Three Fingered Jack: Productively Addressing Platform Diversity

David Sheffield

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-185
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-185.html

November 26, 2013

Copyright © 2013, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Three Fingered Jack: Productively Addressing Platform Diversity

by

David Bradley Sheffield

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering - Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Kurt W. Keutzer, Chair
Professor Krste Asanović
Professor David Wessel

Fall 2013

The dissertation of David Bradley Sheffield, titled Three Fingered Jack: Productively Addressing Platform Diversity, is approved:

Chair _____ Date _____

_____ Date _____

_____ Date _____

University of California, Berkeley

Three Fingered Jack: Productively Addressing Platform Diversity

Copyright 2013
by
David Bradley Sheffield

Abstract

Three Fingered Jack: Productively Addressing Platform Diversity

by

David Bradley Sheffield

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Kurt W. Keutzer, Chair

Moore's Law has given the application designer a large palette of potential computational substrates. The application designer can potential map his or her application onto specialized task-specific accelerators for either energy or performance benefits; however, the design space for task-specific accelerators is large. At one extreme is the conventional microprocessor, easy to program but relatively low-performance and energy inefficient. At the other extreme is custom, fixed-function hardware crafted solely for a given task. Studies have reported energy-efficiency gains using fixed-function hardware from 2× to 100× over programmable solutions. If we wish to evaluate this design space we need prototypes for the elements of it; however, constructing functional prototypes for each hardware substrate is a daunting prospect. This is because each implementation target requires a radically different set of programming and design tools.

To address the challenges of mapping applications across a broad range of targets, this thesis presents Three Fingered Jack. Three Fingered Jack is a highly productive approach to generating applications that run on multicore CPUs or data-parallel processors. Three Fingered Jack also integrates a high-level hardware synthesis engine that has the ability to generate custom hardware implementations.

Three Fingered Jack applies dependence analysis and reordering transformations to a restricted set of Python loop nests to uncover parallelism. By exploiting data parallelism, Three Fingered Jack allows the programmer to use the same Python source to target all three supported platforms. It exploits this parallelism on CPUs and vector-thread processors by generating multithreaded code with short-vector instructions. The high-level hardware synthesis engine uses the parallelism found by the system to both exploit memory-level parallelism and automatically generate multiple parallel processing engines.

On a 3.4 GHz Intel i7-2600 CPU, Three Fingered Jack generated software solutions that obtained performance between 0.97-113.3× of hand-written C++ across four kernels and two applications. Over four kernels, Three Fingered Jacks high-level synthesis results are between 1.5-12.1× faster than an optimized soft-core CPU on a Zynq XC7Z020 FPGA. When evaluated in a 45nm ASIC technology, the results of Three Fingered Jacks high-level synthesis system is 3.6× more efficient than an optimized scalar processor on the key kernels

used in automatic speech recognition. On the same speech recognition kernels, the hardware results are 2.4× more energy efficient than a highly optimized data-parallel processor.

To my family – Mom, Dad, Johnny, and last but certainly not least, my fiancee Genevieve. Thanks to Mom and Dad for building the foundation of who I am today. Genevieve, the last six years would not have been possible without you. Your encouragement and support kept me going when graduation seemed like an impossibility.

Contents

List of Figures **v**

List of Tables **xi**

1 Introduction **1**
1.1 The Diversity of Potential Platforms . . . 2
1.2 The Delivery of an Application on Diverse Platforms . . . 4
1.3 Thesis Contributions . . . 5
1.4 Thesis Outline . . . 6

2 Background and Motivation **8**
2.1 Explosion of Potential Implementation Platforms . . . 8
2.2 Trends in Hardware and Software . . . 12
2.2.1 Mobile hardware . . . 13
2.2.2 Software . . . 15
2.3 Our Solution: Three Fingered Jack . . . 15
2.4 Related Work . . . 16
2.4.1 Software . . . 16
2.4.2 Hardware . . . 17
2.5 Summary . . . 19

3 Three Fingered Jack: Software Approaches **20**
3.1 Dependence Analysis and Reordering Transforms . . . 20
3.1.1 Theory of Loop Dependence . . . 21
3.1.2 Analyzing Loop Dependence . . . 22
3.1.3 Reordering Transforms . . . 24
3.2 A Restricted Subset of Python . . . 25
3.2.1 Supported Grammar . . . 26
3.2.2 Supported Data Types . . . 27
3.3 The Software Architecture of TFJ . . . 28
3.3.1 Python Front-End . . . 28
3.3.2 TFJ's Dependence and Reordering Engine . . . 29
3.3.3 Code Generation . . . 32
3.3.4 Vector-Thread Code Generation . . . 35

3.3.5 Run-time . . . 37
3.4 Summary . . . 38

4 Three Fingered Jack: Evaluation of Software Approaches 39
4.1 Overview and Setup . . . 39
4.2 Numerical Kernels . . . 39
4.2.1 Vector-Vector Addition . . . 40
4.2.2 Matrix Multiply . . . 40
4.2.3 Diagonal Sparse-Matrix Vector Multiply . . . 40
4.2.4 Back Propagation Weight Adjustment . . . 41
4.2.5 Numerical Kernel Results . . . 41
4.3 Small Applications . . . 44
4.3.1 Content-Aware Image Resizing . . . 44
4.3.2 Horn-Schunck Optical Flow . . . 47
4.4 Autotuning Using TFJ Demonstrated With Matrix Multiply . . . 52
4.5 Overheads for Runtime Code Generation . . . 56
4.6 Summary . . . 59

5 Three Fingered Jack: Hardware Approaches 61
5.1 The Architecture of TFJ's Processing Engine Clusters . . . 63
5.2 Our High-Level Hardware Synthesis Flow . . . 66
5.2.1 Datapath Scheduling . . . 67
5.2.2 Generation of Non-blocking Memory Operations . . . 71
5.2.3 Register Binding . . . 72
5.2.4 A Library of Hardware Components . . . 74
5.3 Evaluation of Numerical Kernels on a FPGA . . . 76
5.3.1 Overview of Numerical Kernels . . . 77
5.3.2 Evaluation Setup . . . 78
5.3.3 Results and Analysis . . . 80
5.4 Case Study: Tuning Hardware Matrix Multiply . . . 86
5.4.1 Results . . . 87
5.5 Summary . . . 92

6 A Case Study in Speech Recognition using Three Fingered Jack 94
6.1 Challenges for Mobile Speech Recognition . . . 94
6.2 Speech Recognition Background . . . 98
6.2.1 Profiling Our Speech Recognizer . . . 99
6.3 Accelerated Speech Recognition Kernels . . . 103
6.4 Hardware Verification . . . 105
6.5 Hardware and Software Evaluation . . . 105
6.5.1 Rocket-Hwacha Data-Parallel Processor . . . 105
6.5.2 VLSI Flow . . . 106
6.5.3 VLSI results . . . 108
6.6 Summary . . . 109

7 Conclusions and Future Work **111**
7.1 Contributions 111
7.1.1 Three Fingered Jack 112
7.1.2 A Case Study in Mobile Speech Recognition using Three Fingered Jack 113
7.2 Future Work 114
7.3 Summary 115

Bibliography **116**

List of Figures

1.1 The design space of hardware accelerators: Performance and energy-efficiency, for a given task, increases from left to right while programability decreases. Figure adapted from Fisher [54] 3

2.1 Each target hardware platform has its own development environment – manual design-space exploration is time consuming and error-prone 11
2.2 Processor clock scaling trends for 40 years of Intel CPUs. Figure adapted from Bryan Catanzaro's thesis [30]. 12
2.3 Contemporary mobile devices have performance similar to high-end desktop processors from the mid-2000s. On average, the 1.7 GHz Cortex A15 is 27% faster than the 2.4 GHz Pentium4. SPEC CPU2006 run on Samsung Chromebook XE303 and generic Intel Pentium4 desktop. Chromebook runs Ubuntu 12.04 with GCC 4.7 while our Pentium4 runs Fedora 14 with GCC 4.5. 14
2.4 Nvidia Tegra2 SoC: 49 mm^2 in 45 nm technology. Approximately 20% of the die dedicated to dual-core ARM Cortex-A9 application processor. Figure from AnandTech [9]. 14

3.1 The three types of dependence hazards 21
3.2 Iteration number example: the index variable i assumes iteration values 0,1,and 2. 22
3.3 A loop-nest used as our example for dependence testing. In this example, analysis attempts to determine if a dependence occurs between the left-hand and right-hand sides of the statement. 23
3.4 Three legal loop orderings of Matrix-Multiply 25
3.5 The abstract grammar rules for Three Fingered Jack's automatically parallelizable EDSL. In the grammar shown above, **object** is a built-in Python data type. We require all **objects** to be NumPy single-precision or 32-bit integer data types. The * operator expands a list into positional arguments. We have adapted TFJ grammar rules from the Python 2 AST listing [56] . . 26

3.6 TFJ compiler flow: TFJ compiler flow: We start with computation expressed as a Python loop-nest. After syntactic and semantic checking in Python, we convert the Python AST to an XML representation that is fed to our reordering and optimization engine. The optimization engine then generates machine executable code for high-performance execution. Detailed descriptions of the front-end, reordering engine, code generator, and run-time are provided in Sections 3.3.1, 3.1, 3.3.3 and 3.3.5. 28
3.7 Vector-vector addition written in Python for TFJ 28
3.8 Vector-vector addition in TFJ's XML intermediate representation. TFJ can be quickly ported to a new programming language as the interface between the host language and TFJ is a light-weight XML representation. The XML representation abstracts host language implementation details from the core reordering engine. 30
3.9 A high-level sketch of the parallelization algorithm used by TFJ. The core of the algorithm is based on Allen's **codegen** algorithm; the details of the algorithm are well documented in several publications [6, 5, 4]. We have modified his algorithm to find profitable parallelism on chip-multiprocessors with vector units. In particular, we attempt to reorder loops to achieve unit-stride memory accesses for vectorization and avoid overhead of thread creation and synchronization. 31
3.10 TFJ code generation for vector-vector addition (Figure 3.7) using LLVM. Lines 3 through 5 compute the upper loop bound of the strip-mined loop and check if the function was called with arrays of less than length 32. Lines 7 through 21 perform vectorized addition for vectors of length 32. Lines 26 through 37 perform scalar addition when the function is called with an array length of less than 32. Lines 39 through 55 clean-up the strip-mined vector loop by executing any remaining iterations. Note that our vector lengths (32) are longer than the hardware vector lengths of either AVX (8) or SSE/NEON (4). Using longer vectors enables a variant of software prefetching. 33
3.11 TFJ's C++ intrinsics for machines with four entry vectors, such Intel's SSE or ARM's Neon. We have intrinsics with eight entry vectors to map to Intel's AVX extension too. 34
3.12 Subset of the VTAPI classes and functions used by TFJ to generate code for vector-thread machines. 35

3.13 TFJ code generation for vector-vector addition (Figure 3.7) using the VTAPI. Line 8 configures the vector register file and line 9 configures the hardware vector length. Line 10 computes the upper loop bound for strip-mining. Single-precision vector objects for the source and destination operands are declared in lines 13 through 15, while lines 16 and 17 perform the vector loads for the source operands. The lines 19 and 23 describe the output argument and operand type to the **VT_VFETCH** macro. Lines 21 and 25 provide the same information for the input operands. The **VT_VFETCH** macro performs the vector computation in line 27. The vector store occurs at line 29. Lines 31 through 36 handle strip-mining clean-up on the scalar CPU. Line 32 issues a memory fence to synchronize the vector unit with the scalar processor, and lines 34 through 36 execute the clean-up code for the strip-mined loop. 36
3.14 An example of TFJ generate code running on real vector-thread hardware. We used the code presented in Figure 3.14a with the image shown in Figure 3.14b to generate Figure 3.14c. Results generated on the EOS14 test chip. . 37

4.1 Matrix multiply written in Python for TFJ 40
4.2 Back propagation weight adjustment kernel from the Rodinia benchmarks rewritten in Python for TFJ acceleration 41
4.3 Performance results for the small benchmark kernels presented in Section 4.2. Fully tabulated performance results for these benchmarks are available in Table 4.4. 41
4.4 Matrix multiply written in C. Statically allocated arrays and fixed loop bounds are required to get the Intel C compiler to automatically vectorize/parallelize matrix multiply. 42
4.5 2048×2048 matrix multiply performance comparison. Results generated using GCC 4.6.4 or ICC 13.0.1 use the C source presented in Figure 4.4. TFJ results generated using the Python source shown in Figure 4.1. In the plots labels on the bar chart,"IJK" signifies the IJK loop-nest ordering while "IKJ" implies the "IKJ" ordering. The compiler flags used for the C versions of the benchmark are shown on the bar chart. Both C and Python used single-precision matrices and all experiments were run on a 3.4 GHz Intel Core i7-2600. All implementations compiled with GCC also include flags to enable reordering of floating point operations. 43
4.6 Matrix multiply written in C with manual loop interchange of the J and K loops. 44
4.7 Original image and retargeted image after removal of 750 vertical seams. Original image of the Broadway Tower reproduced from Wikipedia 45
4.8 A portion of the TFJ accelerated kernels used in content-aware image resizing. 45
4.9 Run-times for the kernels used in content-aware image resizing 46
4.10 An example of optical flow 47
4.11 Run-times for the kernels used in optical flow 49
4.12 The key kernels used in solving optical flow 51

4.13 A Python function to generate variants of matrix multiply with different loop tiling dimension. Lines 1 through 18 generate tiled implementations of matrix multiply. The parameters **iblk**, **jblk**, and **kblk** of **make_mm** describe the dimensions of the cache tile. The output of this function is an implementation of loop titled matrix multiply as a string. 53
4.14 An autotuner for matrix multiply written in Python using TFJ. Line 1 allocates a list to keep track of the best performing configuration for each problem size. The problem sizes for which we are generating solutions are shown in line 3. Lines 4 through 16 allocate matrices and then evaluate the performance of the naive TFJ matrix multiply code for the current problem size. Lines 19 through 21 define the loop tiling search space, while lines 22 through 27 generate perform code generation using **make_mm** and write the resulting module to the filesystem. Line 28 imports the newly generated loop tiled matrix multiply into the Python environment. Lines 30 through 35 evaluate the performance of the generated loop tiled matrix multiply. The generated tiled matrix multiply is called on line 32 using the **eval** function. Lines 41 through 46 keep track of the best performing configuration, while Lines 47 and 48 report the best configuration and performance for each problem size. 54
4.15 Autotuned matrix multiply results on an Texas Instruments OMAP4460. We present baseline and tuned results for matrices from size 32 to 2048. 55
4.16 Autotuned matrix multiply results on an Intel Core i7-2600. We present baseline and tuned results for matrices from size 32 to 2048. 55
4.17 Run-time breakdown for the numerical kernels described in Section 4.2. Note the logarithmic time scale on the bar chart. 56
4.18 Run-time breakdown for the TFJ accelerated kernels used in content-aware image resizing. 57
4.19 Run-time breakdown for the TFJ accelerated kernels used in optical flow. . 58
4.20 Array doubler implemented with ASP 58
4.21 Array doubler implemented with TFJ 59

5.1 Our high-level hardware synthesis flow 62
5.2 The benefits of supporting multiple outstanding memory requests when applied to a 64×64 matrix multiply with a variety of main memory latencies. Both configurations use 2 processing engines. Each processing engine has its own 64 word L1 cache and share a 1024 word L2 cache. The configuration labeled "non-blocking" supports many concurrent memory requests to the L2 cache. The configuration supporting multiple inflight requests achieves greater than 4× better performance than the blocking configuration. 63
5.3 The initial architecture of our HLS processing cluster. The processing engines enclosed in orange are automatically generated by our HLS framework. . . . 64
5.4 The final architecture of our HLS processing cluster 65

5.5 Color conversion (source shown in Figure 5.12) system with 8 processing engines. The 8 processing engines are highlighted in yellow, green, orange, dark blue, burnt orange, red, light blue, and magenta. The memory subsystem is highlighted brown. Figure generated using Xilinx PlanAhead 14.1 with a Zynq ZC702 FPGA. 66
5.6 LLVM IR representation of the vector-vector addition kernel without autovectorization. This kernel has 6 basic blocks. The largest basic block has 11 instructions. 68
5.7 LLVM IR representation of the vector-vector addition kernel with autovectorization for a data-parallel operation length of 8. In contrast to Figure 5.6, the largest basic block has 67 instructions. 69
5.8 Datapath configuration files for FPGA and ASIC 70
5.9 A high-level sketch of the algorithm used by TFJ to support memory-level parallelism. This algorithm allows for memory-level parallelism when memory operations execute in the expected number of cycles. When memory operations take longer than expected, this algorithm adds additional wait states into the control finite state machine to handle stalling the processing element. 72
5.10 160×160 matrix-multiply memory-level parallelism for a data-parallel operation lengths from 1 entry to 16 entries and three different cache reload latencies. All results generated for a single custom processing element with no L1 cache. Results generated using logic simulation with ModelSim SE 10.0d. . . 73
5.11 Vector-vector addition used for HW evaluation 77
5.12 Color conversion kernel used for HW evaluation 78
5.13 Matrix multiply kernel used for HW evaluation 78
5.14 Gaussian mixture model evaluation kernel used for HW evaluation 78
5.15 Block diagram of Xilinx Zynq-7000 [145]. Our automatically generated processing engines are implemented in the programmable logic (marked with a red oval). The processing engines interface with the onboard ARM processors using the general-purpose ports marked in green. 79
5.16 ZedBoard Zynq-7000 Development Board used for FPGA evaluation 80
5.17 C++ source used as glue to interface memory-mapped accelerators in Zynq FPGA fabric with ARM cores. Line 2 polls the accelerator status register. Bit zero of the status register indicates a request. As shown in line 4, the second bit of the status register indicates if the transaction is a read or write while the memory operation address is placed in the upper 24 bits of the status register. Lines 5 through 11 perform the actual 16-byte read or write while line 12 acknowledges a complete memory transaction. Finally, lines 14 through 16 check if the accelerator has completed executing a given computation. Using this code, a 16-byte cache line reload takes approximately 100 cycles on the 100 MHz FPGA clock domain. 81
5.18 Scaling with one-cycle reloads (*larger speedup connotes better results)* 83
5.19 Scaling with ten-cycle reloads (*larger speedup connotes better results)* 84

5.20 Scaling with 100 cycle reloads (*larger speedup connotes better results)* 84
5.21 Python implementations of the two kernels used for tuning hardware matrix multiply 87
5.22 Matrix multiply using a single processing element 89

6.1 Energy consumption for 60s of ASR on a menagerie of commercial platforms. All devices achieve real-time performance on Wall Street Journal 5k corpus [115]. Energy recorded using a "watts up? PRO" power meter. 95
6.2 There are a number of ways of providing speech recognition on mobile devices. Different target markets, performance requirements, client-computing profiles, and energy concerns motivate many different solutions. For these reasons, we need the ability to explore the entire design space of general purpose CPUs, data-parallel accelerators, and custom fixed-function hardware. 96
6.3 A system on a chip: a large fraction of the SoC die area is dedicated to custom accelerators such as video encoders/decoders or image processing. The hardware speech recognition solutions presented as a possible solution for mobile ASR in this chapter are intended be a small logic block (under $5mm^2$) on a SoC. 97
6.4 An architecture for Mel-frequency cepstral coefficient generation 98
6.5 Architecture of an HMM-based speech recognizer 98
6.6 Speech recognition correctness, accuracy, and real-time factor for a variety of beam widths. Experiments run on 2404 seconds of audio encapsulating 330 utterances from the Wall Street Journal 5000 word corpus. Plots generated by sweeping beam width from 100 to 10000 in steps of 100 and then from 10000 to 30000 in steps of 500. 100
6.7 Measured cache locality in our speech recognizer for a variety of beam width sizes. To measure locality, we simulate a single-level memory hierarchy and then sweep L1 cache sizes from 64 bytes to 512 mBytes. Our experiments use a direct mapped cache with 32 byte cache lines. 101
6.8 Average memory access time as function L3 cache size and beam width sizes. We use 16 kByte 4-way set-associative L1 cache with 1 cycle access latency. Our L2 cache is 256 kBytes with 8-way associativity and has a 10 cycle access latency while our L3 cache has 32-way associativity and 30 cycle access latency. All 3 levels of the memory hierarchy have 32 byte cache lines. 102
6.9 GMM-based observation probability evaluation kernel in Python for TFJ acceleration 103
6.10 Across-word traversal kernel represented in Python for TFJ acceleration . . . 104
6.11 Speech recognition hardware verification scheme using our decoder and Synopsys VCS. Speech recognizer diagram based adapted from Chong. 106
6.12 VLSI layouts. Note: scale listed for each accelerator. 107
6.13 Energy breakdown of GMM and across-word search kernels running on programmable and fixed-function accelerators. 108

List of Tables

4.1 Image resizing performance results across five different implementations . . . 46
4.2 Optical flow performance results across five different implementations 49
4.3 Optical flow kernel call counts for the results shown in Table 4.2 49
4.4 Performance results for the 4 kernels and 2 applications. **Bold** numbers indicate best results. For the 4 kernels, larger Mflops/sec values indicate faster implementations. Application performance is reported in seconds; therefore, shorter runtimes reflect higher peformance. We have marked categories N/A if we could not find a Python library that implements a given benchmark. On our ARM platform, several benchmarks did not complete in under 24 hours when executed as Python loop-nests. 60

5.1 A comparison of the number of 32-bit registers required by the two different binding algorithms. 74
5.2 FPGA statistics for our floating-point units on the Xilinx Zynq XC702 FPGA 74
5.3 FPGA statistics for our direct-mapped, write-through L1 caches on the Xilinx Zynq XC702 FPGA 75
5.4 FPGA statistics for our direct-mapped, write-back L2 caches on the Xilinx Zynq XC702 FPGA. All L2 caches use 16 byte cache lines. 76
5.5 FPGA statistics for our soft-core RISCV CPU on the Xilinx Zynq XC702 FPGA 79
5.6 FPGA statistics for the four benchmarks of Section 5.3.1 on the Xilinx Zynq XC702 FPGA. All processing engines were generated using the datapath resource configuration file shown in Figure 5.8a. Designs that require more LUTs than provided by the FPGA are marked in red. 82
5.7 L2 cache reload statistics. Reload counts are identical for all three memory latency configurations. Configurations that do not fit in our Zynq device are marked in red. 85
5.8 Single-precision non-blocked matrix multiply FPGA resource statistics for 256 kB L2 with data-parallel operation lengths from 1 to 16. Designs that do not fit in Zynq device are marked in red. 88
5.9 Single-precision blocked matrix multiply FPGA resource statistics for 256 kB L2 with data-parallel operation lengths from 1 to 16. Due to FPGA resource limitations, the eight processing engine design uses 256 word L1 cache. All other designs use 1024 word L1 cache. 88

6.1 Speech recognizer performance (results presented in real-time factor) with caches enabled and disabled on an Intel i7 920. Results generated using 60 seconds of audio from the Wall Street Journal 5k corpus. All levels of the cache hierarchy were disabled by setting the CD flag and clearing the NW flag of CR0 and clearing the memory type range registers [75] 103
6.2 FPGA statistics for our prototype speech recognition accelerator on the Xilinx Zynq xc702 FPGA. The speech recognition solution prototyped on the FPGA platform has two GMM accelerators and two across-word accelerators. The accelerators run at 50 MHz on the Zynq FPGA. 105
6.3 Design statistics for the VLSI layouts presented in Figure 6.12 108
6.4 Energy results for WSJ clip 441c0201 (6.07 seconds) with TFJ generated solutions. The "rest of system" category includes all kernels not accelerated with TFJ. 109
6.5 Expected hours of ASR with hardware/software solutions assuming a 20 kJ battery 109

Chapter 1

Introduction

VLSI scaling enabled massively increased integration capabilities on single chip microprocessors for nearly 30 years. As first observed by Moore [106], the number of transistors on a monolithic silicon die doubles approximately every 18 months. For the first 20 years following Moore's observation, increased integration capabilities were used to improve single-threaded performance on both desktop and workstation computers. During this period, the supply voltage and channel length of transistors used in CMOS VLSI chips scaled according to Denard's classic predictions [48]. In the classic (long-channel devices) CMOS scaling regime, each new process technology node reduced the minimum lithography dimensions by approximately 30% and also allowed for supply voltage scaling. Not only did each new technology node allow for more integration, but new chips consumed the same amount of power as did the previous generation due to supply voltage scaling. In this period, transistor budgets were dedicated to larger cache memories and other micro-architectural features designed to increase performance. Increased single-threaded performance enabled existing software to scale its performance with enhancements in VLSI technology. Stated simply, no changes in programming methodology were required to extract performance from new microprocessors. As an example of the significant performance improvements delivered during this era, consider two instruction-set compatible processors from Intel: the 486 and the Pentium4.

Released in 1989, the Intel 486 ran at 20 MHz and retired a maximum of one instruction per cycle. In contrast, when released in 2000, the Pentium4 ran at 1.5 GHz and retired up to three instructions per cycle [68]. In slightly more than a decade, clock frequencies increased 75× and instruction issue width increased by another factor of 3×. While clock scaling and instruction issue are correlated only with measured single-threaded performance, Intel was able to potentially unlock a 225× increase in performance during this period. Unfortunately, changes in semiconductor physics associated with deep-submicron processes have limited the applicability of aggressive clock scaling in the period after the early 2000s. In contrast to the era of classical CMOS scaling, supply voltage scaling was no longer possible due to the exponential dependence of leakage current on threshold voltage. Furthermore, the delay relationship between wires and logic changed [69, 132]. Wires had historically been fast while logic was slow. This enabled global signals to span an entire chip in a

single clock cycle. In the deep-submicron area, logic became proportionally faster than wires. This prevented global single-cycle on chip communication. The clock rates of Intel's flagship desktop microprocessors reflect the issues associated with deep-submicron processes. Specifically, the clock rate of Intel's flagship processors have been stagnated since 2004 at approximately 3.8 GHz.

While power and wiring issues have complicated computer design in the deep submicron era, CMOS scaling still provides ever increasing transistor budgets every 18 months. To increase performance without increasing clock frequencies, major microprocessor companies have turned to parallel processors by adding multiple processors cores to a monolithic silicon die. By harnessing multiple processors, software can be integer factors faster than the single-threaded equivalent; however, a move to parallel processing implies a radical shift in software engineering methodology. To effectively use parallel processors, legacy software must be rewritten to take advantage of additional processors.

While multicore processors with short-vector extensions have the potential for significant performance improvements in many common applications, writing parallel software has historically been a challenge. It has been plagued with both fundamental and engineering challenges. Traditional efficient algorithms often display little parallelism and require new approaches to find algorithms that can effectively exploit parallel resources. Compounding the parallel programming implementation challenge is the diverse nature of parallel programming models. These models are diverse and often reflect underlying differences in parallel hardware platforms. For example, a cluster of workstations often requires a message-passing programming model to obtain high-performance. This is because processors communicate by sending messages over a network. In contrast, the shared-memory programming model assumes processors communicate directly through a global shared pool of memory. These differences in hardware and programming models make portable parallel programming challenging. Furthermore, exploiting data-parallelism with short-vector units often requires reimplementing data-structures in order to satisfy strict memory alignment rules. When compounded with other issues associated with parallel computing such as scalability, load imbalance, and concurrency bugs, parallel computing has struggled to succeed outside the academic research and high-performance computing communities.

Even with the challenges of parallel programming, we must embrace it because multicore processors now span the gamut from high-end servers to smartphones and tablets. Writing parallel software is the only way to unlock high-performance on these devices. However, even with the performance benefits of multicore processors for some applications, software alone may be unable to deliver a high-performance solution that is both high-performance and energy-efficient. In these situations, algorithms must be moved from the software-domain into a more specialized task-specific accelerator in order to achieve the desired goals.

1.1 The Diversity of Potential Platforms

As shown in Figure 1.1, the design space for specialized task-specific accelerators is large. At one extreme is the conventional microprocessor, easy to program but relatively low-performance and energy inefficient. At the other extreme is custom, fixed-function hardware

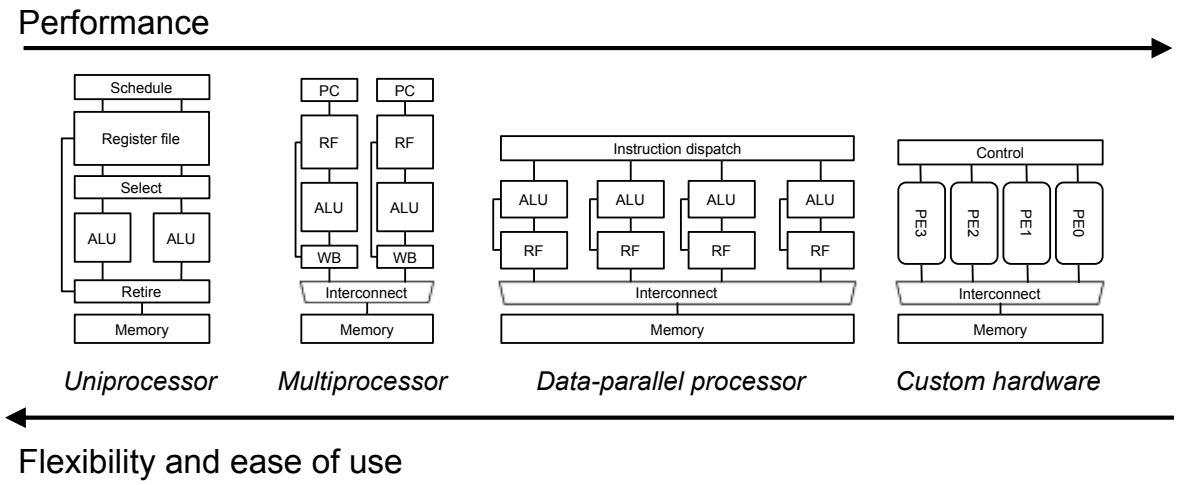


Figure 1.1: The design space of hardware accelerators: Performance and energy-efficiency, for a given task, increases from left to right while programability decreases. Figure adapted from Fisher [54]

crafted solely for a given task. To illustrate the potential efficiency gains made possible by fixed function hardware, Brodersen [26] reported 4 orders of magnitude in energy-efficiency between programmable microprocessors and dedicated hardware solutions. Other studies have reported energy-efficiency gains using fixed-function hardware from 2× to 100× over programmable solutions. The energy-efficiency improvement depends on the characteristics of the underlying algorithm being accelerated with fixed-function hardware (the number of DRAM memory accesses and memory access pattern) and the hardware design-style (full-custom versus high-level hardware synthesis).

If we wish to evaluate this design space we need prototypes for the elements of it; however, constructing functional prototypes for each hardware substrate is a daunting prospect. This is because each implementation target requires a radically different set of programming and design tools. For example, implementing an application on a data parallel processor requires a complete rewrite in languages such as OpenCL or CUDA. Likewise, designing custom hardware requires describing the micro-architecture of an accelerator in a high-level hardware description language such as SystemC. After prototyping potential accelerator micro-architectures in SystemC, the design is manually converted into a lower-level hardware description language such as Verilog or VHDL. A two-tiered approach to hardware system design is used to productively explore many potential designs in SystemC. As SystemC is effectively a C++ class library, designers can use C++ abstractions to model micro-architectural components that quickly evaluate architectures. However, the flexibility afforded by SystemC comes at a cost. There are currently no efficient ways of converting a SystemC design into a efficient hardware implementation. Instead, a designer must manually convert software abstractions used in a SystemC model into implementable hardware primitives. As the translation process from simulation description to hardware description

is done manually, it provides ample opportunity for the introduction of hardware bugs and errors.

Hardware design also introduces new challenges due to the large number of complex tools required. To design an application-specific integrated circuit (ASIC) using a standard-cell based approach, several tools are needed. Modern ASIC design uses logic synthesis to convert a behavioral description of an integrated circuit into a netlist of gates and memories. The gates and memories are then mapped onto a planar silicon die using a place-and-route tool. Finally, a logic simulator is required to verify that the design functions properly at each level of implementation. Not only is the full tool suite complex, the design tools and chip fabrication process are also very expensive, making hardware design accessible only to universities and corporations with significant financial resources.

The challenges of selecting a hardware platform for delivering an application are further complicated by differences of price-point and market demands. For example, a low volume product would be unlikely to afford the large engineering costs associated with designing custom hardware, but a high-volume device would be able to support the development of an accelerator for a popular application. However, even if the market size can support the costs of custom hardware design, the best way to deliver a given application capability is not always clear. For example, an application undergoing rapid algorithmic development might not be mature enough to support a custom hardware implementation. Algorithmic improvements to the application may dominate any performance improvements gained by a custom hardware implementation.

1.2 The Delivery of an Application on Diverse Platforms

To make these issues concrete, consider an application that provides automatic speech recognition (ASR) services on a mobile client. As speech is a natural and efficient way for humans to interact with a mobile device, it has emerged as an exciting application on many mobile devices. As of 2013, the dominant way of providing ASR services is to send raw audio to a cloud-based service for analysis. This approach is used in mobile systems such as Apple's Siri. However, this is not the only way of providing ASR on a mobile device. Other ways of providing mobile speech recognition include:

1. Performing speech recognition entirely in software on the general-purpose application processor
2. Performing speech recognition on an specialized programmable platform, such as an embedded graphics processing unit
3. Performing speech recognition using a customized fixed-function processing engine
4. Performing parts of speech recognition on the general-purpose processor, graphics processor, and custom hardware

Given differences in target market, time-to-market concerns, mobile client-configuration, client-computing profile, and power or energy concerns, there is no single winner for all mobile system configurations. Moreover, these concerns and constraints are rapidly changing: the right solution for this product generation may be suboptimal for the next one. As a result, *system designers require the capability to explore this entire design space of general-purpose microprocessor-based solutions, specialized programmable solutions, and fixed-function hardware accelerators.* Constructive exploration by manually crafting a point-solution for each solution type however, is too time consuming and expensive.

Given the challenges of parallel programming and hardware design, even building a single design point can be challenging. As a consequence, researchers are unable to fully explore the solution design space. The complexities of the target market, time-to market concerns, and power or energy concerns only compound these difficulties. To address the challenges of mapping applications across a broad range of targets, this thesis presents *Three Fingered Jack.* In this disseration, we consider mapping appliactions to general-purpose programmable processors, specialized programmable processors, and fixed-function hardware. Three Fingered Jack builds upon the ideas of Selective, Embedded, Just-in-Time Specialization (SEJITS) [32, 78, 31, 81] and extends those ideas to target a diverse range of potential platforms.

Three Fingered Jack (TFJ) targets this range with a *single input description.* It does so by supporting a restricted subset of loops in the Python programming language. Reordering transformations [6] are used to automatically extract data-parallelism. Once data-parallelism has been extracted, TFJ is able to exploit the parallelism on programmable platforms by mapping execution to multiple processors and vector units. TFJ integrates into a standard Python 2.7 installation, allowing the use of a wide variety of existing Python libraries, such as OpenCV or MatPlotLib. TFJ also includes a high-level hardware synthesis back-end that is able to map parallelism found with reordering transforms into efficient hardware solutions that use multiple customized processing engines (PEs).

The current TFJ implementation targets Intel x86 desktops, ARM mobile platforms, vector-thread processors [86], and automatic hardware generation. We generate software solutions that are 35-100% as efficient as hand-tuned C++ libraries. Our automatically generated hardware solutions are up to 3.6× more energy-efficient than a highly-optimized microprocessor and 2.4× more efficient than an efficient vector-thread processor.

1.3 Thesis Contributions

This work makes the following contributions:

1. Proposal a tool flow that enables a programmer to describe key applications kernels from a high-level description and then generate highly efficient software and hardware solutions from that description. To that end, this thesis research contributes two key software artifacts:

   (a) Design and implementation of an automatically parallelizing and vectorizing programming toolchain for a subset of the Python programming language. Our ap-

proach enables efficient parallel execution on several parallel architectures. Supported architectures include ARM system-on-a-chips used in mobile phones, Intel x86 processors used in desktops and laptops, and specialized data parallel processors built at the University of California Berkeley and the Massachusetts Institute of Technology. Our tool fully supports both just-in-time compilation at runtime and offline static compilation for computer systems that cannot support a full Python installation.

(b) Design and implementation of tools to generate custom hardware from the *same* parallelizable subset of Python used for software solutions. Our hardware generation system uses the dependence analysis performed by our software fronted to generate hardware solutions with multiple parallel processing engines. In addition, our approach to automatic hardware generations includes robust support for memory operations. Our flow generates processing engines that can tolerate multiple outstanding memory requests to memory structures with non-deterministic memory latencies. We evaluate our hardware solutions on both FPGAs and ASICs. Our ASIC solutions are prototyped through synthesis of virtual ASIC prototypes using a modern 45nm cell library.

2. Demonstratation of the effectiveness of our approach to hardware and software codesign by an in-depth case study in large vocabulary continuous speech recognition. We are able to *productively* explore different implementations across all three target platforms supported by TFJ. By using TFJ, we show our approach can automatically generate software solutions that are performance competitive with manually coded implementations with significantly less source code. We also show our automatically generated hardware is significantly more efficient than software-only solutions running on both conventional and vector microprocessors. Specifically, our results show an energy savings of 3.6× over that of a conventional mobile processor and 2.4× over that of a highly-optimized vector processor.

1.4 Thesis Outline

The structure of this thesis follows this outline:

- Chapter 2 provides the requisite background required to understand this thesis and a discussion of related work.
- Chapter 3 describes the subset of the Python language TFJ supports. It then describes the design and implementation of the compilation techniques used in TFJ and provides an evaluation of TFJ on several small benchmark applications.
- Chapter 5 describes the approaches used by TFJ to automatically generate hardware from a high-level Python description. It also evaluates TFJ's hardware backend on a small number of benchmark applications.

- Chapter 6 uses automatic speech recognition as an in-depth case study of TFJ's hardware and software solutions with an emerging application.
- Chapter 7 concludes by providing a summary and discussion of the contributions of this thesis.

Chapter 2

Background and Motivation

This chapter explores the background necessary to understand the motivation behind solutions presented in this dissertation. We begin in Section 2.1 by describing the trend towards an ever increasing number of potential implementation platforms and the challenges of manual hardware and software design approaches. We examine the reasons for the explosion of implementation platforms by summarizing current hardware and software trends in Section 2.2. Our solution to address the explosion of potential implementation platforms, *Three Fingered Jack*, is presented in Section 2.3, while related work is addressed in Section 2.4. The rest of this thesis explores the design, implementation, and evaluation of Three Fingered Jack.

2.1 Explosion of Potential Implementation Platforms

Within the last decade, the trajectory of mainstream computer architecture has radically departed from the traditional uniprocessor workstation. Instead of exclusively deploying applications on a uniprocessor desktop or workstation, compelling applications can now be delivered on devices with a variety of different form factors and computational capabilities.

Many parallel computer architectures and accelerators have been proposed to solve performance scalability and energy efficiency issues associated with traditional approaches to designing high-performance uniprocessor systems. The wide variety of parallel processing solutions available on both the desktop and mobile client reflect a broad spectrum of technical and economic constraints facing modern computer systems. On the technical side, possible constraints include peak performance, software compatability with previous product generations, and the performance scalability of existing parallel software. Economic constraints are often closely coupled with technical challenges. For example, a more expensive parallel processor can afford more execution resources and therefore achieve higher performance.

The challenge becomes even more difficult as some applications naturally fit a specific form factor. As an example of an application best mapped to a specific form factor, consider speech recognition on a smartphone. Smartphones have limited area for a keyboard, and as a result, users are required to interact with the mobile device using either a very small keyboard or through a touch screen based interface. Both approaches are far inferior to a

traditional keyboard and mouse found on a desktop or laptop computer. Speech recognition has emerged as a natural way to interact with a mobile device without a full-sized keyboard. Moving speech recognition to a mobile device presents several new challenges. First, as speech recognition is a computationally demanding application [2], it has been historically limited to the workstation.

Moore's Law comes to our aid here as contemporary mobile devices often have computational capabilities to similar to those of a ten-year old desktop. As shown in Figure 2.3, a modern processor tablet processor is faster than a high performance desktop from the mid-2000s on industry standard benchmarks. The dramatic increase in computation power available on mobile devices has unlocked the ability to perform software-only speech recognition solution is possible on a modern device. Battery life on a mobile device, however, would suffer from continuous use of a software-only speech recognizer. Therefore, specialized hardware may be required to improve the energy efficiency of the the speech recognition computation and increase battery life. To best support a compelling application, the design space of potential implementation platforms continues to grow. As a result of the combination of technical, form factor, and economic constraints, system solutions now span a full gamut of potential system platforms.

While the explosion of potential platforms provides effective solutions for the high performance implementation of compelling applications, it does little to help improve programmer productivity. Modern implementation platforms differ wildly in programming models and middleware, resulting in low developer productivity and challenging program portability. As shown in Figure 2.1, manually implementing high performance applications first requires the programmer to uncover parallelism that can be executed on his or her target platform. After completing this arduous task, the programmer must massage his or her implementation to fit the data and thread-level parallelism programming constructs on the chosen platform.

Source code portability between data-parallel processors is difficult due to differences in hardware intrinsics and often requires complete rewrites for a new platform. Requiring the programmer to manually find and exploit parallelism severely limits developer productivity and application portability when developing high-performance software. If performance or energy concerns require a custom hardware implementation, the development challenge becomes even larger. Attempting a hardware implementation likely requires several iterations of prototyping. A high-level prototype usually starts with a behavioral system simulator written in a modeling language such as SystemC [96] and uses a software implementation of the application or kernel as a functional reference. After verifying correctness of the high-level prototype, a manual reimplementation of the design is required to map it into a hardware description language such as Verilog. The complexity of the hardware design process is reflected in ever increasing ASIC development costs [64].

As the initial development of software on a new platform routinely takes weeks to months, alternative system platforms and algorithmic approaches are rarely explored. Constructing parallel software in a low-level language such as C++ with threading libraries and data-parallel primitives is far too unproductive and error-prone. Case studies show sequential C++ programming is 2 to 5× less productive than languages such as Python [118, 71].

Moving between parallel platforms, for example from an ARM CPU to x86 CPU, often requires extensive re-engineering due to differences in the data-parallel primitives (Neon vs SSE). Finally, parallel programs are rarely performance portable due to differences in data-parallel hardware, memory hierarchy, and core count. Hardware design presents even larger challenges to developer productivity. In summary, parallel computing needs tools that address the portability, performance, and programming challenges of modern systems.

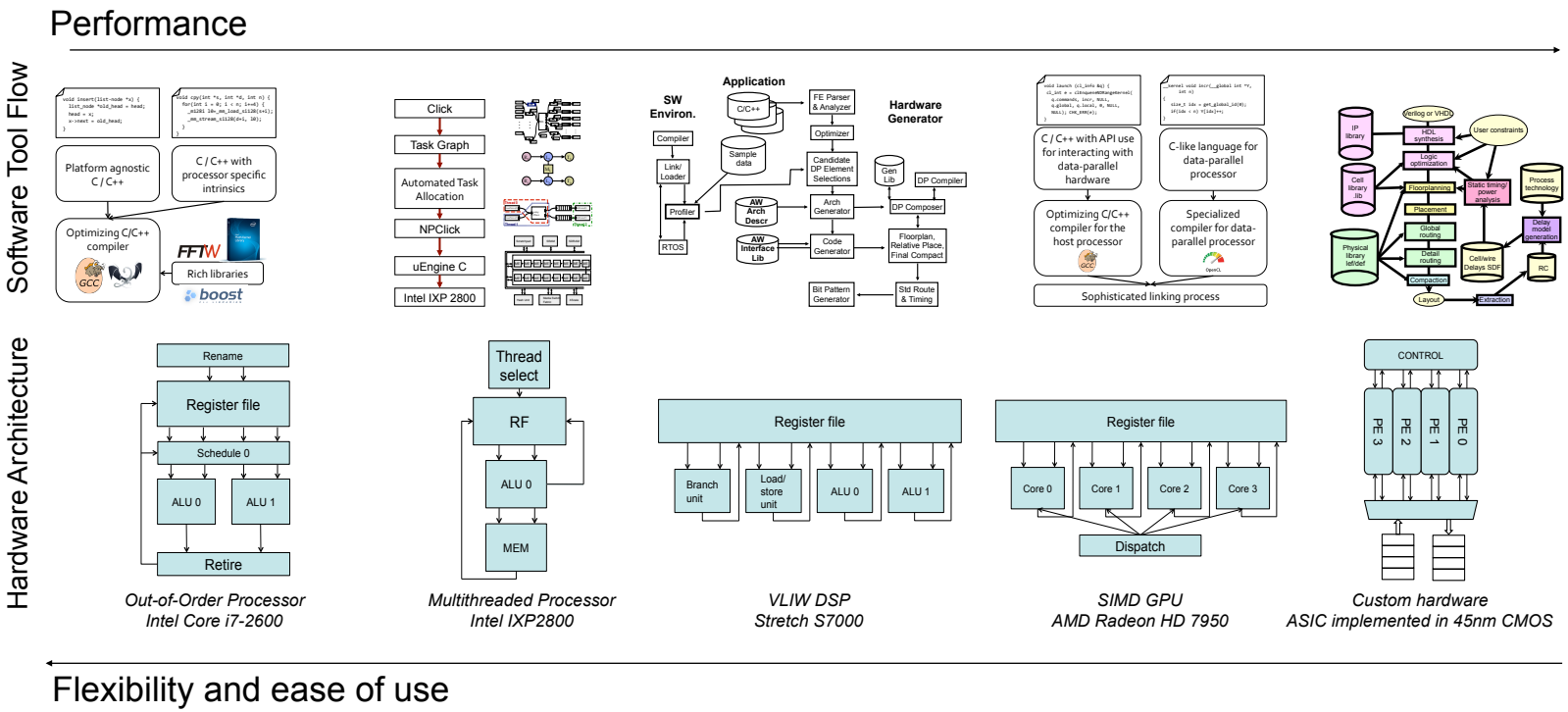


Figure 2.1: Each target hardware platform has its own development environment – manual design-space exploration is time consuming and error-prone

2.2 Trends in Hardware and Software

The technical reasons behind the explosion of potential implementation platforms can be observed by taking an in depth look into the trends driving both hardware and software.

$$\frac{Time}{Program} = \frac{Instructions}{Program} * \frac{Cycles}{Instruction} * \frac{Time}{Cycle} \tag{2.1}$$

As shown in Figure 2.2, changes in semiconductor physics associated with deep-submicron processes have limited the applicability of aggressive clock scaling in the period after the early 2000s. However, while semiconductor device physics has limited clock scaling, Moore's Law still enables transistor doubling every two years. The new challenge becomes how to increase computer performance without the ability to scale the processor clock frequency.

The "Iron Law" of computer performance [52] (Equation 2.1) states: to increase computer performance without clock cycle improvement ($\frac{Time}{Cycle}$), either the number of executed instructions per program or the number of cycles per instruction must decrease. The computer industry chose to increase performance by increasing the number of instructions executed per cycle. However, in contrast to previous approaches of increasing instruction-level parallelism, performance was gained by the addition of multiple processors on a monolithic silicon die.

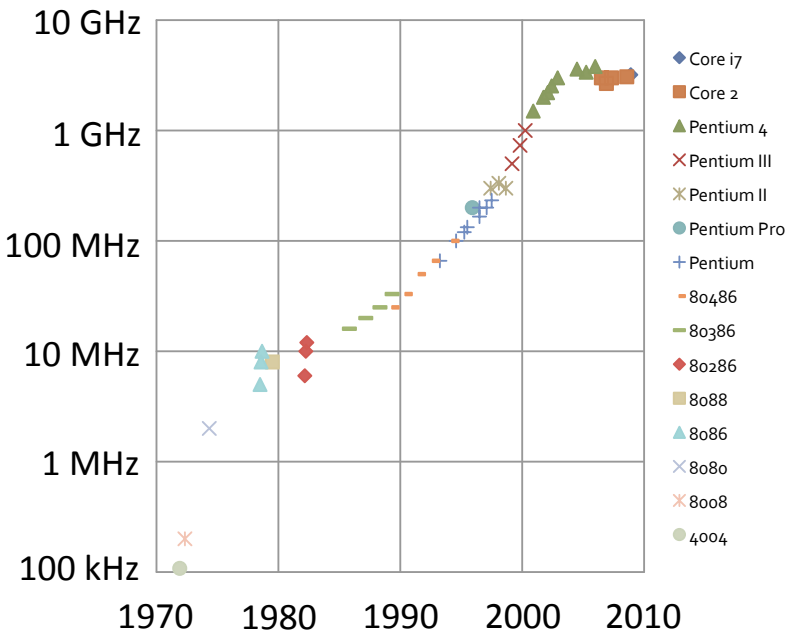


Figure 2.2: Processor clock scaling trends for 40 years of Intel CPUs. Figure adapted from Bryan Catanzaro's thesis [30].

Adding multiple processors to a single monolithic die presents several challenges. First, software must change to exploit parallel processors [13]. Software changes are also required

for other forms of parallelism added to modern processors, such as the AVX extensions added to modern Intel processors. In contrast, both frequency scaling and hardware approaches to increasing instruction-level parallelism are transparent to the software developer. Developing high-performance software on parallel machines now requires the application developer to be intimately familiar with the nuances of his or her parallel platform to extract performance.

The challenges of parallel software development are further compounded by the appearance of the "memory wall" [144]. The performance of logic and DRAM semiconductor processes improve at different rates. This has resulted in a large gap in performance between logic processes and DRAM processes. On modern microprocessors, load and store operations to DRAM can take upwards of 200 clock cycles. In constrast, even traditionally expensive operations, such as a double-precision multiply-accumulate require only a handful of cycles to execute.

Architectural techniques, such as deep cache hierarchies, can improve memory subsystem performance in programs with temporal locality; however, the programmer is often required to reformulate his or her program to take full advantage of the on-chip caches. Adding multiple processors on a monolithic silicon die only increases the challenges of the memory wall due to the increased DRAM memory bandwidth demands of multiple processors. The programmer must now find parallelism in his or her program to take advantage of multiple processors but be aware of locality in his or her program to realize an application speed-up.

2.2.1 Mobile hardware

While limited clock scaling has forced high-end computers to go parallel for increased performance, single-threaded performance has continued to improve on mobile and portable devices. As shown in figure 2.3, the tablet-oriented ARM Cortex A15 has 27% better single-threaded performance on the industry standard SPEC CPU2006 benchmarks than does a high-end desktop from the mid-2000s. Given that mobile devices now have performance characteristics similar to desktops from the mid-2000s, many applications traditionally limited to the desktop or workstation can be implemented in software on mobile systems. For example, large vocabulary speech recognition has traditionally been performed on a workstation; however, it is now possible to perform speech recognition entirely in software on a mobile device.

While CPUs in mobile devices have become ever more capable, mobile system-on-a-chips (SoC) include a multitude of specialized accelerators due to both performance and energy-efficiency concerns. As shown in Figure 2.4, approximately 20% of the die area is dedicated to user programmable ARM Cortex-A9 processors. A large fraction of the die area is dedicated to either fixed-function logic or specialized processors for image, video, audio, and graphics processing. The specialized processors often have limited programmability and are not directly exposed to the end user. Instead, most of the specialized capabilities of a mobile SoC are exposed through library calls. For example, to use the video decoding subsystem of a SoC, the SoC vendor provides a specialized video decoding library for its custom hardware. An ongoing challenge in mobile SoC design is the interplay between increased programmability of the specialized accelerators and processors and energy-efficiency.

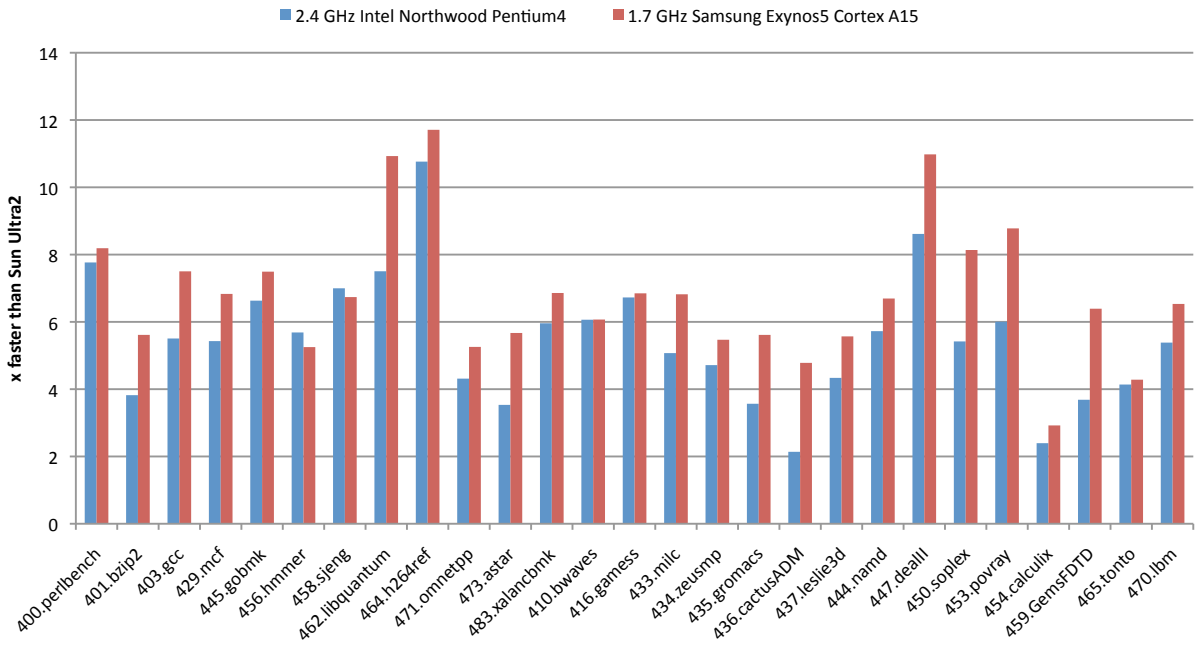


Figure 2.3: Contemporary mobile devices have performance similar to high-end desktop processors from the mid-2000s. On average, the 1.7 GHz Cortex A15 is 27% faster than the 2.4 GHz Pentium4. SPEC CPU2006 run on Samsung Chromebook XE303 and generic Intel Pentium4 desktop. Chromebook runs Ubuntu 12.04 with GCC 4.7 while our Pentium4 runs Fedora 14 with GCC 4.5.

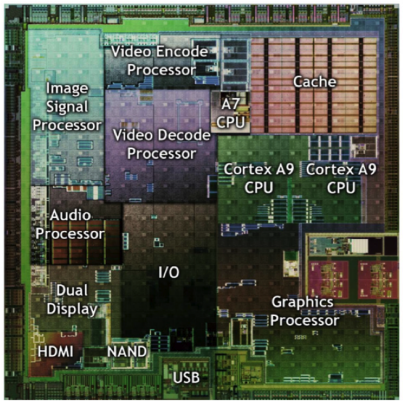


Figure 2.4: Nvidia Tegra2 SoC: 49 mm^2 in 45 nm technology. Approximately 20% of the die dedicated to dual-core ARM Cortex-A9 application processor. Figure from AnandTech [9].

Reusing specialized accelerators for tasks other than the ones for which they were designed also remains a challenge. For example, reusing the image processor on an SoC for a closely related task, such as computer vision, is just beginning to be addressed in commercial products [113]. As part of this thesis, we construct tools that can efficiently map application code to vector-thread processors that are similar in many ways to the more specialized programmable accelerators found on mobile SoCs. We also design and implement a high-level hardware synthesis flow in this thesis that could be used to construct fixed-function accelerators similar to those found on a mobile SoC.

2.2.2 Software

Software development has also changed; however, these changes have not been brought about by performance concerns. Instead, many changes to software development methodologies have occurred in order to support the rapid development of web-based services. Languages such as Python are favored in this domain due to support of high-level programming constructs, meta-programming capabilities, and rich libraries. While languages such as Python are not designed to support high-performance parallel software development, many of the features that make them attractive to web-developers also make them interesting substrates for productive, high-performance parallel software development.

In particular, languages such as Python allow the programmer to develop domain-specific embedded language extentions to simplify common programming tasks. Domain-specific embedded languages keep the syntax of the host language but allow the programmer to refine the programmaing language semantics for a domain-specific subset of the language. In Python, these capabilities are exposed in several different ways. Python includes a "decorator" syntax to mark functions as components of a domain specific language. The decorator intercepts execution and provides the developer with access to the abstract syntax tree and the ability to introspect variables. Web developers use these capabilities for database queries or generation of dynamic HTML for webpages. These features can also be used to generate high-performance domain specific parallel programming environments [30, 82]. This thesis builds a high-performance parallel programming environment using advanced automatic parallelizing algorithms within a domain-specific language embedded in Python.

2.3 Our Solution: Three Fingered Jack

In this thesis, we address the issues of portability, performance, and productivity using Three Fingered Jack (TFJ), an auto-parallelizing and vectorizing embedded domain-specific language (EDSL), for Python loop-nests. In our system, the programmer selects dense loop-nests in Python using the "decorator" syntax that redirects the Python run-time to our compiler. Because our compiler is restricted to loop-nests, we can apply aggressive parallelizing compiler algorithms to them in order to automatically generate high-performance parallel software and hardware implementations.

With our approach, *portability* is guaranteed, as all code is valid Python and can always be executed in the interpreter. If a loop-nest cannot be compiled for parallel execution (for

example, a branch in the inner-loop), we can still compile a large subset of Python for scalar execution on the host CPU. Code that cannot be compiled for efficient execution remains valid Python and will be executed in the Python interpreter. The results in Chapter 4 exemplify the cross-platform *performance* benefits of our system across several kernels and two complete applications. Finally, programmer *productivity* is enhanced by our system as the user is allowed to write his or her application in Python and selectively accelerate specific computations or generate custom hardware. This approach enables one source representation to target both CPUs and generate custom hardware.

Our work is inspired by the Selective Embedded Just-in-time Specialization (SEJITS) methodology that uses EDSLs to help mainstream programmers target Nvidia GPUs and multiprocessor CPUs [32]. We extend the SEJITS ideas with algorithms from parallelizing compilers to target loop-nests. While the parallelization algorithms used in this thesis are built on historical parallelizing compiler algorithms [5], as originally presented they were not designed for just-in-time use, cache-based chip multiprocessors, or hardware generation through high-level hardware synthesis techniques. The challenges of adapting and extending these algorithms for productive, portable, parallel software development is detailed in Chapter 3, while hardware synthesis is addressed in Chapter 5. We use Three Fingered Jack to explore software and hardware solutions for mobile speech recognition in Chapter 6.

2.4 Related Work

As the work in this dissertation spans both software and hardware design, we separately address the two categories in Section 2.4.1 and 2.4.2, respectively.

2.4.1 Software

There are several other projects working to accelerate productivity languages for non-expert programmers. The work presented in this thesis, Three Fingered Jack, builds upon the ideas of Selective, Embedded, Just-in-Time Specialization (SEJITS) [32, 78, 31, 81]. The SEJITS approach promotes building tools that optimize a domain specific subset of a host language. We use Python as the host language as it has a rich set of libraries and interfaces that make optimizing a subset of the host language possible. Other projects attempting to provide high-performance abstractions for efficiency programmers have also chosen Python for the same reasons [123, 31, 81] .

Other projects that enable productivity programmers to write high performance parallel applications in Python have used data-parallel primitives as the basis for extracting parallelism. For example, both Copperhead [31] and Parakeet [123] use data-parallel primitives. In contrast, we use reordering algorithms to extract parallelism on multicore processors with vector-units. We assert that loops are easier to understand for the majority of programmers than functional programming primitives such as map and reduce. The removal of the reduce primitive from the core syntax of Python3 supports our assertion that for-loops are more readable for most programmers [136].

Garg [57] also chose loops as his basis for extracting parallelism from Python programs in order to map execution to a graphics processor. He used dependence analysis to extract parallelism that unlocks performance on his hardware target. More interestingly, he computed the set of operands needed to be transfered between the disjoint host memory and graphics processor using the polyhedra dictated by the dependence analysis. In contrast, we use reordering transforms as a basis for addressing the *diversity challenge* presented by the explosion of potentional implementation platforms. Specifically, using a single Python source with parallelism extracted by reordering transforms, we map to multicore processors and data-parallel accelerators, then generate custom hardware.

There have been several attempts to compile subsets of Python into sequential C [1] or LLVM [8]; however, these approaches are not sufficient as they do not automatically extract parallelism. NumPy and SciPy provide data structures and routines for common mathematical operations such as vector and matrix operations. Expressiveness is limited to the operators supported by Numpy or SciPy. Addressing the parallel programming challenge with libraries is efficient; however, it only works when high-performance libraries exist. In contrast, our approach enables a developer to obtain within 2× hand-tuned parallel software within a matter of minutes using Python.

2.4.2 Hardware

The hardware generation work presented in this thesis addresses two different research areas: high-level hardware synthesis and hardware/software codesign. In this Section, we first present work on hardware/software codesign and then address related work in high-level hardware synthesis.

Hardware/Software Codesign Tools

There has been much research into methodologies and algorithms for hardware/software codesign. At the highest level, the hardware/software codesign problem addresses system design constraints by jointly optimizing both hardware and software concurrently [46]. More concretely, hardware/software codesign usually involves implementing applications described at high-level into software and hardware components. This process can be performed either manually using a specialized design methodology [64] or with automated tools [46]. Sophisticated design and verification flows also become possible using hardware/-software codesign schemes. For example, the correctness of hardware implementations of signal processing algorithms have been verified with a known-good software implementation in MATLAB [91]. The closely related hardware/software cosynthesis problem [65] generates specialized hardware and any software required to orchestrate data movement between a general-purpose processor and specialized accelerator using an automated toolflow. While there has been much research in this area, tools and methodolgies for the hardware/software codesign of embedded digital signal processing applications have proved most successful [135, 77].

In the broadest sense, Three Fingered Jack is a hardware/software codesign tool as it allows a designer to implement applications described as Python loop nests into both

hardware and software implementations. In contrast to previous work in hardware/software codesign, the work presented in this disseration is less focused on embedded applications running on resource constrained processors. While Three Fingered Jack does have the ability to automatically generate hardware, we support a dynamically typed modern programming language, Python. As described in Chapter 3, Three Fingered Jack's software optimization and machine code generation occur at run-time. Only hardware generation occurs offline. This is in constrast to the strictly static approaches traditionally used in hardware/software codesign.

Generation of Hardware from a High-Level Description

The theory of dependence and loop optimizations originated in the field of optimizing FORTRAN compilers for high-performance computing in the 70s and 80s. Since then, several works have attempted to integrate these ideas into traditional high-level synthesis systems. Weinhardt [141] used dependence analysis on FPGA designs to enhance performance by executing independent iterations of a loop-nest through a heavily pipelined circuit. Dependence analysis enabled temporal multiplexing of a single data path to increase throughput. Work on the DEFACTO system [50] used dependence analysis in a similar fashion. In contrast, we use dependence analysis to generate parallel processing elements.

Within the reconfigurable computing research community, there have been many attempts to make FPGA design more productive using high-level programming languages [29] instead of using a traditional register transfer language. These approaches are all broadly categorized as high-level hardware synthesis. Approaches using imperative programming languages include C, C++ and C# [130]. More recently, the xPilot project from UCLA [40] was commericalized as AutoESL and acquired by Xilinx [39]. Modern approaches to building hardware using functional programming techniques include two projects based on Haskell: Lava [24] and Bluespec [112].

The wide variety of high-level synthesis approaches span a wide gamut of source languages and programming paradigms; however, we have been unable to find an existing high-level synthesis system designed to enable design space exploration beyond two platforms. In general, the high-level synthesis systems previously mentioned use a subset of the host language to generate reasonably efficient hardware. If the source language used for hardware generation is executed on the host CPU, high performance results are not guaranteed as the programming paradigm required for efficient hardware generation is not necessarily the same as that required for use of high performance software on a conventional CPU. More importantly these approaches do not readily enable mapping applications to other hardware platforms such as multiprocessor systems. In contrast, the work in this disseration uses a single source representation that can generate efficient custom hardware or map code to a wide variety of programmable accelerators.

Recent work on *Irregular Code Energy Reducers* [12] provides a system for compiling existing irregular C code to specialized processing units. This tool uses a combination of compiler frameworks including LLVM. There are three main differences among our work and the earlier work. First, we target dense loop-nests and optimize for performance by generating parallel processing units, while the ICERs exclusively focus on energy and irregular

serial code. Second, our high-level starting point is Python and our focus is programmer productivity. Finally, Irregular Code Energy Reducers are not designed for high-performance. Energy savings come from lower power consumption instead of reduced program execution time.

FCUDA [116] is a high-level hardware synthesis flow that maps the NVIDIA CUDA programming language to FPGAs. CUDA is an explicitly parallel language and requires the programmer to find parallelism. In contrast, we extract parallelism from Python, a sequential language. While FCUDA allows the designer to explore trade-offs between FPGA and Nvidia GPU implementations, it does little to help the developer explore other potential targets, such as multicore CPUs. OpenCL for FPGAs [129] also uses a GPU programming language to target FPGA platforms; however, OpenCL for FPGAs is not necessarily performance-portable with regards to existing GPU OpenCL implementations. In particular, high performance FPGA OpenCL applications need to be structured such that kernels can be cascaded in a pipelined fashion that avoid communication through off-chip memory.

2.5 Summary

This chapter has outlined the hardware and software challenges facing system designers today. We have introduced our solution, *Three Fingered Jack*, to help developers address *portability*, *performance*, and *productivity*, all from a single Python source representation. The details of how Three Fingered Jack enables these capabilities are detailed in the following chapters.

Chapter 3

Three Fingered Jack: Software Approaches

In this chapter, we describe the design and implementation of the major components of Three Fingered Jack. This includes describing the subset of the Python language supported by our tool and the rational behind its selection. We describe our front-end interface with Python, dependence and reordering engine, machine code generator, and run-time environment. This chapter concludes with a performance evaluation of TFJ using several benchmarks and small applications. We evaluate TFJ on both mobile (ARM) and desktop (x86) platforms.

While we reuse many of the components described in this chapter for high-level hardware synthesis, a detailed description of hardware approaches is deferred to Chapter 5.

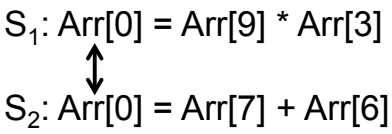
3.1 Dependence Analysis and Reordering Transforms

Exploiting data dependence is the key to TFJ's performance as it gives constraints on the possible ordering of statements in a program. Both the underlying hardware of the target platform and the order in which statements are executed can have a profound impact on performance. Bernstein's conditions for parallel execution [19] give a formal definition of dependence. Stated simply, dependence occurs between two statements, S_A and S_B, if both statements access the same memory location (or register), and at least one of the statements stores a value to memory. As shown in Figure 3.1, there are three different types of dependences in real programs. The types of hazards found in parallelization of software are identical to the class of hazards found in pipelined processor micro-architectures [67]. Details of the three types of dependence are described below:

1. Read-after-write (RAW) – A read-after-write hazard connotates *true dependence* between an operand producer (write) and operand consumer (read). RAW hazards represent dataflow dependences in the source program. An example RAW hazard is presented in Figure 3.1a.

(a) Read-after-write hazard for memory location `Arr[0]`

(b) Write-after-read hazard for memory location `Arr[0]`



(c) Write-after-write hazard for memory location `Arr[0]`

Figure 3.1: The three types of dependence hazards

2. Write-after-read (WAR) – A write-after-write hazard connotates *antidependence* between an operand producer and an operand consumer. In the case of an antidependence, interchanging the source and sink statements will introduce RAW hazard. An example of a WAR hazard is shown in Figure 3.1b.

3. Write-after-write (WAW) – A write-after-write hazard connotates *output dependence*. A WAW hazard is caused by ordering of writes to the same memory location. Figure 3.1c shows an example of a WAW hazard.

So far, we have described dependencies in general terms. For the work in this dissertation, we focus on the acceleration of loop-nests. We must therefore use formalisms to describe and distinguish dependences found in loop-nests. These formalisms allow us to introduce algorithmic tests for *loop dependence*. In turn, results from dependence analysis allows our tools to increase application performance by automatically extracting parallelism.

3.1.1 Theory of Loop Dependence

The formal definition of loop dependence builds upon the earlier description of Bernstein's Conditions. However, loop dependence adds three additional conditions:

1. If a pair of iteration vectors, IV_J and IV_K exists for statements S_A and S_B such that $IV_J == IV_K$ or $IV_J < IV_K$

2. Both S_A and S_B access the same memory location

3. Either statement S_A or S_B write to memory

Many techniques exist for automatically detecting loop dependence [6, 3, 114]; however, before describing these techniques we must first introduce additional terminology: *iteration numbers*, *iteration vectors*, and an *iteration space*. Iteration numbers describe the value

```
for i in range(0,3):
  Y[i] = 2*Y[i]
```

Figure 3.2: Iteration number example: the index variable i assumes iteration values 0,1,and 2.

of an index variable for a given loop iteration. For example, in the loop shown in Figure 3.2, the index variable i assumes iteration numbers 0, 1, and 2. As loops are often nested, iteration vectors extend the idea of iteration numbers to nested loops The elements within an iteration vector capture iteration numbers for each loop in the next; therefore, the length of an iteration vector is equal to the number of loops in a given loop nest. The entries of the iteration vector proceed from the outermost loop index variable to the innermost loop index variable for a nest of loops. As the definition of loop dependence requires an ordering of the iteration vectors, we sort the iteration vectors lexicographically. Finally, the iteration space is the set of all possible iteration vectors.

3.1.2 Analyzing Loop Dependence

The problem of automatic dependence analysis attempts to find iteration vectors (IV_α and IV_β) for a pair of statements (S_A and S_B) such that the following conditions hold:

1. The iteration vector IV_α is less-than or equal to IV_β.
2. The value of each subscript of statements S_A and S_B are equal when evaluated with iteration vectors IV_α and IV_β. We consider only subscripts that are an affine function of the loop index variables.

When both conditions hold, a dependence exists between S_A and S_B. The second condition dictates that S_A and S_B access the same memory location during the course of loop execution. The first condition implies that S_A executes before S_B. Stated another way, S_A is the source for the sink statement S_B. If either condition does not hold, a dependence does not exist between the two statements, and they can be executed in any arbitrary order.

As an example of dependence analysis, consider the loop-nest shown in Figure 3.3. In this example, we are searching for a pair of three entry (one entry for each loop) iteration vectors such the subscripts of the statement are equal. More precisely, in this example we are attempting to find iteration vectors such that all of the following conditions are satisfied:

- The first left and right subscripts are equal: $i_{IV_\alpha} + 2 == i_{IV_\beta}$
- The second left and right subscripts are equal: $j_{IV_\alpha} - 1 == j_{IV_\beta} + 2$
- The third left and right subscripts are equal: $k_{IV_\alpha} == k_{IV_\beta}$

Testing for integer solutions to these equations is equivalent to solving linear Diophantine equations [6]. As solving Diophantine equations is an NP-Complete problem [59], one

```
I Loop:  for i in range(0, 10):
J Loop:    for j in range(0,20):
K Loop:      for k in range(0,30):
               Y[i+2][j-1][k] = 2*Y[i][j+2][k]
```

Figure 3.3: A loop-nest used as our example for dependence testing. In this example, analysis attempts to determine if a dependence occurs between the left-hand and right-hand sides of the statement.

potential way of solving the Diophantine equations found in dependence analysis is to model the problem using integer linear programming. For the example shown in Figure 3.3, testing for dependence can performed by first formulating the problem as in integer program, shown in Equation 3.1, and then checking if the integer program has any solutions.

$$\begin{array}{ll} \min & \emptyset \\ \text{s.t.} & (i_{IV_\alpha} + 2) - i_{IV_\beta} = 0 \\ & (j_{IV_\alpha} - 1) - \left(j_{IV_\beta} + 2\right) = 0 \\ & k_{IV_\alpha} - k_{IV_\beta} = 0 \\ & i_{IV_\alpha} \in \{0, 10\} \\ & i_{IV_\beta} \in \{0, 10\} \\ & j_{IV_\alpha} \in \{0, 20\} \\ & j_{IV_\beta} \in \{0, 20\} \\ & k_{IV_\alpha} \in \{0, 30\} \\ & k_{IV_\beta} \in \{0, 30\} \end{array} \tag{3.1}$$

While using integer linear programming is one possible way of performing dependence analysis, it is not often used in practice. As a polynomial-time solution is not guaranteed to integer programming problems, modern compiler frameworks use conservative dependence testing heuristics to reduce compile time. Conservative heuristics can report the presence of a dependence when none exists; however, a conservative heuristic will never report a lack of dependence when one exists. TFJ uses Banerjee's Test [17] as a conservative testing heuristic of subscripted variables. We do this because it is both fast and accurate for common numerical kernels.

Banerjee's Test relaxes dependence analysis by testing for potential hyperplane intersection using real numbers instead of integers. The Banerjee Test is conservative because it considers intersections for real-valued indices. In Python (and many other programming languages), arrays are indexed using integer values. Using a floating-point value as an array index is a syntax error. However, the conservatism in Banerjee's Test makes it very fast in practice as it involves evaluating a handful of simple arithmetic expressions for each subscript in a loop-nest. As a result, it is suitable for implementation in a dynamic, just-in-time compiler. More exact tests, such as the Omega test [119], that search for integer valued so-

lutions also exist; however, they are both sophisticated and based on integer programming. We believe using Banerjee's Test as the core of our reordering engine is reasonable as it is a good balance between subscript testing accuracy and computational complexity.

Dependence analysis is used to generate a *dependence graph* [89]. The vertices in a dependence graph are the statements in a loop-nest while the edges reflect dependence relationships between statements. Edges are annotated with the type of dependence (true, output, or anti) and the level of loop nesting where the dependence occurs.

3.1.3 Reordering Transforms

After computing dependence information to generate a dependence graph, we are free to perform any number of loop reordering transformations. Reordering transformations change the order in which a loop-nest executes; however, they respect all dependences in order to preserve the meaning of the program [6]. The dependence graph described earlier encapsulates fundamental orderings that a reordering transform must preserve. The benefits of reordering transforms are best illustrated by an example. Figure 3.4 shows three legal orderings of matrix-multiply. Dependence sometimes also allows us to expose parallelism in inner loops by sequentially executing outer loops. In this example, if the I-loop is sequentially executed, then both the J-loop and K-loop may be executed in any order, including in parallel. This is allowed because sequentially executing the I-loop ensures that the left side of the statement will never point to the same memory location as the right side of the statement does.

The inner-loop (K-loop) in Figure 3.4a carries a dependence that prevents inner-loop vectorization because the K loop must run sequentially. This is because it reads and writes the same memory location ($Y[i][j]$) in every iteration. However, dependence analysis also tells us that the I and J loops carry no dependence and can be executed in parallel.

The nest shown in Figure 3.4b is amenable to inner-loop vectorization because the K-loop has been interchanged with the outer-most loop and it has unit-stride memory accesses. As our two conventional ISAs (x86 and ARM) include support for unit-stride vector load and store instructions only, finding an inner loop with unit-stride memory operations is required for vectorization. Now, the outer-loop in Figure 3.4b carries a parallelization-preventing dependence. The middle-loop (I-loop) may be parallelized across multiple CPUs; however, the profitability of this scheme is not guaranteed due to synchronization overhead. This is because the work granularity is relatively small. Finally, Figure 3.4c shows a loop-nest that allows for both outer-loop parallelization across cores and inner-loop parallelism across vector units.

Reordering transformations are automatically performed by TFJ using the algorithm presented in Figure 3.9. We present details our approach to automatic reordering and parallelization in Section 3.3.2.

```
for(i=0;i<n;i++)
  for(j=0;j<n;j++)
    for(k=0;k<n;k++)
     Y[i][j] += A[i][k]*B[k][j]
```

(a) Inner loop is not vectorizable with unit-stride memory instructions; however, the outer loop is parallelizable.

```
for(k=0;k<n;k++)
  for(i=0;i<n;i++)
    for(j=0;j<n;j++)
     Y[i][j] += A[i][k]*B[k][j]
```

(b) Inner loop vectorizable with unit-stride memory instructions. Unfortunately, the outer loop must run sequentially.

```
for(i=0;i<n;i++)
  for(k=0;k<n;k++)
    for(j=0;j<n;j++)
     Y[i][j] += A[i][k]*B[k][j]
```

(c) Inner loop is vectorizable with unit-stride memory instructions and the outer loop is parallelizable.

Figure 3.4: Three legal loop orderings of Matrix-Multiply

3.2 A Restricted Subset of Python

TFJ supports only a small subset of the Python language that is amendable to auto-parallelization techniques. This approach allows the programmer to generate efficient implementations of kernels that manipulate arrays. Applications with kernels that manipulate arrays are common. In the language of computational patterns [84], TFJ is designed to accelerate the dense linear algebra and structured grid patterns. TFJ is also applicable to a subset of the sparse linear algebra pattern in which the structure of matrix enables regular array accesses.

We have chosen to embed TFJ within Python as an embedded domain specific language (EDSL). By using TFJ as a Python EDSL, we gain interoperability with common Python libraries and support the Python syntax already known by thousands of programmers worldwide. We believe exposing TFJ as an EDSL is the correct approach to exporting auto-parallelization technology in Python. As Python is built around a dynamic type system and allows a programmer to mix programming paradigms, attempting to extract parallelism from an arbitrary Python program would face onerous program analysis challenges. However, by embedding TFJ within Python and restricting language support to a small subset of the Python 2.7 language, we can efficiently extract performance for a large number of numerical kernels found in a diverse field of multimedia applications.

We use the Python decorator mechanism to redirect execution to TFJ when it encounters the **@tfj** decorator placed before a function definition. This approach allows the programmer

```
stmt = FunctionDef(identifier name, arguments args,
                   stmt* body, expr* decorator_list)
     | Assign(expr* targets, expr value)
     | For(expr target, expr iter, stmt* body)
     | Global(identifier* names)
     | Expr(expr value)

expr = BinOp(expr left, operator op, expr right)
     | Compare(expr left, cmpop* ops, expr* comparators)
     | Num(object n)
     | Subscript(expr value, slice slice, expr_context ctx)
     | Name(identifier id, expr_context ctx)

expr_context = Load | Store

slice = Index(expr value)

operator = Add | Sub | Mult | Div | Max | Min

cmpop = Eq | NotEq | Lt | LtE | Gt | GtE

arguments = expr* args
```

Figure 3.5: The abstract grammar rules for Three Fingered Jack's automatically parallelizable EDSL. In the grammar shown above, **object** is a built-in Python data type. We require all **objects** to be NumPy single-precision or 32-bit integer data types. The ***** operator expands a list into positional arguments. We have adapted TFJ grammar rules from the Python 2 AST listing [56]

to signal his or her intent by adding the decorator to a conventional Python function. This decorator allows compiler mechanisms of TFJ to introspect the abstract syntax tree (AST) of the function under optimization, type check function arguments, and finally perform parallelization analysis. If the function can be auto-parallelized, the function will execute at upwards of 25000× faster than the naive Python function. If TFJ can not auto-parallelize a function, the function will fall back to execution in the Python interpreter.

3.2.1 Supported Grammar

As TFJ is a language embedded in Python, it adopts a subset of the lexical and grammatical rules of the host language. In Figure 3.5, we present the grammar for TFJ expressions to define the supported subset of the Python language of our EDSL.

The grammar shows the restrictions made to the Python language in order support efficient auto-parallelization as we have removed many constructs from the core Python language. In particular, functions calls from within a kernel and conditional branching are not supported by TFJ. We believe these restrictions are not significant as TFJ targets multimedia applications and these constructs are infrequent in multimedia codes [49]. In addition, we have a more general EDSL that supports lowering a larger subset of Python for efficient execution. However, the more general EDSL does not include support for automatically extracting parallelism.

Auto-parallelizing kernels with function calls in a the loop-nest is rarely supported in commercial compilers due to difficulties with analysis. In these systems, automatic parallelization is usually supported only when function calls can be completely inlined into the calling function. As building the infrastructure required to support this is out of the scope of this dissertation, we did not consider supporting function calls. In addition, function calls are not well supported by the range of hardware targets supported by TFJ. For example, supporting function calls in our high-level hardware synthesis engine would likely increase both the complexity of the tool and generated solutions.

While TFJ does not support function calls in all generality, it does support a small number of intrinsics functions that efficiently map to the underlying hardware. In particular, TFJ supports **max**, **min**, and **pred** intrinsics. The **max** and **min** implement the common maximum and minimum operators, while the **pred** operator implements the conditional ternary operation (similar to the `?:`operator in C/C++). By supporting the conditional ternary operator, the programmer can manually predicate conditional statements for vectorized execution. In addition, our conditional ternary operator is an efficient building block for a future implementation of automatic if-conversion [5].

3.2.2 Supported Data Types

The type checking system in TFJ is currently very simple. It requires that all arrays hold either single-precision floating point values or 32-bit integer values. In addition, the functions used to index arrays must be integer valued expressions. We require single-precision data-types to be NumPy's **float32** and 32-bit integer data-types to be NumPy's **int32**. TFJ performs type-checking at compile-time in the EDSL front-end. Because the type restrictions of TFJ's front-end are not fundamential, it would be straightforward to enable other types, such as double-precision. However, for the class of multimedia applications (speech recognition and optical flow evaluation) evaluated in this disseration, single-precision provides more than enough precision and accuracy.

The compile-time type check uses a traditional static type checking approach [3] to ensure type safety with the arguments passed to the function at the time of first compile. To ensure the argument types match those found during the first compile, the run-time introspects all arguments to make sure they match what was previously found. If the run-time types do not match the compile-time arguments, the entire compilation procedure will be rerun. If the kernel uses data-types not supported by TFJ, it will be executed by the Python interpreter.

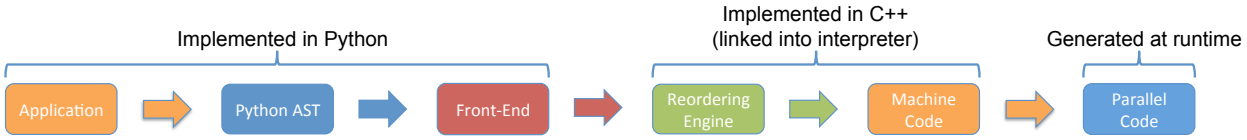


Figure 3.6: TFJ compiler flow: TFJ compiler flow: We start with computation expressed as a Python loop-nest. After syntactic and semantic checking in Python, we convert the Python AST to an XML representation that is fed to our reordering and optimization engine. The optimization engine then generates machine executable code for high-performance execution. Detailed descriptions of the front-end, reordering engine, code generator, and run-time are provided in Sections 3.3.1, 3.1, 3.3.3 and 3.3.5.

```
@tfj
def vvadd(y, a, b, n):
  for i in range(0,n):
    y[i] = a[i]+b[i]
```

Figure 3.7: Vector-vector addition written in Python for TFJ

3.3 The Software Architecture of TFJ

Our dynamic compilation process begins with a dense loop-nest specified in Python using NumPy arrays, as illustrated in Figure 3.6. Our front-end then generates an intermediate XML representation of the abstract syntax tree (AST) that is interpretable by our optimizing compiler. Our compiler analyzes the loop-nest using dependence analysis to enable other transformations such as loop reordering, blocking, and unrolling. Finally, separate backends generate LLVM IR for JIT execution on x86 as well as C++ with vector intrinsics. TFJ's run-time interfaces with the Python interpreter to execute the compiled code. TFJ is implemented in approximately 13000 lines of C++ and 9000 lines of Python.

3.3.1 Python Front-End

When the Python interpreter encounters a function wrapped with the **@tfj** decorator, execution is redirected to our front-end. As described earlier, the front-end requires kernels to be well-typed, and it enforces this restriction using a simple type system. We support loop-nests with no control-flow and affine array indexing functions. These restrictions simplify compiler construction and enable fast dependence checking heuristics. Our run-time compilation has the benefit of dynamic program information. When TFJ accelerated functions are executed, loop bounds are clearly defined and the dimensions of underlying NumPy arrays are known.

Figure 3.7 presents an example of vector-vector addition coded in Python to use the TFJ EDSL. As previously mentioned, our front-end generates an intermediate XML representation of of the abstract syntax tree (AST) for the dependence and code generation

engine. The XML representation of the vector-vector addition kernel is presented in Figure 3.8. Using the XML intermediate representation makes it possible to easily reuse the core dependence and code generation engine with other scripting languages. For example, to port TFJ to a new language, such as Ruby, only the front-end would have to be rewritten. Minor changes to the TFJ run-time would also be required to support interfacing with arrays and objects from a new programming language.

To avoid recompilation, we employ a compiled code cache. If a loop-nest does not use dynamically scoped variables, future recompilations will not be required. The TFJ front-end always attempts to use the code cache first; however, coding using dynamic scoping will require recompilation every time it is called for correct execution. While we have spent considerable effort making all stages of the compiler as efficient as possible, running the complete compilation pipeline requires upwards of 100 milliseconds per invocation on our Intel i7-2600. More discussion on the overheads of run-time compilation is found in Section 4.5.

We have a fall-back compiler that handles a larger subset of Python if and when TFJ rejects a loop-nest due to an unsupported construct. This EDSL does not attempt to exact any parallelism and therefore can handle a larger subset of the Python language, such as branches in loop-nests. We generate optimized sequential code using the same code generation framework described in Section 3.3.3. We do this because sequential Python code rejected by TFJ can become a performance bottleneck and thus in practice requires our fallback EDSL. We currently require the programmer to manually annotate code for the fallback compiler using a Python decorator.

3.3.2 TFJ's Dependence and Reordering Engine

Our reordering engine algorithm is shown in Figure 3.9. It uses a greedy approach when searching the legal reordering space to find profitable parallelism for chip-multiprocessors with vector units. Our reordering algorithms are based on the approach used by Allen and Kennedy [6]; however, we place additional effort on finding unit-stride memory accesses due to the limitations of vector hardware on conventional desktop and mobile microprocessors. Our implementation is also designed to be efficient as it is a key element of our JIT compilation flow.

Our approach is as follows: we sort dependence-free loops by the size of their trip count (iteration space or stated more simply, how many times the loop executes) and select the loop with the largest iteration space for the outer-loop. We desire the outer-most loop to have a large trip count to avoid the overhead of frequent thread creation, if the loop-nest is executed with multiple threads. We then attempt to select a dependence-free inner loop with unit-stride or constant memory accesses in order to use the vector units found on desktop and mobile processors. As an example of how TFJ reorders loops, the matrix-multiply loop-nest shown in Figure 3.4a will be interchanged to achieve the ordering shown in Figure 3.4c. Had we reused Allen and Kennedy's algorithm, it would have produced the KIJ loop ordering of matrix-multiply, shown in Figure 3.4b. While this reordering of matrix-multiply allows for a vectorized inner-loop, it prevents effective use of multicore processors due to

```
<program>
  <Name>vvadd</Name>
  <Var> <Name>y</Name> <Type>float</Type> <Dim>16777216</Dim> </Var>
  <Var> <Name>a</Name> <Type>float</Type> <Dim>16777216</Dim> </Var>
  <Var> <Name>b</Name> <Type>float</Type> <Dim>16777216</Dim> </Var>
  <Var> <Name>n</Name> <Type>int</Type> </Var>
  <ForStmt>
    <LoopHeader>
      <Name>49654096</Name> <Induction>i</Induction>
      <Start>0</Start> <Stop>n</Stop>
    </LoopHeader>
    <LoopBody>
      <MemRef>
  <Name>49654608</Name> <Var>a</Var>
  <Subscript>
    <IndVar>i</IndVar>
    <IndCoeff>1</IndCoeff>
    <Offset>0</Offset>
  </Subscript>
      </MemRef>
      <MemRef>
  <Name>49654864</Name>
  <Var>b</Var>
  <Subscript>
    <IndVar>i</IndVar>
    <IndCoeff>1</IndCoeff>
    <Offset>0</Offset>
  </Subscript>
      </MemRef>
      <CompoundExpr>
      <Name>49654992</Name>
      <Op>ADD</Op>
      <Left>49654608</Left>
      <Right>49654864</Right>
      </CompoundExpr>
      <MemRef>
  <Name>49654352</Name> <Var>y</Var>
  <Subscript>
    <IndVar>i</IndVar>
    <IndCoeff>1</IndCoeff>
    <Offset>0</Offset>
  </Subscript>
      </MemRef>
      <AssignExpr>
  <Name>49655056</Name>
  <Left>49654352</Left>
  <Right>49654992</Right>
      </AssignExpr>
    </LoopBody>
  </ForStmt>
</program>
```

Figure 3.8: Vector-vector addition in TFJ's XML intermediate representation. TFJ can be quickly ported to a new programming language as the interface between the host language and TFJ is a light-weight XML representation. The XML representation abstracts host language implementation details from the core reordering engine.

Algorithm: tfj_codegen
Input: R (Region to analyze for reordering)
k (Current loop nesting depth)
D (Dependence graph for the current region under analysis)
L (List of statements with associated dependence and nesting information)
Output: L (Updated list of statements)

```
1  Compute the set of strongly-connected components in
       the dependence graph for the current region of interest
2  Topologically sort strongly-connected components according
       to the dependence relationship to compute a set of π_i ∈ Π blocks
3  for π_i ∈ Π
4      if π_i has a cycle (it can not parallelized at this loop-nest level)
5          If legal, attempt loop interchange by shifting dependence carried
               at deeper loop nesting depth (k + n) with the current level (k)
6          Remove new region graph, R_i by removing
               all edges that are not in π_i
7          Update L_in with dependence and loop-nesting information for
               the statements in π_i
8          Compute R_i and D_i with dependence and loop-nesting information for
               the statements in π_i
9          Call tfj_codegen(π_i,k + 1,D_i,L) for the region encapsulated by π_i
10     else (π_i can be parallelized at this loop-nest level)
11         Attempt to find a legal permutation of the loop-nests such that
               a dependence-free loop is placed at the outer-most position and
               a dependence-free loop with unit-stride memory access is placed
               at the inner-most position, if found update nesting order for pi_i
12         Update L_in with dependence and loop-nesting information for
               the statements in π_i
```

Figure 3.9: A high-level sketch of the parallelization algorithm used by TFJ. The core of the algorithm is based on Allen's **codegen** algorithm; the details of the algorithm are well documented in several publications [6, 5, 4]. We have modified his algorithm to find profitable parallelism on chip-multiprocessors with vector units. In particular, we attempt to reorder loops to achieve unit-stride memory accesses for vectorization and avoid overhead of thread creation and synchronization.

increased overhead of thread creation and synchronization as the second loop (the I loop) would be run across cores.

Finding unit-stride or constant memory access is particularly important on contempo-

rary microprocessors as the vector units do not currently support strided memory accesses. If the desired memory access patterns can not be extracted, it is unlikely that vectorization on these platforms will be yield high-performance results. Emulating strided vector memory operation requires accesses to be implemented as a series of scalar memory operations followed by a sequence of vector permutations to place the operands in a vector register. The large number of operations required to emulate a strided vector memory operation likely dwarfs the benefits of vectorized arithmetic in all but the most computationally intense kernels.

If we cannot find a reordering that satisfies these criteria, we attempt to expose parallelism in inner loops by allowing outer loops to execute sequentially. For the kernels and applications evaluated in this dissertation, this heuristic works well in practice and generates results comparable to those generated by exhaustive search. This reordering scheme is valid because by running the outer-most dependence-carrying loop sequentially, we are free to reorder all inner-loops as we see fit. This heuristic trades increased parallelism (in particular, vector parallelism) for potentially reduced access locality. As dependence implies operand reduce, placing a dependence carrying loop in an outer loop position could reduce locality. The KIJ reordering (Figure 3.4b) of the canonical IJK matrix-multiply (Figure 3.4a) is an example of a loop interchange scheme that enhances data-parallelism at the expense of operand locality

To enhance opportunities for vectorization, we also perform loop distribution. By distributing loops, we can independently reorder statements in a loop-nest. If loop distribution does not unlock opportunities for vectorization, we apply loop fusion on distributed loops to reconstitute the original loop-nest.

A restricted form of TFJ's dependence engine is used to generate parallel processing engines employing hardware high-level synthesis techniques and described in Chapter 5. The dependence engine used for high-level hardware synthesis is less aggressive when attempting to find unit-stride memory access as it is not required for custom hardware implementations.

3.3.3 Code Generation

After dependence analysis, we generate both C++ and LLVM IR. Our C++ code uses an abstract inline library of vector functions to map different vector ISAs such as ARM Neon, Intel SSE, or IBM AltiVec. The results from our source-to-source translation backend can be used outside our embedded Python environment or on systems that the LLVM JIT poorly supports.

We use LLVM's MCJIT to generate machine code at runtime because it does an excellent job of mapping high-level representations of vector operations in the LLVM IR to the appropriate vector instructions in a processor ISA. TFJ relies on LLVM implementations of scalar optimizations such as common-subexpression elimination, loop-invariant code motion, and strength reduction An example TFJ's LLVM code generation is shown in Figure 3.10 for the vector-vector addition kernel (Figure 3.7). TFJ emits C++ with intrinsics and generates shared objects with GCC at run-time due to the relatively immature nature of MCJIT for ARM (performance is approximately 2× using G++). The C++ intrinsics used

```
1  define void @vvadd(float* noalias %y, float* noalias %a, float* noalias %b, i32 %n) {
2  vvaddentry:
3    %0 = and i32 %n, -32
4    %smZeroTesti = icmp eq i32 %0, 0
5    br i1 %smZeroTesti, label %thirdTermi, label %smLoopi
6
7  smLoopi:
8    %i1 = phi i32 [ %nexti, %smLoopi ], [ 0, %vvaddentry ]
9    %1 = getelementptr float* %y, i32 %i1
10   %2 = getelementptr float* %a, i32 %i1
11   %vector_ld_ptr = bitcast float* %2 to <32 x float>*
12   %3 = load <32 x float>* %vector_ld_ptr, align 1
13   %4 = getelementptr float* %b, i32 %i1
14   %vector_ld_ptr2 = bitcast float* %4 to <32 x float>*
15   %5 = load <32 x float>* %vector_ld_ptr2, align 1
16   %6 = fadd <32 x float> %3, %5
17   %vector_st_ptr = bitcast float* %1 to <32 x float>*
18   store <32 x float> %6, <32 x float>* %vector_st_ptr, align 1
19   %nexti = add i32 %i1, 32
20   %condTesti = icmp slt i32 %nexti, %0
21   br i1 %condTesti, label %smLoopi, label %firstTermi
22
23 secondTermi:
24   ret void
25
26 thirdTermi:
27   %i = phi i32 [ %13, %thirdTermi ], [ 0, %vvaddentry ]
28   %7 = getelementptr float* %y, i32 %i
29   %8 = getelementptr float* %a, i32 %i
30   %9 = load float* %8
31   %10 = getelementptr float* %b, i32 %i
32   %11 = load float* %10
33   %12 = fadd float %9, %11
34   store float %12, float* %7
35   %13 = add i32 %i, 1
36   %14 = icmp slt i32 %13, %n
37   br i1 %14, label %thirdTermi, label %secondTermi
38
39 firstTermi:
40   %preCondTest = icmp eq i32 %nexti, %n
41   br i1 %preCondTest, label %secondTermi, label %smCleanUpi
42
43 smCleanUpi:
44   %i3 = phi i32 [ %cNexti, %smCleanUpi ], [ %0, %firstTermi ]
45   %15 = getelementptr float* %y, i32 %i3
46   %16 = getelementptr float* %a, i32 %i3
47   %17 = load float* %16
48   %18 = getelementptr float* %b, i32 %i3
49   %19 = load float* %18
50   %20 = fadd float %17, %19
51   store float %20, float* %15
52   %cNexti = add i32 %i3, 1
53   %condTesti4 = icmp slt i32 %cNexti, %n
54   br i1 %condTesti4, label %smCleanUpi, label %secondTermi
55 }
```

Figure 3.10: TFJ code generation for vector-vector addition (Figure 3.7) using LLVM. Lines 3 through 5 compute the upper loop bound of the strip-mined loop and check if the function was called with arrays of less than length 32. Lines 7 through 21 perform vectorized addition for vectors of length 32. Lines 26 through 37 perform scalar addition when the function is called with an array length of less than 32. Lines 39 through 55 clean-up the strip-mined vector loop by executing any remaining iterations. Note that our vector lengths (32) are longer than the hardware vector lengths of either AVX (8) or SSE/NEON (4). Using longer vectors enables a variant of software prefetching.

by TFJ are shown in Figure 3.11.

```
//Broadcast v to all four vector entries
inline float4 vecLoadSplatter4(float v);
//Load four entry vector
inline float4 vecLoad4(float *addr);
//Store four entry vector
inline void vecStore4(float *addr, float4 v);
//Add two four entry vectors
inline float4 vecAdd4(float4 a, float4 b);
//Subtract two four entry vectors
inline float4 vecSub4(float4 a, float4 b);
//Divide two four entry vectors
inline float4 vecDiv4(float4 a, float4 b);
//Multiply two four entry vectors
inline float4 vecMul4(float4 a, float4 b);
//Find element-wise minimum of four entry vectors
inline float4 vecMin4(float4 a, float4 b);
//Find element-wise maximum of four entry vectors
inline float4 vecMax4(float4 a, float4 b);
//Select operator of four entry vectors
inline float4 vecSel4(int4 s, float4 t, float4 f);
```

Figure 3.11: TFJ's C++ intrinsics for machines with four entry vectors, such Intel's SSE or ARM's Neon. We have intrinsics with eight entry vectors to map to Intel's AVX extension too.

We generate unaligned vector memory operations because addresses are not guaranteed to be aligned. While unaligned vector loads and stores are potentially slower than their aligned equivalents on older x86 micro-architectures, new micro-architectures significantly close the performance gap. For example, on the Intel Nehalem micro-architecture unaligned vector memory instructions are as fast as aligned operations when the memory operation does not span two cache lines [42].

We also use the C++ generation backend to generate kernels for stand-alone applications. On platforms that do not support Python (or host-based compilation) such as a research operating system or an embedded microprocessor without an OS, we can use TFJ to generate high-performance kernels that are used in a standalone C or C++ application. We have used the offline capability to build a standalone computer-vision based music synthesizer on the Tessellation operating system [37] employing the optical flow application described in Section 4.3.2.

Independent of the backend selected, both approaches provide a function pointer to the JIT compiled code. We register both the function pointer and a hash of the Python source representation with the TFJ runtime system to enable code caching. If a TFJ accelerated function does not change after compilation, we can avoid compilation and executed cached machine code. We also store additional meta-data along with a function pointer for the run-time environment. This meta-data encapsulates if the function is multi-threaded and information, such as the function pointer's prototype, needed to call the JIT-compiled code at run-time.

```
//Set vector length
int set_vlen(int vlen);
//Configure register file and set vector length
int config(int n_intregs, int n_fltregs, int vlen);
//Load unit stride vector
void HardwareVector<T>::load( const T* ptr);
//Load unit strided vector (stride of 0 translates to broadcast)
void HardwareVector<T>::load( const T* ptr, int stride);
//Load unit-stride vector
void HardwareVector<T>::store( const T* ptr);
//Execute ``body of code'' on vector lanes
VT_FETCH( (/*inputs*/), (/*inouts*/), (/*inputs), (/*passthrus*/), ({/*body of code*/ }) );
```

Figure 3.12: Subset of the VTAPI classes and functions used by TFJ to generate code for vector-thread machines.

3.3.4 Vector-Thread Code Generation

Vector-thread (VT) microprocessors [95, 86] merge the ideas from symmetric multiprocessors and vector processors. Following Flynn's Taxonomy [55], VT machines unify architectural ideas about SIMD and MIMD machines. By providing a small amount of additional hardware to a traditional vector processor lane, vector-thread processors provide the illusion of a MIMD machine and the associated ease of programming with the energy-efficiency of a SIMD machine. Our VT code generation framework builds upon the Maven VT Application Programming Interface (VTAPI) [18]. The VTAPI provides a handful of specialized classes, macros, and functions designed to encapsulate the nuances of the VT programming model. The VTAPI provides the programmer with a reasonably productive, yet low-level interface to the underlying VT hardware. As shown in Figure 3.12, TFJ only uses a small subset of the VTAPI; however, for the class of dense and sparse linear algebra kernels targeted by TFJ, these primitives are all that are needed.

The current generation VT processors use the RISC-V instruction-set [140] and are built around the Rocket processor as the scalar CPU and the Hwacha vector-core accelerator. Rocket is an in-order decoupled 5-stage RISC-V processor while Hwacha integrates ideas from both vector-thread and single-lane vector processors[124] to achieve high performance and energy efficiency. TFJ is currently used in an offline compilation mode on VT platforms as cross-compilation is required to generate VT machine code. We compile VT kernels for the RISC-V ISA using GCC 4.6.1 on an x86 workstation running Linux. Recently, Linux has been ported to the RISC-V ISA; therefore, it should be straightforward to run the machine code compilers directly on the RISC-V hardware.

We have run three TFJ accelerated applications on actual VT hardware using the EOS14 test chip. Specifically, we have run a simple convolution kernel (Figure 3.14), the optical flow application of Section 4.3.2, and the speech recognition application discussed in Chapter 6. We have not collected performance results on real VT hardware due to the experimental nature of the EOS14 test chip. While EOS14 CPU core operates at upwards of 1 GHz, the test chip has limited DRAM bandwidth due to a slow memory interface (16-bit read and write channels operating at $\frac{1}{32}$ of the CPU core clock rate).

```
1  __attribute__ ((noinline))
2  void vvadd (float *__restrict__ y,float *__restrict__ a,float *__restrict__ b,int n)
3  {
4    /* starting parloop i*/
5    /* Vectorizable loop */
6    {
7      /* Vectorizable with unit stride operations */
8      vt::config(32,32,32);
9      vt::set_vlen(32);
10     int i_smlen = ((n))- (((n) - 0) % 32);
11     for(int i=0;i<i_smlen;i+=32)
12       {
13          vt::HardwareVector< float >vt_st_ty_1;
14          vt::HardwareVector< float >vt_ld_ta_0;
15          vt::HardwareVector< float >vt_ld_tb_0;
16          vt_ld_ta_0.load((&a[( 0+ 1*i )*1]));
17          vt_ld_tb_0.load((&b[( 0+ 1*i )*1]));
18          VT_VFETCH(
19              (vt_st_ty_1),
20              (),
21              (vt_ld_ta_0,vt_ld_tb_0),
22              (),
23              (VT_FREG),
24              (),
25              (VT_FREG,VT_FREG),
26              (),
27              ({vt_st_ty_1=(vt_ld_ta_0)+(vt_ld_tb_0);
28             }));
29          vt_st_ty_1.store((&y[( 0+ 1*i )*1]));
30       }
31     if( i_smlen!=(n) ) {
32       vt::fence_g_cv();
33     }
34     for(int i=i_smlen;i<(n);i++) {
35       y[( 0+ 1*i )*1]=(a[( 0+ 1*i )*1])+(b[( 0+ 1*i )*1]);
36     }
37     /* ending parloop i*/
38   }
39   vt::fence_g_cv();
40 }
```

Figure 3.13: TFJ code generation for vector-vector addition (Figure 3.7) using the VTAPI. Line 8 configures the vector register file and line 9 configures the hardware vector length. Line 10 computes the upper loop bound for strip-mining. Single-precision vector objects for the source and destination operands are declared in lines 13 through 15, while lines 16 and 17 perform the vector loads for the source operands. The lines 19 and 23 describe the output argument and operand type to the **VT_VFETCH** macro. Lines 21 and 25 provide the same information for the input operands. The **VT_VFETCH** macro performs the vector computation in line 27. The vector store occurs at line 29. Lines 31 through 36 handle strip-mining clean-up on the scalar CPU. Line 32 issues a memory fence to synchronize the vector unit with the scalar processor, and lines 34 through 36 execute the clean-up code for the strip-mined loop.

```
@tfj
def tfj_conv2d(I,O,K,ydim,xdim):
for y in range(3,ydim):
  for x in range(3,xdim):
    for yy in range(-2,3):
      for xx in range(-2, 3):
        O[y][x] = O[y][x] +  K[2+yy][2+xx] * I[y+yy][x+xx];
```

(a) 2d convolution using a 5x5 Laplacian kernel written in Python for TFJ. The Laplacian kernel will sharpen edges in an image.

(b) Image before convolution with an edge sharpening kernel.

(c) Image after convolution with an edge sharpening kernel.

Figure 3.14: An example of TFJ generate code running on real vector-thread hardware. We used the code presented in Figure 3.14a with the image shown in Figure 3.14b to generate Figure 3.14c. Results generated on the EOS14 test chip.

3.3.5 Run-time

To execute a function accelerated by TFJ, our run-time first queries the code cache to check if a compiled version exists. If it does not exist, or the hash changes due to program modifications, the code must be recompiled and the code cache updated. Once the desired code is found, execution meta-data and a function pointer to the compiled machine code are returned to the runtime.

To execute a TFJ accelerated function, TFJ queries the Python interpreter to find references for the arguments to the function and the dimensions (shape) of multidimensional arrays used by the function. The arguments are stored in an array according to the order specified by function pointer prototype. The process of collecting arguments also includes querying the Python interpreter for shape information needed for address calculation used to access elements within a multidimensional array. Arguments and shape data are used to check for pointer aliasing among arrays used as function arguments. If arguments alias the function must be executed in the Python interpreter. By dynamically querying shape information at function call-time, we avoid recompilation when a TFJ accelerated function

is called with a different-sized array. For example, statically including shape information would prevent a function compiled for matrices of size 1024×1024 from working with a matrix of size 1023×1023.

To execute an TFJ accelerated function, the run-time checks whether or not the function has been compiled for multi-threaded execution. If the function is single-threaded, the Python interpreter executes the function by calling the function pointer with the appropriate arguments. If the code has been compiled for multi-threaded execution, the TFJ run-time forks multiple threads for parallel execution using the pThreads API. In this mode, the Python interpreter blocks until the parallel execution completes.

3.4 Summary

In this chapter, we have described the theory and techniques TFJ uses to compile Python for efficient parallel execution. The chapter started with an overview of dependence. We described how dependence applies to programs with nested loops and how to automatically infer dependences in loops.

After describing the key theory behind TFJ's optimizations, we then described the inner workings of TFJ : the front-end, the analysis and reordering engine, and finally the run-time. This involved the description of our approach to loop interchange and our approach to code generation for vector-thread processors. In the next chapter, we provide an in-depth evaluation of TFJ generated software solutions for four kernels and two small applications.

Chapter 4

Three Fingered Jack: Evaluation of Software Approaches

4.1 Overview and Setup

In this chapter, we evaluate performance results for the software backends of TFJ with four kernels (Section 4.2) and two applications (Section 4.3) from the UC Berkeley ParLab [14]. The ParLab applications were chosen to represent compelling new uses of parallel hardware and software. For all benchmarks, we present a comparison to untuned and optimized C++ implementations of the same computation. The untuned C++ implementations are not parallel (thread or data), but they were written either by an experienced programmer or taken from existing applications. We also compare TFJ with optimized Python libraries, if they exist for the given computation. The Python libraries we use for comparison are all implemented in C/C++ for efficiency. For completeness, we also present results when the loop-nests used by TFJ are executed in the Python interpreter. These implementations are represented in the headings of Table 4.4 by "C++ (Untuned)", "C++ (Hand-tuned)", "Python Libraries", and "Pure Python", respectively.

We ran our benchmarks on both x86-based desktop and ARM-based mobile systems. We used a PandaBoard-ES with a dual-core 1.2 GHz Texas Instruments OMAP4460 SoC and 1GB of LPDDR2 RAM for our mobile system. Likewise, we used a 3.4Ghz Intel Core i7-2600k with 8GB of RAM for our desktop system. Both systems run Linux. We used LLVM 3.1 and GCC 4.7.3 for code generators on x86 and ARM, respectively.

4.2 Numerical Kernels

In this section we provide a brief description and provide a source listing of the four kernels used in the evaluation of TFJ.

4.2.1 Vector-Vector Addition

Vector-vector add is the canonical data-parallel benchmark. As vector-vector addition has no data reuse (it streams through memory), it serves as benchmark to show the achieved fraction of memory subsystem performance. A highly optimized implementation of vector-vector addition operating on large vectors (larger than any data caches present in the system) should be able to achieve machine peak DRAM bandwidth. Our Python source for TFJ is shown in Figure 3.7. We used NumPy to compare against a Python library, while our optimized C++ implementation is manually vectorized.

4.2.2 Matrix Multiply

```
@tfj
def matmul(A,B,Y,n):
  for i in range(0,n):
    for j in range(0,n):
     for k in range(0,n):
       Y[i][j]=Y[i][j]+A[i][k]*B[k][j];
```

Figure 4.1: Matrix multiply written in Python for TFJ

Matrix multiply appears in a wide range of applications spanning scientific computing, engineering applications, and the heart of many multimedia applications. For example, it is the key kernel in efficient implementations of both neural network training [22] and support vector machine training [33]. Evaluating Gaussian mixture models, a key kernel in speech recognition, can also be formulated as a matrix multiply [51].

Matrix multiply is a well studied kernel with significant research on both manual [90, 60] and automatic [16, 23] tuning approaches. As an efficient implementation of matrix multiply will be compute bound, this kernel serves as an effective benchmark of the peak floating point performance achievable.

We used 2048×2048 single-precision matrices for matrix multiply because these matrices are too large to fit in any level of the cache hierarchy. Our Python source is shown in Figure 4.1. ATLAS BLAS [143] and NumPy were used for our optimized C++ and Python library comparisons, respectively.

4.2.3 Diagonal Sparse-Matrix Vector Multiply

Optical flow is a key computer vision computation as it relates the motion of objects between two video frames. Many optical flow algorithms address the problem using the conjugate gradient method to solve a system of linear equations. When optical flow is solved with conjugate gradient, a diagonal sparse-matrix vector multiply (SpMV) dominates the solver runtime. We describe the theory behind optical flow when evaluating a whole optical flow application in Section 4.3.2. For our simple kernel benchmark, we used Sundaram's C++ implementations [131] and the SciPy Python library for comparison.

4.2.4 Back Propagation Weight Adjustment

```
@tfj
def bpnn_adjust_weights(delta, ndelta, ly, nlyp1, w, oldw):
for j in range(0, ndelta):
  for k in range(0, nlyp1):
    w[k][j+1] += ((0.3*delta[j+1]*ly[k]) + (0.3*oldw[k][j+1]));
    oldw[k][j+1] = ((0.3*delta[j+1]*ly[k]) + (0.3*oldw[k][j+1]));
```

Figure 4.2: Back propagation weight adjustment kernel from the Rodinia benchmarks rewritten in Python for TFJ acceleration

Back propagation weight adjustment (shown in Figure 4.2) is used to train neural networks. The C++ implementation is from Rodinia [34] and we used the OpenMP accelerated version of the kernel as the optimized implementation. We were unable to find a Python library for this computation.

4.2.5 Numerical Kernel Results

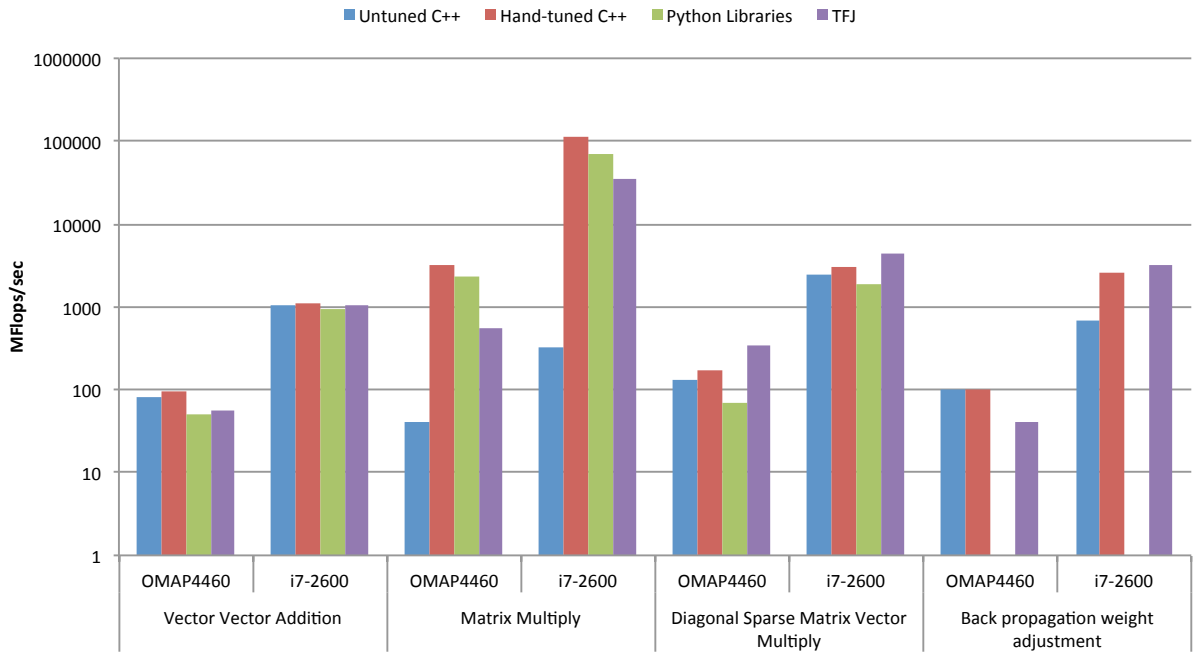


Figure 4.3: Performance results for the small benchmark kernels presented in Section 4.2. Fully tabulated performance results for these benchmarks are available in Table 4.4.

Results for the small kernel benchmarks are presented in Figure 4.3, while fully tabulated results including performance relative to interpreted Python are displayed in Table 4.4. On

the i7-2600, we achieved 95% of the optimized, hand-tuned C++ performance with TFJ for the vector-vector addition kernel. These results indicate TFJ is able to generate solutions that utilize nearly all available DRAM bandwidth on the x86 platform.

We are particularly encouraged by our matrix multiply results on the Intel Core i7-2600 as our performance is within 33% of an optimized, autotuned library. As TFJ uses a compiler-driven approach to code optimization, we compared it to other automatically parallelizing compilers on the same 2048×2048 single-precision matrix multiply problem. As we know of no other automatically parallelizing compilers for Python, we compared with two C compilers: GNU's open-source GCC 4.6.4 and Intel's commercial C compiler, ICC 13.0.1. GCC has claimed support for automatic vectorization since 2004 [109]; however, it does not currently support automatic multicore parallelization. Intel's C compiler has included support for automatic vectorization and parallelization since at least 2004 [21].

```
void mm_2048x2048(float Y[2048][2048], float A[2048][2048],
                  float B[2048][2048])
{
  for(int i=0;i<2048;i++)
    for(int j=0;j<2048;j++)
      for(int k=0;k<2048;k++)
        Y[i][j] += A[i][k]*B[k][j];
}
```

Figure 4.4: Matrix multiply written in C. Statically allocated arrays and fixed loop bounds are required to get the Intel C compiler to automatically vectorize/parallelize matrix multiply.

Our C source for matrix multiply is shown in Figure 4.4. In our C source, we use statically allocated memory to allow the compiler to easily disambiguate memory addresses. If the compiler can not disambiguate array addresses, it must assume the pointer addresses of the matrices could point to the same memory location. This would produce low-performance code as the compiler would be forced to serialize execution due to additional dependences. While there are other approaches to inform the compiler addresses do not alias, we have found static memory allocation to be the most effective approach across a variety of compilers.

Figure 4.5 presents matrix multiply results for the two C compilers and TFJ on an Intel Core i7-2600. All runs with GCC include the additional "-ffast-math" flag to enable reordering of floating point operations. For the two C compilers, we have included baseline performance results when automatic parallelization and vectorization are disabled but full scalar optimizations are enabled. These data points are presented as "IJK GCC -O3" and "IKJ ICC -O3 -no-vec" for GCC and ICC, respectively. Using scalar optimizations alone, ICC is approximately 4× faster than GCC.

We next consider automatic vectorization without multicore parallelization enablined using the three platforms under consideration. As shown in column "IKJ ICC -O3 -vec", automatic vectorization improves ICC's performance by a factor of 4.2×. GCC is unable

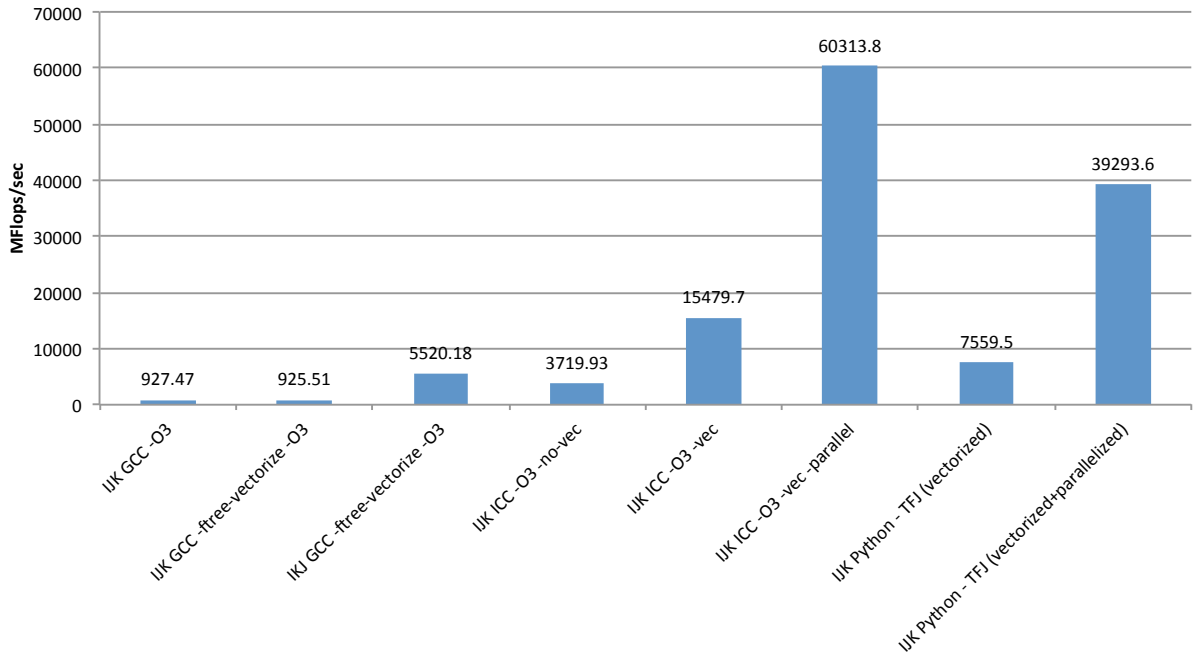


Figure 4.5: 2048×2048 matrix multiply performance comparison. Results generated using GCC 4.6.4 or ICC 13.0.1 use the C source presented in Figure 4.4. TFJ results generated using the Python source shown in Figure 4.1. In the plots labels on the bar chart,"IJK" signifies the IJK loop-nest ordering while "IKJ" implies the "IKJ" ordering. The compiler flags used for the C versions of the benchmark are shown on the bar chart. Both C and Python used single-precision matrices and all experiments were run on a 3.4 GHz Intel Core i7-2600. All implementations compiled with GCC also include flags to enable reordering of floating point operations.

to successfully vectorize the simple matrix multiply code with the IJK loop ordering (see column "IJK GCC -O3 -ftree-vectorize"). However, when we apply manual loop interchange (Figure 4.6) to swap the J and K loops, GCC is able to successfully vectorize matrix multiply, albeit with a minor amount of assistance from the programmer. GCC automatic vectorization results are shown in the column "IKJ GCC -O3 -ftree-vectorize". We also present TFJ results for single-threaded vectorized execution in "Python - TFJ (vectorized)". Our single-threaded vectorized TFJ results are approximately 1.37× faster than the best GCC results and 49% of the single-threaded ICC results. As TFJ performs automatic loop interchange in order to obtain the IKJ loop ordering (manually performed in our use of GCC), we are particularly encouraged by our results.

Finally, we evaluate matrix multiply on TFJ and ICC with both automatic vectorization and parallelization enabled. As GCC does not currently include automatic parallelization capabilities, we limit our evaluation to just TFJ and ICC. TFJ generated results achieve 65% of ICC's performance when all optimizations are enabled.

While TFJ is unable to outperform a commercial compiler, it is able to outperform

```
void mm_2048x2048_IKJ(float Y[2048][2048], float A[2048][2048],
                      float B[2048][2048])
{
  for(int i=0;i<2048;i++)
   for(int k=0;k<2048;k++)
      for(int j=0;j<2048;j++)
        Y[i][j] += A[i][k]*B[k][j];
}
```

Figure 4.6: Matrix multiply written in C with manual loop interchange of the J and K loops.

the ubiquitous open-source compiler GCC. Perhaps it is more important to evaluate TFJ performance with naive Python execution. Naive Python execution of matrix multiply results in 1.4 MFlops/sec, while the same Python source code compiled with TFJ results in nearly 40 GFlops/sec, a 25000× performance improvement. While we are encouraged by our performance results relative to optimizing C compilers, we designed TFJ to help productivity programmers write high performance software in Python. To that end, our results show three-orders of magnitude better performance with TFJ than raw Python. This result clearly indicates that we have achieved our performance goal for small kernels; however, whole applications are required to truly evaluate TFJ. We describe the evaluation of TFJ on whole applications in Section 4.3.

4.3 Small Applications

Kernel performance results are not enough to fully demonstrate a programming system; therefore, we evaluated TFJ on two full applications representative of emerging workloads: content-aware image resizing and the computation of optical flow.

4.3.1 Content-Aware Image Resizing

Content-aware image resizing [15], also known as seam carving, resizes images by removing "boring regions". The canonical example of content-aware image resizing is shown in Figure 4.7. The seam carving algorithm computes an energy function on the image to determine the least interesting regions. It then computes a connected path of least-interest through the image and removes the seam. The algorithm iteratively applies this procedure until the desired image resolution has been achieved.

Our implementation of seam carving is shown in Figure 4.8. We first blur the image using the function **conv2d** (Figure 4.8a) to remove noise and then use the function **grad2d** (Figure 4.8b) to compute the gradient. The gradient extracts edges from an image. Regions with a small gradient have few edges and are unlikely to be interesting. We use a two-dimensional convolution kernel to compute the gradient. After computing the image gradient, we compute the minimum cost path through the image using the function **compute_cost** (Figure

(a) Original image

(b) Retargeted image

Figure 4.7: Original image and retargeted image after removal of 750 vertical seams. Original image of the Broadway Tower reproduced from Wikipedia

4.8c). The minimum cost seam is computed by backtracking through the memorization table generated by **compute_cost**. To compare against Python libraries, we use SciPy and NumPy implementations of convolution and gradient calculation. The **conv2d**, **grad2d**, and **compute_cost** kernels have been manually vectorized and parallelized in our optimized C++ implementation.

```
@tfj
def conv2d(I,O,K,ydim,xdim):
  for y in range(3,ydim):
    for x in range(3,xdim):
      for yy in range(-2,3):
        for xx in range(-2,3):
          O[y][x]+=\
            K[2+yy][2+xx]*\
            I[y+yy][x+xx];
```

(a) Convolution kernel

```
@tfj
def grad2d(I,O,K,ydim,xdim):
  for y in range(3,ydim):
    for x in range(3,xdim):
        O[y][x]=\
          (I[y][x-1]-I[y][x])*\
          (I[y][x-1]-I[y][x])+\
          (I[y-1][x]-I[y][x])*\
          (I[y-1][x]-I[y][x]);
```

(b) Gradient kernel

```
@tfj
def compute_cost(Y,G,ydim,xdim):
  for i in range(5,ydim):
    for j in range(5,xdim):
      Y[i][j]=G[i][j]+min(min(Y[i-1][j-1],Y[i-1][j]),Y[i-1][j+1]);
```

(c) Dynamic programming kernel

Figure 4.8: A portion of the TFJ accelerated kernels used in content-aware image resizing.

We evaluated content-aware image resizing by removing 750 vertical seams from the

![h]

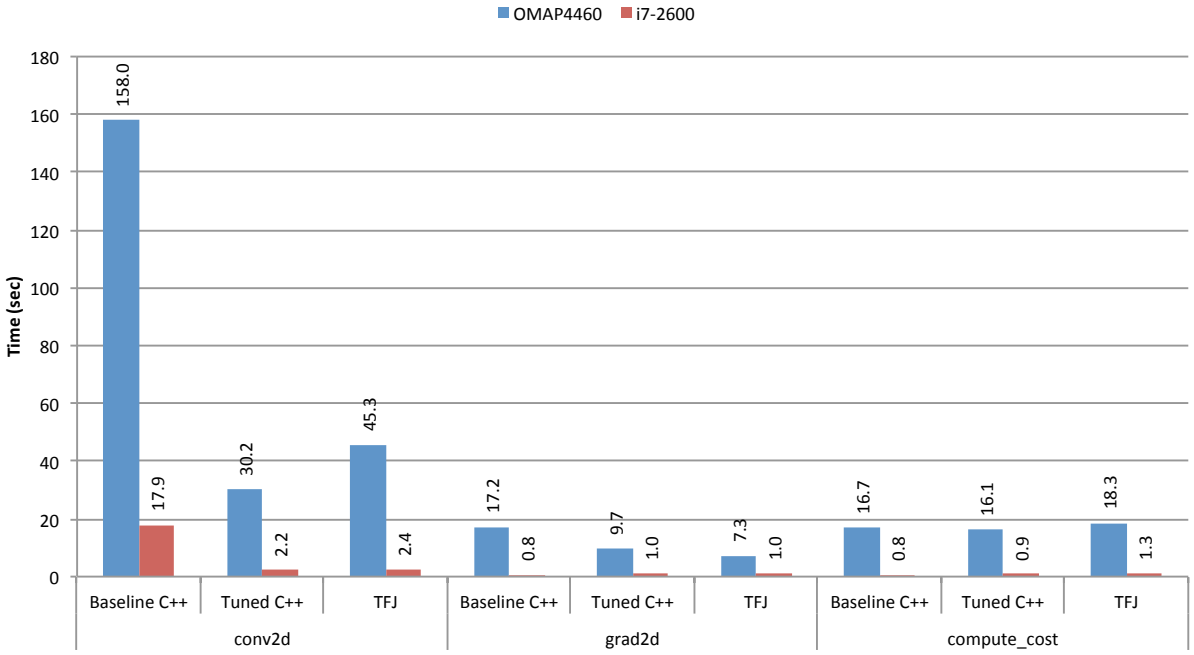


Figure 4.9: Run-times for the kernels used in content-aware image resizing

Implementation	OMAP4460	i7-2600
TFJ	81.2	7.1
Pure Python	>82400.0	34648.8
Python Libraries	18873.1	2238.6
C++ (Untuned)	205.4	20.4
C++ (Hand-tuned)	68.5	4.6

Table 4.1: Image resizing performance results across five different implementations

image of the Broadway Tower shown in Figure 4.7a. The original image has a resolution of 1428×968 and after resizing, the retargeted image has a resolution of 678×968. The time to remove 750 seams using a variety of different implementations is shown in Table 4.1, while a per-kernel runtime breakdown is presented in Figure 4.9. The TFJ implementation of content-aware image resizing outperforms all implementations except for hand-tuned C++ on both ARM and x86 platforms. The highly optimized C++ implementations outperform TFJ by 16% and 35% on ARM and x86 platforms, respectively. As shown in Figure 4.9, performance of TFJ generated kernels and highly tuned C++ are very similar for content-aware image resizing. The performance discrepancy between TFJ and highly-tuned C++ comes from two primary sources. First, the control code for the TFJ implemention remains in Python and interpreted Python is signifantly slower than C++. Second, as described in

Section 3.3, using TFJ accelerated kernels has a small amount of overhead due to kernel code search and argument address lookup.

4.3.2 Horn-Schunck Optical Flow

(a) First frame

(b) Second frame

(c) Optical flow computed between the frames shown in Figure 4.10a and 4.10b.

(d) Optical flow displacement legend. Legend reproduced from Kondermann [85]

Figure 4.10: An example of optical flow

Optical flow computes the apparent motion of pixels between a pair of two images. Optical flow is a building block of many computer vision and multimedia applications. Common uses of optical flow include object tracking [131], video interpolation [142], action recognition [139], and object segmentation [28]. Other uses include motion estimation for video compression [107].

An example of optical flow is shown in Figure 4.10. In this example, we are calculating the motion of the hands between the frame shown in Figure 4.10a and the frame shown in Figure 4.10b. The resulting motion is shown in Figure 4.10c using the displacement

visualization scheme shown in Figure 4.10d.

$$I_x U + I_y V + I_t = 0 \tag{4.1}$$

Solving for optical flow between a pair of images has been approached with a diverse set of techniques. All approaches start with the same motion constraints between a pair of images. The motion constraint equation is shown in Equation 4.1 in which I_x, I_y, and I_t are the time and spatial intensity derivatives. V and U are the vertical and horizontal components of the pixel velocities.

$$\underset{U,V}{\operatorname{argmin}} \int\int \left(I_x U + I_y V + I_t\right)^2 + \alpha^2 \left(|\nabla U|^2 + |\nabla V|^2\right) \mathrm{d}x\mathrm{d}y \tag{4.2}$$

The Horn-Schunck [70] method of computing optical flow solves the motion contraint equation (Equation 4.1) by formulating the problem as an optimization problem using the calculus of variations. The resulting energy functional is given in Equation 4.2. Applying the two-dimensional Euler-Lagrange equation to the functional results in the coupled partial differential equations shown in Equations 4.3 and 4.4.

$$\left(I_x U + I_y V + I_t\right) I_x - \alpha^2 \Delta U = 0 \tag{4.3}$$

$$\left(I_x U + I_y V + I_t\right) I_y - \alpha^2 \Delta V = 0 \tag{4.4}$$

Rearranging Equations 4.3 and 4.4 results in the system of equations shown in Equation 4.5. The spatial derivatives (ΔU and ΔV) can easily be approximated using a numerical finite difference scheme such as a five-point Laplacian stencil. After numerically evaluting the spatial derivates, computing optical flow using the Horn-Schunck method requires solving a large linear system of equations.

$$\begin{bmatrix} -I_t I_x \\ -I_t I_y \end{bmatrix} = \begin{bmatrix} {I_x}^2 & I_x I_y \\ I_x I_y & {I_y}^2 \end{bmatrix} \begin{bmatrix} U \\ V \end{bmatrix} - \alpha^2 \begin{bmatrix} \Delta U \\ \Delta V \end{bmatrix} \tag{4.5}$$

To solve the linear system shown in Equation 4.5, we use the conjugate gradient method as the matrix is positive semi-definite for all non-zero values of α [105]. While we used Horn-Schunck to solve optical flow, techniques for this problem span the full gamut of solutions from custom analog VLSI chips [72] to non-linear, non-convex optimization techniques [27].

We evaluate our implementations of Horn-Schunck optical flow with 100 frames of video. The evaluation video has a resolution of 320×240 pixels and the linear solver uses 50 iterations of the conjugate gradient method. The key kernels used in optical flow are shown in Figure 4.3. To compare TFJ with Python libraries[1], we use the SciPy implementation of conjugate gradient [38]. We also wrote our own C++ implementation of Horn-Schunck optical flow. In our optimized C++ implementation, we manually vectorized all kernels, while we also parallelized the sparse matrix vector kernel (shown in Figure 4.12d) using OpenMP.

[1]Many thanks to Michael Anderson for providing the Python libraries implementation of Horn-Schunck optical flow.

Implementation	OMAP4460	i7-2600
TFJ	83.4	4.5
Pure Python	>82400.0	24146.7
Python Libraries	436.8	29.0
C++ (Untuned)	72.8	4.4
C++ (Hand-tuned)	52.5	1.9

Table 4.2: Optical flow performance results across five different implementations

Kernel	Call count
spmv	5202
vsaxpy	10200
vzaxpy	5100
vdot	10302

Table 4.3: Optical flow kernel call counts for the results shown in Table 4.2

Our optical flow results are shown in Table 4.2. TFJ generated solutions for optical flow are significantly faster than the Python libraries approach for both x86 and ARM. The Python libraries approach uses the highly optimized SciPy's conjugate gradient solver to numerically compute the differential equations used in optical flow. However, TFJ is slightly outperformed by both C++ implementations of optical flow.

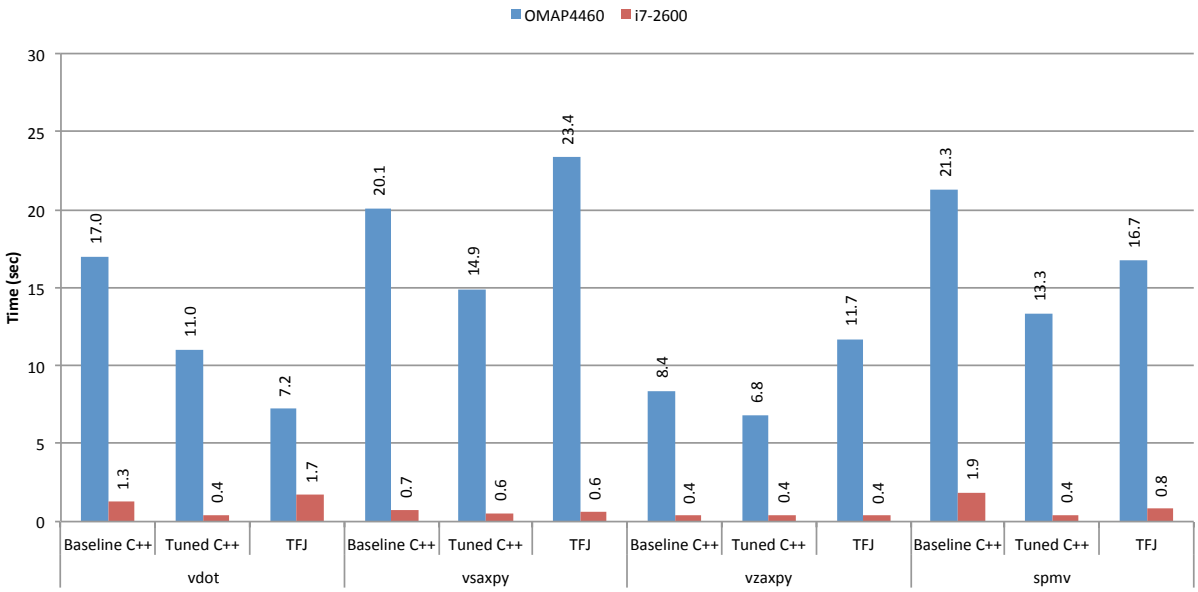


Figure 4.11: Run-times for the kernels used in optical flow

The performance differential between TFJ and the C++ implementations can largely be attributed to the overhead of calling a TFJ accelerated function. As shown in Table 4.3, solving optical flow using conjugate gradient requires calling a small number of kernels many times. In particular, the **vdot** and **vsaxpy** kernels are called 100 times per frame, while **spmv** and **vzaxpy** are called 50 times per frame. The overheads associated with more significant control flow in Python can impact overall performance. A per-kernel runtime breakdown is presented in Figure 4.11. This figure shows time spent per computation from the perspective of the TFJ runtime; therefore, it does not include any of the overheads

associated with calling out of Python into a TFJ accelerated function. On the i7-2600 platform, the total running time of the optical flow application is 1.9s, 4.4s, and 4.5s for the untuned C++, tuned C++, and TFJ implementations, respectively. However, the total time spent in the key kernels (Figure 4.11) is 1.8s, 4.2s, and 3.5s for the same platforms. While the C++ implementations spent more than 95% of the runtime in the key kernels, the TFJ implementation spent only 78% of its runtime in those same key kernels. More detailed profiling of the TFJ implementation shows that at least 20% of the runtime is spent in application control logic that is implemented in interpreted Python.

As for the per-kernel performance discrepancies, the tuned C++ implementation of **vdot** uses an optimized reduction implementation. As TFJ does not currently perform reduction detection, the TFJ implementation of **vdot** is limited to scalar execution, which in turn limits performance. However, TFJ's performance is within a factor of two for the other kernels compared to both C++ implementions. In addition, the TFJ implementation of the **spmv** kernel is more than 2× faster than the untuned C++ implementation. More importantly, using TFJ guarantees a "correct-by-construction" based approach to generating parallel code. The equivalent tuned C++ implementations using SSE or AVX often have subtle bugs related to the low-level nature of the underlying intrinsics. Correcting these bugs can require a significant amount of developer effort; however, using TFJ enables a high performance solution with hours-to-days less developer time.

We are encouraged that TFJ generated results are faster than the equivalent library-based Python program, within 98% performance of an equivalent untuned C++ implementation, and within a factor of three compared to a manually tuned C++ implementation. The results from the TFJ implementation of optical flow indicate that future work should focus on accelerating application control logic in SEJITS-based approaches. In more complex applications with many kernels and significant control flow, the overhead interpreted Python could dominate the benefits provided by software specializers. As Amdahl's Law [7] also applies to software, perhaps a coordination and control specializer for computational patterns is needed. This would allow application control logic to execute more efficiently.

```
@tfj
def vdot(a1, a2, b1, b2, dst, width, height):
  for y in range(0,height):
    for x in range(0,width):
      dst[0] = dst[0] + a1[y][x] * b1[y][x] + a2[y][x] * b2[y][x]
```

(a) **vdot** matrix reduction kernel

```
@tfj
def vsaxpy(alpha, x1, x2, y1, y2, width, height):
  for y in range(0,height):
    for x in range(0,width):
      y1[y][x] = y1[y][x] + alpha[0] * x1[y][x]
      y2[y][x] = y2[y][x] + alpha[0] * x2[y][x]
```

(b) **vsaxpy** scaled vector addition

```
@tfj
def vzaxpy(alpha, x1, x2, y1, y2, width, height):
  for y in range(0,height):
    for x in range(0,width):
      x1[y][x] = alpha[0] * x1[y][x]  + y1[y][x]
      x2[y][x] = alpha[0] * x2[y][x]  + y2[y][x]
```

(c) **vzaxpy** scaled vector addition. Note **vzaxpy** scales the output vector while **vsaxpy** scales input vector.

```
@tfj
def spmv(Ix, Iy, x1, x2, y1, y2, w1, h1):
  for y in range(1,h1):
    for x in range(1,w1):
      y1[y][x] = 0.4 * x1[y][x]+\
          Ix[y][x] * Ix[y][x] * x1[y][x] +\
          Iy[y][x] * Ix[y][x] * x2[y][x] - \
          0.1 * x1[y][x-1] - 0.1 * x1[y-1][x] - \
          0.1 * x1[y][x+1] - 0.1 * x1[y+1][x]
      y2[y][x] = 0.4 * x2[y][x] +\
          Iy[y][x] * Iy[y][x] * x2[y][x] +\
          Ix[y][x] * Iy[y][x] * x1[y][x] - \
          0.1 * x2[y][x-1] - 0.1 * x2[y-1][x] - \
          0.1 * x2[y][x+1] - 0.1 * x2[y+1][x]
```

(d) **spmv** sparse matrix vector multiply kernel

Figure 4.12: The key kernels used in solving optical flow

4.4 Autotuning Using TFJ Demonstrated With Matrix Multiply

Matrix multiply is a well understood numerical kernel with many decades of research spent developing optimization techniques. While matrix multiply is a numerical kernel and not an entire application, the design space for an efficient implementation of matrix multiply is very large and can be explored for trade-offs in both software (this Section) and hardware (Section 5.4) implementations.

Matrix multiply benefits from both locality and parallelism optimizations such as: loop interchange, loop tiling, register tiling, multithreaded execution, and vectorized instructions. In fact, the code transformations that benefit matrix multiply are larger than the set of transformations performed by TFJ. The combination of all potential optimizations creates a design space that is too large for a human to manually explore. Instead, an automatic approach is required.

Autotuning [23] is a programmatic approach to design space exploration by which a program automatically generates potential solutions and then evalutes the performance of it. The solution with the highest performance is recorded and returned to the user. The autotuning approach to code generation is done offline and the result is saved for later execution. For example, when tuning matrix multiply, the best combination of locality and parallelism optimizations are empirically determined during a long offline tuning run. When the user performs a matrix multiply after tuning, the fastest implementation from the tuning run is automatically selected.

As autotuning requires a "compiler-in-the-loop" to generate and evaluate potential solutions, past approaches to autotuning have relied on a combination of shell scripts and templated C code. In this approach, a tuning script generates potential solutions by assigning parameters in the templated C code. While this approach has worked for past autotuning projects, it is relatively brittle due to many external tool dependences. TFJ makes it possible to integrate all the autotuning logic into a single Python script. As Python includes rich string manipulation libraries, such as string templates, and TFJ includes an automatically parallelizing and vectorizing JIT compiler, a complete autotuning environment can be provided without additional scripts or external compilers.

To demonstrate the potential for building autotuners using TFJ, we have constructed a simple autotuner for matrix multiply. This autotuner considers only square matrices with a power-of-two leading dimension; however, this limitation is not fundamental. Extending it to more general matrices would be straightforward but would require handling for significantly more edge cases. In addition, our autotuner considers only a single optimization: loop tiling to enhance cache locality. By enhancing locality, matrix multiply can be greatly accelerated because operands will be fetched from fast cache memories instead of DRAM.

We also exhaustively explore the design space of legal loop tilings to find the best tile size. More efficient search algorithms, such as a genetic algorithm, are also possible and easily implementable in Python. We have decided to use a naive search approach as a more advanced search algorithm complicates the autotuner implementation. In future work, more sophisticated autotuners could easily be built on top of TFJ.

```
1  def make_mm(iblk,jblk,kblk):
2      mb =  '''
3      from VecFunction import *
4      from decorators import *
5      @tfj
6      def bmm$iiblk$jjblk$kkblk(A,B,Y,nI,nJ,nK):
7        for i in range(0,nI):
8          for j in range(0,nJ):
9            for k in range(0,nK):
10             for ii in range(0,$iiblk):
11               for jj in range(0,$jjblk):
12                 for kk in range(0,$kkblk):
13                   Y[ii+i*$iiblk][jj+j*$jjblk] = Y[ii+i*$iiblk][jj+j*$jjblk] +
14                   A[ii+i*$iiblk][kk+k*$kkblk]*B[kk+k*$kkblk][jj+j*$jjblk];'''
15     s = Template(mb);
16     sstr = s.substitute(iiblk=str(iblk),jjblk=str(jblk),kkblk=str(kblk))
17     return sstr
```

Figure 4.13: A Python function to generate variants of matrix multiply with different loop tiling dimension. Lines 1 through 18 generate tiled implementations of matrix multiply. The parameters **iblk**, **jblk**, and **kblk** of **make_mm** describe the dimensions of the cache tile. The output of this function is an implementation of loop titled matrix multiply as a string.

Writing autotuners in Python using TFJ relies on two key language features. First, Python allows for dynamic loading of new modules. We exploit this feature when we generate new configurations using **make_mm** (Figure 4.13). The other language feature we use is the **eval** function that allows the user to execute a dynamically generated code string from within the Python interpreter. Our autotuner uses **eval** to execute the different loop tiled matrix multiply strings generated by **make_mm**.

The results for autotuned matrix multiply are shown in Figures 4.15 and 4.16 for ARM and x86 platforms, respectively. The source of our prototype autotuner for square matrix multiply is shown in Figures 4.14 and 4.13. A line-by-line description of the autotuning software is provided in both captions. We compare the results of autotuned matrix multiply to the baseline matrix multiply shown in Figure 4.1 for correctness checking.

On both platforms, the overhead of loop tiling dominates performance for small matrix sizes. As the small matrices already fit into cache memory, there is no benefit to blocking loops for increased locality. With larger matrices, tiling loops results in increased performance on both platforms. The matrix size at which the autotuned code outperforms the baseline code depends on the specific platform. On the OMAP4460, the autotuned loop tiled code outperforms the naive code with matrices greater than 64×64, while on the i7-2600 matrices need to be larger than 128×128. The crossover point for the platforms is likely due to differences in memory hierarchy. In partcular, the i7-2600 has an eight megabyte L3 cache, while the OMAP4460 does not have a third-level cache.

We achieve a maximum performance improvement of 1.53× and 1.45× over our baseline implementation matrix multiply for our OMAP4460 and i7-2600 platforms, respectively. While our evaluation of autotuning was fairly limited, the performance improvements clearly demonstrate the potential for autotuned codes using TFJ.

```
1  T = list();
2  n_itrs = 8;
3  for i_n in [32,64,128,256,512,1024,2048]:
4      n = np.int32(i_n);
5      A = np.array(np.random.rand(n,n), dtype=np.float32);
6      B = np.array(np.random.rand(n,n), dtype=np.float32);
7      Y0 = np.zeros((n,n), dtype=np.float32); Y1 = np.zeros((n,n), dtype=np.float32);
8
9      sn = 0.0;
10     for r in range(0, n_itrs):
11       Y1 = np.zeros((n,n), dtype=np.float32);
12       start = time.time(); matmul(A,B,Y1,n); elapsed = time.time() - start;
13       fps = 2*(n**3) / (elapsed * 1e6)
14       if(r != 0):
15          sn += fps;
16     m_nflps = sn/(n_itrs-1)
17
18     R = dict();
19     for iblk in (2**x for x in xrange(2,14) if 2**x < n):
20       for jblk in (2**x for x in xrange(2,14) if 2**x < n):
21         for kblk in (2**x for x in xrange(2,14) if 2**x < n):
22           mm_n = 'mm$iiblk$jjblk$kkblk'
23           mm_T = Template(mm_n);
24           mm_S = mm_T.substitute(iiblk=str(iblk),jjblk=str(jblk),kkblk=str(kblk))
25           fp = open(mm_S + '.py', 'w');fp.write(make_mm(iblk,jblk,kblk)); fp.close();
26           fnS = 'mm.b'+mm_S
27           fstr = fnS + "(A,B,Y0,np.int32(n/iblk),np.int32(n/jblk),np.int32(n/kblk))"
28           mm = __import__(mm_S)
29           s = 0.0;
30           for r in range(0, n_itrs):
31             Y0 = np.zeros((n,n), dtype=np.float32);
32             start = time.time(); eval(fstr); elapsed = (time.time() - start);
33             fps = 2*(n**3) / (elapsed * 1e6)
34             if(r > 0):
35               s += fps;
36
37             m_flps = s/(n_itrs-1)
38             fStr = '(' + str(iblk) + ',' + str(jblk) + ',' + str(kblk) + ')';
39             R[fStr] = m_flps;
40
41     fastConf = None; fastVal = 0;
42     for v in R:
43         if(R[v] > fastVal):
44             fastVal = R[v];
45             fastConf = v;
46     T.append((n,fastVal,fastConf,m_nflps));
47 for t in T:
48     print t
```

Figure 4.14: An autotuner for matrix multiply written in Python using TFJ. Line 1 allocates a list to keep track of the best performing configuration for each problem size. The problem sizes for which we are generating solutions are shown in line 3. Lines 4 through 16 allocate matrices and then evaluate the performance of the naive TFJ matrix multiply code for the current problem size. Lines 19 through 21 define the loop tiling search space, while lines 22 through 27 generate perform code generation using **make_mm** and write the resulting module to the filesystem. Line 28 imports the newly generated loop tiled matrix multiply into the Python environment. Lines 30 through 35 evaluate the performance of the generated loop tiled matrix multiply. The generated tiled matrix multiply is called on line 32 using the **eval** function. Lines 41 through 46 keep track of the best performing configuration, while Lines 47 and 48 report the best configuration and performance for each problem size.

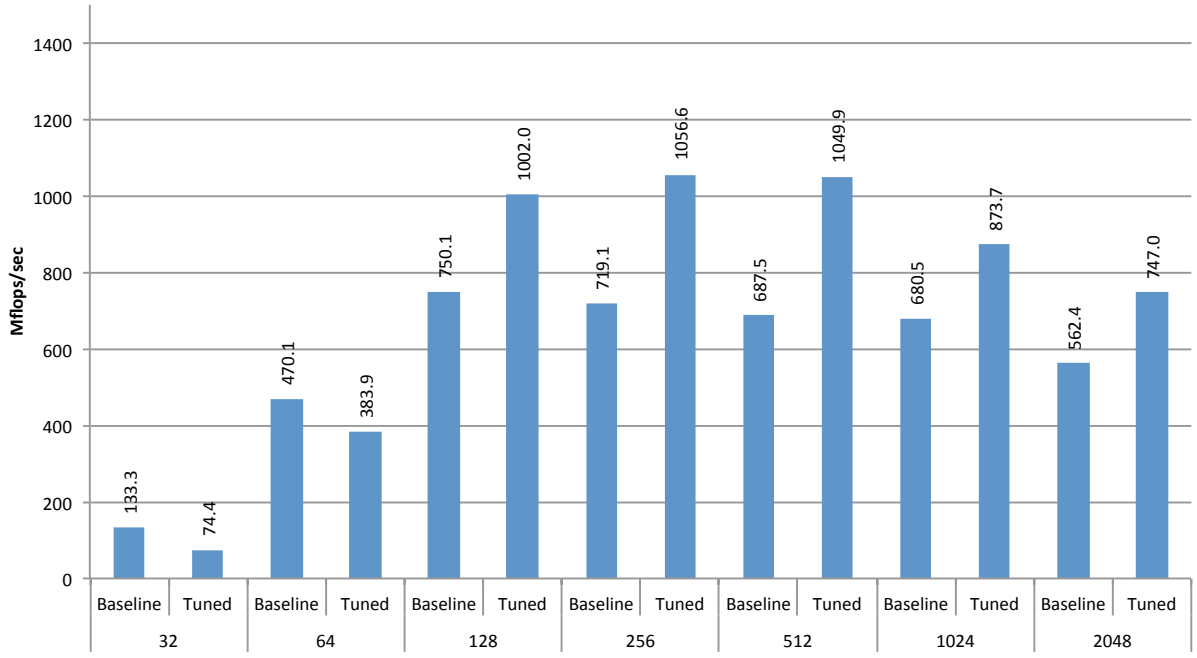


Figure 4.15: Autotuned matrix multiply results on an Texas Instruments OMAP4460. We present baseline and tuned results for matrices from size 32 to 2048.

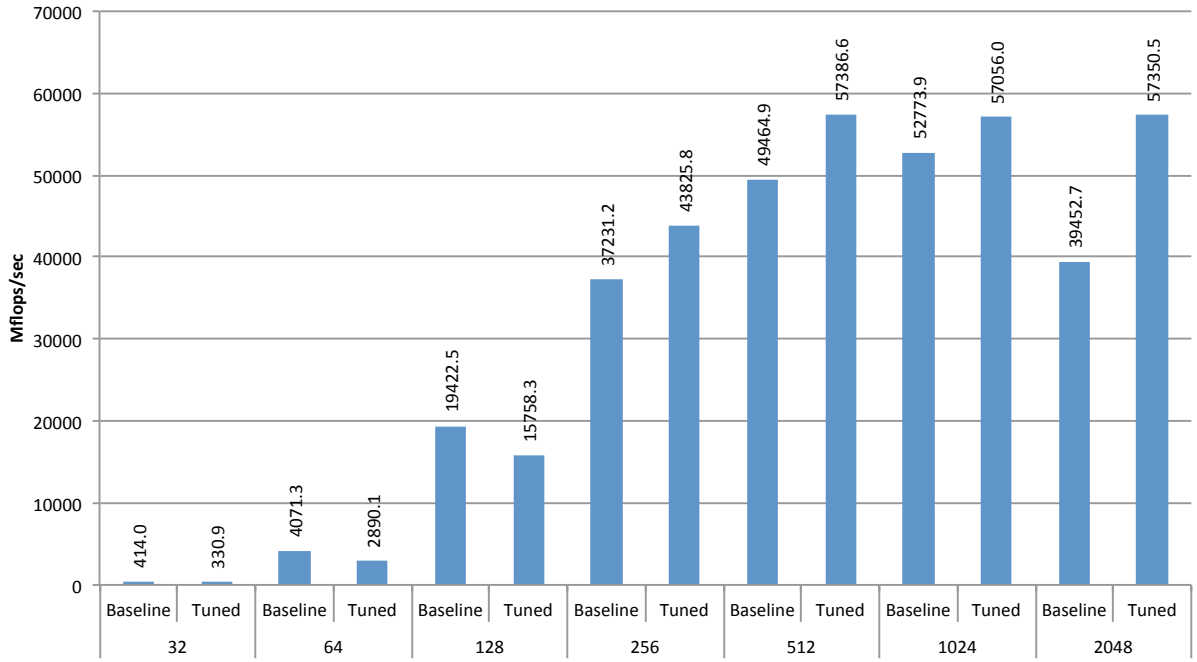


Figure 4.16: Autotuned matrix multiply results on an Intel Core i7-2600. We present baseline and tuned results for matrices from size 32 to 2048.

4.5 Overheads for Runtime Code Generation

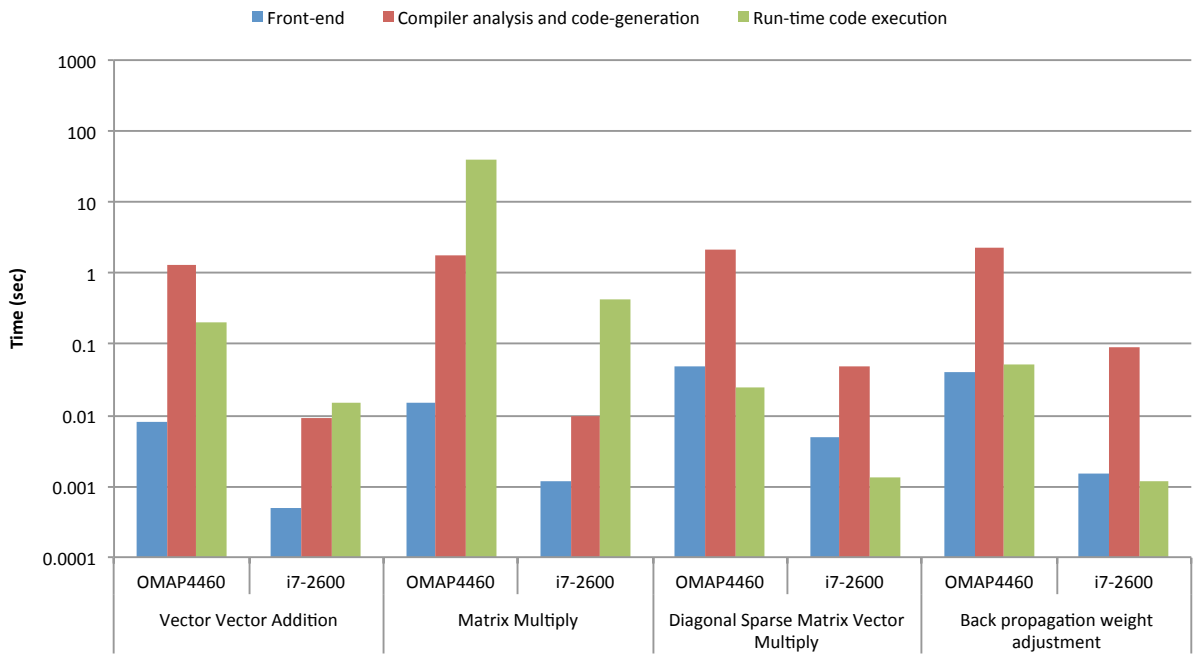


Figure 4.17: Run-time breakdown for the numerical kernels described in Section 4.2. Note the logarithmic time scale on the bar chart.

We have gone to significant lengths to ensure our compiler runs fast enough to be used interactively in a JIT-based framework. Figure 4.17 presents a breakdown of execution time into front-end parsing and syntax checking, compiler analysis and code generation, and execution for the benchmark kernels listed in Section 4.2. These results present only a single execution of a given benchmark. That is, we execute the kernel only once after compiling. This reflects a pathologically bad situation in which recompilation occurs for each call to a TFJ accelerated function.

As mentioned in Section 3.3.3, LLVM MCJIT performance is currently subpar on the ARM platform. This requires the use of the GNU C++ compiler as the machine code generator. The added file I/O and general overhead of the C++ compiler results in greater than one second compile times. We expect these issues to be resolved with a new release of LLVM.

On several benchmarks, the actual code execution runs faster than the compilation steps: front-end parsing, compiler analysis, and machine code generation. In general, we attribute this phenomena to the relatively low computational complexity of our benchmarks and small datasets. Vector-vector addition is a linear time algorithm, while the back propagation weight adjustment and diagonal sparse matrix vector multiply kernels are both quadratic algorithms. In both cases, it results in a short run-time compared to compilation. Only

matrix multiply can significantly dwarf the compilation overhead. Fortunately, we believe TFJ will likely be used such that compiled kernels can be reused many times. Reuse of a compiled kernel will amortize the cost of calling the JIT compiler.

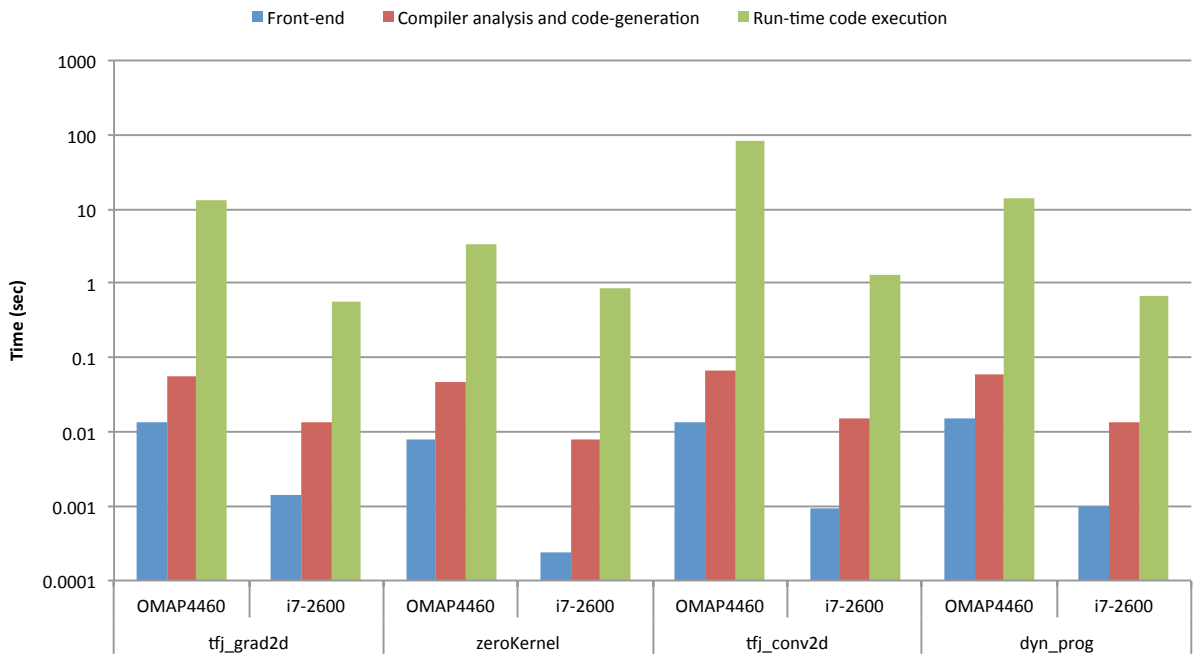


Figure 4.18: Run-time breakdown for the TFJ accelerated kernels used in content-aware image resizing.

The run-time breakdown for the small applications of Section 4.3 paints a different picture of TFJ's JIT compilation than the numerical kernels. As presented in Figures 4.18 and 4.19, the overhead of JIT compilation is effectively amortized as compilation is a small fraction of the overall program run-time.

Compared to other SEJITS-style frameworks, we believe TFJ has excellent compilation performance due to limited use of external compilers (there are no external compilers used on x86) and an efficient, low-level interface with the Python run-time. In contrast, the ASP framework [82] reports compile times greater than 20 seconds for a structured grid computation, while Copperhead compile times are upwards of 10 seconds [31] for their benchmarks. The slow compilation times of the other two frameworks is likely due to calling external compilers and the extensive use of Boost::Python [41] for interoperability with the Python C run-time. Calling external compilers requires slow file I/O; however, the use of Boost::Python results in large compilation units for even the simplest codes due to the extensive use of template metaprogramming.

As an example of the compilation unit size when using Boost::Python, consider the array-doubler example (Figure 4.20) included with the ASP framework [80]. The original source is 13 lines of Python. When compiled by the ASP framework into an intermediate C++

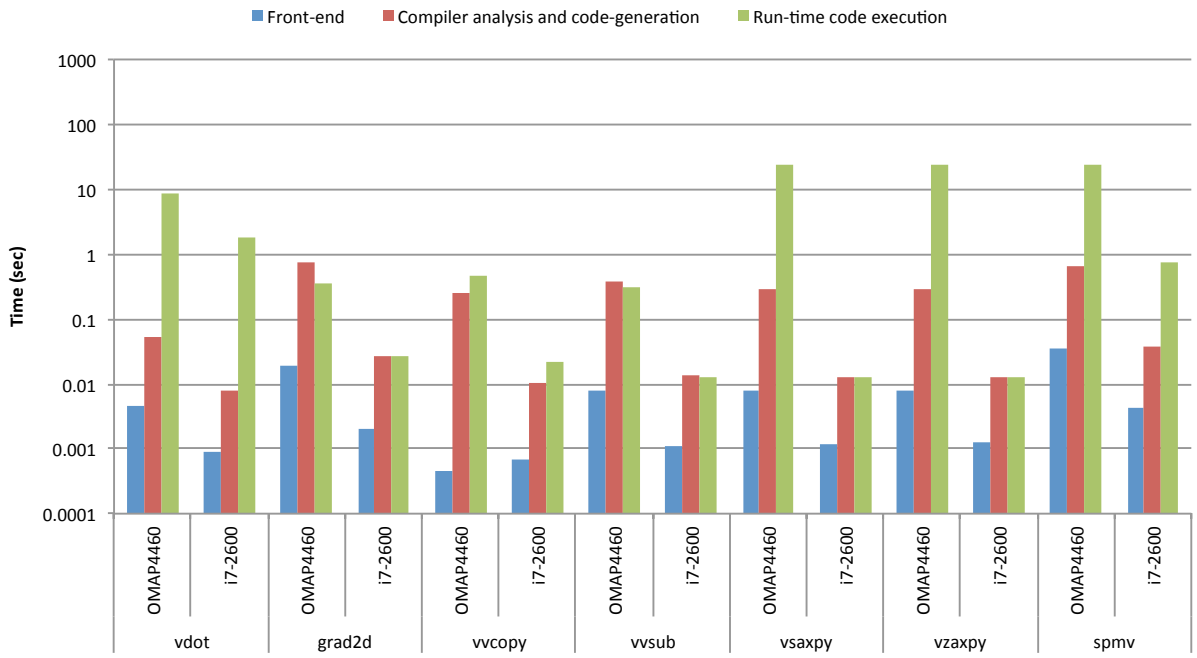


Figure 4.19: Run-time breakdown for the TFJ accelerated kernels used in optical flow.

```
class ArrayDoubler(object):
    def __init__(self):
        self.pure_python = True
    def double_using_template(self, arr):
        import asp.codegen.templating.template as template
        mytemplate = template.Template(filename="templates/
           double_template.mako", disable_unicode=True)
        rendered = mytemplate.render(num_items=len(arr))
        import asp.jit.asp_module as asp_module
        mod = asp_module.ASPModule()
        mod.add_function("double_in_c", rendered)
        return mod.double_in_c(arr)
    def double(self, arr):
        return map (lambda x: x*2, arr)
```

Figure 4.20: Array doubler implemented with ASP

representation, it is still a very managable 19 lines; however, after preprocessing the C++ source of the resulting postprocessed, intermediate representation has ballooned to 64828 lines of C++. This is due to extensive template metaprogramming in the Boost::Python framework.

In constrast, array doubler implemented using TFJ (Figure 4.21) is 4 lines of Python.

```
 @tfj
def array_doubler(y, a, n):
    for i in range(0,n):
        y[i] = 2.0 * a[i];
```

Figure 4.21: Array doubler implemented with TFJ

When we use our C++ backend, compilation results in 14 lines of intermediate representation C++. Preprocessing this C++ expands to just 3456 lines of code, a 19× reduction compared to the equivalent ASP postprocessed representation. In practice, TFJ's compilation process results in compile times under 100 milliseconds on x86 platforms with the LLVM MCJIT backend and less than two seconds on our relatively slow PandaBoard-ES platform using a GNU C++ backend.

4.6 Summary

In this chapter, we have evaluated four kernels and two applications with TFJ. We have also implemented a simple autotuner for matrix multiply and evaluated the overheads for runtime complication. A complete tabulation of our performance results for the four kernels and two applications is presented in Table 4.4. Our resutls show TFJ obtains similar performance to optimized C++ code on both ARM and x86 platforms.

Our prototype matrix multiply autotuner demonstrated the potential for implementing autotuners entirely in Python using TFJ. In addition, we achieve a maximum performance improvement of 1.53× and 1.45× using autotuning on our OMAP4460 and i7-2600 platforms, respectively. Our evaluation of runtime code generation showed that for the two applications presented in this chapter, the overheads of dynamically generating parallelized code at runtime is insignificant.

Finally, we must emphasize that while our TFJ results are within 60% to 80% of the performance of hand-tuned C++. the productivity benefits of our system are enormous. Hand-tuning our simple C++ kernels for correctness and performance took several hours, while tuning the whole applications of Section 4.3 took several weeks. In contrast, TFJ results achieved correct, high-performance results within 15 minutes of developer effort for kernels and within hours for applications. For individuals requiring higher performance, our results are a good starting point for further code optimization as TFJ emits readable C++ that can be manually tuned.

§	Kernel/Application	Device	C++ (Untuned)	C++ (Hand-tuned)	Python Libraries	Pure Python	TFJ
4.2.1	Vector-Vector Add	OMAP4460	81.8	**95.3**		0.4	56.6
	(*Mflops/sec*)	i7-2600	1073.3	**1090.4**	950.6	2.0	1039.2
4.2.2	Matrix Multiply	OMAP4460	40.4	**3268.6**	2292.7	<0.2	546.7
	(*Mflops/sec*)	i7-2600	317.0	**112882.3**	71570.2	1.4	39293.6
4.2.3	Diagonal SpMV	OMAP4460	130.1	175.3	114.0	<0.2	**336.2**
	(*Mflops/sec*)	i7-2600	2457.8	3052.6	1847.3	1.5	**4321.0**
4.2.4	Back propagation	OMAP4460	100.7	**101.4**	N/A	0.07	41.2
	(*Mflops/sec*)	i7-2600	668.3	2568.3	N/A	0.41	**3147.5**
4.3.1	Seam Carving	OMAP4460	205.4	**68.5**	18873.1	>86400.0	81.2
	(*Runtime in sec*)	i7-2600	20.4	**4.6**	2238.6	34646.8	7.1
4.3.2	Optical Flow	OMAP4460	72.8	**52.5**	436.9	N/A	83.4
	(*Runtime in sec*)	i7-2600	4.4	**1.9**	29.0	24146.7	4.5

Table 4.4: Performance results for the 4 kernels and 2 applications. **Bold** numbers indicate best results. For the 4 kernels, larger Mflops/sec values indicate faster implementations. Application performance is reported in seconds; therefore, shorter runtimes reflect higher peformance. We have marked categories N/A if we could not find a Python library that implements a given benchmark. On our ARM platform, several benchmarks did not complete in under 24 hours when executed as Python loop-nests.

Chapter 5

Three Fingered Jack: Hardware Approaches

In this chapter, we describe the design and implementation of Three Fingered Jack's high-level hardware synthesis (HLS) engine. As mentioned in Chapter 2 of this thesis, the work presented in this disseration addresses the challenges associated with the diversity of potential implementation platforms. To target both FPGA and ASIC platforms, we require a flow to generate register-transfer descriptions of hardware. This is in contrast to the approach for generating software for programmable processors described in Chapter 3.

We describe the micro-architecture of our processing cluster in Section 5.1 and the implementation and algorithms of our HLS system in Section 5.2. We evaluate the results of automatically generated hardware for several micro-benchmarks in Section 5.3. Section 5.4 concludes this chapter with a case study in tuning a hardware implementation of matrix multiply.

Our approach to high-level hardware synthesis [126, 127] is similar to traditional approaches [103] in many respects; however, our approach focuses on exploiting vector parallelism (data parallelism) uncovered by the analysis performed in our compiler front-end. We focus on vector parallelism instead of instruction-level parallelism or thread-level parallelism because vector instruction semantics are very beneficial when automatically generating a high-performance memory subsystem. In particular, when a loop is vectorized, we guarantee it carries no dependence; therefore, each iteration of the loop proceeds in parallel. This allows for many inflight memory operations as they are guaranteed to be independent.

We have also placed considerable effort in implementing efficient handling of memory operations in our HLS flow. We support stalling memory accesses, which allows our HLS flow to use machine structures such as cache memories. In contrast, systems such as C-To-Verilog [134], require memories with a fixed access latency. Supporting multiple outstanding memory requests is a particular focus in our system. We support multiple requests inflight to amortize the latency of main memory [58] by exploiting the inherent memory-level parallelism present in vectorized codes. Figure 5.2 demonstrates the benefits of supporting multiple inflight memory requests in a high-level hardware synthesis flow. Performance is improved by 3.76× to 4.6× for main memory latencies between 1 and 100 cycles compared

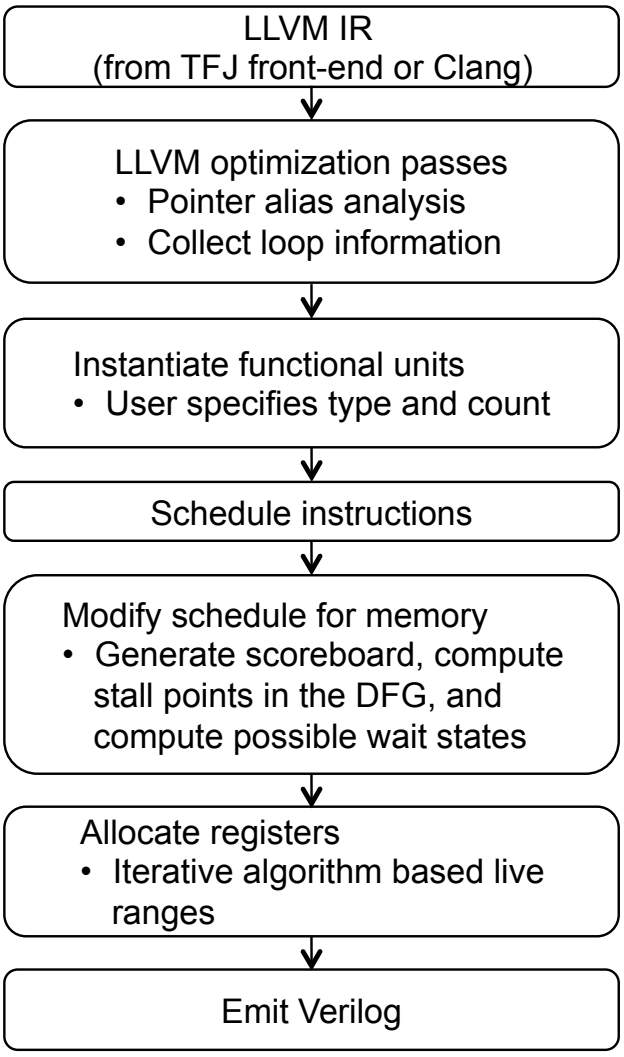


Figure 5.1: Our high-level hardware synthesis flow

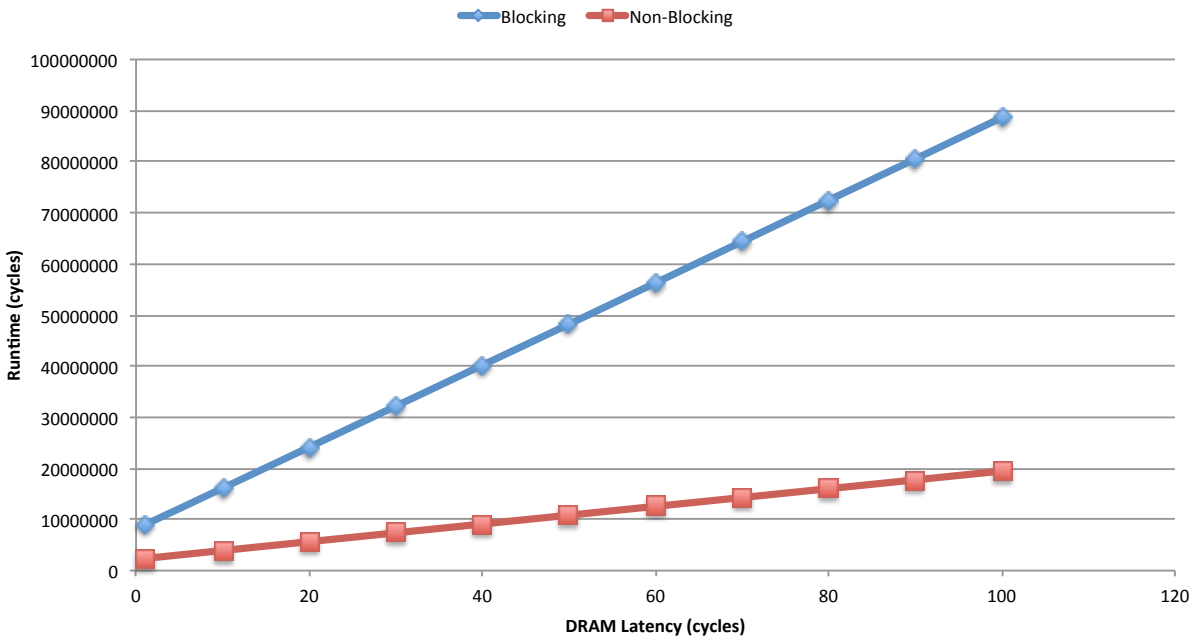


Figure 5.2: The benefits of supporting multiple outstanding memory requests when applied to a 64×64 matrix multiply with a variety of main memory latencies. Both configurations use 2 processing engines. Each processing engine has its own 64 word L1 cache and share a 1024 word L2 cache. The configuration labeled "non-blocking" supports many concurrent memory requests to the L2 cache. The configuration supporting multiple inflight requests achieves greater than 4× better performance than the blocking configuration.

to a serializing approach.

Another possible way of extracting memory-level parallelism to tolerate memory latency is fine-grained multithreading. This approach has recently been popularized on programmable graphics processors such as Nvidia's Tesla [98] and Fermi [111] series processors.

5.1 The Architecture of TFJ's Processing Engine Clusters

Building custom hardware with TFJ relies on both high-level hardware synthesis techniques to generate processing elements for a specific computation and a predesigned system hardware template. The predesigned system template connects the custom generated processing elements to a memory hierarchy and performs any required synchronization between the processing elements.

An example of a predesigned system hardware template is shown in Figure 5.3. In this figure, the processing engines (PEs) enclosed in the orange rounded rectangle are automat-

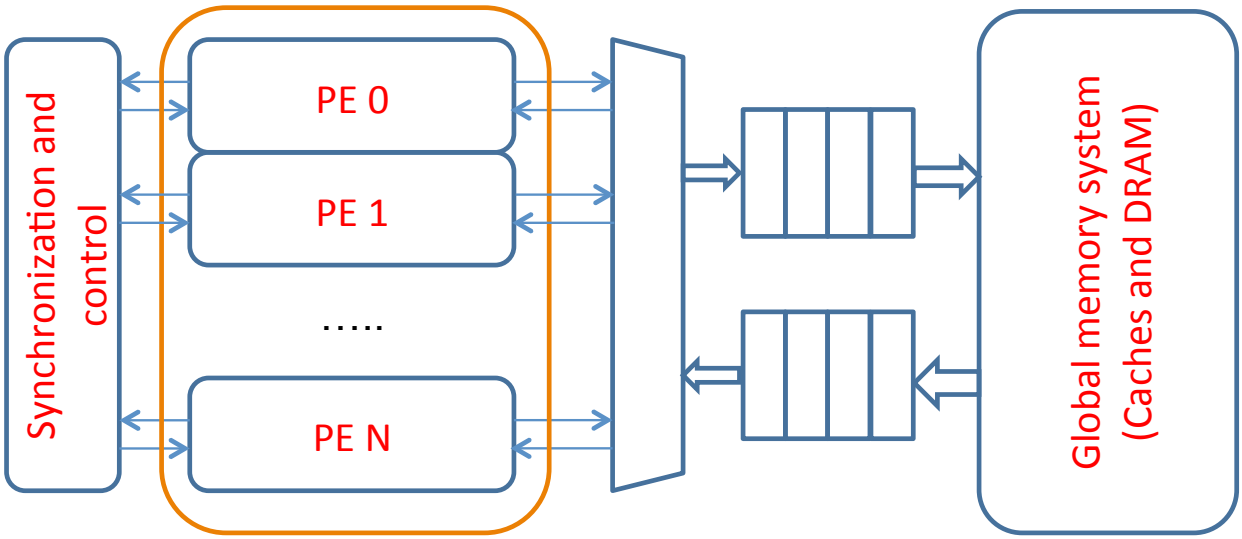


Figure 5.3: The initial architecture of our HLS processing cluster. The processing engines enclosed in orange are automatically generated by our HLS framework.

ically generated by our HLS flow for a specific computation. The rest of the system is provided as parameterized Verilog. The user selects the number of processing engines he or she would like to use for a given system implementation. When the user changes the number of processing engines, the system hardware template is customized for that specific number of processing engines. This includes updating the synchronization and control logic along with connecting the processing engines to the memory hierarchy. The HLS flow generates processing engines with a predefined Verilog interface to make integration into the system hardware template straightforward. The user also selects the size of L1 and L2 caches. Customization of the system hardware template is enabled by heavily parameterized Verilog RTL. While we currently require the designer to select the number of processing elements and size of the caches, it possible to use existing design space exploration techniques [63] to automate this procedure. The details of those approaches however, are outside the scope of this disseration.

We have built two versions of the system hardware template. The details of which are described in the rest of this Section. In our first implementation of TFJ's HLS engine, the flow generated processing engines with blocking memory operations. This simplified the interaction with variable latency cached memory. In this initial implementation, each processing engine had a blocking memory interface. This prevented a single processing engine from saturating the simple memory subsystem. Therefore, in our initial approach, a cluster of processing engines shared a global memory interface as an individual processing element required relatively little memory bandwidth. This system used a direct-mapped, write-back cache with 128-byte cache-lines to back a cluster of processing engines. In the blocking approach, each processing engine can have 1 outstanding memory request, and processing engines arbitrate for the FIFO connecting the processing engines to cache and

global memory.

Our initial approach to HLS generated processing engines with relatively poor utilization of the memory subsystem due to blocking memory accesses. To remedy our memory utilization problem, we modified our HLS scheduling algorithm to generate non-blocking memory requests. We have not found any other HLS systems that support multiple memory instructions inflight to a memory subsystem with variable latency (caches). When memory access have a constant latency, supporting multiple inflight memory requests is no more difficult than any other pipelined operation given sufficient memory address aliasing information. Systems such as C-To-Verilog [134] do support multiple inflight requests but only to memories with a fixed access latency. Other systems, such as Conservation Cores [137], support memory accesses with variable latency, but they do not support simultaneous memory requests from a single processing engine.

As we extract data-parallelism with our automatically parallelizing front-end, we are able to uncover significant amounts of *memory-level parallelism* (MLP) [58] from a program. To exploit the MLP discovered by our parallelizing front-end required the implementation of a new memory scheduling algorithm, described in Section 5.2.2, and modifying the architecture of our processing engine cluster to support multiple inflight memory operations.

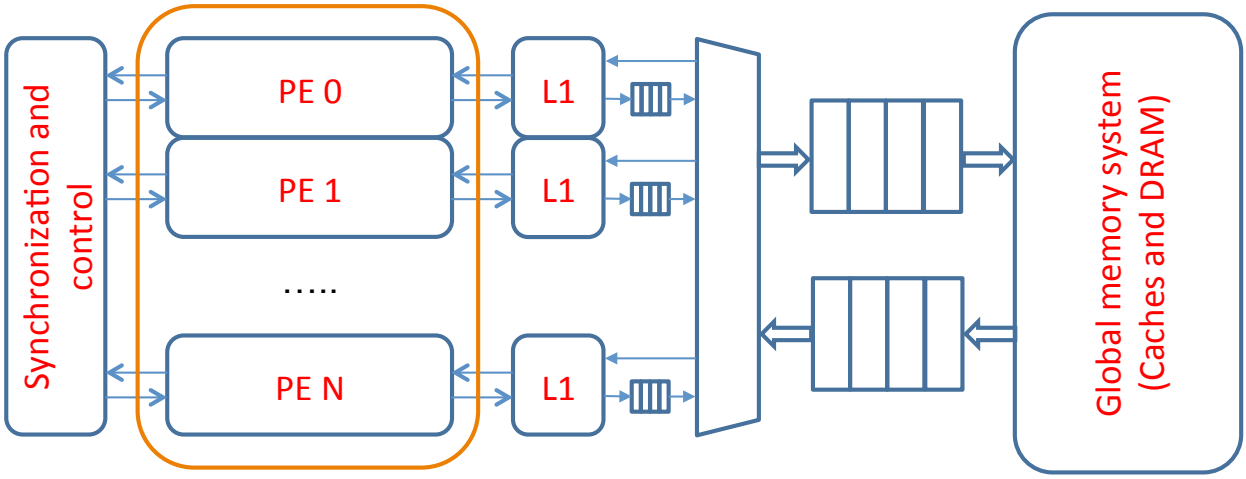


Figure 5.4: The final architecture of our HLS processing cluster

The architecture of the processing cluster after additions to support simultaneous inflight memory requests is shown in Figure 5.4. This architecture has several changes. As described in detail in Section 5.2.2, memory instructions are tagged and the processing engines stall only when they attempt to consume an operand that is still inflight. To support multiple memory instructions inflight, we added per processing engine queues before global memory subsystem arbitration. These queues allow the processing engines to inject memory operations as long as space is available in the queue. After arbitration, memory requests are placed in another queue to await access to the global memory subsystem. When operands hit in the L1 cache, the system template can deliver 1 word per cycle with a latency of 4 cycles. We size our queues following Little's Law [99] which means we have enough storage

in our queues to hold at least 4 requests per processing element. In practice, we suffer cache misses and slightly long queues improve performance. Eight entries per processing element queues combined with a 16 entry global arbitration queue is a reasonable trade off between resource allocation and system performance.

Similar to the blocking implementation of the system template, the global memory system is backed by an L2 cache with a direct-mapped, write-back cache with 128-byte cache lines. We also added small per-processing engine write-through L1 caches to capture temporal locality if it exists in the program. The L1 caches are direct-mapped and have a single cycle access latency. We only support L1 caches if programs have been compiled with the TFJ front-end. The dependence analysis performed by the front-end is used to insert flushes of the L1 caches when needed. The L1 caches are completely optional and can be disabled by the system user.

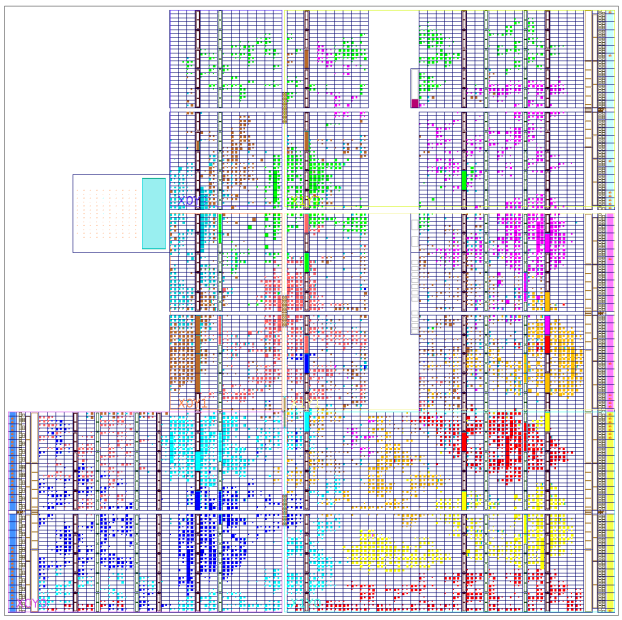


Figure 5.5: Color conversion (source shown in Figure 5.12) system with 8 processing engines. The 8 processing engines are highlighted in yellow, green, orange, dark blue, burnt orange, red, light blue, and magenta. The memory subsystem is highlighted brown. Figure generated using Xilinx PlanAhead 14.1 with a Zynq ZC702 FPGA.

5.2 Our High-Level Hardware Synthesis Flow

The structure of our HLS flow is shown in Figure 5.1. Our HLS back-end uses the LLVM [94] framework because it enables straightforward code optimization and machine code generation passes. In addition, LLVM includes a vast repertoire of traditional compiler transformations that we apply to the intermediate representation generated by our

vectorizing front-end. In particular, we apply dead-code elimination, loop-invariant code motion, and peephole optimizations to optimize the IR generated by our front-end. As described in Chapter 3, we also exploit the ability to generate machine code for our x86 desktop CPUs in LLVM framework.

To generate processing engines, we map LLVM IR to Verilog RTL. Although our RTL generator is similar in principle to C-To-Verilog [134] and other HLS systems that use LLVM for RTL generation, the architecture of the processing engine cluster requires a slightly different set of features than those provide. Above all, we need stalling memory support because contention for the shared memory interface and cache access introduces non-deterministic memory access times. We did consider manual modification of the RTL generated by an existing system, but this approach would have defeated our goal of automatically generating hardware implementations from Python. We also briefly considered using an automatic clock gating scheme with C-To-Verilog to support stalling memory operations, but we decided it was simpler to implement our own tool.

Our RTL generation system is built on many conventional HLS algorithms [103]; however, we have placed considerable focus on efficient support of memory operations. We now provide a brief overview of our HLS system. The user specifies the mix of functional units, latency, and support for pipelined operation. Functional units that can not be easily described in small number of lines of behavioral Verilog are instantiated from a hardware component library (Section 5.2.4). The system then schedules the datapath using a list scheduling formulation; however, we have also experimented with integer programming formulations [138, 73] of scheduling algorithms. More details about datapath scheduling are presented in Section 5.2.1, while register allocation is described in Section 5.2.3. Our approach to efficient memory support is described in Section 5.2.2.

5.2.1 Datapath Scheduling

We schedule LLVM IR to generate both a datapath and control finite-state machinebecause the LLVM IR encapsulates both the control flow and data flow graphs. An example of LLVM IR for vector-vector addition is shown in Figure 5.6. Within the LLVM IR, each basic block (straight-line code segments) has edges to potential successors. The edges between basic blocks represent control flow. Within basic blocks, dependences are explicit and unique as LLVM uses a static single assignment IR representation [44]. In the static single assignment intermediate representation variables are assigned exactly once. This in turn makes use-def chains explicit and allows for a simple straight-line instruction scheduling technique. We use list scheduling to schedule within basic blocks. In addition, we also have an integer programming scheduler. In practice, we find the integer programming based approach gives slightly better results at the expense of a much longer compile time.

As data dependence is explicit in a static single assignment representation, we can easily topologically sort instructions within a basic block via dependence edges. We can then greedily schedule an instruction when all of the instructions on which it depends have completed execution. The basic approach of topologically sorting instructions and issuing instructions when all dependences are resolved is known as *list scheduling*. We convert the

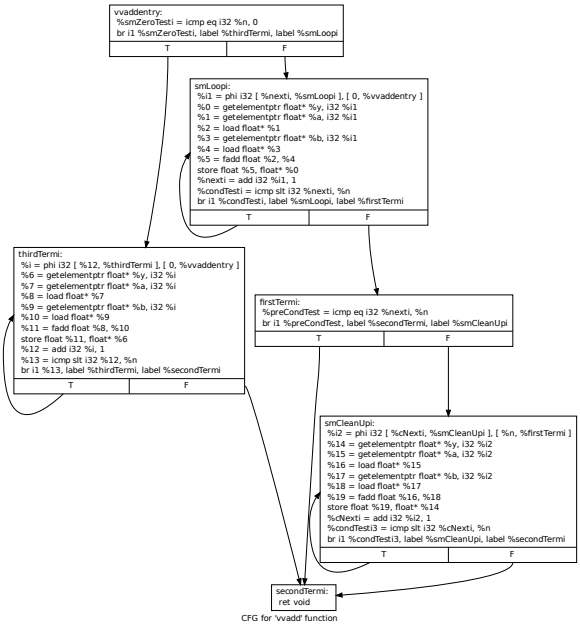


Figure 5.6: LLVM IR representation of the vector-vector addition kernel without autovectorization. This kernel has 6 basic blocks. The largest basic block has 11 instructions.

schedule generated by our scheduling algorithm into a control finite state machine. This control finite state machine sequences the datapath of processing elements. In the worse case, our state machine will have approximately as many states as the longest instruction latency multiplied by the number of instructions in the kernel under consideration. To efficiently handle variable latency memory instructions, we perform a post pass on the control FSM that better handles memory instructions. We describe this is more detail in Section 5.2.2.

We attempt to issue as many instructions per cycle as hardware and dataflow constraints allow. When control flow requires a transition to a different basic block, we require that the pipeline be completely drained to avoid tracking pipeline state across control flow edges. While this restriction appears onerous, if we generate large basic blocks, the pipeline startup and drain overhead can be amortized. Control flow has historically been a challenge for superscalar processors [54] and our high-level synthesis flow faces the same problem. The problem for our system is to generate large enough basic blocks such that the pipeline startup and drain costs will not dominate execution.

List scheduling [61] works well within a basic block; however, its utility is limited across basic blocks. Past approaches to create large basic blocks include trace scheduling [53], superblock formulation [74], hyperblock formulation [101], and treegions [66]. All of these approaches attempt to expand the number of instructions (regions) in which a straight line code scheduling technique, such as list scheduling, can be applied. Many of the approaches listed previously speculate on the control flow through a region of code in order to create a large basic block. These approaches rely on accurate branch profiling in order to generate

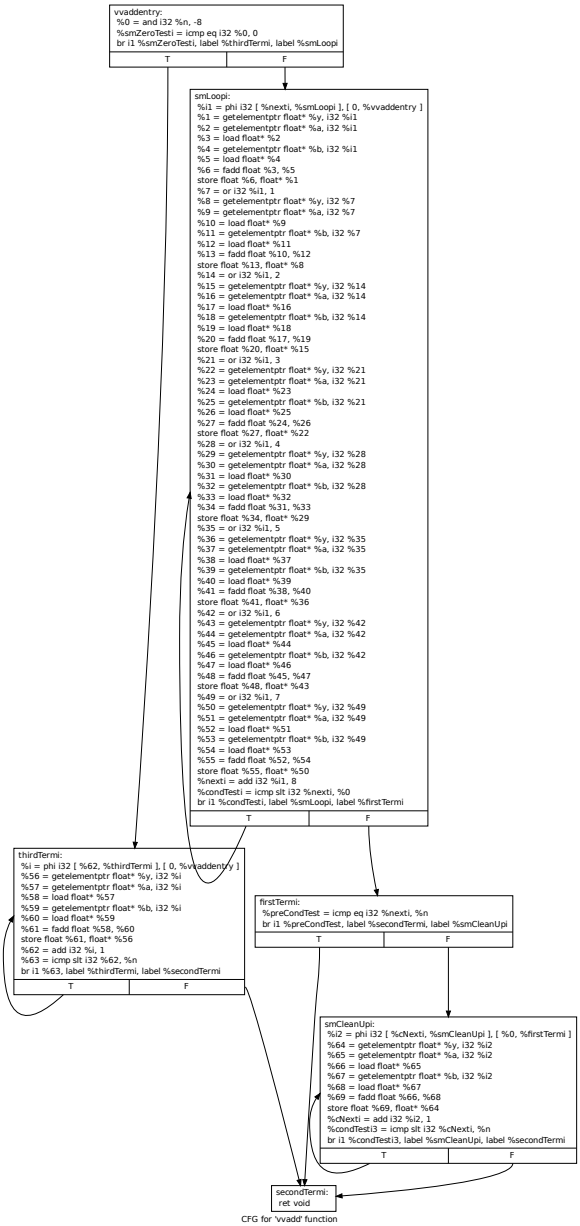


Figure 5.7: LLVM IR representation of the vector-vector addition kernel with autovectorization for a data-parallel operation length of 8. In contrast to Figure 5.6, the largest basic block has 67 instructions.

effective traces. In contrast, TFJ converts data-parallelism found by the autovectorizing front end into instruction-level parallelism to create very large basic blocks. TFJ converts data-parallelism into instruction-level parallelism by unrolling vector operations into a sequence of scalar operations. The large basic blocks can be efficiently scheduled using list

scheduling. Figure 5.7 presents the control flow graph when we enable the autovectorizer with the same Python kernel used to generate the LLVM IR shown in Figure 5.6. TFJ's HLS scheduler predicts the non-vectorized implementation will execute the 11 instructions in the inner basic block in 24 cycles. In contrast, with vectorization, the 67 instructions in the inner basic block will execute in 38 cycles. Using vectorization, the HLS flow improved the instructions per clock (IPC) by nearly a factor of 4×.

```
//FPGA: Type, Count, Latency
ALU,4,1
MUL,2,5
ADDR,1,1
MEM,1,10
FADD,1,5
FMUL,1,6
FCMP,1,1
RET,1,1
PHI,1,1
BRANCH,1,1
SELECT,1,1
NAOP,2,0
```

(a) Datapath configuration file for FPGAs. Notice integer functional units match latency of DSP48 for better resource utilization.

```
//ASIC: Type, Count, Latency
ALU,2,1
MUL,1,2
ADDR,1,1
MEM,1,6
FADD,1,5
FMUL,1,6
FCMP,1,1
RET,1,1
PHI,1,1
BRANCH,1,1
SELECT,1,1
NAOP,2,0
```

(b) Datapath configuration file for ASICs. Functional units have lower latency than the FPGA equivalent due to reduced propagation delay in ASIC fabric.

Figure 5.8: Datapath configuration files for FPGA and ASIC

As shown in Figures 5.8a and 5.8b, the designer selects the number of functional units in the datapath. All functional units have a 1 cycle initiation interval (fully pipelined); however, the user is also allowed to select the latency (in cycles) of all units except for the single-precision adder and multiplier. Our HLS tool generates behavioral RTL descriptions for integer operations and can easily vary the latency to match ASIC or FPGA primitives. For example, Xilinx FPGAs have digital signal processing (DSP48) blocks with hard implementations of multipliers and adders. Using DSP48s results in higher throughput and lower device utilization than implementing the same function in the FPGA fabric. For maximum efficiency, our tool must generate functional units that match the underlying DSP48 primitives. As DSP48 latencies vary among different FPGA families, we require the user to manually input latencies for the device he or she is targeting. Our floating-point units are manually implemented in a structural implementation style. As a result, they are not easily amenable to variable latency operation. Our tool could easily be adapted to a more flexible floating-point implementation. We also tend to use more functional units in FPGA implementations than multiplexing a single functional unit because wide multiplexers are relatively slow in FPGA fabric [133].

5.2.2 Generation of Non-blocking Memory Operations

Efficiently supporting memory-level parallelism (MLP) is at the heart of TFJ's HLS flow. Given that memory operations often have high latency (10 to 100 cycles), as shown in Figure 5.2, it is critical to support multiple inflight memory operations to tolerate memory latency [88]. Our approach to this task is based on the conversion of data-level parallelism found by our vectorizing front-end into memory-level parallelism. As described in Section 5.2.1, using vectorization algorithms to generate scalar operations produces large basic blocks with many independent operations. Within the large basic blocks, there is significant potential for memory-level parallelism; however, the high-level synthesis flow must generate the proper control structures in order to support multiple inflight memory operations.

Conventional high-level synthesis approaches do not attempt to extract memory-level parallelism from cache-based memories due to variable memory latency. In classical HLS approaches [103], a control finite state machine expects all operations to complete in a fixed number of cycles. When coupled with a memory structure, such as an SRAM scratchpad, classical HLS approaches can support memory-level parallelism. However, for the class of kernels and applications addressed in this thesis, the working sets considered are far too large to fit in a scratchpad. A simple approach to supporting variable memory latency is to serialize the processing element when a memory operation occurs. In this approach, the processing element pipeline must be drained before a memory operation occurs, and no new operations are issued until the memory operation completes. If memory accesses are only a small number of cycles (1 or 2 cycles), this approach is likely reasonable; however, when memory has a large access latency, this approach will be highly inefficient.

In order to allow multiple inflight memory operations with variable access latency memory, a more advanced approach is required. Our approach is conceptually simple: we schedule the kernel for expected memory latency and then add additional states to the control finite state machine. These extra states stall the pipeline and perform operand clean up if our latency assumptions are incorrect. Our algorithm for supporting multiple inflight memory instructions is shown in Figure 5.9.

Our approach stalls the processing engine when it attempts to consume a memory operation that is currently inflight due to longer than expected memory latency. This approach potentially prevents useful work from executing during a memory stall as instructions with satisfied operand dependences can not execute during this period. In contrast, modern out-of-order microprocessors [128] would be able to continue execution in the same situation due to the large number of resources dedicated to dynamic instruction scheduling. We believe our approach is a reasonable trade-off between hardware resources and system performance; however, we have not implemented or evaluated other solutions.

We show the impact of our memory-level parallelism enhancing algorithm in Figure 5.10 for a single-precision floating-point 160×160 matrix multiply. Using the naive triply-nested matrix-multiply described in Section 5.4, we measured memory-level parallelism for a variety of different system configurations. We experimented with different lengths of data-parallel operations discovered by the compiler front-end, specifically we explore data-parallel opertions from a single entry to operations of length 16. On the memory subsystem side, we use two different L2 caches sizes and main memory latency from zero cycles (instant reload)

```
Algorithm: tfj_mkmlp
Input:   I (Instructions in kernel)
Output:  S_mlp (Instruction schedule with support for memory-level parallelism)
1  List schedule i ∈ I assuming average case memory latency
1      to generate baseline control finite state machine. Build
1      data-structure F to track instructions issued in each state
2  Create a hardware scoreboard to keep track of inflight memory operations
3  Mark every memory operation with a unique tag.
4  for i ∈ I
5      If i consumes an operand produced by a load from memory
6          Determine predecessor states for i using F
7          Add memory scoreboard test in each predecessor state
7              and additional states to handle instructions retiring
7              while the PE is stalled. The total number of wait states
7              is equal to the longest latency of any inflight instruction
7              when the stall occurs.
```

Figure 5.9: A high-level sketch of the algorithm used by TFJ to support memory-level parallelism. This algorithm allows for memory-level parallelism when memory operations execute in the expected number of cycles. When memory operations take longer than expected, this algorithm adds additional wait states into the control finite state machine to handle stalling the processing element.

to 100 cycles. The results clearly show the effectiveness of our approach. Our configurations that use data-parallel operations of length 16 have greater than 10× more memory-level parallelism than the scalar equivalent. The increased memory-level parallelism also has a significant impact on performance, as shown in the case study of Section 5.4.1. As shown in Figure 5.22, with a 256 kByte L2, performance increases from 5 MFlops/sec to nearly 40 MFlops/sec using the scheduling techniques described in this section.

5.2.3 Register Binding

After scheduling our datapath, we need to bind virtual registers in the LLVM IR to physical registers (flip-flops) to construct a circuit. One approach to binding virtual registers to physical registers is to allocate a physical register for each virtual one. While this naive approach to register binding works, it unnecessarily wastes hardware resources and results in a very large custom compute accelerator.

In contrast, our high-level synthesis uses a register binding algorithm based on live ranges and a list of free registers. Our algorithm is conceptually simple. We first compute the largest number of live registers at any point in the program. After computing the total

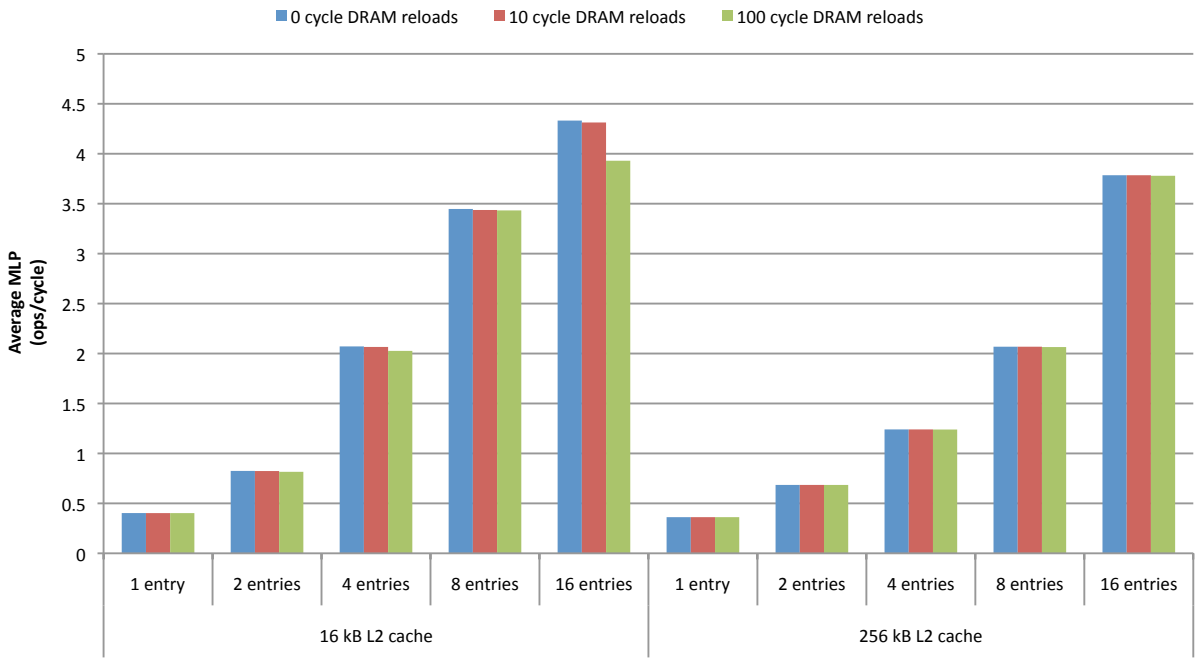


Figure 5.10: 160×160 matrix-multiply memory-level parallelism for a data-parallel operation lengths from 1 entry to 16 entries and three different cache reload latencies. All results generated for a single custom processing element with no L1 cache. Results generated using logic simulation with ModelSim SE 10.0d.

number of physical registers needed, we generate a queue with physical register identifiers. The size of the queue is equal to the largest number of live registers at any point in the kernel. We then use LLVM's dataflow algorithm framework to compute live ranges for each virtual register. Over the live range of a variable, it is allocated to a physical register. This is done by popping a register identifier off the physical register queue. When a variable is killed, the corresponding physical register identifier is pushed back on the physical register queue. Due to nuances of our pipelined memory operations, we might need more registers than predicted by dataflow analysis. When this situation occurs, we add a small number of additional register identifiers to the physical register queue and retry the binding process. We repeat this process until the system finds a successful binding. In practice, it usually takes two or three iterations to find a valid solution.

As each time a register is reused multiplexers are required to forward the operand to a new functional, one potential downside of our simple algorithm is wide multiplexers. While they are less of an issue in ASIC technologies, FPGAs poorly support wide multiplexers. We have not observed register multiplexing in the critical path of any of our benchmark kernels; however, we could modify our binding algorithm to detect when a register becomes heavily multiplexed and then add additional registers to alleviate multiplexer delay.

While our binding algorithm is simple, we find it works well in practice. Table 5.1 presents results for the naive and our optimized algorithms for the benchmarks presented in

	Vector-vector addition	Color conversion	Matrix multiply	Gaussian mixture model
Naive approach	80	25	120	148
Our approach	30	15	45	47

Table 5.1: A comparison of the number of 32-bit registers required by the two different binding algorithms.

Section 5.3. Our algorithm reduces register requirements for the benchmark kernels under consideration between 1.6-3.15×.

5.2.4 A Library of Hardware Components

As described in Section 5.1, the architecture of TFJ-based hardware solutions include both custom processing engines generated by HLS techniques (Section 5.2) and a system micro-architectural template (Section 5.1). Both components of TFJ's hardware backend require a library of hardware components. While the HLS flow is able to generate inline behavioral descriptions of integer arithmetic operations, floating-point components are instantiated as hand-coded Verilog modules. Likewise, the system template (shown in Figure 5.4) is built from hand-coded modules that comprise the memory subsystem interconnect and caches. Parameters such as cache size or memory subsystem queue depth can be modified by the end user.

We evaluate the FPGA resource requirements for components of the hardware library by connecting them to the ARM processors on the Zynq platform using the AXI interface. We describe our methodology for interfacing with the ARM processors on the Zynq platform in Section 5.3.2. By directly connecting components to the AXI interface, we can both verify correctness of individual components and measure resource requirements for a specific component. As logic optimization can potentially modify components in the hardware library to better meet timing or resource constraints in a real system, the resource statistics presented below serve as an approximation to the resource requirements when the component is used in a real system.

Floating-point Units

TFJ's hardware library includes single-precision addition and multiply functional units. These units are fully pipelined. As presented in Table 5.2, the adder has a latency of 4 cycles, while the multiplier has a latency of 5 cycles. If required, these designs could easily be further pipelined. Both units are based on Parhami's designs [117].

Hardware component	Latency	LUTs	DSP48s
Single-precision addition	4 cycles	709	0
Single-precision multiplication	5 cycles	382	2

Table 5.2: FPGA statistics for our floating-point units on the Xilinx Zynq XC702 FPGA

Systems that use floating-point operands routinely need to deal with exceptions. As the processing elements generated by TFJ use hardwired finite-state based control, adding support for exceptions would be impractical. Therefore, TFJ requires the user to ensure that his or her design will generate valid results over the range of possible floating-point inputs.

Caches

The hardware solutions generated by TFJ use up to two levels of caching to back main memory. As shown in Figure 5.4, each processing element has a private L1 cache while all processing elements share a single L2 cache. We have chosen to implement only direct-mapped caches due to the ease of efficient implementation on FPGA platforms. Caches with higher associativity would require wide multiplexers that are inefficient in FPGA fabric.

The private L1 cache design, presented in Table 5.3, is entirely optional. When the L1 cache is used, it has a single-cycle access latency and uses a direct-mapped, write-through scheme. Using a write-through scheme makes synchronization straightforward. When synchronization is required between processing elements (as dictated by dependence analysis performed by our compiler front-end), the caches are kept coherent by clearing valid bits on the L1 caches. This forces the processing elements to reload their L1 caches with synchronized data from the shared L2.

Some kernels with a significant amount of operand reuse, such as matrix multiply, greatly benefit from from the L1 cache. When a code does not have operand reuse, the L1 cache is easily removed from the design to reduce resource requirements.

Size	LUTs	36 kBit RAMs	18 kBit RAMs
64 Bytes	587	0	0
128 Bytes	430	0	1
256 Bytes	489	0	1
512 Bytes	577	0	1
1 kBytes	745	0	1
2 kBytes	1105	0	1
4 kBytes	1880	1	0
8 kBytes	3509	1	1
16 kBytes	6474	2	1

Table 5.3: FPGA statistics for our direct-mapped, write-through L1 caches on the Xilinx Zynq XC702 FPGA

We have two different L2 cache designs that are presented in Table 5.4. Both designs support pipelined cache accesses for high-throughput cache access; however, only our more "optimized" design supports a "hit-under-miss" scheme. Our implementation of "hit-under-miss" allows the L2 cache to continue serving requests as long as there are less than two active L2 cache misses. As shown in Table 5.4, adding support for a "hit-under-miss" scheme requires approximately twice as many LUTs as the equivalently sized "baseline" L2 cache.

As our L2 caches additionally serve as the point of ordering among the processing elements, adding support for advanced features, such atomic memory operations, is reasonably straightforward.The first design, our "baseline" design, has enough additional hardware to support atomic memory operations. The dependence-driven approach to HLS described in this chapter does not use atomic memory operations; however, TFJ has the ability to lower a larger subset of Python to LLVM IR (Section 3.2), which includes support for atomic memory operations.

	Size	Atomics	Hit-under-miss	LUTs	36 kBit RAMs	18 kBit RAMs
Baseline	1 kByte	Yes	No	1104	0	4
	2 kBytes	Yes	No	1087	0	5
	4 kBytes	Yes	No	1095	0	5
	8 kBytes	Yes	No	1063	5	0
	16 kBytes	Yes	No	971	5	2
	32 kBytes	Yes	No	1062	9	3
	64 kBytes	Yes	No	1027	18	2
	128 kBytes	Yes	No	1123	36	3
	256 kBytes	Yes	No	977	72	2
Optimized	1 kByte	No	Yes	1916	0	4
	2 kBytes	No	Yes	1602	0	5
	4 kBytes	No	Yes	1708	0	5
	8 kBytes	No	Yes	2090	1	4
	16 kBytes	No	Yes	2006	5	3
	32 kBytes	No	Yes	1993	9	4
	64 kBytes	No	Yes	2041	18	3
	128 kBytes	No	Yes	1880	36	4
	256 kBytes	No	Yes	1916	72	3

Table 5.4: FPGA statistics for our direct-mapped, write-back L2 caches on the Xilinx Zynq XC702 FPGA. All L2 caches use 16 byte cache lines.

System Template Glue Logic

The hardware system template is comprised of many common circuit elements such as synchronous FIFOs and arbiters. Because many aspects are dependent upon specific parameters fixed by a given system, it is not possible to report tabulated resource statistics. Many of these structures were originally designed and written as part of the ELM project [45].

5.3 Evaluation of Numerical Kernels on a FPGA

We evaluate the automatically generated hardware synthesized by our system with four numerical kernels on an field-programmable gate array (FPGA). We use a FPGA platform for several reasons. First, the design automation tools used with FPGAs accept the same Verilog RTL as the ASIC tool flow does. This allows for verification of the RTL generated

by TFJ without long running software simulations. Second, FPGA implementation allows for a realistic evaluation of the area (using lookup tables) and cycle time. However, as we are using FPGAs for evaluation, the cycle time of our solutions will be slower than a design implemented in a standard-cell ASIC flow due to the nature of the programmable interconnect and lookup table-based logic implementation. Third, it is a cost-effective solution for prototyping hardware designs as FPGAs are inexpensive, particularly when compared to the costs of designing and implementing an ASIC.

To fairly evaluate our system against another programmable substrate, we use a soft-core processor. A soft-core processor is conventional microprocessor pipeline implemented on a FPGA platform. We use a soft-core processor because it allows for a comparison of the performance (run time) and area in the same implementation technology between two design methodologies: TFJ generated hardware and a traditional C-based software development flow. Our evaluation approach oblivates the need to manually generate custom processing blocks in a hardware description language. The details of soft-core evaluation methodology are described in Section 5.3.2.

The benchmarks, presented in full in Section 5.3.1, are representative of numerical operations performed in many multimedia codes. The codes used for evaluation of our system are all data-parallel codes. As our system is designed to handle regular dense loop-nests, it is a natural fit for data parallel applications. Finally, the size of the data manipulated by each benchmark is implied in the loop-bounds.

5.3.1 Overview of Numerical Kernels

Vector-vector addition

A vector-vector addition is a canonical data-parallel benchmark. In addition, it is bandwidth-bound. Our implementation is shown in Figure 5.11.

```
for i in range(0,131072):
 c[i]=a[i]+b[i]
```

Figure 5.11: Vector-vector addition used for HW evaluation

Color conversion

We evaluate a simple color space conversion benchmark for a 128x128 pixel image. Color conversion can be expressed as a 3x3 matrix-transform applied to each pixel in an image. As the color conversion matrix is reused for each pixel, this benchmark has better memory reuse than vector-vector addition does. Our implementation is shown in Figure 5.12.

Matrix multiply

Matrix-matrix multiply is a widely used kernel in dense linear algebra libraries. It serves as the workhorse for many higher-level algorithms such as solving a system of linear equa-

```
for p in range(0,16384):
 for i in range(0,3):
  for j in range(0,3):
   img_out[p][i]=img_out[p][i]+img_in[p][j]*mat[i][j]
```

Figure 5.12: Color conversion kernel used for HW evaluation

tions, and it stresses the compute capability of most devices. Our matrix-matrix multiply is expressed as a triply nested loop and shown in Figure 5.13.

```
for i in range(0,32):
 for j in range(0,32):
  for k in range(0,32):
   c[i][j]=c[i]j]+a[i][k]*b[k][j]
```

Figure 5.13: Matrix multiply kernel used for HW evaluation

Gaussian mixture model evaluation

Modern speech recognition systems model the probability of a sound occurrence using a mixture of multivariate Gaussian distributions. It is common to use a mixture of 16 39-dimensional Gaussians per speech sound (phone). As noted in previous work [76], Gaussian mixture model evaluation accounts for greater than 50% of the run-time in the Sphinx3 system. The code used to evaluate a 3006 phone Gaussian mixture model is shown in Figure 5.14.

```
for i in range(0,3006):
 for m in range(0,16):
  for f in range(0,39):
   LogP[i][m]=LogP[i][m]+(In[f]-M[i][f][m])*(In[f]-M[i][f][m])*(V[i][f][m])
```

Figure 5.14: Gaussian mixture model evaluation kernel used for HW evaluation

5.3.2 Evaluation Setup

For the evaluation of our automatically processing engines, we used a Xilinx Zynq XC7z020-1CLG484 FPGA on a Digilent ZedBoard (Figure 5.16). As shown in Figure 5.15, the Zynq series of FPGAs includes both dual core ARM Cortex-A9 processors and FPGA fabric. In our evaluation, we use the ARM processors on the Zynq platform to drive the inputs and outputs of our solutions.

As described early in Section 5.3, to evaluate the runtime performance and area efficiency of hardware solutions generated by TFJ, we compare them with a soft-core microprocessor. Using a soft-core processor allows for a direct comparison of performance (run-time) and

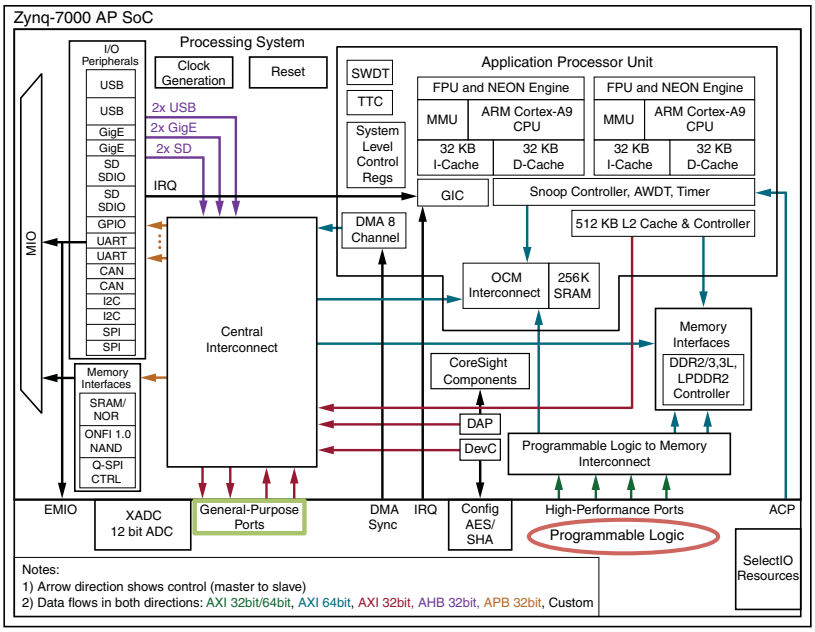


Figure 5.15: Block diagram of Xilinx Zynq-7000 [145]. Our automatically generated processing engines are implemented in the programmable logic (marked with a red oval). The processing engines interface with the onboard ARM processors using the general-purpose ports marked in green.

area using productive design methodologies instead of manually generating hardware solutions with Verilog RTL. Using a FPGA-based solution also obviates the need for software simulation (RTL simulation) of hardware designs in order to collect accurate cycle counts.

LUTs	DSP48s	RAMs	Max Frequency
4800	3	9	50.2 MHz

Table 5.5: FPGA statistics for our soft-core RISCV CPU on the Xilinx Zynq XC702 FPGA

To evaluate against a soft-core processor, we used an in-order 5-stage RISC pipeline. This processor design supports only 32-bit loads and stores, has a fully bypassed pipeline, and includes a small 4-entry branch target buffer. In addition, branches are resolved in the decode stage to reduce branch misprediction penalty. We use a 2-read, 1-write register file to ensure maximum efficiency of our processor micro-architecture implemented on a FPGA.

With single-cycle memory access and predictable branches, the processor retires approximately 1 instruction-per-cycle. Our soft-core processor implements a variant of the RISCV ISA [140] and includes a 4 kByte instruction cache. The rest of the memory system is identical to the soft-core processor and the automatically generated processing engines. We compiled the kernels listed in Section 5.3.1 to the RISCV ISA using GCC 4.4.0. FPGA statistics for our soft-core CPU are shown in Table 5.5.

We used a data-parallel operation length of 16 for all TFJ generated HLS solutions

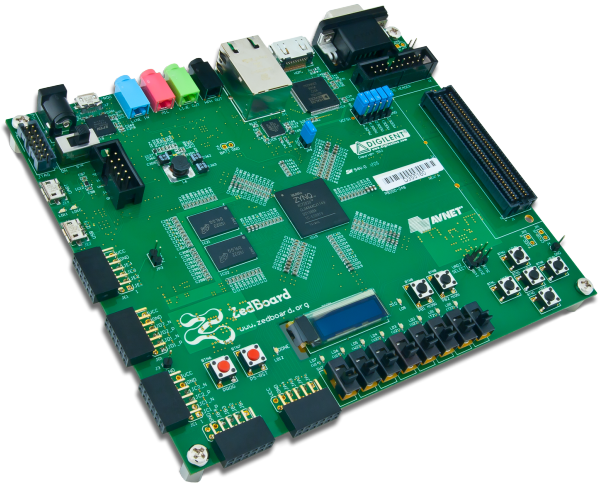


Figure 5.16: ZedBoard Zynq-7000 Development Board used for FPGA evaluation

and to implement our designs, we used Xilinx ISE 14.1 for logic synthesis, mapping, and place-and-route. The ARM cores on the Zynq platform run Linux 3.3.0-14.2. Our hardware solutions are mapped into an uncached physically addressed region of the ARM processors address space [1]. This allows for a simple polling protocol (shown in Figure 5.17) to move data between the FPGA fabric and the ARM processors. All designs operate on a 100 MHz FPGA clock.

5.3.3 Results and Analysis

The FPGA resource usage statistics and static timing analysis of the automatically generated processing engines for each kernel are shown in Table 5.6. Resource usage grows linearly with number of processing engines for all kernels. We are able to evaluate the Gaussian mixture model kernel in systems that have less than 7 processing engines. This is due to limited resources on our Zynq FPGA. The Gaussian mixture model accelerator requires significantly more resources than the other kernels. A single processing engine Gaussian mixture model accelerator requires almost 3× as many LUTs a single processing engine color conversion accelerator does.

We use the same memory subsystem for both the soft-core CPU and the automatically generated processing engines so that we can directly compare LUT utilization between the two implementations. The color conversion kernel uses fewer LUTs than the soft-core CPU does for all configurations. In contrast, a single vector-vector add processing engine with memory subsystem requires approximately the same number of LUTs does as our soft-core CPU. A single processing engine matrix-multiply accelerator requires about 2× the FPGA resources as the soft-core CPU does.

We present results for all kernels with 16 kByte shared write-back cache with 128-byte cache lines. We use this cache configuration as a 16 kByte cache can hold the working sets

[1] Thanks to Andrew Waterman for the implementation of a memory-mapped I/O device.

```
1  while(true) {
2    unsigned int x  = (unsigned int)d.read(0);
3    if(x & 0x1) {
4      unsigned int addr = x >> 8; bool write = x & 0x2;
5      if(write) {
6        mem[addr] = d.read(1); mem[addr+1] = d.read(2);
7        mem[addr+2] = d.read(3); mem[addr+3] = d.read(4);
8      } else {
9        d.write(1, mem[addr]); d.write(2, mem[addr+1]);
10       d.write(3, mem[addr+2]); d.write(4, mem[addr+3]);
11     }
12     d.write(0,(~0));
13   }
14   x = (unsigned int)d.read(7);
15   unsigned done = x >> 31;
16   if(done) break;
17 }
```

Figure 5.17: C++ source used as glue to interface memory-mapped accelerators in Zynq FPGA fabric with ARM cores. Line 2 polls the accelerator status register. Bit zero of the status register indicates a request. As shown in line 4, the second bit of the status register indicates if the transaction is a read or write while the memory operation address is placed in the upper 24 bits of the status register. Lines 5 through 11 perform the actual 16-byte read or write while line 12 acknowledges a complete memory transaction. Finally, lines 14 through 16 check if the accelerator has completed executing a given computation. Using this code, a 16-byte cache line reload takes approximately 100 cycles on the 100 MHz FPGA clock domain.

of all the benchmarks presented in this section. We evaluate the cache-based systems with 1-cycle (Figure 5.18), 10-cycle (Figure 5.19), and 100-cycle (Figure 5.20) cache reloads to show performance under different memory subsystem configurations.

Single-cycle cache reloads represent an idealized memory system. Conflict and cold cache misses will be most apparent in this configuration. If we were to directly connect our accelerator subsystem to a DRAM interface on the Zynq platform, cache reloads would take approximately ten cycles, given our FPGA logic clock [146]. Therefore, the ten-cycle configuration presents a realistic evaluation of our system on an FPGA with a hard (fixed-function) DRAM memory controller. The 100-cycle cache line reload latency represents a realistic memory configuration if our custom processing engines were implemented in a modern ASIC technology. We compute speed-up by comparing the number of execution cycles on the soft-core to the number of execution cycles on the processing engine. This does not account for the difference in obtainable frequency between the soft-core and the processing engines.

As shown in Figure 5.18, our processing engines are highly scalable with single-cycle cache reloads. The Gaussian mixture model kernel scales to greater than 12× soft-core performance with 4 processing engines, while the matrix-multiply kernel scales to just under 12× soft-core performance with 8 processing engines. In addition, the color conversion kernel performance scales to just under 8× soft-core performance with 5 processing engines. With single-cycle cache line reloads, the vector-vector add kernel has the poorest scaling.

	FPGA Resource	Vector-vector addition	Color conversion	Matrix multiply	Gaussian mixture model
1 PE	LUTs	5443	3548	7539	10245
	DSP48s	0	3	3	3
	RAMs	11	11	11	11
2 PEs	LUTs	9956	4915	14213	18060
	DSP48s	0	6	6	6
	RAMs	12	12	12	12
3 PEs	LUTs	14203	6400	20814	25920
	DSP48s	0	9	9	9
	RAMs	14	14	14	14
4 PEs	LUTs	19700	8644	27370	34017
	DSP48s	0	12	12	12
	RAMs	16	16	16	16
5 PEs	LUTs	24457	10473	33286	42031
	DSP48s	0	15	15	15
	RAMs	18	18	18	18
6 PEs	LUTs	28972	11797	39329	49620
	DSP48s	0	18	18	18
	RAMs	20	20	20	20
7 PEs	LUTs	33322	15016	45486	57858
	DSP48s	0	21	21	21
	RAMs	22	22	22	22
8 PEs	LUTs	37293	17273	52597	63100
	DSP48s	0	24	24	24
	RAMs	24	24	24	24

Table 5.6: FPGA statistics for the four benchmarks of Section 5.3.1 on the Xilinx Zynq XC702 FPGA. All processing engines were generated using the datapath resource configuration file shown in Figure 5.8a. Designs that require more LUTs than provided by the FPGA are marked in red.

The performance of this kernel is limited to a maximum of slightly greater than 4× soft-core performance with 5 processing engines. The performance of the vector-vector add kernel is limited by shared cluster memory bandwidth and cache conflicts. However, as vector-vector add is a bandwidth-bound kernel, the results show our automatically-generated processing engines are more efficient with available memory bandwidth than our soft-core microprocessor.

Figures 5.19 and 5.20 show performance with the shared cache for 10-cycle and 100-cycle reloads, respectively. As the matrix-multiply kernel is compute bound, performance scales with additional processing engines under both memory latency configurations. With 10-cycle memory latency, a 10× speed-up is achieved with 8 processing engines. With 100-cycle memory latency, the matrix-multiply kernel achieves a 5× speed-up over the soft-core CPU. Performance results for the matrix-multiply kernel are explainable given the L2 cache reload statistics shown in Table 5.7. The working set for our 32×32 matrix-multiply computation entirely fits in on-chip memory. Only 1025 reloads are required to fill the L2

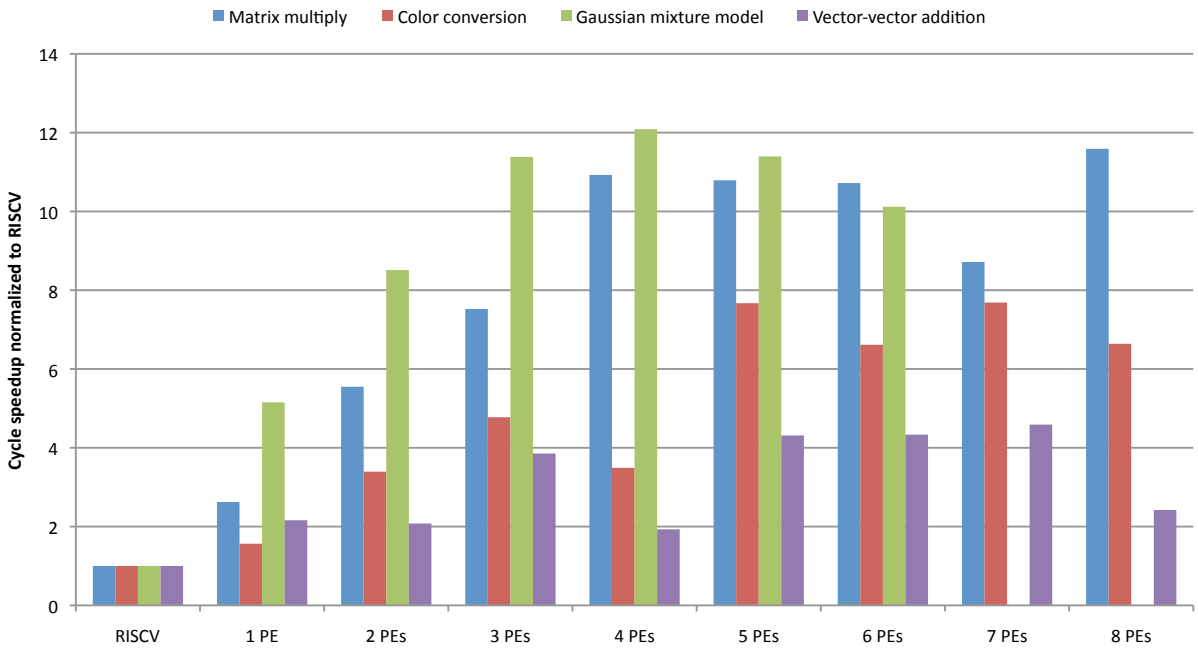


Figure 5.18: Scaling with one-cycle reloads (*larger speedup connotes better results)*

cache. The fact that the kernel has a relatively small number of cold cache misses insulates the accelerators for changes to memory latency.

With 10-cycle reloads, the other kernels achieve between 6× and 2× performance improvements over the soft-core CPU. With 100-cycle memory latency, performance drops to between 3× and 1× soft-core performance for vector-vector addition, color conversion, and Gaussian mixture model kernels. As expected, the bandwidth-bound vector-vector addition kernel suffers the most with increased memory latency. Scalability with 10 or 100 cycle cache reloads is reduced compared with that of single-cycle reloads; nevertheless, adding processing engines increases performance. A larger cache would help all kernels except vector-vector addition. The inclusion of banked caches with multiple memory ports would help performance for all memory latency configurations. Adding associativity to caches would also likely help performance scaling due to reduced cache conflict misses. Finally, a LUT-efficient design would use less than 8 processing engines as peak performance is achieved with a smaller number of processing engines for all kernels except matrix multiply.

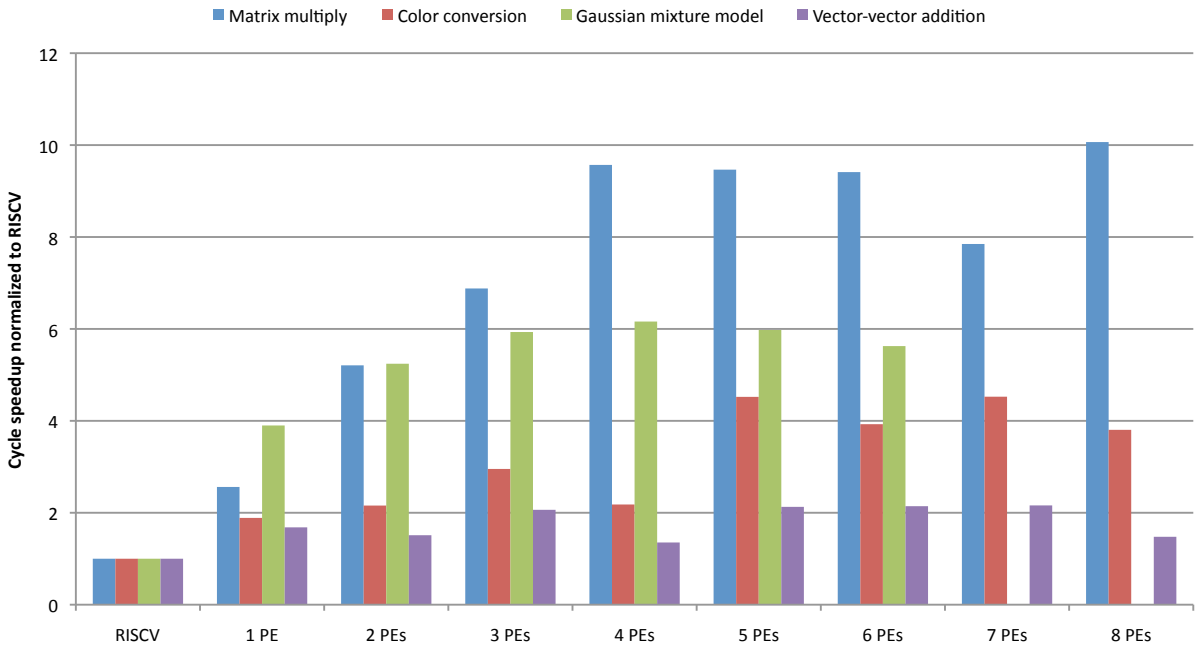


Figure 5.19: Scaling with ten-cycle reloads (*larger speedup connotes better results*)

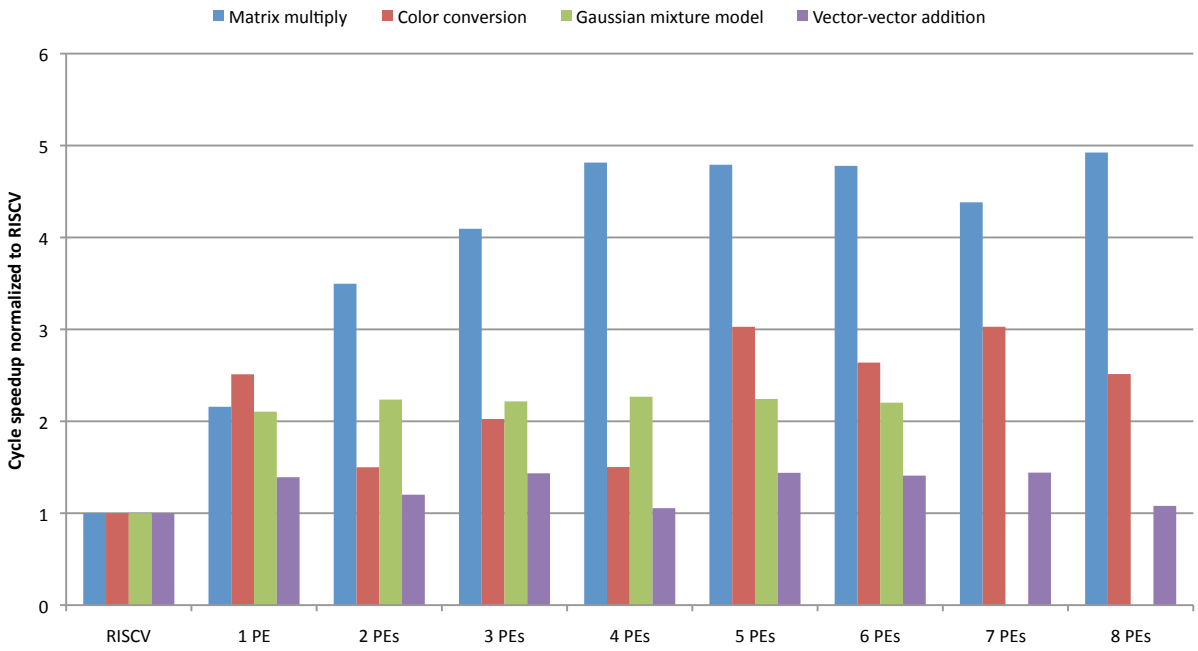


Figure 5.20: Scaling with 100 cycle reloads (*larger speedup connotes better results*)

	RISCV	1 PE	2 PEs	3 PEs	4 PEs	5 PEs	6 PEs	7 PEs	8 PEs
Vector-vector addition	525175	393217	458761	393217	524289	393437	402277	393651	518211
Color conversion	454730	164212	329249	244510	329306	164182	188285	164180	198199
Matrix multiply	946	1025	1025	1025	1025	1025	1025	1025	1025
Gaussian mixture model	1495642	978352	983023	1020905	1000948	1008119	1017435	N/A	N/A

Table 5.7: L2 cache reload statistics. Reload counts are identical for all three memory latency configurations. Configurations that do not fit in our Zynq device are marked in red.

5.4 Case Study: Tuning Hardware Matrix Multiply

Matrix multiply is a well understood numerical kernel with a large design space. We have already explored a small portion of the design space for software-based solutions using TFJ in Section 4.4, and we have investigated hardware results for matrix multiply in Section 5.3.1. However, the hardware results for matrix multiply explored considered only a small 32×32 matrix multiply problem with an integer matrix. In this section, we explore different hardware tuning parameters for a 160×160 single-precision floating-point matrix multiply. To demonstrate the portability TFJ, we use the Python matrix multiply source that was used in Chapter 4 to explore performance behavior of software solutions for matrix multiply.

This case study examines the performance implications of different data-parallel operation length, L2 cache sizes, number of processing elements, and non-blocked versus blocked matrix multiply algorithms. We manually swept the parameters previously described for this case study; however, as mentioned in Section sec:tfj-hw:micro, there exists potential for automating the design space explorating using existing techniques.

We use a 160×160 matrix as it is the largest matrix we can fit in an L2 cache on our FPGA platform. For our blocked matrix multiply (shown in Figure 5.21b), we have chosen a matrix sub-block size of 8×8 as it is large enough to enable a reasonably long data-parallel operation and small enough to fit in a modestly sized L1 cache. We use a 4 kByte L1 cache to hold three 8×8 sub-matrices for all blocked matrix multiply experiments except for the eight processing element system. Due to FPGA resource constraints, we are able to fit only a 1 kByte L1 cache.

In this case study, we generate non-blocked matrix multiply processing elements using data-parallel operation lengths from 1 to 16 and blocked matrix-multiply processing elements from length 1 to 8. We are limited to a data-parallel operation of length eight with the blocked solution because a length greater than eight would be larger than sub-block size and have no benefit to performance. As described in Section 5.2.2, TFJ's HLS flow converts data-level parallelism found by the compiler front-end into memory-level parallelism. By increasing data-parallel operation length, we expect to increase memory-level parallelism and in turn increase the performance of our matrix multiply code. For the matrix multiply computation, vectorization also increases operand locality due to loop interchange. For example, in the three-nested loop matrix multiply shown in Figure 5.21a, the TFJ front-end will interchange the loop nest to achieve the IKJ ordering. The interchange enables loop-invariant code motion to hoist access to the matrix A out of the inner loop, thereby reducing memory traffic and increasing locality. We expect longer data-parallel operation lengths to result in high performance processing elements at the cost of more FPGA resources.

We explore different L2 cache sizes to determine the implications of cache size on matrix multiply performance. Smaller L2 caches demonstrate the impact of memory-level parallelism and the blocked matrix multiply algorithm. While the largest L2 cache implementable on our Zynq platform (256 kBytes) can not entirely hold all three matrices (300 kBytes), it does capture a significant amount of the data set and demonstrates performance when operands are stored largely in the memories on the FPGA. By varying the number of processing elements, we explore the parallel scalability of our system hardware template.

```
def matmul(A,B,Y,n):
  for i in range(0,n):
    for j in range(0,n):
      for k in range(0,n):
        Y[i][j]=Y[i][j]+A[i][k]*B[k][j];
```

(a) Matrix multiply

```
def bmatmul(A,B,Y,n):
  for i in range(0,n):
    for j in range(0,n):
      for k in range(0,n):
        for ii in range(0,8):
          for jj in range(0,8):
            for kk in range(0,8):
              Y[ii+i*8][jj+j*8]=Y[ii+i*8][jj+j*8]+A[ii+i*8][kk+k*8]*\
                                B[kk+k*8][jj+j*8];
```

(b) Blocked matrix multiply with an 8×8 block size

Figure 5.21: Python implementations of the two kernels used for tuning hardware matrix multiply

While we hope performance will increase linearly with the number of processing elements, in practice, we expect performance to plateau after a certain number of them. The limited scaling is due to memory subsystem saturation. This case study characterizes the scalability of multiple processing engine solutions for different combinations of data-parallel operation lengths, L2 cache sizes, and matrix multiply algorithms.

5.4.1 Results

To evaluate our matrix multiply accelerators, we used the Xilinx Zynq platform with the methodology of Section 5.3.2. TFJ's HLS engine used the single-precision floating-point units and optimized L2 cache described in Section 5.2.4. The custom hardware accelerators were both synthesized and placed-and-routed using Xilinx ISE version 14.1. The design space exploration required the implementation of 246 design points, and all design points were evaluated on real hardware using the Digilent ZedBoard. All custom hardware solutions operate at 100 MHz.

FPGA resource utilization for the non-blocked and blocked matrix multiply accelerators are shown in Tables 5.8 and 5.9, respectively. The resource statistics are shown only for designs with a 256 kByte L2 cache; however, as presented in Table 5.4, increasing the size of the L2 cache has little impact on the total number of LUTs used. Of particular interest is the impact of data-parallel operation length on LUT utilization for the non-blocked matrix multiply. Increasing the data-parallel operation length results in a larger processing element

	FPGA Resource	1 entry	2 entries	4 entries	8 entries	16 entries
1 PE	LUTs	3948	5238	6048	7675	10971
	DSP48s	14	14	14	14	14
2 PEs	LUTs	5997	8769	10451	13463	19830
	DSP48s	22	22	22	22	22
4 PEs	LUTs	10162	15820	19160	25113	37920
	DSP48s	38	38	38	38	38
8 PEs	LUTs	18552	29929	36516	48442	N/A
	DSP48s	70	70	70	70	N/A

Table 5.8: Single-precision non-blocked matrix multiply FPGA resource statistics for 256 kB L2 with data-parallel operation lengths from 1 to 16. Designs that do not fit in Zynq device are marked in red.

	FPGA Resource	1 entry	2 entries	4 entries	8 entries
1 PE	LUTs	8199	8148	8148	8471
	DSP48s	14	14	14	14
2 PEs	LUTs	18880	18835	18835	18883
	DSP48s	22	22	22	22
4 PEs	LUTs	34991	34931	34931	34905
	DSP48s	38	38	38	38
8 PEs	LUTs	40095	39935	39935	39935
	DSP48s	70	70	70	70

Table 5.9: Single-precision blocked matrix multiply FPGA resource statistics for 256 kB L2 with data-parallel operation lengths from 1 to 16. Due to FPGA resource limitations, the eight processing engine design uses 256 word L1 cache. All other designs use 1024 word L1 cache.

due to increased operand locality and correspondingly greater hardware register usage. In contrast, the blocked matrix multiply accelerators have approximately identical LUT usage across different data-parallel operation lengths. The fixed loop bounds for inner 8×8 sub-matrix multiply allows the LLVM framework to automatically unroll the inner loop, largely achieving equivalent performance benefits as achieved by the vectorization based approach.

Single Processing Element Solutions

Figure 5.22 presents results for single processing element solutions. A peak performance of nearly 40 MFlops/sec is obtained with a single non-blocked matrix multiply processing element using a data-parallel operation length of 16. With a single processing element, the non-blocked matrix multiply solution is highly dependent on long vectors to achieve high-performance. In contrast, the blocked-matrix multiply solution is not dependent on data-parallel operation length for performance. When vectors are less than length eight, vectorization has little impact on performance. As presented, the non-blocked matrix multiply is highly dependent on a large L2 cache to achieve high performance. With a data-parallel

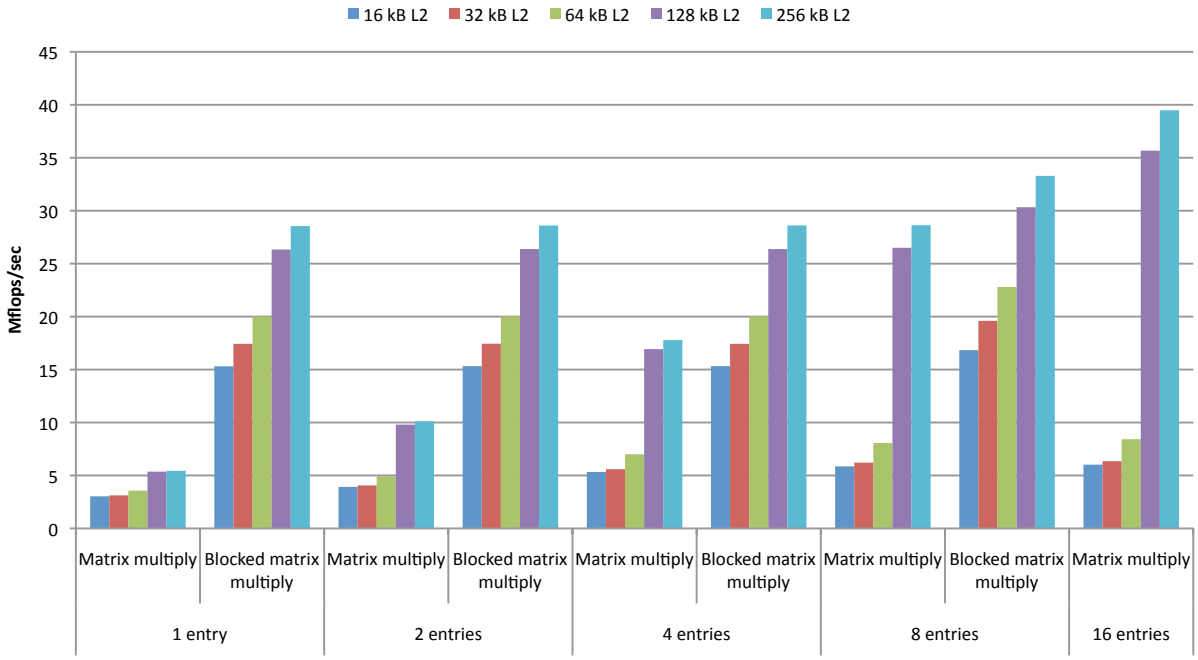


Figure 5.22: Matrix multiply using a single processing element

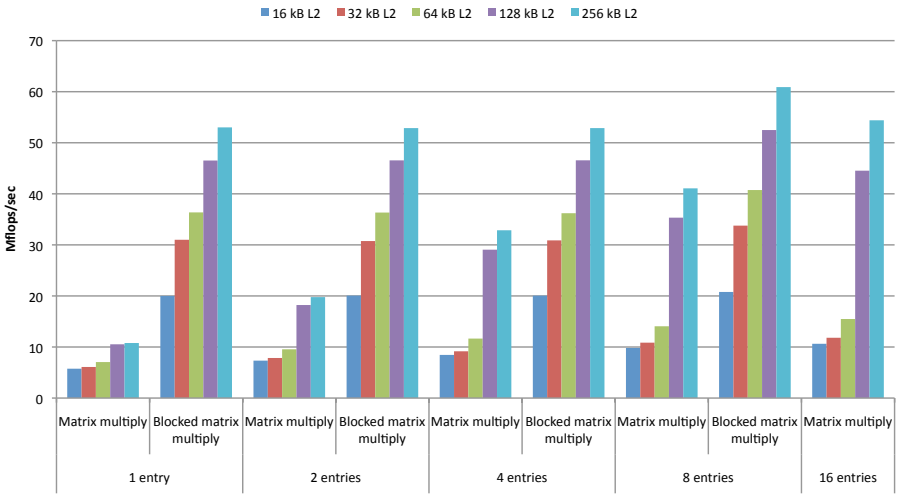
operation length of 16, the non-blocked matrix multiply solution is nearly 7× faster with a 256 kByte L2 than with a 16 kByte L2. On the other hand, the blocked solution is much more tolerant of a small L2 cache. Performance varies by a factor of less than 2× from the smallest to largest L2 cache.

Two Processing Element Solutions

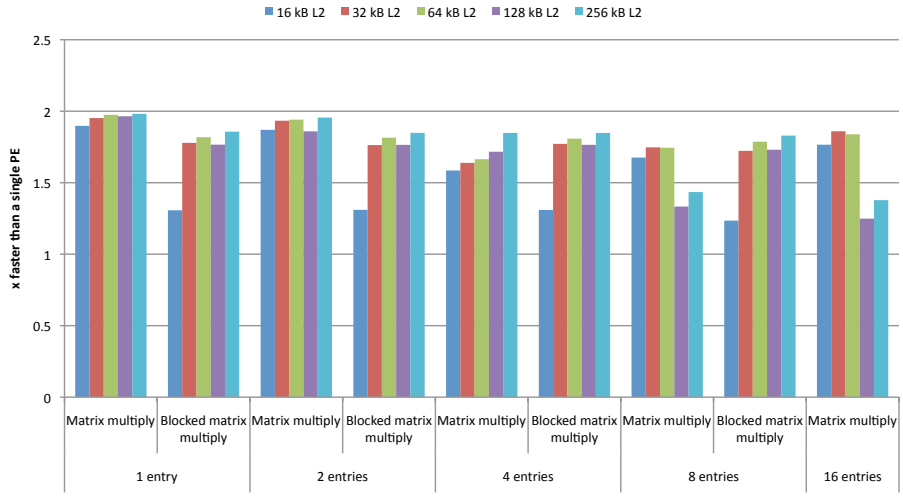
With two processing elements, the performance of our matrix multiply accelerators increases between 1.2× to 1.9× compared to that of a single processing element solution. As shown in Figure 5.23a, raw performance increases to slightly greater than 60 MFlops/sec with a 256 kByte cache and a data-parallel operation length of eight. The non-blocked and blocked matrix multiply accelerators have different behaviors with increased processing elements. The blocked matrix multiply solutions scale well for all L2 cache sizes and data-parallel operation lengths, while non-blocked solutions scale well for data-parallel operation lengths less than eight. This is probably because a single non-blocked processing element with a long data-parallel operation length is able to saturate the memory subsystem.

Four Processing Element Solutions

As shown in Figure 5.23c, performance nearly approaches 100 MFlops/sec with 4 processing elements using blocked matrix multiply. The solutions with 4 processing elements are between 1.0× and 3.9× faster than those with the equivalent single processing element



(a) Raw performance of matrix multiply using 2 processing elements

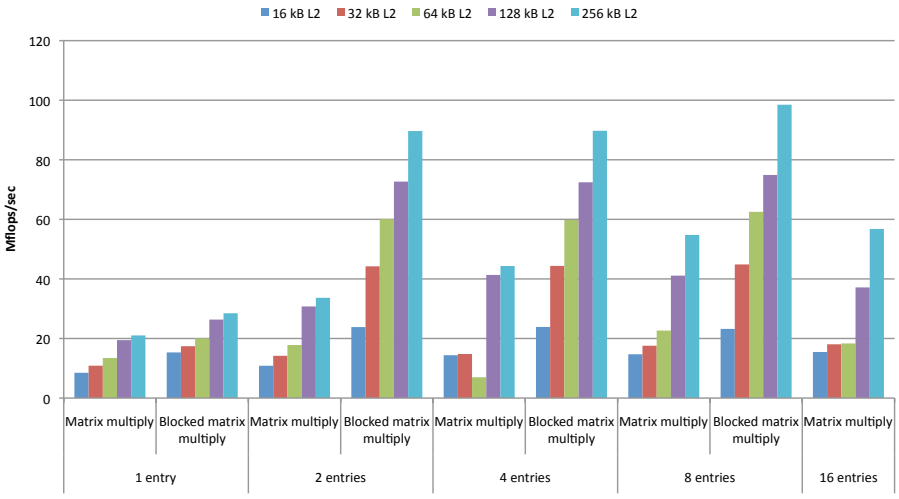


(b) Scalability compared to single processing element matrix multiply

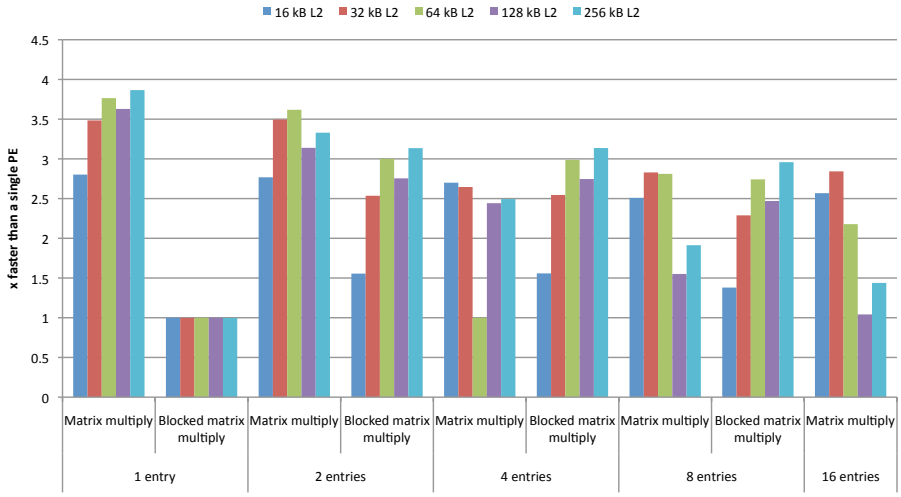
solution. With increased processing elements, longer data-parallel operation lengths have less of an impact on the non-blocked matrix multiply accelerators. While performance gap between non-blocked matrix multiply accelerators with a data-parallel operation length of 1 and 16 was greater than 7× with a single processing element, with 4 processing elements, the gap fell to below 3×. The L2 cache bandwidth for the system template likely becomes an issue with 4 processing elements.

Eight Processing Element Solutions

In order to fit eight processing elements on our Zynq FPGA, we had to make minor modifications to our system hardware template. In particular, we had to decrease the size of the L1 cache from 4 kBytes to 1 kByte for the blocked matrix multiply accelerators. In addition, the non-blocked matrix multiply accelerator built with a data-parallel operation



(c) Raw performance of matrix multiply using four processing elements

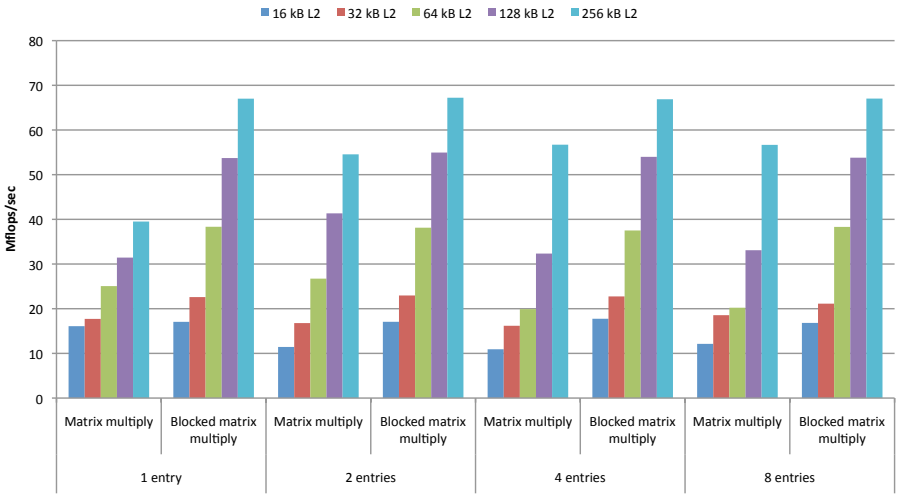


(d) Scalability compared to single processing element matrix multiply

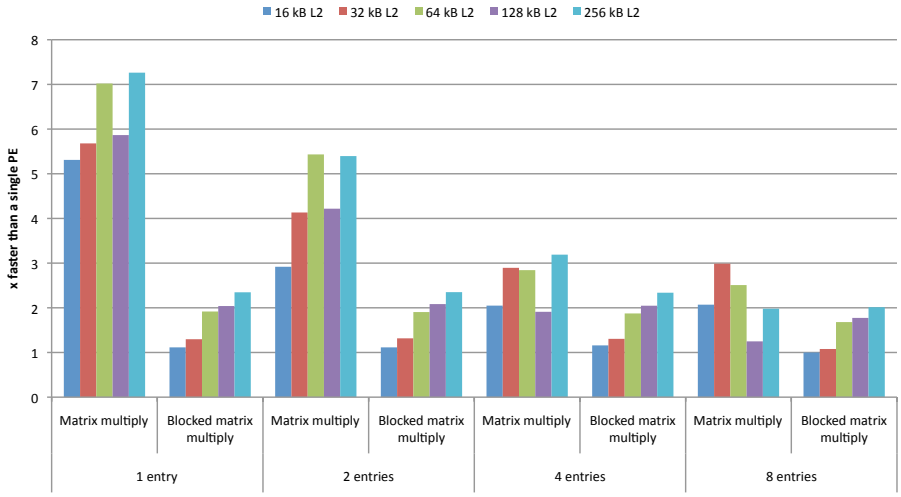
length of 16 would no longer fit with eight processing elements. As shown in Figure 5.23e, doubling the processing elements from 4 to 8 increases memory bandwidth issues seen in the previous configurations. Reducing the size of the L1 cache on the blocked matrix multiply accelerators only exacerbates the memory bandwidth problem. As a result, only the relatively low performance accelerators built with a data-parallel operation length of 1 or 2 increase in performance compared to the 4 accelerator solution.

Summary of Matrix Multiply Case Study

Using TFJ we have generated solutions for matrix multiply that span from three MFlops/sec to slightly under 100 MFlops/sec. These results demonstrate that TFJ can generate efficient hardware solutions to computationally intensive kernels from the same Python source as that used by the TFJ software backends. The results also show the impact of our



(e) Raw performance of matrix multiply using eight processing elements



(f) Scalability compared to single processing element matrix multiply

approach to generating non-blocking memory operations, as longer data-parallel operation lengths result in higher performance solutions. The case study also eludicates room for future improvements on the hardware system template as the performance of our solutions do not scale well beyond 4 processing elements.

5.5 Summary

In this chapter, we have described the design and implementation of TFJ's high-level hardware synthesis engine. We have presented our approach to datapath scheduling, register binding, and generation of non-blocking memory operations. In particular, we have demonstrated the approach TFJ uses to convert data-level parallelism to memory-level parallelism. Figure 5.10 shows our matrix multiply results of processing elements built using

our approach have up to 10× more memory-level parallelism than the equivalent scalar implementation. We also describe in detail the design and implementation of the components used in our system hardware template have also been described in detail.

Using TFJ's HLS engine, we have evaluated performance results for four simple kernels with a highly optimized soft-core processor. Compared to an optimized soft-core CPU, TFJ generated solutions are up to 12× faster. We have also evaluated several different implementations of single-precision matrix multiply to demonstrate the impact of data-parallel operation length and memory hierarchy configuration on the performance of TFJ generated solutions. These experiments varied the number of single-precision matrix multiply accelerators to investigate the parallel scalability of our custom hardware solutions. Using the same starting Python source, we were able to generate non-blocked matrix multiply accelerators with performance from 3 MFlops/sec to 57 MFlops/sec. Likewise, we were able to generate blocked matrix multiply accelerators with performance from 15 MFlops/sec to 98 MFlops/sec without modifying any Python source.

Chapter 6

A Case Study in Speech Recognition using Three Fingered Jack

In this chapter, we use Three Fingered Jack's multiple backends to perform a case study in implementing speech recognition on mobile devices. Section 6.1 describes the challenges of speech recognition on energy-constrained mobile devices. Section 6.2 provides a brief background of the concepts related to large vocabulary continuous speech recognition and profiles our speech recognizer to better understand the memory subsystem behavior. The kernels used with Three Fingered Jack for hardware and software solutions to speech recognition are described in Section 6.3. The methodology used to verify correctness of our hardware solutions is presented in Section 6.4, while detailed energy results for both software and custom hardware solutions are presented in Section 6.5. Section 6.6 provides a summary of the work presented in this chapter.

6.1 Challenges for Mobile Speech Recognition

Automatic speech recognition (ASR) has become an enabling technology on today's mobile devices. Current solutions, however, are not ideal. An ideal mobile ASR solution would be capable of running continuously (always-on), provide a low-latency response, have a large vocabulary, and operate without excessive battery drain while untethered to WiFi. Mobile devices are limited by their batteries that gain only 4% capacity annually from technological innovations [121]. Given the mobile energy constraint, we seek to understand the energy consumption of various speech recognition approaches and provide a productive method for designing energy efficient ASR solutions.

Some current mobile solutions, such as Apple's Siri, provide ASR capabilities by sending audio or feature vectors to a remote server in the cloud over a wireless network. Unfortunately, using the cloud to offload ASR may not be the most energy efficient approach to delivering speech recognition solutions. We calculate that sending the same 60 seconds of speech data as 39-dimension single-precision Mel-Frequency Cepstral Coefficients (MFCCs) feature vectors would consume between 100 and 340 joules using the 3G energy-per-bit values presented by Miettinen [104]. The large range in 3G energy consumption is due

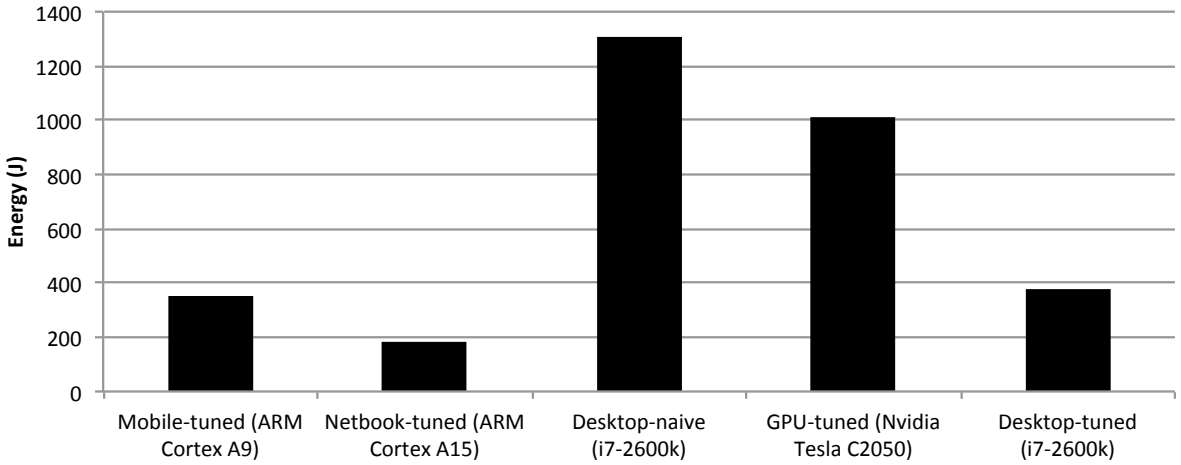


Figure 6.1: Energy consumption for 60s of ASR on a menagerie of commercial platforms. All devices achieve real-time performance on Wall Street Journal 5k corpus [115]. Energy recorded using a "watts up? PRO" power meter.

to differences caused by geographic location, data rate, and radio vendor. At this rate, a typical 20 kJ battery would last between 1 to 3 hours when performing continuous speech recognition.

An alternative to using a cloud-based solution for speech recognition is to employ an ASR solution that runs locally on the mobile device. There are many potential motivational reasons for client-only speech recognition other than increased energy efficiency. Security, reliability, and quick response are enabled when using a client-based solution. Security is enhanced as potential confidential audio is not sent over the network where data can be potentially snooped. Security risks associated with Apple's Siri resulted in its being banned at large corporations [43]. Furthermore, reliability of speech recognition solutions is increased when performed locally as cloud-based solutions do not work in low connectivity environments and thus are incapable of providing robust service in a variety of environments. Finally, locally performing speech recognition reduces response time as speech data does not need to be uploaded to a remote server.

In order to evaluate energy consumption characteristics of software-based local speech recognition solutions, we performed a simple experiment using 60 seconds of audio from the WSJ5k corpus. We measured the energy consumption on several hardware platforms. Figure 6.1 shows the local energy consumption characteristics and summarizes our results: our most efficient platform requires approximately 200 joules to perform ASR on 60s of audio. At this rate, a typical 20 kJ battery would last roughly 100 minutes. With a software-only speech recognition solution, the number of hours that a device can perform speech recognition is not significantly improved. For all day speech recognition, another solution is required.

Speed and energy efficiency can be improved by employing custom hardware designed for speech recognition. Several hardware-based solutions have been proposed during the last 30

years [83, 10, 102, 87, 110, 97, 25]. These approaches claim performance or energy benefits of 10 to 100× over that of a conventional microprocessor. However, these approaches employ an inflexible design process in which high-level algorithmic design decisions are hard-coded into a low-level implementation. This means that the system would potentially require a complete overhaul in order to experiment with a new algorithmic approach or language model. Additionally, these systems are point samples and do not give us a full understanding of the design space for speech recognition hardware solutions.

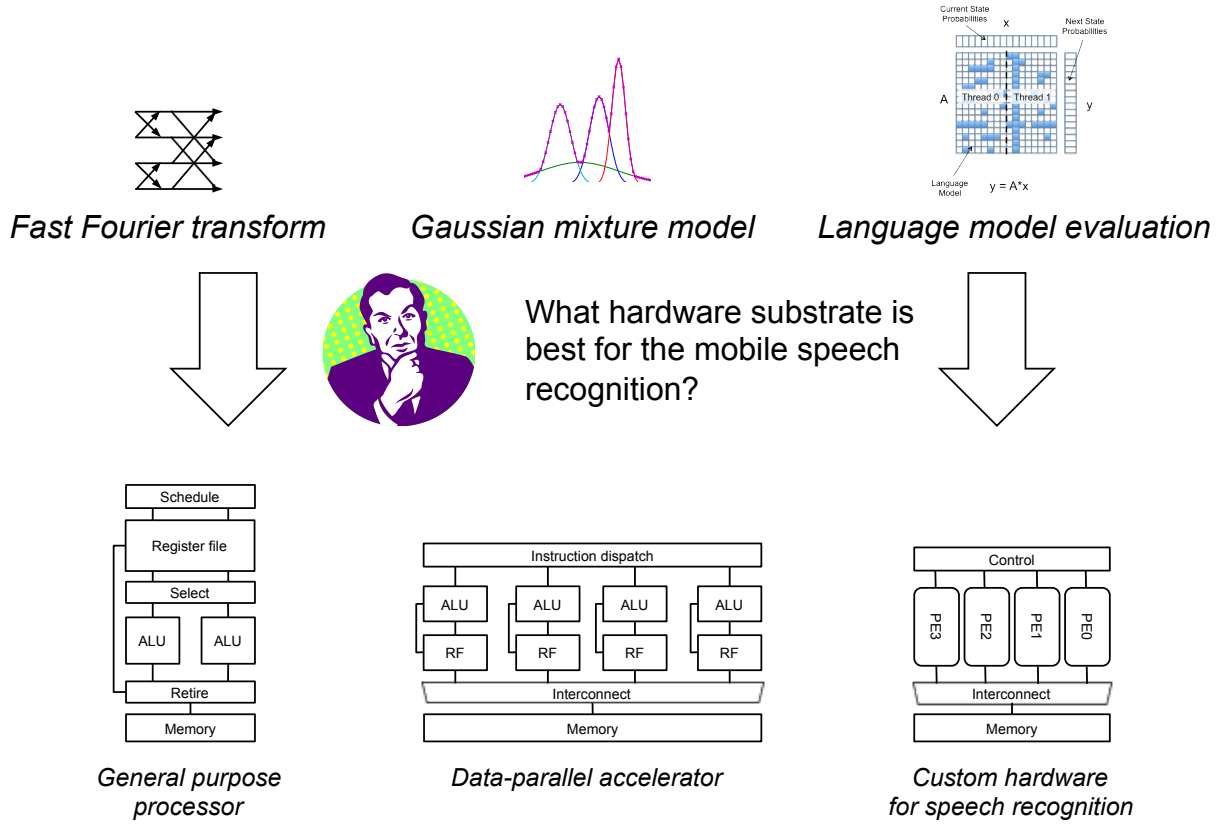


Figure 6.2: There are a number of ways of providing speech recognition on mobile devices. Different target markets, performance requirements, client-computing profiles, and energy concerns motivate many different solutions. For these reasons, we need the ability to explore the entire design space of general purpose CPUs, data-parallel accelerators, and custom fixed-function hardware.

In this case study, we explore high performance software and hardware implementations of an ASR system that can run locally on a mobile device. As shown in Figure 6.2, there are a number of potential ways of providing this capability. As previously mentioned, it is possible to run the speech recognizer entirely on the mobile phone's host processor. However, if the device has a more advanced programmable data-parallel accelerator, it is possible to run key speech recognition kernels on it for a potentially significant energy efficiency improvement. Finally, custom fixed-function hardware can be used for maximal energy efficiency. As shown

in Figure 6.3, we intend our fixed-function speech hardware solutions to be included as a small module on system on a chip (SoC).

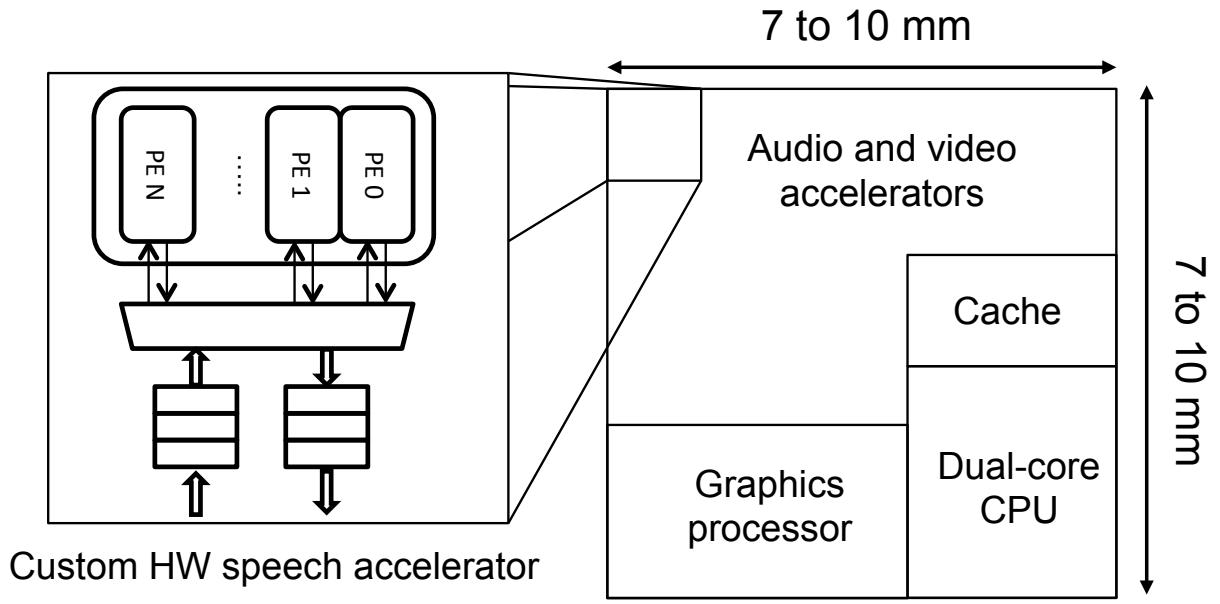


Figure 6.3: A system on a chip: a large fraction of the SoC die area is dedicated to custom accelerators such as video encoders/decoders or image processing. The hardware speech recognition solutions presented as a possible solution for mobile ASR in this chapter are intended be a small logic block (under $5mm^2$) on a SoC.

The correct mobile device configuration for speech recognition also depends on several additional parameters. Given differences in target market, time-to-market concerns, and client-computing profile, it is unlikely that one specific configuration will provide the best speech recognition solution on all mobile system configurations. Moreover, each of the mobile system constraints previously mentioned is rapidly changing; the best solution for this product generation may not be the correct solution for the next one. For these reasons, providing the best speech recognition solutions requires that we explore the *entire* design space of traditional CPU, programmable data-parallel accelerator, and fixed-function cores. Manually crafting a point solution for each design point, however, is too time consuming and error prone to be practical. For these reasons, we automate the generation of key components of our speech recognition system using *Three Fingered Jack*.

Through detailed hardware simulation we are able to produce accurate estimates for energy and performance. We show that our custom solutions are 3.6× and 2.4× more energy efficient than a conventional microprocessor and a highly-optimized data-parallel processor, respectively.

6.2 Speech Recognition Background

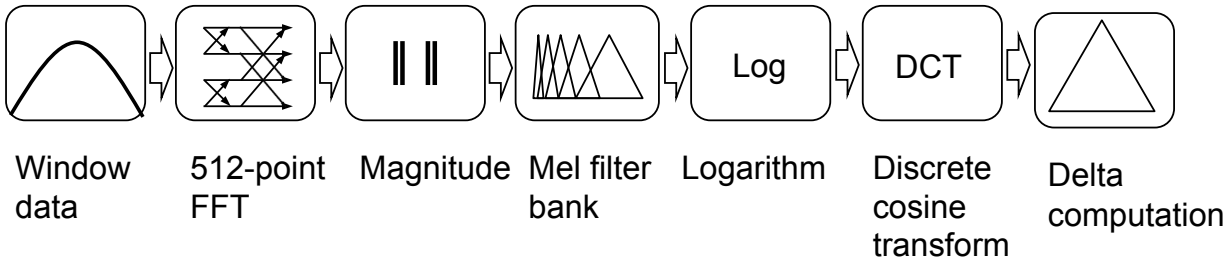


Figure 6.4: An architecture for Mel-frequency cepstral coefficient generation

An ASR application accepts an utterance as input waveform and infers the most likely sequence of words and sentences of that utterance. Our ASR system is built on top of the ICSI Parallex decoder [35]. As shown in Figure 6.5, Parallex is built around a hidden Markov model (HMM) inference engine with a beam search approximation and may be easily decomposed into feature extraction and an inference engine. A feature extraction pipeline (Figure 6.4) generates 39-dimensional MFCCs for every 10ms frame from an analysis window of 25ms. The MFCCs are fed to the inference engine to recognize words and sentences. The inference engine has two key phases: observation probability calculation using a Gaussian Mixture Model (GMM) and a graph-based knowledge network traversal. The GMM computes the probabilities of phonemes in a given acoustic sample. These probabilities are used by the HMM to compute the most likely sequence of words using the Viterbi search algorithm. We use a beam search approximation to prune the search space.

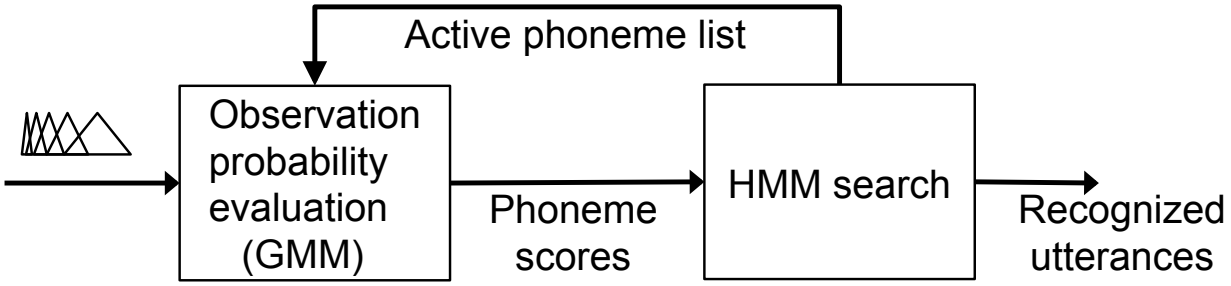


Figure 6.5: Architecture of an HMM-based speech recognizer

The inference engine at the heart of Parallex [35] employs the linear lexical model (LLM) to implement the graph-based knowledge network used for language modeling. The LLM representation distinguishes between two types of transitions: within-word and across-word [120]. LLM has a highly regular structure that makes it favorable to a parallel implementation; however, this regularity comes at a cost as the representation contains many duplicated states [35].

We have two versions of Parallex: a portable C++ version and a hybrid implementation written in a combination of Python and C++. The hybrid implementation is written in

Python to take advantage of TFJ, while the rest of the application remains in C++. This approach allows us to demonstrate the power of TFJ without entirely reimplementing the speech recognizer in Python. We evaluate both versions of Parallex using the 5000-word Wall Street Journal corpus.

In order to focus our optimizations, we profiled our portable C++ speech recognizer running on a PandaBoard [1] to simulate contemporary mobile hardware. The profiling results led us to focus our efforts on accelerating the GMM and across-word transition kernels, as they consume 60% and 25% of the run-time, respectively.

6.2.1 Profiling Our Speech Recognizer

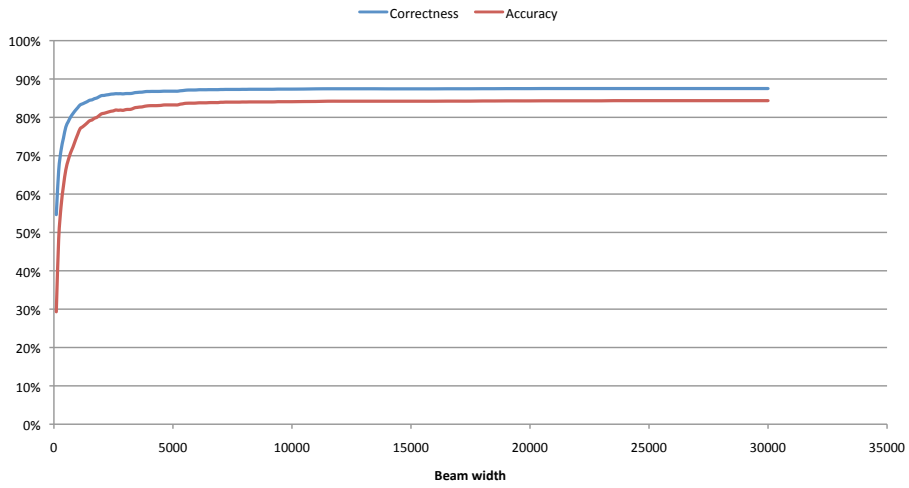
Past computer architecture research has extensively profiled and characterized [2, 102] Carnegie Mellon's Sphinx decoder as it has been included in two editions of the industry standard SPEC benchmarks (SPEC CPU2000 and SPEC CPU2006). Studies of the Sphinx decoder have shown that it extensively stresses the memory hierarchy due to the large working set and accessing the graph-based knowledge network results in an irregular memory access pattern.

To determine the impact of the more regular knowledge network representation used in our decoder, we performed several experiments to elucidate memory behavior. Our studies also include memory subsystem behavior as a function of the beam width used in the beam search approximation. As shown in Figure 6.6b, by limiting the maximum number of active states in our beam search, we both prune the search space and reduce the size of the working set. A smaller beam width has the potential downside of reduced accuracy; however, as shown in Figure 6.6a, large beam widths do not significantly improve the accuracy of our decoder. In fact, there is less than a two percent improvement in recognition correctness beyond a beam width of 3000. The simple experiment of Figure 6.6 demonstrates that a relatively narrow beam width can deliver both acceptable accuracy and high-performance with our speech recognizer. To understand interplay between beam width and the decoding performance of our recognizer, we implemented a simple memory hierarchy simulator. We used this simulator to measure the size of the program working set as a function beam width.

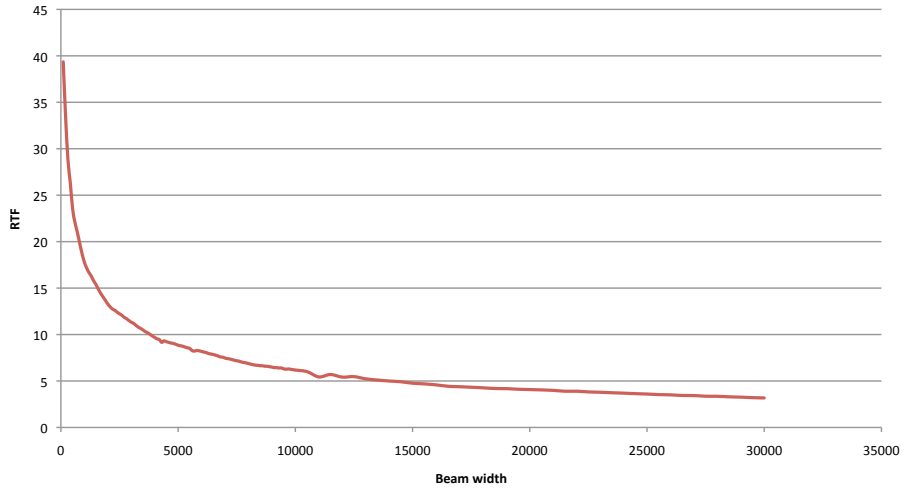
Memory Hierarchy Simulation

Our memory hierarchy simulators are built on top of the Intel Pin dynamic binary instrumentation framework [100]. The Pin framework makes it possible to quickly develop profiling and analysis tools using dynamic recompilation on existing binaries. The PIN API allows the user to insert instrumentation at arbitrary locations throughout an existing binary. Unlike other approaches to dynamic program analysis, Pin does not require the user to recompile his or her application to insert instrumentation code. As a result, off-the-shelf x86 binaries run without modification. In addition to an extensive instrumentation API, Pin emits efficient monitoring code by using a just-in-time compiler to re-optimize the

[1] `www.pandaboard.org`



(a) Speech recognition accuracy and correctness as a function of beam width. Accuracy and correctness computing using the HTK tools [147].



(b) Speech recognition real-time factor as a function of beam width. Large real-time factors imply faster speech decoding. Results generated with single-threaded C++ implementation of our speech recognizer running on an Intel Celeron G530.

Figure 6.6: Speech recognition correctness, accuracy, and real-time factor for a variety of beam widths. Experiments run on 2404 seconds of audio encapsulating 330 utterances from the Wall Street Journal 5000 word corpus. Plots generated by sweeping beam width from 100 to 10000 in steps of 100 and then from 10000 to 30000 in steps of 500.

binary under test after instrumentation code injection. Unlike pure architectural simulation, applications instrumented with Pin still have reasonable performance.

Our analysis framework is based on annotating all memory operations in order to analyze the program address stream. We instruct Pin to intercept the execution of all memory oper-

ations so that the program address stream is fed to our cache model. We have implemented a simple behavioral multi-level cache simulator in C++. The simulator is parameterizable and allows for run-time configuration of cache size, associativity, and number of levels of caching in the memory hierarchy. We use the cache simulation framework with a single-threaded C++ implementation of our speech recognizer for all workload characterization studies.

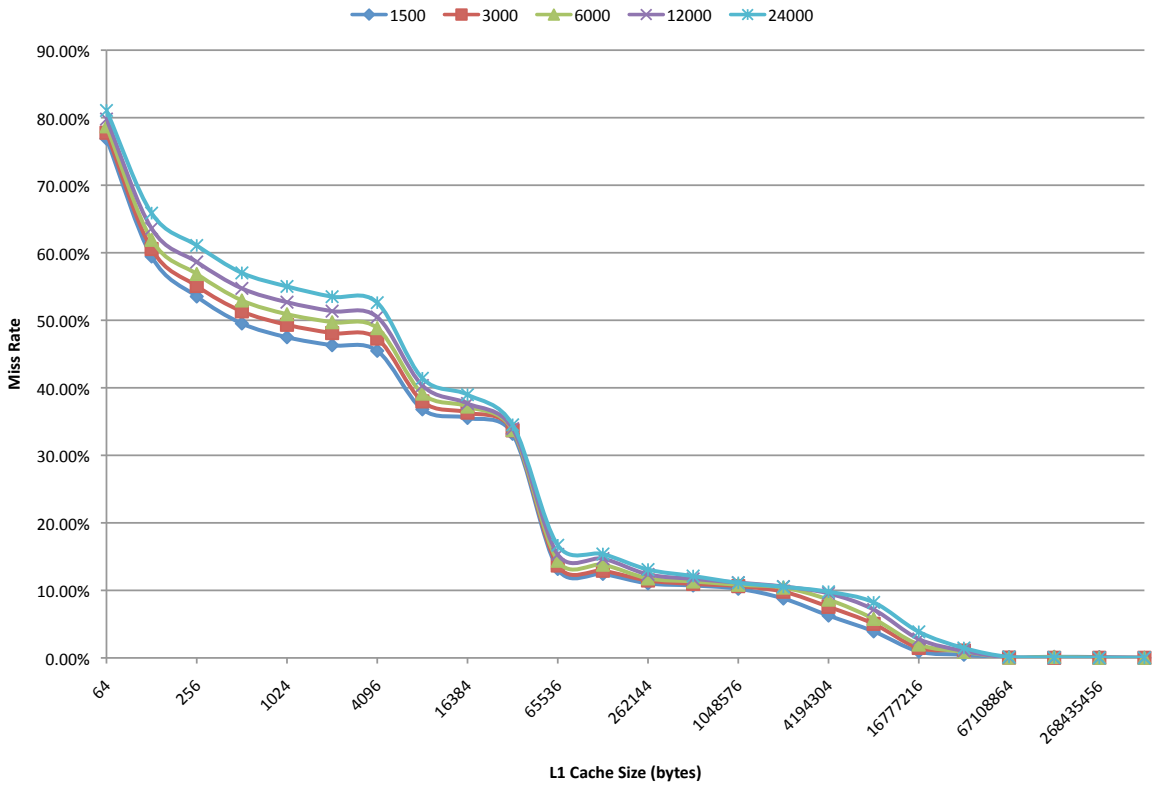


Figure 6.7: Measured cache locality in our speech recognizer for a variety of beam width sizes. To measure locality, we simulate a single-level memory hierarchy and then sweep L1 cache sizes from 64 bytes to 512 mBytes. Our experiments use a direct mapped cache with 32 byte cache lines.

Figure 6.7 presents cache misses rates for five beam width values and a variety of L1 sizes. To measure the working set size, we simulate only a single level of caching. The memory subsystem behavior was recorded while speech recognizer decoded 60 seconds of audio for each beam width. As expected, the simulation shows a higher miss rate for large beam widths across all cache sizes. In addition, there is a sharp knee in the miss rate curve with a 64 kByte cache. With a 64 kByte cache, the miss rate drops from approximately 35% to 15%. Beyond the 64 kByte, the next large dip in miss rate curve occurs with an 8 to 16 mByte cache. At this point, the miss rate drops from approximately 10% to around

1% for all beam widths. While a working set of 8 to 16 mBytes is too large for conventional L1 caches, it is approximately the same size as the L3 cache on today's commodity desktop processors.

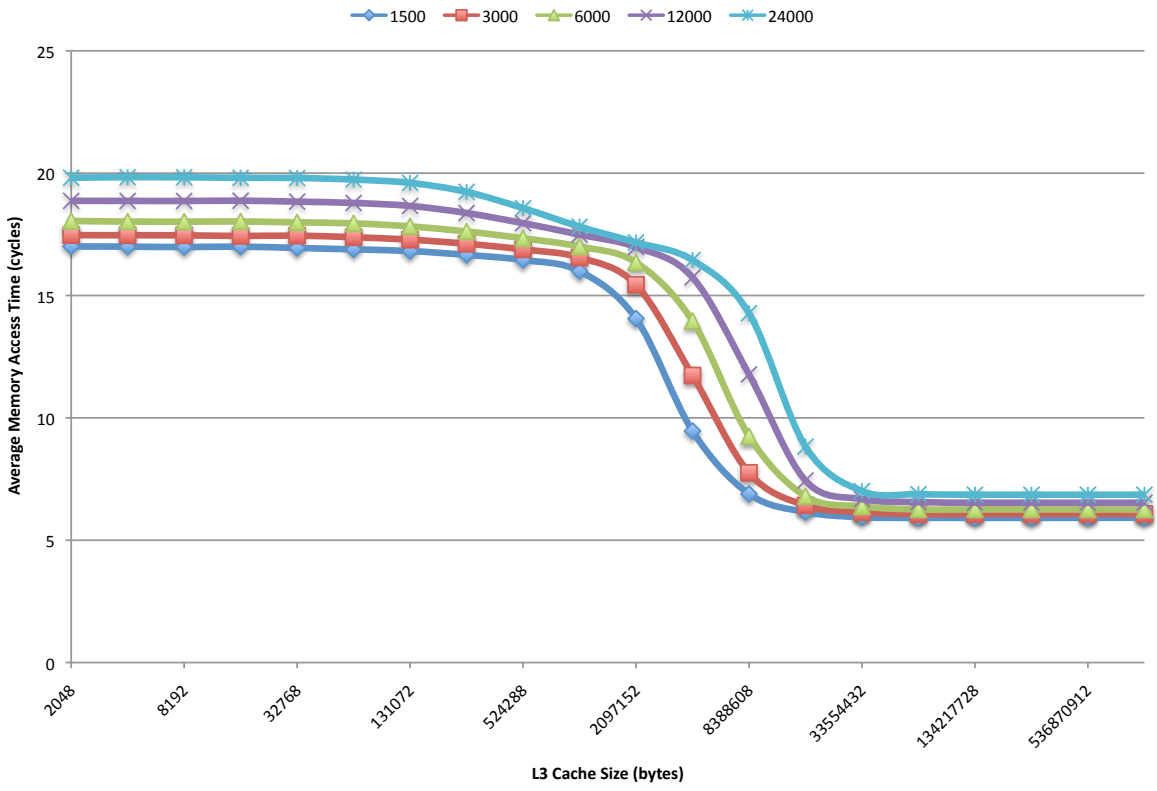


Figure 6.8: Average memory access time as function L3 cache size and beam width sizes. We use 16 kByte 4-way set-associative L1 cache with 1 cycle access latency. Our L2 cache is 256 kBytes with 8-way associativity and has a 10 cycle access latency while our L3 cache has 32-way associativity and 30 cycle access latency. All 3 levels of the memory hierarchy have 32 byte cache lines.

As our working-set simulation studies found, the L3 caches included in today's commodity desktop processors can hold enough of the working set to drop miss rates to below 1%. We investigated the average memory access time (AMAT) as a function of beam width and L3 cache size. As shown in Figure 6.8, we model a 3 level cache hierarchy and record average memory access time. With a multi-level cache hierarchy, moving from a 2 mByte to 8 mByte cache improves the AMAT by approximately a factor of 3× for all beam widths.

To verify our locality simulation results, we ran our speech recognizer on an Intel i7 920 (8 mByte L3) and recorded performance with all levels of the cache hierarchy, enabled and disabled. The results of this experiment are presented in Table 6.1, and, as expected, performance is significantly reduced with caches disabled. The ratio between cached and

Beam width	Caches Disabled	Caches Enabled	Cached vs Uncached Ratio
1500	2.6×	17.8×	6.9
3000	2.3×	12.8×	5.6
6000	1.2×	8.5×	7.1
12000	1.1×	5.4×	4.9
24000	1.0×	3.4×	3.4

Table 6.1: Speech recognizer performance (results presented in real-time factor) with caches enabled and disabled on an Intel i7 920. Results generated using 60 seconds of audio from the Wall Street Journal 5k corpus. All levels of the cache hierarchy were disabled by setting the CD flag and clearing the NW flag of CR0 and clearing the memory type range registers [75]

uncached performance drops with large beam widths indicating that a significant number of cache misses occur when caches are enabled. This reflects the fact that the working set can not fit the cache hierarchy when the recognizer is run with a large beam width. While the ratio of cached to uncached performance is not a linear function of beam width, the general trend reflects the fact that larger beam widths result in a larger working set.

Our simulation results clearly demonstrate that locality exists in our speech recognizer, and it can be captured by reasonably sized caches. In contrast, past approaches to speech recognition on GPUs were unable to exploit the locality in the application due to the lack of a traditional cache hierarchy [36, 35]. The GPU-based approaches to speech recognition required complicated algorithmic reformulation to regularize data structures for efficient use of GPU memory bandwidth. Our simulation results directed our study to use speech recognition algorithms and data structures that are work-efficient in order to exploit a cache hierarchy.

6.3 Accelerated Speech Recognition Kernels

```
def GMM(In, Mean, Var, Out, Idx, n):
  for i in range(0,n):
    for f in range(0,39):
      for m in range(0,16):
        ii = Idx[i];
        Out[ii][m] += (In[f]-Mean[ii][f][m])*(In[f]-Mean[ii][f][m])*(Var[ii][f][m]);
```

Figure 6.9: GMM-based observation probability evaluation kernel in Python for TFJ acceleration

We evaluate the observation probability of labels in the acoustic model using a GMM. The GMM is a computationally intense kernel that consumes 60% of the runtime in our portable recognizer. As shown in Figure 6.9, the GMM computation is a regular dense loop-nest; however, several arrays are indexed with an indirect map as a beam search prunes improbable labels.

```
def step4(..):
    for i in range(0,num):
        thisStateID = endsQ_stateID[i];
        endwrdStTStep = endsQ_wrdStTStep[i];
        endsWordID  = Chain_wpID[ thisStateID ];
        endsFwdProb = Chain_fwrdProb[ thisStateID ];
        prev_likelihood = endsQ_likelihood[i];
        lumpedConst = prev_likelihood + endsFwdProb;
        thisOffset  = bffst[ endsWordID ];
        thisbSize   = bSize[ endsWordID ];

        for b in range(0,thisbSize):
            w = nxtID[b+thisOffset];
            t = prob[b+thisOffset] + lumpedConst;
            bigramBuf[w] = t;
            lock(step4_lck[w]);
            if(bigramBuf[w] < likelihood[w]):
                likelihood[w] = t;
                updateIndices[w] = i;
            unlock(step4_lck[w]);
```

Figure 6.10: Across-word traversal kernel represented in Python for TFJ acceleration

Our LLM knowledge network has two types of transitions: within-word and across-word. We use Chong's [36] specialized data-layout to represent the first, middle, and last states of the triphone chains. His triphone chain layout, used for word pronunciation, enables efficient use of memory bandwidth on parallel platforms.

The within-word kernel used in the LLM representation operates on the first and middle states of the triphone chain to update the middle and last states. Profiling experiments of our portable speech recognizer showed the within-word kernel consumed less than 5% of the runtime. We therefore decided against further optimizations.

In contrast, the across-word kernel shown in Figure 6.10 consumes 25% of the runtime. The across-word kernel operates on the last states of triphone chains to update the first states. Parallelizing this kernel requires fine-grained synchronization. This is because multiple end states transition to the same next state and could result in interleaved updates and thus a race condition. Fine grained synchronization is efficiently handled by TFJ as the custom hardware generator has support for multiword atomic memory operations.

To demonstrate the efficient parallel codes generated by TFJ, we run our hybrid Python and C++ speech recognizer on a conventional PC desktop. As mentioned in Chapter 3, we have a JIT compilation backend on x86 processors for TFJ in a conventional Python install. We obtain a real-time factor [2] (RTF) of 0.0625 when our hybrid Python/C++ speech recognizer runs on a quad-core 3.4 GHz Intel i7-2600 using TFJ. This corresponds to 16× faster than real-time performance. To contextualize the TFJ on a desktop results, the same LLM based recognizer with manually parallelized C++ runs at 0.05 RTF, while the hybrid recognizer without TFJ runs at greater than 330 RTF. This is due to the low performance of the Python interpreter.

[2]The real-time factor (RTF) metric is the ratio of the number of seconds required to process one-second of speech input. An RTF of less than 1.0 connotes better than real-time performance.

6.4 Hardware Verification

In order to validate our custom hardware design points, we modified the software speech recognizer running on our workstation to interface with the Synopsys VCS logic simulator. This configure allows us to selectively verify that our kernel accelerators function properly; unfortunately, each WSJ utterance took well over a day to run, making verification of the WSJ5k corpus unfeasible using logic simulation alone.

LUTs	DSP48s	18 kBit RAMs	36 kBit RAMs
16401	10	7	18

Table 6.2: FPGA statistics for our prototype speech recognition accelerator on the Xilinx Zynq xc702 FPGA. The speech recognition solution prototyped on the FPGA platform has two GMM accelerators and two across-word accelerators. The accelerators run at 50 MHz on the Zynq FPGA.

To remedy our slow simulation runtimes and to provide more credibility to our automatically generated custom hardware solutions, we ported our simulation infrastructure to the Xilinx Zynq SoC FPGA [145]. As mentioned in Section 5.3.2, the Xilinx Zynq platform provides both reconfigurable logic and interconnect (similar to a traditional FPGA), along with two ARM Cortex-A9 processors. The FPGA statistics for the FPGA prototype are shown in Table 6.2. We ran the accelerator logic on a Xilinx Zynq SoC+FPGA, which sped-up our verification process by approximately 465× over that of logic simulation. This enabled 330 utterances (2404 seconds of audio) to run on our simulated hardware in just under 42 hours.

6.5 Hardware and Software Evaluation

We trained the acoustic model that we used in Parallex with HTK [147] using the speaker independent training data from the Wall Street Journal 1 corpus. The acoustic model has 3,006 16-mixture Gaussians, while the LLM recognition network has 123,246 states and 1,596,884 transitions. We set the language model weight to 15. The word error rate of our TFJ accelerated solution is 11.4%, which matches the error rate of the state-of-the-art Parallex recognizer.

6.5.1 Rocket-Hwacha Data-Parallel Processor

We used the in-order decoupled RISC-V 5-stage Rocket processor as our baseline CPU [140]. To evaluate data-parallel solutions, we used the Hwacha data-parallel accelerator with Rocket as its scalar control processor. The Hwacha data-parallel accelerator integrates ideas from both vector-thread [86] and conventional data-parallel processors to achieve high performance and energy efficiency. TFJ was used to generate optimized implementations for Rocket and Hwacha. The resulting kernels were compiled using GCC 4.6.1.

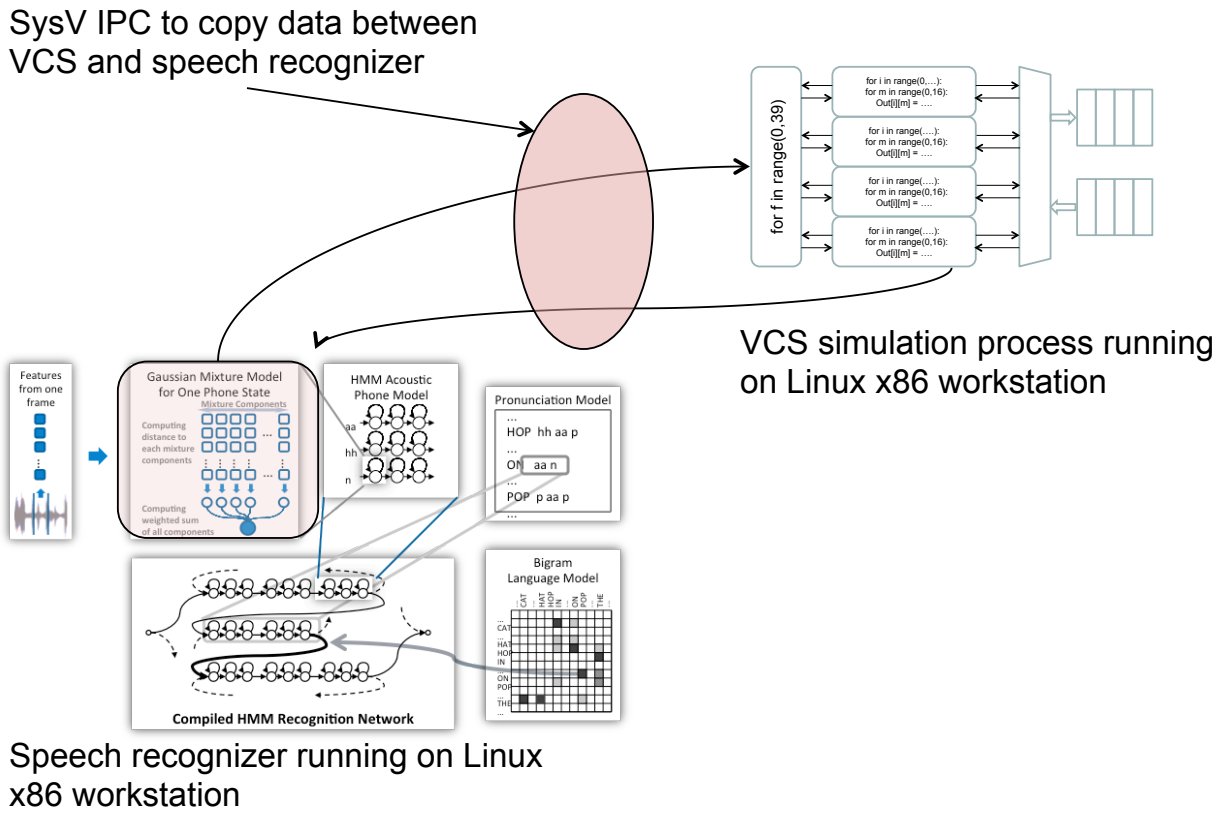
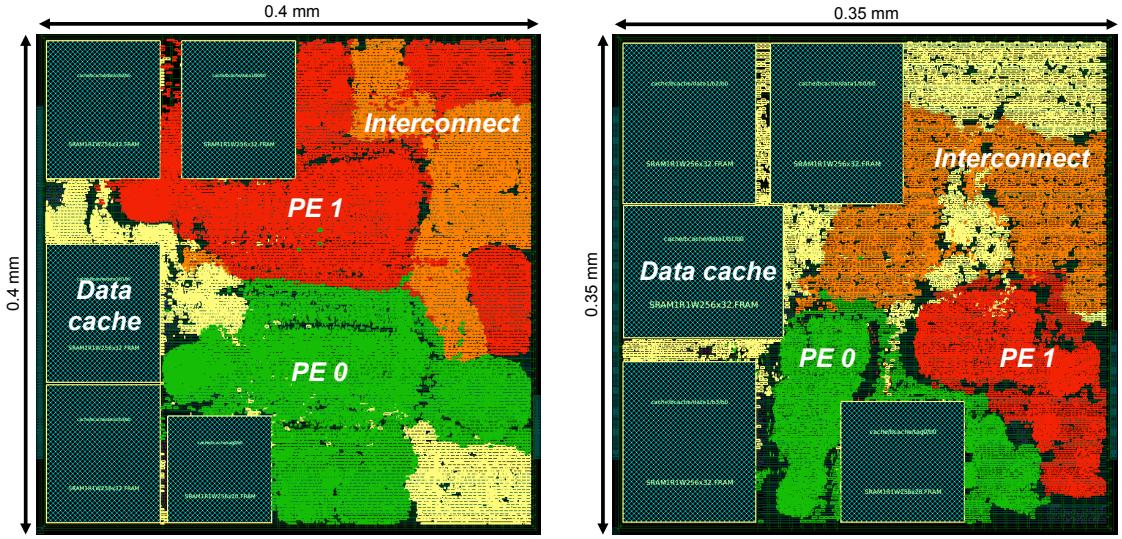


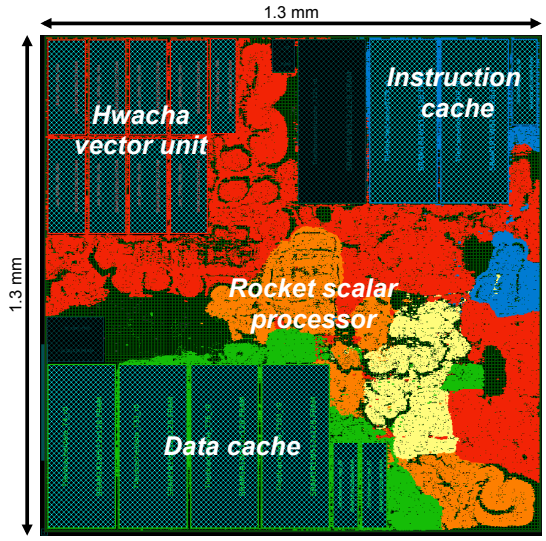
Figure 6.11: Speech recognition hardware verification scheme using our decoder and Synopsys VCS. Speech recognizer diagram based adapted from Chong.

6.5.2 VLSI Flow

We targeted TSMC's 45nm GP CMOS library using a Synopsys-based ASIC toolchain: Design Compiler for logic synthesis, IC Compiler for place-and-route, and PrimeTime for power measurement. Following best industrial practice [20], we used logic simulation to extract cycle counts and detailed circuit-level simulation to record power. The GMM and across-word traversal accelerators have direct-mapped 4 kByte caches ("baseline" caches from Section 5.2.4) share a 256 kByte L2 cache, and do not include a L1 cache. The cache sizes were chosen using the working-set simulation presented earlier in Figure 6.7. The processors with which we compare have a 32 kByte 4-way set-associative L1 data-cache, a 16 kByte 2-way set-associative instruction cache, and a 256 kByte 8-way set-associative L2 cache. The SRAM macros used for all caches were generated by Cacti 6.0 [108]. We used DRAMSim2 to model main memory latency with a DDR3-based memory subsystem [122].



(a) GMM accelerator (b) Across-word search accelerator



(c) Hwacha data-parallel processor

Figure 6.12: VLSI layouts. Note: scale listed for each accelerator.

Design	Features
Hwacha data-parallel processor (Figure 6.12c)	1 Single-Precision Fused Multiply-Adder 8 Single-Precision Comparators 1 Double-Precision Fused Multiply-Adder 8 Double-Precision Comparators 833 MHz Max 213188 Gates, 1.7 mm^2
GMM accelerator (Figure 6.12a)	2 Single-Precision Multipliers 2 Single-Precision Adders 920 MHz Max 38294 Gates, 0.16 mm^2
Across-word search accelerator (Figure 6.12b)	2 2 Single-Precision Adders 2 Single-Precision Comparators 860 MHz Max 19544 Gates, 0.13 mm^2

Table 6.3: Design statistics for the VLSI layouts presented in Figure 6.12

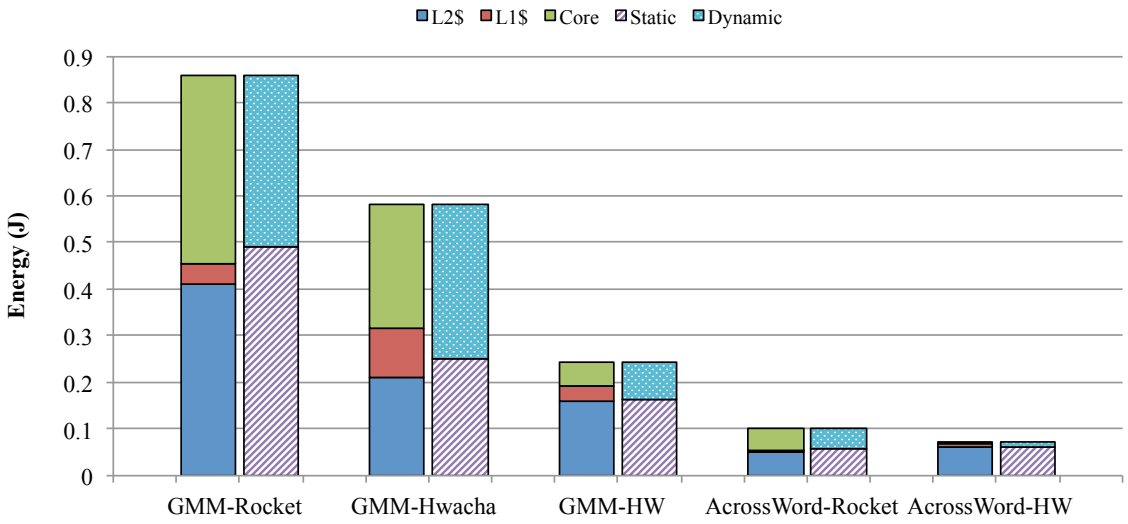


Figure 6.13: Energy breakdown of GMM and across-word search kernels running on programmable and fixed-function accelerators.

6.5.3 VLSI results

To make our study complete, we have included images of VLSI layout from IC Compiler for our data-parallel processor, GMM accelerator, and across-word traversal accelerator (see Figure 6.12). More detailed statistics of each design are listed in Table 6.3. Figure 6.13 shows the detailed energy breakdown of GMM and across-word search kernels running on each of our platforms.

	Rocket	Rocket + Hwacha	Custom HW
GMM	0.86 J	0.58 J	0.24 J
Word-to-word	0.15 J	0.15 J	0.09 J
Rest of system	0.2 J	0.2 J	0.2 J
Complete system	1.21 J	0.93 J	0.54 J

Table 6.4: Energy results for WSJ clip 441c0201 (6.07 seconds) with TFJ generated solutions. The "rest of system" category includes all kernels not accelerated with TFJ.

The energy results of our study are presented in Table 6.4. The Rocket and Rocket+Hwacha based solutions achieve a RTF of 1.0. The automatically generated hardware solutions have a RTF of 0.94. In order to calculate total system power (processor + memory), we assume our memory subsystem will consume 384 mW. These are conservative DDR3 power statistics from a commercial vendor [62]. Table 6.5 estimates the total hours of ASR achievable with the solutions presented in this work.

	Rocket	Rocket + Hwacha	Custom HW
System power	0.20 W	0.15 W	0.09 W
System + memory power	0.58 W	0.54 W	0.47 W
Hours of ASR	9.5 h	10.3 h	11.8 h

Table 6.5: Expected hours of ASR with hardware/software solutions assuming a 20 kJ battery

The bar on the left shows the energy consumption in the core, L1 caches, and the L2 caches. The bar on the right breaks down the energy into dynamic and static portions. Running ASR on fixed-function accelerators is 2.4× and 3.6× more energy efficient than running on a data-parallel processor and a simple scalar processor, respectively. The data-parallel processor is able to reduce the energy consumption in the core as it is able to amortize instruction delivery costs across many elements in a vector. It also runs faster, reducing static energy.

The fixed-function accelerator can further reduce core energy, but the memory system becomes the new energy bottleneck. When we include DRAM energy, the fixed-function accelerator's advantage has been further diminished. The memory energy wall presents a new opportunity for reformulating algorithms and architectures to more efficiently handle through-memory communication [47].

6.6 Summary

We wish to achieve always available mobile ASR untethered to WiFi or 3G networks. To this end, we have proposed and constructed an ASR system with a variety of implementations of the two key kernels used in speech recognition. Our results show the potential energy savings using data-parallel processors and custom hardware for mobile speech recognition. Our results show energy savings of 3.6× over that of a conventional mobile processor and 2.4× over that of a highly-optimized data-parallel processor. We also demonstrated a

productive design-space exploration of potential speech recognition solutions using Three Fingered Jack. Using our best automatically generated solution, given current battery lifetimes, we can run ASR just under 12 hours.

We have demonstrated software and hardware mobile ASR solutions that are capable of running all day while providing real-time performance. We believe our preliminary results clearly demonstrate the benefit of accelerators for mobile ASR and the power of TFJ to generate solutions for multiple platforms.

Chapter 7

Conclusions and Future Work

The challenges associated with scaling high performance uniprocessor systems combined with the rise of mobile systems has caused an explosion of potential implementation platforms. Many applications can be implemented many different ways; however, if we wish to evaluate this design space, we need prototypes for the elements of it. With current tools and design methodologies, constructing functional prototypes for each hardware substrate is a daunting prospect. As developing high performance software on a new platform routinely takes weeks to months when doing initial development, alternative system platforms and algorithmic approaches are rarely explored. This is because each implementation target requires a radically different set of programming and design tools. For example, implementing an application on a data parallel processor requires a complete rewrite in languages such as OpenCL or CUDA.

Specialized hardware is rarely considered due to the even greater challenges of building specializing processing engines, even though performance or energy improvements can be anywhere from 2× to 100× better. Building custom hardware requires first describing the micro-architecture of an accelerator in a high-level hardware description language such as SystemC. As SystemC is effectively a C++ class library, designers can then use C++ abstractions to evaluate architectural ideas. After converging on a final design, the designer must *manually* convert the design into a representation amendable to hardware implementation such as a description in Verilog or VHDL. As the translation process from simulation description to hardware description is done manually, it provides ample opportunity for the introduction of hardware bugs and errors

To address the challenges associated with the explosion of implementation platforms, we have proposed and implemented Three Fingered Jack. Three Fingered Jack is an auto-parallelizing compiler for a subset of the highly productive Python language that allows an end-user to target multicore processors and specialized data parallel processors and to generate custom hardware all from the *same* Python source.

7.1 Contributions

In this section, we summarize the key contributions of this dissertation.

7.1.1 Three Fingered Jack

Parallelizing and optimizing an application on a single parallel platform is challenging; – attempting to target multiple parallel platforms is proportionally more difficult. To address the challenges associated with mapping applications to multiple programmable platforms and generating new custom hardware processing engines, we propose *Three Fingered Jack*.

The Design and Implementation of Three Fingered Jack

Three Fingered Jack extends the SEJITS ideas with key algorithms from parallelizing compilers to target loop-nests. In Chapter 3, we describe the theory behind reordering transformations and their application to automatic parallelization. We then describe the subset of the Python language Three Fingered Jack supports and how unsupported constructs will be executed in the Python interpreter. We detail Three Fingered Jack's approach to loop-nest optimization and describe the software architecture of the tool. The software architecture details our approach to partitioning the framework into a front-end, reordering engine, and code generation backends. We conclude Chapter 3 with the details of our machine code generator for both conventional multicore processors and specialized data parallel processors.

The Evaluation of Three Fingered Jack's Software Backends

Chapter 4 evaluates the performance results of software solutions generated by Three Fingered Jack on conventional desktop and mobile multicore processors. We first evaluate Three Fingered Jack with four numerical kernels: matrix multiply, diagonal sparse-matrix vector multiply, and back propagation weight adjustment. On these four kernels, Three Fingered Jack generates results that are within 33% to 100% of hand-tuned C++ implementations on desktop platforms.

As the evaluation of kernels is not a thorough enough test of a compiler framework, in Chapter 4, we also evaluate Three Fingered Jack with two small applications in Chapter 4. Our first application, content-aware image resizing, resizes an image by removing "uninteresting" regions using a dynamic programming algorithm. Both mobile and desktop platforms have performances that are within a factor-of-two of the best parallelized C++ implementations on their respective platform. We use Horn-Schunck optical flow as our second benchmark. This numerically intensive application calculates the apparent motion of pixels between two frames of video. On our desktop platform, Three Fingered Jack achieves 42% of the hand parallelized C++ equivalent. The Three Fingered Jack solution has slightly better results on our mobile platform as it achieves 63% of the hand parallelized equivalent program. For both applications, we provide detailed profiling results to demonstrate per kernel performance with respect to the equivalent C++ implementation.

Chapter 4 concludes with a demonstration of how an autotuning framework can be implemented using Three Fingered Jack. We use Three Fingered Jack as the code generation framework with the tuning decision logic implemented in interpreted Python to optimize single precision matrix multiply. Our autotuner increases performance by 1.53× and 1.45×

on our mobile and desktop platforms, respectively over our baseline implementation.

The Design and Evaluation of Three Fingered Jack's Hardware Backend

Chapter 5 describes our approach to generating custom hardware from Python and presents performance results on a modern FPGA platform. We begin the chapter by describing the micro-architecture of Three Fingered Jack's predesigned system hardware template. As building custom hardware with Three Fingered Jack relies on both high-level hardware synthesis techniques to generate processing elements for a specific computation and a predesigned system hardware template, the predesigned system template connects the custom generated processing elements to a memory hierarchy and performs any required synchronization between the processing elements.

After detailing our predesigned system template, we then provide a detailed description of the high-level hardware synthesis flow integrated into Three Fingered Jack's hardware backend. Our description of a high-level synthesis engine focuses on how we exploit the data parallelism found by the reordering engine. In particular, we demonstrate how data parallelism can be used to generate memory-level parallelism when our high-level synthesis flow supports non-blocking memory operations. The ability to support multiple in flight memory operations has the potential to significantly increase memory subsystem throughput. To this end, we present our approach to pipelined memory accesses to structures with non-deterministic access latencies. Our matrix multiply results show processing elements built using our approach have up to 10× more memory-level parallelism than does a naive implementation.

We evaluate solutions generated by the hardware backend of Three Fingered Jack on a FPGA. We compare against a soft-core implementation of a RISC-style microprocessor, this allows a comparison of the performance and area between two design methodologies that use the same implementation technology. We use four kernels to evaluate our automatically generated hardware: vector-vector addition, color conversion, matrix multiply, and Gaussian mixture model evaluation. Compared to an optimized soft-core CPU, Three Fingered Jack generated solutions are up to 12× faster and performance scales with increased hardware resources.

To conclude Chapter 5, we investigate several different implementations of single-precision matrix multiply to evaluate the performance impact of Three Fingered Jack's hardware optimizations. By selectively enabling optimizations, we are able to generate non-blocked matrix multiply accelerators with performances from 3 MFlops/sec to 57 MFlops/sec. Likewise, we are able to generate blocked matrix multiply accelerators with performance from 15 MFlops/sec to 98 MFlops/sec without modifying any Python source.

7.1.2 A Case Study in Mobile Speech Recognition using Three Fingered Jack

Finally, we use Three Fingered Jack to explore different implementations of large vocabulary continuous speech recognition in Chapter 6. This case study demonstrates the

effectiveness of our approach to hardware and software codesign using large vocabulary continuous speech recognition. We *productively* explore different implementations across all three target platforms supported by Three Fingered Jack.

By using Three Fingered Jack, we show our approach automatically generates software solutions that are performance competitive with manually coded implementations despite requiring significantly less source code. We also show our automatically generated hardware is significantly more efficient than software-only solutions running on both conventional and vector microprocessors. Specifically, our results show an energy savings of 3.6× over that of a conventional mobile processor and 2.4× over that of a highly-optimized data parallel processor.

7.2 Future Work

Three Fingered Jack has the potential for many future additions; frameworks will succeed with the growth of its user community. If Three Finger Jack begins to be included in mainstream SEJITS implementation infrastructure, it will be adopted by more and more application developers. To that end, TFJ requires minor changes so that it will work with ASP infrastructure. These changes will benefit ASP as well as TFJ because ASP [80] will be improved from the adaptation of TFJs generation framework and its ability to optimize dense loop-nests.

TFJ will benefit from the changes in several ways. First, if TFJ supported a larger subset of Python than it currently does, its expressiveness and ease of use would increase. Second, in the current implementation, users are required to manually predicate computation. Supporting conditional statements through if-conversion would make the framework more approachable to programmers who are not familiar with TFJ. Third, extending TFJ so that whole applications can be automatically tuned in the style of PetaBricks [11] would expand implementation of an autotuner for matrix multiply. Finally, the evaluation of TFJ would greatly benefit from more application studies.

We believe our approach to high-level hardware synthesis has several extension points. Of particular interest is the interaction between reordering transformations and high-level hardware synthesis. We currently use the data parallelism extracted by our reordering engine to generate memory-level parallelism. We do this by producing pipelined scalar operations. As a long-term goal, we would like to generate direct-memory access (DMA) operations from data parallelism rather than from a sequence of scalar operations. Generating DMA operations has two potential benefits. First, it would reduce the amount of state required by the compiler and hardware to track. Second, because the Zynq family of FPGAs includes a high performance DMA engine that can achieve peak memory bandwidth, it would allow Three Fingered Jack to generate much higher performance FPGA solutions.

7.3 Summary

We see platform diversity as one of the fundamental challenges facing application developers today and believe that the problem will only continue to grow. In order to map applications to the most appropriate platform, it is essential that application developers have tools that will allow them to productively explore the hardware platform design space. To overcome the challenges associated with manual design space exploration, this dissertation proposes a tool framework, *Three Fingered Jack*, that allows an end-user to map his or her applications to multicore processors or specialized data parallel processors, or to generate custom hardware all from a *single* Python description.

Our initial results with Three Fingered Jack have been encouraging. In our speech recognition case study, we generated software solutions that are performance competitive with hand parallelized C++ on desktop and mobile platforms, targeted specialized data parallel processors, and automatically constructed highly efficient custom hardware all using the *same* Python source code. We are confident that future applications will use Three Fingered Jack to explore potential implementation platforms.

Bibliography

[1] Cython: C-extensions for python. 2010.

[2] K. Agaram, S.W. Keckler, and D. Burger. A characterization of speech recognition on modern computer systems. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 45–53, 2001.

[3] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, & tools*, volume 1009. Pearson/Addison Wesley, 2007.

[4] John R. Allen and Ken Kennedy. Automatic loop interchange. In *Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, SIGPLAN '84, pages 233–246, New York, NY, USA, 1984. ACM.

[5] Randy Allen and Ken Kennedy. Automatic translation of fortran programs to vector form. *ACM Trans. Program. Lang. Syst.*, 9(4):491–542, October 1987.

[6] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.

[7] 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.

[8] Continuum Analytics. Numpy aware dynamic python compiler using llvm. `https://github.com/numba/numba`.

[9] AnandTech. Lg optimus 2x & nvidia tegra 2 review, 2011.

[10] T. S. Anantharaman and R. Bisiani. A hardware accelerator for speech recognition algorithms. In *Proceedings of the 13th annual international symposium on Computer architecture*, ISCA '86, pages 216–223, 1986.

[11] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: a language and compiler for algorithmic choice. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 38–49, New York, NY, USA, 2009. ACM.

[12] 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 *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 210–213, 2011.

[13] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[14] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[15] Shai Avidan and Ariel Shamir. Seam carving for content-aware image resizing. In *ACM SIGGRAPH 2007 papers*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[16] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, December 1994.

[17] U. Banerjee. Data dependence in ordinary programs. Master's thesis, University of Illinois at Urbana-Champaign, November 1976.

[18] Christopher Francis Batten. *Simplified vector-thread architectures for flexible and efficient data-parallel accelerators*. PhD thesis, Massachusetts Institute of Technology, 2010.

[19] A.J. Bernstein. Analysis of programs for parallel processing. *Electronic Computers, IEEE Transactions on*, EC-15(5):757–763, 1966.

[20] Himanshu Bhatnagar. *Advanced ASIC Chip Synthesis Using Synopsys® Design Compiler® Physical Compiler® and PrimeTime®*. Springer, 2001.

[21] Aart JC Bik. *The software vectorization handbook*. Intel Press Hillsboro, OR, 2004.

[22] J. Bilmes, K. Asanovic, Chee-Whye Chin, and J. Demmel. Using phipac to speed error back-propagation learning. In *Acoustics, Speech, and Signal Processing, 1997. ICASSP-97., 1997 IEEE International Conference on*, volume 5, pages 4153–4156 vol.5, 1997.

[23] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. In *Proceedings of the 11th international conference on Supercomputing*, ICS '97, pages 340–347, New York, NY, USA, 1997. ACM.

[24] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in haskell. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, ICFP '98, pages 174–184, New York, NY, USA, 1998. ACM.

[25] Patrick J Bourke and Rob A Rutenbar. A low-power hardware search architecture for speech recognition. In *Interspeech*, pages 2102–2105, 2008.

[26] R.W Brodersen. Low-voltage design for portable systems, 2002.

[27] Thomas Brox, Andrés Bruhn, Nils Papenberg, and Joachim Weickert. High accuracy optical flow estimation based on a theory for warping. In *Computer Vision-ECCV 2004*, pages 25–36. Springer, 2004.

[28] Thomas Brox and Jitendra Malik. Object segmentation by long term analysis of point trajectories. In *Proceedings of the 11th European conference on Computer vision: Part V*, ECCV'10, pages 282–295, Berlin, Heidelberg, 2010. Springer-Verlag.

[29] João M. P. Cardoso, Pedro C. Diniz, and Markus Weinhardt. Compiling for reconfigurable computing: A survey. *ACM Comput. Surv.*, 42(4):13:1–13:65, June 2010.

[30] Bryan Catanzaro. *Compilation Techniques for Embedded Data Parallel Languages*. PhD thesis, EECS Department, University of California, Berkeley, May 2011.

[31] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 47–56, New York, NY, USA, 2011. ACM.

[32] Bryan Catanzaro, Shoaib Kamil, Yunsup Lee, Krste Asanovic, James Demmel, Kurt Keutzer, John Shalf, Kathy Yelick, and Armando Fox. Sejits: Getting productivity and performance with selective embedded jit specialization. In *First Workshop on Programmable Models for Emerging Architecture at the 18th International Conference on Parallel Architectures and Compilation Techniques*, 2009.

[33] Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. Fast support vector machine training and classification on graphics processors. In *Proceedings of the 25th international conference on Machine learning*, ICML '08, pages 104–111, New York, NY, USA, 2008. ACM.

[34] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44 –54, oct. 2009.

[35] Jike Chong, Ekaterina Gonina, Kisun You, and Kurt Keutzer. Exploring recognition network representations for efficient speech inference on highly parallel platforms. In *11th Annual Conference of the International Speech Communication Association (InterSpeech)*, pages 1489–1492, 2010.

[36] Jike Chong, Youngmin Yi, Arlo Faria, Nadathur Satish, and Kurt Keutzer. Data-parallel large vocabulary continuous speech recognition on graphics processors. In *EAMA*, 2008.

[37] Juan A. Colmenares, Sarah Bird, Henry Cook, Paul Pearce, David Zhu, John Shalf, Steven Hofmeyr, Krste Asanović, and John Kubiatowicz. Resource management in the Tessellation manycore OS. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism (HotPar'10)*, Berkeley, CA, USA, June 2010.

[38] The Scipy community. Sparse linear algebra - scipy.sparse.linalg.cg. `http://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.linalg.cg.html`, 2013.

[39] J. Cong, Bin Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Zhiru Zhang. High-level synthesis for fpgas: From prototyping to deployment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(4):473–491, 2011.

[40] Jason Cong and Zhiru Zhang. An efficient and versatile scheduling algorithm based on sdc formulation. In *Proceedings of the 43rd annual Design Automation Conference*, DAC '06, pages 433–438, New York, NY, USA, 2006. ACM.

[41] Boost Consulting. Boost.python 1.53. `http://www.boost.org/doc/libs/1_53_0/libs/python/doc/index.html`, 2013.

[42] Martyn Corden. Compiling for nehalem. `http://ispass.org/ispass2010/tutorials/Compiling_for_Nehalem_Win_JR_DL.pdf`, 2008.

[43] A. Couts. Ibm bans siri use due to security risks: Should you?

[44] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[45] W.J. Dally, J. Balfour, D. Black-Shaffer, J. Chen, R.C. Harting, V. Parikh, J. Park, and D. Sheffield. Efficient embedded computing. *Computer*, 41(7):27–32, 2008.

[46] G. De Michell and R.K. Gupta. Hardware/software co-design. *Proceedings of the IEEE*, 85(3):349–365, 1997.

[47] Jim Demmel. Communication avoiding algorithms. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, pages 1942–2000, Washington, DC, USA, 2012. IEEE Computer Society.

[48] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V. Leo Rideout, Ernest Bassous, and Andre R. Leblanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9:256–268, 1974.

[49] K. Diefendorff and P.K. Dubey. How multimedia workloads will change processor design. *Computer*, 30(9):43–45, 1997.

[50] P. Diniz, M. Hall, J. Park, B. So, and H. Ziegler. Automatic mapping of c to fpgas with the defacto compilation and synthesis system. *Microprocessors and Microsystems*, 29(2-3):51–62, 2005.

[51] P.R. Dixon, T. Oonishi, and S. Furui. Fast acoustic computations using graphics processors. In *Acoustics, Speech and Signal Processing, 2009. ICASSP 2009. IEEE International Conference on*, pages 4321–4324, 2009.

[52] Joel S. Emer and Douglas W. Clark. A characterization of processor performance in the vax-11/780. In *Proceedings of the 11th annual international symposium on Computer architecture*, ISCA '84, pages 301–310, New York, NY, USA, 1984. ACM.

[53] J. Fisher. Trace scheduling: A technique for global microcode compaction. *Computers, IEEE Transactions on*, C-30(7):478–490, 1981.

[54] Joseph A Fisher, Paolo Faraboschi, and Cliff Young. *Embedded computing: a VLIW approach to architecture, compilers and tools*. Morgan Kaufmann, 2004.

[55] M. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960, 1972.

[56] Python Software Foundation. ast abstract syntax trees. `http://docs.python.org/2/library/ast.html`, 2013.

[57] Rahul Garg and José Nelson Amaral. Compiling python to a hybrid execution environment. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 19–30, New York, NY, USA, 2010. ACM.

[58] Andrew Glew. Mlp yes! ilp no. *ASPLOS Wild and Crazy Idea Session*, 1998.

[59] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical dependence testing. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 15–29, New York, NY, USA, 1991. ACM.

[60] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):12:1–12:25, May 2008.

[61] R. L. Graham. Bounds on multiprocessing anomalies and related packing algorithms. In *Proceedings of the May 16-18, 1972, spring joint computer conference*, AFIPS '72 (Spring), pages 205–217, New York, NY, USA, 1972. ACM.

[62] Marc Greenberg. How much power will a low-power sdram save you, 2009.

[63] Matthias Gries. Methods for evaluating and covering the design space during early design development. *Integr. VLSI J.*, 38(2):131–183, December 2004.

[64] Matthias Gries and Kurt William Keutzer. *Building ASIPs: The MESCAL Methodology*. Springer, 2005.

[65] R.K. Gupta and G. De Micheli. Hardware-software cosynthesis for digital systems. *Design Test of Computers, IEEE*, 10(3):29–41, 1993.

[66] W.A. Havanki, S. Banerjia, and T.M. Conte. Treegion scheduling for wide issue processors. In *High-Performance Computer Architecture, 1998. Proceedings., 1998 Fourth International Symposium on*, pages 266–276, 1998.

[67] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2011.

[68] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, et al. The microarchitecture of the pentium® 4 processor. In *Intel Technology Journal*, 2001.

[69] R. Ho, K.W. Mai, and M.A. Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490–504, 2001.

[70] Berthold KP Horn and Brian G Schunck. Determining optical flow. *Artificial intelligence*, 17(1):185–203, 1981.

[71] Paul Hudak and Mark P Jones. Haskell vs. ada vs. c++ vs awk vs..an experiment in software prototyping productivity. *Yale University Department of Computer Science Technical Report RR-1049*, 1994.

[72] J. Hutchinson, C. Koch, J.Luo, and C. Mead. Computing motion using analog and binary resistive networks. *Computer*, 21(3):52–63, 1988.

[73] Cheng-Tsung Hwang, J.-H. Lee, and Yu-Chin Hsu. A formal approach to the scheduling problem in high level synthesis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 10(4):464–475, 1991.

[74] Wen-Mei W Hwu, Scott A Mahlke, William Y Chen, Pohua P Chang, Nancy J Warter, Roger A Bringmann, Roland G Ouellette, Richard E Hank, Tokuzo Kiyohara, Grant E Haab, et al. The superblock: an effective technique for vliw and superscalar compilation. In *Instruction-Level Parallelism*, pages 229–248. Springer, 1993.

[75] Intel. Intel 64 and ia-32 architectures software developer's manual. volume 3a: System programming guide, part 1. `http://download.intel.com/products/processor/manual/253668.pdf`, 2013.

[76] R. Iyer, S. Srinivasan, O. Tickoo, Zhen Fang, R. Illikkal, S. Zhang, V. Chadha, P.M. Stillwell, and Seung Eun Lee. Cogniserve: Heterogeneous server architecture for large-scale recognition. *Micro, IEEE*, 31(3):20 –31, may-june 2011.

[77] A. Kalavade and E.A. Lee. A hardware-software codesign methodology for dsp applications. *Design Test of Computers, IEEE*, 10(3):16–28, 1993.

[78] S. Kamil, D. Coetzee, and A. Fox. Bringing parallel performance to python with domain-specific selective embedded just-in-time specialization. In *Python for Scientific Computing Conference (SciPy)*, 2011.

[79] S. Kamil, D. Coetzee, and A. Fox. Bringing parallel performance to python with domain-specific selective embedded just-in-time specialization. In *Python for Scientific Computing Conference (SciPy)*, 2011.

[80] Shoaib Kamil. shoibkamil/asp wiki on github. `https://github.com/shoaibkamil/asp/wiki`, 2013.

[81] Shoaib Kamil, Derrick Coetzee, Scott Beamer, Henry Cook, Ekaterina Gonina, Jonathan Harper, Jeffrey Morlan, and Armando Fox. Portable parallel performance from sequential, productive, embedded domain-specific languages. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 303–304, New York, NY, USA, 2012. ACM.

[82] Shoaib Ashraf Kamil. *Productive High Performance Parallel Programming with Auto-tuned Domain-Specific Embedded Languages*. PhD thesis, EECS Department, University of California, Berkeley, Jan 2013.

[83] R. Kavaler, R. Brodersen, T. Noll, M. Lowy, and H. Murveit. A dynamic time warp ic for a one thousand word recognition system. In *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP '84.*, volume 9, pages 375–378, Mar.

[84] Kurt Keutzer, Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. A design pattern language for engineering (parallel) software: merging the plpp and opl projects. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, ParaPLoP '10, pages 9:1–9:8, New York, NY, USA, 2010. ACM.

[85] D. Kondermann. A toolbox to visualize dense image correspondences.

[86] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic. The vector-thread architecture. *Micro, IEEE*, 24(6):84–90, 2004.

[87] Rajeev Krishna, Scott Mahlke, and Todd Austin. Architectural optimizations for low-power, real-time speech recognition. In *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '03, pages 220–231, 2003.

[88] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th annual symposium on Computer Architecture*, ISCA '81, pages 81–87, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[89] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '81, pages 207–218, New York, NY, USA, 1981. ACM.

[90] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, ASPLOS IV, pages 63–74, New York, NY, USA, 1991. ACM.

[91] Uwe Lambrette, Bernd Schmandt, Guido Post, and Heinrich Meyr. Cossap-matlab cosimulation. In *Proc. Int. Conf. on Signal Processing Application and Technology (ICSPAT)*, 1995.

[92] Leslie Lamport. The parallel execution of do loops. *Commun. ACM*, 17(2):83–93, February 1974.

[93] Travis Lanier. Exploring the design of the cortex-a15 processor. `http://www.arm.com/files/pdf/at-exploring_the_design_of_the_cortex-a15.pdf`, 2011.

[94] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75 – 86, march 2004.

[95] Yunsup Lee, Rimas Avizienis, Alex Bishara, Richard Xia, Derek Lockhart, Christopher Batten, and Krste Asanović. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 129–140, New York, NY, USA, 2011. ACM.

[96] Stan Liao, Grant Martin, Stuart Swan, and Thorsten Grötker. *System design with SystemC*. Kluwer Academic Publishers, 2002.

[97] Edward C Lin, Kai Yu, Rob A Rutenbar, and Tsuhan Chen. Moving speech recognition from software to silicon: the in silico vox project. In *Interspeech*, 2006.

[98] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, 2008.

[99] John DC Little. A proof for the queuing formula: L= λw. *Operations Research*, 9(3):383–387, 1961.

[100] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[101] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th annual international symposium on Microarchitecture*, MICRO 25, pages 45–54, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[102] Binu Mathew, Al Davis, and Zhen Fang. A low-power accelerator for the sphinx 3 speech recognition system. In *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '03, pages 210–219, 2003.

[103] Michael C. McFarland, Alice C. Parker, and Raul Camposano. Tutorial on high-level synthesis. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, DAC '88, pages 330–336, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[104] Antti P Miettinen and Jukka K Nurminen. Energy efficiency of mobile clients in cloud computing. In *HotCloud*, pages 4–10, 2010.

[105] A. Mitiche and A.-R. Mansouri. On convergence of the horn and schunck optical-flow estimation method. *Image Processing, IEEE Transactions on*, 13(6):848–852, 2004.

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

[107] K. Mukherjee and A. Mukherjee. Joint optical flow motion compensation and video compression using hybrid vector quantization. In *Data Compression Conference, 1999. Proceedings. DCC '99*, pages 541–, 1999.

[108] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. *HP Laboratories*, 2009.

[109] Dorit Naishlos. Autovectorization in gcc. In *Proceedings of the 2004 GCC Developers Summit*, pages 105–118, 2004.

[110] Sergiu Nedevschi, Rabin K. Patra, and Eric A. Brewer. Hardware speech recognition for user interfaces in low cost, low power devices. In *Proceedings of the 42nd annual Design Automation Conference*, DAC '05, pages 684–689, 2005.

[111] J. Nickolls and W.J. Dally. The gpu computing era. *Micro, IEEE*, 30(2):56–69, 2010.

[112] Rishiyur S. Nikhil. Abstraction in hardware system design. *Commun. ACM*, 54(10):36–44, October 2011.

[113] NVIDIA. Chimera: The nvidia computational photography architecture. `http://www.nvidia.com/docs/IO/116757/Chimera_whitepaper_FINAL.pdf`, 2013.

[114] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Commun. ACM*, 29(12):1184–1201, December 1986.

[115] D.S. Pallett. A look at nist's benchmark asr tests: past, present, and future. In *Automatic Speech Recognition and Understanding, 2003. ASRU '03. 2003 IEEE Workshop on*, pages 483 – 488, nov.-3 dec. 2003.

[116] A. Papakonstantinou, K. Gururaj, J.A. Stratton, Deming Chen, J. Cong, and W.-M.W. Hwu. Fcuda: Enabling efficient compilation of cuda kernels onto fpgas. In *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on*, pages 35–42, 2009.

[117] Behrooz Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, Inc., New York, NY, USA, 2nd edition, 2009.

[118] Lutz Prechelt. Are scripting languages any good? a validation of perl, python, rexx, and tcl against c, c++, and java. *Advances in Computers*, 57:205–270, 2003.

[119] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 4–13, New York, NY, USA, 1991. ACM.

[120] M. Ravishankar. Parallel implementation of fast beam search for speaker-independent continuous speech recognition, 1993.

[121] Stuart Robinson. Cellphone energy gap: Desperately seeking solutions, 2009.

[122] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. Dramsim2: A cycle accurate memory system simulator. *Computer Architecture Letters*, 10(1):16–19, 2011.

[123] Alex Rubinsteyn, Eric Hielscher, Nathaniel Weinman, and Dennis Shasha. Parakeet: a just-in-time parallel accelerator for python. In *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*, HotPar'12, pages 14–14, Berkeley, CA, USA, 2012. USENIX Association.

[124] Richard M. Russell. The cray-1 computer system. *Commun. ACM*, 21(1):63–72, January 1978.

[125] Reinhard C Schumann. Design of the 21174 memory controller for digital personal workstations. *Digital Technical Journal*, 9:57–70, 1997.

[126] D. Sheffield, M. Anderson, and K. Keutzer. Automatic generation of application-specific accelerators for fpgas from python loop nests. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 567 –570, aug. 2012.

[127] David Sheffield, Michael Anderson, and Kurt Keutzer. Automatic generation of application-specific accelerators for fpgas from python loop nests. Technical Report UCB/EECS-2012-203, EECS Department, University of California, Berkeley, Oct 2012.

[128] John Paul Shen and Mikko H Lipasti. *Modern processor design: fundamentals of superscalar processors*. McGraw-Hill Higher Education, 2nd edition, 2005.

[129] Desh Singh. Higher-level programming abstractions for fpgas using opencl. In *Workshop on Design Methods and Tools for FPGA-Based Acceleration of Scientific Computing*, 2011.

[130] Satnam Singh. Computing without processors. *Commun. ACM*, 54(8):46–54, August 2011.

[131] Narayanan Sundaram, Thomas Brox, and Kurt Keutzer. Dense point trajectories by gpu-accelerated large displacement optical flow. In *Proceedings of the 11th European conference on Computer vision: Part I*, ECCV'10, pages 438–451, Berlin, Heidelberg, 2010. Springer-Verlag.

[132] D Sylvester and K. Keutzer. Getting to the bottom of deep submicron. In *Computer-Aided Design, 1998. ICCAD 98. Digest of Technical Papers. 1998 IEEE/ACM International Conference on*, pages 203–211, 1998.

[133] Zhangxi Tan, Andrew Waterman, Rimas Avizienis, Yunsup Lee, Henry Cook, David Patterson, and Krste Asanović. Ramp gold: an fpga-based architecture simulator for multiprocessors. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 463–468, New York, NY, USA, 2010. ACM.

[134] C to Verilog. Automating circuit design. `http://www.c-to-verilog.com`, 2012.

[135] K. Van Rompaey, D. Verkest, I. Bolsens, and H. De Man. Coware-a design environment for heterogeneous hardware/software systems. In *Design Automation Conference, 1996, with EURO-VHDL '96 and Exhibition, Proceedings EURO-DAC '96, European*, pages 252–257, 1996.

[136] G. van Rossum. Whats new in python 3.0. `http://docs.python.org/3.0/whatsnew/3.0.html`.

[137] 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. *SIGARCH Comput. Archit. News*, 38(1):205–218, March 2010.

[138] R.A. Walker and S. Chaudhuri. Introduction to the scheduling problem. *Design Test of Computers, IEEE*, 12(2):60 –69, summer 1995.

[139] Heng Wang, A. Klaser, C. Schmid, and Cheng-Lin Liu. Action recognition by dense trajectories. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pages 3169–3176, 2011.

[140] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. The risc-v instruction set manual, volume i: Base user-level isa. Technical Report UCB/EECS-2011-62, EECS Department, University of California, Berkeley, May 2011.

[141] M. Weinhardt and W. Luk. Pipeline vectorization. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(2):234–248, 2001.

[142] Manuel Werlberger, Thomas Pock, Markus Unger, and Horst Bischof. Optical flow guided tv-l1 video interpolation and restoration. In *Proceedings of the 8th international conference on Energy minimization methods in computer vision and pattern recognition*, EMMCVPR'11, pages 273–286, Berlin, Heidelberg, 2011. Springer-Verlag.

[143] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[144] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[145] Xilinx. Ug585: Zynq-7000 all programmable soc technical reference manual. `http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf`, 2013.

[146] Xilinx. Ug586: 7 series fpgas memory interface solutions v1.9 and v1.9a. `http://www.xilinx.com/support/documentation/ip_documentation/mig_7series/v1_9/ug586_7Series_MIS.pdf`, 2013.

[147] Steve Young, Gunnar Evermann, Dan Kershaw, Gareth Moore, Julian Odell, Dave Ollason, Valtcho Valtchev, and Phil Woodland. The htk book. *Cambridge University Engineering Department*, 3, 2002.