

# Embedded Systems

ARM® Programming and Optimization

Jason D. Bakos

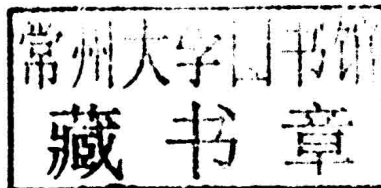
```
#pragma omp parallel for
for (i=0;i<N/4;i+=4) {
    asm("pld %[nextx]\n\t"
        "mov r0,%[coeff_addr]\n\t"
        "vldl.32 {q0},%[x_addr]@128\n\t"
        "vldl.32 {q1},[r0]!@128\n\t"
        "vldl.32 {q2},[r0]!@128\n\t"
        "vldl.32 {q3},[r0]!@128\n\t"
        "vldl.32 {q4},[r0]!@128\n\t"
        "vldl.32 {q5},[r0]!@128\n\t"
        "vldl.32 {q6},[r0]!@128\n\t"
        "vldl.32 {q7},[r0]!@128\n\t"
        "vldl.32 {q8},[r0]!@128\n\t"
        "vmla.f32 q2, q0, q1\n\t"
        "vmla.f32 q3, q0, q2\n\t"
        "vmla.f32 q4, q0, q3\n\t"
        "vmla.f32 q5, q0, q4\n\t"
        "vmla.f32 q6, q0, q5\n\t"
        "vmla.f32 q7, q0, q6\n\t"
        "vmla.f32 q8, q0, q7\n\t"
        "vstl.32 {q8}, %[d_ad-
dr]@128\n\t" : : [x_addr]"m"(x[i]), [nextx-
]"m"(x[i+16]), [d_addr]"m"(d[i]), [coeff_ad-
dr]"r"(coeff_4vector) : "r0","r1","r2","r3","q1",
"q2","q3");
```

# Embedded Systems

## ARM<sup>®</sup> Programming and Optimization

**Jason D. Bakos**

Department of Computer Science and Engineering  
University of South Carolina  
Columbia, SC



AMSTERDAM • BOSTON • HEIDELBERG • LONDON  
NEW YORK • OXFORD • PARIS • SAN DIEGO  
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann is an imprint of Elsevier



**Acquiring Editor:** *Steve Merken*  
**Editorial Project Manager:** *Nathaniel McFadden*  
**Project Manager:** *Sujatha Thirugnana Sambandam*  
**Designer:** *Mark Rogers*

Morgan Kaufmann is an imprint of Elsevier  
225 Wyman Street, Waltham, MA 02451, USA

Copyright © 2016 Elsevier Inc. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from the publisher. Details on how to seek permission, further information about the Publisher's permissions policies and our arrangements with organizations such as the Copyright Clearance Center and the Copyright Licensing Agency, can be found at our website: [www.elsevier.com/permissions](http://www.elsevier.com/permissions).

This book and the individual contributions contained in it are protected under copyright by the Publisher (other than as may be noted herein).

#### Notices

Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods, professional practices, or medical treatment may become necessary.

Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information, methods, compounds, or experiments described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

#### Library of Congress Cataloging-in-Publication Data

A catalog record for this book is available from the Library of Congress

#### British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library

ISBN: 978-0-12-800342-8

For information on all Morgan Kaufmann publications  
visit our website at <http://store.elsevier.com/>



Working together  
to grow libraries in  
developing countries

[www.elsevier.com](http://www.elsevier.com) • [www.bookaid.org](http://www.bookaid.org)

# Embedded Systems



*For Lumi, Jade, and Justin*



# Preface

For many years I have worked in the area of *reconfigurable computing*, whose goal is to develop tools and methodologies to facilitate the use of field programmable gate arrays (FPGAs) as co-processors for high-performance computer systems.

One of the main challenges in this discipline is the “programming problem,” in which the practical application of FPGAs is fundamentally limited by their tedious and error-prone programming model. This is of particular concern because this problem is a consequence of the technology’s strengths: FPGAs operate with fine grain concurrency, where the programmer can control the simultaneous behavior of every circuit on the chip. Unfortunately, this control also requires that the programmer manage fine grain constraints such as on-chip memory usage and routing congestion. The CPU programmer, on the other hand, needs only consider the potential state of the CPU at each line of code, while on-chip resources are automatically managed by the hardware at runtime.

I recently realized that modern embedded systems may soon face a similar programming problem. Battery technology continues to remain relatively stagnant, and the slowing of Moore’s Law became painfully evident after the nearly 6-year gap between 65 and 28 nm fabrication technology. At the same time, consumers have come to expect the continued advancement of embedded system capabilities, such as being able to run real-time augmented reality software on a processor that fits in a pair of eyeglasses.

Given these demands for energy efficiency and performance, many embedded processor vendors are seeking more energy-efficient approaches to microarchitecture, often involving targeting the types of parallelism that cannot be automatically extracted from software. This will require cooperation of the programmers to write parallel code. This is a lot of to ask of programmers, who will need to juggle both functionality and performance on a resource- and power-constrained platform that includes a wide range of potential sources of parallelism from multi-cores to GPU shader units.

Many universities have developed “unified” parallel programming courses that cover the spectrum of parallel programming from distributed systems to manycore processors. However, the topic is most often taught from the perspective of high-performance computing as opposed to embedded computing.

With the recent explosion of advanced embedded platforms such as the Raspberry Pi, I saw a need to develop curriculum that combines topics from computer architecture and parallel programming for performance-oriented programming of embedded systems. I also wanted to include interesting and relevant projects and case studies for the course to avoid the traditional

types of dull course projects associated with embedded systems courses (e.g., blink the light) and parallel programming courses (e.g., write and optimize a Fast Fourier Transform).

While using these ideas in my own embedded systems course, and I often find the students competing among themselves to achieve the fastest image rotation or the fastest Mandelbrot set generator. This type of collegial competition cultivates excitement for the material.

## USING THIS BOOK

This book is intended for use in a junior- or senior-level undergraduate course in a computer science or computer engineering curriculum. Although a course in embedded systems may focus on subtopics such as control theory, robotics, low power design, real-time systems, or other related topics, this book is intended as an introduction to *performance-oriented* programming for lightweight system-on-chip embedded processors.

This book should accompany an embedded design platform such as a Raspberry Pi, on which the student can evaluate the practices and methodologies described.

When using this text, students are expected to know the C programming language, have a basic knowledge of the Linux operating system, and understand basic concurrency such as task synchronization.

## INSTRUCTOR SUPPORT

Lecture slides, exercise solutions, and errata are provided at the companion website: [textbooks.elsevier.com/9780128003428](http://textbooks.elsevier.com/9780128003428)

# Acknowledgments

Several students assisted me in the development of this book.

During spring and summer 2013, undergraduate students **Benjamin Morgan**, **Jonathan Kilby**, **Shawn Weaver**, **Justin Robinson**, and **Amadeo Bellotti** evaluated the DMA controller and performance monitoring unit on the Raspberry Pi's Broadcom BCM2835 and the Xilinx Zynq 7020.

During summer 2014, undergraduate student **Daniel Clements** helped develop a uniform approach for using the Linux `perf_event` on the ARM11, ARM Cortex A9, and ARM Cortex A15. Daniel also evaluated Imagination Technology's OpenCL runtime and characterized its performance on the PowerVR 544 GPU on our ODROID XU Exynos 5 platform.

During summer 2015, undergraduate student **Friel "Scottie" Scott** helped evaluate the Mali T628 GPU on the ODROID-XU3 platform and proofread Chapter 5.

Much of my insight about memory optimizations for computer vision algorithms were an outgrowth of my graduate student **Fan Zhang**'s dissertation on auto-optimization of stencil loops on the Texas Instruments Keystone Digital Signal Processor architecture.

I would like to thank the following reviewers, who provided feedback, insight, and helpful suggestions at multiple points throughout the development of the book:

- **Miriam Leeser**, Northeastern University
- **Larry D. Pyeatt**, South Dakota School of Mines and Technology
- **Andrew N. Sloss**, University of Washington, Consulting Engineer at ARM Inc.
- **Amr Zaky**, Santa Clara University

I would like to thank Morgan Kaufmann and specifically to **Nate McFadden** for his constant encouragement and limitless patience throughout the writing. I am especially grateful for Nate's open-mindedness and flexibility with regard to the content, which continually evolved to keep current with new ARM-based embedded development platforms being released while I was developing the content. I also wish to thank **Sujatha Thirugnana Sambandam** for her detail-oriented editing and to **Mark Rogers** for designing the cover.



# Contents

Preface .....	xiii
Acknowledgments .....	xv
<b>CHAPTER 1 The Linux/ARM embedded platform.....</b>	<b>1</b>
1.1 Performance-Oriented Programming .....	3
1.2 ARM Technology .....	6
1.3 Brief History of ARM .....	7
1.4 ARM Programming .....	8
1.5 ARM Architecture Set Architecture.....	8
1.5.1 ARM general purpose registers.....	9
1.5.2 Status register .....	11
1.5.3 Memory addressing modes.....	12
1.5.4 GNU ARM assembler .....	13
1.6 Assembly Optimization #1: Sorting.....	14
1.6.1 Reference implementation.....	14
1.6.2 Assembly implementation .....	15
1.6.3 Result verification.....	18
1.6.4 Analysis of compiler-generated code.....	21
1.7 Assembly Optimization #2: Bit Manipulation.....	22
1.8 Code Optimization Objectives .....	25
1.8.1 Reducing the number of executed instructions.....	25
1.8.2 Reducing average CPI .....	25
1.9 Runtime Profiling with Performance Counters.....	28
1.9.1 ARM performance monitoring unit .....	28
1.9.2 Linux <i>Perf_Event</i> .....	29
1.9.3 Performance counter infrastructure.....	30
1.10 Measuring Memory Bandwidth.....	34
1.11 Performance Results .....	37
1.12 Performance Bounds.....	38
1.13 Basic ARM Instruction Set .....	38
1.13.1 Integer arithmetic instructions.....	39
1.13.2 Bitwise logical instructions .....	39
1.13.3 Shift instructions.....	39
1.13.4 Movement instructions .....	40
1.13.5 Load and store instructions .....	40

1.13.6 Comparison instructions .....	42
1.13.7 Branch instructions .....	42
1.13.8 Floating-point instructions.....	42
1.14 Chapter Wrap-Up .....	44
Exercises .....	45
 <b>CHAPTER 2 Multicore and data-level optimization:</b>	
<b>OpenMP and SIMD .....</b>	<b>49</b>
2.1 Optimization Techniques Covered by this Book...	50
2.2 Amdahl's Law .....	52
2.3 Test Kernel: Polynomial Evaluation .....	53
2.4 Using Multiple Cores: OpenMP.....	55
2.4.1 OpenMP directives .....	56
2.4.2 Scope.....	58
2.4.3 Other OpenMP directives .....	62
2.4.4 OpenMP synchronization .....	63
2.4.5 Debugging OpenMP code .....	66
2.4.6 The OpenMP parallel for pragma .....	68
2.4.7 OpenMP with performance counters .....	70
2.4.8 OpenMP support for the Horner kernel .....	71
2.5 Performance Bounds.....	71
2.6 Performance Analysis.....	73
2.7 Inline Assembly Language in GCC .....	74
2.8 Optimization #1: Reducing Instructions per Flop .....	76
2.9 Optimization #2: Reducing CPI .....	79
2.9.1 Software pipelining.....	81
2.9.2 Software pipelining Horner's method .....	84
2.10 Optimization #3: Multiple Flops per Instruction with Single Instruction, Multiple Data.....	92
2.10.1 ARM11 VFP short vector instructions.....	94
2.10.2 ARM Cortex NEON instructions .....	97
2.10.3 NEON intrinsics.....	100
2.11 Chapter Wrap-Up .....	101
Exercises .....	102
 <b>CHAPTER 3 Arithmetic optimization and the Linux Framebuffer .....</b>	
3.1 The Linux Framebuffer .....	106
3.2 Affine Image Transformations .....	108
3.3 Bilinear Interpolation.....	110

3.4 Floating-Point Image Transformation .....	110
3.4.1 Loading the image .....	113
3.4.2 Rendering frames .....	115
3.5 Analysis of Floating-Point Performance .....	119
3.6 Fixed-Point Arithmetic .....	120
3.6.1 Fixed point versus floating point:	
Accuracy .....	121
3.6.2 Fixed point versus floating point:	
Range .....	121
3.6.3 Fixed point versus floating point:	
Precision .....	122
3.6.4 Using fixed point .....	123
3.6.5 Efficient fixed-point addition .....	123
3.6.6 Efficient fixed-point multiplication .....	127
3.6.7 Determining radix point position .....	130
3.6.8 Range and accuracy requirements for	
image transformation .....	131
3.6.9 Converting from floating-point to	
fixed-point arithmetic .....	132
3.7 Fixed-Point Performance .....	134
3.8 Real-Time Fractal Generation .....	134
3.8.1 Pixel coloring .....	137
3.8.2 Zooming in .....	138
3.8.3 Range and accuracy requirements .....	139
3.9 Chapter Wrap-Up .....	140
Exercises .....	141

## **CHAPTER 4 Memory optimization and video processing .....**

4.1 Stencil Loops .....	148
4.2 Example Stencil: The Mean Filter .....	149
4.3 Separable Filters .....	150
4.3.1 Gaussian blur .....	151
4.3.2 The Sobel filter .....	153
4.3.3 The Harris corner detector .....	156
4.3.4 Lucas-Kanade optical flow .....	158
4.4 Memory Access Behavior of 2D Filters .....	160
4.4.1 2D data representation .....	161
4.4.2 Filtering along the row .....	162
4.4.3 Filtering along the column .....	163
4.5 Loop Tiling .....	164

4.6 Tiling and the Stencil Halo Region .....	167
4.7 Example 2D Filter Implementation.....	167
4.8 Capturing and Converting Video Frames .....	172
4.8.1 YUV and chroma subsampling .....	172
4.8.2 Exporting tiles to the frame buffer.....	174
4.9 Video4Linux Driver and API.....	176
4.10 Applying the 2D Tiled Filter.....	181
4.11 Applying the Separated 2D Tiled Filter.....	182
4.12 Top-Level Loop.....	182
4.13 Performance Results .....	183
4.14 Chapter Wrap-Up .....	184
Exercises .....	184

## CHAPTER 5 Embedded heterogeneous programming

<b>with OpenCL</b> .....	187
5.1 GPU Microarchitecture.....	189
5.2 OpenCL.....	190
5.3 OpenCL Programming Model, Idioms, and Abstractions .....	191
5.3.1 The host/device programming model .....	191
5.3.2 Error checking .....	192
5.3.3 Platform layer: Initializing the platforms.....	194
5.3.4 Platform layer: Initializing the devices.....	197
5.3.5 Platform layer: Initializing the context .....	199
5.3.6 Platform layer: Kernel control .....	201
5.3.7 Platform layer: Kernel compilation.....	202
5.3.8 Platform layer: Device memory allocation..	205
5.4 Kernel Workload Distribution.....	207
5.4.1 Device memory.....	208
5.4.2 Kernel parameters.....	210
5.4.3 Kernel vectorization .....	213
5.4.4 Parameter space for Horner kernel .....	214
5.4.5 Kernel attributes .....	216
5.4.6 Kernel dispatch.....	216
5.5 OpenCL Implementation of Horner's Method: Device Code .....	222
5.5.1 Verification .....	225
5.6 Performance Results .....	227
5.6.1 Parameter exploration.....	227

5.6.2 Number of workgroups.....	227
5.6.3 Workgroup size.....	228
5.6.4 Vector size .....	229
5.7 Chapter Wrap-Up .....	229
Exercises .....	230
Appendix A Adding PMU support to Raspbian for the Generation 1 Raspberry Pi .....	233
Appendix B NEON intrinsic reference .....	237
Appendix C OpenCL reference.....	253
Index .....	297

