CUSTOM ARITHMETIC FOR HIGH-SPEED, LOW-RESOURCE ASR SYSTEMS

Jonathan Malkin, Xiao Li, Jeff Bilmes

SSLI Lab, Department. of Electrical Engineering
Univ. of Washington, Seattle

ABSTRACT

With the skyrocketing popularity of mobile devices, new processing methods tailored for low-resource systems have become necessary. We propose the use of custom arithmetic, arithmetic logic tailored to a specific application. In a system with all parameters quantized to low precision, such arithmetic can be implemented through a set of small, fast table lookups. We present here a framework for the design of such a system architecture, and several heuristic algorithms to optimize system performance. In addition, we apply our techniques to an automatic speech recognition (ASR) application. Our simulations on various architectures show that on most modern processor designs, we can expect a cycle-count speedup of at least 3 times while requiring a total of only 59kB of ROMs to hold the lookup tables.

1. INTRODUCTION

The burgeoning development of mobile devices has brought about a great need for a more friendly and convenient user interface. Automatic speech recognition (ASR) will undoubtedly become a dominant method since it provides convenient human-machine interaction. However, unlike desktop applications with ample memory and a perpetual power supply, portable devices suffer from limited computational and memory resources and strict power consumption constraints. Consequently, the development of a high-speed, low-resource ASR system becomes crucial to the prevalence of speech technologies on mobile devices.

In the literature, there are many techniques to speed up computation at the software level. Vector quantization (VQ) and sub-vector quantization (SVQ) are some of the most successful methods. They are applicable to both observation features and model parameters [1]. As a result, the expensive state likelihood computation can be more efficiently realized via quantized arithmetic. But VQ or SVQ methods have not been extended beyond the observation features and model parameters, and full precision is used. Also, aside from the initial lookups, all calculations are made at full precision.

The problem can also be approached from the hardware side. With 32-bit computing having reached the embedded market and after years of finding ways to make general purpose chips more powerful, the use of custom logic seems a rather curious choice. For some applications, though, such decisions may be warranted. Floating point operations are quite slow, and require a rather large chip area when implemented. Additionally, speech recognizers typically need a large dynamic range, but do not use the precision of a floating point representation efficiently. Fixed point arithmetic offers only a partial solution. Operations can be much faster and power consumption can be significantly reduced using a fixed point implementation [2, 3, 4]. But in a power-constrained system, memory use is critical, and fixed-point cuts the available dynamic range without addressing the problem of excess or too little precision. Additionally, some operations can still take numerous processor cycles to complete.

Therefore, we present in this work a low-resource custom arithmetic architecture based on high-speed lookup tables. By using custom arithmetic, lookup tables in this case, we can ensure that we represent only values that are likely to be used while providing a fast implementation. Doing so yields a small, compact representation that minimizes memory bandwidth and also removes many costly arithmetic operations. By using data-driven quantization techniques, we can use a ROM-based implementation to provide a substantial increase in speed and with it a decrease in power use. There are limitations, however, in that the variables used in the system must be quantizable to very low precision without a significant deterioration of performance. In [5], a companion paper, we propose several quantization techniques for ASR. This paper is entirely separate: we propose a new table-based, custom arithmetic architecture and heuristic algorithms for optimizing the associate tables.

The rest of the paper is organized as follows. Section 2 discusses the general mechanism of custom arithmetic and its implementation for speech recognition. Section 3 covers the bit-width allocation search algorithms in a large custom arithmetic system. Section 4 mentions our experimental framework and gives specific details of our system's ROMs. Section 5 presents our results where we show that a fast speech recognizer based on lookup tables is a feasible option for many modern embedded systems. Finally, Section 6 discusses the results.

2. A ROM-BASED ARCHITECTURE

2.1. Design Methodology

The first step in implementing a custom arithmetic system is to split all complex expressions into sequences of two-operand operations. We can then express any operation on variables V_i and V_j with result V_k as a function,

$$V_k = F_g(V_i, V_j),$$

where $F_g(\cdot)$ can be an arbitrary arithmetic operation or sequence of operations. An example of such a two-operand sequence of operations is $V_k = (V_i - V_j)^2$.

After splitting all complex operations, the next step is to quantize each free variable in the system. This is done by creating a separate codebook for each variable: the codewords are the quantized values of that variable and the indices of those codewords are sequential integers. For a variable $V_i = x$, we will denote the index of its closest codeword in the codebook as $I_{V_i}(x)$. A codebook indexed with n bits can hold 2^n codewords.

This work was funded by NSF grant ITR/RC-0086032.

Once the codeword values for the inputs and the output are known, all possible values for a function can be precomputed and stored in a table. Each address in the table is determined by the indices of the input operand codewords, and the output is the integer index of the result's codeword. If the output and the two inputs have bit-widths of n_0, n_1, and n_2, respectively, then the table requires a total storage of $n_0 \cdot 2^{n_1+n_2}$ bits. We will refer to the table created for a function $F_g(\cdot)$ as T_{F_g}.

The final step in designing this custom arithmetic system is to replace all floating-point values in the program with the corresponding integer indices, providing approximated computation for all values. The calculation $z = F_g(x, y)$ would therefore become

$$I_{V_k} = T_{F_g}(I_{V_i}, I_{V_j}).$$

By storing the integer index of the codeword of the output value, we reduce the storage requirements of the table. We can also use the output index directly as the input of the next table lookup, so that all data flow and storage are represented as low-precision integers and all complex expressions become a series of simple table lookups.

2.2. Design for ASR

The nature of the equations for decoding with a mixture of Gaussians CHMM (Continuous Hidden Markov Model) makes that task particularly suited to our new method. For instance, part of calculating the Mahalanobis distance for the likelihood evaluation requires the equation given earlier, $V_k = (V_i - V_j)^2$. Rather than a floating point subtraction followed by a floating point multiplication, this entire calculation could be completed with a single table lookup.

Similarly, many ASR systems use log values to store probabilities. The addition of probabilities therefore requires log addition. Representing $z = x + y$ in the log domain gives $\bar{z} = \bar{x} + log(1 + e^{\bar{y}-\bar{x}})$ where $\bar{z}$, $\bar{x}$, and $\bar{y}$ are the logarithms of their unlogged counterparts. Again, this calculation is a function with two input operands meaning that a lookup can replace many operations, including the expensive calls to the log and exponential library functions.

3. BIT ALLOCATION SEARCH ALGORITHMS

So far, we have discussed table construction, but have not addressed how to determine the size of each table. The goal is to come up with a system-wide optimization algorithm to allocate resources among all variables. We aim to find the bit-width allocation scheme which minimizes the cost of the resources while maintaining baseline performance.

The algorithms will be presented for a system with L variables $\{V_i\}_{i=1}^{L}$. We now make the following definitions:

- bw_i – the bit-width of V_i. bw_i can take on any integer value below 32 (the number of bits used for single-precision floating-point representation);
- $\mathbf{fp}$ – when $bw_i = \mathbf{fp}$, V_i is unquantized. Typically, $\mathbf{fp} = \mathbf{32}$;
- $\vec{bw} = (bw_1, bw_2, ..., bw_L)$ – a bit-width allocation scheme;
- $\Delta\vec{bw_i}$ – an increment of 1 bit for variable V_i, where $\vec{bw} + \Delta\vec{bw_i} = (bw_1, ..., bw_i + 1, ..., bw_L)$;
- $wer(\vec{bw})$ – the word error rate (WER) evaluated at $\vec{bw}$;
- $cost(\vec{bw})$ – total cost of resources evaluated at $\vec{bw}$.

Note that the cost function can be arbitrarily defined depending on the specific goals of the allocation. In this paper, we use the total storage of the tables as the cost. Additionally, we define the gradient δ_i as the ratio of the decrease in WER to the increase in cost evaluated in the direction of bw_i.

$$\delta_i(\vec{bw}) \triangleq \frac{wer(\vec{bw}) - wer(\vec{bw} + \Delta\vec{bw_i})}{cost(\vec{bw} + \Delta\vec{bw_i}) - cost(\vec{bw})} \quad (1)$$

Equation (1) reflects the rate of improvement along the bw_i direction. We can extend this definition to the gradient along multiple directions. For example,

$$\delta_{ij}(\vec{bw}) \triangleq \frac{wer(\vec{bw}) - wer(\vec{bw} + \Delta\vec{bw_i} + \Delta\vec{bw_j})}{cost(\vec{bw} + \Delta\vec{bw_i} + \Delta\vec{bw_j}) - cost(\vec{bw})} \quad (2)$$

is the gradient in the joint direction of bw_i and bw_j.

Assuming the baseline system gives a WER of BWER $\triangleq wer(\vec{bw})|_{\vec{bw}=(\mathbf{fp},\mathbf{fp},...,\mathbf{fp})} + \epsilon$ with $\epsilon > 0$ a tolerance for increased error due to quantization, our goal can be interpreted as

$$\vec{bw}^* = \underset{\vec{bw}:wer(\vec{bw})\leq \text{BWER}}{\text{argmin}} \; cost(\vec{bw}) \quad (3)$$

This search space is highly discrete and, due to the effects of quantization noise, only approximately smooth. The combination of these factors means the optimization is very difficult. An exhaustive search for $\vec{bw}^*$, evaluating $wer(\vec{bw})$ and $cost(\vec{bw})$ at every possible $\vec{bw}$, is clearly exponential in L. Even constraining the bit-width of each variable to a restricted range of possibile values gives a very large search space. In the following subsections, we present several heuristics that work well experimentally. The basic idea is that we start with a low-cost $\vec{bw}$ that has low enough a WER that gradients are not meaningless (they provide no information if bit widths are too low) and greedily increase the bit-widths of one or a small group of variables until we find an acceptable solution. This is similar to the method used in [6] to optimize floating-point bit-widths.

3.1. Single-variable quantization

Finding a reasonable starting point is an important part of these algorithms. One logical place to start is with the results of single-variable quantization, since the results of those quantizations are needed in order to create the tables. Specific methods for quantizing the variables are beyond the scope of this paper. For those interested in several quantization techniques specific to ASR that we developed, see [5].

In general, it is expected that the noise introduced into the system by quantizing an additional variable will produce a result that is not better than without the extra variable quantized. For that reason, we take the starting point to be the minimum number of bits needed to quantize a single variable to produce baseline WER results. We call that result m_i, the minimum bit-width of variable V_i. We determine an upper bound M_i based on inspection.

Once we determine the boundaries for each single variable we have constrained our search for an optimal point to the hypercube bounded by $m_i \leq \vec{bw}_i \leq M_i$, $i = 1..L$. It then makes sense to start each of the following algorithms at $\vec{bw}_{init} = (bw_1 = m_1, bw_2 = m_2, ..., bw_L = m_L)$. In all cases, it is assumed that $wer(\vec{bw}_{init}) >$ BWER since the algorithms are designed to stop when they have found a $\vec{bw}$ with a WER as low as the target rate.

3.2. Single-dimensional increment based on static gradients

This is the simplest, most naive algorithm. In it, we allow one variable's bit-width to increment by 1 each iteration, based on static gradients, $\delta_i^s(bw_i) = \delta_i(\vec{bw})|_{\vec{bw}=(\mathbf{fp},\ldots,\mathbf{bw_i},\ldots,\mathbf{fp})}$. The static gradients are obtained from the results of single-variable quantization, with all other variables at **fp**. The algorithm is:

1. Compute the static gradients $\delta_i^s(bw_i)$, $i = 1..L$ at point $\vec{bw}$, based on the WERs of the single-variable quantization. Choose the direction $k = \underset{i}{\text{argmax}}\ \delta_i^s(bw_i)$, and increase bw_k by one if it does not exceed the upper bound M_k;
2. Run a test to get the new $wer(\vec{bw})$. If $wer(\vec{bw}) \leq$ BWER return $\vec{bw}^* = \vec{bw}$ or, if there is no improvement in WER observed, undo the increment before returning $\vec{bw}^* = \vec{bw}$; otherwise repeat step 1.

In this case, the static gradient is used to approximate the true gradient. The gradients can therefore be pre-computed from the single-variable quantization results, but may be inaccurate. This method gives a pre-determined sequence to test, so the task becomes checking the WER at each point until we have a satisfactory one.

3.3. Single-dimensional increment with dynamic gradients

This algorithm, like the first one, allows only single-dimensional increments, but it uses the true gradient $\delta_i(\vec{bw})$ as a measure of improvement, requiring online evaluation. The steps are:

1. Evaluate the gradients $\delta_i(\vec{bw})$, $i = 1..L$ for the current $\vec{bw}$ according to Equation (1), where L tests are needed to obtain the WERs;
2. Choose the direction $k = \underset{i}{\text{argmax}}\ \delta_k(\vec{bw})$, and increase bw_k by one if $bw_k + 1 \leq M_k$;
3. Run a test to get the new $wer(\vec{bw})$. If $wer(\vec{bw}) \leq$ BWER return $\vec{bw}^* = \vec{bw}$ or, if there is no improvement in WER observed, undo the increment before returning $\vec{bw}^* = \vec{bw}$; otherwise repeat step 1.

For speech recognition, online evaluation for a test takes a significant amount of time, requiring a full test cycle (although not retraining). This algorithm consequently takes much more time to complete than does the first one. As this takes place during the design stage, this is not a problem; at recognition time there is no penalty.

This algorithm updates $\delta_i(\vec{bw})$ at each step which gives true gradients in the L-dimensional space. It might be the case that no improvement exists along each of the L directions, but that one does exist with joint increments along multiple dimensions. With this algorithm, the search might become stuck in a local optimum.

3.4. Multi-dimensional increment with dynamic gradients

Finally, the bit-widths of multiple variables could be increased in parallel. Considering the computational complexity, we only allow one- or two-dimensional increments, leading to $L + \binom{L}{2}$ possible candidates. We could extend this to include triple increments, but it would take an intolerably long time to finish. The algorithm modifies steps 1 and 2 from the above algorithm as follows:

1. Evaluate the gradients $\delta_i(\vec{bw})$ and $\delta_{ij}(\vec{bw})$, $i = 1..L$, $j = 1..L$ on current $\vec{bw}$ according to Equations (1) and (2) respectively, where $L + \binom{L}{2}$ tests are needed to obtain the WERs;
2. Choose the direction k or a pair of directions $\{k, l\}$ where $\delta_k(\vec{bw})$ or $\delta_{kl}(\vec{bw})$ is the maximum among all the single-dimensional and pair-wise increments. Increase the bit-width of V_k or those of $\{V_k, V_l\}$ by one if no one exceeds its upper bound.

This algorithm is superior to the previous two in the sense that it explores many more candidate points in the search space. It considers only one additional direction and may still fall into a local optimum, but is less likely to do so than in the previous case. Due to the measures it takes to avoid local optima, this algorithm takes substantially longer to complete. Again, this is acceptable since it happens during the design stage, not during recognition.

4. EXPERIMENTAL FRAMEWORK

4.1. Speech Recognizer

Our ASR system uses the NYNEX PhoneBook [7] database for training and testing. It is designed for isolated-word recognition tasks and consists of 93,667 isolated-word utterances recorded over a telephone channel with an 8,000 Hz sampling rate. Each sample is encoded with 8 bits via μ-law. For details of the training and testing sets, see [8].

We use standard MFCCs plus log energy and their deltas, yielding 26-dimensional acoustic feature vectors. Each vector has mean subtraction and variance normalization applied to both static and dynamic features.

The word model is a set of phone-based CHMMs concatenated together into words based on pronunciation. These pronunciation models determine the transition probabilities between phones. Doing this provides a customizable vocabulary without the need for retraining, a desirable goal for an embedded device.

The system has 42 phone models, each composed of 4 emitting states except for the silence model. The state probability distribution is a mixture of 12 diagonal Gaussians.

4.2. Lookup Tables

We simulated lookup table implementation using SimpleScalar [9], an architecture-level execution-driven simulator. All tables were pre-calculated by SimpleScalar at runtime. We extended the instruction set and modified the assembler to support new lookup instructions. Our tables use a total of 13 variables for 8 functions, 6 in the likelihood evaluation and 2 in the Viterbi search. In most cases, quantized bit-widths were restricted to between 1 and 10 bits.

We simulated results for a variety of architectures. The first is the SimpleScalar default configuration, roughly akin to a 3rd generation Alpha, an out-of-order superscalar processor. The second represents a more modern desktop machine; cache sizes and latencies have been updated from the defaults to reflect more current values. Our third configuration attempts to represent a typical high-end DSP chip. Since most DSP chips are VLIW, an architecture feature not replicable with SimpleScalar, we chose instead to force in-order execution as a very rough approximation of VLIW. Finally, we simulated a very simple processor with a single pipeline and no cache. In each case, we allowed multiple tables to

be accessed in parallel, although a specific table could be used by only 1 instruction at a time.

5. RESULTS

5.1. Allocation Search Algorithms

The results of running each allocation algorithm appear in Table 1. The baseline WER for the system was 2.07%.

	Min WER	Table Size (kB)	Points Tested
1-D, static δ	3.01%	94.00	13
1-D, dynamic δ	3.09%	36.44	52
2-D, dynamic δ	2.53%	58.75	454

Table 1. Minimum WER, total table size, and points tested for each algorithm

Not too surprisingly, the 1-D, static gradient algorithm did not perform particularly well. It managed to reach a slightly better WER than did the 1-D dynamic gradient algorithm, but it required several times more storage to do so. It reached a local optimum quickly, after trying only 13 configurations. The 1-D dynamic gradient algorithm tested a total of 52 configurations, managing to find a fairly small ROM size. The non-linear nature of the search space posed problems for this algorithm, though, which is likely why its WER was the highest. Finally, the 2-D dynamic gradient algorithm found a much better WER than the other two algorithms, even if its total table size was not the smallest. Although the final result does not reach the baseline, it gives an acceptable WER, especially when considering the potential speedup, as is done next.

5.2. Lookup Tables

We simulated each of the target machines for a range of lookup table speeds. This is independent of WER since, assuming the table is not too large, the precision of the output of a lookup has little to do with the speed of the lookup. Because we did not try to implement a low-power front-end, a feature that would be necessary for a final realization of this system, we used pre-calculated MFCCs and subtracted initialization time when calculating speedup. Figure 1 shows the speedup obtained versus the number of cycles required for each lookup. The speedup ranges from just under 2.5 to nearly 4 when lookups take only 1 cycle, and falls off as they become more expensive. In all cases, the speedup is quite significant even when lookups cost several cycles. If the ROM tables are small enough to fit on the processor chip, which is the case using the 59kB results from Table 1, a 1-cycle lookup is quite realistic for a state-of-the-art system. Much benefit actually comes from replacing sequences of instructions with a single lookup, as the dynamic instruction count fell nearly as much as execution time.

6. DISCUSSION

In this paper, we have presented a methodology for the design of high-speed, low-resource systems using custom arithmetic. We also demonstrate several resource allocation search heuristics suitable for finding acceptable points in an otherwise intractable search space. We then applied our findings to a CHMM-based ASR system. Using the allocation search algorithms we developed, we found a table configuration giving satisfactory performance with only a small additional amount of storage. The 59kB of tables is small enough that it can be added to any chip with an access time of 1 cycle. When implementing this design on a modern processor, we show that the expected speedup is at least 3, and possibly larger. Additionally, the smaller operand size means we can make more efficient use of caches to reduce cache misses, and could even narrow the memory bus width providing savings in both chip area and power consumption.

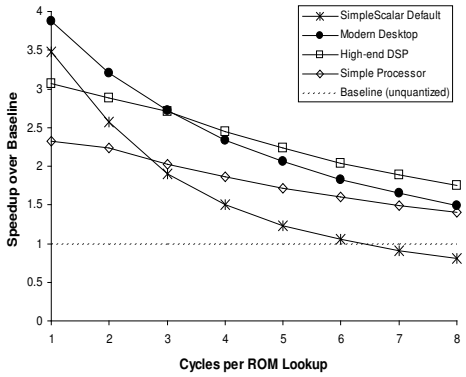

Fig. 1. Speedup (in cycles) over baseline versus cycles per ROM lookup. Baseline uses 32-bit floating point values and no lookups.

We would like to thank Chris Bartels and Gang Ji for proofreading and Carl Ebeling for much helpful advice.

7. REFERENCES

[1] K.Filali, X.Li, and J.Bilmes, "Data-driven vector clustering for low-memory footprint ASR," in *Proc. Int. Conf. on Spoken Language Processing*, 2002, pp. 1601–1604.

[2] Y.Gong and U.-H.Kao, "Implementing a high accuracy speaker-independent continuous speech recognizer on a fixed DSP," in *IEEE ICASSP*, 2000, pp. 3686–3689.

[3] E.Cornu, N.Destrez, A.Dufaux, H.Sheikhzadeh, and R.Brennan, "An ultra low power, ultra miniature voice command system based on hidden markov models," in *IEEE ICASSP*, 2002, vol. 4, pp. 3800–3803.

[4] I.Varga and et.al., "ASR in mobile phones – An industrial approach," *IEEE Trans. on Speech and Audio Processing*, vol. 10, no. 8, pp. 562–569, 2002.

[5] X.Li, J.Malkin, and J.Bilmes, "Codebook design for ASR systems based on custom arithmetic," in *IEEE ICASSP*, 2004.

[6] F.Fang, T.Chen, and R.A.Rutenbar, "Floating-point bit-width optimization for low-power signal processing applications," in *IEEE ICASSP*, 2002, pp. 3208–3211.

[7] J.F.Pitrelli, C.Fong, and H.C.Leung, "PhoneBook: A phonetically-rich isolated-word telephone-speech database," in *IEEE ICASSP*, 1995.

[8] J.A.Bilmes, "Buried Markov models for speech recognition," in *IEEE ICASSP*, 1999.

[9] Doug Burger, Todd M. Austin, and Steve Bennett, "Evaluating future microprocessors: The simplescalar tool set," Tech. Rep. CS-TR-1996-1308, 1996.