Hardware/Software Codesign for Mobile Speech Recognition

David Sheffield,Michael Anderson,Yunsup Lee,Kurt Keutzer

Department of EECS, University of California, Berkeley, USA
{dsheffie,mjanders,yunsup,keutzer}@eecs.berkeley.edu

Abstract

In this paper, we explore high performance software and hardware implementations of an automatic speech recognition system that can run locally on a mobile device. We automate the generation of key components of our speech recognition system using Three Fingered Jack, a tool for hardware/software codesign that maps computation to CPUs, data parallel processors, and custom hardware. We use Three Fingered Jack to explore energy and performance for two key kernels in our speech recognizer, the observation probability evaluation and across-word traversal.

Through detailed hardware simulation and measurement, we produce accurate estimates for energy and area and show a significant energy improvement over a conventional mobile CPU.

Index Terms: hardware, compiler, speech recognition, energy-efficient

10.21437/Interspeech.2013-182

1. Introduction

Automatic speech recognition (ASR) has become an enabling technology on todays mobile devices. However, current solutions 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 technology innovations[1]. Given the mobile energy constraint, we seek to understand the energy consumption of various speech recognition approaches and provide a productive means for designing energy efficient ASR solutions.

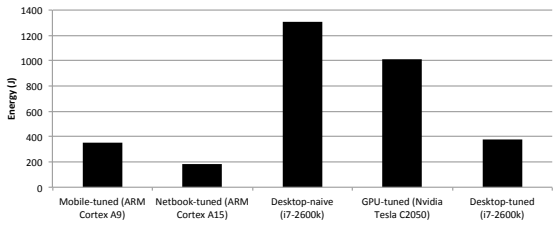


Figure 1: Energy consumption for 60s of ASR on a menagerie of commercial platforms. All devices achieve real-time performance on Wall Street Journal 5k corpus[2]. Energy recorded using a "watts up? PRO" power meter.

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[3]. The large range in 3G energy consumption is due 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. Moreover, cloud-based solutions also do not work in low connectivity environments and thus are incapable of providing robust service in a variety of environments.

Another option is to employ an ASR solution that runs locally on the mobile device. In order to evaluate software-based 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 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. On the plus side, this solution works with or without connectivity.

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 [4, 5, 6, 7, 8, 9, 10]. These approaches claim performance or energy benefits of 10 to 100× over 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.

In this paper we explore high performance software and hardware implementations of an ASR system that can run locally on a mobile device. We automate the generation of key components of our speech recognition system using *Three Fingered Jack* (TFJ) [11], a vectorizing compiler for CPUs, data parallel processors, and custom hardware. 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 vector processor, respectively.

1.1. Related Work

To our knowledge, no prior research has used automatic hardware/software co-design tools to explore energy trade-offs among different ways of providing ASR. Previous work has shown the performance benefits of data-parallel processors [12, 13, 14] or power and energy benefits of custom hardware

[8, 6, 7, 9, 10] for ASR.

2. Three Fingered Jack

As shown in Figure 2, the design space for speech 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 hardware crafted solely for speech recognition. Constructing functional prototypes of ASR systems for each hardware substrate is a daunting prospect as each 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 register-transfer language (RTL) such as Verilog or VHDL. Building a single accurate speech recognition design point is challenging. As a consequence, researchers are unable to fully explore the solution design space.

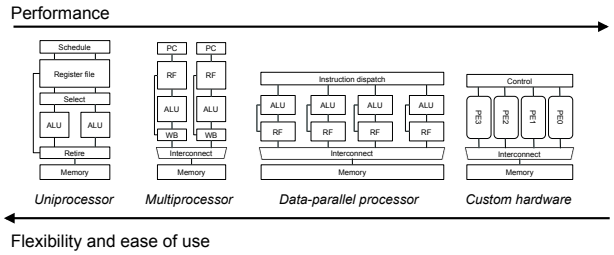


Figure 2: The design space of hardware accelerators: Performance, for a given task, increases from left to right while programability decreases. Figure adapted from Fisher [15]

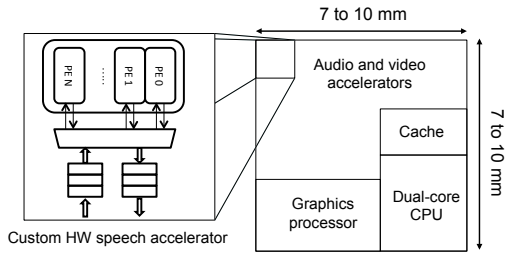


Figure 3: A system on a chip: a large fraction of the SoC die area is dedicated to custom accelerators, such as video encoders/decoders or image processing. The speech recognition solutions explored in this paper are intended be a small logic block (under $5mm^2$) on a SoC.

We use *Three Fingered Jack* (TFJ) to explore implementations of speech kernels across multicore CPUs, data-parallel processors, and custom generated hardware. TFJ applies ideas from optimizing compilers, such as dependence analysis and reordering transformations[16], to a restricted set of Python loop nests. It does this to uncover parallelism. Once parallelism has been discovered, TFJ is able to map it to a number of potential platforms.

The TFJ compilation process begins with a dense loop nest specified in Python using NumPy arrays. The TFJ front-end then generates an intermediate XML representation of the abstract syntax tree (AST) that is interpretable by TFJ's optimization engine. TFJ then uses dependence analysis to compute valid partial orderings of the loop-nest to unlock parallelism. After extracting parallelism, separate backends are used to generate code for data-parallel processors (vector processors) and multicore processors. It also generates Verilog hardware descriptions for custom processing engines. The *multi-processor and data-parallel processor* backends generate C++ with intrinsics for data-parallel processors and calls to low-level threading libraries (such as pThreads) for multiprocessor support. TFJ also supports LLVM [17] just-in-time (JIT) compilation on x86 multicore. This allows for software development on a conventional PC workstation as TFJ is a loadable extension to a Python 2.7 installation.

The *custom hardware backend* automatically generates processing engines (PEs) to be included on a future system on a chip (SoC). The PEs operate like a data-parallel processor; however, instead of executing a generic data-parallel instruction (such as vector-add or vector-load), TFJ PEs execute the entire body of a loop as an application-specific data-parallel instruction. The custom hardware synthesizer maps the LLVM intermediate representation produced by TFJ's reordering engine to Verilog RTL. It does this to create PEs. The configuration of individual PEs can be tuned by allowing for more hardware resources (e.g. more floating-point adders or integer multipliers).

3. Our ASR system

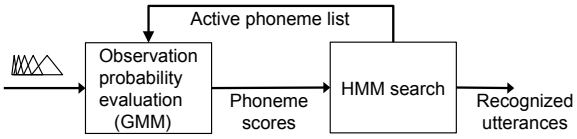


Figure 4: Architecture of our speech recognizer.

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 [13]. As shown in Figure 4, 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. Feature extraction 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 phones 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.

The inference engine at the heart of Parallex uses 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[18]. 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[13].

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[19] 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.

3.1. Accelerated 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 5: 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 5, the GMM computation is a regular dense loop-nest; however, several arrays are indexed with an indirect map as 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: 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 [12] 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, 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. This could result in interleaved updates resulting in a race condition. Fine grained synchronization is efficiently handled by TFJ as the custom hardware generator has support for multi-word 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 section 2, we have a JIT compilation backend on x86 processors for TFJ in a conventional Python install. We obtain a real-time factor [1] (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.

4. Results

4.1. Experiential setup

We trained the acoustic model that we used in Parallex with HTK[20] 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.

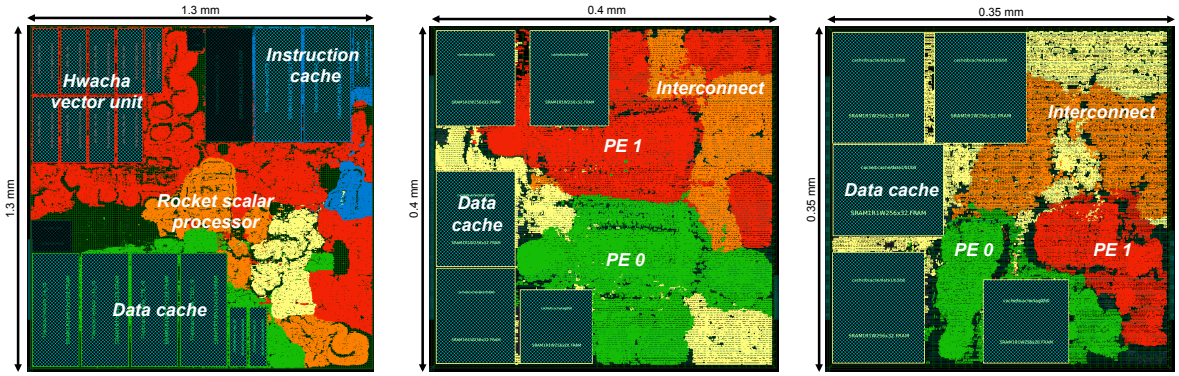
We used the UC Berkeley Rocket processor as our baseline CPU; it is an in-order decoupled 5-stage RISC-V processor [21]. The Rocket processor was used because we were able to fully instrument the processor for energy fine-grained energy measurement. In addition, we believe the energy and performance characteristics of Rocket are very similar to ARM's newest low-power CPU, the ARM Cortex-A7[22]. To evaluate data-parallel solutions, we used the Hwacha vector-core accelerator for Rocket. The Hwacha vector-core integrates ideas from both vector-thread[23, 24] and conventional vector processors[25] to achieve high performance and energy efficiency. TFJ was used to generate optimized C implementations for Rocket and Hwacha. The resulting kernels were compiled for the RISC-V ISA using GCC 4.6.1.

We ensure that all accelerator solutions achieve real-time performance. On the other hand, our solutions are designed for embedded solutions. We see little benefit of over provisioning hardware sources that achieve better than real-time performance.

4.2. Experimental 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. To remedy our slow sim-

[1] 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.



(a) Hwacha vector processor (b) GMM accelerator (c) Within-word search accelerator

Figure 7: VLSI layouts. Note: scale listed for each accelerator.

ulation runtimes and to provide more credibility to our automatically generated custom hardware solutions, we ported our simulation infrastructure to the Xilinx Zynq SoC FPGA[26]. The Xilinx Zynq platform provides both reconfigurable logic and interconnect (like a traditional FPGA) along with two ARM Cortex-A9 processors. We ran the accelerator logic on a Xilinx Zynq SoC+FPGA, which sped-up our verification process by approximately 90× over that of logic simulation. This enabled 330 utterances (2,404 seconds of audio) to run on our simulated hardware in just under 42 hours.

4.3. VLSI results and energy statistics

	Rocket + Hwacha	GMM HW	Within-word HW
Floating-point multipliers	2	2	0
Floating-point adders	2	2	2
Floating-point comparators	2	0	2
Clock frequency (MHz)	833	920	860
Gate count	213188	38294	19544
Total area (mm^2)	1.7	0.16	0.13

Table 1: VLSI statistics for the three designs under consideration.

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[27], 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 and share a 256 kByte L2 cache. The processors we compare with 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 [28]. We used DRAMSim2 to model a DDR3-based memory subsystem[29].

To make our study complete, we have included images of VLSI layout from IC Compiler for our vector processor, GMM accelerator, and across-word traversal. These results are shown in Figures 7a, 7b, and 7c respectively. More detailed statistics of each design are listed in Table 1. Four Rocket processors were required to achieve real-time performance on the GMM kernel. Both Hwacha and custom generated hardware required two processors for real-time performance.

The energy results of our study are presented in Table 2.

	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 2: 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 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[30]. Table 3 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 3: Expected hours of ASR with hardware/software solutions assuming a 20 kJ battery

5. 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 vector 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. We plan to extend our research to larger vocabulary models.

6. References

[1] S. Robinson, "Cellphone energy gap: Desperately seeking solutions," 2009.

[2] D. 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*, nov.-3 dec. 2003, pp. 483 – 488.

[3] A. P. Miettinen and J. K. Nurminen, "Energy efficiency of mobile clients in cloud computing," in *HotCloud*, 2010, pp. 4–10.

[4] 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.*, vol. 9, Mar, pp. 375–378.

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

[6] B. Mathew, A. Davis, and Z. 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*, ser. CASES '03, 2003, pp. 210–219.

[7] R. Krishna, S. Mahlke, and T. 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*, ser. CASES '03, 2003, pp. 220–231.

[8] S. Nedevschi, R. K. Patra, and E. A. Brewer, "Hardware speech recognition for user interfaces in low cost, low power devices," in *Proceedings of the 42nd annual Design Automation Conference*, ser. DAC '05, 2005, pp. 684–689.

[9] E. C. Lin, K. Yu, R. A. Rutenbar, and T. Chen, "Moving speech recognition from software to silicon: the in silico vox project," in *Interspeech*, 2006.

[10] P. J. Bourke and R. A. Rutenbar, "A low-power hardware search architecture for speech recognition," in *Interspeech*, 2008, pp. 2102–2105.

[11] 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*, aug. 2012, pp. 567 –570.

[12] J. Chong, Y. Yi, A. Faria, N. Satish, and K. Keutzer, "Data-parallel large vocabulary continuous speech recognition on graphics processors," in *EAMA*, 2008.

[13] J. Chong, E. Gonina, K. You, and K. Keutzer, "Exploring recognition network representations for efficient speech inference on highly parallel platforms," in *11th Annual Conference of the International Speech Communication Association (InterSpeech)*, 2010, pp. 1489–1492.

[14] J. Chong, K. You, Y. Yi, E. Gonina, C. Hughes, W. Sung, and K. Keutzer, "Scalable hmm based inference engine in large vocabulary continuous speech recognition," in *Multimedia and Expo, 2009. ICME 2009. IEEE International Conference on*, 28 2009-July 3, pp. 1797–1800.

[15] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded computing: a VLIW approach to architecture, compilers and tools*. Morgan Kaufmann, 2004.

[16] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.

[17] 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*, march 2004, pp. 75 – 86.

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

[19] "Pandaboard." [Online]. Available: http://pandaboard.org/

[20] S. Young, G. Evermann, D. Kershaw, G. Moore, J. Odell, D. Ollason, V. Valtchev, and P. Woodland, "The htk book," *Cambridge University Engineering Department*, vol. 3, 2002.

[21] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi, "The risc-v instruction set manual, volume i: Base user-level isa," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2011-62, May 2011. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-62.html

[22] ARM, "Cortex-a7 processor." [Online]. Available: http://www.arm.com/products/processors/cortex-a/cortex-a7.php

[23] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović, "Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators," in *ISCA*, 2011.

[24] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic, "The vector-thread architecture," in *Proceedings of the 31st annual international symposium on Computer architecture*, ser. ISCA '04, 2004, pp. 52–63.

[25] R. M. Russell, "The cray-1 computer system," *Commun. ACM*, vol. 21, no. 1, pp. 63–72, Jan. 1978.

[26] Xilinx, "Ds190 zynq-700 all programmable soc overview."

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

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

[29] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, 2011.

[30] M. Greenberg, "How much power will a low-power sdram save you," 2009. [Online]. Available: https://www.denali.com/en/whitepaper/2009/lpddr2