BALANCING EFFICIENCY AND FLEXIBILITY IN SPECIALIZED 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

Rehan Hameed
December 2013

This dissertation is online at: http://purl.stanford.edu/qx015pt3899

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, 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.

Mark Horowitz

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

CMOS-based integrated circuits have hit a power wall, and future performance increases cannot rely on increased power budgets. This means we need to create more energy efficient solutions if we want performance to continue to scale.

A proven way to gain high efficiency is to build special-purpose ASIC chips for the application of interest. These designs can achieve 2-3 orders of magnitude higher energy efficiency and performance compared to general-purpose processors. However ASIC design has become prohibitively expensive making it difficult to justify the investments in design effort for all but the few applications with very large volumes and stable code bases. General-purpose processors amortize the design cost over a large number of applications, and provide standard development tools, resulting in higher productivity and lower development costs. However, this flexibility comes at a cost of much higher energy consumption.

This thesis examines the tradeoff between flexibility and efficiency with an aim to develop architectures that combine the low energy consumption of customized units with the reusability of general-purpose processors. A number of approaches are already being tried to lower the energy consumption of programmable systems, such as a move to homogenous and heterogeneous multi-core systems, augmenting the processors with hardware accelerators, and creating application-specific processors. However our work takes a step back to first understand and quantify what makes a general-purpose processor so inefficient and whether it is at all possible to get close to ASIC efficiencies within a programmable framework. The insights from this work are then used as a basis to derive new architectural ideas for efficient execution.

Specifically, we propose building domain customized functional units as a solution

for balancing efficiency with flexibility. As a case study, we look at the domain of imaging and video processing. These workloads are becoming ubiquitous across all computing devices and have very high computing requirements often served by special purpose hardware. At the same time there are a large number of emerging applications in this domain with diverse requirements, so going forward there is a great need for flexible platforms for this domain. Thus it is an ideal candidate for our study. We demonstrate two programmable functional units for this domain - the Convolution Engine and the Bilateral Engine. A number of key computational motifs common to most applications in this domain can be implemented very efficiently using these engines. The resulting performance and efficiency is within 2-3x of custom designs but an order of magnitude better than general-purpose processors with data-parallel extensions such as SIMD units.

We also argue that domain customized functional units demand a slight change in the mindset of system designers and application developers – instead of always wanting to fit the hardware to algorithm requirements, we optimize a number of key computational motifs and then restructure our applications to make maximum use of these motifs. As an example, we look at modifying the *bilateral filtering* algorithm - a key non-linear filter common to most computational photography applications - such that it is a good fit for the capabilities of our proposed hardware units. The resulting implementation provides over 50x energy reduction over the state of the art software implementation for this algorithm.

Our work suggests that identifying key data-flows and computational motifs in a domain and creating efficient-yet-flexible domain customized functional units to optimize these motifs is a viable solution to address the energy consumption problem faced by designers today.

Acknowledgements

Looking back at my years at Stanford, it has been a long journey that started when I arrived here for my masters—full of ambition but having no idea of the path ahead. Along the way I have been supported and helped by a huge number of people around me who have made possible the completion of this work.

First and foremost, I would like to thank my advisor Professor Christos Kozyrakis. I came to Stanford about the same time when Christos joined Stanford as a faculty member. A few quarters of stalking him finally convinced him to take me as his student, and since then he has been providing me guidance. Christos is a great advisor and very interactive in his discussions. When you go to him with an idea, he quickly zeros in on what you are trying to do, and within minutes you might be deep into a conversation, discussing every possible aspect of the problem to explore. His mentorship and support has been invaluable for me.

I am equally thankful to Professor Mark Horowitz. While Mark was not my primary advisor, I have worked with him during most of my PhD, starting when I joined the Smart Memories project up until today. Mark likes to dive deep into a problem. He makes you challenge your own assumptions, and makes you really think about what it is that you should be doing. I have been fortunate to have his advise over the years, which has benefited all the work presented in this thesis.

I would also like to thank Stephen Richardson, not just for serving on my reading committee but also for always being a helpful figure, willing to listen to ideas, give feedback, and help in any way he could.

During my stay at Stanford I got a chance to work with many amazing people in my research group. I would specially like to mention Wajahat Qadeer, who is a

colleague as well as a friend. I have collaborated extensively with Wajahat and much of this thesis draws as much on his work, as my own. I would also like to thank Megan, whom I also collaborated with during this work; Ofer and John, who are always there as sounding boards to throw ideas at and get feedback; Alex and Amin for guiding me in my early days with the group; and several other members of the group I have had the pleasure to work with, including Omid, Sameh, Kyle, Andrew and Artem.

Being away from home in a foreign country is always a challenge, and it becomes even more daunting when most of your family is at the opposite end of the globe, a 30-hour flight away. Luckily I have been blessed with a large network of friends inside and outside Stanford who have made my stay in US a great experience. These include Wajahat, Salman, Shahzad, Inam, Hammad, Farhan, Irfan, Aamir, Khurram, Tahir, Usman, Sikandar, Shumail, Zeeshan, Waqas and many others. Thanks to all of you for being there for me and my family!

I would also like to thank Teresa Lynn and Sue George who have helped me on countless occasions in dealing with all sorts of administrative issues.

Throughout my life, my father, Abdul Hameed Awan, had been a driving force for me and the same is true for my PhD work. He devoted much of his life to his children, and whatever I have achieved so far I owe it to him. My mother passed away early in my life, but her love has stayed with me and gives me strength. I would also like to thank my brothers Nauman and Salman for their continuous support and being always there for me despite the distance.

I must also thank my wife's family, including her Father Tajammal Hussain, her mother Rehana Kausar, as well as her sisters and brothers who have all given me much love and encouragement along the way.

My daughters Zenia and Zunaira are a joy, enriching our lives every day. I thank them for being so understanding despite always wondering why dad still goes to school. Finally, I can't say enough about my wife and my companion Mehjabeen. If the PhD is a journey then Mehjabeen has been with me every step of the way offering support, and I thank her and love her for that.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

A major challenge faced by computer architects today is to create new architectures that achieve higher performance at fixed power. Most computing systems today are power limited. In case of servers and desktops, power is constrained by our inability to cool these devices effectively once power crosses a threshold. Mobile and embedded devices not only have even lower cooling limits, but also aim to maximize battery life by minimizing power consumption. While these have very different power envelops—a few hundred watts for servers vs. a few watts for mobile platforms—both operate under strict power limits. Since power is operations/sec x energy/operation, we need to decrease the energy cost of each operation if we want to continue to scale performance at constant power.

With such an emphasis on low energy designs it might be a shock to know that current general-purpose processors are extremely inefficient when it comes to energy consumption. Studies such as [29] and [20] have shown that a general-purpose processor often consumes up to 1000 times higher energy compared to dedicated hardware solutions for compute intensive applications! In past, this steep cost of programmability was hidden by the energy reduction achieved through voltage scaling. However with voltage-scaling coming to a stop, and at the same time power budgets hitting a ceiling, we can no longer continue to waste this much energy. This suggests we

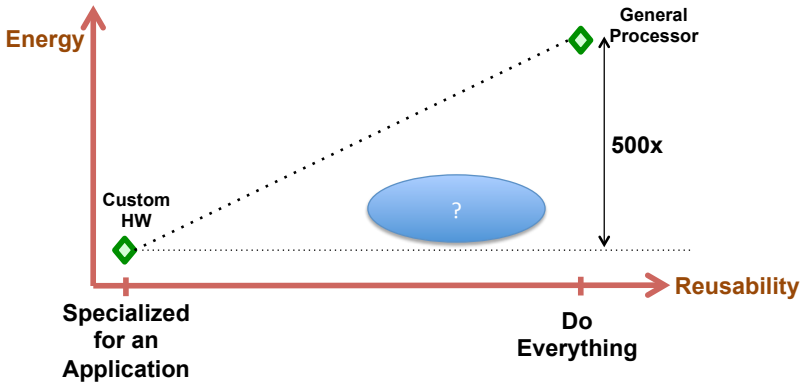


Figure 1.1: Design space for programmability vs. energy-efficiency tradeoff. We would like a new design point lying somewhere in the blue region, with energy consumption close to custom hardware, and having high flexibility.

should move away from general-purpose processors and create custom ASICs, which make much more efficient use of energy and silicon area. Indeed, this has been the approach for many high-volume applications. However with today's billion transistor chips, designing and manufacturing an ASIC has become extremely expensive, and very few applications have a big enough market to justify the cost.

Processors have a big advantage that you design a processor once and then re-use it again and again over a larger number of applications, effectively amortizing the design cost as well as the manufacturing cost. Moreover processors provide a well-established development environment, as well as a much greater ability to react to changes in application requirements.

This thesis explores the large design space between ASICs and GP processors. As depicted in Figure 1.1, we want to look for new tradeoff points in this space that have energy consumption close to custom hardware and, yet, do not give up too much in flexibility compared to a fully programmable processor. Ideally such design points would fit somewhere in the blue region of Figure 1.1.

1.2 Thesis Overview

Chapter 2 discusses in detail the energy constraints faced by designs today as well as the challenges faced by future hardware systems designed under these constraints. Specially, we concentrate on image processing applications, which are rapidly becoming an integral part of most computing systems and at the same time have very high computation requirements.

We approach the goal of creating flexible, yet efficient designs by first understanding what are the sources of energy overheads in a general-purpose processor, and quantifying the contribution of each of these to the overall inefficiency of the processors. Chapter 3 explores this topic by presenting energy breakdown of a typical RISC instruction and also relates that to the energy cost of cache and DRAM memory accesses. For most of the instructions, cost of an ALU operation is less than 1% of instruction energy - much of the energy cost goes into (i) fetching and decoding the instruction stream to implement a programmable control flow, and (ii) moving data from a memory / cache to the ALU inputs. Conceptually, the primary mechanisms for reducing these overheads include (i) amortizing the instruction sequencing cost by executing tens to hundreds of ALU operations per instruction, and (ii) minimizing cache / memory accesses by capturing most of the data-reuse in low-energy, high bandwidth storage structures local to the processor datapath. To gain high efficiency, then, we would like to incorporate these characteristics into a general-purpose computation framework.

Of course not every application class can achieve 3 orders of magnitude energy reduction. Highest gains are achieved for applications that have a high degree of parallelism as well as significant data locality. While these conditions might seem restrictive, these span most of the applications for which efficient custom solutions exist.

One architectural idea that already employs similar principles is a SIMD unit, which is now common in most modern processors including embedded processors[56, 48]. By restricting the domain of computation to 1-D data-parallel operations, a

SIMD unit loses some flexibility but gains performance and energy efficiency. However, are these types of broad specialization strategies good enough to overcome the energy overheads and get close to the efficiency of an algorithm specific custom hardware? And, if not, then how much further specialization is needed to bring the processor close to dedicated hardware designs?

Chapter 4 attempts to answer these question with a case study on HD H.264 video encoding - a highly compute intensive application often implemented as a custom ASIC. We transform a generic multi-processor RISC system into to a highly specialized multiprocessor optimized for this application. Initially we incorporate only broad specializations and optimizations, including SIMD, VLIW, and fused operations. While the resulting processor system achieves around 10x energy reduction for data-parallel components of the application, it still leaves a lot on the table consuming about 50x more energy overall compared to the ASIC. Further specialization brings the customized processors within 2-3x of the ASIC solution. However, the specialized datapaths that we add to achieve that, look fairly similar to their ASIC counter parts. Unlike the ASIC, the programmable control flow breaks the ties from a specific algorithm flow. However, in practice the highly algorithm-tuned nature of the datapath implies that these customized processors are not very useful beyond the algorithm they are designed for.

Chapter 5 shows that truly re-usable specialized cores can be created by specializing for recurring data-flow patterns instead of specializing for a specific application. The key is identifying patterns that are not only re-used across a range of applications, but also meet both the parallelism and data-reuse requirements that we have outlined. One such common motif in our target imaging domain is a convolution-like data-flow: apply a function to a stencil of the data, then perform a reduction, then shift the stencil to include a small amount of new data, and repeat. Chapter 5 looks at building a flexible Convolution Engine for this motif. We show that the resulting unit is able to accelerate a diverse set of applications that are typically implemented today using a combination of multiple custom hardware units and programmable application processors and GPUs. Impressively it can do that with about two orders of magnitude less energy compared to a general processor, and in fact the energy

consumption is within 2-3x of custom hardware. It is even an order of magnitude better than a SIMD unit, which is already specialized for data parallel algorithms.

Of course not everything in the imaging domain maps to this convolution abstraction. One such example is bilateral filtering [58], which is a non-linear filtering technique widely used in computational photography algorithms [8]. Bilateral filtering has a large working set and thus does not immediately offer the short-term data re-use exploited in the convolution abstraction. However, as Chapter 6 shows the algorithm still has a large degree of locality. By restructuring the algorithm to expose that locality, we bring it to a form that we know how to optimize using similar techniques as used for the convolution based algorithms. The work on optimizing bilateral filtering also reinforces our belief that a domain customized approach requires a slight change in the mindset about how we approach hardware design. Instead of tailoring the hardware to the requirements of each individual algorithm, we would like to identify widely applicable computational motifs that we can implement efficiently in hardware, and then restructure the algorithms if needed to make use of these highly optimized computational motifs.

The convolution engine together with the bilateral engine provides the core capabilities required to accelerate a large number of imaging applications. This includes a multitude of emerging computational photography applications, which aim to enhance the imaging experience and quality through use of advanced image processing techniques, but which are currently limited by the power or performance limitation of current imaging platforms.

Chapter 2

Energy Constraints And The Need For Flexibility

For years digital integrated circuit chips have been on a growth path that is nothing short of extraordinary. In just 30 years, we have gone from simple chips with only a few thousand transistors to today's extremely complex designs with billions of transistors - a million-fold increase. This growth has been made possible by semiconductor technology scaling - with every new generation we could not only fit exponentially higher number of transistors in a given chip area, but could also switch these transistors at exponentially lower energy to keep power in check. Designers leveraged this ever-increasing number of lower-energy transistors, and at the same time used higher and higher power, delivering bigger and faster designs with explosive growth in performance.

However, power consumption of these chips has hit the maximum limit in recent years, and at the same time energy reduction through technology scaling has slowed down. Thus, the traditional approach to getting more performance - using a larger number of transistors and running them at faster speed - no longer works as that would exceed the power limit. Energy consumption has thus become a primary design constraint today, and we need innovations in computer architecture creating designs that could do more work with the same number of transistor switches. In other words, these new designs have to make more efficient use of the transistor resources.

Another consequence of this exponential growth is a huge increase in design cost of the chips - a billion transistor chip is much more complex than a thousand transistor chip. In fact, the cost of designing a new dedicated hardware is now prohibitive for most applications [49, 28]. This is leading chip designers to use more and more programmable and reusable components such as processors. This, however, goes against the need to create energy efficient designs - as [29] and [20] show, a general-purpose processor can consume a hundred to a thousand times more energy compared to a dedicated hardware design. This puts us in a dilemma - given the energy constraints, we would like to move away from these inefficient processors and create efficient custom designs. However, the design complexity and reuse considerations push towards using programmable processor systems.

In this chapter, we discuss the implications of these constraints for future system design. Section 2.1 explains in detail how technology has evolved over the years and why scaling is no longer the answer. Section 2.2 then explains some approaches currently in use to solve this issue, including a move to multi-core systems as well as heterogeneous systems, and limitations of these approaches. Specifically in 2.3 we look more closely at heterogeneous systems currently in use by most embedded imaging systems and explain why these designs are not well-equipped to meet the future needs of such systems within the given power constrains.

2.1 Technology Scaling and Energy Constraints

Technology scaling over the past few decades has been fueled by our ability to make the transistors smaller and smaller through advances in photolithography process, and pack a larger and larger number of these transistors on a single chip. Apart from increasing the transistor count, these smaller transistors also require less energy to switch them. Shacham et al. have presented a detailed account [52] of how semiconductor technology has scaled over the years, as well as the corresponding trends in processor performance and power over that period. In this section, we summarize some of these trends to help understand the current landscape for hardware system design.

As Shacham et al. point out, understanding semiconductor technology scaling trends is best done by considering Moore's Law [40] and Dennard scaling [22]. Moore's Law, presented by Gordon Moore in 1965, predicted that the number of transistors on a chip will grow exponentially, roughly doubling every two years. Semiconductor chips have closely followed that trend, resulting, as stated previously, in a million-fold increase in transistor count over the last 30 years.

Dennard's work [22] on the other hand established scaling principles that allow not only increasing transistor densities, but at the same time gaining higher transistor switching speeds at constant power dissipation. Under Dennard scaling, when we shrink transistor dimensions by k, the operating voltage is scaled by the same factor k to maintain a constant electric field. Table 2.1 reproduced from [22] lists the impact this has on various aspects of device performance. As the table suggests, following this scaling scheme, not only does the number of transistor in a given area increase by k^2, the delay decreases by k, which in turn means the design could run at k-times higher frequency. At the same time, despite the increased switching frequency, the switching power per transistor goes down by $1/k^2$. If we then account for the fact that we have now k^2 time more transistors in the same area, the total power to switch all the transistors in that area remains constant. The implication is that we now have k^2 more transistors switching at k-times higher frequency, without requiring any increase in silicon area or power. Thus technology scaling alone could enable increasingly higher performance chips without increasing the power budget, as long as designers could find ways to convert these extra transistors into higher performance.

However, as Shacham et al. further point out, in practice the semiconductor chips have deviated from the scaling path prescribed by Dennard, targeting even higher performance than would be achievable through Dennard scaling. Designers have pushed for higher frequencies through deeper pipelines, faster circuit configurations, etc. and have also increased chip areas to get even more transistors. The result is that instead of power density as well as power remaining constant, both of these have been increasing rapidly. One illustration of that can be seen in Figures 2.1 and 2.2 reproduced from [52], which present historical power and power density data for microprocessor designs. While this did not present a problem initially, gradually the

Table 2.1: Scaling results for circuit performance. Reproduced from [22].

Device or Circuit Parameter	Scaling Factor
Device dimension t_{ox}, L, W	$1/k$
Doping Concentration N_a	k
Voltage V	$1/k$
Current I	$1/k$
Capacitance $\epsilon A/t$	$1/k$
Delay time/circuit VC/I	$1/k$
Power dissipation/circuit VI	$1/k^2$
Power density VI/A	1

chips started becoming too hot to cool economically. At the same time, more and more of the computing started moving to mobile platforms, which are constrained by limited battery life and even lower power envelopes. These factors explain why power in Figure 2.1 tapered off soon after year 2000.

While hitting this power wall placed limitations on the techniques designers could employ, Dennard scaling was still available at hand to continue to scale performance within a fixed power budget. However in recent years even Dennard scaling has stopped. With leakage current becoming significant and accounting for a major component of chip power, it is no longer possible to scale down the threshold voltage, V_{th}. Consequently, we can no longer decrease the operating voltage, V_{dd}, without adverse impact on performance. Thus voltage scaling has pretty much stopped starting with the 90nm technology node. In the absence of voltage scaling, if we switch, as before, k^2 times higher number of transistors at k-times higher frequency, there will be a sharp increase in power, which is not feasible. In fact, even without increasing the frequency, switching k^2 times higher number of transistors at constant frequency would still increase the power. Thus we need to perform a larger number of computations per second to get higher performance, but can't proportionally increase the total number of transistors switched per second and consequently the energy dissipated per second. That in turn implies that each individual operation has to be performed using a lower number of transistor switches, using lower energy. This requires a move to fundamentally more efficient ways to perform computations.

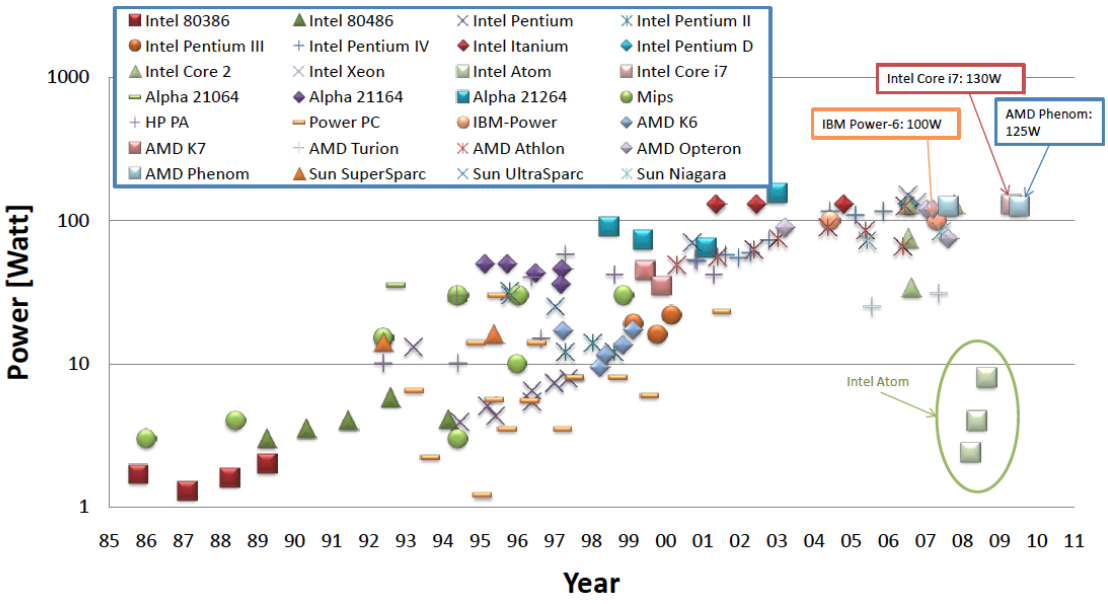


Figure 2.1: Microprocessor power consumption over the years [52].

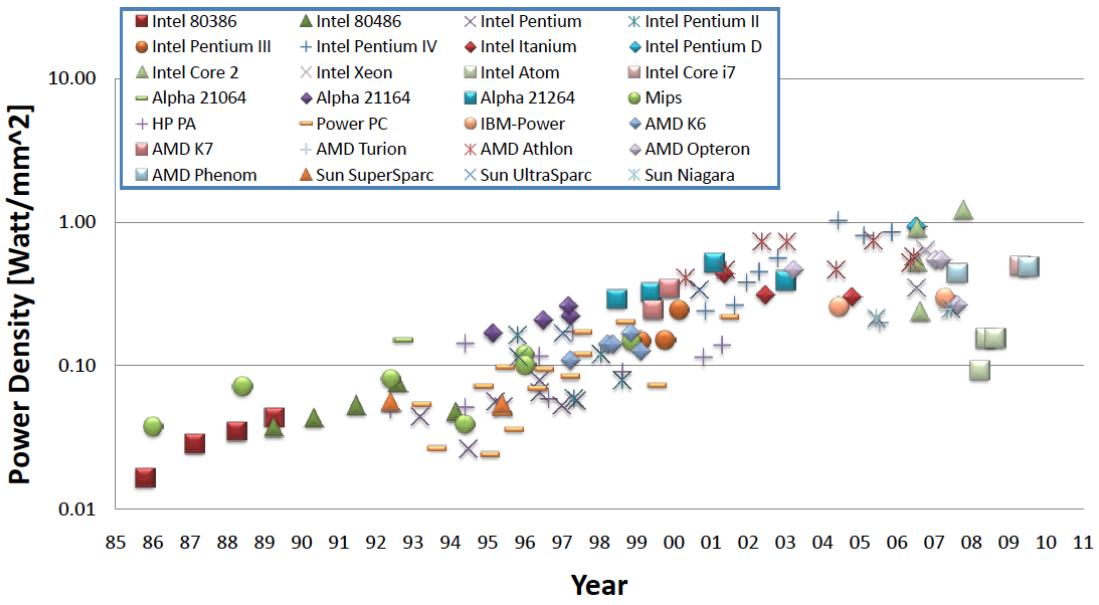


Figure 2.2: Microprocessor power densities over the years [52].

2.2 Energy Efficiency and Processor Systems

As the last section elaborated, increasing performance of a system without an increase in power now requires performing each computation at a lower energy. One way to achieve that goal is to move away from programmable processors towards custom hardware design. Given their programmable nature, general-purpose processors have higher overheads and as a result incur a large degree of wasted work to accomplish a given task. As seen with the H.264 ASIC example in the previous chapter, a hardware block specialized for a single algorithm can perform that task using much less resources and consequently much lower energy. Thus an obvious way to handle the energy constraints is to build custom hardware instead of programmable.

However, the NRE cost of designing and verifying an ASIC is increasing exponentially due to increased complexity and there are very few applications that have a broad enough market to justify this cost of designing dedicated hardware [49, 28]. A processor on the other hand can be designed once and then used for a large number of applications thus amortizing the cost over many systems.

Moreover, the software cost of designing drivers and firmware for each new chip is also increasing exponentially and in fact becoming the dominant cost now. Processors, on the other hand, have a well-established software infrastructure and familiar development tools thus substantially reducing the software cost. Another advantage of a processor-based design is the ability to adapt to changes in algorithms and requirements. With a custom hardware any changes late in the design cycle are costly both in terms of re-design effort as well as time to market. Processors can accommodate such changes at any point during or after the release. A huge testament to the advantages of programmable platforms can be found in the overwhelming popularity and rapid adoption of smartphones. What were once largely fixed function devices with a limited use case, now offer a very diverse set of functionalities as evidenced by millions of third-party applications in place for iOS and android-based smartphones.

There are thus compelling reasons to use processors as basic building blocks for digital systems, and many applications that were once based on completely custom ASICs are now making more and more use of flexible processors. This desire to

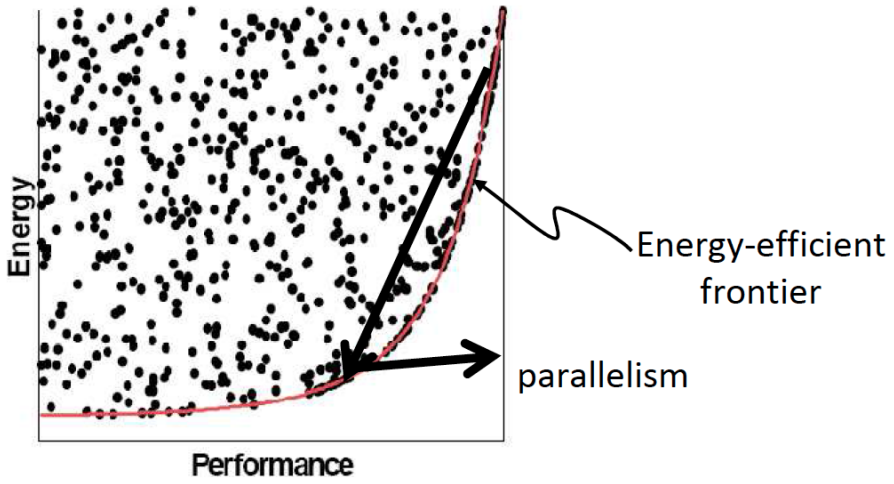


Figure 2.3: The energy-performance space. The Pareto-optimal frontier line represents efficient designs—no higher performance design exists for the given energy budget. The recent push for parallelism advocates more, but simpler, cores. This backs off the higher performance high-power points and uses parallelism to keep/increase performance [52].

leverage flexibility of processor-based designs, however, competes against the need for specialization to improve energy efficiency. Moreover, in order to extract higher and higher performance out of these processors, designers have employed more and more aggressive techniques such as out of order execution, branch target buffers, larger caches, deep pipelining, speculative execution and so on. These aggressive higher-performing processors not only take huge area, but as Azizi et al. [10] show, they are also even more inefficient in terms of energy consumption, thus further exacerbating the situation.

There is thus a growing interest in building more energy efficient programmable systems. This has also driven a move towards parallel multi-core systems based on simpler processor cores.

2.2.1 Multi-Core Systems

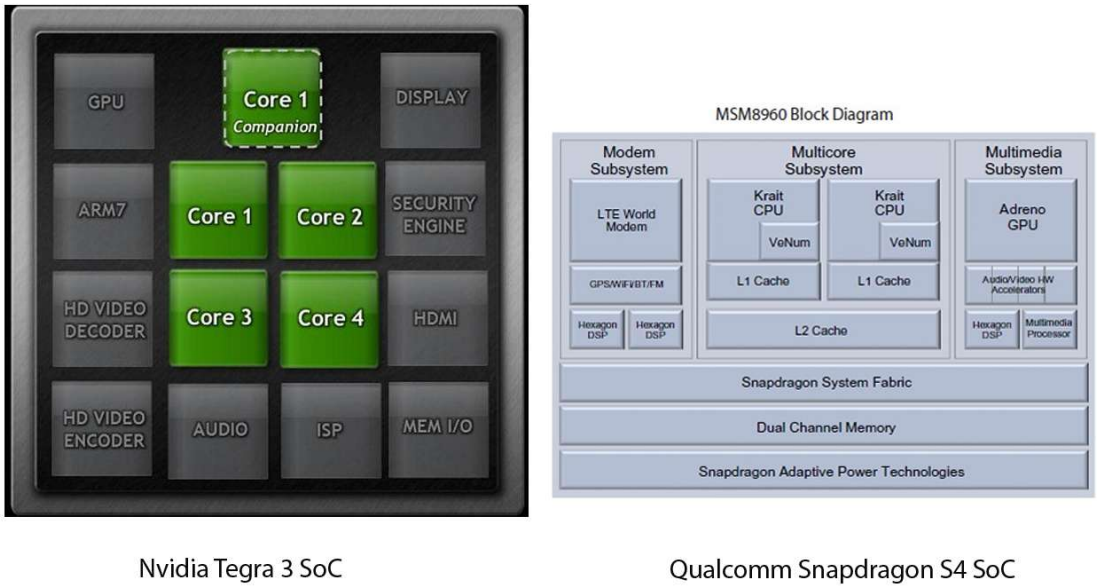
As the last section alluded, pushing processor performance to higher levels requires aggressive energy-hungry techniques. To illustrate that better, Figure 2.3 reproduced

from [52] shows the tradeoff space for energy vs. performance in processor design. The red line marks the efficient frontier consisting of designs that achieve the lowest energy consumption for a given performance level. As the figure shows, performance can be scaled to modest levels without a huge increase in energy consumption. However once we push the performance beyond a certain point, large incremental increases in energy consumption are incurred for relatively small gains in performance. Traditionally the processors were being built with a goal to maximize single core performance and the designs were pushed to limit extracting every last bit of performance using aggressive techniques. Resulting designs, while fast, made highly inefficient use of energy.

The emergence of the power wall, however, has forced designers to rethink this strategy. Instead of pushing the performance of a single core to the steepest region of the energy vs. performance curve, designers have moved towards systems with multiple less aggressive cores. As indicated in Figure 2.3, these cores target a level of performance that can be achieved without going into the most energy-inefficient parts of the trade-off curve. Multiple such cores then work together to achieve the performance goals. Since each of the cores makes more efficient use of energy the higher performance is now achieved at a lower energy cost. There are of course limitations to this approach, as not every application has the parallelism needed to make use of multiple cores.

However, more significantly, even for highly parallel computations that can make good use of multiple cores, this approach provides limited gain. The simpler cores used in such multi-core systems are indeed much more efficient than the extreme designs of the past, they are still highly inefficient. However, as the next chapter will show, even a simple 32-bit RISC processor design, without aggressive hardware units to boost performance, spends over 99% of its energy in wasted overheads and can consume 2-3 orders of magnitude more energy compared to a dedicated ASIC!

This has further led designers to heterogeneous multi-core systems, which we discuss next.



Nvidia Tegra 3 SoC

Qualcomm Snapdragon S4 SoC

Figure 2.4: Two widely used mobile SOC platforms.

2.2.2 Heterogeneous Multi-Core Systems

Unlike homogenous multi-core systems, which employ identical cores replicated many times, heterogeneous systems bring together different processor cores each specialized for a specific task. Figures 2.4 shows a couple of examples of such heterogeneous SOCs for mobile systems. The SOCs in these examples include not only standard RISC processors, but also specialized cores including GPUs to handle 2D and 3D graphics display, DSPs for signal processing tasks, imaging processors to handle camera subsystem and video processors for video encode and decode operations.

This approach helps with energy and performance in two ways. First, a specialized processor can get rid of some resources that are required in a general-purpose processor but not needed for their intended use. For example, GPU cores don't make use of expensive out-of-order execution, speculative execution and other similar mechanisms that are likely to be found in a high performance state-of-the-art CPU. This reduces energy by eliminating the energy wasted on these resources and also frees up area to implement more of these simpler cores, thus enabling higher performance. Secondly,

these specialized cores could include extra resources that help with their target applications and yet cannot be justified in general processor. For example DSPs feature dedicated addressing modes for FFT calculation as FFT is a prevalent operation in signal processing.

While designing and verifying such heterogeneous platforms is more complex compared to a homogenous platform, the low power and energy budgets of mobile devices often necessitates this approach. Nevertheless while the current specialized processors in these heterogeneous platforms improve upon the efficiency of generic CPU cores, they still leave a lot on the table. As we discuss next in the section on imaging and video systems, the most computationally intensive computations in these platforms are still offloaded to fixed function custom hardware blocks, as the GPU and DSP cores are unable to match the extremely low energy consumption of these dedicated blocks.

2.3 Imaging and Video Systems

Imaging and video systems are already deeply integrated into many of our devices. In addition to traditional cameras; cell phones, laptops and tablets now all capture high-resolution images and video. Most smartphone cameras today are equipped with 8-13 megapixels sensors and go even as high as 41 megapixels. On the video side high definition 1080p video at 30fps is now the norm. These imaging systems push a large number of pixels through an image pipeline in real time and thus have very high computational requirements. For each image or video frame a number of processing steps are required to convert raw data from the sensor into a viewable RGB image. Figure 2.5 shows a simplified imaging pipeline with a subset of the typical processing steps (which does not include any of the advanced computational photography algorithms). Furthermore, compressing individual video frames into a video bit-stream requires complex video codecs such as H.264. Performing these computations consumes a lot of energy, and that is a concern given the small batteries and small power budgets of these devices.

Looking back at Figure 2.4 of two SOCs commonly used in mobile devices such

Figure 2.5: Simplified image capture pipeline, which converts RAW data from sensor into RGB images or video frames.

as cell phones and tablets, we see that these systems use custom algorithm-specific hardware to provide the required computation power for imaging tasks at low energy. The Image Signal Processor (ISP) is responsible for implementing an imaging pipeline like the one depicted in Figure 2.5, and the video codec units are responsible for encoding video frames while recording a video clip and decoding the video frames at playback time. This is despite the fact that these SOCs already contain a variety of cores, which can be used to implement imaging algorithms. Most imaging algorithms have a large degree of data-parallelism - a domain that GPUs excel at. Similarly the RISC CPU cores in these devices are also equipped with SIMD extensions [56, 48], which again target data-parallel algorithms. However these cores are unable to provide the performance and energy efficiency required to enable the high compute requirements of imaging and video pipelines necessitating the use of custom dedicated hardware for these operations. What is even more telling is that we get not one but up to three different hardware units for imaging - one dealing with the basic camera pipeline, and others for video encode and decode. Each of the units is customized for a specific task with limited flexibility. While this approach worked well until now when camera sub-systems had a relatively well-defined functionality, new computational photography and computer vision based applications are becoming common, which pose a problem for these platforms.

The field of computational photography aims to use algorithmic techniques to address the limitations of camera sensors and optics. Consumer cameras are typically limited by their small sensors. Cell phones in particular are constrained by the strict requirements on their form factor, and not only have fairly small sensors but also have modest optics. As a result, despite continuous improvements, their image quality lags far behind even cheap entry-level DSLR cameras. However at the same time cell phones are becoming more and more the primary imaging tool for a

large number of consumers due to their convenience. Computational photography techniques use advanced image processing to overcome the small sensor limitations and get enhanced image quality [7, 47]. These include high dynamic range imaging [7, 21], synthetic aperture [36], flash-no-flash imaging [45], digital image stabilization [39], super resolution [24] and video stabilization [37] to name a few. Apart from computational photography, there is also a growing interest in implementing new functionalities such as augmented reality [59], which makes use of imaging as well as computer vision techniques.

However, unlike the well-defined set of functions required in the basic image pipeline and video codec operations, these algorithms require a new and diverse range of computations and the custom imaging units in today's imaging SOCs are not equipped to handle these. The alternative is to make use of general-purpose CPU cores and/or the GPU, which together can provide the needed flexibility as well as computational power for these algorithms. However as we have already seen, this flexibility comes at the cost of 100-1000 times more energy, which is highly undesirable given the battery life constraints. Imaging systems of the future thus face the same competing goals that we have described earlier, requiring, on one hand, high flexibility and programmability to handle a diverse set of computational photography and computer vision algorithms, and at the same time requiring very high energy efficiency.

Chapter 3

Overheads In General Purpose Processors

In this chapter, we look closely at the energy consumption in a processor, and quantify the energy spent on useful computations versus overheads introduced due to the programmable nature of the processors. Once the overheads are quantified we discuss what architectural constructs could help eliminate or reduce these overheads, and what types of application are likely to benefit from these constructs. Finally we present our approach to finding new architectural solutions that incorporate such constructs in a programmable system.

To set the stage for this discussion, let's examine the energy consumption numbers shown in Table 3.1 for software [32] vs. ASIC [16] implementations of high definition H.264 video encoding. The software implementation employs an Intel processor executing highly optimized SSE code. It is also worth noting that the software implementation relies on various algorithmic simplifications, which drastically reduce the computational complexity, but result in a 20% decrease in compression efficiency for a given SNR [32]. Despite these optimizations the software implementation lags far behind the ASIC version, which consumes over 500x less energy using a fraction of the silicon area and has a negligible drop in compression efficiency. This explains why the mobile SOC platforms depicted in Figure 2.4 contain a custom hardware unit for video encode / decode functions. It also suggests that the processor is making an

Table 3.1: Intel's optimized H.264 encoder vs. a 720p HD ASIC. The second row scales Intel's SD data to HD. Original 180nm ASIC data was scaled to 90nm.

	FPS	**Area** (mm^2)	**Energy/Frame(mJ)**
CPU (720x480 SD)	30	122	742
CPU (1280x720 HD)	11	122	2023
ASIC (1280x720 HD)	30	8	4

extremely inefficient use of energy. The next section looks at the energy breakdown of a typical 32-bit RISC instruction to understand where all the energy is spent in the processor.

3.1 Anatomy of a Typical RISC Instruction

Figure 3.1 shows the breakdown of energy consumed in the execution of a 32-bit ADD instruction on a 32-bit RISC processor. The Tensilica processor [25] used for this analysis is a simple single-issue core targeted towards embedded systems. In 45nm technology, execution of this ADD instruction consumes about 70pJ of energy. However, strikingly, only 0.5pJ out of that 70pJ goes into the actual 32-bit ADD logic. The rest of the energy goes into various overheads. One of the biggest overheads is the instruction supply, which involves reading from instruction-cache the instruction to execute and then decode that instruction in the processor to perform the required operation. For a reasonably sized 32-KByte instruction cache, a single cache access consumes about 25pJ(!), which is much larger than the 0.5pJ ADD logic. Similarly, reading the input operands from the register file and writing the output back consumes another 4pJ. And then there is quite a lot more energy that goes into various control operations in the processor, such as instruction decode, program sequencing, pipeline management etc. Thus 99% of instruction energy is the overhead of the programmable machinery built around the basic arithmetic unit.

However that's not all. As Figure 3.2 illustrates, there are further overheads beyond what we have shown. Before the add instruction executes, its operands need to be loaded from memory into the register file, which requires load instructions. Similarly the output of the add operation has to be stored back from the register file to

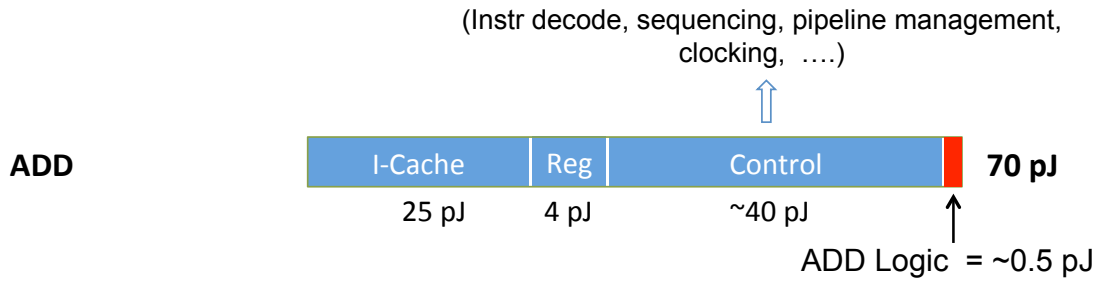


Figure 3.1: Energy breakdown of a typical 32-bit RISC instruction in 45nm technology. Assumes a 32-KByte I-Cache.

memory, requiring a store operation. Each of these load and store operation is in itself a RISC instruction, each incurring an energy cost similar to the ADD instruction. In fact loads and store also need to access the data-cache, which is another 25pJ per access, assuming a 32-KByte data cache. Other overhead instructions include branches to implement loops, further arithmetic operations for address calculations and so on. Once we put that all together, it is clear that the execution of a 0.5pJ add operation could involve hundreds of pJ's of energy spent in overhead instructions as well as overhead logic within each instruction. That also clearly explains the 500x energy difference we have observed in ASIC vs. processor-based implementations.

To understand how to get rid of this waste, we categorize the waste into two main categories. The first category is the overheads that arise from the programmable nature of the processor. These include the energy spent on instruction fetch and decode, sequencing support, pipeline management and other related control logic. These are unique to a processor core and do not exist in custom hardware, which is based on hard-wired control flow. However since our goal is to retain programmable control flow to gain maximum flexibility, one of our key challenges is to find out how to gain efficiency while retaining this instruction-based control flow.

The second category is the overhead of accessing data from memory outside the core, such as the d-cache in the example above. The cost of such an access is far greater than simple arithmetic operations, and that would severely limit efficiency if every arithmetic operation had to get its operands from an outside memory. This challenge is not unique to processors and in fact ASICs need to solve the same problem. That

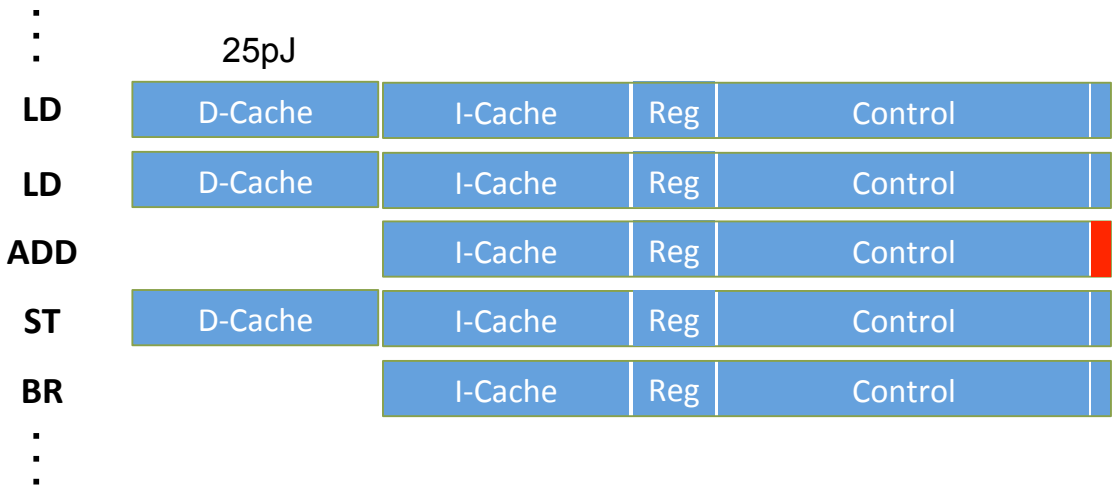


Figure 3.2: Further overheads on top of a RISC instruction. Executing the ADD instruction requires further support instructions such as loads and stores, branches, address calculations and so on. The loads and stores involve costly D-Cache accesses.

Figure 3.3: Amortize the cost of instruction fetch and control overheads over a large number of simple operations

then suggests that we could look learn from how typical ASICs solve this problem and try to incorporate that into our processor-based computational framework.

The next section discusses at an abstract level the mechanisms that can be used to address both these types of overheads and derives conditions for efficient execution within a programmable core. That is the build-up for the next chapter, which then performs a case study for a real application, discussing how to create a highly efficient programmable multi-processor system for this application that achieves performance and energy consumption close to ASIC hardware.

Table 3.2: Energy cost of accessing a 32-bit word from various levels of memory hierarchy and register file (45nm technology node). Third column shows the ratio of these energy costs to the cost of a 32-bit ADD operation.

	Energy/Access (pJ)	Ratio to 32-bit ADD
DRAM	2000	4000
2MB L2-Cache	65	130
32KB L1 D-Cache	25	50
16-Word Register File	4	8

3.2 Removing Processor Overheads

As Figure 3.3 shows, one way to address the overheads of instruction fetch and decode and other related control mechanisms is to create instructions that perform a large number of basic arithmetic operations per instruction fetch. Instead of paying the instruction delivery cost for each 0.5pJ ADD operation, this way the cost is amortized over a much larger number of operations. Note also that for simple arithmetic operations like ADD, we need hundreds of them per instruction to be able to fully amortize the overhead cost. While such an instruction could clearly amortize the overheads, it would need to access hundreds of data elements to operate. That leads to the second constraint i.e. the high cost of accessing data memory.

Table 3.2 shows the energy cost of accessing a 32-bit word from various levels of memory hierarchy as well as the processor register file. It also compares these costs to the cost of a basic 32-bit arithmetic operation. A DRAM access consumes energy that is thousands of times the cost of a 32-bit ADD operation. Clearly if every operation needs to access the DRAM then the energy consumption of arithmetic operations would be dwarfed by the DRAM accesses. Luckily the caching schemes that processor systems already use to reduce the latency of memory accesses also serve to reduce the energy cost of memory accesses. A smaller buffer close to the memory such as an L1 cache consumes far less energy than a larger buffer away from the processor, such as L2 cache or DRAM. Therefore cost of data memory accesses would be minimized if a vast majority of data accesses is serviced from the lowest energy L1 cache. Nevertheless even the L1 cache consumes about 50 times the energy of an

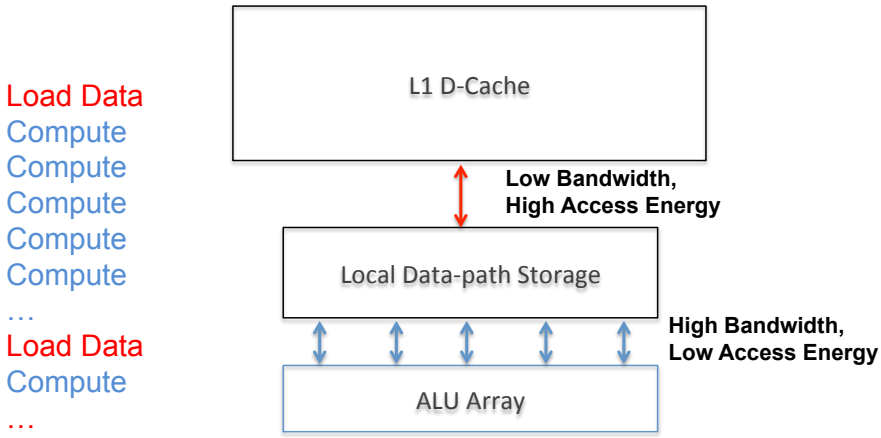


Figure 3.4: Amortize each memory access over a large number of instructions.

arithmetic operation, so even if 100% of data accesses are served by the L1 cache, the execution cost would be still dominated by memory access energy. Moreover we have setup our instructions to perform hundreds of operations on each invocation and a D-Cache could not provide a wide enough interface to load all of that in one go. Executing a series of costly load instructions before each compute operation would kill the efficiency that we attempt to gain through our very wide compute instructions.

Thus if we are to create these very wide efficient instructions, most of the data to operate on should reside in the local register storage of the processor, eliminating the need to go to the costly and lower-bandwidth memories. The standard processor register file however is not suitable for this. As Table 3.2 shows, the cost to access the register file is still somewhat high relative to the compute cost and it is limited to reading a few operands per instruction. To prevent memory accesses from becoming the performance and energy bottleneck, such a processor needs an alternate local storage structure that can offer high bandwidth as well as even lower-energy data access. Figure 3.4 depicts this. We haven't yet discussed what such storage structures might look like. However as was already mentioned, ASICs have had to solve the same data access problem and thus we could study custom hardware units for various applications to understand what type of storage structures are needed. This will be one of the topics for the next chapter.

Figures 3.3 and 3.4 thus summarize the two conditions for highly efficient execution

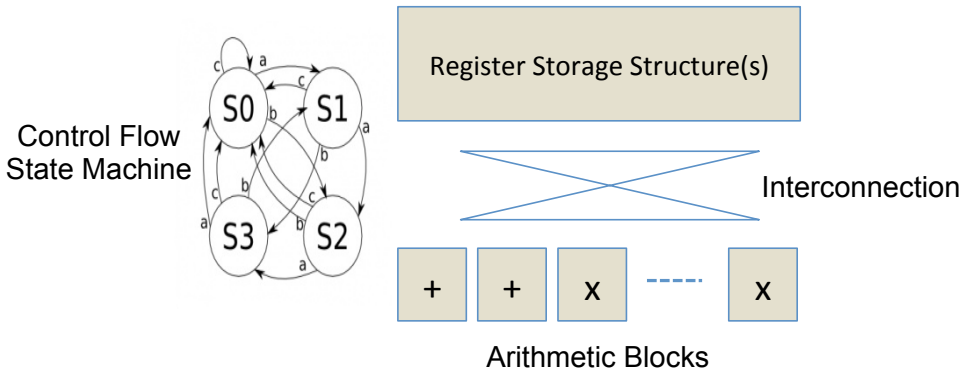


Figure 3.5: Four distinct components of a computing machine.

within a processor framework. We need to perform a large number of simple ops per instruction and most of the data requests should be served by a local low-energy, high bandwidth structure with very infrequent accesses to data memory. Of course not all applications could meet these conditions. Only those applications with a large degree of data-level parallelism could fully utilize such instructions with very wide compute. At the same time the application needs to have a significant degree of data locality if we are to meet the goal of having most data accesses from local storage without the need to go to data memory frequently. This could happen in two ways, either an input data element fetched from data memory is used multiple times in multiple computations, or most computations operate on intermediate data generated and stored in local storage and subsequently consumed locally. While these conditions might seem very restrictive, they span most applications for which efficient hardware implementations exist. At the same time such applications also tend to be the most computationally expensive parts of a system. Thus, creating programmable abstractions that could handle such workloads at ASIC-like efficiency is valuable.

The next section presents our approach to finding these new abstractions.

3.3 Our Methodology

A computing system can be thought of as having the four distinct components illustrated in Figure 3.5: (i) a set of arithmetic units, which perform computations,

(ii) a set of storage elements to hold data on which the computation is performed, (iii) an interface to connect the storage elements to the compute elements in some fashion, and (iv) a control flow logic, which orchestrates the use of these resources in a specific sequence to implement the required algorithm. An ASIC incorporates specific computation and storage elements fused together as per the application need and controlled by hardwired control logic. A processor on the other extreme has a set of general-purpose ALUs, connected to a general-purpose register file with a generic read/write interface and a programmable sequencer to implement virtually any algorithmic flow as needed.

One approach to creating reusable efficient hardware is to start with an ASIC and add limited flexibility to each of the four components described above. An example of this approach would be the video codec chips used in mobile devices, which are designed to handle multiple codec formats [34]. Since most video codec algorithms have a similar algorithmic structure, it is possible to create a single hardware design that targets all of the codecs by adding a pre-defined set of options to each of the four components. By choosing appropriate configuration settings, any of the supported codec variants can be used. The major limitation however is that the flexibility is limited to choosing from a set of predefined closely related algorithmic flows rather than having the ability to create arbitrary new algorithms.

Moving beyond this limited flexibility, based on a predefined set of options, requires a flexible control logic such as the programmable sequencer in a processor. In our work, therefore, we take the opposite approach. Instead of starting from an ASIC and make it more flexible, we start from a processor and explore if it can be converted into a highly efficient computing machine. While gaining efficiency necessarily involves losing some flexibility we would still like the final system to retain true programmability such that the programmable sequencer can create arbitrary algorithm flows using the computation and storage resources at its disposal. As we already alluded to earlier, some specialized architectures such as SIMD units, DSPs and GPUs already take a similar approach and trade some flexibility to gain efficiency and yet retain programmability within their target domain. However these designs are still up to two orders of magnitude less efficient compared to custom hardware. We want

to push this further towards the point where a programmable solution is much closer to custom hardware in terms of energy consumption.

To develop an architecture that can meet these goals, we use a step-wise investigation methodology. Before evaluating a specific design approach we first go back to the basics and perform an in depth analysis of the energy consumption in a general-purpose RISC processor to understand where the energy is wasted. The goal is to understand which components in a processor system spend most of the energy, which then helps us understand what impact various optimizations have on each of these sources of inefficiency. Chapter 3 presents that analysis and derives a set of conditions required to achieve high efficiency within a processor framework.

The next step in our methodology is to apply these insights and create a highly energy-efficient processor system specialized for a specific application i.e. H.264 video encoding. Note that our eventual goal is to create a processor that is useful across a range of application. However, we start with a design for a single application so as to answer a few important questions. First we want to come up with a bound on how close we can get to ASIC performance within a programmable processor framework. Clearly our eventual programmable processor would be at best as good, though more likely worse, compared to a processor customized for a single application in terms of energy consumption. Therefore this experiment gives us a useful upper bound on the efficiency we could hope to achieve. Moreover, building on the insights from this experiment, we can then build more generic programmable abstractions for algorithms in the imaging domain.

The next chapter applies these ideas to an actual application transforming a generic RISC based chip-multiprocessor (CMP) into a processor system optimized to perform H.264 video encoding.

Chapter 4

Removing Processor Overheads - H.264 Case Study

Chapter 3 introduced two conditions for efficient execution in a processor, along with resulting constraints on the applications that we could hope to implement with high efficiency. In this chapter we transform these abstract ideas into a concrete form by building very low-energy processors for high definition H.264 video encoding. The huge energy gap in the ASIC vs. plain vanilla processor implementations of this application have already been highlighted in previous chapters. Now we augment a processor with new instructions and storage structures to bring its energy consumption for H.264 encoding close to ASIC implementation of this application. The new instructions and storage that we add are specialized for H.264 and designed with aforementioned efficiency conditions in mind.

This works as a limit study for efficiency achievable within a processor framework. The eventual goal is to build an efficient core that is useful across a range of algorithms. However, initially we restrict the computation to a single highly data-parallel application to understand how close such a specialized processor can get get to a dedicated ASIC in terms of performance and energy consumption, and whether there are any constraints that stop a specialized processor implementation from reaching ASIC-like efficiency. Moreover the study would help us understand the level of customization required to reach that level of energy efficiency. Many processors today

already employ some broad specializations such as SIMD units [56, 48] targeting data-parallel algorithms, and fused operations[27], which combine frequently occurring instruction sequences into a single instruction. But are those enough to take us close to custom hardware? And if we need to go beyond these then how far do we need to go down the specialization path before we get the efficiency gain we seek?

4.1 H.264 Computational Motifs

Let's start with an overview of H.264 encoder application - H.264 is a block-based video encoder that divides each video frame into 16x16 macro-blocks, and encodes each one separately. Each block goes through five major functions: (i) IME: Integer Motion Estimation (ii) FME: Fractional Motion Estimation (iii) IP: Intra Prediction (iv) DCT/Quant: Transform and Quantization, and (v) CABAC: Context Adaptive Binary Arithmetic Coding.

IME finds the closest match for an image block versus a previous reference image. While it is one of the most compute intensive parts of the encoder, the basic algorithm lends itself well to data parallel architectures. In the software implementation on a 32-bit RISC processor, IME takes up 56% of the total encoder execution time and 52% of total energy.

The next step, FME, refines the initial match from integer motion estimation and finds a match at quarter-pixel resolution. FME is also data parallel, but it has some sequential dependencies and a more complex computation kernel that makes it more difficult to parallelize. FME takes up 36% of the total execution time and 40% of total energy on our base chip-multiprocessor (CMP) design.

IP uses previously encoded neighboring image blocks within the current frame to form an alternate prediction for the current image-block. While the algorithm is still dominated by arithmetic operations, the computations are much less regular than the motion estimation algorithms. Additionally, there are sequential dependencies not only within the algorithm but also with the transform and quantization function.

Next, in DCT/Quant, the difference between a current and predicted image block

is transformed and quantized to generate coefficients to be encoded. The basic function is relatively simple and data parallel. However, it is invoked a number of times for each 16x16 image block, which calls for an efficient implementation. For the rest of this paper, we merge these operations into the IP stage. The combined operation accounts for 7% of the total execution time and 6% of total energy.

Finally, CABAC is used to entropy-encode the coefficients and other elements of the bit-stream. Unlike the previous algorithms, CABAC is sequential and control dominated. While it takes only 1.6% of the execution time and 1.7% of total energy on our base design, CABAC often becomes the bottleneck in parallel systems due to its sequential nature.

We note that IME and FME together dominate the computational load of the encoder, and thus optimizing these algorithms is essential for an efficient H.264 system design. The good news is that these are also the algorithms that most closely match the application characteristics that we had outlined in last chapter for efficient implementation in a processor framework. The next section outlines the methodology we use to create optimized implementations of these and other components of the H.264 encoder.

4.2 Analysis Methodology

As previously mentioned, our aim is to transform a generic multiprocessor into a chip-multiprocessor (CMP) specialized for H.264. The first step then is to create the base multiprocessor implementation for H.264. H.264's video encoding path is very long and suffers from sequential dependencies that restrict parallelism. Chen et al. [17] have carried out a detailed analysis of various H.264 partitioning schemes, and suggest partitioning the H.264 encoder into a four-stage macro-block (MB) pipeline shown in Figure 4.1. This mapping exploits task level parallelism at the macro block level and significantly reduces the communication bandwidth requirements between the pipeline stages. This mapping has been subsequently adopted by ASIC implementations such as [16]. We use a similar partitioning, and modify the H.264 encoder reference code JM 8.6 [31] to remove dependencies and allow mapping of the five major algorithmic

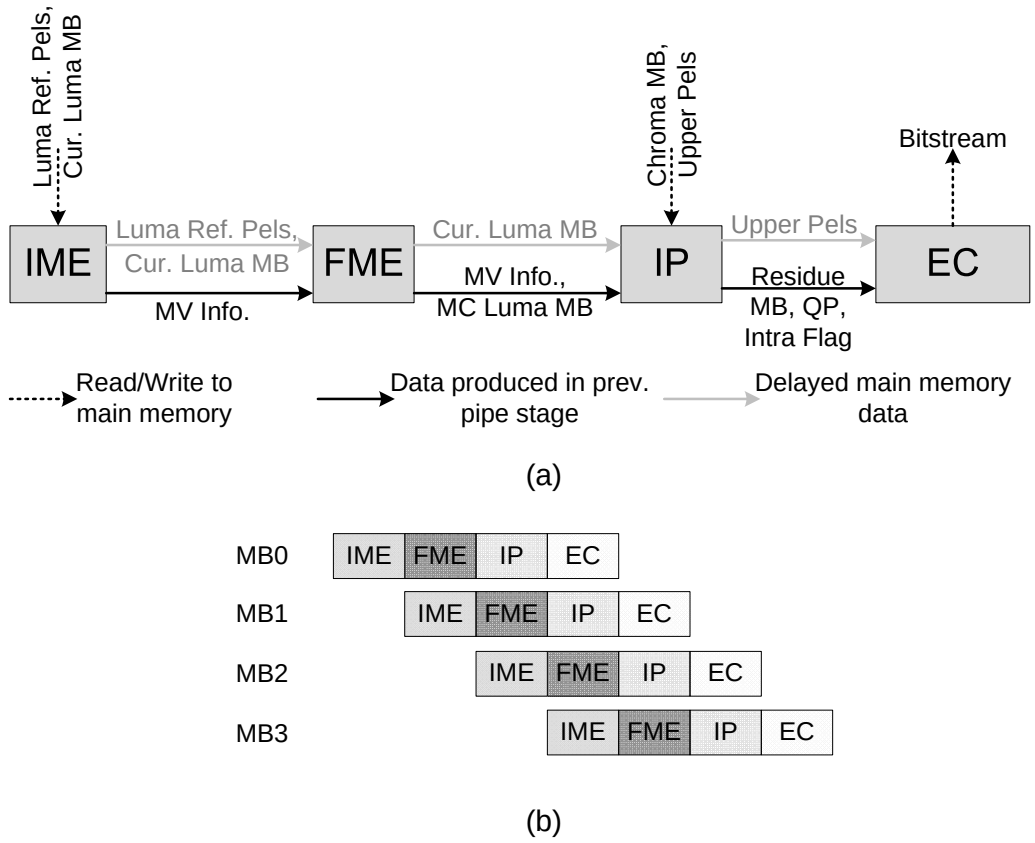


Figure 4.1: Four-stage macroblock partition of H.264. (a) Data-flow between stages. (b) How the pipeline works on different macroblocks. IP includes DCT + Quant. EC is CABAC.

blocks to this pipeline.

In the base system, we map this four-stage macro-block partition to a chip-multiprocessor (CMP) system with four Tensilica embedded processors [25] running at 400Mhz in 90nm technology. All four processors are identical—each of these is a 32-bit single issue RISC core with 16KB 2-way set associative instruction and data caches. Each of these processors will be individually customized using a processor extension language supported by Tensilica toolchain [51].

Figure 4.2 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

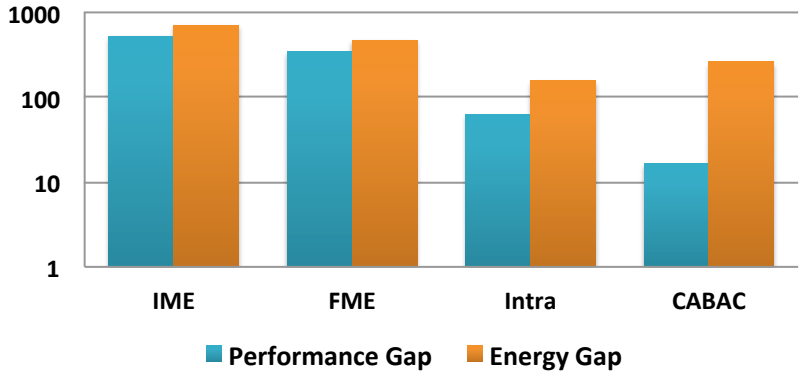


Figure 4.2: The performance and energy gap for base CMP implementation when compared to an equivalent ASIC. Intra combines IP, DCT, and Quant.

in Figure 4.3) 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, DCT, Quant and CABAC are much smaller parts of the total energy/delay, even they need about 100x energy improvement to reach ASIC levels.

Note that the processor performance numbers in Figure 4.2 assume that each processor is running independently, achieving the highest performance possible for the algorithm assigned to it. However, when these processors operate in a pipeline, IME processor becomes the bottleneck, limiting system performance to only 0.06 FPS, even though IP and CABAC processors can independently achieve 0.48 FPS and 1.82 FPS respectively. It could be argued that the base system performance can be improved with a balanced pipeline giving more processor resources to IME and FME, which constitute over 90% of the computation. However this would have negligible impact on energy consumption, and performance improvement would also be limited to less than 2x. Thus, this imbalance has negligible contribution in the 500x performance and energy gap that we are trying to close.

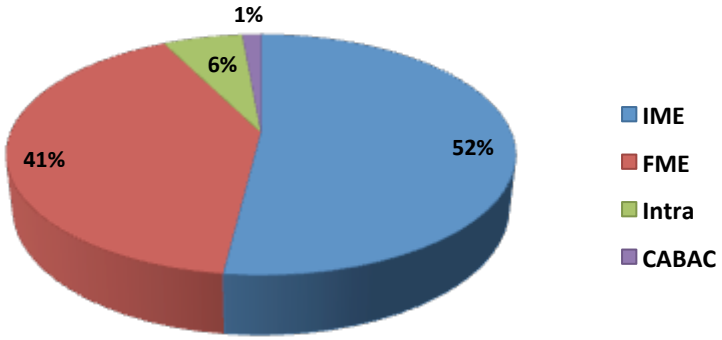


Figure 4.3: Processor energy breakdown for a 32-bit RISC implementation of the high definition H.264 encoder. The graph shows distribution across H.264 sub-algorithms. Note: Intra combines IP, DCT, and Quant.

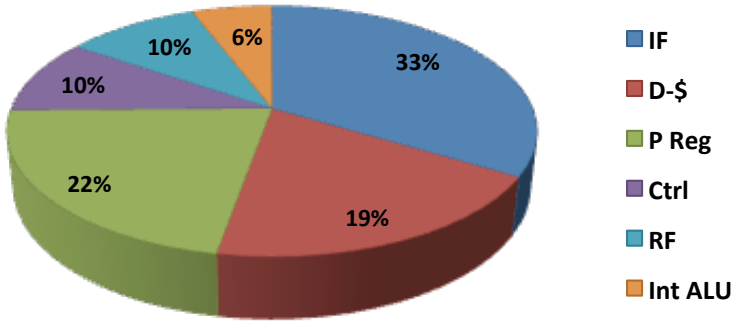


Figure 4.4: Processor energy breakdown for base implementation. IF is instruction fetch/decode. D-$ is data cache. P Reg includes the pipeline registers, buses, and clocking. Ctrl is miscellaneous control. RF is register file. FU is the functional units.

Table 4.1: SIMD resource sizes used for each processor. CABAC does not benefit from SIMD, so no SIMD unit is added to the EC processor.

	SIMD Width	Element Width	RegFile Size	Memory Port	VLIW Slots
IME	16 Elements	8 bits	16 Vectors	128-bit	2
FME	18 Elements	9 bits	32 Vectors	128-bit	2
IP	8 Elements	8 bits	16 Vectors	64-bit	2

At approximately 8.6B instructions to process one frame, our base system consumes about 140 pJ/instruction in 90nm technology—a reasonable value for a general-purpose system. 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 4.4. 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.

So now we customize each processor for the specific algorithms running on that particular processor. The customization is done by adding custom instructions to each processor using a processor extension language supported by Tensilica processors [51]. As noted earlier, we first want to analyze how far we could go with broad specializations such as SIMD units. The next section talks about the results of that.

4.3 Broad Specializations

At first, we consider acceleration strategies that represent current state-of-the-art optimized CPUs. These are relatively general-purpose data parallel optimizations and consist of SIMD extensions as well as use of multiple instruction issue / cycle (we use the VLIW strategy). Some extension units, such as those in GPUs and Intel's SSE [30], further incorporate a limited degree of algorithm specific customization in the form of operation fusion—the creation of new instructions that combine frequently occurring sequences of instructions. We incorporate similar fused operations

at this step. Like Intel's SSE, these new instructions are constrained to the existing instruction formats (i.e. two input operands, one output) and fit existing datapath structures.

Using Tensilica's FLIX (Flexible Length Instruction eXtension) [57] feature we create processors with up to 3-slot VLIW instructions. Using Tensilica's extension language called TIE [57], we add SIMD execution units to the base processor. SIMD widths and vector register file sizes vary across each algorithm and are summarized in Table 4.1. Each SIMD instruction performs multiple operations (8 for IP, 16 for IME and 18 for FME), reducing the number of instructions and increasing performance. At the same time this has the effect of amortizing the fetch and control cost as already discussed. VLIW instructions execute 2 or 3 operations per cycle, further reducing cycle count. 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 4.4 get a reduction through use of these enhancements.

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, these instructions are restricted to use up to two input operands and can produce only one output. As an example we add a SIMD sum-of-absolute-differences instruction to the IME processor. This instruction combines three SIMD operations—SIMD subtraction, SIMD absolute value operation, and SIMD 4-to-1 reduction—into a single multi-cycle instruction. Operation fusion improves energy efficiency by aggregating more operations into a single instruction, and also 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, e.g. multiplication implemented using shifts and adds.

We now analyze how these designs fare in terms of the efficiency conditions we have derived. Figures 4.5 and 4.6 summarize the results. CABAC is not data parallel and benefits only from LIW and op fusion with a speedup of merely 1.1x. For IME, FME and IP, which are data-parallel algorithms, SIMD, VLIW and op fusion together

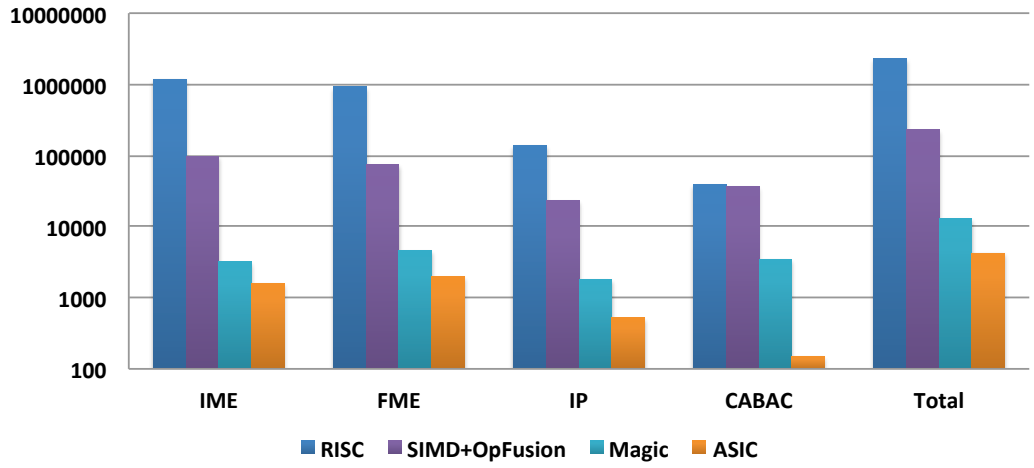


Figure 4.5: Each set of bar graphs represents energy consumption (mJ) at each stage of optimization for IME, FME, IP and CABAC respectively. The first bar in each set represents base RISC energy; followed by RISC augmented with SIMD+OpFusion; and then RISC augmented with magic instructions. The last bar in each group indicates energy consumption by the ASIC.

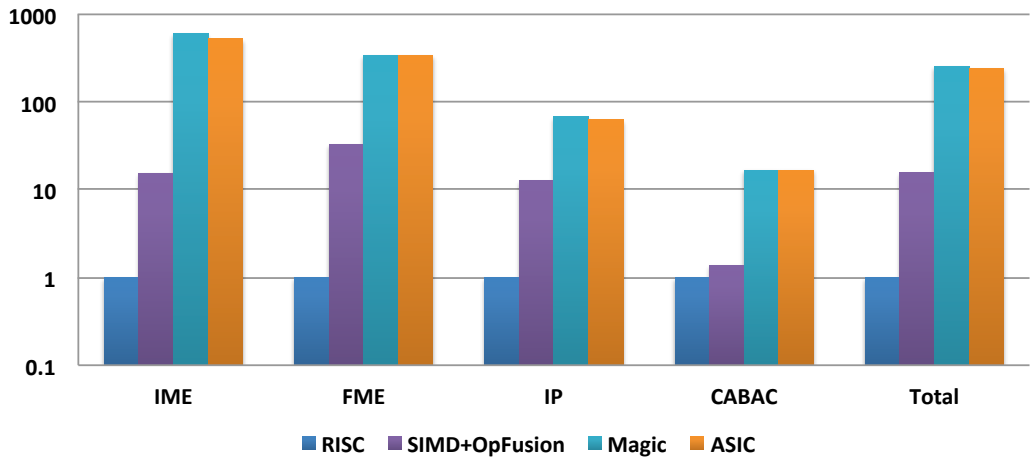


Figure 4.6: Speedup at each stage of optimization for IME, FME, IP and CABAC.

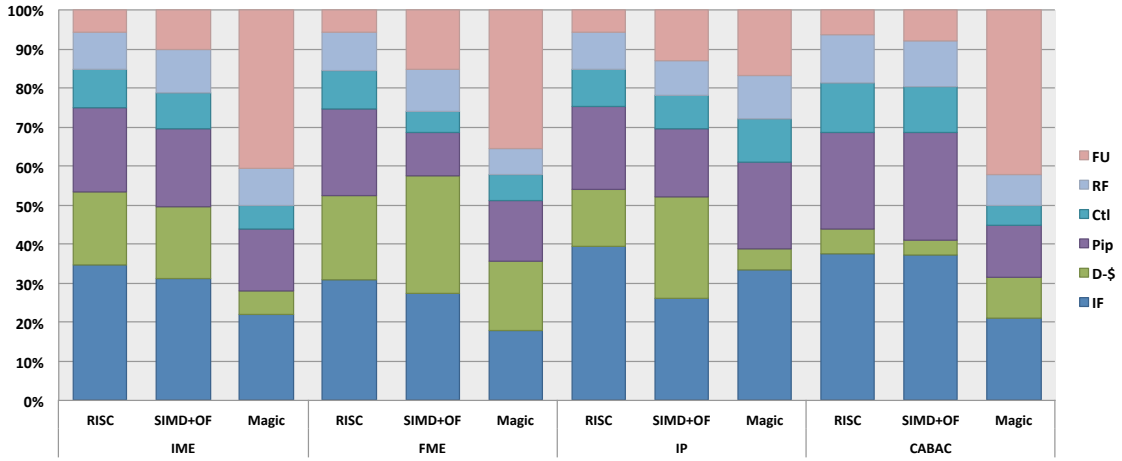


Figure 4.7: Processor energy breakdown for H.264. SIMD+OF is SIMD with operation fusion. IF is instruction fetch/decode. D-$ is data cache. Pip is the pipeline registers, buses, and clocking. Ctl is random control. RF is the register file. FU is the functional elements. Only the top bar or two (FU, RF) 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.

result in about 10-30 simple operations per instruction. As a result processors achieve speedups of around 15x, 30x and 10x respectively. However, 10-30 operations per instruction are not enough to amortize the instructions overheads, and thus as Figure 4.7 shows these overheads still dominate the instruction cost. Similarly while wider accesses help reduce energy cost of register and memory accesses, these costs are still substantial. Overall, the application gets an energy efficiency gain of almost 10x, but still uses greater than 50x more energy than an ASIC.

4.3.1 Building a Wider SIMD

The obvious way to improve over these result is to build a wider SIMD engine, however simply creating wider arithmetic instructions does not solve the problem of feeding enough data to keep these large number of arithmetic arrays busy. Trying to naively scale the SIMD model quickly becomes limited by load / store operations and/or data rearrangement operations.

To understand why that's the case let's assume we create a huge 256-way SIMD

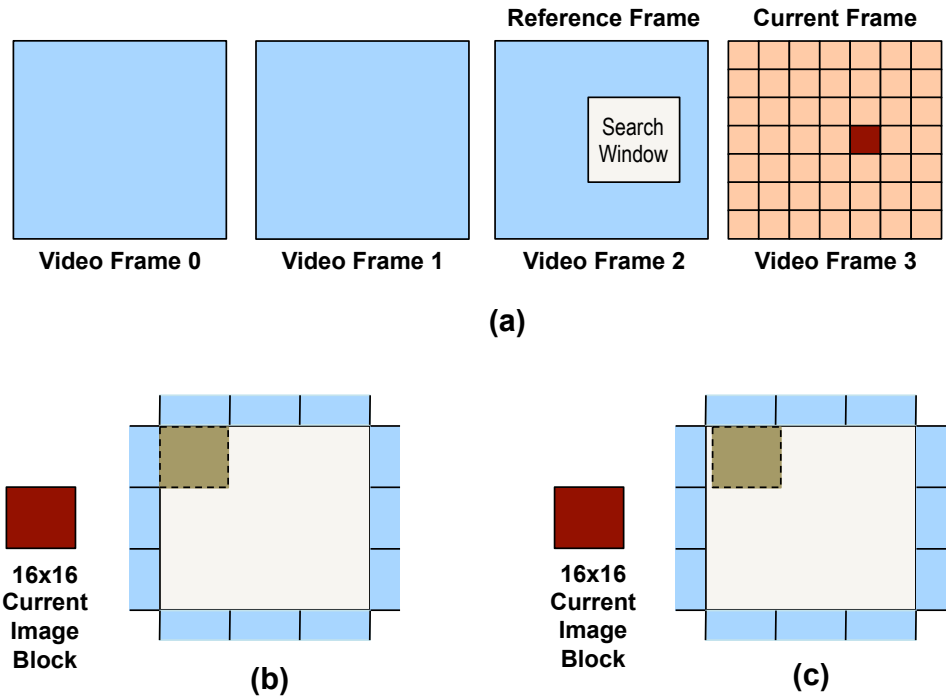


Figure 4.8: IME algorithm: (a) IME operates on a stream of video frames. Here we are encoding frame 3 using a previous frame in the stream as reference (frame 2). The red square is a 16x16 image *macroblock*, and the goal is to find its closest match within a 48x48 search window in the reference frame. (b) A blowup of the search windows, showing the first 16x16 search location in the search window that is compared against the current macroblock (c) The second search block is shifted from the first search block by just one pixel - most of the pixels from previous comparison are re-used.

unit and try to map one of the H.264 algorithms - Integer Motion Estimation - on it. Figure 4.8 shows an overview of the integer motion estimation algorithm. The algorithm operates on a sequence of video frames. Each frame is divided into 16x16 image blocks called macroblocks (MBs), and the goal is to find for each MB in current frame, the closest matching 16x16 patch in a previous *reference* frame. The search for the matching block is carried out over a 48x48 window in the reference frame. Figures 4.8(b) and 4.8(c) show the first two search locations within the search window. As illustrated each search location is offset from the previous one by only one pixel in horizontal or vertical direction.

This has a few implications. First there is a huge amount of computation to be performed - each comparison is a sum-of-absolute-differences operation requiring 256 difference operations, 256 absolute value operations, and finally a reduction or summation step to combine these 256 results. And for a 48x48 search range, 1024 such comparisons are required for just a single 16x16 image MB. At the same time the operations are highly data-parallel making this algorithm a good match for a SIMD-like computation model. At the same time since most of the pixels are re-used from one search location to the next, there is a potential of extracting a large amount of data-reuse if there exist the right storage structure in the datapath to exploit this data-flow.

Now imagine mapping this to the very wide 256-element SIMD unit depicted in Figure 4.9. Loading the first search location from Figure 4.8(b) into one of the wide 256-element vector registers requires sixteen 128-bit loads. Note that the data is not stored in contiguous memory locations so having a wider access port to data memory would not help. Once we process the first search location, it is time to move on to the next location depicted in 4.8 (c). Most of the data for this second location is already present in the vector register used for first search location, and we just need one extra pixel from each of the 16 rows. However reading these 16-pixels and inserting them at the right places in the 256-element register would require a large number of additional loads and data re-arrangement instructions. The other option is to reload the second search window from the data memory, avoiding the need for data shuffling but still incurring costly D-Cache accesses and load operations (further complicated by the fact that these data accesses would go to unaligned addresses). Clearly such an implementation would severely limit the utilization of the wide compute instructions, consequently limiting any gain in energy efficiency. We observed similar constraints in mapping FME and IP to a wide SIMD unit.

The next section explores adding more specialized instructions to the processors, inspired by the ASIC implementation.

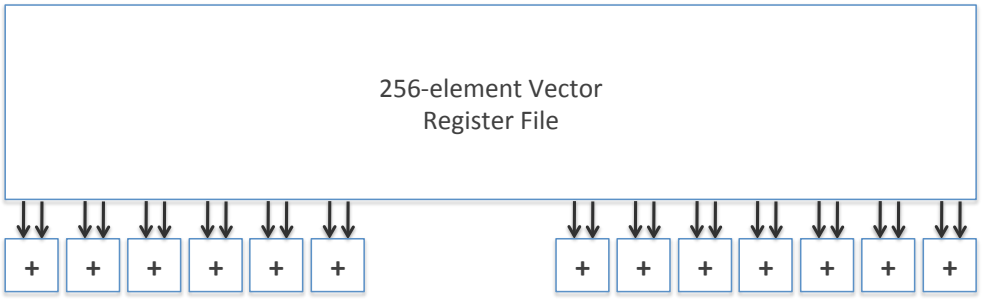


Figure 4.9: A very wide hypothetical SIMD unit with 256-element compute array and a register file with 256-element wide vector registers.

4.4 Application Specific Customizations

The root cause of the large energy difference between processor and ASIC implementations of H.164 is that the basic operations in H.264 are very simple and low energy. They only require 8- to 16-bit integer operations, so the fundamental energy/-operation is on the order of hundreds of femtojoules in a 90nm process. Thus these operations are even lower energy compared to the 32-bit arithmetic operations in our standard RISC and as a result all the processor overheads that we have discussed—instruction fetch, register fetch, data fetch, control, and pipeline registers—are even greater (140 picojoules vs. a few hundred femtojules). Thus even with SIMD and fusion combining to issue tens of these extremely low energy operations per cycle we still have a machine where around 90% of the energy is going into overhead functions.

As the previous section has shown, simply scaling a traditional SIMD unit to a wider number of elements is not the answer either - the resulting unit would not have storage and compute structures matched to the data-flow requirements of the problem. An ASIC incorporates hardware that has minimal control overheads, as well as has storage structures specialized to the algorithm. These storage structures not only help retain most of the data to be processed within the datapath, but also provide much higher bandwidth connection to a large number of arithmetic units than is possible for a standard register file. These features allow the ASIC to exploit large amounts of parallelism efficiently, achieving high performance at low energy cost.

To make possible our approach of creating very wide instructions that amortize

the per-instruction energy overheads over hundreds of these simple operations, similar specialized storage structures must be added to the processor datapath. These custom storage structures have algorithm-specific communication links to directly feed large amounts of data to arithmetic units without explicit register accesses.

Once this hardware is in place, the machine can issue "magic" instructions that accomplish large amounts of computation at very low cost. This type of structure eliminates almost all the processor overheads for these functions by eliminating most of the communication and control cost associated with processors. We call these instructions "magic" because they can have a large effect on both the energy and performance of an application and yet they would be 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 greater effort on the part of the designer. Since the IP stage uses techniques similar to FME, the rest of the section will focus on IME, FME and CABAC.

4.4.1 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. As discussed earlier in Section 4.3.1, finding the best match requires scanning one image block over a larger piece of the image, and while this requires a large number of calculations, it also has very high data locality. Figure 4.10 shows the custom datapath elements added to the IME processor to accelerate this function. At the core is a 16x16 SAD array, which can perform 256 SAD operations in one 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 that 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 it can be reduced in place (summed) to create the single desired output. Furthermore,

because we need to check many overlapping search locations, the custom storage structure has support for parallel shifts both horizontal and vertical directions, thus allowing one to shift the entire comparison image in only one cycle. Thus unlike the SIMD example of Section 4.3.1, the overlapping block can be easily re-used without a need for a large number of loads or data-rearrangement operation. In fact with this arrangement once the initial 16 rows have been filled into the shift register, only a single 256-bit load is needed for every 16 comparisons. Thats just a single 256-bit load for every 4K absolute difference operations!

Thus this shift register structure drastically reduces the instructions wasted on loads, shifts and pointer arithmetic operations as well as data cache accesses, and enables 256 fused absolute-difference operations per instruction. "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, functional units finally consume 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 generic data-parallel engines.

4.4.2 FME Strategy

FME improves the output of the IME stage by refining the alignment to a fraction of a pixel. To perform the fractional alignment, the FME stage interpolates one image to estimate the values of a 4x4 pixel block at fractional pixel coordinates. This operation is done by a filter and upsample block, which again has high arithmetic intensity and high data locality. In H.264, upsampling uses a six-tap FIR filter that requires one new pixel per iteration. To reduce instruction fetches and register file transfers, we augment the processor register file with a custom 8-bit wide, six entry shift register structure that 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

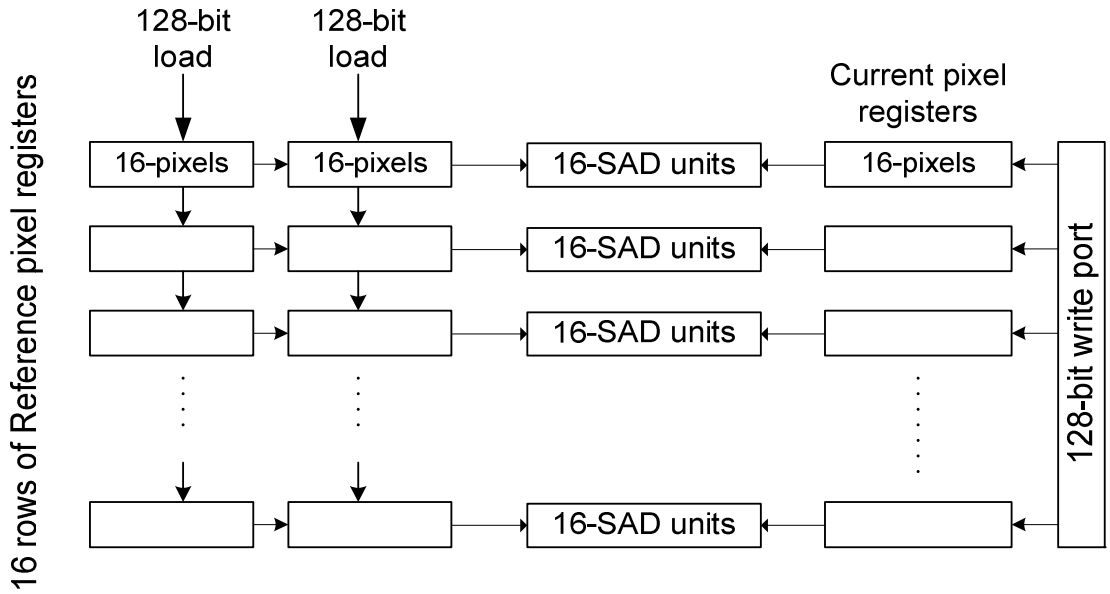


Figure 4.10: Custom storage and compute for IME 4 X 4 SAD. current and ref-pixel register files feed all pixels to the 16 X 16 SAD array in parallel. Also, the ref-pixel register file allows horizontal and vertical shifts.

entries can now be accessed in parallel and we create a six input multiplier/adder which can do the calculation in a single cycle and also can be implemented much more efficiently than the composition of normal 2-input adders. Finally, since we need to perform the upsampling in 2-D, we build a shift register structure that stores the horizontally upsampled data, and feeds its outputs to a number of vertical upsampling units (Figure 4.11).

This transformation yields large savings even beyond the savings in instruction fetch 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 16x16 pixel blocks to 4x4 blocks, and not

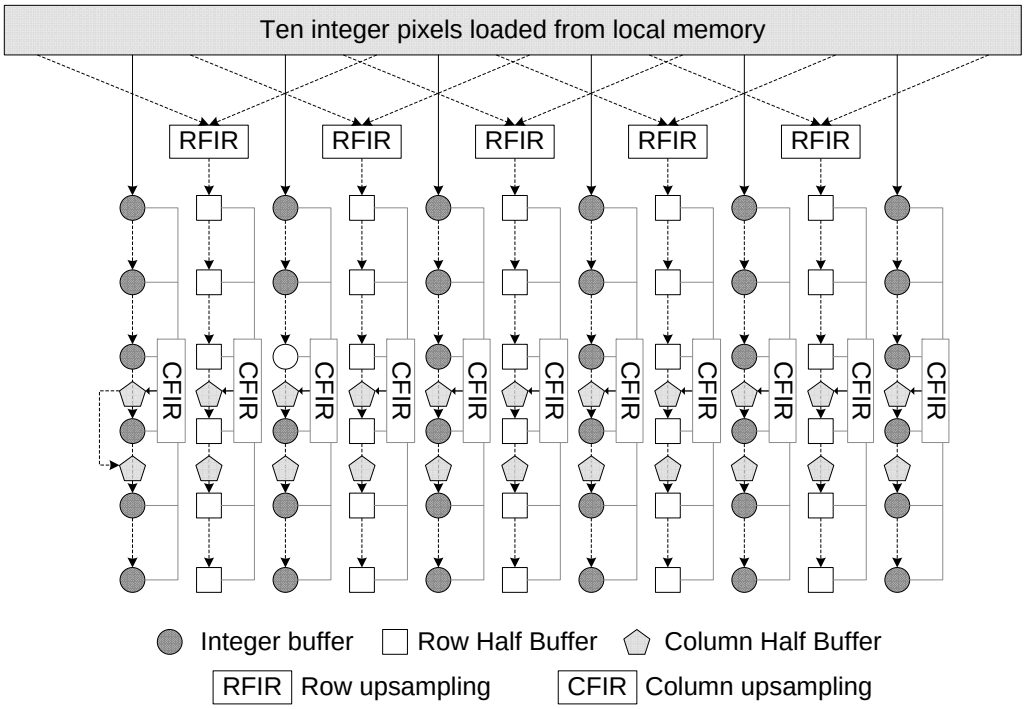


Figure 4.11: FME upsampling unit. Customized shift registers, directly wired to function logic, result in efficient upsampling. Ten integer pixels from local memory are used for row upsampling in RFIR blocks. Half upsampled pixels along with appropriate integer pixels are loaded into shift registers. CFIR accesses six shift registers in each column simultaneously to perform column upsampling.

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. That results in code replication and consequently increases I-fetch power because of the larger I-cache required.

To avoid these issues, the custom hardware upsampler processes 4x4 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 4.7 shows, 35% of the energy is now going into the functional

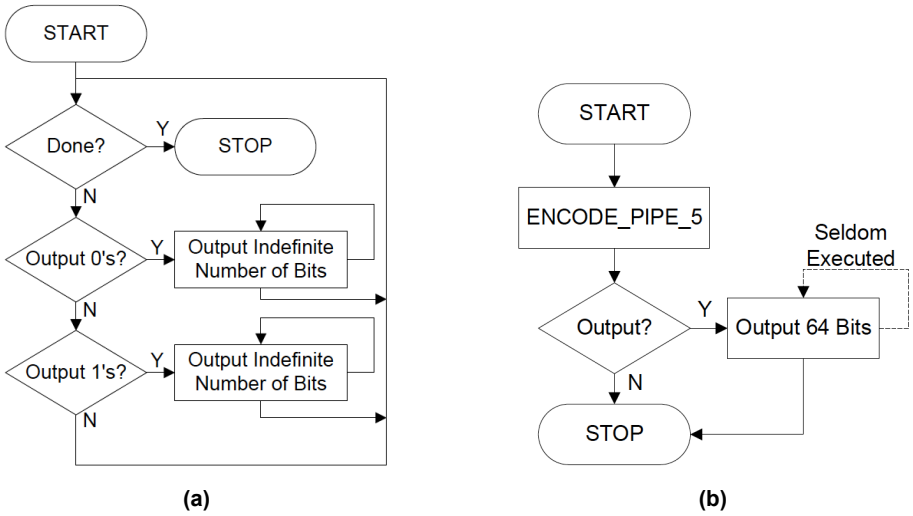


Figure 4.12: CABAC arithmetic encoding loop. (a) H.264 reference code. (b) After insertion of magic instructions. Much of the control logic in the main loop has been reduced to one constant time instruction ENCODE_PIPE_5.

units, and again the energy efficiency of this unit is close to an ASIC.

4.4.3 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. 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 with small amounts of computation but complex control flow. We break 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 4.12(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

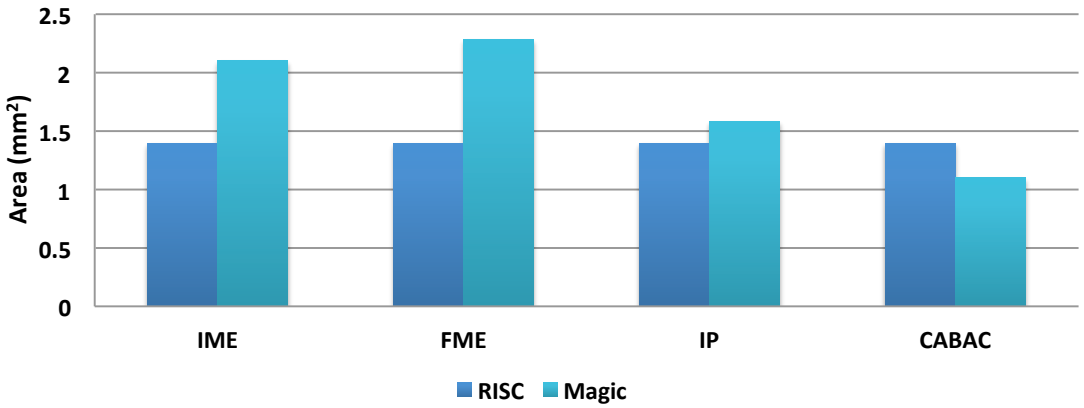


Figure 4.13: Area in mm^2 for magic instruction designs compared to the area of base processor (90nm technology). These numbers include both processor core area, as well as I-Cache and D-Cache area.

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 4.12(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 that help reduce datapath energy. These modifications are inspired by the ASIC implementation described in [53]. CABAC is optimized to achieve the bit rate required for H.264 level 3.1 at 720p video resolution.

4.5 Area Cost of Magic Instructions

Figure 4.13 shows the area for each of the four customized processors we have created. The numbers for RISC include the processor core area, as well as the area of L1 caches.

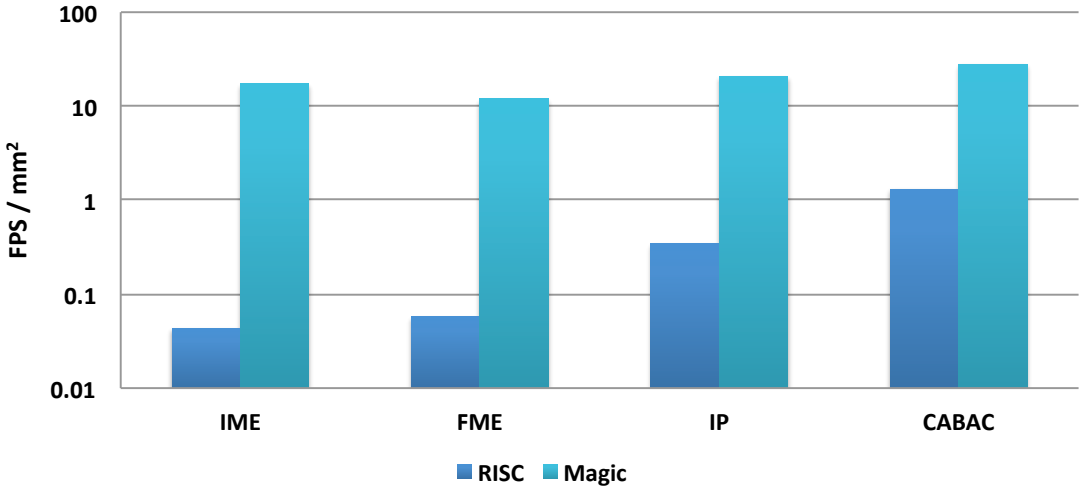


Figure 4.14: Area efficiency of magic vs. RISC cores.

For magic processors, the area includes processor core area, extension unit area, and the area of L1 caches. Note that the L1 caches for each magic core are sized according to the requirements of the optimized code running on that core. In some cases these sizes can be substantially lower than the starting base core. This explains why the final CABAC processor is smaller than the base processor.

IME and FME add substantially large extension units to the processor. Some of the area is reclaimed by the reduction in I-cache and D-cache sizes, but the total processor area still increases by around $0.7mm^2$ and $0.9mm^2$ for IME and FME respectively. However, unlike the base processor, these units make a far more efficient use of this area. As Figure 4.14 shows, customized processors for IME and FME achieve more than two orders of magnitude higher FPS/mm^2 figures over their RISC counterparts. IP and CABAC require much smaller extensions, as the computation in these tasks is much simpler. Still, these relatively small extensions enable a significant performance boost, and as a result the area efficiency (FPS/mm^2) goes up by more than an order of magnitude.

4.6 Magic Instructions Summary

To summarize, the magic instructions perform up to hundreds of operations each time they are executed, so the overhead of the instruction is better balanced by the work performed. This is hard to do in a general way, since bandwidth requirements and utilization of a larger SIMD array would be problematic. We solved this problem by building custom storage units tailored to the application, and then directly connecting the necessary functional units to these storage units. These custom storage units greatly amplified the register fetch bandwidth, since data in the storage units are 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. For CABAC total performance gain is 17x and energy gain is 8x. Figure 4.7 provides the final energy breakdowns. The efficiencies found in these custom datapaths are impressive, since, in H.264 at least, they take advantage of data sharing patterns and create very efficient multiple-input operations. This means that even if researchers are able to a create a processor that decreases the instruction and data fetch parts of a processor by more than 10x, these solutions will not be as efficient as solutions with "magic" instructions.

4.7 Beyond H.264

Achieving ASIC-like efficiency required 2-3 special hardware structures for each sub-algorithm, which is significant customization work. Some might even say we are just building an ASIC in our processor. We feel that adding this hardware in an extensible processor framework has many advantages over just designing an ASIC. These advantages come from the constrained processor design environment and the software, compiler, and debugging tools available in this environment. Many of the low-level issues, like interface design and pipelining, are automatically handled. In

addition, since all hardware is wrapped in a general-purpose processor, the application developer retains enough flexibility in the processor to make future algorithmic modifications.

However our eventual goal is to create efficient processors that are useful across a wide range of algorithms and not just a single application, and that's the topic for the next chapter.

Chapter 5

Convolution Engine

Earlier in Section 3.3 we identified four major components of every hardware system. Table 5.1 compares various hardware solutions we have discussed so far, in terms of flexibility they offer in each of these components. Our specialized processors from last chapter get very close to ASIC level efficiencies by adding custom datapath elements to the processor core. Unlike ASICs, these are built around a programmable program sequencer, however, as Table 5.1 points out, they still have algorithm-specific arithmetic units, algorithm-specific storage structures and algorithm-specific interconnection just like ASICs, and that limits their applicability to a particular algorithm. Broadening the applicability of these processors requires making the storage and compute structures more flexible as well. Unfortunately trying to cover too broad a space would take us back to the SIMD computation model, which targets the entire domain of data-parallel algorithms, but in the processes loses at least an order of magnitude in energy efficiency. Thus, the goal of this chapter is to find the right balance between flexibility and efficiency that would maximize the re-use potential of the processor without spending too much extra energy.

We have also learnt in the previous chapter that efficient designs are built around specialized storage structures and interconnections which facilitate exploiting the data-reuse and parallelism inherent in an algorithm. This tuning of storage and interconnect to an algorithm is the part that makes the resulting hardware algorithm-specific instead of general. A key realization however is that data-flow patterns are

Table 5.1: Comparison of various hardware solutions in terms of the flexibility they offer in various components. Last column summarizes the energy efficiency achieved by each design point relative to the base RISC, for H.264 study in previous chapter. Designs are listed in the order of increasing efficiency

	Control Flow	Arithmetic	Storage	Interconnection	Eficiency
General Processor	Programmable	General	General	General	1
SIMD Processor	Programmable	Somewhat Specialized	Somewhat Specialized	Somewhat Specialized	10
Specialized Processor	Programmable	Custom	Custom	Custom	250
ASIC	Hard Wired	Custom	Custom	Custom	500

often not unique to an algorithm. In fact algorithms in a domain tend to share similar data-flow patterns. Therefore the efficient storage and interconnection structures that we create need not be limited to serving a single application. Adding a small degree of flexibility to these structures can make them useful across a range of algorithms based on similar data-flow. Since we expect the efficiency cost of this added flexibility to be small, this could provide a way to significantly increase the reusability of the processor without incurring a big efficiency loss. In other words we argue that to build efficient yet reusable processors, we should specialize for a data-flow rather than specializing for an algorithm or application.

We present an example, the Convolution Engine (CE), specialized for the convolution-like data-flow that is common in computational photography, image processing, and video processing applications. CE achieves energy efficiency by capturing data reuse patterns, eliminating data transfer overheads, and enabling a large number of operations per memory access. We quantify the tradeoffs in efficiency and flexibility and demonstrate that CE is within a factor of 2-3x of the energy and area efficiency of custom units optimized for a single kernel. CE improves energy and area efficiency by 8-15x over a SIMD engine for most applications.

5.1 Convolution Abstraction

Section 2.3 previously highlighted the growing number of computational photography, image processing, and video processing applications in mobile systems. Interestingly, even though the range of applications is very broad, a large number of kernels in these applications look similar. These are kernels where the same computation is performed

over and over on (overlapping) stencils within, and across frames. For example, demosaicing takes squares of n x n Bayer patterned pixels and interpolate the RGB values for each pixel. This is similar to the sum-of-absolute-differences (SAD) applied on n x n stencils used for motion estimation that we have already discussed in previous chapter. Other examples that fit this pattern include feature extraction and mapping in scale-invariant-feature-transform (SIFT), windowed histograms, median filtering, 2D FIR filtering and many more. We categorize this class of stencil operation as convolution-like.

Convolution is the fundamental building block of many scientific and image processing algorithms. Equation 5.1 and 5.2 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.

$$(Img * f)[n] \stackrel{\text{def}}{=} \sum_{k=-\infty}^{\infty} Img[k] \cdot f[n-k] \tag{5.1}$$

$$(Img * f)[n,m] \stackrel{\text{def}}{=} \sum_{l=-\infty}^{\infty} \sum_{k=-\infty}^{\infty} Img[k] \cdot f[n-k, m-l] \tag{5.2}$$

We generalize the concept of convolution by identifying two components of the convolution: a *map* operation and a *reduce* operation. In Equation 5.1 and 5.2, 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 5.2 from $x \cdot y$ to $|x - y|$ while leaving the reduce operation as summation, yields a sum of-absolute-differences (SAD) function for H.264 motion estimation. Further replacing the reduce operation from $\sum$ to *max* will yield a max of absolute differences operation. Equation 5.3 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 5.3) are the pseudo instructions, and c is the size of the convolution.

$$(Img \overset{CE}{*} f)[n,m] \overset{\text{def}}{=} R_{|l|<c}\{R_{|k|<c}\{Map(Img[k], f[n-k, m-l])\}\} \tag{5.3}$$

Unfortunately programmable platforms today do not handle convolution-like computation efficiently because their register files are not optimized for convolution. To address this issue, traditional camera and camcorder manufacturers typically use their own ASICs to implement camera pipelines such as Canon's DIGIC series [1], Nikon's Expeed processors [18] and Sony's Bionz processors [19]. Cellphone SoCs, including TI's OMAP [2], Nvidia's Tegra [4] and Qualcomm's Snapdragon [3] platform take a similar approach.

In contrast to the custom hardware approach used by current imaging solutions, we have created a flexible domain-specific processor core for all these applications which we call Convolution Engine (CE). Section 5.2 introduces the four application kernels we target in this study, and how they map to the generalized convolution abstraction. Section 5.3 gives an overview of how convolution operations perform on existing data-parallel architectures and what are the limitations. Section 5.4 describes the CE architecture focusing primarily on features that improve energy efficiency and/or allow for flexibility and reuse. Next we compare the energy efficiency of the Convolution Engine to general-purpose cores with SIMD extensions, as well as highly customized solutions for individual kernels. Section 5.5 first introduces the methodology for that evaluation and Section 5.6 shows that the CE is within a factor of 2-3x of custom units and almost 10x better than the SIMD solution for most applications.

5.2 Target Applications

The following sections describe the two test applications—motion estimation and SIFT.

5.2.1 Motion Estimation

Motion estimation is a key component of many video codecs including H.264 encoding. As previously noted in Chapter 4, motion estimation accounts for ∼90% of the execution time in the software implementations of H.264 encoder. Here we review the two H.264 motion estimation steps—IME and FME—, and describe how to express these in terms of *map* and *reduce* operations of Equation 5.3.

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. Note how SAD fits quite naturally to a convolution engine abstraction: the *map* function is absolute difference and the *reduce* function is summation.

Fractional Motion Estimation: 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*.

5.2.2 SIFT

Scale Invariant Feature Transform (SIFT) looks for distinctive features in an image [38]. Typical applications of SIFT use these features to find correspondences between images or video frames, performing object detection in scenes, etc. To ensure scale invariance, Gaussian blurring and down-sampling is performed on the image to create a pyramid of images at coarser and coarser scales. A Difference-of-Gaussian (DoG) pyramid is then created by computing the difference between every two adjacent image scales. Features of interest are then found by looking at the scale-space extrema in

Table 5.2: Mapping kernels to convolution 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.

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

the DoG pyramid [38].

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.

5.2.3 Mapping to Convolution Abstraction

Table 5.2 summarizes the kernels we use and how they map to the convolution abstraction. The table further describes how each algorithm is divided into the *map* and *reduce* operator and what is its data-flow pattern such as 2D convolution or 1D vertical convolution. Although, two kernels could have identical *map* and *reduce* operators and data-flow patterns, they may have differences in the way they handle the data. For example up-sampling in FME produces four times the data of its input image. These requirements differentiate them from simple filtering operations and require additional support in hardware as we will see next.

5.3 Convolution on Current Data-Parallel Architectures

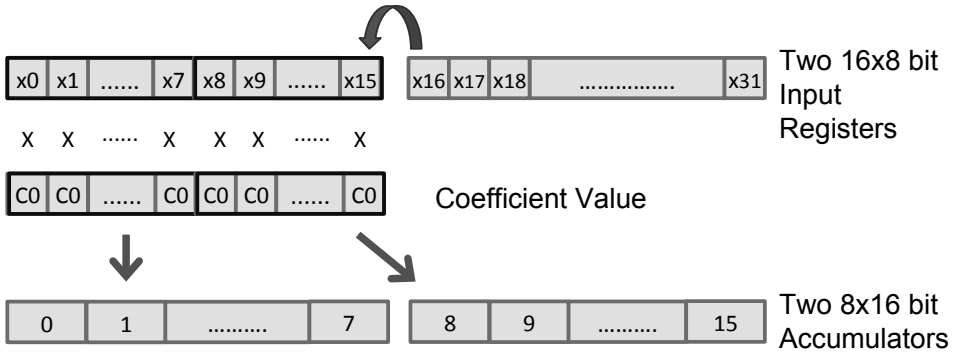
Convolution operators are highly compute-intensive, particularly for large stencil sizes, and being data-parallel they lend themselves to vector processing. However, existing SIMD units are limited in the extent to which they can exploit the inherent

$$Y_0 = x_0 * c_0 + x_1 * c_1 + x_2 * c_2 + x_3 * c_3 + \ldots\ldots + x_n * c_n$$
$$Y_1 = x_1 * c_0 + x_2 * c_1 + x_3 * c_2 + x_4 * c_3 + \ldots\ldots + x_{n+1} * c_n$$
$$Y_2 = x_2 * c_0 + x_3 * c_1 + x_4 * c_2 + x_5 * c_3 + \ldots\ldots + x_{n+2} * c_n$$
.....

Figure 5.1: 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.

parallelism and locality of convolution due to the organization of their register files. Section 4.3.1 in the last chapter already illustrated this for H.264 Integer Motion Estimation, which is based on a 2D convolution operation. We now discuss another example, this time of a 1D filtering operation. Figure 5.1 presents equations for an n-tap 1D convolution. Figure 5.2 demonstrates 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 [56], and the SIMD datapath is similar to those found in many current processors. To enable the datapath to utilize the vector 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.

As we have already established, given the short integer computation that is required, one needs a large amount of parallelism per instruction to be energy efficient. While n-tap 1D convolution 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 5.2, 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.




```
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 5.2: 1D horizontal 16-tap convolution on a 128-bit SIMD machine, similar to optimized implementation described in [56]. 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.

GPUs also 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 [55] using GPGPUSim simulator [12] with GPUWattch energy model [35]. We simulate Nvidia's GTX480 GPU, which has 15 *Streaming Multiprocessors*, each with 32 CUDA cores, for a total of 480 CUDA cores. The GPU implementation runs forty times faster compared to an embedded 128-bit SIMD unit with 16x8bit vector operations, reflecting the larger number of computing units in the GPU. However with a

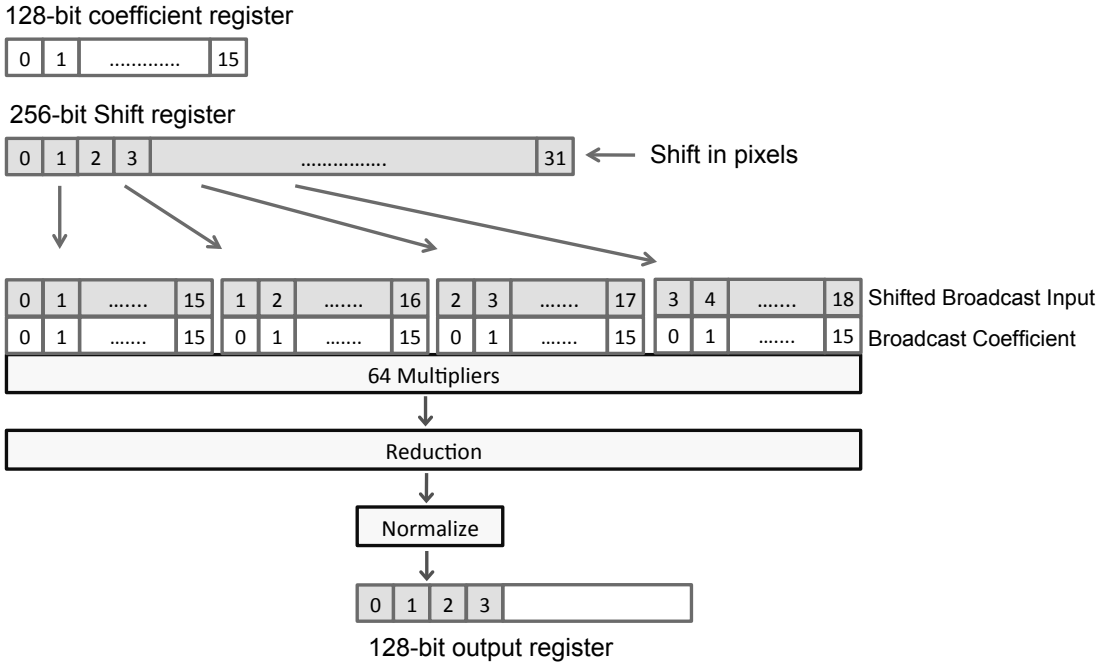


Figure 5.3: 1D horizontal 16-tap convolution using a shift register with shifted broadcast capability. Computes 4 output pixels per instruction.

die area of $529mm^2$ in 40nm, compared to less than $1mm^2$ for the SIMD processor, the GPU clearly uses a very large number of resources to achieve this performance. In fact the simulation results suggest that the GPU is significantly less efficient compared to the embedded SIMD core, consuming thirty times higher energy for this convolution task.

5.4 Convolution Engine

Convolution Engine (CE) is a flexible domain-specific processor core which implements the generalized convolution abstractions described in Section 5.1. It builds on the ideas we developed in the last chapter. One of the kernels discussed in that chapter was 16x16 SAD calculation—a 2D convolution operation. There, a 2D shift register file with horizontal and vertical shift capability, was central to the efficient IME datapath targeting this computation. Since all 2D convolution operations have

the same basic flow, the heart of the CE is a similar 2D shift register, augmented with additional resources to support the generalized *map* and *reduce* operations.

In a manner similar to the 2D case, a 1D shift register file helps accelerate the 1D convolution operations. As shown in Figure 5.3, 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 just a small 256-bit register file saving valuable register file access energy as well as area. Thus, CE also incorporates a generalized version of this 1D shift register to handle 1D convolution flows.

The key blocks in Convolution Engine are depicted in Figure 5.4. As discussed, 1D and 2D shift registers form its core. Memory data is loaded to these register through load/store units. The interface units route data from these registers to the ALUs, with the routing pattern dependent on the size and type of convolution flow. The ALU array, performs the arithmetic operations for the map step. Finally the reduction tree performs the appropriate type of reduction. CE also includes a light-weight SIMD unit to perform further processing on convolution output when needed.

CE is developed as a domain specific hardware extension to Tensilica's extensible RISC cores [26]. Augmented with user-defined hardware interfaces called TIE ports, developed using Tensilica's TIE language [57], these RISC cores control the program flow of CE by first decoding the instructions in their instruction fetch unit and then routing the appropriate control signals to CE using TIE ports. Since the number of cores that interface with CE can be more than one, the TIE ports are muxed. The cores are also responsible for memory address generation, but the data is transferred directly to/from the register files within CE.

The next few sections describe in more detail the various blocks in CE. Note that at this step we have not assigned sizes to various resources shown in the block diagram. Section 5.4.5, which comes later in this chapter, contains a detailed discussion of how we determine the right resource sizes based on energy efficiency considerations as well as application performance needs.

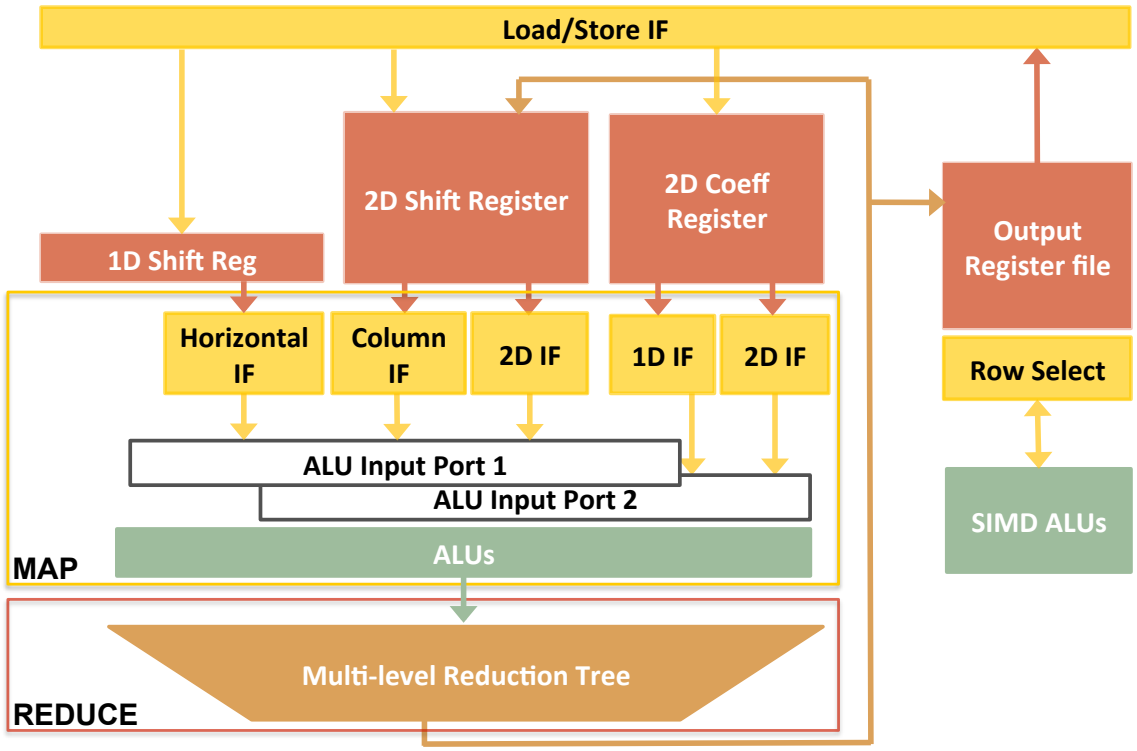


Figure 5.4: Block diagram of Convolution Engine. The interface units (IF) connect the register files to the functional units and provide shifted broadcast to facilitate convolution.

5.4.1 Load/Store Unit and Register Files

The load/store unit loads and stores data to and from the various register files. To improve efficiency, it supports multiple memory access widths with the maximum memory access width being 256-bits and can handle unaligned accesses.

The 1D shift register supplies data for horizontal convolution operations along image rows. New image pixels are shifted horizontally into the 1D register as the 1D stencil moves over an image row. 2D shift 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. A standard vector register file, due to its limited design, is incapable of providing all of this functionality.

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 resulting 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 5.4.

5.4.2 MAP & Reduce Logic

As described earlier we abstract out convolution as a map 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: Interface Units (IF) arrange data from the shift register 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 registers, 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 accesses 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.

Map 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 difference of absolute 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.

5.4.3 SIMD & Custom Functional Units

To enable an algorithm to perform 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.

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 Register Files. 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.

5.4.4 A 2-D Filter Example

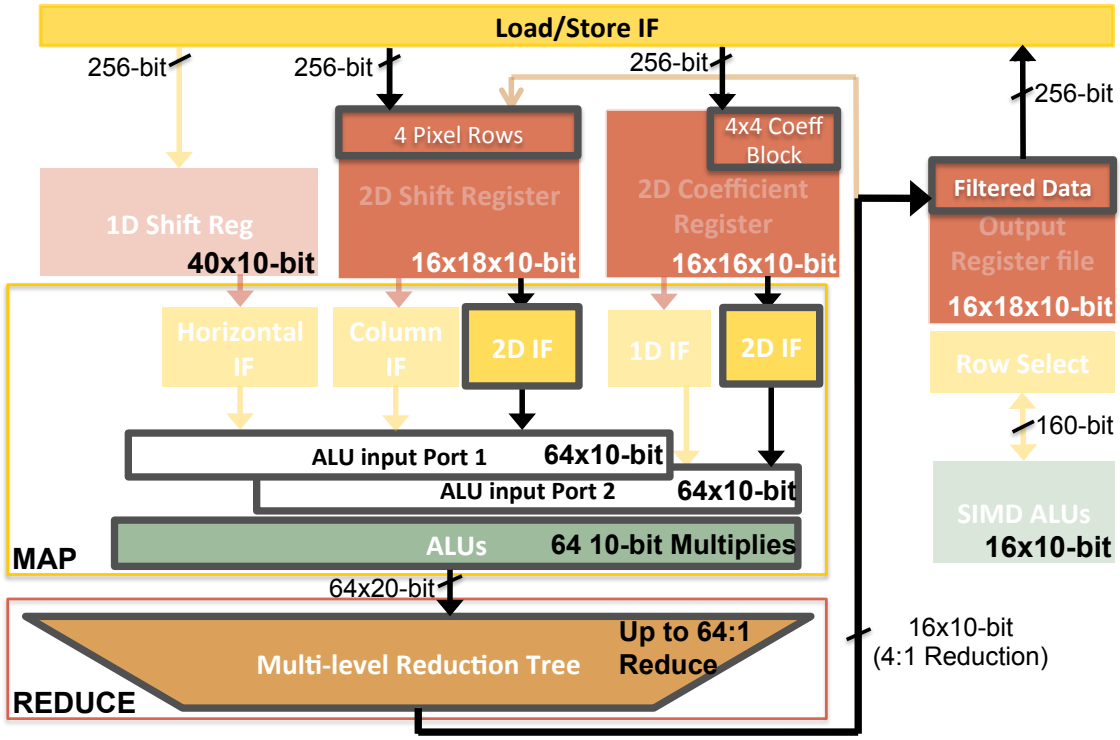


Figure 5.5: Executing a 4x4 2D filter on CE. The grayed out boxes represent units not used in the example. The sizes of all of the resources are defined. The of choice these particular resource sizes will be explained in a later section.

Figure 5.5 shows how a 4x4 2D filtering operation maps onto the convolution engine. Filter coefficients reside in the 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, and in this case it 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 the 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. With this support, the 1D 16-tap filtering of Figure 5.1 can exploit that to achieve even higher efficiency.

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.

5.4.5 Resource Sizing

Energy efficiency and performance requirements of target applications drive the sizes of various resources within CE. In the datapaths of the last chapter, overheads such as instruction fetch and decode were amortized by performing hundreds of arithmetic operations per instruction. However, while algorithms such as motion estimation employ extremely low-energy 8-bit addition/subtraction, filtering is typically dominated by multiplication that consumes higher energy per operation. To determine

Table 5.3: Sizes for various resources in CE.

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

how to size the ALUs for CE with the goal of keeping overheads as low as possible, we present the energy dissipated in executing filtering instructions using 32, 64 and 128 10-bit ALUs (the precision required) in Table 5.4. 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.

With 32 ALUs, a substantial percentage of energy is consumed by the overheads. As the number of ALUs increases, the overhead energy as a percentage of the total energy reduces. At 64 ALUs, the overhead is 12%, which is low but not negligible. As we go to 128 ALUs, the overhead percentage goes down slowly and we start moving towards diminishing returns. Moreover, keeping 128 or more ALUs busy is hard for smaller convolution kernels. As an example, for a 4x4 symmetric convolution kernel, 32 output pixels must be computed in a single instruction to keep 128 ALUs busy. To store enough data for computing that many output pixels in parallel, the 2D register must be wider than 32-elements. Thus, we choose 64 as the number of ALUs in CE as this achieves good efficiency without requiring very wide register resources. For larger kernels, requiring more ALUs to meet the performance goal, multiple CE instances can be chained together.

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 5.3. 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

Table 5.4: Energy for filtering instructions implemented as processor extensions with 32, 64 or 128 ALUs. Overhead is the energy for instruction fetch, decode and sequencing.

	32 ALUs	**64 ALUs**	**128 ALUs**
Total Energy (pJ)	156	313	544
Overhead Energy (pJ)	37	39	44
Percent Overhead	24	12	8

us to handle boundary conditions common in convolution operations efficiently.

5.4.6 Convolution Engine CMP

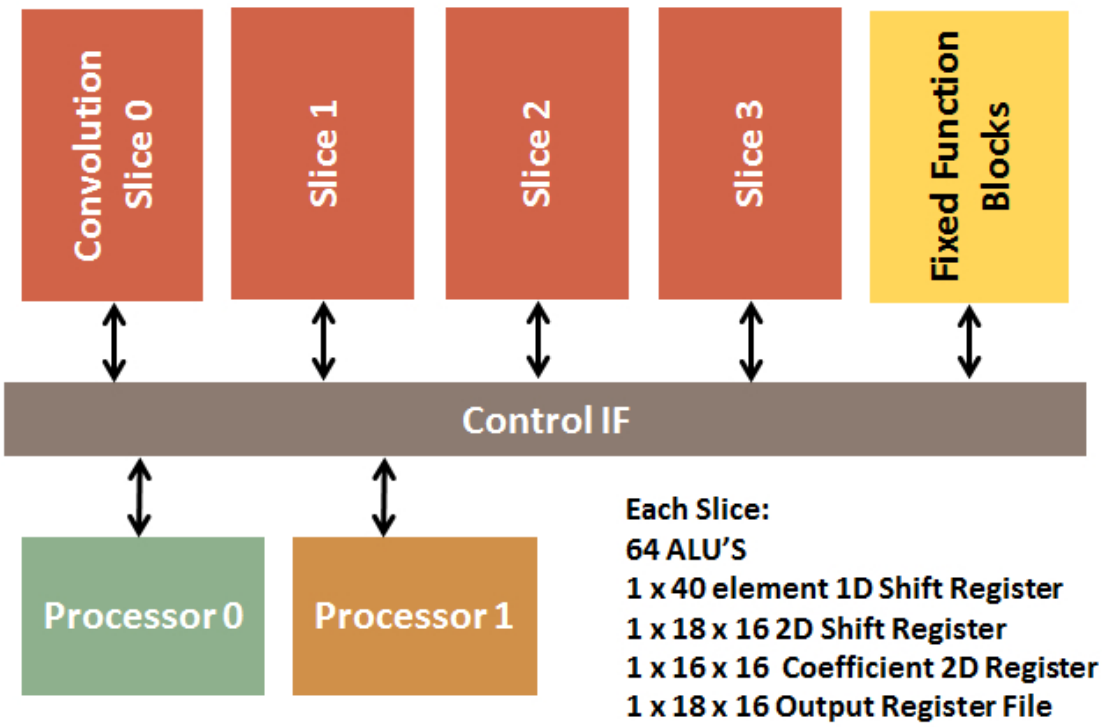


Figure 5.6: Convolution Engine CMP.

To meet the diverse performance and energy requirements of different applications effectively, we have developed a CE chip-multiprocessor (CMP) shown in Figure 5.6.

The CMP consists of four CEs and two of 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 operate completely independent of other slices and also 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, these wires waste very little energy; therefore, not influencing 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. The processors and the 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.

5.4.7 Programming the Convolution Engine

Convolution Engine is implemented as a processor extension and adds a small set of instructions to processor ISA. These CE instructions can be issued as needed in regular C code through compiler intrinsics. Table 5.5 lists the major instructions that CE adds to the ISA and Listing 5.1 presents a simplified example code, which implements 15-tap horizontal filtering for a single image row. There are mainly 3

Table 5.5: Major instructions added to processor ISA.

	Description
SET_CE_OPS	Set arithmetic functions for MAP and REDUCE steps
SET_CE_OPSIZE	Set convolution size
LD_COEFF_REG_n	Load n bits to specified row of 2D coeff register
LD_1D_REG_n	Load n bits to 1D shift register. Optional shift left
LD_2D_REG_n	Load n bits to top row of 2D shift register. Optional shift row down
ST_OUT_REG_n	Store top row of 2D output register to memory
CONVOLVE_1D_HOR	1D convolution step - input from 1D shift register
CONVOLVE_1D_VER	1D convolution step - column access to 2D shift register
CONVOLVE_2D	2D Convolution step with 2D access to 2D shift register

types of instructions. Configuration instructions set options that are expected to stay fixed for a kernel such as convolution size, ALU operation to use etc. Other options that can change on a per instruction basis are specified as instruction operands. Then there are load and store operations to 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). Finally there are the compute instructions, one for each of the 3 supported convolution flows—1D horizontal, 1D vertical and 2D. For example the CONVOLV_2D instruction reads one set of values from 2D and coefficient registers, performs the convolution and writes the result into row 0 of the 2D output register. The load, store and compute instructions are issued repeatedly as needed to implement the required algorithm.

The code example in Listing 5.1 brings it all together. First, CE is set to perform multiplication at Map stage and addition at Reduce stage, as these are the required settings for filtering. Then 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.

```
// 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);

   // 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 5.1: Example C code implements a 15-tap filter for one image row and adds 2 to each output.

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 giving 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.

5.4.8 Controlling Flexibility in CE

The programmable convolution engine as described has full flexibility in silicon to perform any of the supported operations, and the desired operation is selected at run time through a combination of configuration registers and instruction operands. However, we also want to study the individual impact of various programmability options present in CE. To facilitate that, 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 configurations such as the one shown in Figure 5.3, to the fully programmable instance discussed in previous sections. When the fixed kernel instance of Figure 5.3 is generated, 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 matches the custom cores. The programmability options that convolution engine has over this fixed kernel datapath can be grouped into three classes that 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 the 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).

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.

5.5 Evaluation Methodology

To evaluate the Convolution Engine approach, we map each target application onto a CE based CMP described in Section 5.4.6. As already discussed this system is fairly flexible and can easily accommodate algorithmic changes such as changes in motion estimation block size, changes in down-sampling technique etc. Moreover, it can be used for other related algorithms such as a different feature detection scheme like SURF, or other common operations like sharpening or denoising etc.

To quantify the performance and energy cost such a programmable unit, we also build custom heterogeneous chip-multiprocessors (CMPs) 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 [57]. 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 multiprocessor 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

Figure 5.7: Mapping of applications to Convolution Engine CMP.

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 45nm technology at 0.9V operating voltage with a target frequency of 800MHz. All units are pipelined appropriately to achieve the frequency target.

To further extend the analysis, we quantify the individual cost of different programmability options discussed in Section 5.4.8. Starting from a fixed kernel datapath closely matching the custom hardware, we add the programmability options in steps. That way we can identify the incremental cost of each programmability class and understand if some types of programmability options are costlier than others.

Figure 5.7 presents how each application is mapped to our CE based CMP. This mapping is influenced by the application's performance requirements. In this study, like most video systems these days, we support HD 1080P video at 30FPS. This translates to an input data rate of around 60 MPixels/s. For still images we want to support a similar data rate of around 80-100 MPixels/s, which can be translated for example to processing 10MP images at 8-10FPS or 5MP images at a higher rate of 16-20FPS etc. H.264 motion estimation only deals with video data, whereas SIFT can be applied to both video and still images. Now, we describe the mapping in detail for each application.

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.

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.

5.6 Results

Figures 5.8 and 5.9 compare the performance and energy dissipation of the proposed Convolution Engine against a 128-bit datap-arallel (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. For SIMD implementation of motion estimation, one SIMD engine performs IME and another performs FME. Similarly for SIFT, one SIMD engine handles DOG, and the other handles extrema detection.

The fixed function data points truly highlight the power of customization: for

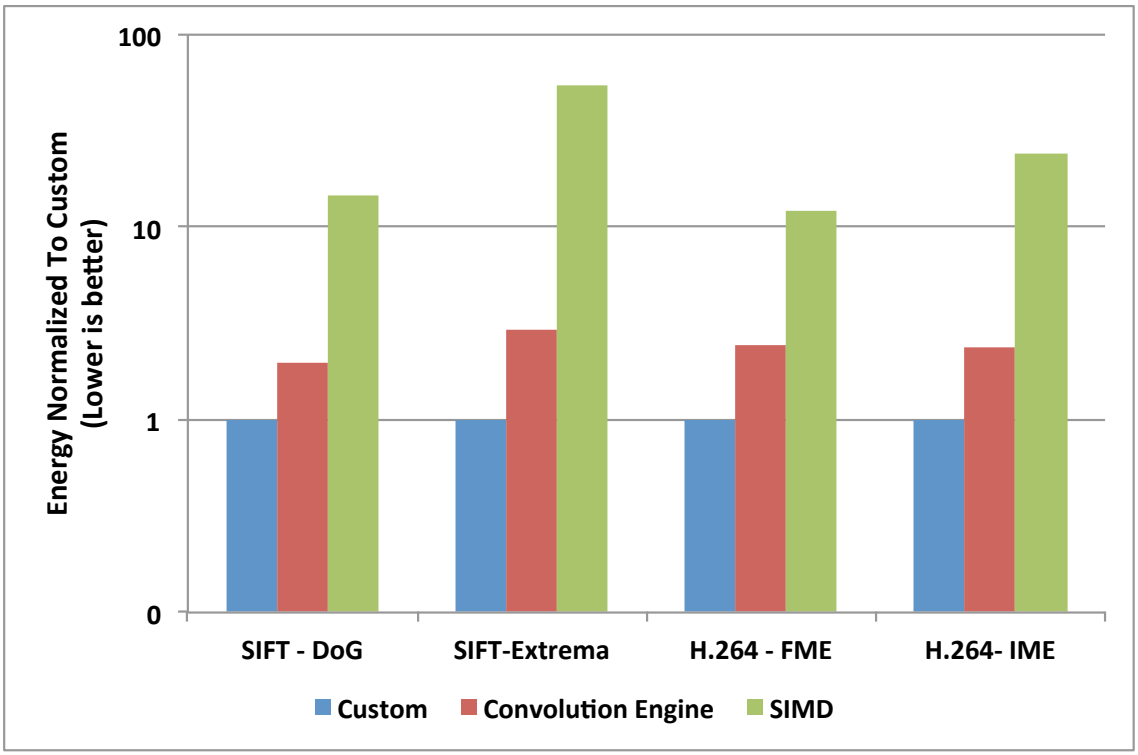


Figure 5.8: Energy consumption normalized to custom implementation: Convolution Engine vs. custom cores and SIMD.

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.

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

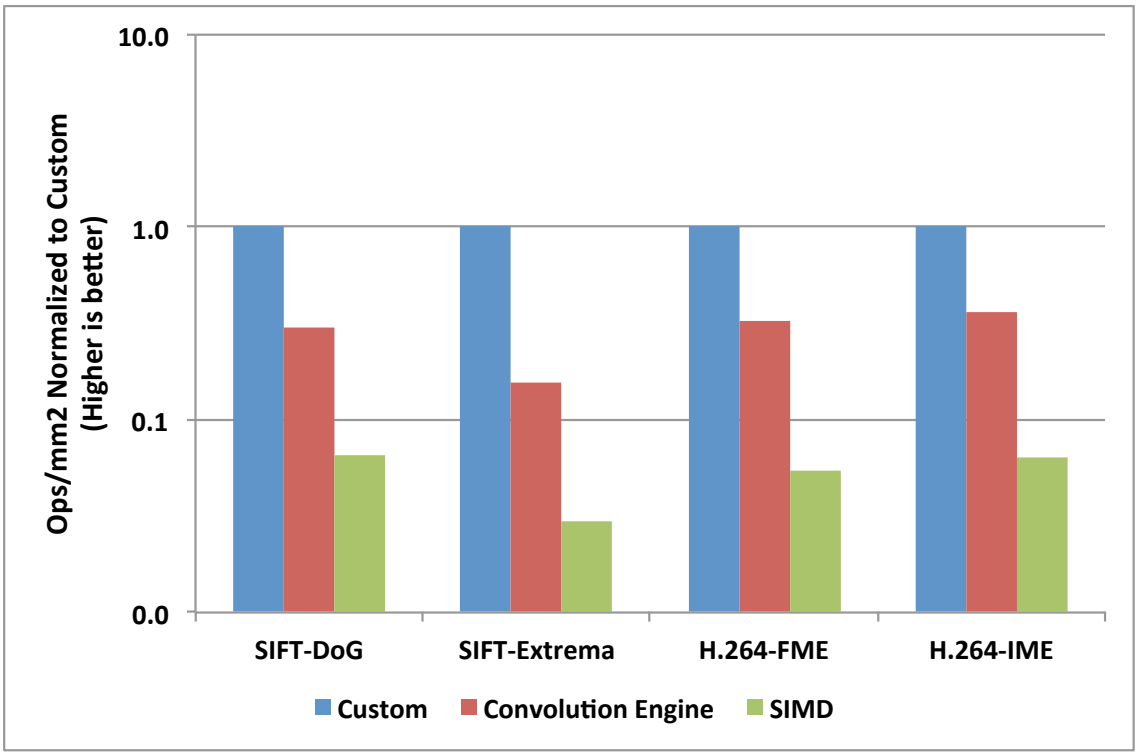


Figure 5.9: Ops/mm2 normalized to custom implementation: Number of image blocks each core processes in one second, divided by the area of the core. For H.264 an image block is a 16x16 macroblock and for SIFT it is a 64x64 image block.

larger number arithmetic units to work in parallel, better amortizing instruction and data fetch.

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.

The energy overhead of the CE implementation over an application specific accelerator is modest (2-3) for the other applications, and requires only twice the area. While these overheads are small, we want to understand which of the programmability option discussed in Section 5.4.8 contribute the most to this overhead. To generate these results, for each convolution algorithm we start with an accelerator

specialized to do only the specific convolution kernels required for that algorithm, and then gradually add flexibility. These results are shown in Figures 5.10 and 5.11.

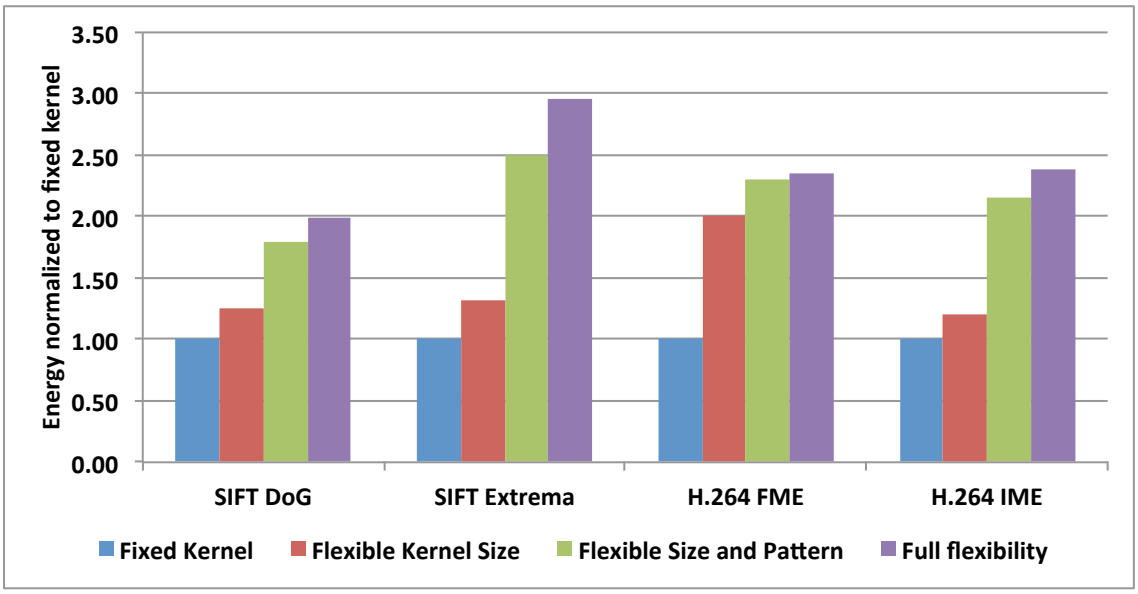


Figure 5.10: Change in energy consumption as programmability is incrementally added to the core.

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 and consequently the leakage energy increases as well. The final step of adding multiple compute units has a relatively negligible impact of 10%.

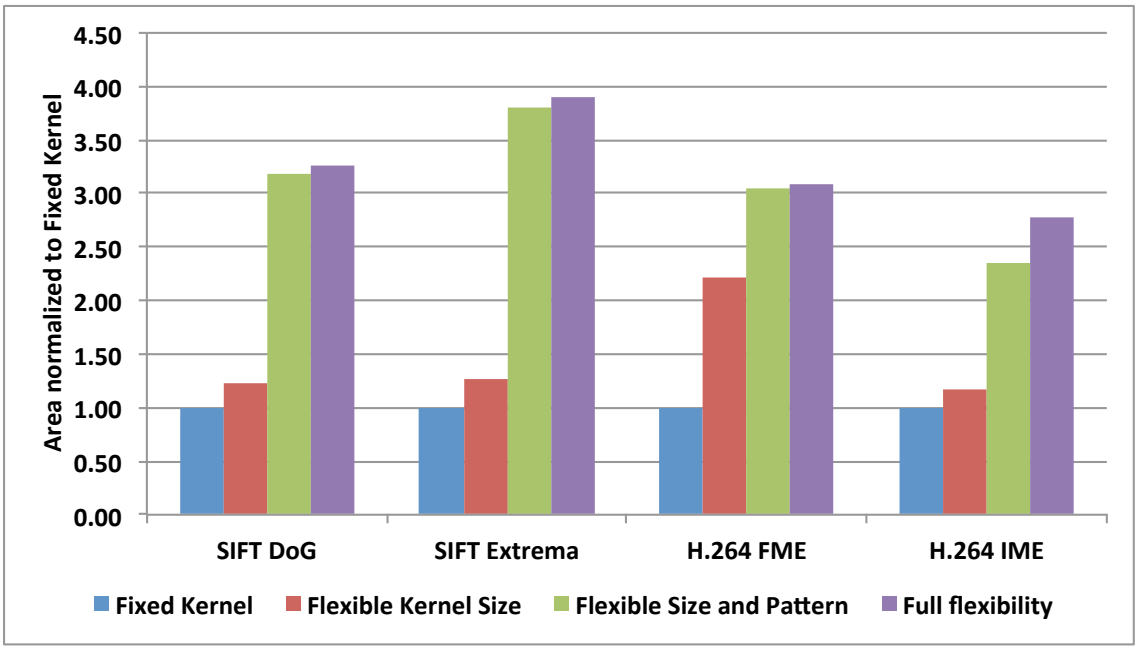


Figure 5.11: Change in area as programmability is incrementally added to the core.

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 the 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, which is fairly significant at such small feature sizes. 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, flexible functional units provide flexibility at low cost.

5.7 Convolution Engine 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 accelerates 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. While the CE is a single example, we hope that similar studies in other application domains will lead to other efficient, programmable, specialized accelerators.

Chapter 6

Bilateral Filtering

The algorithms that we have considered so far have modest working sets, which fit in a relatively small storage structure. However once we move to algorithms with much larger working sets—say a 128x128 convolution kernel—it is no longer feasible to store the entire set into local registers. The size of the buffer required would be big enough to not provide much energy advantage over accessing the data from the memory and at the same time area and leakage would become very high. So even though there is an even larger degree of parallelism and data-reuse present in these larger kernels, we can't exploit that using the same techniques that we have used so far.

For linear convolution operations, a few different approaches exist to reduce the computation and data requirements of large kernels [33]. These include techniques such as converting the convolution kernel into a recursive filter [54]; implementing convolution by first converting the image and kernel into frequency domain where convolution becomes a multiplication [54]; and tiling methods such as overlap-save and overlap-add [9], which break the kernel and/or the image into smaller pieces that are computed independently and then merged together. For blur kernels such as Gaussian filters, which effectively reduce the spatial resolution of the image, another effective technique is to first decimate the image to reduce its dimensions, apply a smaller blur kernel on that smaller image, and then up-sample back to the original size. Moreover some convolution kernels such as the Gaussian kernels are separable

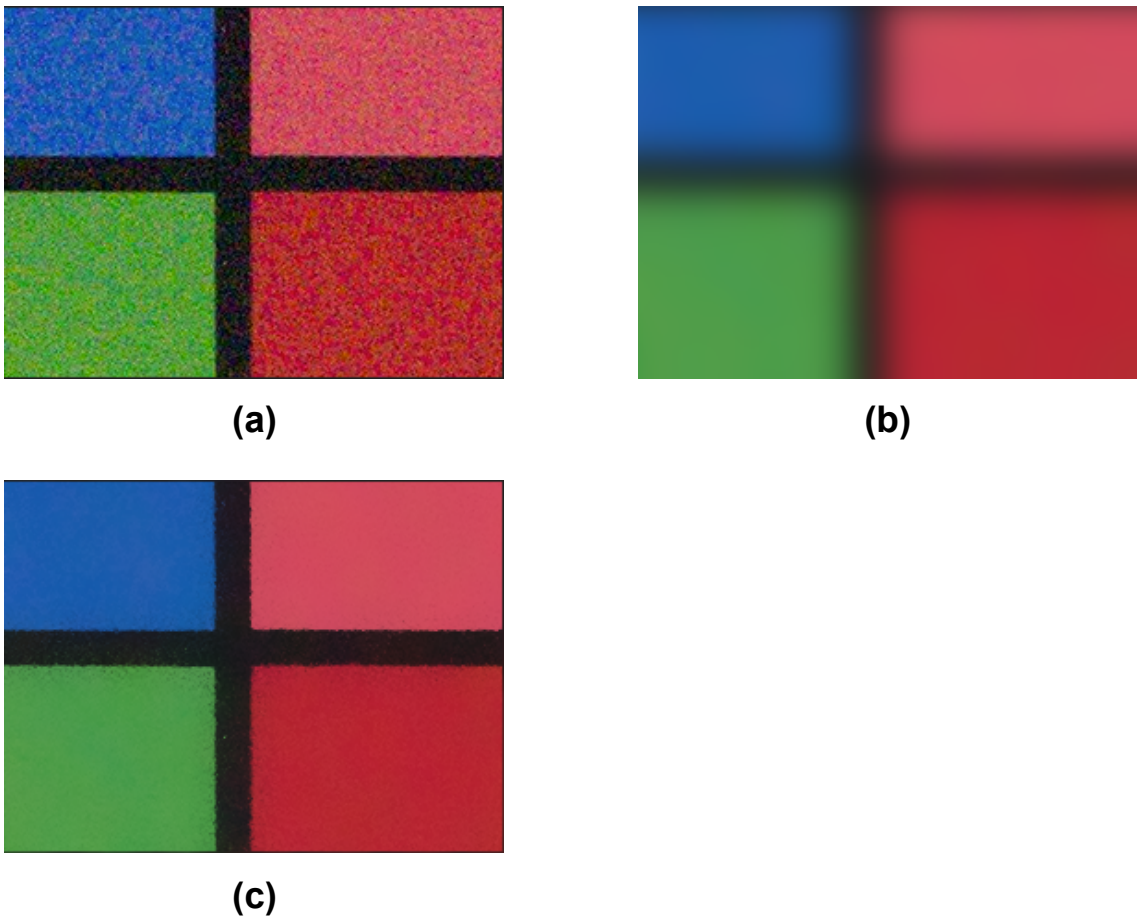


Figure 6.1: Smoothing using a standard Gaussian blur kernel vs. a bilateral filter. (a) Original image (b) Gaussian blur (c) Bilateral filter.

and thus can be applied as a horizontal 1D convolution along image rows, followed by a vertical 1D convolution along image columns. All of these techniques can bring the problem size back to a level where it can be efficiently attacked without being limited by the memory accesses. However not all stencil operations can be simplified that way, and alternate strategies are needed to handle those. In this chapter we discuss efficient hardware implementation of one such algorithm called bilateral filtering, which is a non-linear filter used for image smoothing.

6.1 Bilateral Filtering

A common task in image processing is to blur an image. One reason to blur an image is to reduce photographic noise introduced by the camera. Another typical reason is to separate out the base and detail layers of the image, which can then be further processed independently [11]. The simplest method of blurring an image involves convolving the image with a smoothing kernel such as a Gaussian Blur kernel. That type of a convolution kernel can be easily implemented using the convolution engine we discussed in last chapter. The problem with that approach, as shown by Figure 6.1, is that while it smoothes out the noise, it also blurs away the edges in the image. Bilateral filtering is an edge-aware smoothing technique, which performs piece-wise smoothing of the image without blurring across the edges [58]. Figure 6.1 (c) illustrates that for our example image.

6.1.1 Gaussian Blur

The Gaussian blur of Figure 6.1, replaces each input pixel by a weighted average of all the pixels in the neighborhood of that pixel. Equation 6.1 gives Gaussian blur operation for grayscale images. $\vec{x}_i$ is the position vector for a pixel in the image at row y_i and column x_i. $\hat{I}(\vec{x}_i)$ is the intensity of the output pixel at $\vec{x}_i$. j represents the set of input pixels in the neighborhood over which the weighted average is computed. Intensity $I(\vec{x}_j)$ of each input pixel in the neighborhood is weighted by a factor that depends on the vector distance between $\vec{x}_j$ and $\vec{x}_i$. The exponential weight quickly falls to zero as the distance increases. Thus only the pixels in the close vicinity of $\vec{x}_i$ have an impact on the output and neighborhood j just needs to be large enough to cover these pixels. Note that the equation as given describes a Gaussian kernel with standard deviation = 1. Increasing or decreasing the standard deviation controls how slowly or rapidly the Gaussian falls off, and consequently the size of neighborhood to be included in the computation of the blur. Equation 6.1 can be modified to incorporate the standard deviation term. However scaling the position vectors by an appropriate factor and applying Equation 6.1 achieves the same result.

Equation 6.2 gives the corresponding equation for RGB images. Instead of scalar

intensities, each pixel value is now represented as an RGB vector $\vec{v} = (r, g, b)$.

$$\hat{I}(\vec{x}_i) = \sum_j I(\vec{x_j}) \cdot e^{-|\vec{x}_i - \vec{x}_j|^2/2} \tag{6.1}$$

$$\hat{\vec{v}}(\vec{x}_i) = \sum_j \vec{v}(\vec{x}_j) \cdot e^{-|\vec{x}_i - \vec{x}_j|^2/2} \tag{6.2}$$

Gaussian blur works on the assumption that neighbors in the vicinity of a pixel are similar to that pixel, and thus the weighted sum accumulates the contributions from these similar pixels. The size of blur kernels is typically small (5x5 or 7x5 are common) to help ensure this assumption holds. However the assumption breaks down close to edge boundaries. At these boundaries, dissimilar pixels across the edge also contribute to the weighted average thus blurring the edge. Bilateral filtering solves this problem as described next.

6.1.2 Bilateral Blur

Equation 6.3 shows the bilateral filtering operations for grayscale images. The weight applied to intensity $I(\vec{x}_j)$ now depends on not just the x,y distance $|\vec{x}_i - \vec{x}_j|$ but also on the intensity difference $I(\vec{x}_i) - I(\vec{x}_j)$. The additional Gaussian term falls off with the difference in intensity. Equation 6.4 gives the equivalent bilateral filtering operation for color RGB images, where the additional Gaussian term falls off with the vector distance between RGB values i.e. $|\vec{v}(\vec{x}_i) - \vec{v}(\vec{x}_j)|$. As a result of these additional Gaussian terms, neighboring pixels with intensities or RGB values very different from the current pixel, get a low weight. Thus dissimilar pixels across the edge do not contribute significantly to the weighted sum, avoiding the blurring across the edge.

$$\hat{I}(\vec{x}_i) = \sum_j I(\vec{x_j}) \cdot e^{-|\vec{x_i} - \vec{x_j}|^2/2} \cdot e^{-(I(\vec{x}_i) - I(\vec{x}_j))^2/2} \tag{6.3}$$

$$\hat{\vec{v}}(\vec{x}_i) = \sum_j \vec{v}(\vec{x}_j) \cdot e^{-|\vec{x}_i - \vec{x}_j|^2/2} \cdot e^{-|\vec{v}(\vec{x}_i) - \vec{v}(\vec{x}_j)|^2/2} \tag{6.4}$$

Bilateral filtering is used widely in computational photography, computer vision and medical imaging. A typical use is to decompose an image into a smoothed-out base layer as well as a detail layer and then manipulate these independently before combining them back to produce the output image [11]. Applications include sharpening [11], high dynamic range imaging [23], video de-noising [13], optical flow regularization [60] and texture removal [43] to name a few. In the next section we discuss existing techniques for accelerating this operation, and their limitations.

6.2 Bilateral Filtering - Existing Implementations

With its ability to reject contributions from dissimilar pixels, bilateral filter no longer relies on spatial closeness to find similar pixels. Instead it can use larger stencils so a pixel can get contributions from similar pixels farther away in the image. Thus large stencil sizes such as 100x100 are common. As a result, a direct application of Equation 6.4, would be prohibitively costly. Also while bilateral filter operations have a structure similar to the Gaussian blur, these are non-linear filters as the weights now depend on not just the position but also the pixel values. As a result the usual techniques for dealing with large linear convolution operations, identified earlier in this chapter, do not apply.

Techniques such as *bilateral grid* [15], as well as constant time O(1) bilateral filtering approaches presented in [46] and [61] provide efficient means to handle the grayscale case. A low-power real-time hardware implementation of bilateral grid scheme has also been presented in [50]. However these techniques do not scale well to RGB bilateral filtering [5]. In our work we have concentrated on RGB case for a couple of reasons. First, unlike the grayscale case, no satisfactory low power implementations exist for the RGB case. Moreover, it provides a greater challenge for the type of efficient hardware abstractions we have defined so far.

The state-of-the art software implementation for RGB images is a permutohedral lattice based approach presented in [5]. This technique, as well as others such as Gaussian KD-Tree based approach [6] and bilateral grid approach, convert bilateral filtering into higher dimensional linear filtering. As [44] shows, bilateral filtering

operations of Equations 6.3 and 6.4 can be converted into a higher dimensional linear filtering operation by expressing them as a Gauss Transform operation given in Equation 6.5:

$$\vec{\hat{v}}(\vec{p_i}) = \sum_j \vec{v}(\vec{p_j}) \cdot e^{-|\vec{p_i} - \vec{p_j}|^2/2} \tag{6.5}$$

Vectors $\vec{p_i}$ and $\vec{p_j}$ are now *positions* in a higher dimensional space instead of 2D (x, y) locations. For the grayscale bilateral filtering, these are of the form (x, y, I), giving a 3D space. For RGB case, position vectors are of the form (x, y, r, g, b) i.e. a 5D space. Note that the input and output values here are defined over a 3D (grayscale) or 5D (RGB) space. Thus the image pixels first have to be populated at appropriate points in the 3D or 5D space before the higher dimensional linear Gaussian blur kernel is applied to blur the whole space. Moreover as [44] shows, for this linearization strategy to work, the input and output values use a homogenous representation. So in case of RGB images, the input values are represented as (r, g, b, 1). The output values of Equation 6.5 then have the form (r, g, b, w), where w gives the weighting factor. The final RGB output value is computed as (r/w, g/w, b/w).

Gauss transform of Equation 6.5 applies a high dimensional Gaussian blur to a high dimensional space. Thus the computational requirements of directly implementing this operation would be even higher than the cost of Equations 6.3 and 6.4. However since the blur kernel is linear, some of the techniques that we discussed earlier for dealing with large convolutions can now be applied. We discuss two such schemes - bilateral grid and permutohedral lattice - as our work combines ideas from these two.

6.2.1 Bilateral Grid

Bilateral grid represents the higher dimensional space of equation 6.5 using a coarse decimated grid [15]. For the grayscale bilateral filtering case discussed in [15], this is a 3D grid decimated along each of x, y and I axes. Each dimension in the space is decimated by a factor equal to the standard deviation σ of the Gaussian in that

Table 6.1: Execution time and storage requirements of grayscale vs. RGB bilateral filtering schemes for an HD video frame. Decimation factor is 16 in all dimensions. Entries column refers to the array or hash-table entries required to store the points in high dimensional space. Bytes column refers to the size of storage structure in bytes. Direct algorithm is direct application of Equation 6.3.

	Space	**Entries**	**Bytes**	**Time**
Direct - Grayscale	2D	-	-	6m 49s
Bilateral Grid - Grayscale	3D	142K	1.1MB	0.5s
Bilateral Grid - RGB	5D	41M	531MB	11s
Permutohedral Lattice - RGB	5D	1M	22MB	2.6s
Modified Bilateral Grid - RGB	5D	90K	1.5MB	1s

dimension. The algorithm has 3 steps called splat, blur and slice. Splat projects each image pixel to appropriate grid locations based on its position (x, y) and intensity I. The blur step blurs all the locations in the 3D grid using a 3D blur kernel. Finally slice step computes the pixels in output 2D image by sampling appropriate positions in the blurred 3D grid. Figure 6.2 reproduced from [8] illustrates this process for a 1D grayscale image.

The cost of the blur step goes down dramatically in this scheme. Due to aggressive decimation the size of the grid is fairly small. As Table 6.1 shows, for a typical decimation factor of 16 in spatial and intensity dimensions, the 3D grid corresponding to a HD video frame only has about 129K entries, requiring a modest 1.1MB of storage memory. Moreover only a small 3x3x3 or 5x5x5 blur kernel is needed in this decimated space. Finally the Gaussian blur kernel is separable so the 3D blur can be implemented as a series of 3 small linear blurs, one along each of the x, y and I dimensions. As Table 6.1 shows, the processing time of this algorithm for an HD video frame is only 0.5s second compared to around 7 minutes required for direct application of equation 6.3!

However this approach does not scale well to the 5D RGB case as the size of grid scales exponentially with dimensions. Bilateral grid for an RGB HD video frame contains 41 million points using about 656MB of storage. The time to blur this grid also gets a corresponding exponential increase and the overall execution time goes up from 0.5s per frame for the grayscale case to 11s for the RGB case.

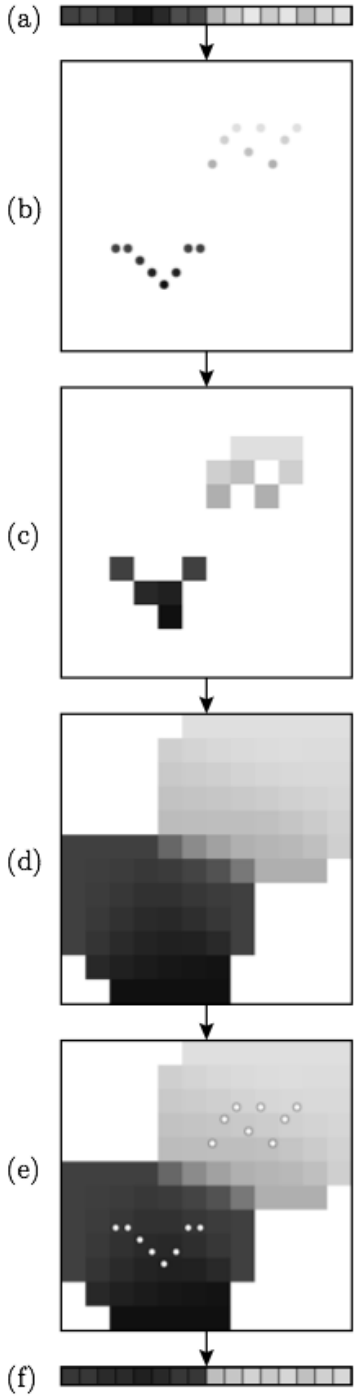


Figure 6.2: The splat, blur and slice steps used in bilateral grid algorithm - reproduced from [8]. (a) Input "image" is a 1D grayscale signal. (b) Each pixel in the image is treated as a point in the 2D (x, I) space with a 2D position vector. (c) Each pixel is assigned to the coarse grid square it falls in. The size of the square depends on the decimation factor. (d) The entire 2D space is blurred using a 2D blur kernel. The lighter pixels group is at a distance from the darker pixel group so no significant mixing of the two occurs. (e) For slicing the blurred 2D space is sampled at positions corresponding to the original image pixels. (f) Output pixels are now blurred without crossing the edges.

6.2.2 Permutohedral Lattice

Note from previous section that the number of grid locations in the 5D case far exceeds the number of pixels in the image, and thus the space is only sparsely populated. Thus storage requirements as well as blur time can be reduced using a sparse representation of populated points in the space. However to achieve good quality, when a pixel is splatted to the grid it distributes some of its weight to each of the 2^d vertices of the d-dimensional hyper-cube that it falls in. Figure 6.4 illustrates that for a 2D space. Thus the populated points in space, while sparse, are still exponential in dimensions, and so is the cost of splat, slice and blur operations.

To solve that issue, permutohedral lattice partitions the higher dimensional space into uniform simplices instead of the hyper-cubes. A uniform simplex is the generalization of a uniform triangle and a d-dimensional simplex thus has d+1 vertices. In the permutohedral lattice approach, each image pixel is splatted to the d+1 vertices of the uniform simplex enclosing it. Thus the complexity of splat step becomes linear in number of dimensions. Consequently the number of populated points in the space also become linear in d, and are sparsely stored using a hash-table. The execution of blur step is proportional to number of populated points in space and thus no longer scales exponentially with dimensions. Finally the slice step is also linear in d. As Table 6.1 shows, for an RGB image, permutohedral lattice algorithm requires much less storage and is significantly faster compared to the bilateral grid.

This algorithm represents the state of the art in fast bilateral filtering for RGB images. However from an energy consumption perspective it is not very efficient. Most of the operations in this algorithm operate on the hash-table structure, which is far too big to fit in the on-chip memory and has to reside in the DDR. Figure 6.3 shows the energy breakdown for an implementation of this algorithm on 32-bit RISC platform based on our base Tensilica core. About 40% of the dynamic energy consumption goes into DDR accesses. That means that unless these DRR accesses could be eliminated, accelerating the compute could not reduce dynamic energy consumption by more than 2x. Leakage also accounts for about 33% of energy consumption. However leakage energy scales linearly with execution time, so making the computation faster would cause this component of energy to go down.

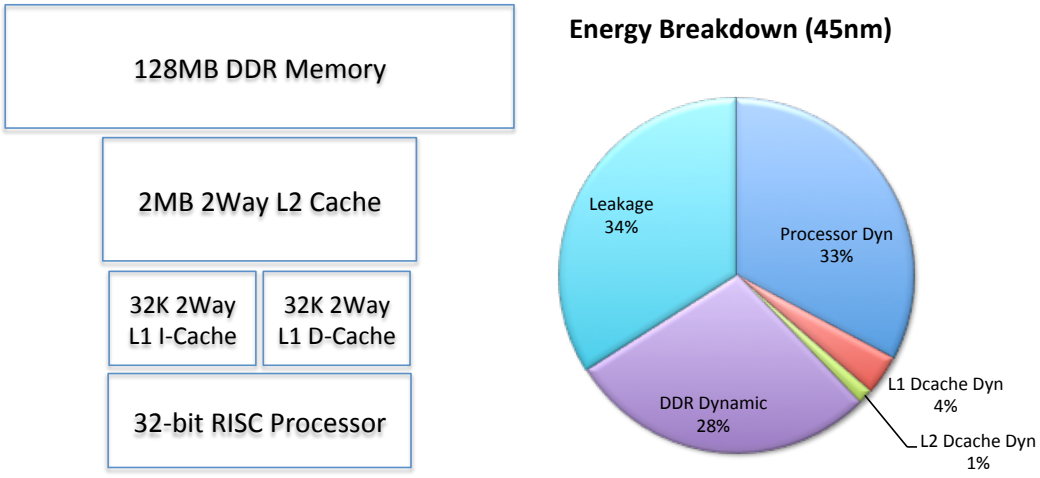


Figure 6.3: Permutohedral lattice based implementation of bilateral filtering on a Tensilica based 32-bit RISC platform. The system includes 32KB L1 instruction and data caches, as well as a 2MB L2 cache. DDR accesses account for more than 40% of dynamic energy dissipation. Leakage energy is also very significant and mostly comprises the leakage from the large L2 cache.

6.3 Extracting Locality - Modified Bilateral Grid

We have created a modified bilateral filtering algorithm that has a much smaller memory foot print eliminating the DDR accesses and offers locality both at L1 and L2 cache level. To enable these memory optimizations, it uses a grid structure. However unlike the regular bilateral grid, each pixel in our proposed algorithm updates only d+1 vertices in the grid. These include the closest vertex of the hyper-cube enclosing that pixel, as well as d neighbors of that vertex, one along each dimension. Figure 6.4 illustrates for a 2D space and compares that to the regular bilateral grid.

To understand the motivation for this change, we discuss in a bit more detail how blurring works in these high-dimensional bilateral filtering approaches. In all these algorithms, grid or lattice vertices serve as stores for "pixel energy", accumulating energy from nearby pixels. Blurring the grid locations then spreads this energy to farther regions of the image. As a result the output pixels sliced from these blurred vertices get contributions from a large spatial neighborhood. To ensure uniform spread in all dimensions, the regular bilateral grid algorithm splats a pixel to all 2^d

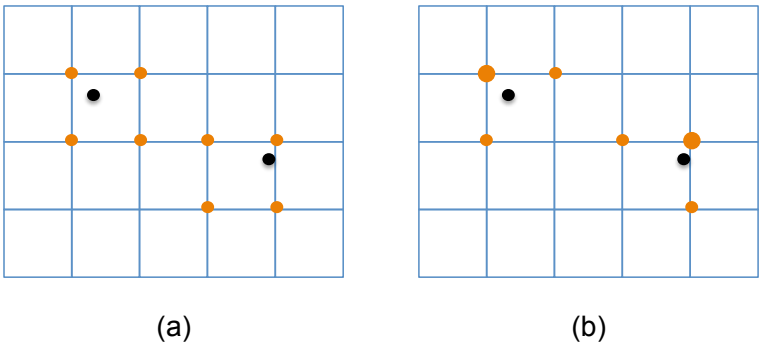


Figure 6.4: Splatting in regular bilateral grid and modified grid schemes: (a) In regular bilateral grid algorithm each pixel imparts its energy to all the 2^d vertices of the d-dimensional hyper-cube that it falls in. In this 2D illustration, each pixel updates all 4 vertices of its enclosing square. (b) In the modified grid scheme, each pixel updates its closest grid vertex as well as the d neighbors - one along each dimension.

vertices of the hyper-cube. Permutohedral lattice construction uses a smaller number of vertex updates to achieve an approximately uniform energy spread, with only a small SNR loss [5].

Our modified bilateral grid scheme similarly spreads the energy in an approximately uniform manner using only $d + 1$ vertices instead of 2^d, and incurs a comparable SNR loss. Table 6.2 compares the SNR achieved by this scheme with others including regular bilateral grid, permutohedral lattice and Gaussian KD-Tree. We also show results for a version of bilateral grid where only the nearest vertex in the grid is updated. As shown this modified grid construction fairly closely matches the characteristics of the permutohedral lattice implementation. However the use of a regular grid structure now enables data-locality optimizations that are hard to do with the permutohedral implementation.

As Figure 6.5 shows, instead of representing the entire 5D space as a single global hash table, we now represent it as a dense XY grid of small hash tables. Each of these hash tables stores the sparse RGB-sub space corresponding to a particular x, y location. The XY grid is just a decimated version of the original image x, y coordinates. Typical decimation factors for x and y dimensions are generally 16 or 32 and correspond to the standard deviation of the Gaussian blur kernels in x and y.

Table 6.2: Quality comparison of various schemes for RGB bilateral filtering of an HD video frame. The metric for output quality is the PSNR relative to an exact solution of equation 6.4. Our *modified bilateral grid* performs similar to *permutohedral lattice*.

Algorithm	**PSNR**
Bilateral Grid	52
Permutohedral Lattice	48.7
Modified Bilateral Grid	47.9
Gaussian KD-Tree	41
Nearest Neighbor Bilateral Grid	40.2

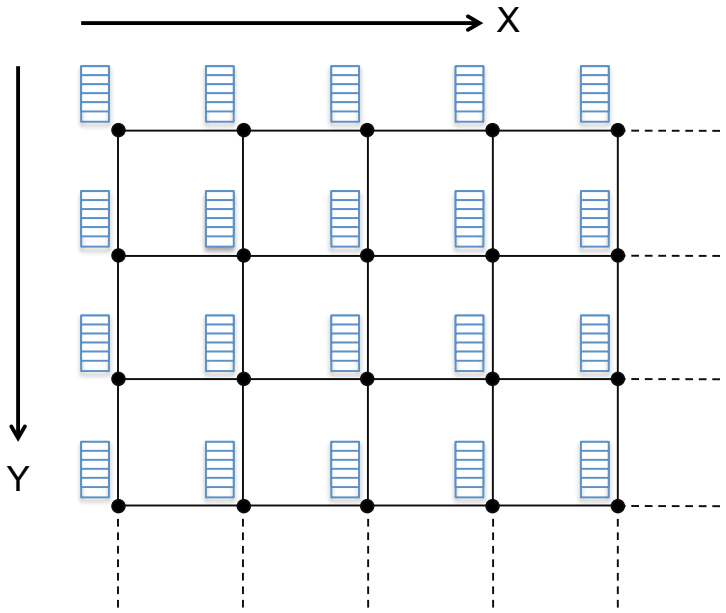


Figure 6.5: Modified Bilateral Grid uses an XY-Grid of hash tables to represent the decimated space (5D for RGB case). The X-Y grid represents the points along X, Y axes of the 5D space, and the hash table at each grid location stores the sparsely populated RGB sub-space corresponding to that X, Y co-ordinate.

Assuming a spatial decimation factor of 32, every grid "square", encapsulates a 32x32 region of the original image. Figure 6.6 highlights one such square. Every pixel in this square would update 3 of the 4 hash tables - the one residing on its closest x, y vertex and those residing on the two neighbors of that vertex along x and y. For a variety of images ranging from 1MP to 10MP, the maximum size of individual hash tables was found to remain within a few KBytes each, and thus an L1 data cache

can fit all four hash-tables at once. Therefore we perform splatting and slicing in a block-by-block fashion resulting in high temporal locality in hash table accesses.

Moreover, we no longer need to splat the whole image before starting the blur step. Blur across R, G and B dimensions can be performed locally within each hash-table, right after all the splats to that hash table are complete. Similarly, assuming we process the 32x32 blocks in horizontal scan-line order, blur across X can be performed once three 32x32 blocks have been processed. That also implies that to fully exploit temporal locality, our L1 D-Cache should be big enough to hold at least eight hash-table structures corresponding to three 32x32 image blocks. The Y-Blur however is not possible until three rows of image blocks have been processed. For a HD video frame using a block size of 32x32, this requires storing about 250 of these hash tables before vertical blur can be performed. Given that each hash-table could be up to a few KBytes, a 1-to-2 MB L2 cache can store these without the need to ever write the hash-table entries to DRAM.

Figure 6.7 compares the performance of this algorithm against the permutohedral lattice algorithm. Dynamic energy contribution from DDR goes to almost zero in our modified scheme. The execution goes down by about 3, partly because no complex lattice calculations are needed and at the same time no cycles are wasted waiting for the data from DDR. As expected, leakage energy goes down by the same factor as execution time. With DDR energy no longer being the bottleneck, it finally makes sense to accelerate the compute using the sort of techniques we have previously employed.

6.4 Hardware Acceleration of Modified Bilateral Grid

Since most operations in the modified bilateral grid work on (r, g, b, w) pixel values, these can be naturally expressed as 4-way SIMD operations. Of course we are looking for a much larger degree of parallelism both to improve performance and to amortize the instruction overheads. To achieve that extra degree of parallelism, we process multiple pixels in parallel. Thus if we use a 64-wide compute array as we did for

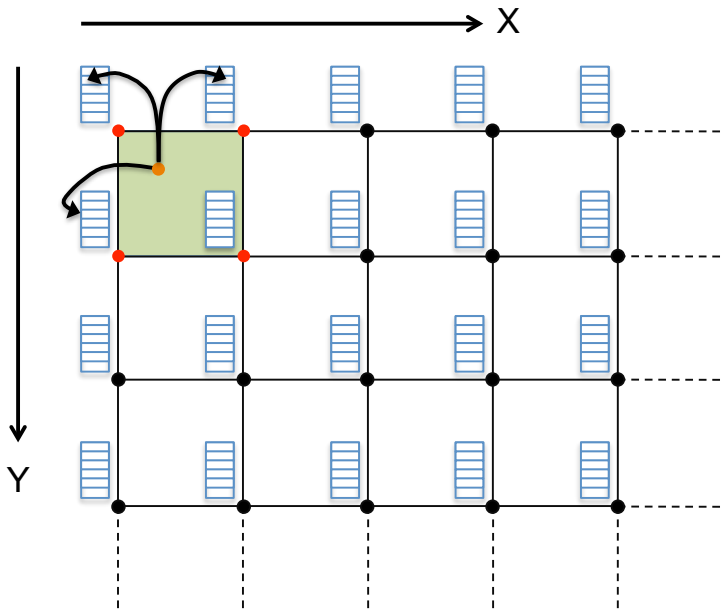


Figure 6.6: Hash table updates in modified grid: X, Y co-ordinates of the grid correspond directly to the X, Y pixels locations in the image. Assuming a decimation factor of 32 in X, Y, the green highlighted square in X, Y grid corresponds to the first 32x32 block in the image. The highlighted pixel in that 32x32 block updates the hash-table on its nearest XY vertex, as well as the neighboring hash-tables along X and Y.

convolution engine, we could process 16 pixels in parallel. This is however complicated by the accesses to the hash tables. Each step of the algorithm involves accesses to the hash table. Splat updates a total of 5 different hash table entries across 3 different hash tables for each input pixel. Blur operates across each of the 5 dimensions, reading and writing values from multiple hash tables. Similarly slice operation again needs to access 5 different hash table entries across 3 different hash tables to produce each output pixel. With convolution operations of Chapter 5, feeding data to the 64 ALUs was easy as adjacent pixels operated on adjacent and overlapping data values. Here each of the 16 sets of ALUs potentially needs data from a different part of the hash table. This has implications on both the data access support as well as the address generation and control flow support that is needed.

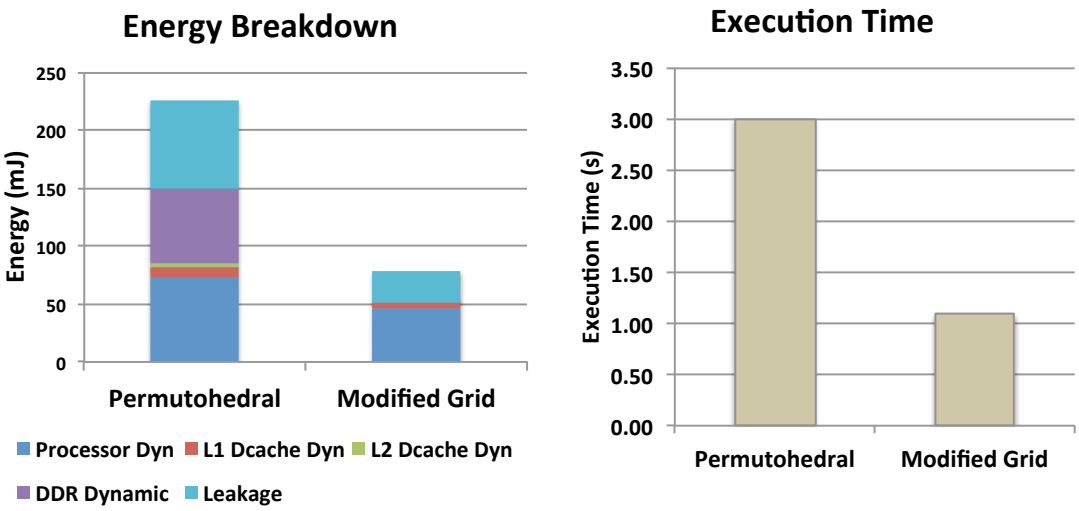


Figure 6.7: Energy consumption and execution time for permutohedral lattice based algorithm vs. our modified bilateral grid. The test image is an HD video frame. The decimation factor is 32 in spatial dimensions and 16 in RGB dimensions.

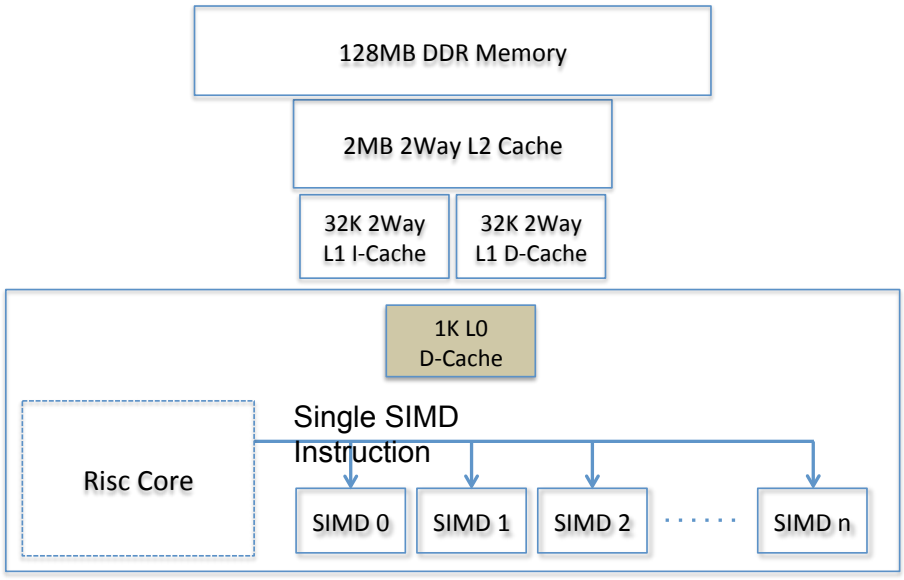


Figure 6.8: Proposed datapath for modified bilateral filtering. It consists of an array of 4-way SIMD units, all operated by the same instruction stream. While not explicitly shown, each SIMD unit has its own address generation unit as well as a dedicated port to access the L0-cache.

Figure 6.8 shows the structure of the proposed datapath. The datapath is organized as an array of 4-wide SIMD ALUs. We evaluate multiple sizes of arrays, starting from a single 4-way SIMD ALU, going up to 8, 16 or higher number of SIMD ALUs. Since each pixel goes through the same computation, these SIMD ALUs are controlled by a single instruction stream, providing the amortization of instruction fetch cost as needed. Since each 4-ALU SIMD unit needs data from different parts of memory, each of these units has its own address generation unit, with dedicated address registers.

The datapath also includes a small 1K cache between the processor and the L1 cache. Inclusion of this cache is motivated by the observation that there exists a large degree of short-term temporal locality in the algorithm. Adjacent pixels in the image generally have similar color values and thus have a high probability of updating the same hash table entries. The L0 cache captures this very effectively and has about 90% hit rate. Due to its smaller size it also has roughly 5 times lower access energy compared to the 32-K L1 D-Cache, resulting in about 60% reduction in overall memory access energy. This increases the amount of data-parallelism that could be exploited before memory becomes a bottleneck again. The second use of this structure is to provide the larger bandwidth needed to feed the large number of SIMD arrays, and it does that by providing multiple access ports.

6.4.1 Path Divergence

In normal execution, all the SIMD ALUs operate in lock-step executing the same set of operations from a single execution stream. There are however some instances where the execution paths diverge depending on the data, and special support is needed to handle those cases. Small data-dependent control-flow deviations among these SIMD units are handled through conditionally executed instructions. Computing the co-ordinates of say the x-neighbor of our closest grid point is one such example. Depending on the pixel position, this could be on right or left of the closest grid point. Thus both of these paths are executed and depending on the condition flags, each SIMD unit gets the result of one or the other path.

Another divergence happens when one or more of the SIMD ALUs miss in the L0 cache. In this case we assume that the misses are sequentially serviced, all units remain stalled until misses have been served and then the execution resumes in lock step. The third divergence happens when one of the SIMD units misses on a hash table access. This happens when the index predicted by the hash function does not have a matching key, and multiple iterations are needed to get to the correct hash table index. In this case we assume that the execution becomes serial, where the missing unit continues on its own until it gets to the correct index.

6.5 Acceleration Results

6.5.1 Simulation Methodology

Unfortunately some aspects of this proposed architecture are not expressible using the Tensilica tool system that we have used so far. These include support for a multi-ported L0 cache, as well as the control flow support for detecting and handling the path divergence arising from cache misses or hash table misses. Therefore we use Tensilica simulation in combination with some estimation based on program statistics. The version with a single SIMD unit can be simulated directly in the Tensilica flow without any manual estimation. For the multiple SIMD case, we use TIE to create the wide datapath instructions including the compute instructions as well as address generation and hash computation instructions. We also use Tensilica's TIE compiler tool to get the energy estimation for these instructions. Then we analyze the inner loop of the simulated single-SIMD execution and manually replace the energy for the narrow SIMD operations by the energy of these wider SIMD operations to get an energy estimate. For memory system we developed a small memory simulation system in C to get the cache miss and hit statistics from program execution trace. These statistics were used in combination with cache and DRAM accesses energies calculated using CACTI 6.0 [41], to find the energy for various levels of memory hierarchy. Finally the cache miss and hash table miss statistics gathered from single-unit SIMD execution were used to estimate the additional stall cycles incurred to

handle the path divergence. These stall cycles were then added to execution time and at the same time leakage energy as well as processor dynamic energy numbers were updated to reflect these additional idle cycles.

6.5.2 Results

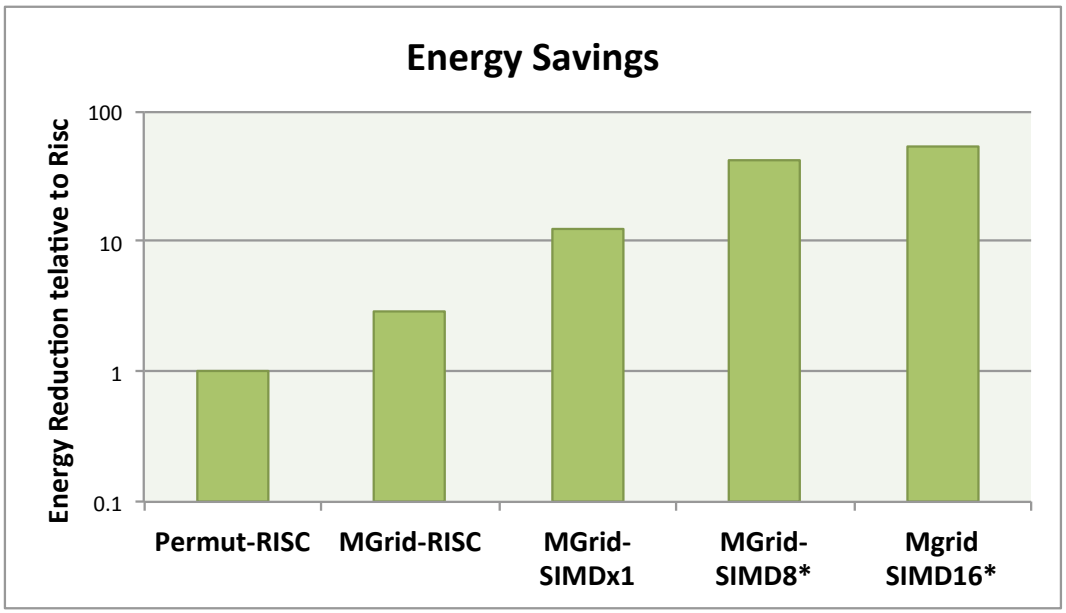


Figure 6.9: Energy consumption for various implementations of RGB bilateral filtering for a HD video frame. Permut refers to permutohedral lattice implementation, MGrid refers to modified grid algorithm. For modified grid we present results for RISC implementation as well as SIMD implementations with one, 8 and 16 SIMD units. Results are normalized with respect to permutohedral lattice RISC implementation. The results marked with '*' use a combination of simulation and manual estimation.

Figure 6.9 shows the energy consumption for various implementations of RGB bilateral filtering for a HD video frame. The single-SIMD-unit implementation gets more than 4x improvement over the RISC implementation with a total reduction of about 12x over the permutohedral lattice implementation. The reduction is higher than SIMD width of 4 because apart from SIMD-parallelism this implementation also benefits from dedicated address registers, custom instructions that fuse computing

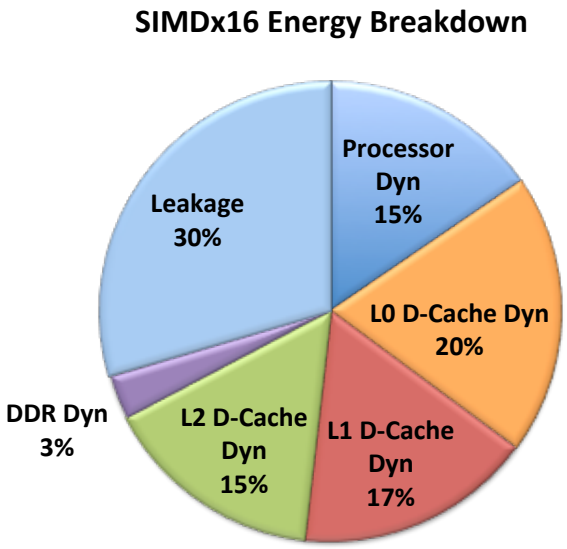


Figure 6.10: Energy consumption breakdown for 16-SIMD-Unit implementation of *modified grid* bilateral filtering.

the hash and accessing the indexed key into one instruction, as well as conditional execution removing some of the branch overheads. Having eight of these SIMD units gives another 3.5x reduction in energy, with a total reduction of just over 40x, and finally 16 SIMD units only gives an improvement of about 1.25x over 8 SIMD units. The diminishing returns in going from 8 to 16 SIMD units can be explained by looking at the energy breakdown of 16 SIMD unit case, presented in Figure 6.10.

As shown the processor core dynamic energy now represents only 15% of the energy consumption and energy for various caches now dominates. Therefore unless further reductions could be made in memory accesses to higher-level caches, further parallelism would not result in a significant reduction in energy consumption. Overall the optimized 16-SIMD-unit version gains a 50x energy reduction compared to the state-of-the-art permutohedral lattice implementation.

6.6 Conclusion

The bilateral filtering algorithm flow is very different and complex compared to the linear convolution operations of Chapter 5. However, the final optimized hardware for this algorithm draws on the same basic architectural ideas. Parallelism is achieved using a wide arithmetic array, which is supported by a small local store that provides low-energy, high bandwidth memory accesses by exploiting the re-use. However the algorithmic changes required to exploit this abstraction go beyond simple software optimization and require domain expertise to come up with a fundamentally new algorithm that could map well to our desired abstraction. This is similar to what has been happening in GPU world. GPU hardware achieves very high performance for a specific type of data-parallel operations common in Graphics. However subsequently a number of algorithms that were not the intended target of the architecture have been restructured to make use of the high performance available in the GPUs. This has also lead to the development of CUDA framework [42], which enables tapping the power of GPUs for a range of algorithms outside graphics. We believe that the domain-customized functional unit approach requires a similar mindset on the part of system designers and application developers and relies as much on our ability to transform the algorithms to make use of well-optimized hardware abstractions, as it depends on our ability to create efficient units for frequently used computation patterns.

Chapter 7

Conclusions

As specialization emerges as the main approach to addressing the energy limitations of current architectures, there is a strong desire to extract maximal re-use from these specialized engines. We propose handling that challenge by identifying and targeting computation patterns that have two properties - they are used across a broad range of algorithms, and they can be implemented very efficiently in hardware. The second property places an important constraint. Some existing computing abstractions such as 1D SIMD model as well as traditional DSP architectures have wide applicability, but fail to achieve the very high performance and extremely low energy consumption of dedicated custom hardware. Other abstractions such as GPU computing model achieve very high performance across a variety of applications, however the large resource requirements of that model preclude low-energy implementations comparable to specialized hardware.

One of the contributions of this thesis is to identify the characteristics that a computation pattern must have for efficient implementation to be possible. To offset the high energy-cost of data memory accesses, a computation pattern must have enough reuse in input and/or intermediate data values that close to 99% of the data accesses could be served by a small storage structure local to the datapath. Moreover the computation pattern should have a large-degree of parallelism, so that instruction supply cost can be amortized over a large number of arithmetic operations per instruction.

The general architectural template of a hardware unit targeting such computations

includes a large compute array, combined with a small, low-energy local data buffer that could also provide very high bandwidth. Our work shows that providing this high bandwidth normally requires that the data routing from the storage structures to the compute array is highly tuned to the data-access requirements of the algorithm. Thus efficient hardware units are generally tied to a specific data-flow pattern. Thus our quest for efficiently-implementable reusable computation abstractions translates to finding data-flow patterns with large parallelism and data locality that repeatedly occur across many applications.

We believe that key is to concentrate on specific computation domains as algorithms in a domain often share similar data-flow patterns. Thus our convolution engine targets the convolution abstraction prevalent in image and media processing applications. The computing model is more restricted compared to say a SIMD machine but fully user programmable and useful across a range of media processing applications. Importantly as our work shows it can be implemented with an order of magnitude higher efficiency compared to a traditional SIMD unit. While the CE is a single example, we hope that similar studies in other application domains will lead to other efficient, programmable, specialized accelerators.

Another important question is whether every new application domain and computation pattern that we explore requires a complete effort from scratch to create completely new efficient abstraction, or can some of work be leveraged for other domains as well. Our work on bilateral filtering provides some answers to that question. While bilateral filtering has a very different computation structure from convolution operations, the hardware abstractions that we use to implement it have a similar structure to convolution hardware. There are however a couple of issues. Bilateral filtering required a major algorithmic modification to make it fit our parallelism model. Such algorithmic modifications are often carried out when creating custom hardware as well. However in the context of domain-customized functional units, the restructuring must be done keeping in view the computation abstractions that can be implemented efficiently by domain customized units. Also while bilateral unit and Convolution unit both utilize a small local buffer to capture re-use, the interface between this buffer and the compute array, which enables re-use as well as

high-bandwidth access, is completely different. Nevertheless the general architectural template could still be re-used across domains.s

In our work we have primarily considered only one parallelism model based on a single, wide compute array, in which each element performs the same arithmetic operation. In essence this is just an extension of the SIMD model with appropriate data storage and access support to enable a larger degree of parallelism than is possible with the traditional SIMD machine. In a less constrained setting each arithmetic unit could be performing a different operation. At the same time instead of implementing a single, wide computation the arithmetic units could be arranged in a cascade to implement an instruction flow graph over multiple cycles, with parallelism extracted through pipelining. It is not obvious how to create a datapath structure to handle these models in a generic way, and how to control such a structure using a single instruction stream with a reasonably friendly programming model. Nevertheless some of our early work in this area shows promise and invites further exploration.

Another important question is how to program these specialized units. Convolution Engine is programmed in standard C augmented with intrinsics for special CE instructions. This has the advantage of providing a familiar well-established development environment to control any specialized unit. It also eliminates the need to partition an application into "kernels" running on the accelerator and higher-level application code running on an application processor. The accelerator instructions could be mixed into regular C code as needed. However making good use of these intrinsic instructions requires the application developer to have a fairly detailed understanding of the underlying hardware. An alternate approach is to use Domain Specific Languages (DSLs) [14], where a domain expert can express an algorithm using a set of algorithmic constructs common to the domain. Compared to a general C compiler, the DSL compiler has lot more information about the structure of the computations and can automatically generate optimized code to run on the programmable accelerator.

Bibliography

[1] Digic Processors, Canon Inc.

[2] Omap 5 platform, texas instruments. www.ti.com/omap.

[3] Snapdragon Processors, Qualcomm Inc.

[4] Tegra processors. NVIDIA Corporation.

[5] A. Adams, J. Baek, and M. A. Davis. Fast high-dimensional filtering using the permutohedral lattice. In *Computer Graphics Forum*, volume 29, pages 753–762. Wiley Online Library, 2010.

[6] A. Adams, N. Gelfand, J. Dolson, and M. Levoy. Gaussian kd-trees for fast high-dimensional filtering. *ACM Transactions on Graphics (TOG)*, 28(3):21, 2009.

[7] A. Adams, D. Jacobs, J. Dolson, M. Tico, K. Pulli, E. Talvala, B. Ajdin, D. Vaquero, H. Lensch, M. Horowitz, et al. The frankencamera: an experimental platform for computational photography. *ACM Transactions on Graphics (TOG)*, 2010.

[8] A. B. Adams. *High-dimensional gaussian filtering for computational photography.* Stanford University, 2011.

[9] V. O. Alan, W. S. Ronald, and R. John. Discrete-time signal processing. *New Jersey, Printice Hall Inc*, 1989.

[10] O. Azizi, A. Mahesri, B. C. Lee, S. Patel, and M. Horowitz. Energy-Performance Tradeoffs in Processor Architecture and Circuit Design: A Marginal Cost Analysis. In *ISCA '10: Proc. 37th Annual International Symposium on Computer Architecture*. ACM, 2010.

[11] S. Bae, S. Paris, and F. Durand. Two-scale tone management for photographic look. In *ACM Transactions on Graphics (TOG)*, volume 25, pages 637–645. ACM, 2006.

[12] 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.

[13] E. P. Bennett and L. McMillan. Video enhancement using per-pixel virtual exposures. In *ACM Transactions on Graphics (TOG)*, volume 24, pages 845–852. ACM, 2005.

[14] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, pages 35–46. ACM, 2011.

[15] J. Chen, S. Paris, and F. Durand. Real-time edge-aware image processing with the bilateral grid. In *ACM Transactions on Graphics (TOG)*, volume 26, page 103. ACM, 2007.

[16] T. C. Chen. Analysis and architecture design of an HDTV720p 30 frames/s H.264/AVC encoder. *IEEE Transactions on Circuits and Systems for Video Technology*, 16(6):673–688, 2006.

[17] T.-C. Chen, Y.-W. Huang, and L.-G. Chen. Analysis and design of macroblock pipelining for h. 264/avc vlsi architecture. In *Circuits and Systems, 2004. ISCAS'04. Proceedings of the 2004 International Symposium on*, volume 2, pages II–273. IEEE, 2004.

[18] N. Corporation. *Expeed Digital Image Processors*. Nikon Corporation., http://imaging.nikon.com/lineup/microsite/d300.

[19] S. Corporation. *BIONZ Image Processing Engine*. Sony Corporation., http://www.sony-mea.com/microsite/dslr/10/tech/bionz.html.

[20] W. Davis, N. Zhang, K. Camera, F. Chen, D. Markovic, N. Chan, B. Nikolic, and R. Brodersen. A design environment for high throughput, low power, dedicated signal processing systems. In *Custom Integrated Circuits Conference(CICC)*, 2001.

[21] P. Debevec, E. Reinhard, G. Ward, and S. Pattanaik. High dynamic range imaging. In *ACM SIGGRAPH 2004 Course Notes*, page 14. ACM, 2004.

[22] 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)*, 87(4):668–678, 1999.

[23] F. Durand and J. Dorsey. Fast bilateral filtering for the display of high-dynamic-range images. *ACM Transactions on Graphics (TOG)*, 21(3):257–266, 2002.

[24] S. Farsiu, M. D. Robinson, M. Elad, and P. Milanfar. Fast and robust multiframe super resolution. *Image processing, IEEE Transactions on*, 13(10):1327–1344, 2004.

[25] R. Gonzalez. Xtensa: A Configurable and Extensible Processor. *Micro, IEEE*, 20(2):60–70, Mar/Apr 2000.

[26] R. Gonzalez. Xtensa: a configurable and extensible processor. *Micro, IEEE*, 20(2):60–70, Mar/Apr 2000.

[27] D. Goodwin and D. Petkov. Automatic generation of application specific processors. In *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 137–147. ACM, 2003.

[28] D. Grose. Keynote: From Contract to Collaboration Delivering a New Approach to Foundry. DAC '10: Design Automation Conference, June 2010.

[29] 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.

[30] Intel Corporation. Intel SSE4 Programming Reference.

[31] ITU-T. Joint Video Team Reference Software JM8.6.

[32] V. Iverson, J. McVeigh, and B. Reese. Real-time H.264/avc Codec on Intel architectures. *IEEE Int. Conf. Image Processing ICIP'04*, 2004.

[33] P. Karas and D. Svoboda. Algorithms for efficient computation of convolution. *Design and Architectures for Digital Signal Processing. 1st ed. Rijeka (CRO): InTech*, pages 179–208, 2013.

[34] M. Kimura, K. Iwata, S. Mochizuki, H. Ueda, M. Ehama, and H. Watanabe. A full hd multistandard video codec for mobile applications. *Micro, IEEE*, 29(6):18–27, 2009.

[35] J. Leng, S. Gilani, T. Hetherington, A. ElTantawy, 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.

[36] M. Levoy, B. Chen, V. Vaish, M. Horowitz, I. McDowall, and M. Bolas. Synthetic aperture confocal imaging. *ACM Transactions on Graphics (TOG)*, 23(3):825–834, 2004.

[37] F. Liu, M. Gleicher, J. Wang, H. Jin, and A. Agarwala. Subspace video stabilization. *ACM Transactions on Graphics (TOG)*, 30(1):4, 2011.

[38] D. Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[39] Y. Matsushita, E. Ofek, X. Tang, and H. Shum. Full-frame video stabilization. In *Computer Vision*.

[40] G. Moore. Cramming More Components onto Integrated Circuits. *Electronics Magazine*, 38(8), April 1965.

[41] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. Cacti 6.0: A tool to model large caches. *HP Laboratories*, 2009.

[42] C. Nvidia. Compute unified device architecture programming guide. 2007.

[43] B. M. Oh, M. Chen, J. Dorsey, and F. Durand. Image-based modeling and photo editing. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 433–442. ACM, 2001.

[44] S. Paris and F. Durand. A fast approximation of the bilateral filter using a signal processing approach. In *Computer Vision–ECCV 2006*, pages 568–580. Springer, 2006.

[45] G. Petschnigg, R. Szeliski, M. Agrawala, M. Cohen, H. Hoppe, and K. Toyama. Digital photography with flash and no-flash image pairs. In *ACM Transactions on Graphics (TOG)*. ACM, 2004.

[46] F. Porikli. Constant time o (1) bilateral filtering. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8. IEEE, 2008.

[47] R. Raskar. Computational photography. In *Computational Optical Sensing and Imaging*. Optical Society of America, 2009.

[48] V. G. Reddy. Neon technology introduction. *ARM Corporation*, 2008.

[49] W. C. Rhines. Keynote: World Semiconductor Dynamics: Myth vs. Reality. Semicon West '09, July 2009.

[50] R. Rithe, P. Raina, N. Ickes, S. V. Tenneti, and A. P. Chandrakasan. Reconfigurable processor for energy-efficient computational photography. 2013.

[51] C. Rowen and S. Leibson. Flexible architectures for engineering successful SOCs. *Design Automation Conference, 2004. Proceedings. 41st*, pages 692–697, 2004.

[52] 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. Why design must change: Rethinking digital design. *IEEE Micro*, 30(6):9 –24, nov.-dec. 2010.

[53] H. Shojania and S. Sudharsanan. A VLSI Architecture for High Performance CABAC Encoding. In *Visual Communications and Image Processing*, 2005.

[54] S. W. Smith et al. The scientist and engineer's guide to digital signal processing. 1997.

[55] 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.

[56] Tensilica Inc. ConnX Vectra LX DSP Engine Guide.

[57] Tensilica Inc. Tensilica Instruction Extension (TIE) Language Reference Manual.

[58] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In *Computer Vision, 1998. Sixth International Conference on*, pages 839–846. IEEE, 1998.

[59] D. Van Krevelen and R. Poelman. A survey of augmented reality technologies, applications and limitations. *International Journal of Virtual Reality*, 9(2):1, 2010.

[60] J. Xiao, H. Cheng, H. Sawhney, C. Rao, and M. Isnardi. Bilateral filtering-based optical flow estimation with occlusion detection. In *Computer Vision–ECCV 2006*, pages 211–224. Springer, 2006.

[61] Q. Yang, K.-H. Tan, and N. Ahuja. Real-time o (1) bilateral filtering. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 557–564. IEEE, 2009.