawy, N. S. Kim, T. M. Aamodt, and V. J. Reddi, “GPUWattch: Enabling Energy Optimizations in GPGPUs,” in *ISCA 2013: International Symposium on Computer Architecture*, 2013.
- [52] Tensilica Inc., “Tensilica Instruction Extension (TIE) Language Reference Manual.”
- [53] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, “Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0,” *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 3–14, 2007.
- [54] E. C. Lin, *A High Performance Custom Hardware Backend Search Engine for a Speech Recognition System*. PhD thesis, Carnegie Mellon University, 2007.
- [55] D. B. Paul and J. M. Baker, “The Design for the Wall Street Journal-Based CSR Corpus,” *Proceedings of the Workshop on Speech and Natural Language*, pp. 357–362, 1992.

- [56] P. J. Bourke, *A Low-Power Hardware Architecture for Speech Recognition Search*. PhD thesis, Carnegie Mellon University, 2011.
- [57] H. Mei, *Subphonetic Acoustic Modeling for Speaker-Independent Continuous Speech Recognition*. PhD thesis, Carnegie Mellon University, Dec 1993.
- [58] E. C. Lin and R. A. Rutenbar, “A Multi-FPGA 10x-Real-Time High-Speed Search Engine for a 5000-Word Vocabulary Speech Recognizer,” in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 83–92, Feb 2009.
- [59] E. C. Lin, K. Yu, R. A. Rutenbar, and T. Chen, “A 1000-Word Vocabulary, Speaker-Independent, Continuous Live-Mode Speech Recognizer Implemented in a Single FPGA,” in *Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays*, pp. 60–68, 2007.

ProQuest Number: 28122755

INFORMATION TO ALL USERS

The quality and completeness of this reproduction is dependent on the quality and completeness of the copy made available to ProQuest.



Distributed by ProQuest LLC (2020).

Copyright of the Dissertation is held by the Author unless otherwise noted.

This work may be used in accordance with the terms of the Creative Commons license or other rights statement, as indicated in the copyright statement or in the metadata associated with this work. Unless otherwise specified in the copyright statement or the metadata, all rights are reserved by the copyright holder.

This work is protected against unauthorized copying under Title 17, United States Code and other applicable copyright laws.

Microform Edition where available © ProQuest LLC. No reproduction or digitization of the Microform Edition is authorized without permission of ProQuest LLC.

ProQuest LLC  
789 East Eisenhower Parkway  
P.O. Box 1346  
Ann Arbor, MI 48106 - 1346 USA