Measuring the Gap between Programmable and Fixed-Function Accelerators: A Case Study on Speech Recognition

Yunsup Lee, David Sheffield, Andrew Waterman, Michael Anderson, Kurt Keutzer, Krste Asanović

University of California, Berkeley

{yunsup,dsheffie,waterman,mjanders,keutzer,krste}@eecs.berkeley.edu

1. Introduction

As power and energy consumption have become the key design constraint of mobile systems, mobile system-on-chip (SoC) architects have dedicated a progressively larger area budget to custom accelerators: graphics processors, audio/video codecs, and image signal processors abound. Fixed-function accelerators now occupy more than half of the die area of these chips [2], and we foresee this trend only to progress in the near future. Indeed, for applications that have adopted a well-defined standard, constructing dedicated fixed-function accelerators may prove to be the best approach [5]. But for applications that are not standardized and are still in flux, we ask: *Is building fixed-function accelerators the right answer, or should dedicated accelerators become more programmable?* To this end, we first measure the performance and energy consumption gap between programmable and fixed-function accelerators for automatic speech recognition (ASR), an application that is not yet standardized but whose adoption is on on the near horizon. We then analyze these results to gain insight into this problem.

Several hardware-based ASR solutions have been proposed during the last 30 years [8, 9]. 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.

Constructing functional prototypes of ASR systems for both programmable and fixed-function accelerators is a daunting prospect as each target requires a radically different set of programming and design tools. To make the problem of exploring this vast design space tractable—and, more importantly, fair—we use the Three Fingered Jack (TFJ) system [11] to take the same code and automatically generate key components of our speech recognition system to target both accelerators. TFJ takes Python loop nests as an input and can generate C++ code for CPUs, vectorized C++ code for data-parallel processors, and Verilog RTL for fixed-function accelerators.

Through detailed hardware simulations using methodologies described in [3], we produce accurate estimates for energy and performance. We show that for ASR, fixed-function accelerators are 2.4× and 3.6× more energy efficient than a highly-optimized data-parallel processor and scalar processor, respectively. Detailed analysis shows operand delivery consumes approximately the same amount of energy for the fixed-function and programmable solutions. The gap between the scalar processor and the fixed-function hardware is mostly attributable to instruction fetch energy and static energy (since the scalar processor is slower). The data-parallel processor helps close the gap by amortizing instruction delivery energy and running faster, reducing static energy.

2. Setup

An ASR application accepts an utterance as an input waveform and infers the most likely sequence of words and sentences in that utterance. Our ASR system is built on top of the ICSI Parallex decoder [4]. As shown in Figure 1, 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 each 10ms frame. These 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 search (see Figure 2 and 3). The GMM computes the observation probabilities of atomic units of speech in a given acoustic sample. These observation 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 profiling results led us to focus our efforts on the GMM and across-word transition kernels, as they consume 60% and 25% of the run-time, respectively.

2.1 Three Fingered Jack

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 [1], to a subset of Python loop nests. The TFJ compilation process begins with a dense loop nest specified in Python using NumPy arrays. 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 multicore CPUs, data-parallel processors, and custom hardware. The *multicore CPU* backend and the *data-parallel processor* backend generates C++ with intrinsics for its target ar-

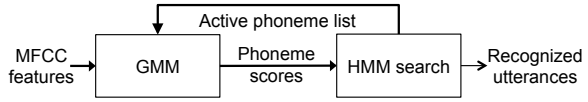


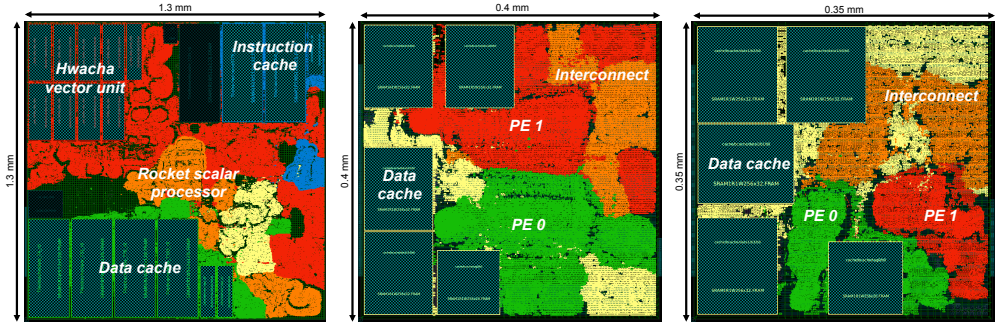
Figure 1: Architecture of our speech recognizer.

```
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 2: GMM-based observation probability evaluation kernel

```
def acrossword(..):
    for i in range(0,num):
        thisStateID = endsQ_stateID[i];
        endsWordID  = Chain_wpID[thisStateID];
        for b in range(0, bSize[endsWordID]):
            w = nxtID[b+bffst[endsWordID]];
            t = prob[b+bffst[endsWordID]] +
                 endsQ_likelihood[i] + Chain_fwrdProb[
                 thisStateID];
            bigramBuf[w] = t;
            lock(step4_lck[w]);
            if(bigramBuf[w] < likelihood[w]):
                likelihood[w] = t;
                updateIndices[w] = i;
            unlock(step4_lck[w]);
```

Figure 3: Across-word search kernel



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

	Features
(a)	1 SFMA, 8 SCMP, 1 DFMA, 8 DCMP 833 MHz Max 213188 Gates, 1.7 mm^2
(b)	2 SMUL, 2 SADD 920 MHz Max 38294 Gates, 0.16 mm^2
(c)	2 SADD, 2 SCMP 860 MHz Max 19544 Gates, 0.13 mm^2

(d) Statistics

Figure 4: VLSI layouts. The scales are listed in the layout for each accelrator. S prefix = single-precision, D prefix = double-precision. FMA suffix = floating-point fused multiply add, ADD suffix = floating-point add, MUL suffix = floating-point multiply, CMP suffix = floating-point compare.

chitecture. The *custom hardware backend* automatically generates fixed-function hardware by mapping the intermediate representation produced by TFJ's reordering engine to a control FSM and collection of functional units. The fixed-function hardware generated by TFJ uses a memory tagging scheme to support a non-blocking memory system. Dependence analysis allows TFJ to schedule many overlapping memory operations. The caches used with TFJ-generated hardware support atomic-memory operations for fine-grained locking in non-data-parallel code, such as the across-word kernel (Figure 3) used in this study.

2.2 Rocket-Hwacha Vector Processor

We used the in-order decoupled RISC-V 5-stage Rocket processor as our baseline CPU [12]. 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 [6, 7] 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.

2.3 VLSI Flow

We targeted TSMC's 45nm GP CMOS library using a Synopsys-based ASIC toolchain. We used logic simulation to extract cycle counts and back-annotated simulation to record power for all platforms. The HW accelerators have direct-mapped 4 KB caches while the processors have a 32 KB 4-way set-associative L1 data cache and a 16 KB 2-way set-associative L1 instruction cache. All configurations include a 256 KB 8-way set-associative L2 cache. Cacti [10] was used to generate SRAM macros.

3. Results and Conclusions

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 accelerator (see Figure 4). More detailed statistics of each design are listed in Table 4(d). Figure 5 shows the detailed energy breakdown of GMM and across-word search kernels running on each of our platforms. 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. Had we included DRAM energy, the fixed-function accelerator's advantage would have been further diminished.

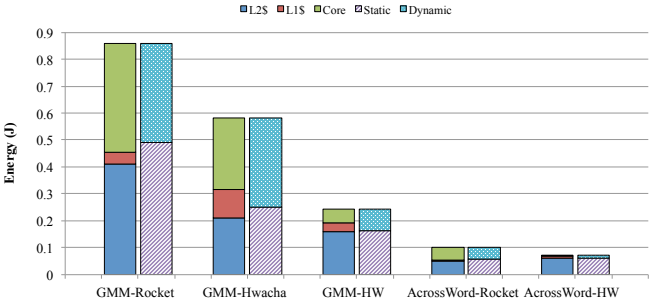


Figure 5: Energy breakdown of GMM and across-word search kernels running on programmable and fixed-function accelerators.

These results guide us to take a holistic approach spanning from applications to hardware when approaching the dark silicon era. We plan to develop new algorithms that specialize and optimize communication along with computation to minimize energy consumption in the memory system. We believe that pattern-based data-parallel processors, which incorporate fixed-function compute units, will be able to close the gap between programmable and fixed-function accelerators.

Acknowledgments

Research funded by DARPA Award Number HR0011-12-2-0016. Approved for public release; distribution is unlimited. The content of this paper does not necessarily reflect the position or the policy of the US government and no official endorsement should be inferred.

References

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.
[2] AnandTech. *LG Optimus 2X & NVIDIA Tegra 2 Review*.
[3] H. Bhatnagar. *Advanced ASIC Chip Synthesis Using Synopsys Design Compiler Physical Compiler and PrimeTime*. Springer, 2001.
[4] J. Chong et al. Exploring recognition network representations for efficient speech inference on highly parallel platforms. IS, 2010.
[5] J. A. Fisher et al. *Embedded computing: a VLIW approach to architecture, compilers and tools*. Elsevier, 2004.
[6] R. Krashinsky et al. The vector-thread architecture. ISCA, 2004.
[7] Y. Lee et al. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. ISCA, 2011.
[8] E. C. Lin et al. Moving speech recognition from software to silicon: the in silico vox project. IS, 2006.
[9] B. Mathew et al. A low-power accelerator for the sphinx 3 speech recognition system. CASES '03, pages 210–219.
[10] N. Muralimanohar et al. Cacti 6.0: A tool to model large caches. *HP Laboratories*, 2009.
[11] D. Sheffield et al. Automatic generation of application-specific accelerators for fpgas from python loop nests. FPL, 2012.
[12] A. Waterman et al. *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA*. Number UCB/EECS-2011-62.