UNIVERSITY OF CALIFORNIA SAN DIEGO

Specialization as a Candle in the Dark Silicon Regime

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

in

Computer Science

by

Nathan Goulding-Hotta

Committee in charge:

Professor Steven Swanson, Co-Chair
Professor Michael Bedford Taylor, Co-Chair
Professor Rajesh Gupta
Professor Ryan Kastner
Professor Ramesh Rao

2020

The Dissertation of Nathan Goulding-Hotta is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Co-Chair

Co-Chair

University of California San Diego

2020

DEDICATION

To those who embrace humanity (and a sense of humor).

EPIGRAPH

Maybe in order to understand mankind, we have to look at the word itself. "Mankind."
Basically, it's made up of two separate words—"mank" and "ind."
What do these words mean? It's a mystery, and that's why so is mankind.

—Jack Handey

Hofstadter's Law:
It always takes longer than you expect,
even when you take into account Hofstadter's Law.

—Douglas Hofstadter

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the help of myriad giants. I'm eternally grateful to the people listed here and countless others for their support over the years. I'm only using first names to protect the innocent (and not-so-innocent—you know who you are).

First and foremost, I'd like to express my deepest appreciation for my amazing advisors, Professor Steven Swanson and Professor Michael Taylor. Ever supportive, they have guided and encouraged me since the beginning, and they never gave up on me finishing. Steve, a paper, grant, and code writing machine, you taught me the value of efficiency, higher standards of one's best effort, and how to draw a good graph (among other things). Michael, a tenacious uberhacker, you taught me to attempt the impossible, to persevere, and how to tell a great story about it afterward. You both gave me the foundation necessary to succeed in life after UCSD.

Thank you to my committee members, Professors Rajesh Gupta, Ryan Kastner, and Ramesh Rao, for your work that inspired me, your thoughtful feedback, and your willingness to serve on the committee for nearly a decade. Thanks also go to Julie Conner, for dusting off my file again and answering my never-ending questions.

The work presented in this dissertation was in close collaboration with a great number of highly-talented individuals. I'm especially thankful to my partners in crime, Jack and Ganesh. Jack, a walking Wikipedia, you taught me something about everything and everything about some things. Your generosity and creativity are exemplified by your primordial dinner parties. Ganesh, for someone so easygoing you sure get a lot done. It was a blast working with you late nights; I laughed the hardest with you. I'm also grateful to my other coauthors, especially Sravanthi, Qiaoshi, Sat, Vikram, and Jose. I could not have asked for a better research group.

Thank you to my classmates, office mates, roommates, foosball mates, and Chez Bob mates, for making UCSD such a fun place to be: Emmett, Laura, Adrian, Anshuman, Donghwan, Pat, Ravi, Zach, Jan, Joel, John, Justin, Ming, Hung-Wei, Edward, and so many more. Steve Checkoway saved me enormous time with his invaluable LaTeX dissertation template [Che].

Special thanks to Professors Ranjit Jhala and Sorin Lerner, for stimulating impromptu discussions every time they stopped by our office providing (or in search of) snacks. And thank you to Ryoko-san, for filling the halls with music.

I'd like to acknowledge the support of my colleagues at UCSC and GlobalFoundries: Professor Jose Renau, Rigo, Luigi, Vito, Edward, Shobhit, and the whole group, for welcoming me with open arms, for teaching me how to tackle the CAD tools, and for always helping me fix one last DRC.

I'm extremely grateful to and humbled by my fellow Googlers and Paintboxers, a team of serious superstars: Adam, Albert, Alex, Andrea, Artem, Ashok, Asif, Ben, Bill, Bobbie, Cheng, Daniel, Dave, Don, Ed, Hua, Hyunchul, Jason, Ji, John, Jolin, Jon, Karthika, Masumi, Michelle, Neeti, Penny, Rolf, Scott, Sean, Sha, Todd, Trevor, Victor, and intern-turned-distinguished-software-engineer Professor Dave Patterson. A very special thank you to Ofer Shacham, who, with his big heart and dedication, showed me what it means to be a true leader.

Thank you to my friends and family for their constant support and patience. To my BFFs, Alex and Carlos, for their subtle and not-so-subtle encouragement until I finished. To my brother Kevin, for all his helpful "ideas;" to my sister Emily, whose bubbliness and inside jokes always make me feel special; and to my niece and nephews, who put a smile on my face during tough times writing this dissertation. To my loving and lovely parents, for instilling confidence and unbridled optimism in me, and for teaching me how to always see life with light and laughter.

Finally, my biggest and heartfeltest thank you to Kana, my partner, my best friend, and my favorite person in the world. (Don't tell my BFFs I said that.)

Thank you!

Chapters 2 and 3 contain material from "Conservation Cores: Reducing the Energy of Mature Computations," by Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor, which has appeared in the Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ©2010 ACM. The dissertation author is a primary contributor and third author of this paper.

Chapters 2 and 4 contain material from "GreenDroid: A Mobile Application Processor for a Future of Dark Silicon," by Nathan Goulding, Jack Sampson, Ganesh Venkatesh, Saturnino Garcia, Joe Auricchio, Jonathan Babb, Michael Bedford Taylor, and Steven Swanson, which has appeared in Hot Chips 22: A Symposium on High Performance Chips, ©2010 IEEE. The dissertation author is a primary contributor and first author of this paper.

Chapter 3 contains material from "Efficient Complex Operators for Irregular Codes," by Jack Sampson, Ganesh Venkatesh, Nathan Goulding-Hotta, Saturnino Garcia, Steven Swanson, and Michael Bedford Taylor, which has appeared in the Proceedings of the 17th International Symposium on High Performance Computer Architecture, ©2011 IEEE. The dissertation author is a primary contributor and third author of this paper.

Chapter 3 contains material from "QsCores: Trading Dark Silicon for Scalable Energy Efficiency with Quasi-Specific Cores," by Ganesh Venkatesh, Jack Sampson, Nathan Goulding-Hotta, Sravanthi Kota Venkata, Michael Bedford Taylor, and Steven Swanson, which has appeared in the Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, ©2011 IEEE/ACM. The dissertation author is a contributor and third author of this paper.

Chapter 4 contains material from "The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future," by Nathan Goulding-Hotta, Jack Sampson, Ganesh Venkatesh, Saturnino Garcia, Joe Auricchio, Po-Chao Huang, Manish Arora, Siddhartha Nath, Vikram Bhatt, Jonathan Babb, Steven Swanson, and Michael Bedford Taylor, which has appeared in IEEE Micro, ©2011 IEEE. The dissertation author is a primary contributor and first author of

this paper.

Chapters 5 and 6 contain material from "Pixel Visual Core: Google's Fully Programmable Image, Vision and AI Processor for Mobile Devices," by Jason Redgrave, Albert Meixner, Nathan Goulding-Hotta, Artem Vasilyev, and Ofer Shacham, which has appeared in Hot Chips 30: A Symposium on High Performance Chips, ©2018 IEEE. The dissertation author is a primary contributor and third author of this paper.

This dissertation was powered by 346 cups of tea.

VITA

2007	B.S. in Electrical Engineering New Mexico Tech
2007	B.S. in Computer Science New Mexico Tech
2011	M.S. in Computer Science University of California San Diego
2011	C.Phil. in Computer Science University of California San Diego
2015–	Diplomatic Hardware Engineer Google
2020	Ph.D. in Computer Science University of California San Diego

PUBLICATIONS

Jason Redgrave, Albert Meixner, Nathan Goulding-Hotta, Artem Vasilyev, and Ofer Shacham, "Pixel Visual Core: Google's Fully Programmable Image, Vision and AI Processor for Mobile Devices," Hot Chips 30, Cupertino, CA, Aug. 2018.

Qiaoshi Zheng, Nathan Goulding-Hotta, Scott Ricketts, Steven Swanson, Michael Bedford Taylor, and Jack Sampson, "Exploring Energy Scalability in Coprocessor-Dominated Architectures for Dark Silicon," ACM Transactions on Embedded Computing Systems, Apr. 2014.

Vikram Bhatt, Nathan Goulding-Hotta, Qiaoshi Zheng, Jack Sampson, Steven Swanson, and Michael Bedford Taylor, "SiChrome: Mobile Web Browsing in Hardware to Save Energy," 1st Dark Silicon Workshop (DaSi), 39th International Symposium on Computer Architecture (ISCA), Portland, OR, Jun. 2012.

Nathan Goulding-Hotta, Jack Sampson, Qiaoshi Zheng, Vikram Bhatt, Joe Auricchio, Steven Swanson, and Michael Bedford Taylor, "GreenDroid: An Architecture for the Dark Silicon Age," 17th Asia and South Pacific Design Automation Conference (ASP-DAC), Sydney, Australia, Feb. 2012.

Ganesh Venkatesh, Jack Sampson, Nathan Goulding-Hotta, Sravanthi Kota Venkata, Michael Bedford Taylor, and Steven Swanson, "QsCores: Trading Dark Silicon for Scalable Energy Efficiency with Quasi-Specific Cores," 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Porto Alegre, Brazil, Dec. 2011.

Jack Sampson, Manish Arora, Nathan Goulding-Hotta, Ganesh Venkatesh, Jonathan Babb, Vikram Bhatt, Steven Swanson, and Michael Bedford Taylor, "An Evaluation of Selective Depipelining for FPGA-based Energy-Reducing Irregular Code Coprocessors," 2011 International Conference on Field Programmable Logic and Applications (FPL), Crete, Greece, Sep. 2011.

Manish Arora, Jack Sampson, Nathan Goulding-Hotta, Jonathan Babb, Ganesh Venkatesh, Michael Bedford Taylor, and Steven Swanson, "Reducing the Energy Cost of Irregular Code Bases in Soft Processor Systems," 19th Annual International IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), Salt Lake City, UT, May 2011.

Nathan Goulding-Hotta, Jack Sampson, Ganesh Venkatesh, Saturnino Garcia, Joe Auricchio, Po-Chao Huang, Manish Arora, Siddhartha Nath, Vikram Bhatt, Jonathan Babb, Steven Swanson, and Michael Bedford Taylor, "The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future," IEEE Micro, Mar./Apr. 2011.

Jack Sampson, Ganesh Venkatesh, Nathan Goulding-Hotta, Saturnino Garcia, Steven Swanson, and Michael Bedford Taylor, "Efficient Complex Operators for Irregular Codes," 17th IEEE International Symposium on High Performance Computer Architecture (HPCA), San Antonio, TX, Feb. 2011.

Nathan Goulding, Jack Sampson, Ganesh Venkatesh, Saturnino Garcia, Joe Auricchio, Jonathan Babb, Michael Bedford Taylor, and Steven Swanson, "GreenDroid: A Mobile Application Processor for a Future of Dark Silicon," Hot Chips 22, Stanford, CA, Aug. 2010.

Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor, "Conservation Cores: Reducing the Energy of Mature Computations," 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Pittsburgh, PA, Mar. 2010.

Nathan Goulding, Gopi Tummala, Emmett McQuinn, and Steven Swanson, "ReMOS Technology and the Mighty Morphin Microprocessor," Wild and Crazy Ideas session of ASPLOS 2010, Pittsburgh, PA, Mar. 2010.

Thomas Claytor, Joel Marquez, Lian-Jie Huang, Brett Nadler, Nathan Goulding, and Emily Prewett, "Ultrasonic Imaging Techniques for Early Breast Cancer Detection," Review of Progress in Quantitative Nondestructive Evaluation (QNDE), Golden, CO, Jul. 2007.

Nathan Goulding, Joel Marquez, Emily Prewett, Thomas Claytor, Lian-Jie Huang, and Brett Nadler, "Ultrasonic Imaging Techniques for Breast Cancer Detection," International Modal Analysis Conference, Orlando, FL, Feb. 2007.

Nathan Goulding, Jason Hamlet, Gar Hassall, Furqan Chiragh, and Scott Cason, "An Expressive Stereoscopic Vision Tracking System," Electrical Manufacturing Expo, Indianapolis, IN, Sep. 2006.

ABSTRACT OF THE DISSERTATION

Specialization as a Candle in the Dark Silicon Regime

by

Nathan Goulding-Hotta

Doctor of Philosophy in Computer Science

University of California San Diego, 2020

Professor Steven Swanson, Co-Chair
Professor Michael Bedford Taylor, Co-Chair

For decades computer architects have taken advantage of Moore's law to get bigger, faster, and more energy-efficient chips "for free," reaping the benefits of silicon process improvements and shrinking technology nodes. Each new technology node brought exponentially more transistors, balanced by exponentially lower transistor switching power, allowing the power budget for a fixed silicon area to remain relatively constant. Architects could count on more transistors—and use them to build more complex designs—without substantially increasing the total power budget for a chip.

Today, however, rising CMOS leakage currents have limited further reductions in supply

voltage, leading to a power-limited utilization wall and an end to classical Dennard scaling. This breakdown results in a new regime of *dark silicon*, in which vast swaths of silicon area must remain "dark" (powered down or under-clocked) most of the time. Architects must turn to novel approaches to squeeze ever more performance out of every last square-millimeter of silicon.

This dissertation demonstrates that one viable approach to the dark silicon problem is *specialization*. Rather than relying solely on bigger, faster, general-purpose processors, chip architects have been increasingly augmenting their systems with special-purpose *accelerators*. These accelerators can speed up a given computation, allow it to run with less energy, or both. Using less energy frees up power and thermal budgets, allowing more computations to run in parallel and extending the computational capabilities we've come to demand from silicon.

This dissertation presents two such specialized architectures. The first is GreenDroid, a mobile application processor built with custom accelerators targeting Android. The accelerators are energy-saving specialized circuits called conservation cores, or c-cores. In a 45-nm process, just 7 mm^2 of silicon dedicated to c-cores covers approximately 95% of our Android workload. Powered by c-cores, GreenDroid uses 11$\times$ less energy on average than a general-purpose CPU.

The second is Pixel Visual Core, a commercial accelerator from Google that enables energy-efficient computational photography and machine learning in the Pixel 2 and Pixel 3 smartphones. Pixel Visual Core is powered by an 8-core Image Processing Unit with 4,096 16-bit ALUs capable of performing 3.1 Tera-operations/second in under 5 watts. Compared to a 10-nm general-purpose application processor, the 28-nm Pixel Visual Core runs key compute kernels 3-6$\times$ faster and with 7-16$\times$ less energy.

Chapter 1

Introduction

For more than five decades chip architects and programmers have taken advantage of Moore's law to get bigger, faster, and more energy-efficient chips "for free," reaping the benefits of silicon process improvements and shrinking technology nodes. Each new technology node brought exponentially more transistors, balanced by exponentially lower transistor switching power, which meant the power budget for a fixed silicon area remained relatively constant. Architects could count on more transistors—and use them to build more complex designs—without substantially increasing the total power budget for a chip.

Today, however, rising leakage currents have limited further reductions in supply voltage, so architects no longer see an exponential decrease in transistor switching power. As a result, chips have run up against a *utilization wall* [VSG+10]: With each successive process generation, the percentage of a chip that can actively switch drops exponentially because of power constraints.

The utilization wall and breakdown of classical CMOS scaling results in a new regime of *dark silicon* [GSV+10]. Under the dark silicon regime, vast swaths of silicon area must remain "dark" (powered down or under-clocked) most of the time. Architects must turn to novel approaches to squeeze ever more performance out of every last square-millimeter of silicon.

This dissertation demonstrates that one viable approach to the dark silicon problem is *specialization*. Rather than relying solely on bigger, faster, general-purpose processors, chip architects have been increasingly augmenting their systems with special-purpose *accelerators*.

These accelerators can speed up a given computation, allow it to run with less energy, or both. Using less energy frees up power and thermal budgets, allowing more computations to run in parallel and extending the computational capabilities we've come to demand from silicon.

Specialization itself is not new (e.g., in 1980 Intel introduced a discrete coprocessor, the 8087 floating point unit, to augment its 8086 line of processors). But what is new is the number and variety of specialized architectures that have gained popularity in recent years. This dissertation explains two such specialized architectures in detail. The first is a research prototype called *GreenDroid*, a multicore application processor with custom accelerators for Android. The second is a commercial chip called *Pixel Visual Core*, an image and machine learning accelerator in Google's Pixel 2 and Pixel 3 smartphones. Both architectures demonstrate the viability of specialization.

The dissertation is organized as follows. First, Chapter 2 explains the dark silicon problem and proposes specialization as a viable solution to continued scaling. The chapter starts with background material on CMOS scaling theory. We show how the classical scaling model has broken down under rising leakage currents, resulting in the utilization wall. Our research on the utilization wall predicted the rise of dark silicon, and we discuss some approaches to the problem. The most promising of these approaches is specialization. Our early research predicted the appearance of larger and larger numbers of specialized accelerators as a path forward in the dark silicon regime.

Chapter 3 describes our first approach to address dark silicon: a class of specialized accelerators called *conservation cores*. Conservation cores, or *c-cores*, are application-specific hardware circuits created to improve performance and reduce the energy consumption of computationally-heavy applications. Whereas traditional accelerators focus on improving performance, at a potentially worse, equal, or better energy efficiency, conservation cores focus primarily on reducing energy and energy-delay. This chapter describes the c-core architecture and execution model, as well as our toolchain for automatically synthesizing c-cores directly from program source code.

Chapter 4 shows how c-cores can be applied to mobile SoCs running Android. Android is well-suited for c-cores because the hot code is concentrated—in application libraries, the virtual machine, and Linux kernel—meaning a relatively small amount of silicon dedicated for hardware accelerators can cover a relatively large percentage of dynamic execution. We propose a c-core-based application processor called GreenDroid and detail its architecture. This chapter presents our experiments with a fully placed-and-routed GreenDroid tile that includes 9 c-cores generated from Android source code. We show how this tile could be included in a larger many-tile system to cover modern Android workloads. The dissertation author also collaborated with researchers at the University of California, Santa Cruz and GlobalFoundries to design a 28-nm, 2x2-tile version of GreenDroid called MiniDroid. This chapter concludes with the MiniDroid physical implementation in GlobalFoundries' 28-nm SLP process.

The remainder of the dissertation shows how our original predictions on dark silicon and the rise of specialization have come to pass in industry. After working on conservation cores and GreenDroid at UCSD, the dissertation author worked as part of a broad team at Google to design and implement a specialized programmable accelerator called the *Image Processing Unit*, or *IPU*.

Chapter 5 describes the IPU architecture in detail, starting with the motivation and need for a programmable image processing accelerator. Traditional Image Signal Processors (ISPs) are built from fixed-function hardware with limited computational power and minimal or no ability to update the hardened image processing algorithms after manufacture. In contrast, the IPU provides a programmable, energy-efficient, and high-performance compute engine that enables the latest computational photography techniques on mobile devices. We describe the IPU's architecture and programming model, and show how the IPU can accelerate HDR+ image processing.

Chapter 6 introduces the first implementation of the IPU in silicon, Google's *Pixel Visual Core*. Pixel Visual Core (PVC) is included in Google's Pixel 2 and Pixel 3 smartphones. It comprises an 8-core IPU accelerator along with control processor, on-chip interconnect, I/O

interfaces, and stacked DRAM dies in one system-in-package. PVC provides raw performance up to 3.1 Tera-ops/second (1.7 Tera-ops/sec arithmetic) on 16-bit integer data. It achieves about 0.84 pJ/op, or 1.5 pJ/op including only arithmetic operations without data movement. Despite a three-generation process gap, the 28-nm PVC runs key HDR+ kernels 3-6× faster and with 7-16× less energy than a 10-nm general-purpose application processor with DSP.

In the intermediate years between the author's work on GreenDroid and Pixel Visual Core, academia and industry have created many other examples of specialized silicon architectures. Chapter 7 reviews related work in dark silicon research, as well as accelerators for mobile, datacenter, and ambient computing domains. Each of these domains includes accelerators to optimize for different metrics—for example, performance per watt, total cost of ownership (TCO), latency, or ultra-low power. Yet all share a common goal of reaping the benefits of specialization.

Finally, in Chapter 8 we summarize the contributions of this dissertation in the context of related work in specialization and accelerators, and conclude with some final remarks.

Chapter 2

The Rise of Dark Silicon

For more than five decades chip architects and programmers have taken advantage of Moore's law [Moo65] to get bigger, faster, and more energy-efficient chips "for free," reaping the benefits of silicon process improvements and shrinking technology nodes. Each new technology node brought exponentially more transistors, balanced by exponentially lower transistor switching power, which meant the power budget for a fixed silicon area remained relatively constant. Architects could count on more transistors—and use them to build more complex designs—without substantially increasing the total power budget for a chip.

Today, however, limits on threshold voltage scaling have stopped the downward scaling of per-transistor switching power. Consequently, the rate at which we can switch transistors is far outpacing our ability to dissipate the heat created by those transistors. The result is a technology-imposed *utilization wall* [VSG+10] that limits the fraction of a chip that we can use at full speed at one time. The utilization wall forces chip architects into a new regime of *dark silicon* [GSV+10], in which the majority of a chip must remain off most of the time.

This chapter describes the fundamental breakdown of classical CMOS scaling in the face of rising leakage currents, the utilization wall, and the resulting dark silicon problem. But the dark silicon problem also presents a tremendous opportunity, ushering in a new computing era of extreme specialization.

2.1 The Utilization Wall

This section provides background in CMOS scaling theory, shows how traditional scaling breaks down in the face of rising leakage currents, and describes the resulting utilization wall.

2.1.1 CMOS Scaling Theory

Table 2.1 shows the classical CMOS scaling theory as described by Dennard [DGR+74]. The parameter S is the scaling factor between technology nodes, typically $\sqrt{2} \approx 1.4\times$. In the classical scaling regime, transistor threshold voltage V_t continues to decrease as $1/S$ because of improvements in material science and transistor manufacturing. Meanwhile transistor supply voltage V_{dd} is constrained by V_t (typically V_{dd} must be at least $3\times$ higher), so V_{dd} also decreases as $1/S$. Decreasing V_t leads to increasing leakage current, but leakage was still a small fraction of overall chip power in process nodes up to $\sim$130 nm.

Each new process node also brings an S^2 increase in the per-unit-area number of transistors, and a factor of S increase in clock frequency. This would increase power by the same factor (S^3), but a $1/S$ lower V_{dd} brings with it a corresponding $1/S^2$ decrease in per-transistor switching power, on top of $1/S$ lower gate capacitance, balancing the increase in per-transistor switching power. Multiplied together, this means the total chip power at full frequency remained relatively constant across process generations. In other words, in the classical scaling regime chip architects could still *use* all of the new transistors simultaneously—i.e., at fixed power, utilization $U = 100\%$ across process generations.

2.1.2 The End of Dennard Scaling

Starting beyond the 130-nm process node, rising transistor leakage currents started to become a significant fraction of total chip power and could no longer be ignored [HAP+05]. Leakage is highly dependent on threshold voltage V_t, so in the leakage-limited regime V_t can no longer continue scaling by $1/S$ [SKS+13]. As a result, supply voltage V_{dd} also stops scaling by

Table 2.1. CMOS scaling theory and the utilization wall The utilization wall is a consequence of CMOS scaling theory and current-day technology constraints, assuming fixed power and chip area. The parameter S is the scaling factor between technology nodes, typically $\sqrt{2} \approx 1.4\times$. The Classical Scaling column assumes that V_t can be lowered arbitrarily. In the Leakage Limited case, constraints on V_t, necessary to prevent unmanageable leakage currents, hinder scaling and create the utilization wall.

Param.	Description	Relation	Classical Scaling (>130 nm)	Leakage Limited (<90 nm)
B	power budget		1	1
A	chip size		1	1
V_t	threshold voltage		$1/S$	1
V_{dd}	supply voltage	$\sim V_t \times 3$	$1/S$	1
t_{ox}	oxide thickness		$1/S$	$1/S$
W, L	transistor dimensions		$1/S$	$1/S$
I_{sat}	saturation current	WV_{dd}/t_{ox}	$1/S$	1
p	device power at full frequency	$I_{sat}V_{dd}$	$1/S^2$	1
C_{gate}	**capacitance**	WL/t_{ox}	$\mathbf{1/S}$	$\mathbf{1/S}$
F	**device frequency**	$\frac{I_{sat}}{C_{gate}V_{dd}}$	$\mathbf{S}$	$\mathbf{S}$
D	**devices per chip**	$A/(WL)$	$\mathbf{S^2}$	$\mathbf{S^2}$
P	**full die, full frequency power**	$D \times p$	**1**	$\mathbf{S^2}$
U	**utilization at fixed power**	B/P	**1**	$\mathbf{1/S^2}$

$1/S$ and must be held relatively constant. As shown in the Leakage Limited column of Table 2.1, device power at full frequency (p) no longer scales with each process generation.

2.1.3 The Utilization Wall

Traditionally, thanks to Moore's law, chip architects could count on exponentially more transistors with each process generation. They could use these transistors to build faster and more complex designs, such as superscalar, out of order, and multicore CPUs. And because individual transistor switching power also decreased, the total power for a fixed die area did not increase substantially.

Today, however, with the end of Dennard scaling, architects no longer see an exponential decrease in transistor switching power. As a result chips have run up against the *utilization wall*, which states:

> *With each successive process generation, the percentage of a chip that can actively switch drops exponentially because of power constraints.*

Consider an example when scaling from 32 nm to 22 nm [Tay13]. In this example $S = 32/22 = 1.4\times$. According to classical scaling theory, in the newer node we should have $2\times$ more transistors running at $1.4\times$ higher clock frequency, for a total of $2.8\times$ better computational capability. But the utilization wall limits us to achieving only $1.4\times$ of this benefit—a gap of $2\times$. This is a serious problem that gets exponentially worse with every generation.

To quantify the impact of the utilization wall, we synthesized several datapaths using Synopsys Design Compiler and IC Compiler. Table 2.2 summarizes the results. For each process, we used the corresponding TSMC standard cell libraries to evaluate the power and area of a 40-mm^2 chip filled with 64-bit operators, to approximate the active logic in a mobile processor. Each operator is a 64-bit adder with registered inputs, running at maximum frequency for that process. In a 90-nm TSMC process, running this chip at full frequency would require 61 W, which means that only 8.2% of the chip could be used simultaneously within a 5-W power budget. In a 45-nm process, a similar design would require 163 W, resulting in just 3.1% utilization for

Table 2.2. Experiments quantifying the utilization wall Our experiments used Synopsys CAD tools and TSMC standard cell libraries to evaluate the power and utilization of a 40-mm^2 chip filled with 64-bit adders, separated by registers, which is used to approximate active logic in a mobile processor. At a fixed power budget the utilization drops exponentially with each process node, a phenomenon called the utilization wall.

Process	90 nm TSMC	45 nm TSMC	32 nm ITRS
Frequency (GHz)	2.1	5.2	7.3
mm^2 Per Op.	.00724	.00164	.00082
# Operators	5.5k	24k	49k
Full Chip Watts	61	163	320
Utilization at 5 W	8.2%	3.1%	1.6%

the same 5-W power budget. Table 2.2 also extrapolates to 32 nm based on ITRS[1] data for 45- and 32-nm processes. Based on ITRS data the full chip running at full frequency would require 320 W, resulting in just 1.6% utilization.

As we'll show in the next section, the utilization wall fundamentally changes the way silicon architects must think about continued performance scaling.

2.2 Dark Silicon

Because of the utilization wall, ever smaller fractions of a chip can remain simultaneously active at full frequency. Huge regions of a chip must remain under-used, under-clocked, or entirely powered off most of the time—these regions are known as *dark silicon* [Mer09] [GSV+10]. The next two sections introduce the dark silicon problem, and recast dark silicon as an opportunity ushering in a new era in specialized computing.

2.2.1 The Dark Silicon Problem

Dark silicon is a huge problem, and it gets exponentially worse with each process generation. For each new process node, instead of the expected 2.8× scaling in compute performance

[1] International Technology Roadmap for Semiconductors [itr09], succeeded in 2017 by the International Roadmap for Devices and Systems [ird17].

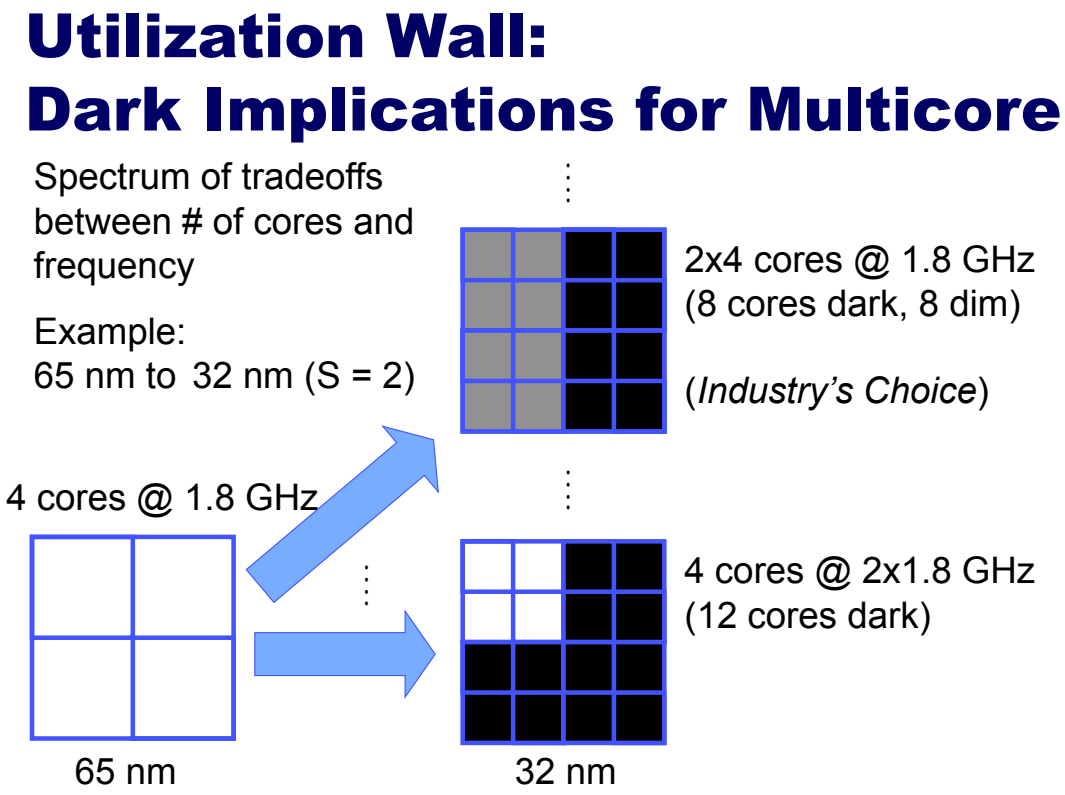


Figure 2.1. Spectrum of multicore designs in the dark silicon regime In newer process generations, architects can choose between more cores running at approximately the same clock frequency (dark silicon), or fewer cores running in bursts at higher clock frequencies (dim silicon). Figure from [GSV+10].

predicted by Dennard scaling, in the dark silicon regime we see only 1.4× improvement—a gap of 2×.

Industry's initial response to the dark silicon problem was switching to multicore designs. Figure 2.1 shows a spectrum of multicore options, with tradeoffs between the number of active cores and the clock frequencies of those cores. Architects can either have fewer cores running at higher clock frequencies, or more cores running at lower clock frequencies (an approach called "dim" silicon). Industry has primarily chosen the dim silicon approach.

Unfortunately, just switching to multicore designs is not a long-term scalable solution

[GSV+10][VSG+10][EBA+11]. Even with optimistic ITRS projections, multicore scaling is limited to a fraction of expected Moore's law gains.

2.2.2 Dark Silicon Solutions

As dire as the dark silicon problem appears, it also creates new opportunities for rethinking traditional computer architecture and chip design. In [Tay12], Taylor describes four potential approaches to handling the influx of dark silicon:

Shrinking Silicon

Per-chip cost is proportional to silicon area, so one approach to dark silicon is to just build smaller, cheaper chips. Although this could provide a short-term cost reduction, the approach has several problems at scale. The cost of the silicon die is just a fraction of the total chip cost after considering packaging, testing, NRE, and other costs. As dies shrink these other costs grow proportionally. Smaller chips are also not always feasible, since smaller chips can quickly become dominated by I/O pad area, which has not scaled as well as transistors. Finally, just building smaller chips forfeits the Moore's law performance benefits architects have come to depend on, meaning an end to "free" performance scaling. For these reasons shrinking chips is likely only a last resort, if we can find no more practical uses for dark silicon.

Dim Silicon

Dim silicon refers to general-purpose logic that is typically underclocked or used infrequently, in order to meet power constraints [SVGH+11][HRSS11]. Some dim silicon techniques include near-threshold voltage (NTV) computing [DWB+10][PSD+12], coarse-grained reconfigurable architectures (CGRAs) [PFM+08], building bigger caches, or temporal dimming techniques such as Intel's Turbo Boost [RNR+11] and computational sprinting [RLC+12].

"Deus Ex Machina" Silicon

In literature, *deus ex machina* refers to a solution that appears suddenly and unexpectedly. It is possible the dark silicon problem will be solved with new breakthroughs in semiconductors or device physics. Industry has a history of inventing techniques for one-time improvements to continue Moore's law scaling, for example high-k dielectrics, metal gates, FinFETs, or upcoming gate-all-around FETs [SAB+06]. However, these one-time breakthroughs are unpredictable and should not be relied upon.

Specialized Silicon

For decades, industry and computer architects have focused on general-purpose computing, relying on new process nodes to deliver more transistors, and using those transistors to build architectural improvements in order to continue performance scaling. A fourth approach to dark silicon is to instead increase the focus on *specialization* and special-purpose computing. Special-purpose accelerators can be 10-1000× faster and more energy-efficient than general-purpose processors running the same workloads [CMHM10]. As chips fill up with dark silicon, building specialized accelerators becomes an attractive option. Architects can put otherwise dark silicon to good use by building accelerators for specific, key workloads, freeing up the power and thermal budgets for additional computations. Effectively, specialization in the dark silicon regime lets architects trade a relatively "cheap" resource (silicon area) for a more valuable one (energy).

2.3 Specialization as a Candle in the Dark

Of the four approaches presented in the previous section, this dissertation argues that the most promising is specialized silicon. Specialized accelerators have been getting increasingly more attention lately because they let architects trade customized silicon area for performance and energy efficiency. At the heart of most accelerators' performance is the fact that architects

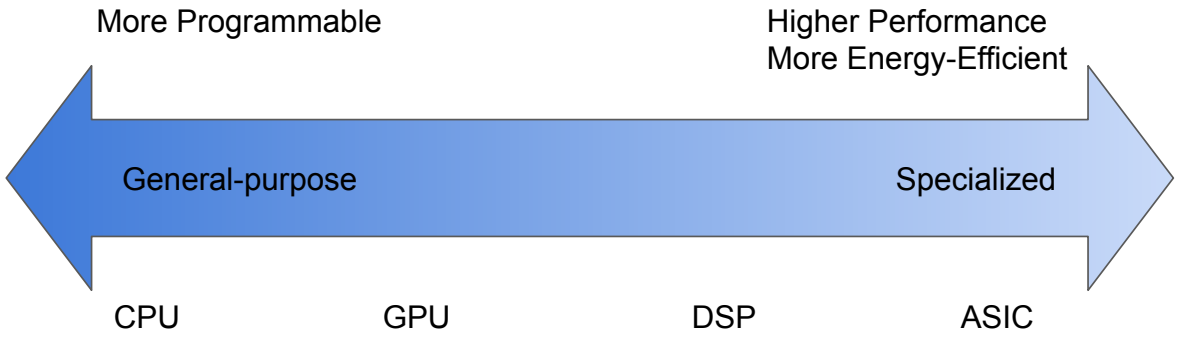


Figure 2.2. Efficiency spectrum of general-purpose versus specialized hardware Specialization presents a spectrum with tradeoffs between generality and efficiency.

have figured out how to attain parallel execution of the underlying algorithm, realized efficiently in hardware.

The challenge in creating accelerators is in reorganizing the algorithm to achieve parallel execution. Being able to do this effectively depends on the availability of exploitable parallelism in the algorithm and the ability to expose this parallelism in the form of an accelerator circuit without errors or excessive effort, complexity, or cost. In particular, creating accelerators for irregular code that's difficult to analyze or lacks parallelism is often challenging, if not impossible.

This section highlights the benefits of specialization as well as challenges, and we show that many of the challenges can be addressed with automation and layers of abstraction.

2.3.1 Benefits of Specialization

Specialized hardware can provide orders of magnitude better performance and energy efficiency compared to general-purpose hardware running the same workload [BVCG04]. General-purpose processors (e.g., CPUs) are flexible but relatively slow and inefficient compared to special-purpose hardware. This comes from extra overheads such as a general-purpose compute engine, complicated instruction fetch and decode logic, reorder buffers, branch predictors, and others. By reducing the scope and targeting a specific workload, specialized hardware can eliminate or reduce many of these overheads.

Specialization is not a binary design decision—rather, specialization presents a spectrum with tradeoffs between generality and efficiency (see Figure 2.2). General-purpose processors like CPUs, and to some extent GPUs, are more programmable and can handle more workloads, at a cost in energy efficiency [FKDM09]. More specialized hardware, such as DSPs, loop accelerators [CHM08] [AS01], and at the extreme end single-purpose ASICs [SAR+00], can run specific workloads with better performance and much better energy efficiency. Some approaches include [KAS+02], [VSL08], [WKMR01], and [YGBT09].

2.3.2 Challenges of Specialization

Specialization is not free from challenges, especially for coprocessor-dominated architectures (CoDAs) with 100s or even 1000s of accelerators [ZGHR+14]. This section describes some of the challenges of specialization and suggests ways to overcome or mitigate them.

Complexity

In general, simpler systems are easier to design, verify, use, and debug. Adding accelerators to a system introduces additional complexity. Hardware complexity comes from increased quantity and diversity of compute engines, each of which requires some form of interconnect (e.g., on-chip network or point-to-point connections) for inter-block communication and access to the memory system. To manage complexity, an accelerator system should use common communication protocols and interconnect when possible, for example ARM's AMBA protocols or Celerity's Tiered Accelerator Fabric [DXT+18].

Accelerators also introduce complexity during the chip design, verification, and manufacturing process. Each accelerator requires its own design and (maybe more importantly) verification efforts. Verification becomes especially tricky if the accelerators interact with each other. Accelerators increase the physical design effort (synthesis, place and route), and may complicate the top-level floorplan, where placement is key to good QoR. Each block also requires DFX (Design For Test and Manufacturability) structures such as scan chains and built-in

self-test (BIST) logic. Some of this increased complexity can be addressed with automation (e.g., high-level synthesis and high-level verification [KLG14]), and by using common interfaces between accelerators.

Programmability

A system with heterogeneous accelerators can be difficult to program. Many accelerators support the C or C++ programming languages, but accelerators may also support more targeted languages or language subsets. For example, Nvidia GPUs can be programmed with OpenGL, OpenCL, or CUDA, while DSPs like Qualcomm's Hexagon can be programmed with domain-specific languages such as Halide [hal]. In addition to needing to learn a new language, accelerator programmers must also acquire a deep understanding of the hardware architecture in order to extract maximum performance. Help for these challenges may come from smarter, advanced compilers, and accelerators may use translation layers to hide some complexity from programmers (see Section 5.3.2 for examples).

Scalability

Dark silicon designs will some day require 100s or 1000s of accelerators [ZGHR+14], but scaling a chip to include so many coprocessors is a significant challenge [CGG+12]. The hardware for each accelerator must be designed and implemented. Part of this burden can be reduced through the use of automation and high-level synthesis tools (see Section 3.6 for an example). Each coprocessor also needs access to a general-purpose host CPU and the memory system. Section 3.1 describes how tiled architectures with replicated CPUs and a distributed memory system may help.

Limits on Efficiency Gains

The benefits of any optimization are limited by Amdahl's law [Amd67], and specialized architectures are no exception. Performance and efficiency gains from accelerators are limited by the fraction of the workload that *can't* run on the accelerators, as well as overhead required to

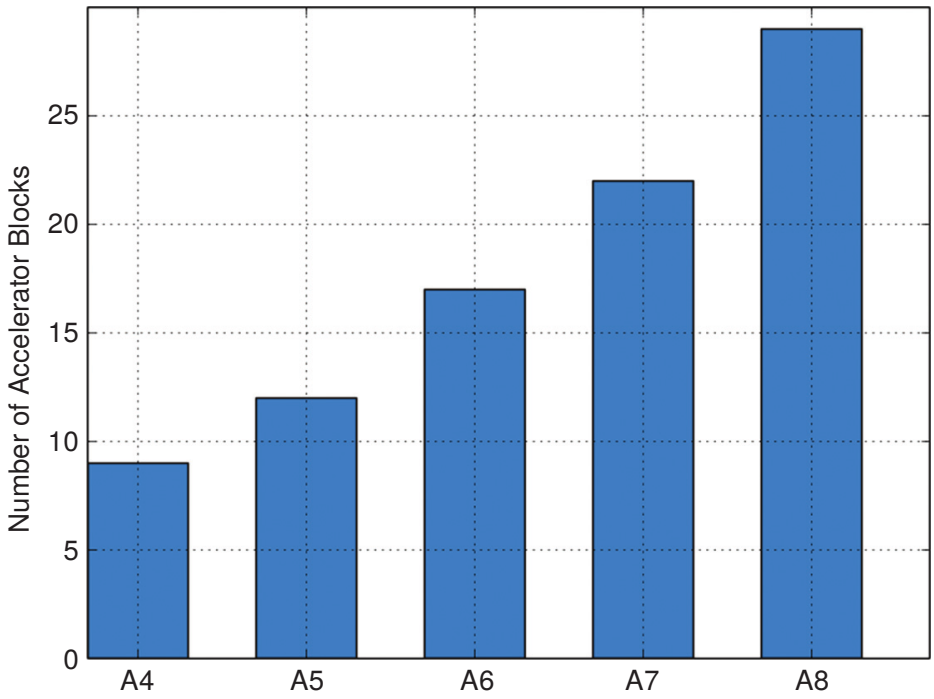


Figure 2.3. iPhone accelerator count The number of accelerators in Apple's iPhone SoCs has been rising exponentially. Figure from [SRWB15].

set up execution and transfer data. For example, even if 99% of a workload can be accelerated by 1000×, the remaining 1% of the workload limits the total system performance improvement to just 91×. Clearly, "nines matter," and the best way to get more nines is to target as much code as possible—not just regular, parallel code, but irregular, hard-to-parallelize code too. Chapter 3 presents one approach to do this.

2.3.3 Predictions Come True: Industry Trends

The author's initial research in dark silicon ([GSV+10][VSG+10][GHSV+11]) predicted a significant increase in the use of specialization and hardware accelerators. Ten years later we see that those predictions have come to pass in industry. Today, industry's mobile application processor SoCs couple multicore CPUs with GPUs and dozens of specialized accelerators. For example, Apple started building its own in-house A-Series SoCs, with more and more specialized accelerators every year. Figure 2.3 shows how this accelerator count has indeed been rising

exponentially. Today these accelerators offload a multitude of functions such as graphics, digital signal processing, multimedia encode/decode, cryptography, security, image signal processing, machine learning, I/O, and more ([Qua19][App19][Sam19][Hua19]).

In modern application processors, the vast majority of die area is now used for accelerators and specialized hardware. For example, of the Qualcomm Snapdragon 845's 95-mm^2 die, just 12% of the silicon area is occupied by the 8-core CPU and L3 cache [YW18], while the rest is reserved for GPU, I/O, modem, and accelerators.

2.4 Summary

This chapter has described the origin of dark silicon and its dark implications for chip design. We have shown how hardware specialization is a viable option for continued performance scaling in the dark silicon regime. Indeed, industry has agreed, by dedicating more and more silicon area to special-purpose accelerators in modern application processor SoCs.

The next two chapters present the author's work on an energy-saving specialized architecture called conservation cores, and then show how conservation cores can be built for Android in an application processor called GreenDroid.

Acknowledgements

This chapter contains material from "Conservation Cores: Reducing the Energy of Mature Computations," by Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor, which has appeared in the Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ©2010 ACM. The dissertation author is a primary contributor and third author of this paper.

This chapter also contains material from "GreenDroid: A Mobile Application Processor for a Future of Dark Silicon," by Nathan Goulding, Jack Sampson, Ganesh Venkatesh, Saturnino

Garcia, Joe Auricchio, Jonathan Babb, Michael Bedford Taylor, and Steven Swanson, which has appeared in Hot Chips 22: A Symposium on High Performance Chips, ©2010 IEEE. The dissertation author is a primary contributor and first author of this paper.

Chapter 3

Conservation Cores

In the previous chapter we explained the dark silicon phenomenon and suggested specialized architectures as a path forward. This chapter introduces one such specialized architecture, *conservation cores* [VSG+10]. Conservation cores, or *c-cores*, are highly specialized, application-specific hardware circuits derived automatically from program source code, with a primary goal of reducing energy. Whereas traditional accelerators focus on improving performance, at a potentially worse, equal, or better energy efficiency, conservation cores focus primarily on reducing energy and energy-delay, with performance acceleration as a secondary goal.

This chapter presents an overview of c-core-based systems and then explains the c-core architecture and programming model. We describe the ability of c-cores to stay useful even as the source code evolves, through a set of generalized datapath operators and firmware-patching mechanisms. We also describe our toolchain for automatically generating c-core hardware directly from source code (nearly unrestricted C), along with simulation models and synthesized, placed, and routed netlists.

3.1 System Overview

A c-core-based system includes many c-cores embedded in a multicore tiled array like the one shown in Figure 3.1(a). Each tile in the array contains a general-purpose processor (CPU), cache, on-chip network connection, and a collection of c-cores. The c-cores execute "hot"

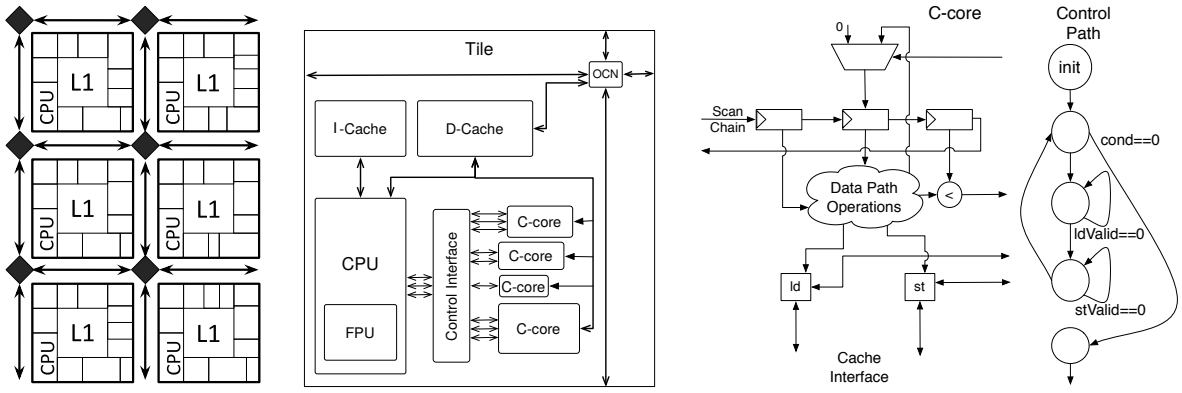


Figure 3.1. Organization of a c-core-based system A c-core-based system (a) comprises multiple individual tiles (b), each of which contains multiple c-cores (c). Conservation cores communicate with the rest of the system through a coherent memory system and simple scan-chain-based control interface to a general-purpose CPU. Different tiles may contain different c-cores or other accelerators.

regions of specific applications that represent significant fractions of the target workload. The CPU serves as fallback for the parts of applications that are not supported by any c-cores. Using a tiled architecture ([SMSO03][TLM+04][SNH+03]) ensures each c-core is close to a host CPU, to minimize execution overheads and distribute access to the memory system.

Within a tile, the c-cores are tightly coupled to the host CPU via shared connection to the L1 data cache, and by a collection of architecturally-visible scan chains that allow the CPU to read and write all register state within each c-core (Figure 3.1(b)). Each c-core consists of a fixed compute datapath under the control of a hardware state machine (Figure 3.1(c)).

Figure 3.2 shows the life cycle of a c-core from program source code to hardware accelerator in a many-core processor: (a) The life of a c-core starts with C code from a set of relatively stable target applications. (b) The c-core toolchain is used to profile the target applications and extract the energy-intensive code regions. The toolchain converts these regions into patchable c-core specifications. (c) The specifications are used to generate synthesizable Verilog and simulation models for each c-core. (d) Sets of c-cores are placed on different tiles in a many-core tiled processor. (e) As new versions of the target applications are released over time,

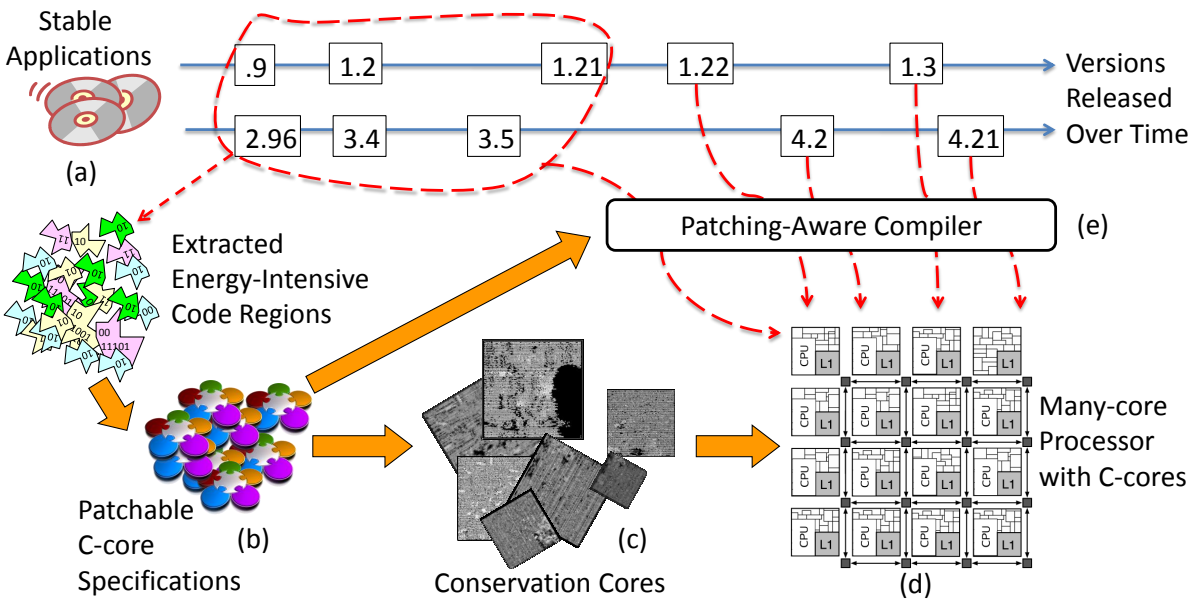


Figure 3.2. Conservation core life cycle The life of a c-core begins with program source code and results in a patchable hardware accelerator in a many-core tiled processor.

the toolchain can analyze the differences between versions and patch the firmware of existing c-cores, extending the useful lifetime of the c-cores.

3.2 C-core Architecture

This section describes the conservation core architecture in more detail. First we discuss the original, baseline implementation of c-cores. Then we discuss upgrades to the baseline architecture that improve performance and generality.

3.2.1 Baseline C-core Architecture

Each c-core serves as a drop-in replacement for a piece of code, specifically a leaf function in the target workload. The principle components of a c-core are a computation datapath and control unit, a cache interface, and a control interface to the CPU.

By design, the c-core datapath and control unit very closely resemble the internal representation that our toolchain extracts from the C source code. The code's data flow graph (DFG) serves as a blueprint for the c-core's datapath, which contains functional operators, such as

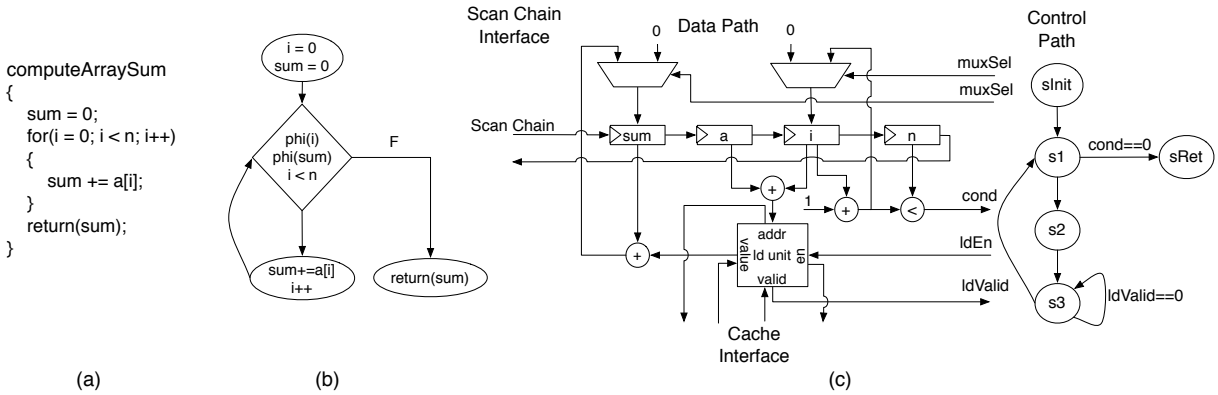


Figure 3.3. C-core example translation from source code An example c-core starting from C code (a), translated into the compiler's internal representation (b), and finally a hardware datapath controlled by state machine (c).

adders, shifters, and comparators, for each mathematical operation in the code. Muxes are used to implement control decisions, and registers are instantiated only when needed, to hold program values across basic block boundaries or between iterations of a loop.

The control unit is a hardware state machine that mimics the original program's control flow graph (CFG). It tracks branch outcomes computed in the datapath to determine which state to enter on each cycle. The control unit sets the enable and select lines on the registers and muxes in the datapath so that the correct basic block is active each cycle. The control unit's state machine also includes self-loops to wait for the results of memory operations.

The first implementation of c-cores enforces memory ordering constraints by issuing at most one memory operation per cycle to a pipelined, in-order cache interface. Both the c-core and the cache block on misses. The load/store units connect to a coherent data cache that ensures that all loads and stores are visible to the rest of the system regardless of which addresses the c-core accesses. Subsequent, improved versions of c-cores (described in Section 3.2.2) allow multiple memory operations to be in flight simultaneously.

Figure 3.3 shows an example translation from C code (a) to internal representation (b) and finally hardware datapath and control unit state machine (c). The hardware corresponds very closely to the DFG and CFG of the sample code. It has muxes for variables `i` and `sum`

corresponding to the ϕ-functions in the CFG. The c-core's state machine is almost identical to the CFG, but with an additional self-loop to wait for the valid signal from the memory load operation. The valid signal is similar to the memory ordering token used in systems such as Tartan [MCC+06] and WaveScalar [SMSO03].

The close correspondence between the program's structure and the c-core is important for two reasons: First, it makes it easier to enforce the correct memory ordering from the original program. The control unit enforces an ordering that corresponds to the same order that the program counter provides in a general-purpose processor, and we use the same ordering to enforce memory dependencies. Second, by maintaining a close correspondence between the original program and the c-core hardware, it is more likely that small changes in the source code (which are the common case) will result in correspondingly small patches to the hardware.

3.2.2 Improvements to C-cores

After the initial design and evaluation of c-cores in [VSG+10], we made significant changes to the c-core architecture to improve their performance, energy efficiency, and generality. Whereas the original implementation allowed at most one memory operation per basic block, upgraded c-cores can combine multiple memory operations along with dozens of datapath operations into large basic blocks called *fat operators*. We employ fat operators in *efficient complex operator cores* [SVGH+11], which extend the baseline architecture with two techniques that improve c-core efficiency: selective depipelining and cachelets. Additionally, the improvements in *quasi-specific cores* [VSGH+11] generalize the c-core datapath and patching mechanisms to support even broader ranges of applications. These techniques are fundamental to the design of c-cores but also apply to any architecture that uses fat operators, such as the "magic" instructions discussed in [HQW+10].

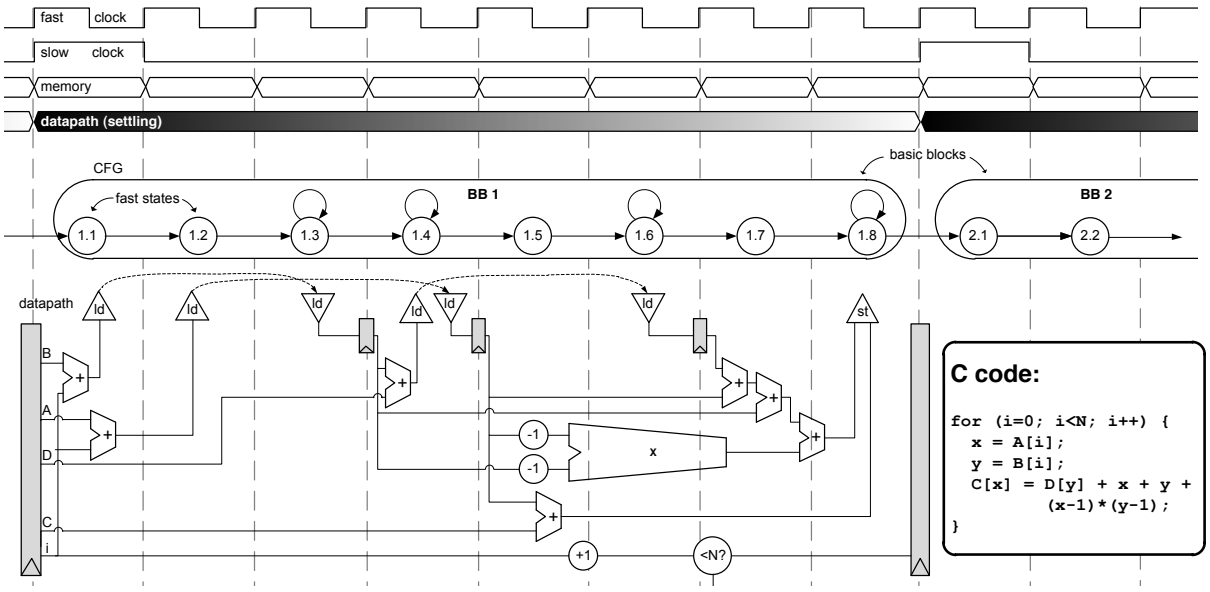


Figure 3.4. Selective depipelining in c-cores An example of selective depipelining (SDP) for one complex basic block, called a fat operator. Under SDP, non-memory datapath operators chain freely within a basic block under the control of a slow clock pulse, while memory operators and associated load registers align to fast clock boundaries.

Selective Depipelining

Selective depipelining (SDP) is a novel pipelining scheme that employs two clocks operating at different speeds: a *fast clock* allows memory requests to operate at a faster clock rate than the datapath, improving memory performance, while a *slow clock* saves power in the rest of the datapath. Figure 3.4 illustrates SDP for one example basic block.

The fast clock effectively replicates the memory interface *in time* (by exploiting pipeline parallelism), while the datapath runs at a slower clock rate, saving power and leveraging instruction-level parallelism by replicating compute resources *in space*. By driving the non-memory datapath with a slower clock, synthesis tools can use smaller and more energy-efficient operators, since these timing paths are permitted multiple fast-clock cycles to settle. SDP also saves energy by removed many registers from the datapath (see Figure 3.5), since only the final live-out values need to be captured, at fat operator basic block boundaries.

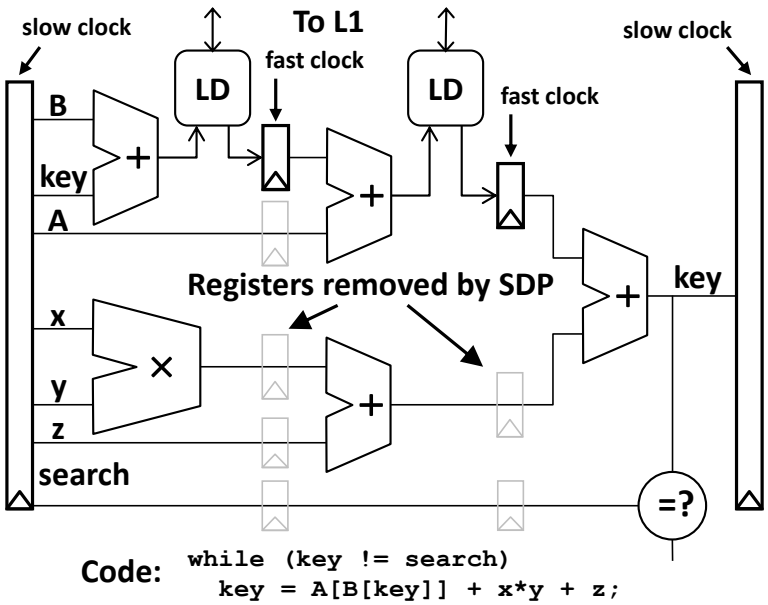


Figure 3.5. Using selective depipelining to remove registers The light-gray boxes highlight registers that are removed with selective depipelining. These intermediate datapath values do not need to be captured, and only the final live-out values are captured in registers by the slow clock, at fat operator basic block boundaries.

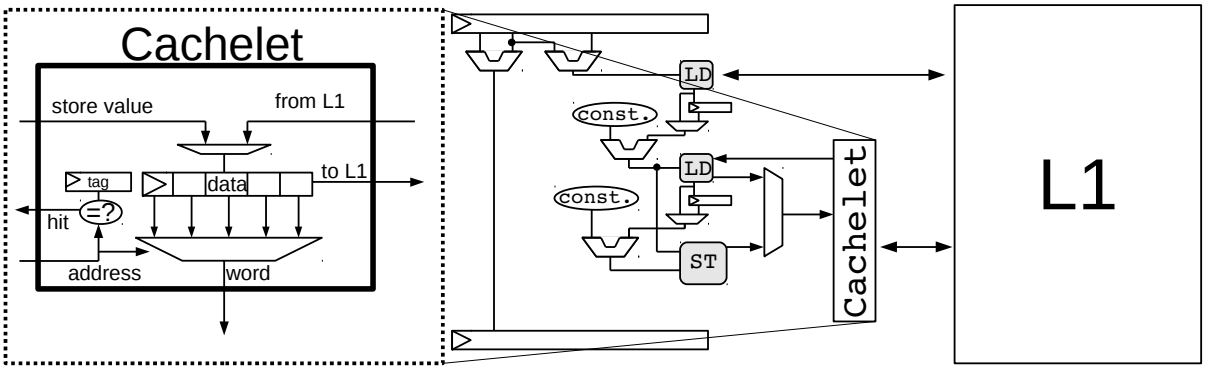


Figure 3.6. C-core cachelet architecture Cachelets are small, distributed L0 caches that improve memory latency in the common case. Memory operations with good locality are mapped to cachelets, while other operations continue to interface directly with the L1 cache.

Cachelets

In a conventional processor all loads and stores go to a single cache, since all memory instructions execute on a small set of load/store functional units. C-cores, however, can optimize load and store operations in isolation through the use of cachelets. Cachelets are small (one- to four-line), very fast, coherent, distributed L0 caches embedded in the c-core datapath to reduce load-use latency. Figure 3.6 shows the internal architecture. Cachelets enable sub-cycle load-use latency in the common case, which is 6× faster than the L1 in our system. For more details please see [SVGH+11].

Quasi-specific Cores

Quasi-specific cores are generalized c-cores built to support multiple, similar code regions across many applications. Quasi-specific cores are based on the insight that similar code patterns exist within and across applications.

To generate these cores, the c-core toolchain analyzes the program dependence graphs (PDGs) [FOW87] of hot spots from multiple applications in the target workload. The toolchain locates similar code segments across hot spots by searching for isomorphic subgraphs across the PDGs and identifying subgraphs that are suitably similar. Next, the toolchain generates generalized datapaths that can be configured to handle each of the subgraphs. The generalized datapaths operate similarly to the c-core patching mechanisms (see Section 3.5).

Quasi-specific cores improve the scalability of c-core-based systems because they increase the total workload coverage while simultaneously decreasing the total number of accelerators needed. For more details please see [VSGH+11].

3.3 Integration with CPU

Traditional accelerators typically execute coarse-grained "jobs" offloaded from the CPU. As long as the jobs are large enough and can run independently, these accelerators don't require

a very tight coupling with the CPU.

In contrast, conservation cores are designed to cover much smaller, fine-grained pieces of code. Execution transfers between c-cores and the CPU much more frequently. To minimize overhead and maximize performance, c-cores are tightly integrated with a general-purpose CPU and share direct access to the CPU's L1 data cache.

3.3.1 Shared L1 Data Cache

Within a tile, c-cores are tightly coupled to the host CPU via a direct, multiplexed connection to the L1 data cache. This enables very quick transitions back and forth between CPU and c-cores, because the cache is already warm and no flushing, refilling, or data transfers are necessary.

A coherent, shared memory interface allows us to construct c-cores for applications with unpredictable access patterns. Conventional accelerators cannot speed up these applications, because they can't extract enough memory parallelism.

Since only one c-core within a tile is active at any time, and since the CPU and c-cores do not simultaneously access the cache, the impact on the CPU's cycle time is minimal, because the c-cores can multiplex in through non-critical, pre-existing paths that are used to handle cache fills.

3.3.2 Control Interface

Apart from the shared cache, the only connection between the CPU and the c-cores is a control interface. The control interface allows the CPU to access all of a c-core's internal state. The control interface is used to transfer input arguments and also to install patches to the c-core firmware (see Section 3.5).

The first version of conservation cores ([VSG+10]) used a simple control interface based on scan chains. A scan chain is a 1-bit wide network that connects every flop of every register (or a subset of registers) serially. Data can be read or written by shifting the entire chain one bit

State Tree Address

Field	c-core id	basic block id	register id
Width	6	13	13
Bit	31		0

Figure 3.7. C-core state tree address format Every c-core register is accessible with a 32-bit address consisting of c-core, basic block, and register IDs.

at a time until the desired word is in place.

Scan chains are typically inserted during the physical design stage of a chip, and they are normally reserved for debug, testing, and screening during manufacture. C-cores give the scan chains an additional purpose, and expose the scan chains at the architectural level. To access the scan chains, we added three new instructions to the CPU's instruction set:

- **MTSC** (Move To Scan Chain): Writes a 32-bit word to the specified scan chain
- **MFSC** (Move From Scan Chain): Reads a 32-bit word from the specified scan chain
- **SCRL** (Scan Chain Rotate Left): Rotates the specified scan chain left by n bits

Scan chains scale well physically in terms of silicon area, since each chain adds only a 3-bit interface to a module (scan-in, scan-out, and scan-enable), and scan chains are already included in the silicon for chip manufacturing and testing. However, accessing data via scan chain is very slow, since an n-bit chain must be shifted for n clock cycles to access every (or any) register. It is also power-hungry, because cycling through an n-bit chain will cause n^2 flop toggles. This may be acceptable for infrequently-accessed scan chains (e.g., for patching firmware), but for this reason the original implementation of c-cores limited performance-critical scan chains such as input arguments to a maximum length of 64 bits.

In [SVGH+11] we replaced the scan chain interface with a pipelined register tree network. The tree network assigns a unique address to every register. This address comprises the c-core ID, basic block ID, and register number within the basic block (see Figure 3.7). To save power

and area, unnecessary address bits are stripped off at each level of the tree.

It takes just 3 cycles to write any register in the tree, and just 6 cycles for reads: 3 cycles to send the address down to the leaf register and another 3 to send the data back up the tree. The tree is pipelined to allow for back-to-back reads and writes, enabling much faster system initialization and reducing the overhead of exceptions and interrupts. This provides both faster access and random access to register state in any c-core.

To access the state tree we added two new instructions to the CPU:

- **MTST** (Move To State Tree): Writes a 32-bit word to the specified tree register
- **MFST** (Move From State Tree): Reads a 32-bit word from the specified tree register

3.4 Programming and Execution Model

When compiling an application or library containing functions that are compatible with a c-core, the compiler inserts stubs that enable the code to choose between running on the c-core hardware or running in software on the CPU. This choice will happen dynamically at runtime.

At runtime, when the application calls the function, the stub checks for an available c-core. If it finds one, the CPU uses the control interface to pass arguments to the c-core and, if necessary, install the correct firmware patch. If no suitable c-core is available, the application falls back to running the original (software) implementation on the CPU. A c-core might not be available if it is already in use by another process, or no longer supports the latest version of the code.

To start execution, the CPU configures each c-core with the necessary firmware. This configuration only happens when the CPU requests a version of the code that is different (newer or older) than that which the hardware is based on. Patching the c-core firmware is typically done at most once per device boot or application launch, for example when the application first starts executing or when a shared library is loaded into memory. C-cores may be patched as

many times as needed, though, for example if multiple applications require switching between different versions of the original code.

After launching execution on a c-core, the CPU can either context switch to another process or save power by going to sleep and waiting for an interrupt from the c-core. A c-core will interrupt the CPU when it is finished, or if it needs to trigger an exception. The most common exception occurs when a c-core needs to fall back to the CPU to perform some computation it doesn't support. This can happen if a newer version of the application requires additional features, or if the control flow changes significantly, such that it's no longer possible or energy-efficient to continue executing on the c-core (see next section for details).

3.5 Patching Support

Although c-cores are created to support existing versions of specific applications and libraries, they also need to support newer versions of the software that are released after the original c-cores are synthesized, or older versions to maintain some backwards compatibility. To do this, c-cores include reconfiguration bits which allow their behavior to adapt to common changes found in programs. This section describes several *patching mechanisms* [Bry09] that allow c-cores to adapt to these software changes, extending the useful lifetime of c-core hardware.

Our analysis of successive versions of our applications revealed a number of common change patterns. In mature software the most common changes are small bug fixes, including: fix an off-by-one error or modify a constant value; insert a new field into a data structure, which shifts existing field offsets; correct an operation error, for example by swapping an addition for subtraction; or insert a new block of code, such as a new conditional or function call.

To support these types of common changes, c-cores include the following patching mechanisms:

- **Configurable constants:** Instead of hard-wiring immediate values, c-cores instantiate configurable registers to support changes to the values of compile-time constants, and the

insertion, deletion, or rearrangement of structure fields, which change the field address offsets. Many of these constants have small absolute values (i.e., most of the upper bits are all zeros or all ones), and changes to field offsets similarly only affect the lower bits. To save area and power, 32-bit constants may have only the lower 8 bits configurable, leaving the upper 24 bits hard-wired with the original program value.

- **Generalized datapath operators:** In hardware, it is relatively inexpensive to generalize some datapath operators. For example a fixed adder can become an add/subtract unit with just one extra configuration bit and minor changes to the logic. C-cores also extend single comparison operations (e.g., less-than) into generalized comparators that can evaluate all six comparisons ($<, \leq, >, \geq, =, \neq$). Similarly, c-cores also generalize individual bitwise operators (e.g., AND or OR) into configurable bitwise ALUs.

- **Control flow exceptions:** In order to support changes to a program's CFG, or to handle more significant datapath changes that can't be handled with the other patching mechanisms, c-cores include a flexible exception mechanism that allows control to fall back to the CPU on any basic block transition. Each transition in the control unit state machine contains a configuration bit that determines whether the c-core should treat it as an exception. When the state machine makes an exceptional transition, it gets a round of applause and transfers control to the CPU. The exception handler extracts current values from the c-core registers via the state tree network, performs the required computation, transfers the new values back into the c-core, and resumes c-core execution. The exception handler can restore control to any point in the CFG, so exceptions can arbitrarily alter the control flow or replace arbitrary portions of the original CFG.

The c-core toolchain employs these patching mechanisms in order to support newer software versions. When a new version is released, the toolchain generates a patch to the c-core firmware using an algorithm described next.

Changes to the software may happen at either the high-level source code or assembly code. In order to handle either type of change and to stay as general as possible, the patching algorithm operates directly on the program's DFG and CFG, since these representations can be generated from either source code or a compiled binary. The goal of the patching algorithm is to generate a firmware patch that will allow the new, *target* software to run on the *original* hardware.

The patching algorithm operates on basic blocks and proceeds in four stages. First, the algorithm identifies which basic blocks in the original hardware are candidates to run the new target code. The patching mechanisms often allow multiple suitable options. Second, the algorithm builds a map between the control flow graphs of the original and target versions. The result is a set of hardware regions that map to basic blocks in the original c-core, as well as a set of software regions that don't map to any part of the c-core hardware and will need to execute on the general-purpose CPU via the exception mechanism. The third stage of the patching algorithm generates a consistent mapping between registers in the original and target basic block pairs. This mapping is used to ensure consistency, and any mismatching hardware regions are converted into software regions if needed. At this point the algorithm has all the information needed to generate a firmware patch. The firmware patch is divided into three sections: configuration bits for each of the generalized datapath operators along with values for configurable constants; exception bits for each control flow edge that passes from a hardware region to a software region; and CPU code to implement any of the remaining software regions.

By supporting multiple past and future versions of applications, the patching mechanisms and patching algorithm extend the useful lifetime of c-cores.

3.6 Toolchain for Automatic C-core Generation

As described in Chapter 2, dark silicon designs will require scaling to 100s or 1000s of accelerators. The key to achieve this level of scaling is automation. Figure 3.8 shows our

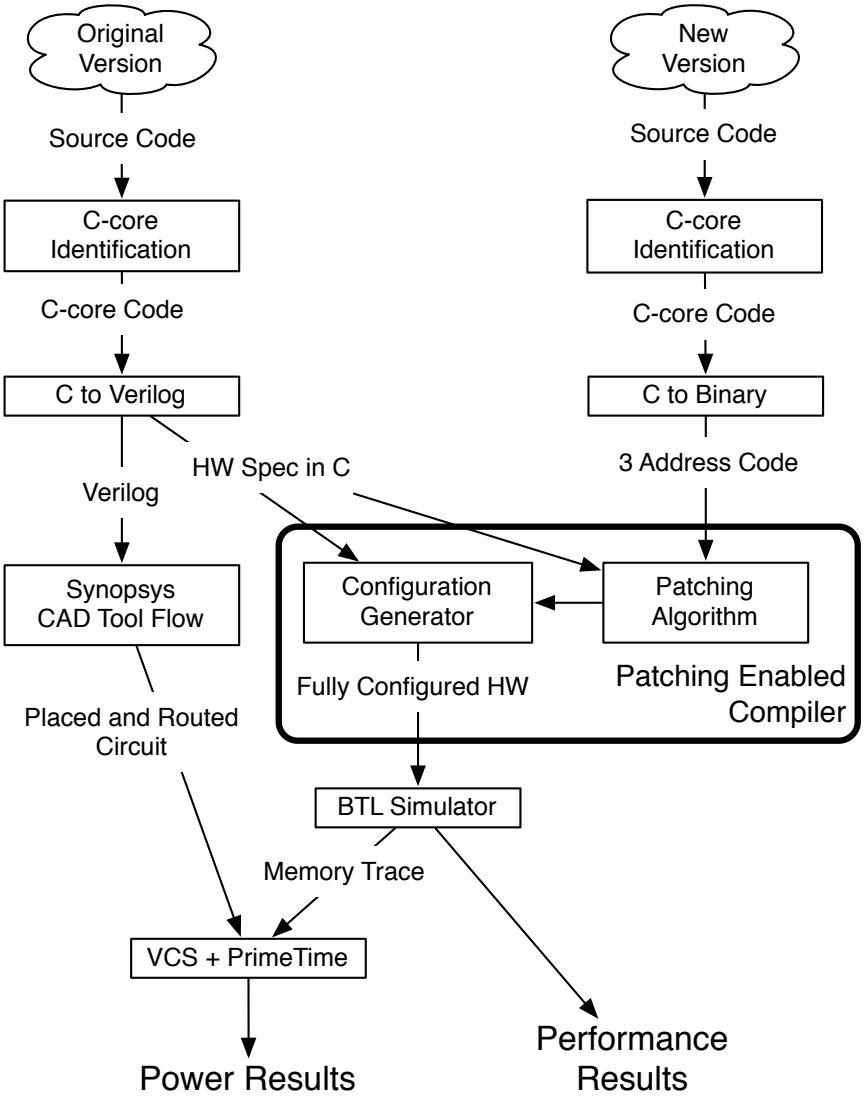


Figure 3.8. C-core toolchain The c-core toolchain compiles C code to RTL Verilog, then runs a Synopsys CAD flow for synthesis, placement, and routing. As new versions of the application code are released, the toolchain can generate patches to the c-core firmware to support them on existing hardware.

toolchain for automatically generating and synthesizing c-cores directly from application source code. This section describes each stage of the toolchain in detail.

3.6.1 C-core Selection

The goal of a c-core is to reduce energy and accelerate hot spots in target applications, while minimizing c-core execution overhead. The overhead is incurred every time a c-core is called, e.g., to pass input arguments and transfer control from the CPU. There is also overhead with each exception in the CFG. To reduce these overheads, the best candidates for c-cores are "fat" leaf functions that spend a lot of time executing within each call.

Because the c-core toolchain can handle nearly unrestricted C, a wide body of code can be converted into c-cores. To achieve the maximum performance and energy savings, relatively stable code is preferred because the c-cores will have to rely on fewer patching mechanisms if fewer software updates are released. Prime candidates for c-cores include stable operating system code and shared libraries, as well as parts of specific applications.

To select a set of c-core candidates, the first step is to profile the target workload applications and identify hot regions of code. If needed, function inlining and outlining can help shape the source code to form suitable leaf functions. The only user input required is a list of function names. Selected functions are then passed into the c-core C-to-Verilog compiler for hardware generation.

3.6.2 Compiler Toolchain

The c-core toolchain is based on the OpenIMPACT (1.0rc4) [ope], CodeSurfer (2.1p1) [Gra], and LLVM (2.4) [LA04] compilers. The compiler accepts a large subset of the C language, including arbitrary pointer references, switch statements, and loops with complex branch conditions. Unlike traditional high-level synthesis tools, which require meticulous programmer annotation and support only limited code structures, the c-core compiler accepts nearly arbitrary leaf functions. The programmer only has to specify a list of function names for

conversion to c-cores, without modifying the code itself.

The heart of the c-core compiler is a custom module added to OpenIMPACT that generates synthesizable Verilog from an internal representation (IR). The IR is a 3-address code with unlimited virtual registers allocated in static single assignment (SSA) form [RWZ88]. SSA is a natural fit for c-core hardware generation: operations in the source code become operators in the datapath, basic block live-out values become registers, and SSA ϕ-functions become multiplexers.

When creating datapaths for a c-core, the toolchain estimates the amount of time required to get through the logic of each fat operator basic block. The timing estimates are based on technology node and standard cell library (see Section 3.6.4). For a given technology node and cell library, we synthesize every pair of chained operators to generate a timing table used for scheduling operators. This table improves timing estimates for fat operators, since the synthesis tools perform cross-module optimizations: for example, the delay through two chained adders will be less than $2\times$ the delay of a single adder.

3.6.3 C-core Simulation

For each c-core, the compiler also generates a C model that can run in the cycle-accurate c-core simulator. The c-core simulator is based on *btl*, the Raw simulator [TLM+04], which models a many-core tiled architecture like the one in Figure 3.1(a). Our modifications to btl include adding the c-core control interface and new instructions (see Section 3.3.2), adding a cache-coherent memory shared among a tile's CPU and c-cores, as well as integrating the c-core simulation models themselves.

The simulator is used to measure total system performance, with and without c-cores. The simulator also records input and output traces that are used to drive Verilog simulation, verification, and power estimation.

Listing 3.1. Example multicycle timing constraint from one c-core

```
set PREFIX tile00/ccMonitor/genblk1_cc_0
set_multicycle_path \
    -from    ${PREFIX}_SRInst1_91_state_reg_*_ \
    -through ${PREFIX}_cpInst/cmp31 \
    -to      ${PREFIX}_${PCREG}_state_reg_*_ \
    -setup   5
```

3.6.4 ASIC Synthesis

The c-core compiler generates synthesizable register transfer level (RTL) Verilog, suitable for ASIC or FPGA CAD flows [ASGH+11][SAGH+11]. The c-core experiments in this dissertation target a TSMC 45-nm GS (General Purpose and High Speed) process [TSMb], using Synopsys Design Compiler (C-2009.06-SP2) and IC Compiler (C-2009.06-SP2). The toolchain runs automated synthesis, floorplan, placement, clock tree, and routing flows. This gives very accurate timing and area results for each c-core.

As described in Section 3.2, c-core datapaths consist of fat operators that can take several or even tens of cycles to settle. This requires specifying complex multicycle timing constraints in the CAD tools during synthesis. The c-core toolchain generates these timing constraints for each c-core. Listing 3.1 shows an example timing constraint that allows up to 5 cycles for a timing path from one live-in data register through a comparator used to determine the next PC. Our largest c-cores have thousands of such timing constraints governing hundreds of thousands of individual paths in the datapath. It would not be feasible to specify so many constraints by hand, but the c-core toolchain generates them automatically.

Specifying multicycle timing paths allows the synthesis tools to use smaller, slower gates. This results in smaller silicon area and lower static and dynamic power.

3.6.5 Power Estimation

In order to estimate power, the c-core simulator captures traces of execution during multiple sampling windows. Each sample starts with a complete snapshot of all register state

in the c-core, and then records all inputs and outputs to the c-core for the next *N* cycles. We use a sampling policy that captures 10,000 out of every 50,000 cycles, and we discard sampling periods corresponding to the initialization phase of an application.

After synthesis and c-core simulation in btl, the toolchain replays the trace from each sample in a gate-level netlist simulation using Synopsys VCS (C-2009.06). VCS generates a VCD waveform activity file, which feeds into Synopsys PrimeTime-PX (C-2009.06-SP2) to estimate static and dynamic power consumed during the sampling window.

Power for other system components is modeled as follows. Processor and clock power values are derived from specifications for a MIPS 24KE processor in TSMC 90-nm and 65-nm processes [TAB+], as well as component ratios for Raw reported in [KTMW03]. We have scaled these values to a 45-nm process, and assume a 1.5-GHz MIPS core running at 0.077 mW/MHz. Finally, we use CACTI 5.3 [TMAJ08] for I- and D-cache power.

3.7 Summary

This chapter presented one example of a specialized architecture, conservation cores, that attacks the dark silicon problem with extreme specialization. Conservation cores allow architects to trade dark silicon area, which has become relatively cheap, for power, which has become relatively expensive. The key to this level of specialization is automation, and we presented our toolchain for automatically generating c-cores directly from program source code. In the next chapter we will apply the toolchain to generate and evaluate c-cores for Android.

Acknowledgements

This chapter contains material from "Conservation Cores: Reducing the Energy of Mature Computations," by Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor, which has appeared in the Proceedings of the 15th International Conference on Architectural

Support for Programming Languages and Operating Systems, ©2010 ACM. The dissertation author is a primary contributor and third author of this paper.

This chapter also contains material from "Efficient Complex Operators for Irregular Codes," by Jack Sampson, Ganesh Venkatesh, Nathan Goulding-Hotta, Saturnino Garcia, Steven Swanson, and Michael Bedford Taylor, which has appeared in the Proceedings of the 17th International Symposium on High Performance Computer Architecture, ©2011 IEEE. The dissertation author is a primary contributor and third author of this paper.

This chapter also contains material from "QsCores: Trading Dark Silicon for Scalable Energy Efficiency with Quasi-Specific Cores," by Ganesh Venkatesh, Jack Sampson, Nathan Goulding-Hotta, Sravanthi Kota Venkata, Michael Bedford Taylor, and Steven Swanson, which has appeared in the Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, ©2011 IEEE/ACM. The dissertation author is a contributor and third author of this paper.

Chapter 4

GreenDroid

In this chapter we apply the conservation core toolchain to target Android workloads, and we show how c-cores can run Android code using 11× less energy than a general-purpose processor. To that end, this chapter introduces *GreenDroid* [GHSV+11], a multicore application processor for mobile devices running Android.

The chapter is organized as follows. First we describe mobile application processors in general, and then demonstrate the suitability of the c-cores approach for accelerating Android workloads. We describe the GreenDroid system architecture, including details about each tile, and then present results from generating c-cores for Android in a 45-nm process. The results include a fully placed-and-routed GreenDroid tile with nine Android c-cores, along with area and energy results. We conclude the chapter with the author's work on a 28-nm, 2x2-tile version of GreenDroid called MiniDroid. We describe details of MiniDroid's on-chip network, pad ring, off-chip communication channels, package, and physical implementation in a 28-nm GlobalFoundries process.

4.1 Application Processors

Heterogeneous, specialized hardware has been demonstrated as an effective approach to scaling the utilization wall ([HQW+10] [SB15] [MKGT16]). Mobile platforms such as smartphones already exploit specialized hardware to address power and performance concerns,

by integrating specialized accelerators along with general-purpose CPUs on one system-on-chip (SoC). Smartphones such as Apple's iPhone and Google Android phones rely on general-purpose *application processors* with scalable performance. These application processors must run an extremely diverse collection of software that relies upon general-purpose software execution models. The utilization wall threatens to limit the performance scaling of these processors, which will impede the evolution of what is becoming the dominant computing platform for much of the world.

The remainder of the chapter demonstrates how GreenDroid overcomes these barriers through the use of conservation cores. GreenDroid's conservation cores allow architects to trade dark silicon area, which has become relatively cheap, for power, which has become relatively expensive, for frequently executed regions of an application processor's workload.

4.2 Android's Suitability to C-cores

GreenDroid targets the Android operating system and mobile platform. At the time of GreenDroid's design in early 2010, Android was still a nascent operating system for mobile phones. It was even questioned if Android would ever be able to compete with Apple's iOS! Today Android is the most popular operating system in the world, powering more than 85% of all new smartphones [Gar18].

Android's software architecture is a good match for the c-cores approach, because it relies heavily on a relatively small number of shared software components, and there is a core set of commonly used applications (such as the web browser, video, music, and navigation programs) that represent common-case usage across a large number of users. Since GreenDroid was first presented in [GSV+10], Google has made a few significant changes to the Android software stack, but the c-cores approach is still applicable. The rest of this chapter focuses on Android at the time of GreenDroid's design.

The core of the Android platform (see Figure 4.1) comprises a collection of native libraries

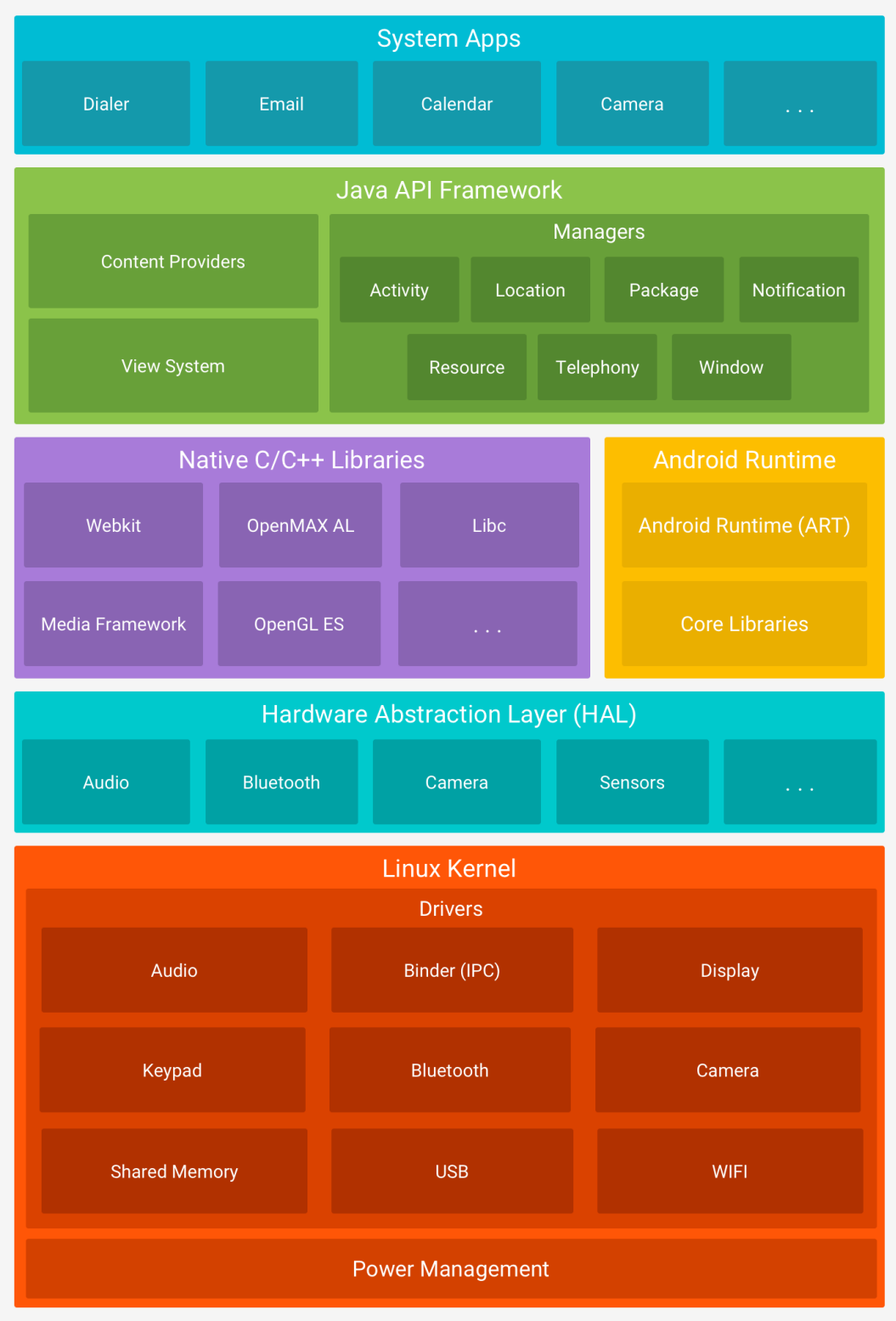


Figure 4.1. Android software stack Android is based on a Linux kernel foundation plus hardware abstraction layer (HAL) supporting the Android Runtime and native C/C++ libraries. These layers support applications that call into Java APIs. Prior to Android version 5.0 (API level 21), the Dalvik virtual machine was the Android runtime. Figure from [Gooa].

written in C and C++ that implement basic services such as window compositing, 2D graphics, 3D graphics, and HTML rendering. This layer also contains the Dalvik virtual machine[1]. At compile time, a translator converts Java .class and .jar files into Dalvik executables. The native libraries are available via Java Native Interface calls. A consequence of this architecture is that much of the performance-critical "hot" code is concentrated in native libraries that applications reuse. The remainder of the code, much of which remains relatively "cold," runs on the virtual machine, making the VM code "hot" as well.

Our research shows that in this system c-cores can be used in two ways. First, c-cores can target key portions of the native libraries and Dalvik, providing energy reduction and program acceleration across the general class of applications that run on the phone. We refer to these shared c-cores as *broad-based* c-cores. In our studies, broad-based c-cores can collectively cover an average of 72% of the execution of an Android mobile phone workload.

Second, c-cores can target key applications that many users run. These *targeted* c-cores can, depending on the silicon area dedicated to them, cover up to 95% of the targeted workload [GSV+10][BGHZ+12]. Although targeted c-cores may not accelerate new Android applications that are deployed after a GreenDroid processor tapes out, the rapid replacement cycle of smartphones suggests it is reasonable to continuously deploy new c-cores into future processors as new applications become popular. Conversely, as applications wane in popularity, targeted c-cores can be removed without affecting backward compatibility, because applications can always fall back to running on the CPU (at a cost of increased energy).

To summarize, Android is well-suited for c-cores for several reasons. First, although many applications are available for download, Android has a core set of commonly used applications. Hot code is concentrated in these applications as well as shared libraries, the virtual machine, and parts of the Linux kernel. Because the hot code is highly concentrated, targeting these components with c-cores lets us attain high coverage over the source base and a significant impact on overall energy usage. Although c-cores support patching, which reduces the impact of

[1] In Android version 5.0, the Dalvik virtual machine was replaced with the Android Runtime (ART) [Goob].

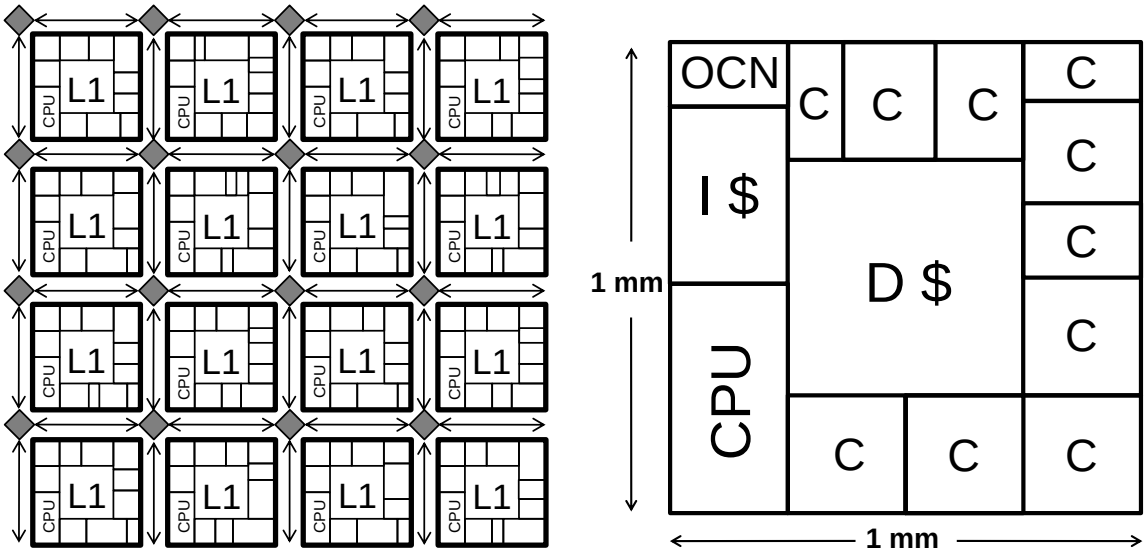


Figure 4.2. GreenDroid architecture and tile floorplan (a) The GreenDroid architecture contains 16 non-identical tiles. (b) Each tile contains components common to every tile: a CPU with instruction cache, on-chip network (OCN) interface, and shared L1 data cache, as well as space for multiple conservation cores of various sizes. Different tiles can contain different c-cores and other accelerators.

post-silicon source code changes, we are also aided by smartphones' short replacement cycle (typically 2-3 years), which lets smartphone chip designers deploy new c-cores regularly to target the latest popular applications.

4.3 GreenDroid Architecture

A GreenDroid processor contains an array of many heterogeneous tiles with dozens of conservation cores and other types of accelerators, as well as associated I/O and interface logic. This section describes the architecture at the system level and at the tile level.

4.3.1 System Architecture

As shown in Figure 4.2(a), a complete GreenDroid chip comprises many tiles. The majority of tiles contain c-cores targeting different hot spots in the Android software stack (see Section 4.4). Some tiles are also reserved for other, traditional accelerators. These include

dedicated hardware for audio and video processing, JPEG image compression, and 2D graphics acceleration. An integrated GPU accelerates 3D graphics computation. GreenDroid also includes standard I/O, memory, and camera interfaces, including PCIe, DDR, USB, Flash, and MIPI.

Collectively, the tiles in a GreenDroid system exceed the system's power budget. A GreenDroid chip must incorporate aggressive clock gating and power gating to reduce energy consumption. In the dark silicon regime, the majority of the tiles—and even the majority of components on an active tile—will be power-gated most of the time.

As a tiled architecture, GreenDroid distributes execution to multiple threads or processes running on different tiles. The on-chip network is similar to the network in the Raw microprocessor [TLM+04]. It is a point-to-point mesh interconnect, used for memory traffic and synchronization. This interconnect enables message passing between tiles and facilitates parallel execution of multiprocessor code across tiles.

Across the tiles, c-cores are clustered on the basis of profiling Android workloads, examining both control flow and data movement between code regions. Related c-cores are placed on the same or nearby tiles, and in some cases c-cores may be replicated. As described in Section 3.4, at runtime an application starts on one of the general-purpose CPUs, and whenever the CPU enters a hot-code region, execution transfers to the appropriate c-core. Execution moves from tile to tile on the basis of the applications that are currently active and the c-cores they use [Ric11].

4.3.2 Tile Architecture

Each GreenDroid tile contains a general-purpose host CPU, L1 data and instruction caches, an interface to the on-chip network (OCN), and a collection of conservation cores and other accelerators. The c-cores share access to the D-cache with the CPU (see Section 3.3 for details).

Each tile contains a different collection of c-cores and accelerators, though most tiles follow a common floorplan template, shown in Figure 4.2(b). In a 45-nm process each tile is

1 mm^2 and contains: general-purpose CPU (0.09 mm^2), on-chip network (0.03 mm^2), 16 KB of I-cache and 32 KB of D-cache (0.30 mm^2 total), and a set of c-cores (0.58 mm^2). More than half of the silicon area is reserved for Android c-cores.

The tile CPU is a low-power, 32-bit, single issue, 7-stage MIPS processor [TAB+]. The CPU has an integer multiplier unit and a pipelined single-precision floating point unit (FPU) that can also be accessed by the c-cores [Mar10]. The frequency target of 1.5 GHz is set by the cache access time, and is reasonably aggressive for a 45-nm design.

4.4 Generating C-cores for Android

As described in Section 4.2, Android is well suited to the c-cores approach because c-cores can target hot spots across the entire Android software stack. The hot spots include portions of the Linux kernel, the Dalvik virtual machine (or its replacement the Android Runtime), other hot spots in the Android core libraries, and key portions of important applications.

To create our list of candidate c-cores, we performed a detailed trace-based analysis of an Android system to identify system hot spots. The traces include hot spots across the entire Android system, running a typical user-level workload that spans a diverse set of applications including the Google Browser, Gmail, Google Maps, Google Music, Google Video, Pandora, Photo Gallery, Photoshop Mobile, and others. The results show that a relatively small amount of code accounts for a large fraction of dynamic execution, and that there is significant intersection between application hot spots.

This level of reuse drives down the amount of static code c-cores need to cover in order to achieve good overall coverage of dynamic execution. In a 45-nm process, 7 mm^2 of silicon dedicated to c-cores will implement approximately 43,000 static instructions and cover 95% of dynamic execution across our workload (see Figure 4.3). Of this 95%, approximately 72% of the code is library or Dalvik code shared between multiple applications in the workload, suitable for conversion into broad-based c-cores.

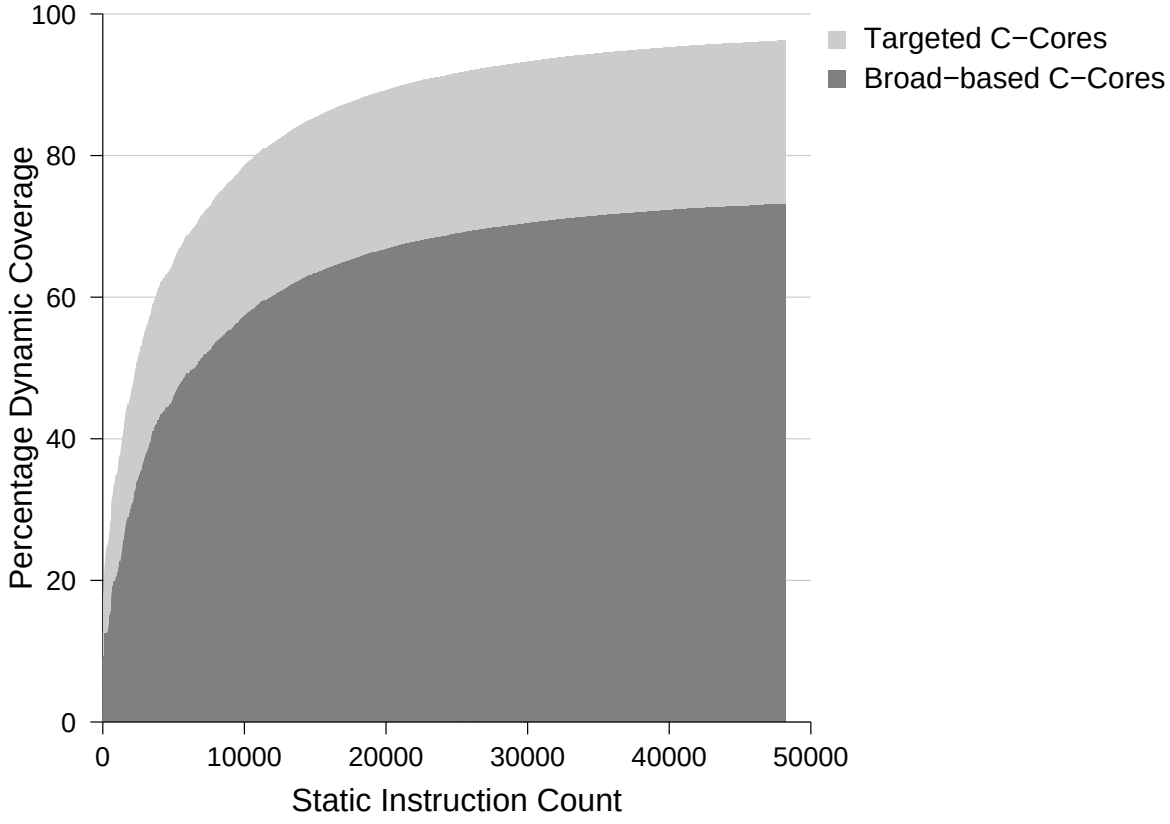


Figure 4.3. Android dynamic execution code coverage Dynamic execution coverage for a typical Android workload. 95% of execution can be covered with a combination of broad-based and targeted c-cores built from 43,000 static instructions.

A tile-based, c-core-based approach allows for alternative tradeoffs between silicon area and code coverage. For instance, 10,000 static instructions provide 76% coverage, and 20,000 static instructions provide 87% coverage. Thus, even as diversity in commonly used Android applications grows, there are tradeoffs that can be exercised between per-application (targeted) coverage and broader coverage across multiple applications.

4.5 Placed-and-Routed GreenDroid Tile

In order to prove the GreenDroid concept and to get more accurate area and power estimates, we designed one complete GreenDroid tile using the c-core toolchain and 45-nm synthesis flow described in Section 3.6.

First, we identified a list of candidate functions from our Android profiling exercise in Section 4.4. Based on area estimates we converged on nine c-cores that collectively cover 10.6% of execution (see Table 4.1). Seven of these functions come from *libskia*, a 2D graphics library that provides window compositing, object rendering, and geometry calculations for most Android applications. The other two c-cores come from a JPEG decompression library and a fast Fourier transform (FFT) function.

We used our fully automated c-core toolchain [Jia13] to generate and synthesize RTL Verilog for each c-core. Then, starting with the floorplan template in Figure 4.2(b), we placed and routed all nine c-cores, along with the CPU, I- and D-cache, and on-chip network. The result is shown in Figure 4.4. Collectively the c-cores occupy 0.58 mm^2 and run at 1.57 GHz on average.

To measure the energy savings of GreenDroid c-cores, we ran post-route gate-level netlist simulations in VCS and PrimeTime-PX, using the methodology described in Section 3.6.5. We compared the energy of functions running on the c-cores versus running in software on the MIPS CPU. Figure 4.5 shows the comparison and energy breakdown. Compared to the MIPS core's 91 pJ per instruction, on average the c-cores use just 8 pJ—an improvement of 11×.

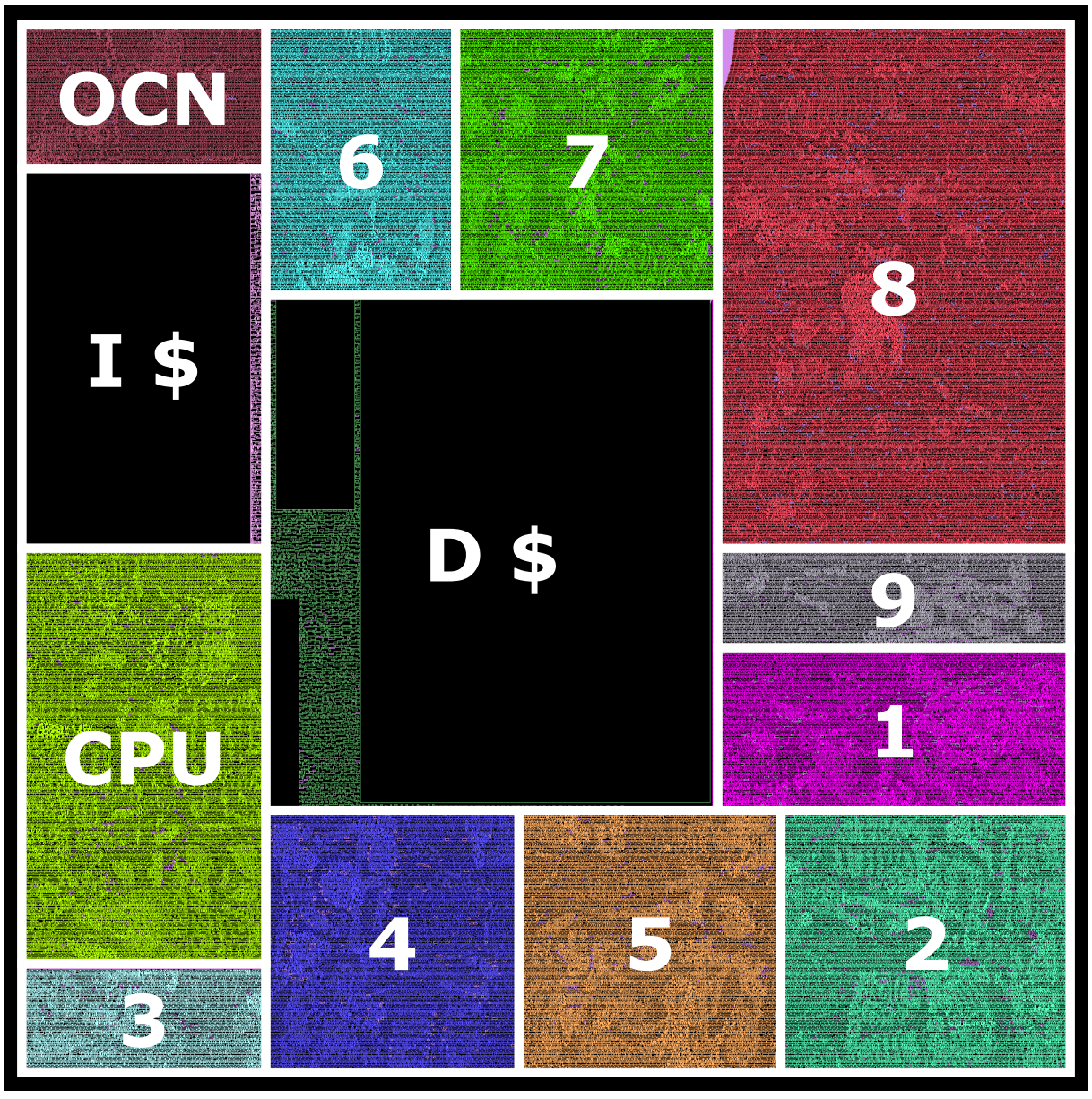


Figure 4.4. Placed-and-routed GreenDroid tile with 9 Android c-cores The layout for one GreenDroid tile includes an interface to the on-chip network (OCN), a general-purpose MIPS CPU with 16-KB I-cache, and a 32-KB D-cache shared with and surrounded by 9 c-cores generated from Android source code (see Table 4.1). This tile is 1.0 mm^2 in a 45-nm process.

Table 4.1. Android c-cores generated for one GreenDroid tile This list of c-cores appears in the tile layout in Figure 4.4. The c-cores cover functions in libskia (a 2D graphics library), JPEG image decompression, and a fast Fourier transform. Collectively the c-cores cover 10.6% of dynamic execution in 0.58 mm^2 of silicon.

No.	Description	Android function	Dynamic execution coverage (%)	Static instr. count	Size (mm^2)
1	Dithering function	S32A_D565_Opaque_Dither	2.78	80	0.052
2	Downsampling	S32_opaque_D32_filter_DXDY	2.20	86	0.070
3	Bit-block image transfer subroutine	S32A_Opaque_BlitRow32	1.15	96	0.024
4	Render with overlay	Sprite_D16_S4444_Opaque::blitRect	1.11	96	0.059
5	Saturating downsampling	Clamp_S16_D16_filter_DX_shaderproc	0.80	97	0.063
6	Fast Fourier transform	fft_rx4_long	0.76	138	0.066
7	Image format conversion	ycc_rgba_8888_convert	0.61	92	0.046
8	Lempel-Ziv decompression routine for GIF files	DGifDecompressLine	0.59	334	0.168
9	Image format conversion for dithering	Sample_Index_D565_D	0.57	67	0.032
Sum			10.57	1086	0.580

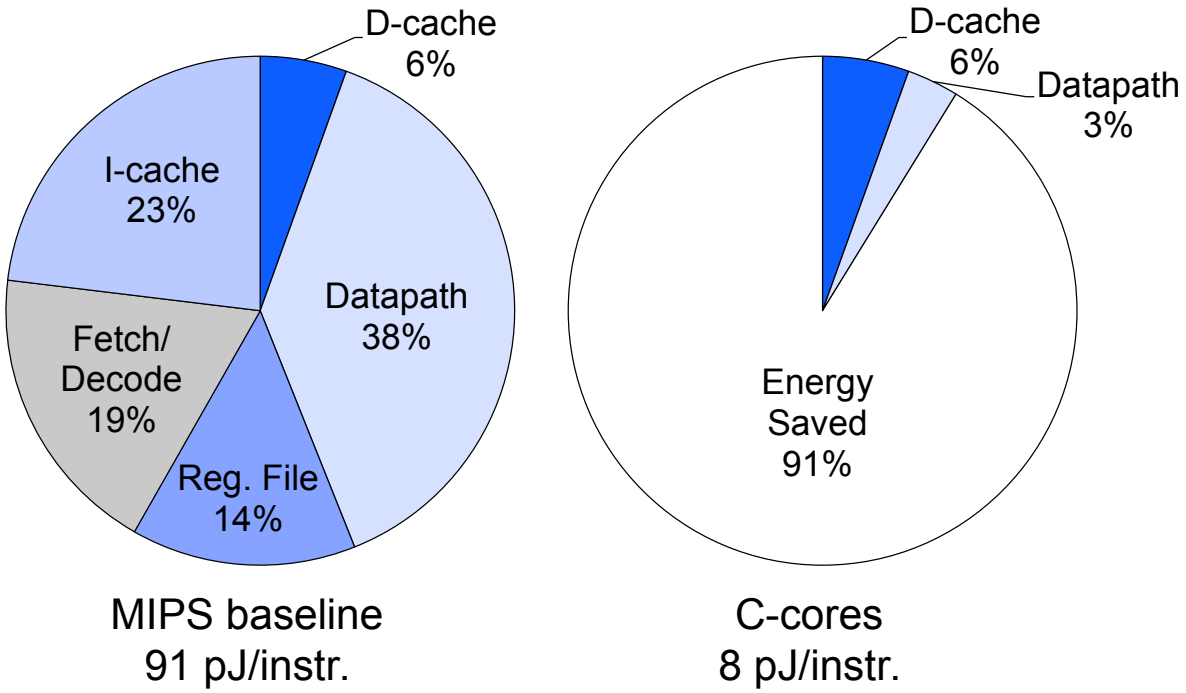


Figure 4.5. Energy savings in c-cores compared to CPU C-cores save energy by eliminating hardware structures like instruction fetch and decode logic, and by having much more energy-efficient datapaths than a general-purpose processor. The c-cores use 11× less energy on average. [GHSZ+12]

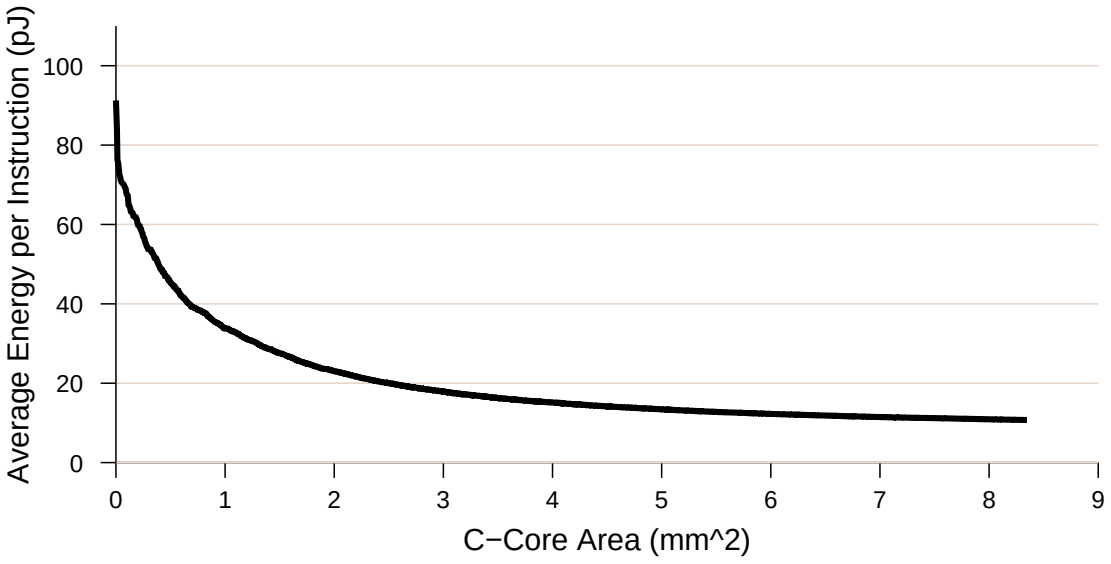


Figure 4.6. Energy vs. area tradeoff for GreenDroid c-cores As more area is dedicated to GreenDroid c-cores, the average energy per instruction continues to decrease. This allows architects to trade area for energy in the dark silicon regime.

The energy savings come from several sources. First, c-cores don't execute software instructions, so they don't access the I-cache and don't require logic for instruction fetch and decode. Second, instead of using a conventional (and power-hungry) register file, c-cores only instantiate registers where they are needed (to preserve values across basic block boundaries), and most of these registers are only activated once per basic block or memory operation. Finally, the c-core's ASIC datapath composed of fat operators is much more energy-efficient than the CPU's general-purpose ALUs.

It should be noted that the CPU and the c-cores require the same D-cache energy, because they both perform the exact same memory operations. Figure 4.5 illustrates the importance of Amdahl's law limits on total energy savings: for non-cache computation energy, the difference in consumption is more than $21\times$.

At the system level, the total energy savings depends on how much of the workload runs on c-cores, which in turn depends partly on how much silicon area is made available for c-cores.

Figure 4.6 shows the average energy per instruction as a function of c-core area. As additional silicon area is made available, more and more of the workload can be offloaded from the CPU. Smaller chips can get significant savings from just 1-2 mm^2 of c-cores, while larger chips with bigger area budgets can save even further. This is a practical example of trading area for energy savings in the dark silicon regime.

4.6 GreenDroid in 28 nm: MiniDroid

The dissertation author collaborated with researchers at the University of California, Santa Cruz, to develop a 28-nm prototype of GreenDroid in partnership with GlobalFoundries, as part of the Multi-University Research Network (MURN) project [DR12]. MURN's goal is to provide a silicon framework that enables researchers to more easily prove architectural concepts through chip fabrication. MURN helps academic institutions gain access to state-of-the-art foundry technology.

As part of this effort, the dissertation author built a 2x2-tile version of GreenDroid called *MiniDroid*. The work included synthesis, layout, and physical design for MiniDroid in a brand new 28-nm process at GlobalFoundries. The author also developed extensions to a new VLSI design flow called Catalyst [Dic12].

4.6.1 Chip Architecture

A MURN chip begins with a baseline silicon platform, shown in Figure 4.7. The baseline platform includes logic and pads for off-chip I/O, a clock management unit with PLL, clock-gating and power-gating infrastructure, and access to an on-chip ring network. Researchers then provide their own design nodes (custom blocks or IPs) that connect to the ring network following the MURN protocol. This allows multiple institutions to reuse a common infrastructure and share the costs of chip fabrication. A MURN chip can be thought of as a heterogeneous architecture with a common communication network.

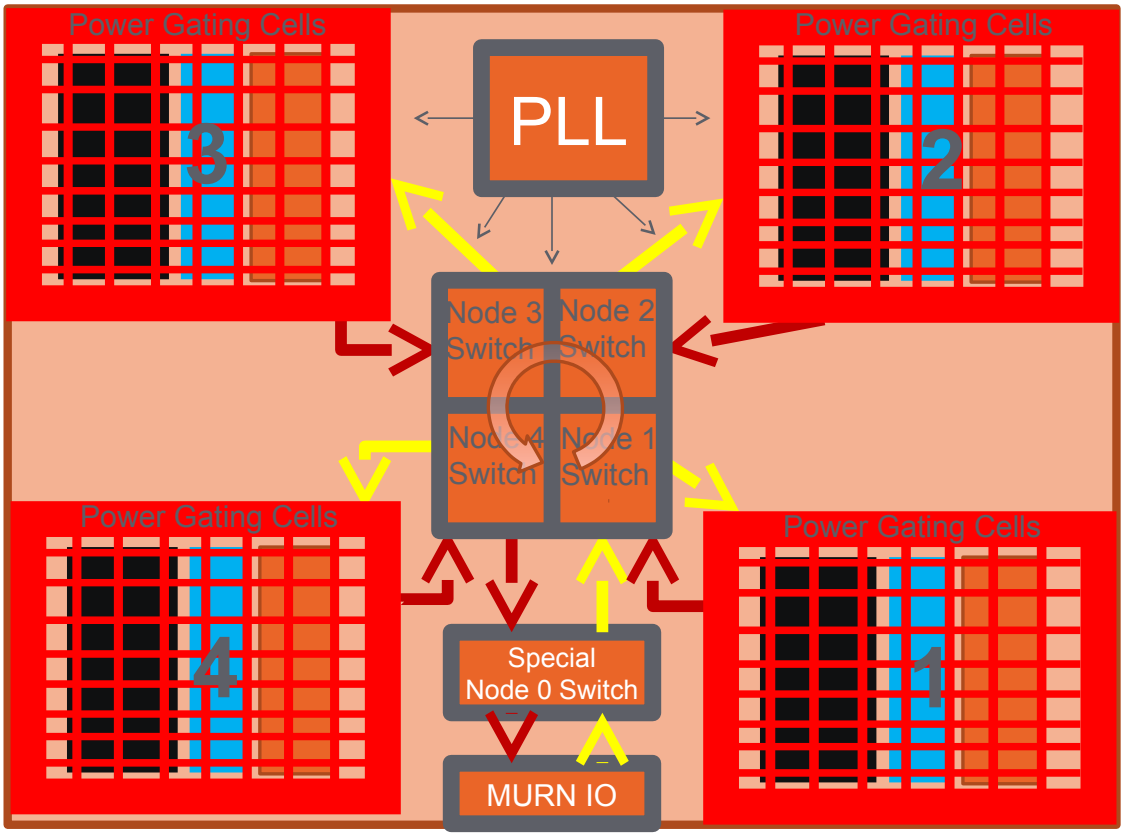


Figure 4.7. MURN conceptual diagram with on-chip ring network A MURN chip comprises shared I/O, pads, clock, and on-chip ring network, as well as multiple design nodes (IPs) from individual researchers. Each design node can be independently power-gated. Figure from [DR12].

MURN Network Packet Format

Field	data	opcode	cmd	dest id	src id
Width	64	7	1	4	4
Bit	79				0

Figure 4.8. MURN network packet format MURN network packets consist of source and destination node IDs, a 1-bit command field indicating if the packet is intended for the design node (0) or network switch (1), an optional 7-bit design opcode, and a 64-bit data payload.

MURN Network

The on-chip network is a unidirectional ring comprising network switches that connect the MURN I/O block and every design node on the chip. The network routes packets containing data or commands between the design nodes. The network also handles power management, reset, and enable/disable for each design node.

Each network switch has a unique ID. ID 0 is reserved for the I/O block, and the other IDs are assigned to the design nodes. Figure 4.8 shows the format of MURN network packets. Each packet contains 4-bit source and destination IDs. The `command` bit indicates if the packet is intended for the design node (0) or for the ring network switch attached to the node (1). The optional `opcode` field is design-dependent and can be used to send predefined commands to the design node or network switch. Each packet can also carry up to 64 bits of data.

The network enables power management for all of the design nodes. Each design node has its own independent power domain, which can be power-gated when the design is not in use. Most of the nodes will be powered down most of the time, but the network switches remain active so the ring is always intact.

I/O Block

The I/O block handles all off-chip communication and translates between internal ring network packets and external data transmitted over four independent 8-bit data channels. The channels are bidirectional and use a simple valid/ack protocol with a source-synchronous clock.

Table 4.2. MiniDroid chip pads The MURN pad ring has 250 pads arranged as two concentric rings along the perimeter of the die. The left and right sides of the die have 62 pads each, while the top and bottom sides have 63. Half the pads are used for power and ground.

# Pads	Purpose
88	Four bidirectional data channels
16	Reset, clocks, PLL
16	Spare I/O and test outputs
5	JTAG
40	Core V_{dd} (1.0 V)
37	Core V_{ss}
25	I/O V_{dd} (1.8 V)
23	I/O V_{ss}
250	Total

Each channel can be enabled independently. Having multiple independent channels provides several benefits. Channels can be reserved for traffic from specific design nodes, or pooled together to achieve higher bandwidth. Having multiple channels also increases redundancy in the case of some hardware failures. For example, if there is a defect in the package, wire bonds, or silicon die (common in a brand new process), defective channels can be disabled (at a cost of lower off-chip bandwidth).

Pad Ring

The chip pads are arranged in two concentric rings along the perimeter of the die (see Figure 4.12). The outer ring contains power and ground pads that provide power to the core logic (Core V_{dd} and Core V_{ss}) as well as the pad ring (I/O V_{dd} and I/O V_{ss}). The inner ring contains input and output pads for data signals, which are interleaved with additional, unbonded power and ground pads to provide shielding for better signal integrity.

Table 4.2 lists the pad allocation. Half of the 250 pads are used for power and ground (core and I/O), and the other half are used for data signals. Each of the four off-chip I/O channels uses 11 pads in each direction: 8 data bits, 1 valid, 1 acknowledge/response, and one pad for the source-synchronous clock. Other I/O pads are used for global clocks and reset, PLL, and JTAG.

Package

The chip package is a plastic ball grid array (PBGA) from BroadPak. The PBGA has four layers for routing signals and power to the die: 1. a top-level 50-Ω impedance routing layer; 2. a V_{ss} ground plane; 3. a V_{dd} power plane, split into sections to supply Core V_{dd} and I/O V_{dd}; 4. another routing layer, also used for ball attachment. The package is 15x15 mm in size.

Figure 4.9 shows the internal layout of the package. The center cavity holds a 3x3-mm silicon die. Surrounding the die cavity are a V_{ss} ring (shown in green), a V_{dd} ring (red), and wire bond pads for individual I/O signals (yellow).

Figure 4.10 shows a detail view. The power and ground pads from the die's outer pad ring are wire bonded with short hops to the power and ground rings in the package. Then the I/O signals are wire bonded from the die's inner pad ring, *over* the power pads and package rings, to the individual signal bond pads in the package. These signals are brought out to the package balls via the package's routing layers.

4.6.2 Catalyst CAD Flow

Catalyst [Dic12] is the MURN synthesis and physical design CAD flow. The Catalyst flow takes a design node from RTL to GDSII, for inclusion in a MURN chip. Users provide a design in Verilog or SystemVerilog RTL along with a set of timing and physical design constraints.

Timing constraints include clock specifications (period, duty cycle, skew, uncertainty), input and output port delay, operating conditions (e.g., process/voltage/temperature corners), and any timing exceptions such as false or multicycle paths, which are especially important for c-cores (see Section 3.6.4).

Physical constraints start with an optional floorplan. For small or simple designs, the user can just specify an aspect ratio and target cell utilization. For more complicated designs such as MiniDroid, the user can provide a complete floorplan with explicit macro placement,

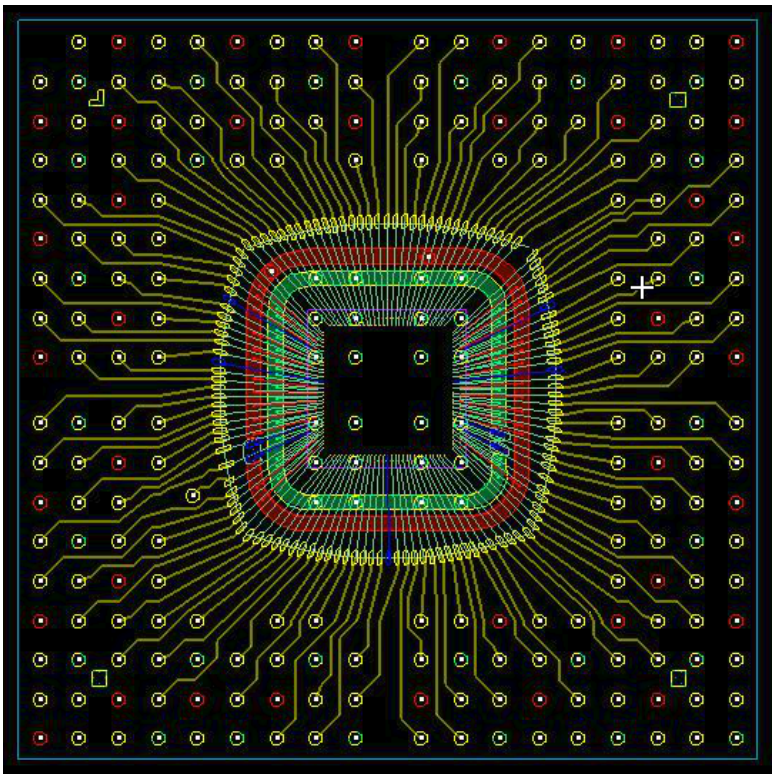


Figure 4.9. MiniDroid package layout The center cavity holds a 3x3-mm silicon die. Wire bonds connect pads on the die with either the power and ground rings in the package (shown in red and green, respectively), or with signal pads in the package (shown in yellow). The signal wire bonds fly over the power wire bonds. Internal package routing to external pins is shown for some signals.

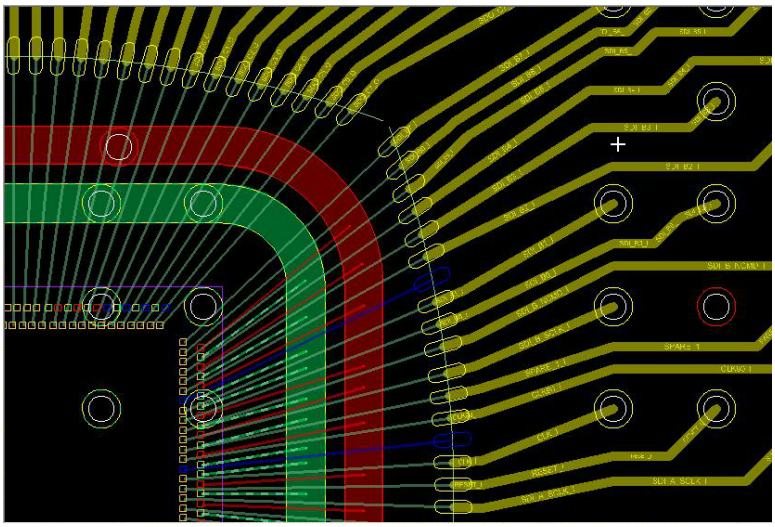


Figure 4.10. MiniDroid package layout detail Zooming in shows a more detailed view of the signal pads and internal routing.

bounding boxes, and pin constraints (location, spacing, metal layers). The user also specifies a power grid plan for each metal layer, including strap width and spacing, as well as power rings needed around any macros such as SRAMs.

The Catalyst flow includes the following stages: synthesis, floorplan, placement, clock tree synthesis, routing, filler cell insertion, static timing analysis (STA), metal fill, design rule check (DRC), layout versus schematic (LVS) check, and power estimation. Formal verification ensures the gate-level netlist is equivalent to the original RTL. At the end of the flow Catalyst generates a standalone GDSII hard macro for the design that can be incorporated into a MURN chip. Catalyst also produces reports on timing, area, power, and a quality of results (QoR) summary.

Catalyst uses the following Synopsys tools and versions: Milkyway (2010.03-SP5), Design Compiler Topographical (2010.12-SP2), Formality (2010.12-SP5), IC Compiler (2010.12-SP2), PrimeTime (2010.12-SP2), and VCS (2010.06-4). Catalyst also uses Mentor Graphics Calibre (2010.3_37) for metal fill, DRC, and LVS checks. The flow is based on a combination of TCL and XML. MiniDroid relies on a custom floorplan generator written in Python.

4.6.3 MiniDroid Physical Implementation

The first MURN chip includes a 2x2-tile version of GreenDroid, called MiniDroid. MiniDroid uses a 28-nm SLP process from GlobalFoundries. Figure 4.12 shows the chip floorplan.

The die size is 3000x3000 μm (9.0 mm^2). After subtracting space for the dual pad ring, the usable core area is 2400x2400 μm (5.8 mm^2). Each GreenDroid tile is 750x750 μm. Each of the two lower tiles has an I/O block with connections to the MURN ring network. Remaining space outside of the MiniDroid tile array is reserved for other design nodes and the common infrastructure: the MURN network, off-chip I/O block, power management, and clock management unit with PLL. Surrounding the core area are the dual pad rings and boundary power straps.

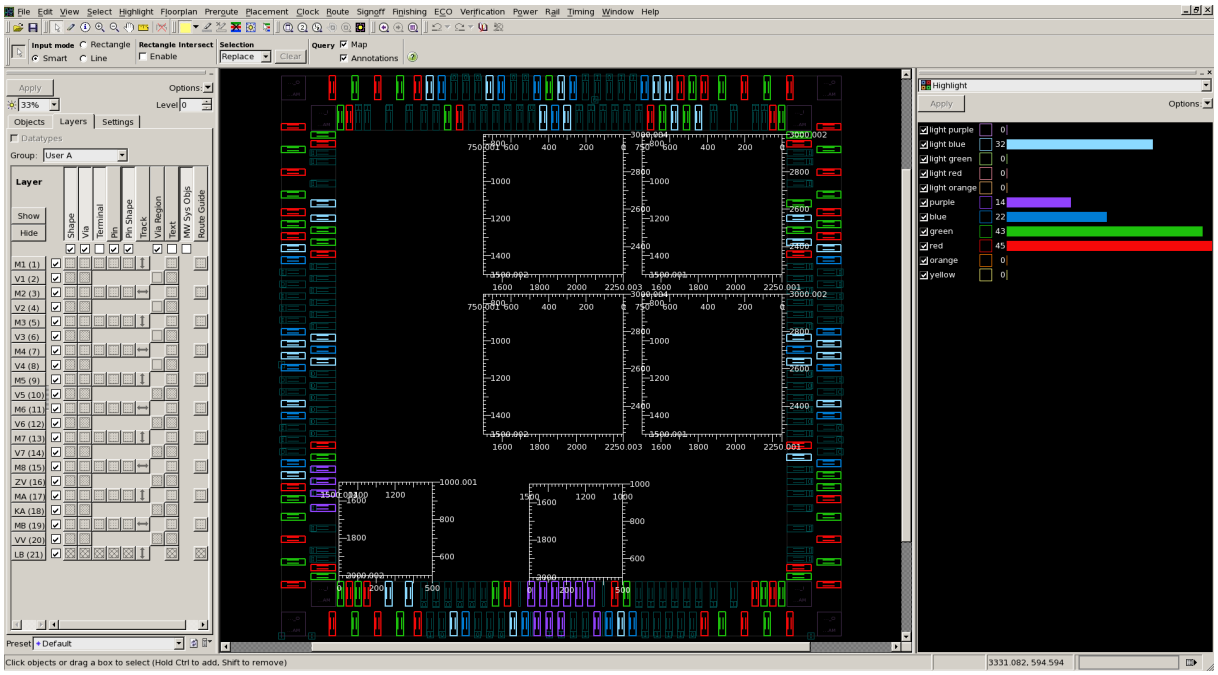


Figure 4.11. Sketching a MiniDroid floorplan Using IC Compiler to sketch a rough floorplan for MiniDroid.

The chip relies on a 28-nm standard cell library from ARM. The library includes cells that are 9 tracks high, with a mix of cells optimized for performance, power, and area. The library provides different options for transistor threshold voltage (V_t) and channel length. Low-V_t cells are faster and can be used in performance-critical timing paths, at a cost of higher leakage power. Similarly, cells with shorter channel length also increase performance and leakage, while longer-channel cells can be used to reduce leakage on non-critical paths. The cells are footprint-compatible, so place-and-route tools can swap them in place more easily to meet timing. The cells have a constant height of 0.9 μm, and variable width based on drive strength and cell complexity.

The metal stack (see Table 4.3) has 10 copper layers plus a terminating aluminum layer at the top used for bond pads. The first metal layer, M1, is reserved for internal use by the standard cells and macros. The lower metals M2-M6 and B1-B2 are primarily used for individual design nodes, while the upper metals IA and IB deliver power and route global signals. The PLL uses metal layers M1-M5.

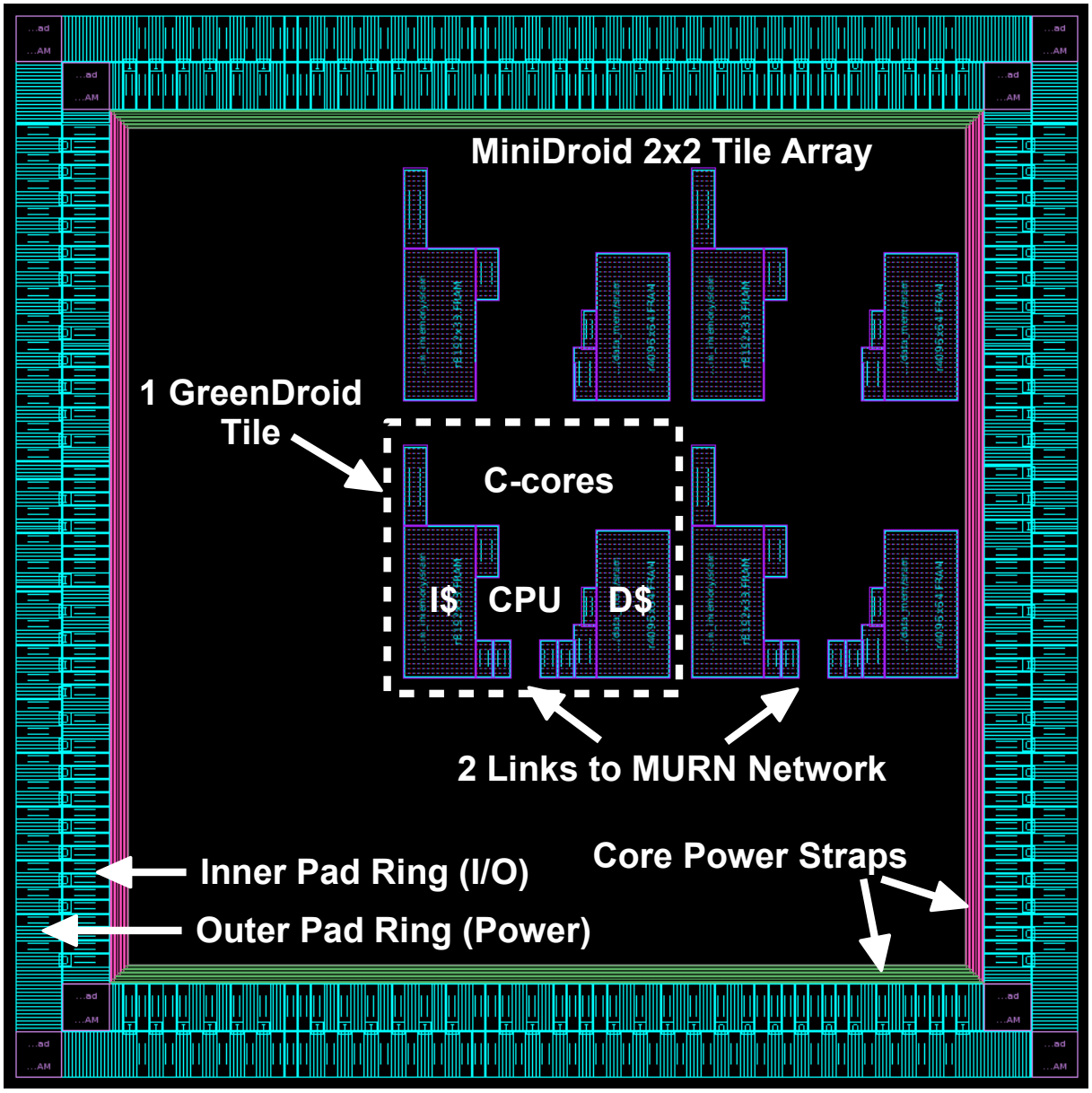


Figure 4.12. MiniDroid 28-nm chip floorplan with pad ring and 2x2 tile array The MiniDroid die is 3000x3000 μm (9.0 mm^2) in a 28-nm SLP GlobalFoundries process. Each GreenDroid tile is 750x750 μm. The lower two tiles have connections to the MURN ring network. Remaining space outside of the tile array is reserved for other design nodes, the MURN network and I/O block, and the PLL.

Table 4.3. MiniDroid metal stack The metal stack has 10 copper layers plus a terminating aluminum layer at the top for bond pads.

Layer	Thickness	Direction	Primary Purpose
LB	2.1 μm	N/A	Bond pad metal
IB	8x	H	Global power and routing
IA	8x	V	Global power and routing
B2	2x	H	Local and global power and routing
B1	2x	V	Local and global power and routing
M6	1x	H	Local routing
M5	1x	V	Local routing
M4	1x	H	Local routing, SRAM internals and power rings
M3	1x	V	Local routing, SRAM internals and power rings
M2	1x	H	Local routing, standard cell power rails
M1	1x	Both	Reserved for standard cell internal routing

The chip relies on the ARM Artisan SRAM compiler for generating memory macros. The compiler supports high speed and high density options, and can generate single- or dual-ported SRAMs. The SRAMs require metal blockages on M1-M4 for internal use. Routing can occur over the SRAMs on M5 and above. Each SRAM requires power rings on M3 and M4, and pin connections on one side of the macro on M2-M4.

4.7 Summary

The author's work on MiniDroid's physical implementation helped him gain valuable experience in chip architecture for a state-of-the-art silicon process. At the time, the 28-nm SLP process was still under development—in fact the author even uncovered and fixed several bugs in the CAD flows and process design kit (PDK).

The GreenDroid prototypes demonstrate that c-cores offer a new technique to convert dark silicon into energy savings and increased parallel execution under strict power budgets. GreenDroid validates the basic c-core approach—that it is possible to spend area to increase energy efficiency at little or no cost to performance. In the next two chapters, we will explore another specialized architecture that has been commercialized in a modern smartphone.

Acknowledgements

This chapter contains material from "GreenDroid: A Mobile Application Processor for a Future of Dark Silicon," by Nathan Goulding, Jack Sampson, Ganesh Venkatesh, Saturnino Garcia, Joe Auricchio, Jonathan Babb, Michael Bedford Taylor, and Steven Swanson, which has appeared in Hot Chips 22: A Symposium on High Performance Chips, ©2010 IEEE. The dissertation author is a primary contributor and first author of this paper.

This chapter also contains material from "The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future," by Nathan Goulding-Hotta, Jack Sampson, Ganesh Venkatesh, Saturnino Garcia, Joe Auricchio, Po-Chao Huang, Manish Arora, Siddhartha Nath, Vikram Bhatt, Jonathan Babb, Steven Swanson, and Michael Bedford Taylor, which has appeared in IEEE Micro, ©2011 IEEE. The dissertation author is a primary contributor and first author of this paper.

Chapter 5

Image Processing Unit

At Google, the dissertation author joined a broad team to design and implement another specialized architecture, *Pixel Visual Core* [RMGH+18]. Pixel Visual Core (PVC) is a programmable, energy-efficient accelerator for real-time image processing, computational photography, computer vision, and machine learning applications on mobile devices. Introduced in Google's *Pixel 2* smartphones, PVC enables Rapid and Accurate Image Super Resolution (RAISR) ML software zoom technology [RIM17][Mil16] and High Dynamic Range (HDR+) image processing [HSG+16] for all Android apps that use the Camera API [SR17][Sha18].

This chapter describes PVC's main compute accelerator, the *Image Processing Unit (IPU)*. The IPU is a domain-specific architecture [HP17] for image processing and computer vision. In contrast with GPUs, which synthesize new outputs, the IPU processes inputs (raw camera sensor data or other input images) and generates modified output images.

The IPU's most innovative features include: a large (16x16) SIMD array of ALUs in each of 8 Stencil Processor cores; a 2D shift network within each ALU array to support the spatial access patterns found in pixel image computations; and 2D hardware line buffers that store intermediate results between compute kernels, reducing the number of off-chip memory accesses. These features make it easy and efficient to perform the stencil computations needed in the latest image processing and machine learning algorithms.

This chapter first explains the motivation for building a programmable image accelerator,

including an explanation of traditional Image Signal Processors and their limitations. Then we describe the IPU architecture, programming, and execution model. Subsequently, the following chapter details the first implementation of the IPU, in the Pixel Visual Core SoC.

5.1 IPU Motivation

Recent years have ushered in a revolution in digital cameras, particularly cameras embedded in smartphones such as iPhone and Android devices. These devices overcome the physical limitations of ultra-thin form factors and tiny optical components by using advanced computational photography. Computational photography uses software processing to sharpen images, correct defects, and achieve better photos. In particular, computational photography helped Pixel phones earn the top DxOMark score in 2016, 2017, and 2018 [Car16][Car17][Car18].

Traditionally, digital cameras have relied on Image Signal Processor (ISP) pipelines for converting raw CMOS sensor data into human-viewable images. To meet strict power and performance targets, ISPs rely on specialized hardware—much of the ISP is implemented as a fixed-function ASIC, where the algorithms are tuned before mass production and hardened into the device. Once hardened, however, devices can no longer take advantage of any further improvements to the software algorithms [VBP+16]. Furthermore, most ISPs have to make significant tradeoffs, sacrificing image quality to meet minimum throughput requirements (e.g., reducing resolution or quality factor to encode video at 120 frames per second). Thus a fixed-function ISP encounters two problems: First, the fixed-function ASIC is inflexible and can't be updated after manufacture, and second, traditional ISPs have limited compute resources, sacrificing image quality to meet performance targets.

Google's IPU is designed to address these problems. First, the IPU is fully programmable, so it can run new versions of algorithms via software updates. Since the IPU is programmable it can also be applied to other problems and other domains, such as computer vision or machine learning. Second, as we'll show in Section 5.2, the IPU has a significant number of compute

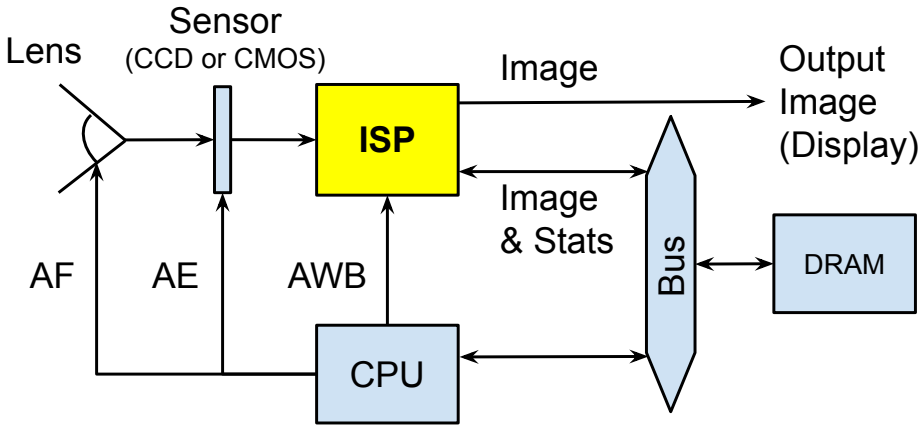
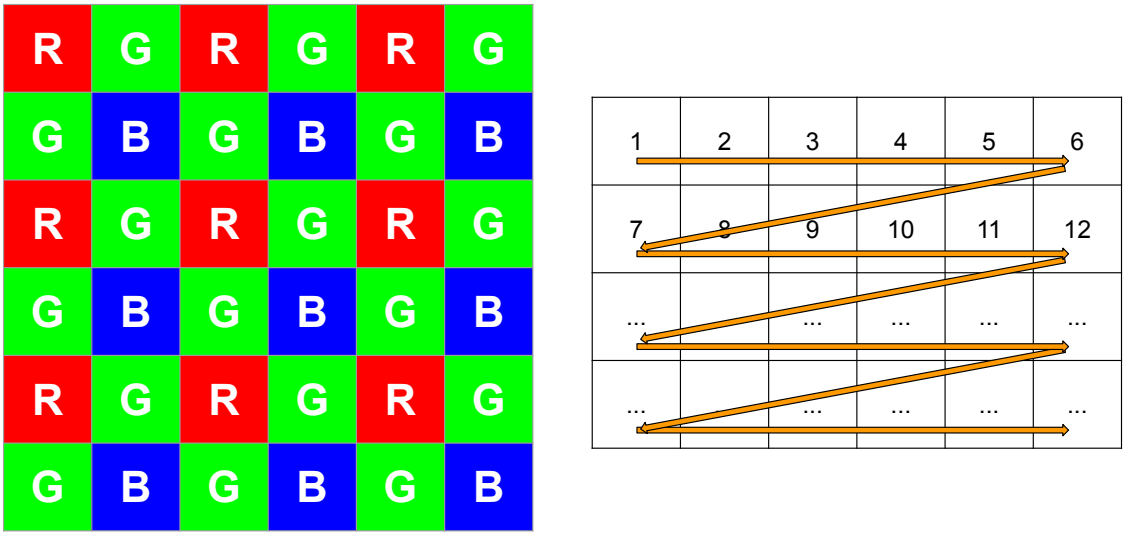


Figure 5.1. Organization of a digital camera's lens, sensor, and ISP Raw data captured by the sensor stream through the ISP, with feedback paths for Auto Focus (AF), Auto Exposure (AE), and Auto White Balance (AWB), collectively known as the 3As. Figure from [HP17].

resources that enable massive parallel computation. This accelerates computational photography techniques that would otherwise not be feasible for interactive applications (e.g., instant sharing). Third, as a domain-specific architecture the IPU also benefits from high energy efficiency through specialization.

5.1.1 Image Processing and Stencil Computations

In a typical digital camera system, shown in Figure 5.1, the camera lens focuses photons onto a CMOS sensor. The sensor produces raw data in a Bayer pattern of alternating red, green, and blue pixels, shown in Figure 5.2(a). Newer sensors may also interleave depth information from infrared, lasers, or embedded phase-difference detectors to assist with auto focus (e.g., PDAF). The data are fed into the ISP in raster scan order, in which input pixels are streamed into memory row-wise from top-left to bottom-right, shown in Figure 5.2(b). As CMOS sensors continue to increase in resolution, the width of each line can become quite long, requiring additional memory for line buffers. For example, in a 12-megapixel image (4032x3024 pixels) the ISP must buffer several complete lines, each 5 KB or more in size, before it can start computation.



(a) (b)

Figure 5.2. Bayer pattern generated in raster scan order An example CMOS sensor Bayer pattern of red, green, and blue pixels (a), and a depiction of row-major raster scan ordering (b). As image sensor resolutions increase, the width of each line grows, requiring more intermediate storage for line buffers in memory.

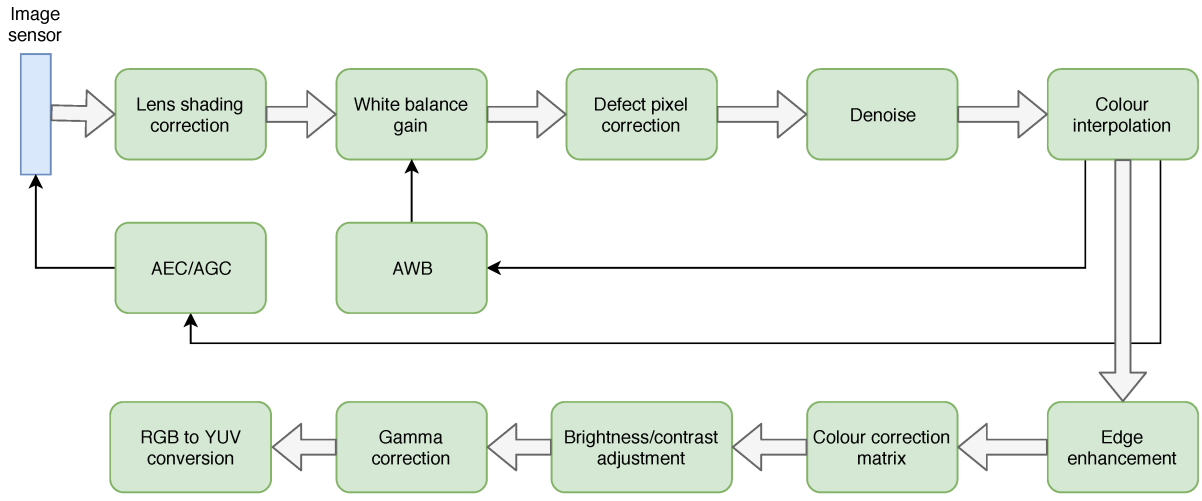


Figure 5.3. Image Signal Processor (ISP) pipeline example In a traditional ISP, in order to produce a visually pleasing image the raw sensor data must pass through many compute kernels. In between compute kernels, intermediate results are buffered in DRAM. Figure from [YHD+19].

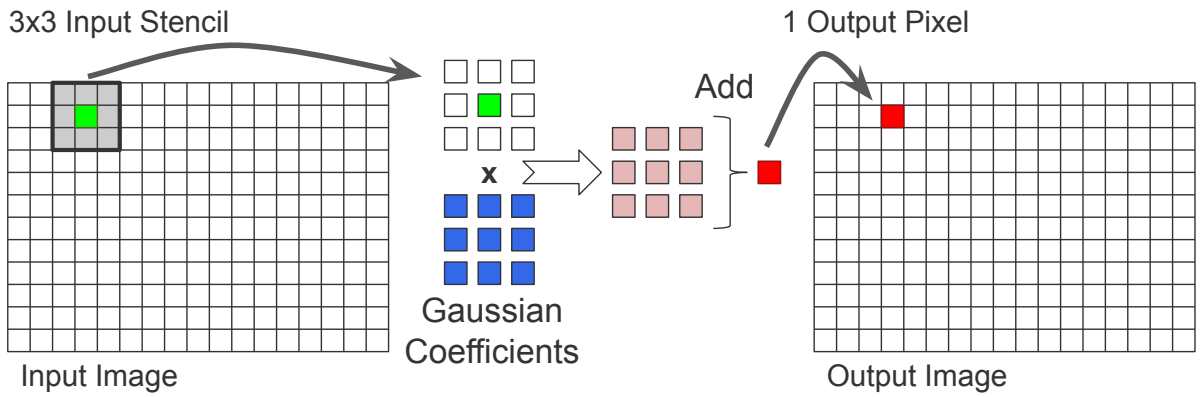


Figure 5.4. Stencil computation with a 3x3-pixel support region A stencil computation requires some number of input pixels (the support region) surrounding the target output pixel. In this example the support region is 3x3 pixels for a 3x3 Gaussian blur to produce each output pixel.

Figure 5.3 shows an example image processing pipeline, which consists of multiple stages (compute kernels) that perform different passes over an image such as denoising or color interpolation. In between each kernel the intermediate results are usually stored in DRAM, since each frame can be quite large. For example, a single 10-bit, 12-megapixel frame requires more than 14 MB of memory, and some of the ISP kernels read or write multiple frames (e.g., when converting to RGB planar format). All of these memory reads and writes cost time (performance) and energy.

Every compute kernel in the pipeline relies on outputs from a previous kernel. Computing one output pixel usually requires reading a number of input pixels in the surrounding area. This surrounding area is known as the support region or *stencil* of the computation. A stencil typically consists of a center pixel surrounded by some number of pixels in the horizontal and/or vertical dimensions, for example 5x5 or 1x3. Figure 5.4 shows an example. In order to compute a 3x3-pixel convolution over the entire image, the computation for each output pixel must read the values of 9 nearby input pixels (and 9 convolution coefficients). Note there is significant reuse of input data in both the horizontal and vertical dimensions, especially for larger stencil sizes. A general-purpose CPU or traditional ISP can only compute one or a few output pixels at a time, necessitating the same input data to be re-read many times from memory. The next section shows

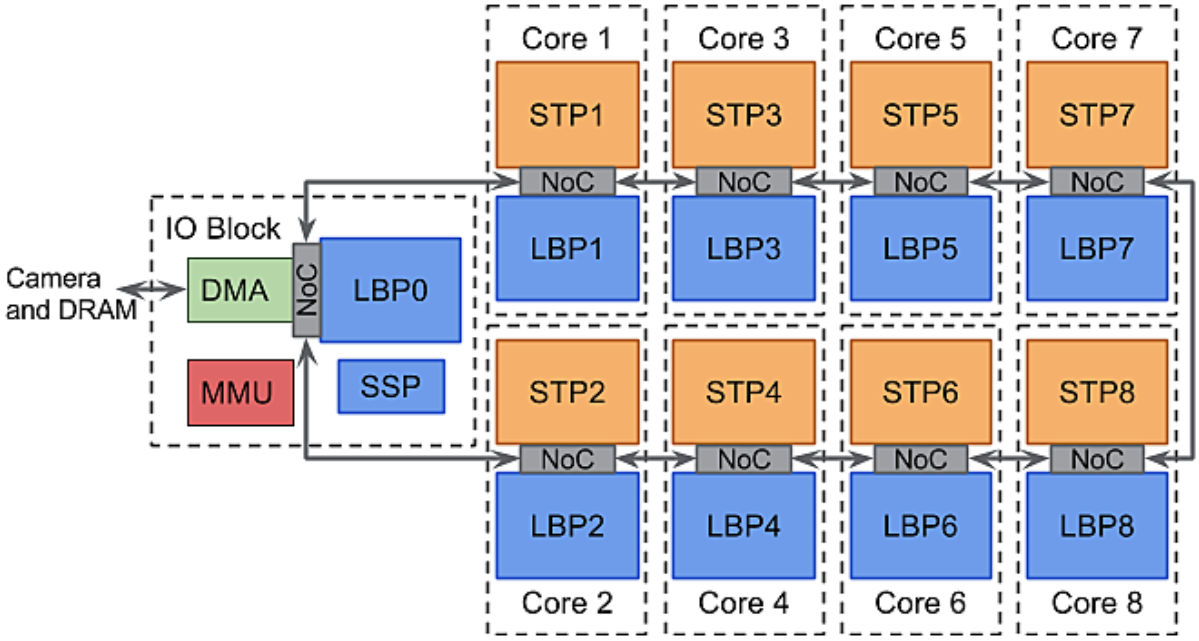


Figure 5.5. Image Processing Unit architecture Pixel Visual Core's Image Processing Unit contains 8 compute cores, an I/O block, and an on-chip ring network (NoC). Each of the 8 cores comprises a Stencil Processor, Line Buffer Pool, and connection to the NoC. Data stream from camera sensors or DRAM through a pipeline of compute kernels running on the Stencil Processors, with intermediate results buffered in the Line Buffer Pools. Data stream out back to DRAM or as MIPI output data sent to the AP for further processing.

how the Image Processing Unit's specialized architecture takes advantage of all this data locality, both in the IPU compute engines, which can access neighboring data in a 20x20 compute array, and in the IPU's hardware line buffers, which store required lines and parts of lines in between computer kernels, avoiding extra accesses to DRAM.

5.2 IPU Architecture

This section describes the architecture of the Image Processing Unit. Figure 5.5 shows an IPU with 8 processing cores. Each core contains a 2D compute array called a Stencil Processor, a hardware Line Buffer Pool with 128 KB of SRAM, and access to a bidirectional on-chip ring network (NoC).

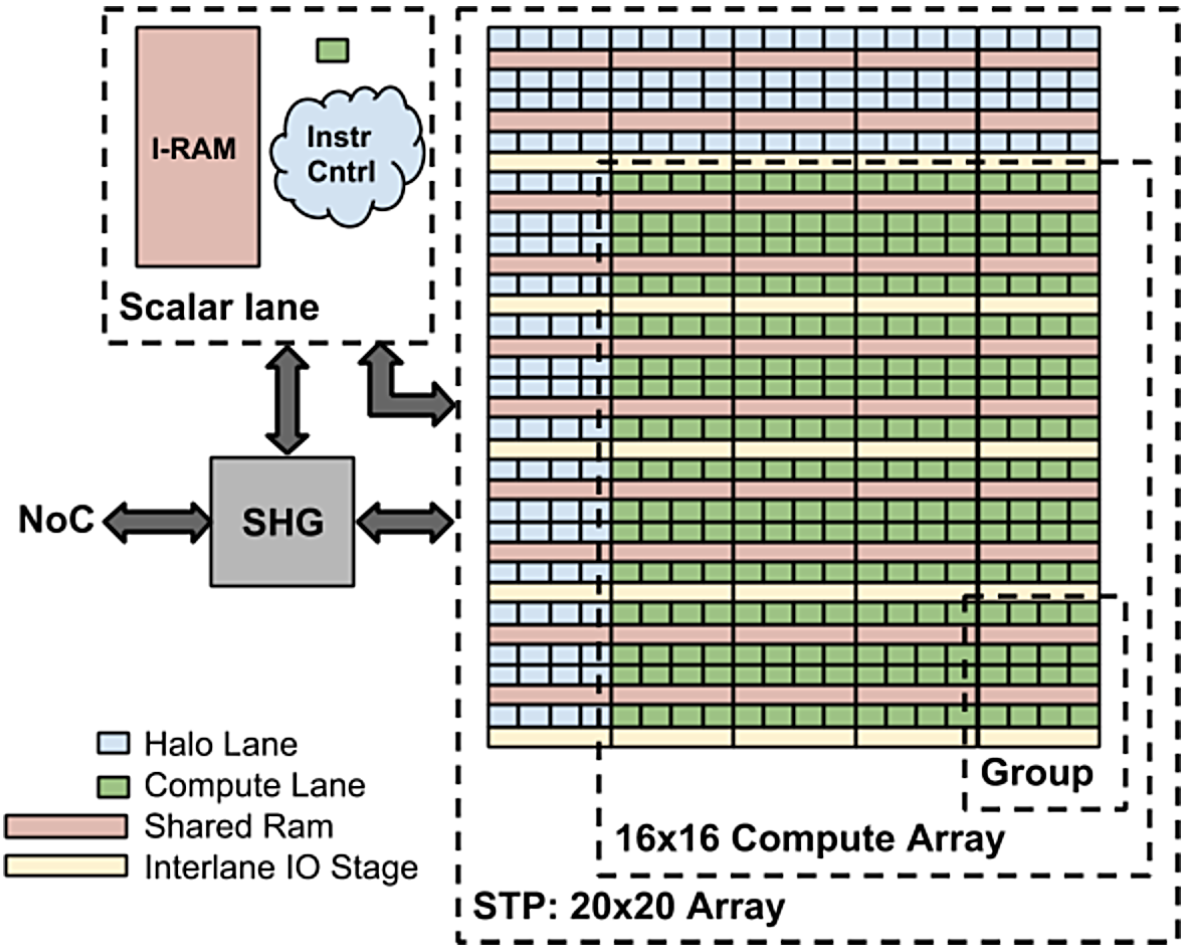


Figure 5.6. IPU Stencil Processor architecture The Stencil Processor consists of a 16x16-lane compute array (20x20 including halo lanes), Scalar Lane control processor, and a load-store unit called the Sheet Generator (SHG).

5.2.1 Stencil Processor

At the heart of the IPU is the Stencil Processor (STP), shown in Figure 5.6. Each STP contains a massively-parallel 20x20 SIMD array with 256 compute lanes and 144 halo lanes, which are directed by a scalar control lane and fed data from a load-store unit called the Sheet Generator. The STP uses a very long instruction word (VLIW) format containing three types of instructions: scalar lane, vector compute, and vector memory (see Section 5.3.4).

Scalar Lane

The Scalar Lane (SCL) is a scalar control processor that directs the execution of the main compute array and Sheet Generator. The SCL issues instructions from a dedicated 32 KB instruction RAM, handles control flow, generates interrupts, and broadcasts shared data to the compute array. Each cycle the SCL issues a new VLIW instruction. The SCL executes the scalar instructions, while the vector instructions are executed in the array.

2D Compute Array

The center of the IPU's power comes from its 2D SIMD compute array. The array is organized as a 20x20-lane grid, split into 256 compute lanes containing ALUs, and 144 halo lanes that are used to store the support region data for stencil computations at the boundary of the compute lanes.

Compute Lanes

Figure 5.7 shows the STP's single-cycle pipeline compute lane. Every compute lane contains two 16-bit ALUs, a multiply-add (MAD) unit, a 10-entry register file, shifters for accessing data in neighboring lanes, and access to scratchpad SRAMs shared by groups of 8 compute lanes.

The native word length is 16 bits. Each lane can compute up to two 16-bit arithmetic operations each cycle (either independent or chained), or the ALUs can be paired together to compute one 32-bit arithmetic operation each cycle. The IPU does not support native floating-point arithmetic, but it does simplify fixed-point multiplication with MAD instruction extensions that automatically scale multiplication results back to the correct radix point.

The register file contains ten 16-bit general-purpose registers. Four of these registers are accessible directly from any other lane up to four hops away in each cardinal direction. The register file has five read and three write ports to accommodate all of the operations that can happen each cycle.

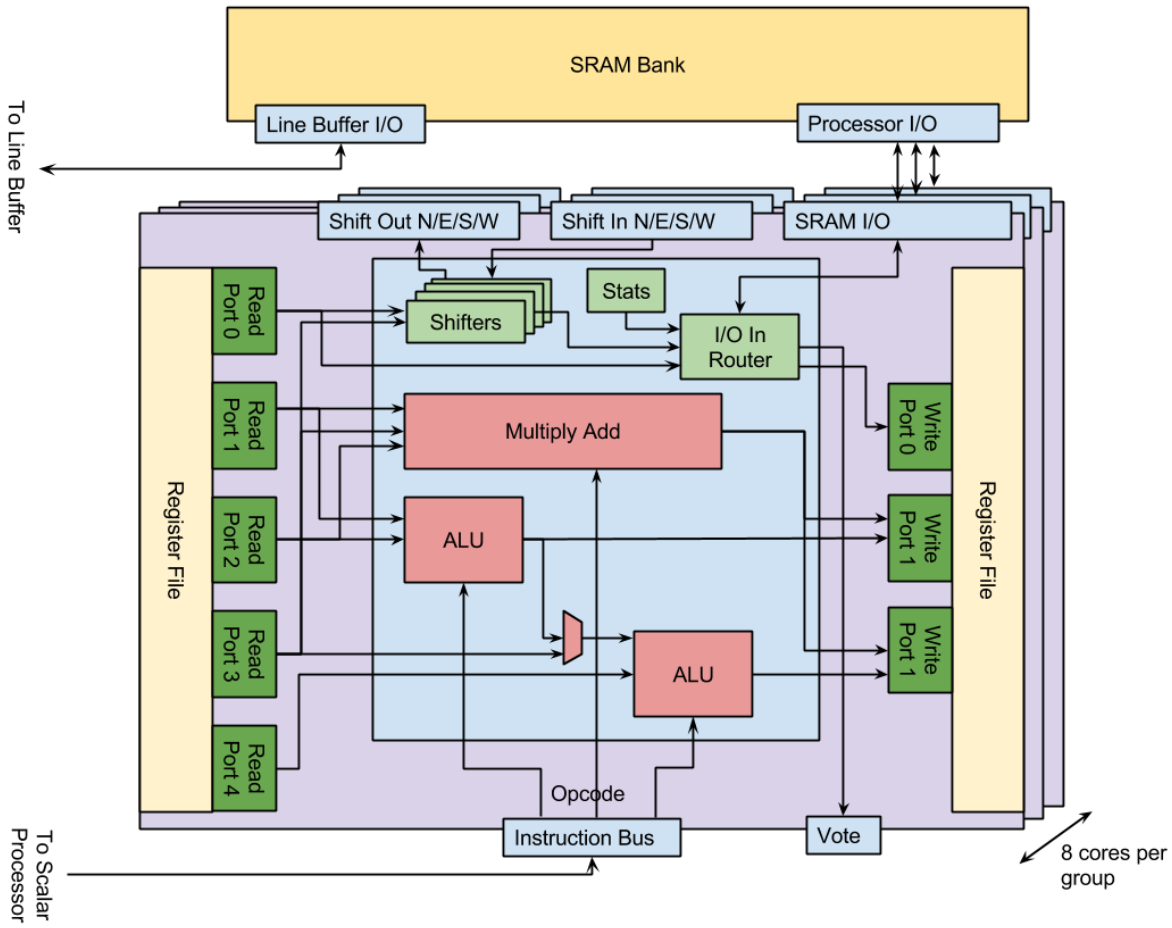


Figure 5.7. Stencil Processor compute lane Each compute lane in the STP contains two 16-bit ALUs, a multiply-add unit, register file, SRAM, and access to the shift network. The datapath is a single-cycle pipeline in order to reduce complexity and save energy.

Halo Lanes

The halo lanes do not perform computation; they are used to store input data needed for stencil computations at the border of the compute array. The halo lanes contain only four 16-bit registers and half the amount of scratchpad SRAM as the compute lanes. Since there are four rows and four columns of physical halo lanes, an STP can natively compute stencils up to 5x5 pixels in size. Larger 2D stencil sizes are emulated by the compiler. 3D stencils are emulated by storing data in each lane's SRAM.

Shift Network

The STP array contains a 2D shift network that allows any lane to read any of the four designated shift registers belonging to its nearby neighbors. A lane can access a neighbor's data up to four hops away in any direction (north, south, east, west), wrapping at the array edges. Logically, the shift network is a torus that wraps around vertically and horizontally. Physically, the compute and halo lanes are interleaved in the chip floorplan to avoid long hops around the edges and to equalize the wire delay between lanes.

The shift network is a key feature of the IPU that exploits data locality to enable extremely efficient stencil computation.

Sheet Generator

The Sheet Generator (SHG) is a load-store unit responsible for moving data between the STP array and any Line Buffer Pool. It can also manipulate data on the fly, including up/downsampling, striding, and transposition. The SHG gets its name from the fact that it operates on 2D *sheets* of pixels, typically 16x16 or 20x20 pixels to match the physical size of the compute array. The SHG transfers one 4x4 (32-byte) block into the array every cycle, so the 20x20 array can be filled in 25 cycles. SHG load and store operations overlap with STP computation, allowing the scalar lane to prefetch upcoming input data to hide some of this latency.

5.2.2 Line Buffer Pool

Every Stencil Processor is paired with a Line Buffer Pool (LBP). Each LBP holds a collection of hardware line buffers, which are essentially abstractions of 2D single-writer multi-reader storage. The line buffers serve as data buffers between producer and consumer STPs. They improve both latency and energy by helping to minimize costly accesses to DRAM.

Each LBP contains 128 KB of SRAM that can be used by up to eight independent line buffers within the LBP. The LBP also has flow control logic for handling STP stalling (no space available to write output data) and STP starving (input data not ready yet).

LBPs allow data to be accessed with (x, y) image coordinates as 2D blocks called sheets. The LBP also handles border conditions (e.g., zero padding, repeat edge, or mirror edge), which occur when requested data are outside of the stored image region. Each line buffer in an LBP has a single write pointer for the producer, and eight read pointers supporting up to eight simultaneous consumers. The compiler controls when to advance read pointers. When all of the read pointers have moved past an address, the line buffer is allowed to reclaim the space for future writes.

5.2.3 Network-on-Chip

The IPU has a network-on-chip (NoC) for transferring data between STP and LBP cores and the I/O block. The network is organized as a bidirectional ring. Each IPU core has a connection to the ring. Bypass paths in the NoC allow unused core pairs to be powered down while maintaining ring continuity.

Clockwise around the ring, the core IDs are assigned first with increasing odd numbers and then decreasing even numbers (see Figure 5.5). This allows any number of core pairs to be powered down, starting from the highest numbers, while maintaining a contiguous numbering scheme from 1 to N, for N powered-on cores. For example if cores 7 and 8 are powered down, the remaining cores are still numbered continuously from 1 to 6, simplifying code generation in the compiler and code reuse.

5.2.4 I/O Block

The I/O block transfers data on and off chip via a set of direct memory access (DMA) engines. The DMA engines can read and write to DRAM via the AXI bus, or read camera data directly via a set of high-speed MIPI input/output interfaces (see Section 6.1.4). The I/O block is controlled by the A53 CPU via AMBA interfaces (AXI, APB), and it communicates with image sensors and the ISP on the main application processor through MIPI.

The IPU DMA has 16 independent channels, allowing up to 16 simultaneous data transfers between off-chip resources and any LBP in the system, including the I/O block's own dedicated Line Buffer Pool, LBP0. The DMA is also used to program the STP instruction RAMs at the beginning of program execution.

The I/O block also has its own memory management unit (MMU) to facilitate address translation, and a 256-KB shared storage pool (SSP) to buffer DMA and MIPI packets.

5.2.5 Scalability

The IPU is a scalable multicore architecture. It is easy to scale the number of cores up or down depending on available silicon area and desired performance. From a physical design perspective, the cores are instantiated and placed in pairs. New core pairs can be added to the ring by instantiating additional NoC modules. If the number of core pairs grows even further, other network topologies could be considered, for example adding more entry points to the ring or multiple I/O blocks.

5.3 IPU Programming

The IPU architecture is especially suited to executing pipelines of compute kernels organized as a directed acyclic graph (DAG). Each kernel in the DAG is written in the high-level domain-specific language Halide [hal]. The Halide program is converted into IPU machine code via a two-step compilation process, shown in Figure 5.8 and described in this section.

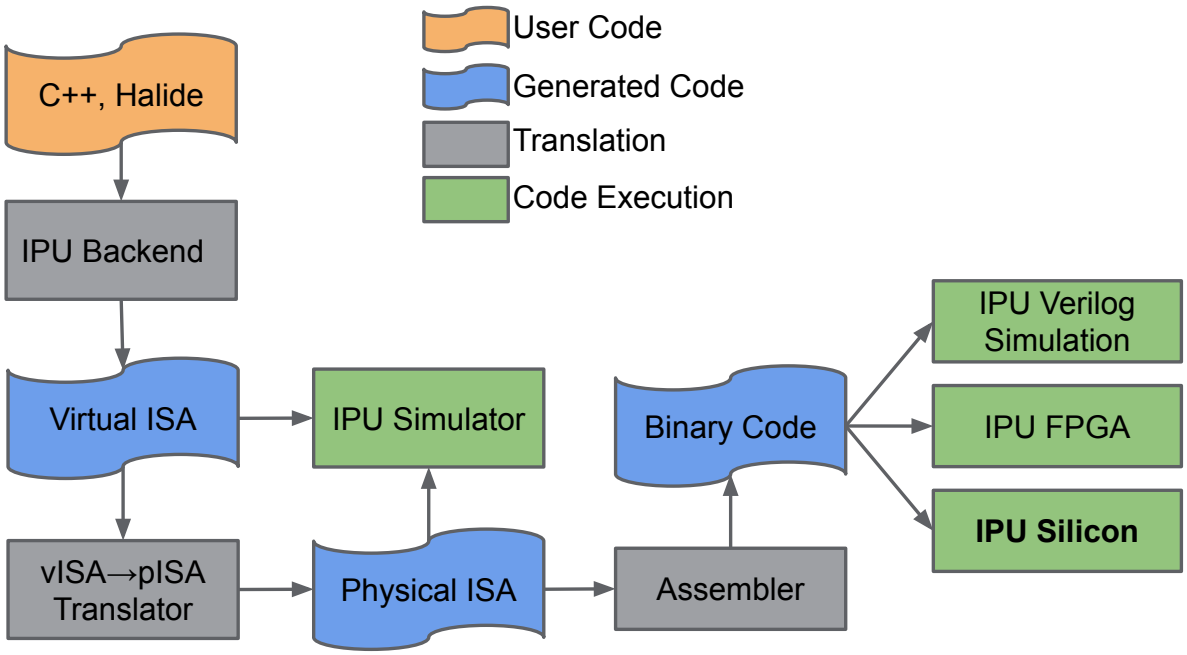


Figure 5.8. IPU toolchain The IPU toolchain first compiles Halide to a virtual ISA and then to the physical ISA for a specific instance of the IPU. Either ISA can run in the IPU architectural simulator for functional and performance verification. The assembled binary code can run in Verilog simulation, on the IPU FPGA, or on silicon.

Listing 5.1. Halide code for 3x3 blur [hal][DB18]

```
Func blur_3x3(Func input) {
  Func blur_x, blur_y;
  Var x, y, xi, yi;

  // Algorithm (no defined order or storage)
  blur_x(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y)) / 3;
  blur_y(x, y) = (blur_x(x, y-1) + blur_x(x, y) + blur_x(x, y+1)) / 3;

  // Schedule (defines order of execution and implies storage)
  blur_y.tile(x, y, xi, yi, 256, 32).vectorize(xi, 8).parallel(y);
  blur_x.compute_at(blur_y, x).vectorize(x, 8);

  return blur_y;
}
```

5.3.1 Halide Language

Halide is a domain-specific functional programming language designed to make it easier to write high-performance image and array processing code [RKBA+13]. Halide code is embedded in C++ using Halide's C++ API. Code is then compiled into an object file or JIT-compiled and run in the same process. The code can target many different architectures including CPUs (x86, ARM, MIPS), DSPs such as Hexagon [Cod13][SLBL+14], and GPUs via frameworks such as CUDA and OpenGL. A key feature of Halide is the separation of a kernel's *algorithm* (the set of computations to be performed on each pixel or input datum) from its *schedule* (how those computations are scheduled onto available compute resources). This separation allows code to be reused and optimized for different architectures without changing the underlying algorithm.

Listing 5.1 contains Halide code for a simple 3x3-pixel blur function. The first half of the code describes the blur algorithm. First the input is blurred in the X dimension with `blur_x()` by averaging each pixel with its left and right neighbors. Then the intermediate result is blurred in the Y dimension with `blur_y()` by averaging each pixel with its top and bottom neighbors. The result is an output image that appears blurry because each input pixel has been averaged with all of its neighbors in a 3x3 stencil. Note that the first half of the code (the algorithm) operates implicitly on every pixel of the input, so the code can handle images of arbitrary size.

Listing 5.2. Halide blur code from Listing 5.1, scheduled to run on the IPU

```
Func blur_3x3(Func input) {
  Func blur_x, blur_y;
  Var x, y;

  // Algorithm
  blur_x(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y)) / 3;
  blur_y(x, y) = (blur_x(x, y-1) + blur_x(x, y) + blur_x(x, y+1)) / 3;

  // Schedule
  if (has_ipu) {
    blur_x.ipu(x, y);
    blur_y.ipu(x, y);
  }

  return blur_y;
}
```

The second half of the code defines the schedule, which describes explicitly how to schedule the inner and outer loops over an entire input image of arbitrary size. For example, `.tile()` splits the input into 256x32-pixel blocks, while `.vectorize()` makes use of 8-wide SIMD execution units in the target architecture (e.g., x86).

The power of Halide comes from the ability to separate the algorithm from the schedule. The programmer can write the algorithm once, and then provide multiple schedules, each optimized for a different target architecture such as CPU or GPU, and as we'll see in the next section, the IPU.

5.3.2 Halide for IPU Programming

Programming for the IPU can be as simple as adding new scheduling directives to the Halide code. Listing 5.2 shows how the previous schedule can be replaced with `.ipu()` directives. Each `.ipu()` directive will create a new compute kernel running on a unique Stencil Processor. In between each STP, a line buffer in an LBP will store intermediate results, so the pipeline can execute over the entire image without needing to access DRAM more than once. Figure 5.9 shows the mapping of kernels onto STPs and LBPs for this example.

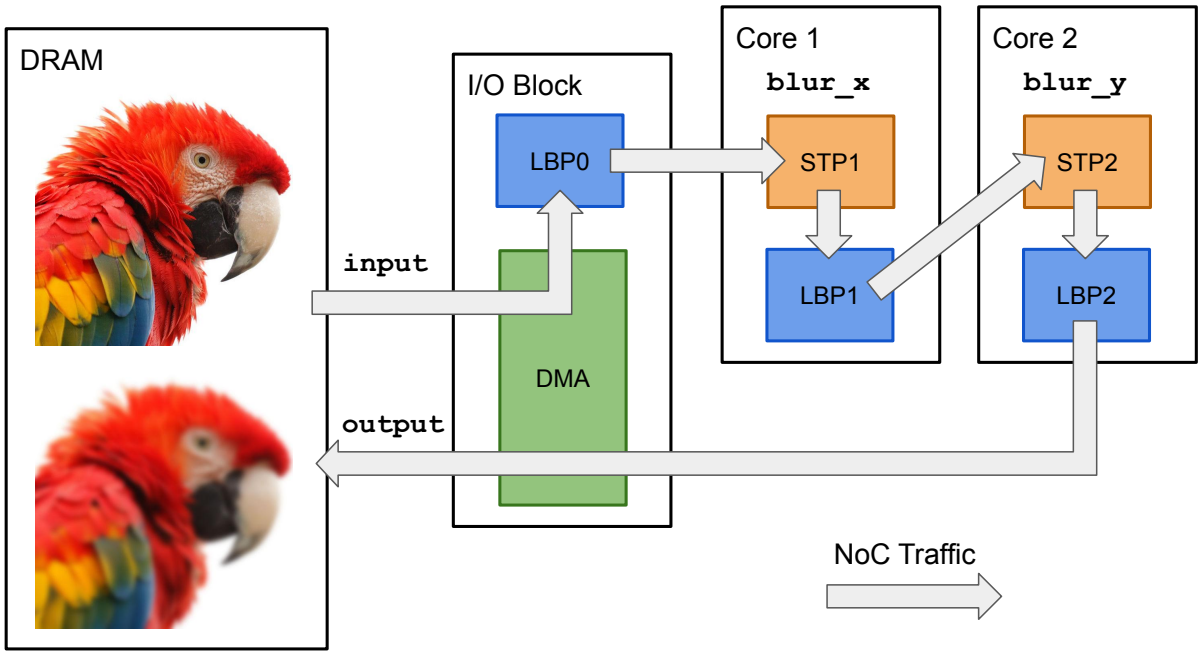


Figure 5.9. 3x3 blur mapped onto IPU The `blur_x()` and `blur_y()` kernels from Listing 5.2 are mapped onto two different STP cores, while line buffers are used to store intermediate results between kernels. The entire image is only read once and written once to DRAM.

In addition to `.ipu()`, the IPU compiler supports other directives to give more precise control of the schedule. For example, the programmer can merge kernels, schedule a kernel onto a specific STP/LBP, or override default compiler options. The programmer can also tell the compiler to split the input image into stripes and replicate a kernel to process the stripes in parallel on multiple STPs, to improve performance.

The IPU compiler extends the Halide compiler by interpreting `.ipu()` and the other scheduling directives to compile a program. Programs are compiled to IPU machine code in a two-step process, first to a virtual ISA before final translation to a physical ISA, described in the next two sections, respectively.

5.3.3 Virtual Instruction Set (vISA)

In the first compilation step, the IPU compiler translates Halide code into virtual instruction set architecture (vISA) instructions. vISA is an abstract representation of a virtual IPU.

Listing 5.3. vISA code for the 3x1 `blur_x` kernel from Listing 5.2

```
[blur_x]
!visa
input.b16 t0 <- __input[x*1+(-1)][y*1+0][0];
input.b16 t1 <- __input[x*1+0][y*1+0][0];
input.b16 t2 <- __input[x*1+1][y*1+0][0];
add.b16 t3 <- t0, t1;
add.b16 t4 <- t3, t2;
div.b16 t5 <- t4, 3;
output.b16 __blur_x[x*1+0][y*1+0][0] <- t5;
terminate;
```

The virtual IPU has an unbounded two-dimensional compute array, unlimited memory locations, and unlimited virtual registers allocated in static single assignment (SSA) form [RWZ88]. Later, vISA code is lowered to a physical ISA for a real machine with finite dimensions (see Section 5.3.4).

A vISA kernel describes the sequence of operations needed to generate a single output pixel at abstract location (x,y), given any number of input pixels inside the input stencil. The input and output pixels are referenced with image coordinates, specified as offsets from (x,y). For example, to generate pixel (x,y) in a 3x1 blur, the kernel will read inputs (x-1,y), (x,y), and (x+1,y). In vISA the maximum stencil size is unlimited—inputs can come from neighbors at any distance.

Listing 5.3 contains vISA code for the `blur_x()` kernel from Listing 5.2. The three `input` instructions read the left, current, and right neighbor pixels into virtual temporary registers. These three values are averaged with `add` and `divide` instructions to compute the output pixel. Each instruction is annotated with its bit width (e.g., `.b16` for 16-bit operations). Although not needed in this example, pixel coordinates can also include a multiplicative scaling factor to support upsampling and downsampling.

vISA code also supports extended arithmetic operations that may not exist in a particular implementation of the IPU, like modulo or floating-point operations. These operations can be emulated using physical instructions during the lowering phase to the physical ISA.

5.3.4 Physical Instruction Set (pISA)

The next compilation step lowers vISA code to physical ISA (pISA) instructions for a specific implementation of the IPU. As new versions of the IPU are designed, the pISA instruction set can evolve as instructions are added or removed, or as opcode behaviors are modified. Separating vISA and pISA allows code to be reused across multiple hardware generations that support different feature sets.

During translation the compiler maps vISA code into pISA using a number of parameters about the IPU's physical implementation. These parameters include STP array dimensions, halo size, the number of registers and local SRAM sizes, number of STP cores, number and size of LBPs, and any preferred mapping of kernels to STPs and LBPs. Virtual registers are allocated from the limited physical register set, and abstract memory locations are mapped into physical SRAM locations with real addresses.

Whereas vISA describes the operations for a single output pixel, without needing to consider actual image dimensions, pISA describes precisely how an entire image of a specific size will be processed. pISA divides the input image into sheets that are NxN pixels, matching the size of the STP compute array. For example, for a 16x16 compute array, a 12-megapixel image (4032x3024 pixels) will be split into 47,628 sheets (252x189). This means the innermost pISA loop will execute 47,628 times.

The Image Processing Unit's pISA code consists of very long instruction word (VLIW) instructions. Figure 5.10 shows the 119-bit instruction format. Each pISA VLIW contains instructions for the scalar and vector lanes, where the vector lanes can execute vector math and vector memory instructions. The 16-bit general-purpose immediate value can be used by the scalar and vector lanes. The 10-bit special-purpose memory immediate is used by the vector lanes for vector memory instructions with absolute addresses or fixed values that are known at compile time, such as local memory addresses or neighbor offsets when reading stencil data from the shift network.

VLIW

Field	padding	scalar	vector math	vector memory	immediate	mem. imm.
Width	9	43	38	12	16	10
Bit	127					0

Scalar Instruction

Field	mode	opcode0	opcode1	dst0	dst1	src0	src1	src2	src3	bcast0
Width	3	6	6	4	4	4	4	4	4	4
Bit	118									76

Vector Math Instruction

Field	mode	opcode0	opcode1	dst0	dst1	src0	src1	src2	src3
Width	2	6	6	4	4	4	4	4	4
Bit	75								38

Vector Memory Instruction

Field	opcode	dst0	src0
Width	4	4	4
Bit	37		26

Figure 5.10. pISA VLIW instruction format Each pISA instruction is a 119-bit VLIW with scalar, vector math, and vector memory instructions, as well as two fields for immediate values. The VLIW instructions are zero-padded to 128 bits outside of the IPU instruction RAMs.

Listing 5.4. An example pISA instruction

```
set.b16 bcast0 <- 4096 | add.b16 st1 <- st1, st2 |
  or.b16&sub.b16 t2, t0 <- t4, t3, t5, bcast0 | rdnxyi.b16 s1 <- s3, 4;
```

The VLIW instructions are zero-padded to 128 bits outside of the STP instruction RAMs. The padding provides some space to extend the VLIW format in future generations. However, this padding is stripped when loading a program into the instruction RAMs to reduce the total SRAM and wires required on chip.

The pISA instruction set includes standard arithmetic operations (e.g., add, subtract), bitwise operations (and, or, not, xor, shift left/right), and comparators (e.g., seq, slt). There are also specialty instructions including max/min, absolute value, count leading zeros, and an 8-cycle iterative integer divide. Each lane also has a multiply-add (MAD) unit that can do a single 16×16-bit multiply plus 32-bit add/subtract per cycle. The MAD instructions support an optional fractional shift at the output to set the radix in fixed-point computations.

The scalar lane supports a superset of the vector math instructions and also has dedicated instructions for flow control (branch, stall, interrupt), load/store to the scalar lane's own local memory, and sheet load/store to move data with the Sheet Generator between the vector array and a line buffer.

The vector lanes include vector memory instructions for reading pixel values from neighboring lanes via the shift network, and load/store instructions for accessing the local scratchpad memory within a lane group. These accesses can be either eight parallel load/store ops to the same line (with offsets automatically determined based on position within the group), or one divergent load/store op to the memory of any single lane within the lane group. There is also a read status instruction that returns a lane's physical (x, y) position in the array, needed by some algorithms.

Listing 5.4 shows one example pISA instruction. Each instruction consists of four parts, separated by | characters. The first part describes the 16-bit value, if any, to broadcast to every

lane in the vector array. The broadcast value can be any scalar register or (shown here) a 16-bit immediate value. The next three parts are assembly instructions for the scalar, vector math, and vector memory instructions in the VLIW. This example has a simple addition in the scalar lane, two independent 16-bit operations (`or` and `sub`) that will execute in all 256 vector compute lanes, and a vector memory operation `rdnxyi`. The `rdnxyi` operation reads a neighboring lane's register up to four hops away in any cardinal direction, and will execute in all 400 compute and halo lanes to move data around via the shift network.

5.4 Execution Model

In Pixel Visual Core, the IPU is tightly coupled with an ARM Cortex-A53 general-purpose CPU (see Chapter 6 for more details). This CPU runs the lightweight operating system Embedded Linux (eLinux) [eli]. On top of eLinux there is an IPU system software stack including driver, scheduler, and resource manager, known collectively as the IPU Runtime. This section describes how the IPU Runtime starts executing after Pixel Visual Core first boots during Pixel phone power-on, and then describes how individual jobs execute from the Runtime.

5.4.1 PVC and IPU Runtime Boot Sequence

1. The phone powers on.
2. During Android boot, the application processor (AP) transfers a ramdisk image file from flash memory into PVC's DRAM via PCIe.
3. The A53 boot ROM unpacks the ramdisk and creates a temporary file system (tmpfs) in PVC's DRAM.
4. The A53 finishes booting eLinux and starts the IPU Runtime service.
5. The A53 goes to sleep, suspending to PVC's DRAM in a low-power suspend mode.
6. Android finishes booting.

5.4.2 PVC and IPU Runtime Job Execution Sequence

After booting, PVC is ready to accept requests from the AP. The sequence is as follows:

1. The user launches an Android app that uses the camera (e.g., Instagram).
2. The AP wakes up the A53, which readies the IPU Runtime by switching it from suspend to standby mode. Standby uses higher power than suspend but allows a much faster transition to active mode.
3. The user pushes the camera shutter button, launching an image capture request via the Android Camera API, which interrupts the A53.
4. The A53 switches the Runtime into active mode.
5. For each set of compute kernels needed by the app:
   (a) The Runtime configures the IPU by setting control and status registers (CSRs), programming the STP instruction RAMs, configuring each LBP, and configuring the DMA engines for each input and output data stream.
   (b) The Runtime initiates IPU execution by writing one final CSR.
   (c) Data streams through the IPU's STPs and LBPs, with input data coming from DRAM or one of the camera's MIPI input channels (see Section 6.1.4). Output data can also be written to DRAM or a MIPI output channel.
   (d) When finished with the computation, the IPU raises an interrupt on the A53.
6. When the last set of kernels has finished executing, the Runtime alerts the AP that the final result is ready in PVC's DRAM or is streaming out over MIPI.
7. The AP transfers the final image into its own DRAM over PCIe, and provides it to the Android app.
8. Once the user exits the app, the A53 and Runtime return to suspend mode to save power.

5.5 Summary

This chapter presented the Image Processing Unit, a domain-specific architecture for image and vision processing in mobile devices. The IPU hardware is designed to exploit the two- and three-dimensional data locality common to image processing algorithms, through large SIMD compute arrays and hardware line buffers. The intra-array shift network facilitates data sharing between neighboring pixels. These features make it easy and efficient to perform the stencil computations needed in the latest image processing and machine learning algorithms.

Acknowledgements

This chapter contains material from "Pixel Visual Core: Google's Fully Programmable Image, Vision and AI Processor for Mobile Devices," by Jason Redgrave, Albert Meixner, Nathan Goulding-Hotta, Artem Vasilyev, and Ofer Shacham, which has appeared in Hot Chips 30: A Symposium on High Performance Chips, ©2018 IEEE. The dissertation author is a primary contributor and third author of this paper.

Chapter 6

Pixel Visual Core

This chapter presents the first implementation of the Image Processing Unit in silicon, Google's Pixel Visual Core. This chapter first describes the PVC SoC architecture, which comprises an 8-core IPU accelerator along with control processor, on-chip interconnect, I/O interfaces, and stacked DRAM dies in one package. This is followed by a discussion of the SoC's physical implementation in a 28-nm TSMC process, and then power and performance evaluation for key workloads.

Pixel Visual Core provides raw performance of up to 3.1 Tera-ops/second (1.7 Tera-ops/sec arithmetic) on 16-bit integer data. Designed for the mobile domain, where power is very limited, the only way to get Tera-ops/second performance is to optimize performance per watt, also known as operations per picojoule (Ops/pJ). This is because:

$$\text{Power} = \text{Energy/Second} = \text{Ops/Second} \times \text{Energy/Op}$$

With a fixed power budget, the only way to increase performance (Ops/Second) is to decrease Energy/Op [Sha11].

The 28-nm PVC achieves about 0.84 pJ/Op, or 1.5 pJ/Op including only arithmetic operations without data movement. Despite a three-generation process gap, the 28-nm PVC runs key HDR+ kernels 3-6× faster and with 7-16× less energy than a 10-nm general-purpose application processor with DSP.

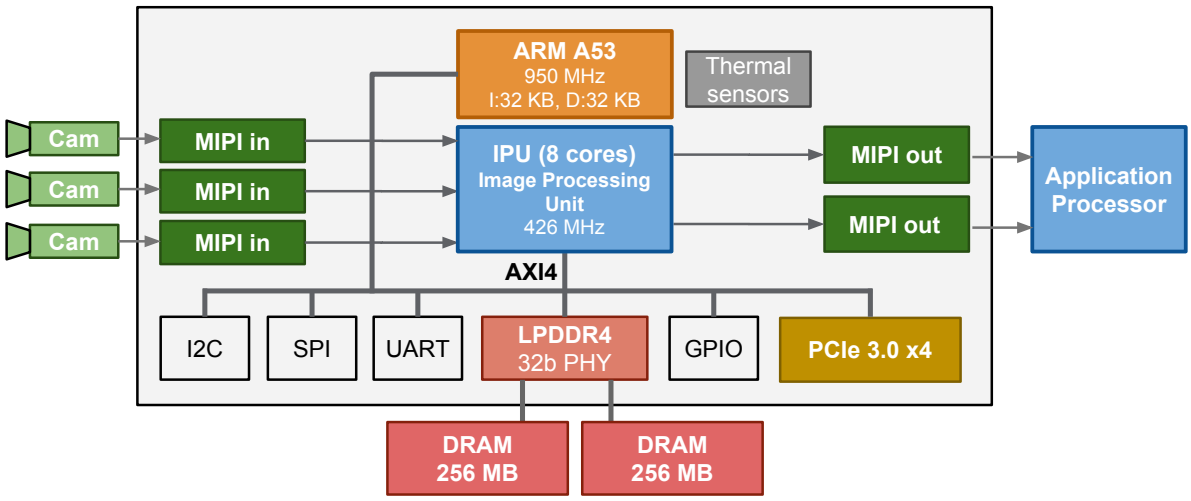


Figure 6.1. Pixel Visual Core SoC architecture Pixel Visual Core contains an 8-core Image Processing Unit (IPU), ARM Cortex-A53 CPU, DDR memory and PCIe controllers, high-speed MIPI camera interfaces (3x input, 2x output), and low-speed I/O (I2C, SPI, UART, GPIO).

6.1 Chip Architecture

Pixel Visual Core is a system-on-chip (SoC). The heart of the SoC is the Image Processing Unit. The rest of the SoC is dedicated to keeping the IPU busy, feeding it input data and control signals, and interacting with the rest of the system including the Pixel phone's main application processor and cameras.

Figure 6.1 shows a block diagram of the SoC. At the center is an 8-core IPU, connected to the SoC's main interconnect, an AXI bus. Also sharing the bus are a general-purpose control processor (ARM Cortex-A53 CPU), high-speed I/O interfaces (DDR and PCIe), and several low-speed I/O interfaces (I2C, SPI, UART, GPIO). Separate from the AXI bus are five high-speed MIPI input and output interfaces. The following sections describe each of these components in more detail.

6.1.1 Image Processing Unit

As described in Chapter 5, the IPU architecture is scalable in the number and size of Stencil Processors and Line Buffer Pools. In Pixel Visual Core, the IPU has 8 STP/LBP cores

plus one LBP in the I/O block. The STP array size is 16x16 (20x20 including halo lanes), and the LBPs each have 128 KB of SRAM. The IPU's max clock rate is 426 MHz. With 8 cores, PVC's IPU has more than 4,096 ALUs and more than 1 MB of on-chip SRAM.

6.1.2 Control Processor

As described in Section 5.4, the Pixel Visual Core SoC has a control processor that manages jobs on the IPU and executes the IPU Runtime software. This control processor is a 32-bit general-purpose ARM Cortex-A53 CPU supporting the ARMv7 instruction set with NEON extensions [ARM]. It has separate 32-KB L1 instruction and data caches and a shared 128-KB L2 cache. At maximum speed it runs at 950 MHz, but can be clocked lower to save power.

6.1.3 Interconnect

The SoC's on-chip interconnect follows ARM's Advanced Microcontroller Bus Architecture (AMBA) standard. There is a high speed Advanced eXtensible Interface (AXI) bus for sending data quickly between IPU, CPU, PCIe, and memory. There is also a low speed Advanced Peripheral Bus (APB) used to program the IPU via control and status registers (CSRs). The AXI bus runs at 600 MHz, and the APB runs at 250 MHz.

6.1.4 I/O Interfaces

The Pixel Visual Core SoC supports many different high-speed and low-speed interfaces, described here.

MIPI

Most mobile phones including Pixel use the Mobile Industry Processor Interface (MIPI) Camera Serial Interface 2 (CSI-2) standard [MIP] for sending data from camera sensors to host devices. MIPI supports high speed data transfer of many different image resolutions, data formats (e.g., planar or interleaved RGB), padding, blanking, and bit widths.

A significant challenge of MIPI is that there is no mechanism for back-pressure or flow control. The consumer must be able to keep up with processing producer data, or else drop frames or reduce the bit rate by lowering the resolution or frame rate. This means timing and performance analysis are crucial on these critical paths.

Pixel Visual Core supports five MIPI CSI-2 D-PHY interfaces: 3 input channels for receiving data from the front and rear cameras, and 2 output channels for sending data downstream to the application processor. Each channel runs at 156.25 MHz and supports up to 10.0 Gbps (4 lanes at 2.5 Gbps per lane).

In Pixel 2, Pixel Visual Core is a "bump in the wire" between camera sensors and application processor. This allows the IPU to process camera data immediately as it streams in instead of going through memory, reducing latency. PVC also has a MIPI bypass path that can be used when IPU processing is not needed.

PCIe

Pixel Visual Core includes a Peripheral Component Interconnect Express (PCIe) interface for sending data to and from the application processor and the phone's main memory. The interface is PCIe Gen 3 and supports 4 lanes up to 4 GB/s. PCIe is the primary interface for sending jobs from the application processor to Pixel Visual Core. PCIe is also used to transmit input and output data for offline (non-MIPI) jobs.

LPDDR

Pixel Visual Core includes off-chip but in-package DRAM (see Section 6.1.5). This is managed by a Low-Power Double Data Rate (LPDDR) memory controller. The LPDDR4 controller has a 32-bit PHY and supports up to 9.6 GB/s.

Low-speed I/O

In addition to the high-speed interfaces described above, PVC supports several low-speed I/O interfaces. These include Inter-Integrated Circuit (I2C), Serial Peripheral Interface (SPI),

Universal Asynchronous Receiver-Transmitter (UART), and general-purpose I/O (GPIO) pins. Some uses of these interfaces are used for communication with the PMIC and temperature sensors, debugging, logging, and accessing an interactive terminal shell in eLinux on the A53 CPU.

6.1.5 DRAM

The Pixel Visual Core package includes 512 MB of dedicated LPDDR4 DRAM memory. Physically the memory consists of two 256-MB dies in the package (see Section 6.2.3). Keeping this memory inside the package reduces memory latency and increases bandwidth. Since PVC has its own dedicated memory there is no contention with the main application processor.

The size of the DRAM was chosen based on PVC's primary application, HDR+ image processing. HDR+ requires buffering up to 10 still images before starting computation. With a 12-megapixel sensor (4032x3024 pixels) and 10-bit sensor data expanded into PVC's native 16-bit word size, a minimum of 233 MB is needed for storing the input data. Also stored in DRAM are intermediate results and final outputs, as well as memory needed for eLinux, the IPU Runtime, and application software. Limited physical memory, with no backing-store/paging mechanism, increases the burden on software engineers, who must allocate memory buffers very conservatively and vigilantly squash memory leaks.

6.2 Physical Implementation

This section describes the physical implementation of Pixel Visual Core, including process technology, details about the 28-nm SoC die, and the combination of SoC plus DRAM in a single system-in-package.

6.2.1 Process Technology

Pixel Visual Core is manufactured using a TSMC 28-nm HPC (High Performance Compact) process with 10 metal layers [TSMa]. The process uses High-k Metal Gate (HKMG)

transistors. High-k refers to the transistor gate insulator, which uses a material with a higher dielectric constant than silicon dioxide in order to reduce gate current leakage. Metal gate refers to the gate material, which has much better conductivity and also reduces leakage compared to traditional polysilicon.

6.2.2 SoC Die

The Pixel Visual Core's SoC die is 43.2 mm^2 (6.0x7.2 mm). Figure 6.2 shows a photomicrograph. More than half of the die is dedicated to the 8-core IPU and its I/O block. The remainder of the silicon serves only for system control and to get data into and out of the IPU. A key point is that almost all of the non-IPU silicon serves no purpose except data movement. Most of this overhead comes from the fact that PVC is a standalone accelerator, and needs its own PCIe, DDR, and MIPI controllers to interact with the rest of the system. Integrating an IPU directly into an application processor die would reduce area, power, and performance overheads.

6.2.3 System-in-Package

Pixel Visual Core is built as a system-in-package (SiP), shown in X-ray in Figure 6.3. A SiP consists of multiple dies stacked vertically or tiled horizontally and connected together by a single substrate in one package. The Pixel Visual Core SiP contains three vertically stacked dies in one 81.6 mm^2 package: two 256-MB DRAM memory dies on top of the main SoC die.

The SoC die uses flip chip technology to connect to the package substrate. In a flip chip, the chip pads are located on the top-level metal above the rest of the chip. Solder bumps are deposited onto the pads and the entire chip is flipped over and soldered directly to the top of the package substrate.

In contrast with flip chip, the memory dies are wire bonded to the substrate. Space between the dies is filled with epoxy to stabilize the structure mechanically and redistribute heat. If the heat is not distributed evenly, the dies and the substrate can expand and contract at different rates. This can lead to mechanical and thermal stress that can cause pads to fracture.

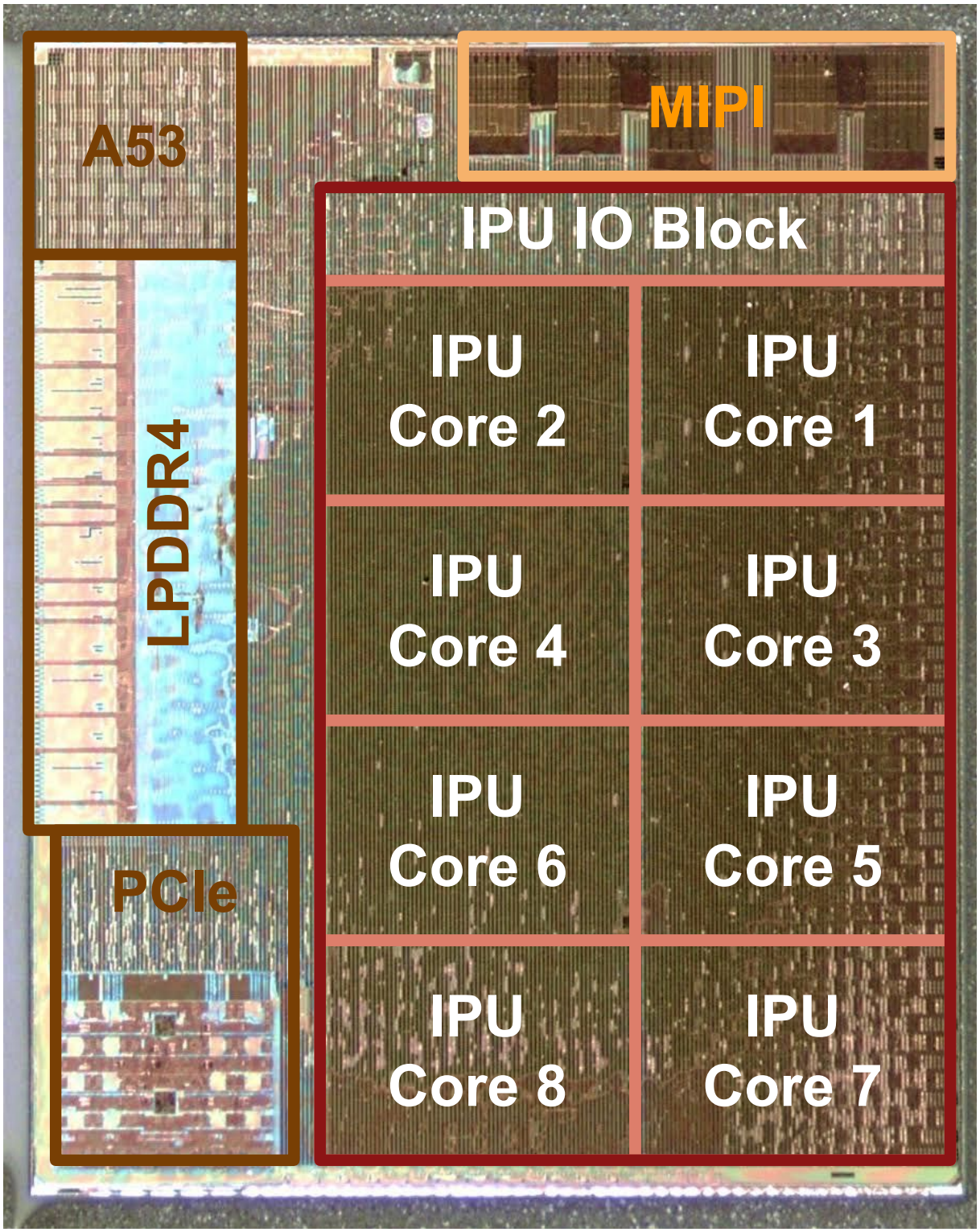


Figure 6.2. Pixel Visual Core photomicrograph The die photo highlights the 8-core Image Processing Unit, ARM Cortex-A53 CPU, LPDDR4 memory controller, PCIe Gen 3 controller, and high-speed MIPI camera interfaces. The die is 43.2 mm^2 (6.0x7.2 mm) in a 28-nm process.

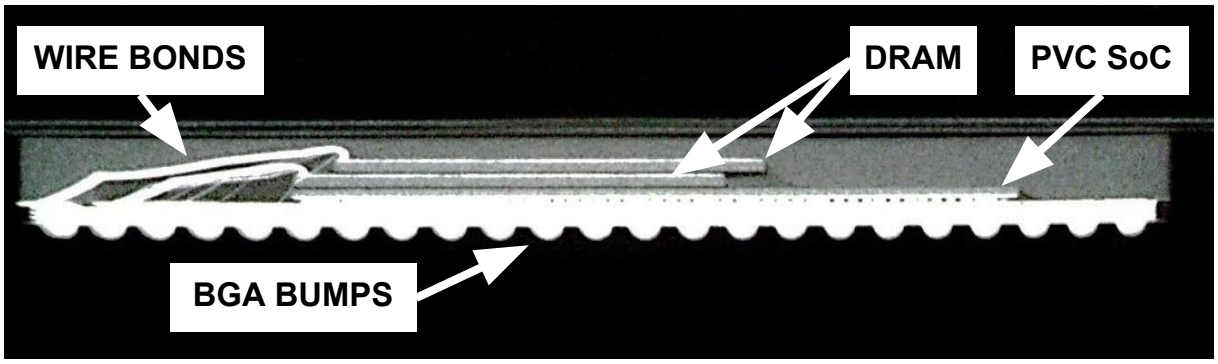


Figure 6.3. X-ray radiograph of the Pixel Visual Core system-in-package The package contains two wire-bonded DRAM dies stacked above the flip chip SoC.

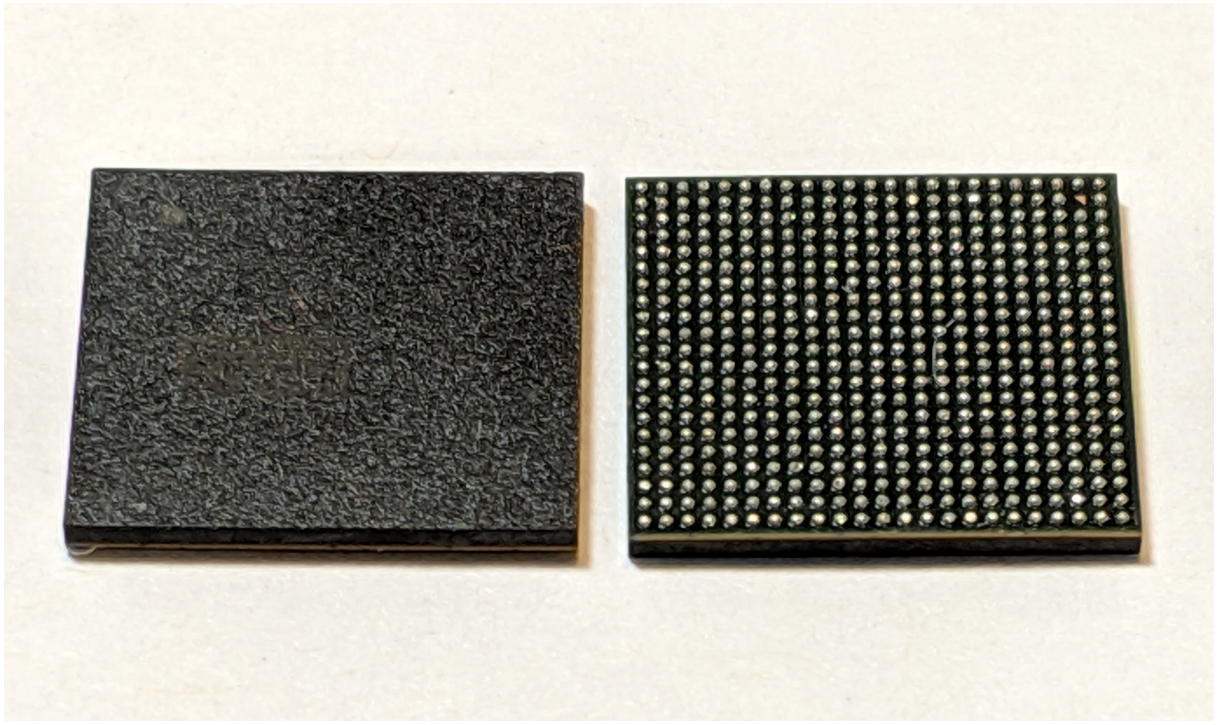


Figure 6.4. Pixel Visual Core BGA package Photo taken with Pixel Visual Core of course!

Pixel Visual Core resides in a ball grid array (BGA) package, shown in Figure 6.4. The BGA has 482 balls arranged in a grid: 23x21, minus one for the notch shown in the top-right corner that indicates pin 1.

6.3 Evaluation

We evaluate Pixel Visual Core by measuring its raw compute power and performance. We also evaluate PVC's power and performance compared to a recent Snapdragon SoC on HDR+ processing kernels.

6.3.1 Maximum Performance

Pixel Visual Core's IPU is capable of executing two 16-bit arithmetic operations and one 16-bit memory/move/shift operation per lane, per STP core, every cycle. This means PVC can achieve up to 1.74 Tera-ops/second for pure arithmetic operations, or up to 3.11 Tera-ops/second including memory and lane shift operations:

Arithmetic only (256 compute lanes):

1.74 Tops/sec = 426 MHz $\times$ 8 cores $\times$ (256 compute lanes/core $\times$ 2 ALUs/lane)

Arithmetic + memory/shifting (256 compute + 144 halo lanes):

3.11 Tops/sec = 426 MHz $\times$ 8 cores $\times$ (256$\times$2 compute + 400$\times$1 halo)

Pixel Visual Core brings Tera-ops/second performance to mobile devices in under 5 W.

6.3.2 HDR+ Benchmarks

In the Pixel 2 phone, Pixel Visual Core's primary application is accelerating the High Dynamic Range (HDR+) image processing pipeline. We use this pipeline as a benchmark for evaluation. Table 6.1 presents power and performance results for the three main HDR+ compute kernels running on PVC compared to an 8-core Snapdragon 835 application processor [Qua17]. The 10-nm Snapdragon 835 contains eight ARM Kryo 280 cores (four running at 2.4 GHz and four at 1.9 GHz) and a Hexagon 682 DSP coprocessor with HVX extensions [SLBL$^+$14].

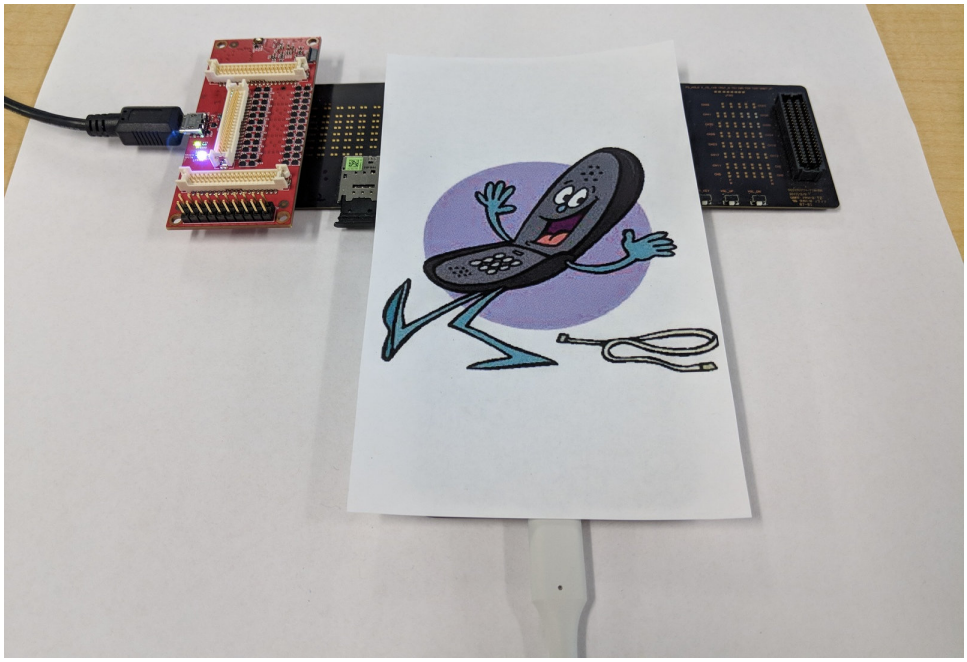


Figure 6.5. Pixel Visual Core experimental setup The breakout board exposes internal power rails for Snapdragon (with separate rails for the 2.4-GHz and 1.9-GHz CPU cores, DSP, DDR controller, DRAM, many others) and Pixel Visual Core (SoC with DDR controller, DRAM, others). Power, voltage, and current data are captured at 30 kHz and transferred to a host PC over USB. The Pixel 2 phone is intentionally covered in this pre-release photo.

Table 6.1. Pixel Visual Core power and performance results for HDR+ kernels Comparison of Pixel Visual Core vs. Snapdragon 835 for the three main compute kernels of HDR+. Pixel Visual Core is 7-16× more energy-efficient despite a three-generation process gap.

Kernel	**Snapdragon 835 (10 nm)**		**Pixel Visual Core (28 nm)**		**Pixel Visual Core vs. Snapdragon**	
	Time (ms)	Power (mW)	Time (ms)	Power (mW)	Time	Energy
Align	175	6909	62	2792	2.8×	6.9×
Merge	432	9970	154	3945	2.8×	7.1×
Finish	630	7672	108	2676	5.8×	16.7×

For both architectures we used the same algorithms written in Halide, but the scheduling was optimized for each target. The HDR+ kernels are processing a 12-megapixel image burst comprising five input frames. Measured Snapdragon power includes just the eight CPU cores and DSP (no DRAM, DDR controller, or MIPI), while PVC results conservatively include the entire IPU SoC (which includes its own DDR controller). For all kernels PVC is 2.8-5.8× faster and uses 6.9-16.7× less energy, despite being three process generations older and running at one-fifth the clock frequency. It should be noted that after 3-4 shots the Snapdragon slows down due to thermal throttling, while PVC can sustain 100s of shots with no throttling.

6.4 Summary

Pixel Visual Core's performance and energy efficiency further demonstrate the benefits of specialized architectures, adopted by industry in a commercial accelerator. PVC's Image Processing Unit is a programmable accelerator, which allows it to support ever-evolving algorithms and even new domains. The IPU is most efficient when applied to its target domain (image and vision processing), but it is general-purpose enough to also handle future workloads.

Acknowledgements

This chapter contains material from "Pixel Visual Core: Google's Fully Programmable Image, Vision and AI Processor for Mobile Devices," by Jason Redgrave, Albert Meixner, Nathan Goulding-Hotta, Artem Vasilyev, and Ofer Shacham, which has appeared in Hot Chips 30: A Symposium on High Performance Chips, ©2018 IEEE. The dissertation author is a primary contributor and third author of this paper.

Chapter 7

Related Work

This chapter presents an overview of related work in dark silicon research and specialized accelerators. More and more accelerators are appearing in chips across many domains, especially in mobile phones, datacenters, and emerging ambient computing. The recent explosion in accelerator research makes an exhaustive survey infeasible, but this chapter attempts to at least capture a review of highlights.

7.1 Dark Silicon Research

Although we didn't coin the term dark silicon ourselves, we were among the first to use it in the architecture research community because it so perfectly described the consequences of the utilization wall [GSV+10]. In fact, the term dark silicon was coined while our first conservation cores paper [VSG+10] was undergoing review for ASPLOS in 2009, and we had many prior submissions and grant proposals that discussed the dark silicon problem dating back to 2006. Our research was used as an input to the ITRS 2007 roadmap, which ARM CTO Mike Muller cited as his inspiration for the term, first used publicly at ARM's annual technical conference in 2009 [Mer09]. Our ASPLOS paper also had the first analysis of dark silicon's effect on multicore architectures.

Before dark silicon, many architects focused primarily on performance and silicon area, but that focus has shifted to power and energy as being first-order design constraints. To address

these new concerns, researchers have explored novel techniques at the level of transistors, circuits, microarchitectural pipelines, and entire cores, as well as considering dedicated vs. reconfigurable logic, and developing new models, languages, and design tools to help with scaling and automation.

The authors of [EBA+11] and [EBSA+12] model the limits of multicore scaling by combining device scaling, single-core scaling, and multicore scaling together to measure the speedup potential for a set of parallel workloads. Their work agrees with our findings that multicore scaling is limited by power in the dark silicon regime.

Focusing on server workloads, [HFFA11] and [Har12] argue that in the dark silicon regime traditional multicore CPUs won't scale beyond the low hundreds of cores, and instead propose building specialized CMPs with large and diverse arrays of heterogeneous cores.

Others model the effects of dark silicon in the "uncore" components of a chip, including shared cache and on-chip interconnect [CZZ+15], 3D CMPs [AOFJM17], and DRAM [PRH+17]. [LHWB12] presents the Accelerator Store, a scalable architectural component to minimize area by sharing SRAM memories in systems with many (tens to hundreds) of accelerators.

The authors of [HRSS11] and [WS13] examine dark silicon from the perspective of thermal design power (TDP), and they model tradeoffs between using area for application-specific accelerators vs. reconfigurable logic. For diverse workloads they suggest that reconfigurable logic supporting multiple accelerators can be more beneficial, especially when the accelerators have lower workload coverage individually. Some other examples of reconfigurable architectures include Plasticine [PZK+17], a spatially-reconfigurable architecture designed to execute applications composed of parallel patterns, and work on other coarse-grained reconfigurable architectures (CGRAs) such as [PFM+08], [PPM09], and [PPM12].

Computational sprinting [RLC+12] [RES+13] is a dim silicon approach that exploits the thermal capacitance of materials to allow silicon to operate in bursts above the chip's steady-state thermal limit. By "racing to idle" [AA14], sprinting can improve both performance and energy efficiency by idling "uncore" components more frequently.

Near-threshold voltage (NTV) computing, another dim silicon approach, offers extreme energy efficiency by lowering the operating voltage to just above V_t [PSD+12]. NTV computing makes up for the performance penalties with additional parallelism, scalable within limits modeled in [PDS+13].

Other researchers approach the dark silicon problem at the device level. Work on tunnel field-effect transistors (TFETs) [MMK+09] proposes a technique with heterogeneous device types, using a combination of energy-efficient cores built from TFETs in addition to traditional CMOS cores for the performance-critical phases of applications [SKS+11] [SKS+13]. This heterogeneous approach combines both dark (fewer cores, higher voltage) and dim (more cores, lower voltage) silicon techniques.

As discussed in Chapter 2, specialized solutions to the dark silicon problem will require imposing levels of scaling to generate the necessary numbers of accelerators. Some research aims to enable better automation through the use of new languages and design tools. Spatial [KFP+18] is a domain-specific language and compiler for higher-level descriptions of application accelerators. The language provides hardware-centric abstractions to improve programmer productivity and design performance, targeting FPGAs and CGRAs. Darkroom [HBD+14] compiles high-level image processing code to Verilog RTL for ASIC and FPGA hardware.

High-level synthesis (HLS) is a broad category of tools that seek to raise the abstraction level required to create accelerators. [Bab01] is early work targeting gate-reconfigurable architectures. SPARK [GDGN03] transforms behavioral descriptions in C into synthesizable VHDL. [MLAK16] is a more recent example that targets FPGAs. [NSP+16] provides a recent survey of academic and commercial HLS tools.

All of these among many others are excellent work looking at different aspects of the dark silicon problem and potential solutions. Many of these works propose specialization as critical to continued performance scaling in the dark silicon regime.

7.2 Mobile Phone/SoC Accelerators

The rise of all-in-one smartphones such as iPhone and Android devices has bolstered a recent explosion in specialized accelerators for mobile SoCs [SB15]. Today's mobile application processor SoCs couple multicore CPUs with GPUs and dozens of specialized accelerators. These accelerators offload functions such as graphics, digital signal processing, multimedia encode/decode, cryptography, security, image signal processing, machine learning, I/O, and more (for recent commercial examples see [Qua19][App19][Sam19][Hua19]).

Several researchers have created architectural simulators and tools to model accelerators on SoCs. Aladdin [SRWB14] is a pre-RTL accelerator simulator that speeds up SoC development by enabling larger, faster design space explorations for new accelerators in an SoC. Gables [HJ19] extends the Roofline model [WWP09] for accelerators on mobile SoCs, allowing architects to apportion work concurrently among different accelerators and calculate an SoC performance upper bound. [RYK18] presents an industry perspective of the challenges facing mobile computer architecture, and includes "ten commandments" for mobile processor design [RYK19].

Early mobile hardware accelerators include Digital Signal Processors (DSPs) that speed up complex data processing tasks using less energy than a general-purpose processor. Examples include Tensilica's fixed-function audio, voice, and vision DSPs, as well as its customizable Xtensa cores [WKMR01], which let architects extend their processors with application-specific instructions. Qualcomm's Hexagon DSPs [Cod13] have been increasing in core quantity and capabilities, including the recent addition of vector instruction extensions HVX [Cod15] and Hexagon Tensor Accelerator.

Machine learning accelerators abound. Domain-specific neural network accelerators can achieve more than two orders of magnitude improvements over CPUs and GPUs in terms of performance and energy efficiency [GYP+19]. Just a few examples include the work in [CDS+14], [DFC+15], [HLM+16], and [RWA+16], as well as the recently-announced successor to PVC, Pixel Neural Core [Rak19]. Another example is Celerity [AAHA+17][RZA+19], which

includes a tiered accelerator fabric (TAF) with three key architectural tiers: a general-purpose tier with OS-capable cores; a massively-parallel tier made of scalable programmable arrays of small, tightly coupled cores; and a specialization tier of highly specialized algorithmic accelerators built from HLS for specific computations, in particular convolutional neural networks [DXT+18].

Google's Titan M [Xin18] security chip, a mobile version of Titan [JRR+18], is a companion to the Pixel 3 application processor that provides a silicon root of trust. The chip is hardened against software and hardware attacks through mechanisms such as on-chip voltage, temperature, and glitch detectors, and physical defenses that make physical tampering difficult and detectable.

Although not necessarily designed for mobile phones (yet!), FPGA-based accelerators are used in multitudinous domains, including: neural network accelerators such as FastWave [HJN+19] and the work in [ZSZ+17], [SGK17], and [ZLS+15]; simultaneous localization and mapping (SLAM) [GAK19] [BB17]; 3D registration and mapping [BBK18]; approximate computing [LRY+16]; non-volatile memory (NVM) controllers [DGGS13] [CDC+10]; and many others.

The work discussed in this chapter just scratches the surface of the rich body of accelerator research in our community. In addition to mobile, there are countless accelerators targeting other domains. Datacenter accelerators include ASIC Clouds [MKGT16] such as Google's TPU [JYP+17] and Microsoft's FPGA-based accelerators [PCC+14][CCP+16][FOP+18]. An emerging class of ambient/always-on/edge computing includes accelerators such as Google's Edge TPU [Rhe18] and the Movidius Myriad 2 Vision Processing Unit [MBR+14], among others. We live in a golden age of specialized accelerator design!

Chapter 8

Synthesis and Conclusion

This final chapter presents a high-level synthesis[1] of the ideas and specialized architectures discussed in the dissertation. This dissertation presented the origins of the dark silicon problem, from the end of Dennard scaling to our predictions for future (now current) commercial processors in industry. Our principal prediction was an increase in the amount of silicon die area reserved for specialized accelerators that would take over more and more computation on complex SoCs. Growing the number of specialized accelerators requires tackling the challenges of massive complexity and scaling, where automation is key. Some of this automation can come from high-level synthesis tools, such as the conservation core toolchain. We presented conservation cores and applied them to the domain of mobile processing with GreenDroid. The author took the experience gained in this work and applied it in industry at Google, helping a team design and build the Pixel Visual Core accelerator. This chapter concludes with a discussion on trends in specialized hardware, including accelerators optimized for different metrics and targeting various domains (mobile, datacenter, ambient computing).

In Chapter 2 we showed how rising leakage currents led to the end of Dennard scaling, the stalled reduction of transistor threshold and supply voltages, and the exponential growth of dark silicon. The dark silicon problem forces architects into a new design regime, in which silicon area is considered cheap relative to power, which has become an expensive first-order design constraint. We recast the dark silicon problem as an opportunity to usher in a new era

[1] Pun very much intended.

in specialized computing. Our early research predicted that dark silicon would result in the acceleration[2] of work on accelerators in research and commercial processors, which has come true.

When the dissertation author first started at UCSD, it was just four days before Apple launched its now-ubiquitous iPhone. The first iPhone was powered by a relatively simple Samsung SoC with a 32-bit RISC ARMv6 processor in 90-nm technology. Since then smartphones including iPhone and Android devices have become the dominant computing platform of the world. Today's application processors contain complex heterogeneous superscalar and out-of-order multicore CPUs with integrated GPUs and dozens of other accelerators.

The author's first accelerator research included the design of conservation cores. Conservation cores lie near one end of the efficiency spectrum for specialized hardware shown previously in Figure 2.2. Besides the obvious benefit of energy efficiency, one of the key strengths of the c-cores approach is automation. Designing hardware accelerators by hand is a labor-intensive and error-prone process. Hand-written accelerators must be architected, implemented, and verified individually.

High-level synthesis can alleviate some of these problems. HLS tools seek to raise the level of abstraction required to create accelerators. HLS research has a long and rich history, which has culminated in the availability of several commercial tools, such as Mentor Graphics's Catapult C or Synopsys's Synphony ([NSP+16] provides a recent survey of academic and commercial tools).

Because traditional HLS tools seek to infer parallel execution from serial code, they have many of the same limitations that parallelizing compilers suffer from—namely, the difficulties of analyzing pointers in free-form code, extracting memory parallelism, and extracting and formulating efficient parallel schedules for the operations in critical loops. Without successful parallelization, the code is unlikely to run much faster in specialized silicon than it would on a general-purpose processor core.

[2] Also intended.

To address the parallelization challenges, most high-level synthesis tools place limits on the input language (for example, no pointers, no dynamic memory allocation, and no goto statements) and rely on user-transformed code or programmer annotations and pragmas to guide the tool to generate designs with good results. These tools' expected usage model is that the user will shepherd code through the tools, modifying the code and performing trial-and-error transformations to attain the expected quality of results. Typically, operating system and I/O code are considered unsynthesizable, and either the HLS tool ignores this code or the user must comment it out—further limiting the amount of code that can be converted into special-purpose hardware easily.

GreenDroid targets a system with millions of lines of difficult-to-parallelize code, including the Linux kernel, so its focus is different: the c-core HLS toolchain must automatically reduce the energy of large bodies of nonparallelizable code without user intervention. The code base is too large to afford manual intervention. Our toolchain doesn't need user pragmas for effective transformation or require any source code modifications to remove unsupported constructs. It supports code that has I/O and system calls, so even parts of the operating system can be translated. Through automation, the conservation cores approach provides the extreme level of scaling necessary under the severe constraints of the dark silicon regime.

At Google, the dissertation author joined a broad team to design and implement another specialized architecture, the Image Processing Unit and Pixel Visual Core. Compared to conservation cores, PVC is a more traditional parallel accelerator, providing massive parallelism with minimal overhead. The first generation of the IPU has 4,096 ALUs contained in eight 256-lane SIMD Stencil Processor cores. It achieves 3.1 Tera-ops/second performance within a 5-watt power budget through energy-efficient innovations that include a 2D shift network that enables efficient access to neighboring data for stencil computations, and 2D hardware line buffers that drastically reduce required DRAM accesses, lowering total energy and latency. As shown in Section 6.3, the 28-nm Pixel Visual Core is 7-16$\times$ more energy-efficient than a 10-nm Snapdragon SoC, despite a three-generation process gap.

Pixel Visual Core and conservation cores are complementary, since they target different kinds of code. PVC accelerates image processing and machine learning algorithms with plenty of parallelism and particular data access patterns. Jobs are fairly coarse-grained, allowing the CPU to offload large tasks to the off-chip accelerator, which has its own dedicated DRAM and control processor. C-cores, on the other hand, target irregular code that is hard to parallelize, i.e., all the leftovers. Because c-core jobs are fine-grained and require more frequent communication and data-sharing with a general-purpose host, c-cores are more closely coupled with a CPU, sharing the CPU's L1 data cache. Along with other specialized accelerators, Pixel Visual Core's IPU would be a great addition to GreenDroid alongside the Android c-cores.

Pixel Visual Core and conservation cores are optimized for similar, though slightly different, metrics. In the mobile domain, where power is very limited, the only way to get Tera-operations/second performance is to optimize performance per watt, also known as operations per picojoule (Ops/pJ). This is because:

$$\text{Power} = \text{Energy/Second} = \text{Ops/Second} \times \text{Energy/Op}$$

With a fixed power budget, the only way to increase performance (Ops/Second) is to decrease Energy/Op. Whereas Pixel Visual Core is optimized for performance per watt, conservation cores are optimized for a related metric, energy-delay product (EDP):

$$\text{EDP} = \text{Energy} \times \text{Delay} = \text{Energy} \times \text{Seconds/Op} = \text{Power/Op}$$

Optimizing for EDP differs from Energy/Op in that any decrease in Energy/Op yields immediate improvements in performance, while under EDP some performance degradation is acceptable if the c-cores can compensate with still-lower energy.

Depending on the domain and application, specialized accelerators can be designed to optimize for different metrics, for example performance per watt, total cost of ownership, latency, or ultra-low power. Accelerators can also be built to serve a specific functional purpose, such as demonstrating AI supremacy in the cloud [JYP+17] or bringing it to your hand [Rak19].

Datacenter accelerators approach specialization from another point of view. ASIC Clouds [MKGT16] may optimize for the total cost of ownership (TCO) of a system. TCO includes

upfront development and capital costs like silicon NRE, as well as ongoing maintenance costs including electricity, cooling, and capital interest payments. Although specialized accelerators may have a higher upfront development cost, they can reduce TCO because they decrease the need for so many general-purpose processors and the electricity to feed and cool them. Google's TPU [JYP+17] is a good example of this. Other accelerators like Microsoft's FPGA-based datacenter accelerators strive for minimum latency [PCC+14][CCP+16][FOP+18].

Ambient and edge computing promise more privacy and potentially better performance, since more data can remain local instead of being sent over a network to servers in the cloud. But always-on ambient computing must be even more conscious of power limitations because most of this computing runs on battery-powered devices [PCD+01]. Ambient computing accelerators must optimize for ultra-low energy to enable always-on processing and new features like personal assistants and on-device voice transcription [IS19].

Computing technology is no stranger to rapid disruption. Just as ARM displaced Intel in processor volume with the rise of smartphones and ubiquitous computing, another displacement may be just around the bend. There is a surge in open-source hardware, CAD tools, and even instruction sets—in particular RISC-V makes it very easy to add custom instructions to a processor, royalty free (watch out ARM!).

Computer architecture is all about levels of abstraction. In a processor, the instruction set is a contract that promises to execute specific operations without needing to provide details of the underlying implementation. In an ASIC standard cell library, the cells are mini black boxes that abstract CMOS transistors away into simple logic gates. With CAD synthesis tools, an RTL hardware description language can hide the details of the translation into standard cells. And the RTL itself can be generated from higher-level languages and a good HLS tool. With a "sufficiently high enough" HLS tool, the vision of going from natural language descriptions of algorithms to push-button systems of specialized accelerators is not so far away.

Appendix A

Acronyms

ALU	Arithmetic Logic Unit
AMBA	Advanced Microcontroller Bus Architecture (ARM)
AP	Application Processor
APB	Advanced Peripheral Bus (ARM)
API	Application Programming Interface
AXI	Advanced eXtensible Interface (ARM)
BGA	Ball Grid Array
CFG	Control Flow Graph
CMOS	Complementary Metal Oxide Semiconductor
CMP	Chip Multiprocessor
CNN	Convolutional Neural Network
CoDA	Coprocessor-Dominated Architecture
CPU	Central Processing Unit
CSI-2	Camera Serial Interface 2 (MIPI)
CSR	Control and Status Register
DAG	Directed Acyclic Graph
DFG	Data Flow Graph
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processor
GPIO	General Purpose I/O
GPU	Graphics Processing Unit
HDR+	High Dynamic Range algorithm [HSG+16]
HKMG	High-k Metal Gate
HLS	High-Level Synthesis
I2C	Inter-Integrated Circuit
IPU	Image Processing Unit
IRDS	International Roadmap for Devices and Systems
ISP	Image Signal Processor
ITRS	International Technology Roadmap for Semiconductors

JTAG	Joint Test Action Group (debug standard)
LBP	Line Buffer Pool (IPU)
LPDDR	Low-Power Double Data Rate DRAM
MAD	Multiply-Add Unit
MIPI	Mobile Industry Processor Interface
MMU	Memory Management Unit
NoC	Network-on-Chip
OCN	On-Chip Network
PCIe	Peripheral Component Interconnect Express
PDK	Process Design Kit
pISA	Physical Instruction Set Architecture (IPU)
PLL	Phase-Locked Loop
PMIC	Power Management Integrated Circuit
PVC	Pixel Visual Core
PVT	Process, Voltage, and Temperature corner
RTL	Register Transfer Level
SCL	Scalar Lane (IPU)
SDP	Selective Depipelining (c-cores)
SHG	Sheet Generator (IPU)
SIMD	Single Instruction Multiple Data
SiP	System-in-Package
SoC	System-on-Chip
SPI	Serial Peripheral Interface
STP	Stencil Processor (IPU)
TSMC	Taiwan Semiconductor Manufacturing Company
UART	Universal Asynchronous Receiver-Transmitter
vISA	Virtual Instruction Set Architecture (IPU)
VLIW	Very Long Instruction Word
Vt	Transistor Threshold Voltage
WTF	Wow, Thesis Finished!

Bibliography

[AA14] Susanne Albers and Antonios Antoniadis. Race to idle: New algorithms for speed scaling with a sleep state. *ACM Trans. Algorithms*, 10(2), February 2014.

[AAHA+17] Tutu Ajayi, Khalid Al-Hawaj, Aporva Amarnath, Steve Dai, Scott Davidson, Paul Gao, Gai Liu, Atieh Lotfi, Julian Puscar, Anuj Rao, Austin Rovinski, Loai Salem, Ningxiao Sun, Christopher Torng, Luis Vega, Bandhav Veluri, Xiaoyang Wang, Shaolin Xie, Chun Zhao, Ritchie Zhao, Christopher Batten, Ronald G. Dreslinski, Ian Galton, Rajesh K. Gupta, Patrick P. Mercier, Mani Srivastava, Michael Bedford Taylor, and Zhiru Zhang. Celerity: An open source RISC-V tiered accelerator fabric. In *IEEE Hot Chips 29 Symposium (HCS)*, Aug 2017.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[AOFJM17] Arghavan Asad, Ozcan Ozturk, Mahmood Fathy, and Mohammad Reza Jahed-Motlagh. Optimization-based power and thermal management for dark silicon aware 3D chip multiprocessors using heterogeneous cache hierarchy. *Microprocessors and Microsystems*, 51:76–98, April 2017.

[App19] Apple. Apple A13 Bionic. https://en.wikipedia.org/wiki/Apple_A13, 2019.

[ARM] ARM. ARM Cortex-A53. https://developer.arm.com/products/processors/cortex-a/cortex-a53.

[AS01] Shail Aditya and Michael S. Schlansker. ShiftQ: A bufferred interconnect for custom loop accelerators. In *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '01, pages 158–167, New York, NY, USA, 2001. Association for Computing Machinery.

[ASGH+11] M. Arora, J. Sampson, N. Goulding-Hotta, J. Babb, G. Venkatesh, M. B. Taylor, and S. Swanson. Reducing the energy cost of irregular code bases in soft processor systems. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 210–213, May 2011.

[Bab01] Jonathan William Babb. *High Level Compilation for Gate Reconfigurable Architectures*. PhD thesis, Massachusetts Institute of Technology, 2001.

[BB17] K. Boikos and C. Bouganis. A high-performance system-on-chip architecture for direct tracking for SLAM. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–7, Sep. 2017.

[BBK18] M. Barrow, S. M. Burns, and R. Kastner. A FPGA accelerator for real-time 3D non-rigid registration using tree reweighted message passing and dynamic Markov random field generation. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 335–342, Aug 2018.

[BGHZ+12] Vikram Bhatt, Nathan Goulding-Hotta, Qiaoshi Zheng, Jack Sampson, Steven Swanson, and Michael Bedford Taylor. SiChrome: Mobile web browsing in hardware to save energy. In *DaSi: First Dark Silicon Workshop, ISCA*, June 2012.

[Bry09] Vladyslav Sergeevich Bryksin. ASIC life extension through hardware patch interfaces. Master's thesis, University of California San Diego, 2009.

[BVCG04] Mihai Budiu, Girish Venkataramani, Tiberiu Chelcea, and Seth Copen Goldstein. Spatial computation. In *Proceedings of the Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 14–26, New York, NY, USA, 2004. ACM.

[Car16] David Cardinal. Pixel smartphone camera review: At the top. https://www.dxomark.com/pixel-smartphone-camera-review-at-the-top, Oct 2016.

[Car17] David Cardinal. Google Pixel 2 reviewed: Sets new record for overall smartphone camera quality. https://www.dxomark.com/google-pixel-2-reviewed-sets-new-record-smartphone-camera-quality, Oct 2017.

[Car18] David Cardinal. Google Pixel 3 camera review: The best Pixel yet. https://www.dxomark.com/google-pixel3-camera-review, Dec 2018.

[CCP+16] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016.

[CDC+10] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 385–395, Dec 2010.

[CDS+14] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. DianNao: A small-footprint high-throughput accelerator

for ubiquitous machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 269–284, New York, NY, USA, 2014. Association for Computing Machinery.

[CGG+12] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. Architecture support for accelerator-rich CMPs. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 843–849, New York, NY, USA, 2012. Association for Computing Machinery.

[Che] Stephen Checkoway. UCSD dissertation LaTeX class file. https://github.com/stevecheckoway/ucsddissertation.

[CHM08] N. Clark, A. Hormati, and S. Mahlke. VEAL: Virtualized Execution Accelerator for Loops. In *35th Annual International Symposium on Computer Architecture (ISCA)*, pages 389–400, June 2008.

[CMHM10] Eric S. Chung, Peter A. Milder, James C. Hoe, and Ken Mai. Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs? In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 225–236, USA, 2010. IEEE Computer Society.

[Cod13] L. Codrescu. Qualcomm Hexagon DSP: An architecture optimized for mobile multimedia and communications. In *IEEE Hot Chips 25 Symposium (HCS)*, Aug 2013.

[Cod15] L. Codrescu. Architecture of the Hexagon 680 DSP for mobile imaging and computer vision. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–26, Aug 2015.

[CZZ+15] H. Cheng, J. Zhan, J. Zhao, Y. Xie, J. Sampson, and M. J. Irwin. Core vs. uncore: The heart of darkness. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2015.

[DB18] Andrea Di Blas. Keynote: Google Pixel Visual Core: A portable domain-specific processor for computational photography and machine learning. In *2018 IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, July 2018.

[DFC+15] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. ShiDianNao: Shifting vision processing closer to the sensor. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 92–104, New York, NY, USA, 2015. Association for Computing Machinery.

[DGGS13] A. De, M. Gokhale, R. Gupta, and S. Swanson. Minerva: Accelerating data analysis in next-generation SSDs. In *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 9–16, April 2013.

[DGR+74] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, Oct 1974.

[Dic12] Rigo Dicochea. 28SLP Catalyst Project User Documentation. Technical report, UCSC and GlobalFoundries, 2012.

[DR12] Rigo Dicochea and Jose Renau. Multi-University Research Network. Technical report, UCSC and GlobalFoundries, 2012.

[DWB+10] R. G. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge. Near-threshold computing: Reclaiming Moore's law through energy efficient integrated circuits. *Proceedings of the IEEE*, 98(2):253–266, Feb 2010.

[DXT+18] S. Davidson, S. Xie, C. Torng, K. Al-Hawai, A. Rovinski, T. Ajayi, L. Vega, C. Zhao, R. Zhao, S. Dai, A. Amarnath, B. Veluri, P. Gao, A. Rao, G. Liu, R. K. Gupta, Z. Zhang, R. Dreslinski, C. Batten, and M. B. Taylor. The Celerity open-source 511-core RISC-V tiered accelerator fabric: Fast architectures and design methodologies for fast chips. *IEEE Micro*, 38(2):30–41, Mar 2018.

[EBA+11] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, June 2011.

[EBSA+12] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Power limitations and dark silicon challenge the future of multicore. *ACM Trans. Comput. Syst.*, 30(3), August 2012.

[eli] Embedded Linux. https://www.elinux.org.

[FKDM09] K. Fan, M. Kudlur, G. Dasika, and S. Mahlke. Bridging the computation gap between programmable processors and hardwired accelerators. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pages 313–322, Feb 2009.

[FOP+18] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger. A configurable cloud-scale DNN processor for real-time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, June 2018.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[GAK19] Q. Gautier, A. Althoff, and R. Kastner. FPGA architectures for real-time dense SLAM. In *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, volume 2160-052X, pages 83–90, July 2019.

[Gar18] Gartner. Gartner says worldwide sales of smartphones returned to growth in first quarter of 2018. https://www.gartner.com/en/newsroom/press-releases/2018-05-29-gartner-says-worldwide-sales-of-smartphones-returned-to-growth-in-first-quarter-of-2018, May 2018.

[GDGN03] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. SPARK: A high-level synthesis framework for applying parallelizing compiler transformations. In *16th International Conference on VLSI Design*, pages 461–466, Jan 2003.

[GHSV+11] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. B. Taylor. The GreenDroid mobile application processor: An architecture for silicon's dark future. *IEEE Micro*, 31(2):86–95, March 2011.

[GHSZ+12] N. Goulding-Hotta, J. Sampson, Q. Zheng, V. Bhatt, J. Auricchio, S. Swanson, and M. B. Taylor. GreenDroid: An architecture for the dark silicon age. In *17th Asia and South Pacific Design Automation Conference*, pages 100–105, Jan 2012.

[Gooa] Google. Android platform architecture. https://developer.android.com/guide/platform.

[Goob] Google. Android Runtime (ART) and Dalvik. https://source.android.com/devices/tech/dalvik/index.html.

[Gra] GrammaTech. CodeSurfer. http://www.grammatech.com/products/codesurfer.

[GSV+10] N. Goulding, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, J. Babb, M. B. Taylor, and S. Swanson. GreenDroid: A mobile application processor for a future of dark silicon. In *IEEE Hot Chips 22 Symposium (HCS)*, Aug 2010.

[GYP+19] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. TANGRAM: Optimized coarse-grained dataflow for scalable NN accelerators. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 807–820, New York, NY, USA, 2019. Association for Computing Machinery.

[hal] Halide: A language for fast, portable computation on images and tensors. http://halide-lang.org.

[HAP+05] M. Horowitz, E. Alon, D. Patil, S. Naffziger, R. Kumar, and K. Bernstein. Scaling, power, and the future of CMOS. In *IEEE International Electron Devices Meeting (IEDM)*, pages 7–15, Dec 2005.

[Har12] Nikos Hardavellas. The rise and fall of dark silicon. *USENIX ;login:*, 37:7–17, April 2012.

[HBD+14] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. Darkroom: Compiling high-level image processing code into hardware pipelines. *ACM Trans. Graph.*, 33(4), 2014.

[HFFA11] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4):6–15, July 2011.

[HJ19] M. Hill and V. Janapa Reddi. Gables: A roofline model for mobile SoCs. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 317–330, Feb 2019.

[HJN+19] S. Hussain, M. Javaheripi, P. Neekhara, R. Kastner, and F. Koushanfar. FastWave: Accelerating autoregressive convolutional neural networks on FPGA. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, Nov 2019.

[HLM+16] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. EIE: Efficient Inference Engine on compressed deep neural network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 243–254, June 2016.

[HP17] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition, 2017.

[HQW+10] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In *37th Annual International Symposium on Computer Architecture (ISCA)*, pages 37–47, New York, NY, USA, 2010. Association for Computing Machinery.

[HRSS11] W. Huang, K. Rajamani, M. R. Stan, and K. Skadron. Scaling with design constraints: Predicting the future of big chips. *IEEE Micro*, 31(4):16–29, July 2011.

[HSG+16] Samuel W. Hasinoff, Dillon Sharlet, Ryan Geiss, Andrew Adams, Jonathan T. Barron, Florian Kainz, Jiawen Chen, and Marc Levoy. Burst photography for high dynamic range and low-light imaging on mobile cameras. *ACM Trans. Graph.*, 35(6):192:1–192:12, November 2016.

[Hua19] Huawei. Huawei Kirin 990 series, rethink evolution. https://consumer.huawei.com/en/campaign/kirin-990-series, 2019.

[ird17] International Roadmap for Devices and Systems. https://irds.ieee.org, 2017.

[IS19] Itay Inbar and Nir Shemy. The on-device machine learning behind Recorder. https://ai.googleblog.com/2019/12/the-on-device-machine-learning-behind.html, Dec 2019.

[itr09] International Technology Roadmap for Semiconductors. http://www.itrs2.net/itrs-reports.html, 2009.

[Jia13] Fei Jia. The Arsenal tool chain for the GreenDroid mobile application processor. Master's thesis, University of California San Diego, 2013.

[JRR+18] Scott Johnson, Dominic Rizzo, Parthasarathy Ranganathan, Jon McCune, and Richard Ho. Titan: enabling a transparent silicon root of trust for cloud. In *IEEE Hot Chips 30 Symposium (HCS)*, Aug 2018.

[JYP+17] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a Tensor Processing Unit. In *44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12, June 2017.

[KAS+02] V. Kathail, S. Aditya, R. Schreiber, B. Ramakrishna Rau, D. C. Cronquist, and M. Sivaraman. PICO: Automatically designing custom computers. *Computer*, 35(9):39–47, Sep. 2002.

[KFP+18] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 296–311, New York, NY, USA, 2018. Association for Computing Machinery.

[KLG14] Sudipta Kundu, Sorin Lerner, and Rajesh K. Gupta. *High-Level Verification: Methods and Tools for Verification of System-Level Designs*. Springer, New York, 2014.

[KTMW03] J. S. Kim, M. B. Taylor, J. Miller, and D. Wentzlaff. Energy characterization of a tiled architecture processor with on-chip networks. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pages 424–427, Aug 2003.

[LA04] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization (CGO)*, pages 75–86, March 2004.

[LHWB12] Michael J. Lyons, Mark Hempstead, Gu-Yeon Wei, and David Brooks. The accelerator store: A shared memory framework for accelerator-based systems. *ACM Trans. Archit. Code Optim.*, 8(4), January 2012.

[LRY+16] A. Lotfi, A. Rahimi, A. Yazdanbakhsh, H. Esmaeilzadeh, and R. K. Gupta. Grater: An approximation workflow for exploiting data-level parallelism in FPGA acceleration. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1279–1284, March 2016.

[Mar10] Jose E. Lugo Martinez. Strategies for sharing a floating point unit between SPEs. Master's thesis, University of California San Diego, 2010.

[MBR+14] D. Moloney, B. Barry, R. Richmond, F. Connor, C. Brick, and D. Donohoe. Myriad 2: Eye of the computational vision storm. In *2014 IEEE Hot Chips 26 Symposium (HCS)*, pages 1–18, Aug 2014.

[MCC+06] Mahim Mishra, Timothy J. Callahan, Tiberiu Chelcea, Girish Venkataramani, Seth C. Goldstein, and Mihai Budiu. Tartan: Evaluating spatial computation for whole program execution. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 163–174, New York, NY, USA, 2006. Association for Computing Machinery.

[Mer09] Rick Merritt. ARM CTO: power surge could create 'dark silicon'. https://www.eetimes.com/document.asp?doc_id=1172049, Oct 2009.

[Mil16] Peyman Milanfar. Enhance! RAISR sharp images with machine learning. https://ai.googleblog.com/2016/11/enhance-raisr-sharp-images-with-machine.html, Nov 2016.

[MIP] MIPI Alliance. MIPI Camera Serial Interface 2 (MIPI CSI-2). https://mipi.org/specifications/csi-2.

[MKGT16] I. Magaki, M. Khazraee, L. V. Gutierrez, and M. B. Taylor. ASIC Clouds: Specializing the datacenter. In *43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 178–190, June 2016.

[MLAK16] J. Matai, D. Lee, A. Althoff, and R. Kastner. Composable, parameterizable templates for high-level synthesis. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 744–749, March 2016.

[MMK+09] S. Mookerjea, D. Mohata, R. Krishnan, J. Singh, A. Vallett, A. Ali, T. Mayer, V. Narayanan, D. Schlom, A. Liu, and S. Datta. Experimental demonstration of 100nm channel length In0.53Ga0.47As-based vertical inter-band tunnel field effect transistors (TFETs) for ultra low-power logic and SRAM applications. In *2009 IEEE International Electron Devices Meeting (IEDM)*, pages 1–3, Dec 2009.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[NSP+16] R. Nane, V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, Oct 2016.

[ope] OpenIMPACT. http://impact.crhc.illinois.edu.

[PCC+14] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *41st Annual International Symposium on Computer Architecture (ISCA)*, pages 13–24, June 2014.

[PCD+01] D. Panigrahi, C. Chiasserini, S. Dey, R. Rao, A. Raghunathan, and K. Lahiri. Battery life estimation of mobile embedded systems. In *Fourteenth International Conference on VLSI Design*, pages 57–63, Jan 2001.

[PDS+13] N. Pinckney, R. G. Dreslinski, K. Sewell, D. Fick, T. Mudge, D. Sylvester, and D. Blaauw. Limits of parallelism and boosting in dim silicon. *IEEE Micro*, 33(5):30–37, Sep. 2013.

[PFM+08] Hyunchul Park, Kevin Fan, Scott A. Mahlke, Taewook Oh, Heeseok Kim, and Hong-seok Kim. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 166–176, New York, NY, USA, 2008. ACM.

[PPM09] Yongjun Park, Hyunchul Park, and Scott Mahlke. CGRA Express: Accelerating execution using dynamic operation fusion. In *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '09, pages 271–280, New York, NY, USA, 2009. Association for Computing Machinery.

[PPM12] Y. Park, J. J. K. Park, and S. Mahlke. Efficient performance scaling of future CGRAs for mobile applications. In *International Conference on Field-Programmable Technology*, pages 335–342, Dec 2012.

[PRH+17] A. Pedram, S. Richardson, M. Horowitz, S. Galal, and S. Kvatinsky. Dark memory and accelerator-rich system optimization in the dark silicon era. *IEEE Design & Test*, 34(2):39–50, April 2017.

[PSD+12] N. Pinckney, K. Sewell, R. G. Dreslinski, D. Fick, T. Mudge, D. Sylvester, and D. Blaauw. Assessing the performance limits of parallelized near-threshold computing. In *DAC Design Automation Conference 2012*, pages 1143–1148, June 2012.

[PZK+17] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun. Plasticine: A reconfigurable architecture for parallel patterns. In *44th Annual International Symposium on Computer Architecture (ISCA)*, pages 389–402, June 2017.

[Qua17] Qualcomm. Snapdragon 835 mobile platform. https://www.qualcomm.com/products/snapdragon-835-mobile-platform, 2017.

[Qua19] Qualcomm. Snapdragon 855 mobile platform. https://www.qualcomm.com/products/snapdragon-855-mobile-platform, 2019.

[Rak19] Brian Rakowski. Pixel 4 is here to help. https://blog.google/products/pixel/pixel-4, Oct 2019.

[RES+13] A. Raghavan, L. Emurian, L. Shao, M. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. M. K. Martin. Utilizing dark silicon to save energy with computational sprinting. *IEEE Micro*, 33(5):20–28, Sep. 2013.

[Rhe18] Injong Rhee. Bringing intelligence to the edge with Cloud IoT. https://www.blog.google/products/google-cloud/bringing-intelligence-to-the-edge-with-cloud-iot, Jul 2018.

[Ric11] Scott Ricketts. Efficient cache-coherent migration for heterogeneous coprocessors in dark silicon limited technology. Master's thesis, University of California San Diego, 2011.

[RIM17] Y. Romano, J. Isidoro, and P. Milanfar. RAISR: Rapid and Accurate Image Super Resolution. *IEEE Transactions on Computational Imaging*, 3(1):110–125, March 2017.

[RKBA+13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language*

Design and Implementation, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[RLC+12] A. Raghavan, Y. Luo, A. Chandawalla, M. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. M. K. Martin. Computational sprinting. In *IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–12, Feb 2012.

[RMGH+18] J. Redgrave, A. Meixner, N. Goulding-Hotta, A. Vasilyev, and O. Shacham. The Pixel Visual Core: Google's fully programmable image, vision and AI processor for mobile devices. In *IEEE Hot Chips 30 Symposium (HCS)*, Aug 2018.

[RNR+11] E. Rotem, A. Naveh, D. Rajwan, A. Ananthakrishnan, and E. Weissmann. Power management architecture of the 2nd generation Intel core microarchitecture, formerly codenamed Sandy Bridge. In *IEEE Hot Chips 23 Symposium (HCS)*, Aug 2011.

[RWA+16] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernandez-Lobato, G. Wei, and D. Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 267–278, June 2016.

[RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 12–27, New York, NY, USA, 1988. ACM.

[RYK18] V. J. Reddi, H. Yoon, and A. Knies. Two billion devices and counting: An industry perspective on the state of mobile computer architecture. *IEEE Micro*, 38(1):6–21, January 2018.

[RYK19] Vijay Janapa Reddi, Hongil Yoon, and Allan Knies. Google's ten commandments for mobile computing. In *Infrastructure and Methodology for SoC Performance & Power Modeling Workshop, ASPLOS*, April 2019.

[RZA+19] A. Rovinski, C. Zhao, K. Al-Hawaj, P. Gao, S. Xie, C. Torng, S. Davidson, A. Amarnath, L. Vega, B. Veluri, A. Rao, T. Ajayi, J. Puscar, S. Dai, R. Zhao, D. Richmond, Z. Zhang, I. Galton, C. Batten, M. B. Taylor, and R. G. Dreslinski. Evaluating Celerity: A 16-nm 695 giga-RISC-V instructions/s manycore processor with synthesizable PLL. *IEEE Solid-State Circuits Letters*, 2(12):289–292, Dec 2019.

[SAB+06] N. Singh, A. Agarwal, L.K. Bera, T.Y. Liow, R. Yang, S.C. Rustagi, C.H. Tung, R. Kumar, G.Q. Lo, N. Balasubramanian, and D.-L. Kwong. High-performance fully depleted silicon nanowire (diameter$\leq$5 nm) gate-all-around CMOS devices. *IEEE Electron Device Letters*, 27(5):383–386, May 2006.

[SAGH+11] J. Sampson, M. Arora, N. Goulding-Hotta, G. Venkatesh, J. Babb, V. Bhatt, S. Swanson, and M. B. Taylor. An evaluation of selective depipelining for FPGA-based energy-reducing irregular code coprocessors. In *2011 21st International Conference on Field Programmable Logic and Applications*, pages 24–29, Sept 2011.

[Sam19] Samsung. Exynos 9825 processor: Specs, features. https://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-9825, 2019.

[SAR+00] R. Schreiber, S. Aditya, B. Ramakrishna Rau, V. Kathail, S. Mahlke, S. Abraham, and G. Snider. High-level synthesis of nonprogrammable hardware accelerators. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pages 113–124, July 2000.

[SB15] Y. S. Shao and D. Brooks. *Research Infrastructures for Hardware Accelerators*. Morgan & Claypool, 2015.

[SGK17] M. Samragh, M. Ghasemzadeh, and F. Koushanfar. Customizing neural networks for efficient FPGA implementation. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 85–92, April 2017.

[Sha11] Ofer Shacham. *Chip Multiprocessor Generator: Automatic Generation of Custom and Heterogeneous Compute Platforms*. PhD thesis, Stanford University, 2011.

[Sha18] Ofer Shacham. Use Pixel 2 for better photos in Instagram, WhatsApp and Snapchat. https://www.blog.google/products/pixel/use-pixel-2-better-photos-instagram-whatsapp-and-snapchat, Feb 2018.

[SKS+11] K. Swaminathan, E. Kultursay, V. Saripalli, V. Narayanan, M. Kandemir, and S. Datta. Improving energy efficiency of multi-threaded applications using heterogeneous CMOS-TFET multicores. In *IEEE/ACM International Symposium on Low Power Electronics and Design*, pages 247–252, Aug 2011.

[SKS+13] K. Swaminathan, E. Kultursay, V. Saripalli, V. Narayanan, M. T. Kandemir, and S. Datta. Steep-slope devices: From dark to dim silicon. *IEEE Micro*, 33(5):50–59, Sep. 2013.

[SLBL+14] M. Saint-Laurent, P. Bassett, K. Lin, Y. Wang, S. Le, X. Chen, M. Alradaideh, T. Wernimont, K. Ayyar, D. Bui, D. Galbi, A. Lester, and W. Anderson. 10.1 A 28nm DSP powered by an on-chip LDO for high-performance and energy-efficient mobile applications. In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 176–177, Feb 2014.

[SMSO03] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. WaveScalar. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-36, pages 291–302, Dec 2003.

[SNH+03] K. Sankaralingam, R. Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *30th Annual International Symposium on Computer Architecture (ISCA)*, pages 422–433, June 2003.

[SR17] Ofer Shacham and Masumi Reynders. Pixel Visual Core: image processing and machine learning on Pixel 2. https://www.blog.google/products/pixel/pixel-visual-core-image-processing-and-machine-learning-pixel-2, Oct 2017.

[SRWB14] Y. S. Shao, B. Reagen, G. Wei, and D. Brooks. Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *41st Annual International Symposium on Computer Architecture (ISCA)*, pages 97–108, June 2014.

[SRWB15] Y. S. Shao, B. Reagen, G. Wei, and D. Brooks. The Aladdin approach to accelerator design and modeling. *IEEE Micro*, 35(3):58–70, May 2015.

[SVGH+11] J. Sampson, G. Venkatesh, N. Goulding-Hotta, S. Garcia, S. Swanson, and M. B. Taylor. Efficient complex operators for irregular codes. In *IEEE 17th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 491–502, Feb 2011.

[TAB+] Chinh Tran, Chijioke Anyanwu, Sanjai Balakrishnan, Anshul Bhargava, James Jiang, and Radhika Thekkath. The MIPS32 24KE core family: High-performance RISC cores with DSP enhancements. https://www.mips.com.

[Tay12] M. B. Taylor. Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse. In *Design Automation Conference (DAC)*, pages 1131–1136, June 2012.

[Tay13] M. B. Taylor. A landscape of the new dark silicon design regime. *IEEE Micro*, 33(5):8–19, Sep. 2013.

[TLM+04] Michael B. Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *31st Annual International Symposium on Computer Architecture (ISCA)*, pages 2–13, June 2004.

[TMAJ08] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P. Jouppi. CACTI 5.1 Technical Report. https://www.hpl.hp.com/techreports/2008/HPL-2008-20.pdf, 2008.

[TSMa] TSMC. TSMC 28nm Technology. https://www.tsmc.com/english/dedicatedFoundry/technology/28nm.htm.

[TSMb] TSMC. TSMC 45nm Technology. https://www.synopsys.com/dw/emllselector.php?f=TSMC&n=45&s=wMoRVw.

[VBP+16] A. Vasilyev, N. Bhagdikar, A. Pedram, S. Richardson, S. Kvatinsky, and M. Horowitz. Evaluating programmable architectures for imaging and vision applications. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016.

[VSG+10] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation Cores: Reducing the energy of mature computations. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 205–218, New York, NY, USA, 2010. ACM.

[VSGH+11] Ganesh Venkatesh, Jack Sampson, Nathan Goulding-Hotta, Sravanthi Kota Venkata, Michael Bedford Taylor, and Steven Swanson. QsCores: Trading dark silicon for scalable energy efficiency with quasi-specific cores. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 163–174, New York, NY, USA, 2011. ACM.

[VSL08] F. Vahid, G. Stitt, and R. Lysecky. Warp processing: Dynamic translation of binaries to FPGA circuits. *Computer*, 41(7):40–46, July 2008.

[WKMR01] A. Wang, E. Killian, D. Maydan, and C. Rowen. Hardware/software instruction set configurability for system-on-chip processors. In *Proceedings of the 38th Design Automation Conference*, pages 184–188, June 2001.

[WS13] L. Wang and K. Skadron. Implications of the power wall: Dim cores and reconfigurable logic. *IEEE Micro*, 33(5):40–48, Sep. 2013.

[WWP09] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[Xin18] Xiaowen Xin. Titan M makes Pixel 3 our most secure phone yet. https://www.blog.google/products/pixel/titan-m-makes-pixel-3-our-most-secure-phone-yet, Oct 2018.

[YGBT09] S. Yehia, S. Girbal, H. Berry, and O. Temam. Reconciling specialization and flexibility through compound circuits. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pages 277–288, Feb 2009.

[YHD+19] Lucie Yahiaoui, Jonathan Horgan, Brian Deegan, Senthil Yogamani, Ciaran Hughes, and Patrick Denny. Overview and empirical analysis of ISP parameter tuning for visual perception in autonomous driving. *Journal of Imaging*, 5(10):78, 2019.

[YW18] Daniel Yang and Stacy Wegner. Samsung Galaxy S9 teardown. https://www.techinsights.com/blog/samsung-galaxy-s9-teardown, Mar 2018.

[ZGHR+14] Qiaoshi Zheng, Nathan Goulding-Hotta, Scott Ricketts, Steven Swanson, Michael Bedford Taylor, and Jack Sampson. Exploring energy scalability in coprocessor-dominated architectures for dark silicon. *ACM Trans. Embed. Comput. Syst.*, 13(4s):130:1–130:24, April 2014.

[ZLS+15] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pages 161–170, New York, NY, USA, 2015. Association for Computing Machinery.

[ZSZ+17] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. Accelerating binarized convolutional neural networks with software-programmable FPGAs. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 15–24, New York, NY, USA, 2017. Association for Computing Machinery.