Extracting Posteriors from a Gaussian Mixture Model

Xinkun Nie and Paul Myers

Abstract

The design of a system for extracting posterior distributions from a Gaussian mixture model is presented and its implementation on a FPGA is discussed; the system is intended for eventual integration into a processing pipeline for real-time speaker identification. An existing software implementation of the posterior extraction algorithm, written using the Kaldi speech recognition toolkit, was used as the basis of the design, which comprises three primary modules. A detailed description of the microarchitecture for each module is described, as well as the memory layout. The performance and numerical accuracy of the system were evaluated and found to be satisfactory for most applications of interest.

I. Introduction

Recent advances in computational infrastructures and the rapid development of rich statistical theories have resulted in a surge in interest in artificial intelligence (AI) applications. One of the most commonly performed tasks in many AI systems is classification, the purpose of which is to assign to each observation in a set of data an appropriate label from a set of categories. Although classification is applied to a large number of highly varied domains, the algorithms used to perform the task generalize across applications. For the purpose of this project, the application domain of choice is speaker identification. Speaker identification may be used to identify individual users of a number of systems, including telecommunication systems used by banks to conduct financial transactions, and, more recently, personal assistants such as Apple's Siri and Amazon's Echo. While many speaker classification algorithms exist, the one chosen for this project relies on mathematical constructs known as Gaussian mixture models (GMM) and i-vectors [1][2]. Prior to performing the classification task, a Gaussian mixture model and an i-vector extractor must be trained. Then, raw feature vectors may be extracted from the user population and filtered through a voice activity detection system. The GMM may then be used to extract posterior probabilities from the processed feature vectors, and these posterior probabilities may then be given as input to the i-vector extractor; the output of the extractor may then be used to perform the desired classification task. An example of such a pipeline is shown graphically in Fig. 1.

While each of the processes listed in the above algorithm is important to the overall performance of the system, for the purpose of this project, the step in which posterior probabilities are extracted from the GMM will be considered only. Since real-time operation is essential in most practical speaker identification systems, a custom design in hardware is desirable; implementing this step of the algorithm on a FPGA is therefore justified.

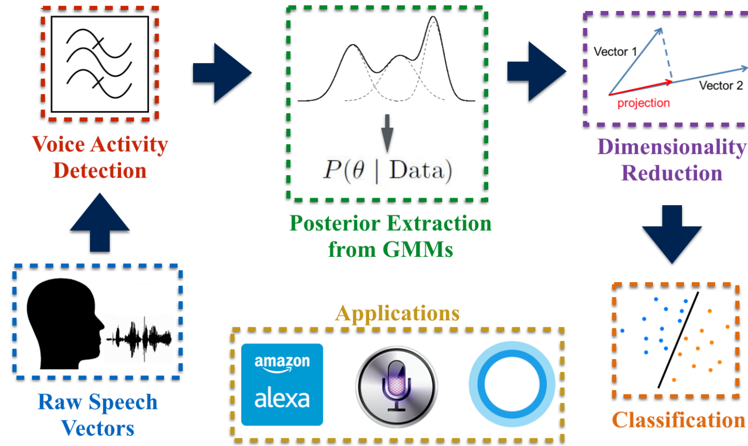

Figure 1: Diagram of an example speaker identification pipeline. The posterior extraction algorithm implemented in this project is represented by the green box in the center of the figure.

II. System Overview

The hardware implementation of the GMM system is based on a software implementation of the algorithm in the Kaldi speech recognition toolkit [3]. On a FPGA, the following two tasks must be performed: storing the trained GMM in memory, and computing the posterior probabilities of processed feature vectors generated by the GMM. The second step utilizes a large number of Gaussian mixture components. To achieve computational efficiency, an algorithm used to select the few mixture components that have a very high posterior probability will be implemented following a function called *gmm-select* in the Kaldi software package; this function essentially discards the off-diagonal terms in the covariance matrix of each Gaussian mixture. The mixtures generated by this function are then given as input to a second function that uses the full covariance matrix to produce a more accurate posterior probability; this algorithm is implemented in a Kaldi function called *fgmm-global-gselect-to-post*. The system will output these posterior probabilities, which could then be given as input to a test-bench written using Kaldi to perform speaker identification in software.

A high-level diagram illustrating the system architecture is presented in Fig. 1. The data flow corresponds to the algorithm described above, with the block labeled “Diagonal Gaussian Selection” corresponding to the *gmm-select* Kaldi function and the block labeled “Full Gaussian Selection” corresponding to the *fgmm-global-gselect-to-post* Kaldi function. The Feature Matrix contains the set of feature vectors generated by a voice activity detection preprocessing step in software, with each row of the matrix corresponding to the feature vector generated by a given audio frame. The frame rate is assumed to be 100 frames per second, indicating that a new frame will be delivered to the system every 10 milliseconds. Frames are stored in DRAM in batches of a certain size, and all frames in the batch are processed simultaneously.

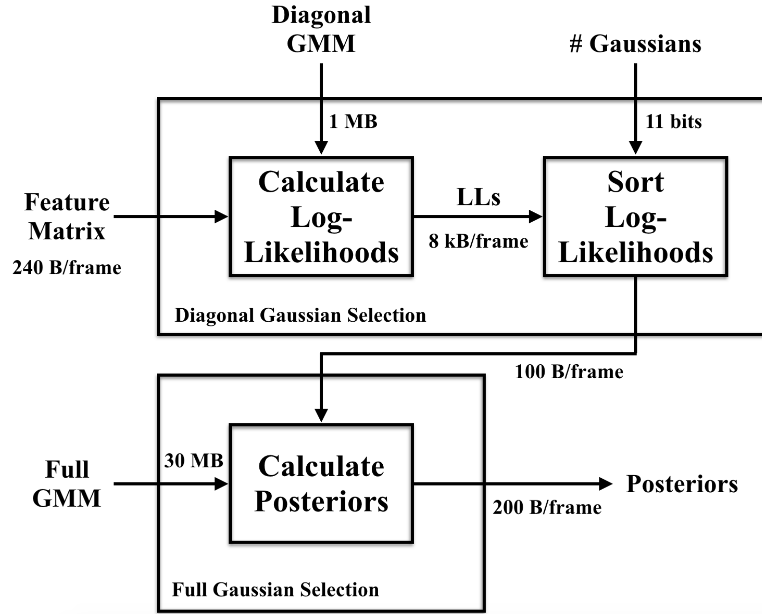

Figure 2: Block diagram of the system.

The number of frames processed simultaneously is a parameter that must be set prior to synthesis; for testing, each batch consisted of two frames. Each frame consists of 60 dimensions, with each dimension being a real number. In order to simplify the implementation and reduce the amount of hardware required for the processing steps, fixed-point notation will be assumed everywhere in the system; given the size of the numbers being processed and the precision requirements of the system, it is expected that 32-bit fixed-point notation should be sufficient. Using 32-bit precision enables one dimension to be stored in a single word that is four bytes in size.

Referring again to Fig. 1, a predetermined number of frames is given as input to a module that will compute the log-likelihood of each Gaussian component in the diagonal GMM and will output an array of 32-bit fixed-point numbers, where the number of columns in the array is the number of Gaussian components in the GMM, in this case 2048, and the number of rows is the number of frames processed simultaneously. This result is then given as input to a module that sorts the log-likelihoods by magnitude and outputs a matrix of 11-bit numbers, the number of columns of which is set by user input and the number of rows of which corresponds to the number of frames; 11-bit numbers are used, since there are 2048 Gaussians. This result, along with the feature matrix input and full GMM are given as input to a module that will calculate the posteriors of the pre-selected Gaussian components. The final output of the system is an array that contains the posteriors calculated from each Gaussian, with the number of rows being the number of frames processed simultaneously and the number of columns being the number of Gaussian components selected; the Gaussian components that are not selected are assigned a posterior likelihood of zero.

III. System Evaluation

The primary evaluation metric for this project is numerical accuracy, since the relatively low frame rate of the input features does not require high performance to achieve real-time operation. Power consumption and area are additional features to consider. While area may be estimated from the resource utilization reports generated after synthesis, power consumption cannot be easily measured or estimated using the available tools for the FPGA; thus, numerical accuracy and area will be considered exclusively. In order to verify functional correctness, a software testbench was written using the Kaldi code and the system output was compared with the software output; numerical accuracy was likewise verified by computing the error between the output of the system and the software result and comparing the error to a pre-defined tolerance.

IV. Memory Layout

On-board memory is limited to approximately 6.75 MB. Since the diagonal GMM requires 1 MB of memory, the full GMM requires 30 MB of memory, and each frame requires 240 B of memory, it is not possible to store all of the data required for each computation locally. All GMMs and all frames are initially stored in DRAM, and are then read and stored locally to perform the necessary computations. Each DRAM address corresponds to a 64 B-line. The data are stored sequentially beginning at address zero, with the diagonal GMM being stored first, the full GMM being stored second, and the frames being stored third. For each Gaussian in the diagonal GMM, it is necessary to store a single 32-bit constant and two 60-element vectors of 32-bit numbers. For the full GMM, it is necessary to store a single 32-bit constant, a 60-element vector of 32-bit numbers, and a 1830-element vector of 32-bit numbers. Each frame consists of two 60-element vectors and one 1830-element vector. For the diagonal GMM, the constant for each Gaussian is stored in the first byte of a single line and the remaining bytes in that line are set to zero. The first 60-element vector is then stored in the next four lines, with the last four bytes of the last line set to zero, and the second 60-element vector is stored in the next four lines. The allocation for the next Gaussian begins on the line immediately after the last line of the second 60-element vector of the previous Gaussian. The allocation for the full GMM begins on the line following the last line of the last Gaussian in the diagonal GMM. As with the diagonal GMM, the constant for the full GMM is stored in the first byte of a single line. The 60-element vector is stored in the next four lines, with the last four bytes of the last line being set to zero, and the 1830-element vector is stored in the next 124 lines, with the data in all empty lines being set to zero. For each frame, the two 60-element vectors are stored in eight lines; after all such vectors have been stored, the 1830-element vectors are stored. A schematic of the memory layout is shown in Fig. 3.

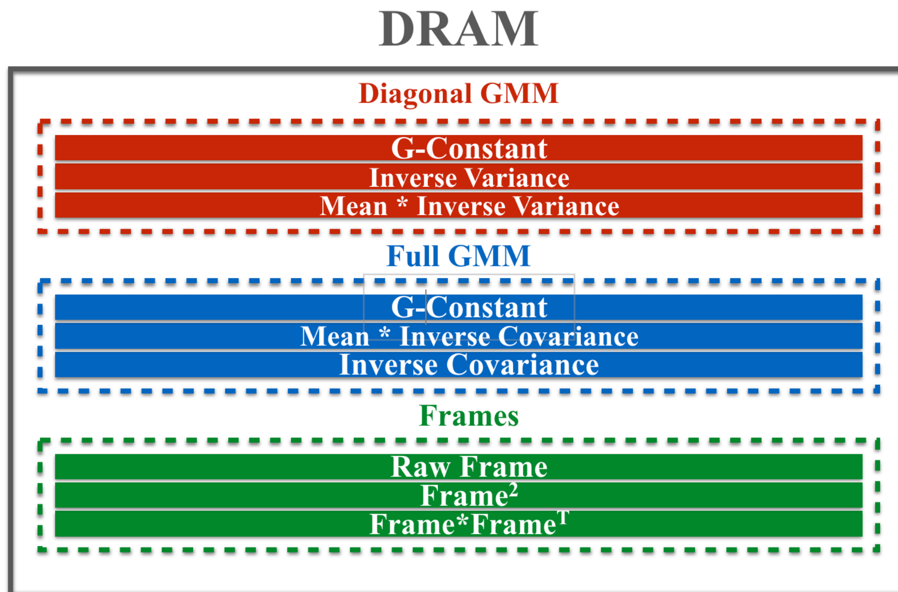

Figure 3: Schematic of the memory layout.

V. Microarchitecture

V.I. Diagonal GMM Selection Module

Fig. 2 presents a block diagram of the microarchitecture for the diagonal GMM selection unit, labeled "Calculate Log-Likelihoods" in Fig. 1. A single log-likelihood is calculated from a single frame and a single Gaussian component by performing a series of matrix operations on the constituents of each Gaussian component. The first set of operations requires computing the dot product between a single frame, which is represented as a 60-element vector of 32-bit fixed-point numbers, and the product of the mean and inverse variance of each Gaussian, which is also a 60-element vector of 32-bit fixed-point numbers. The product of the mean and inverse variance is assumed to be computed in software and stored directly in the DRAM, so the computation performed in hardware is strictly the dot product between two 60-element vectors. The second aspect of the computation is a second 60-element dot product, this time between the same frame with each element squared and scaled by -0.5 and the inverse variance of the same Gaussian. The squaring and scaling of the frame vector is assumed to be done in software and stored directly in the DRAM. To perform these dot products, a functional unit comprising a set of 64 parallel 32-bit multipliers and a five-level tree of 32-bit adders was designed. The computation proceeds in stages, as follows. First, the corresponding elements of the two vectors are provided as input to the bank of multipliers, which produces 64 32-bit numbers. These 64 results are then summed pairwise by a bank of 32 parallel adders, which produce 16 32-bit numbers; the computation proceeds to subsequent layers of adder banks until a single 32-bit result is produced.

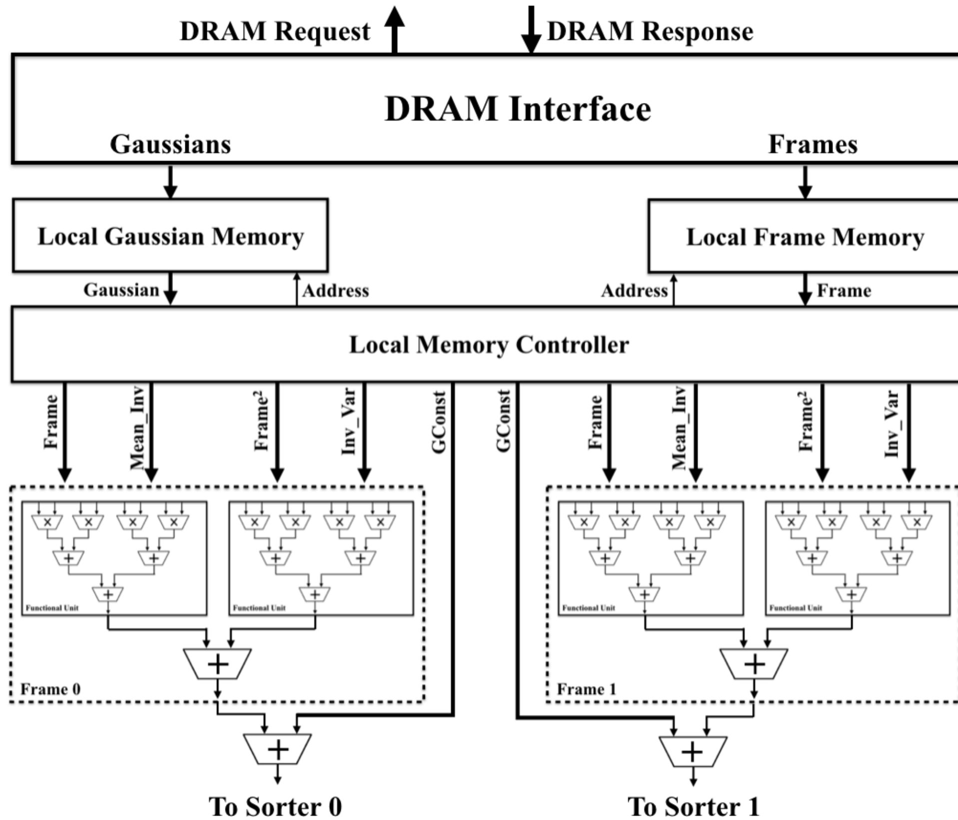

Figure 4: Microarchitecture of the diagonal GMM selection module.

Although each vector is of 60 dimensions, 64 parallel multiplications are done. This choice was made to ensure that each layer of the adder tree consists of an even number of adders; the final four elements are set to zero, thereby leaving the dot product unchanged. The two dot products described above are computed in parallel and the results are summed; this result is then summed with a constant that is unique to each Gaussian to produce a single log-likelihood. In the final design, a variable number of frames could be processed in parallel; for simplicity, a batch of two frames was chosen.

Coordinating the transfer of data between the DRAM and the diagonal GMM module is complicated by the restriction that each response from the DRAM is limited to 64 B. For each 60-element vectors of 32-bit numbers, it is therefore necessary to make four requests to the DRAM, since each dimension of the vector is 4 B in size for a total of 240 B per vector. The DRAM Interface shown in Fig. 2 is responsible for recording which DRAM addresses are needed by the processing modules and making the necessary requests to the DRAM. Since full 60-element vectors are requested from the DRAM in sets of four, the DRAM Interface unit is also responsible for concatenating the four DRAM responses obtained for each vector into a single vector for use by the functional units. The single resultant vector is then stored either in the Local Gaussian

Memory or the Local Frame Memory depending on the type of data being used. The Local Memory Controller is then responsible for generating requests to the local Gaussian and frame memories and sending the responses to the functional units. The DRAM Interface and Local Memory Controller are designed internally as pipelined state machines.

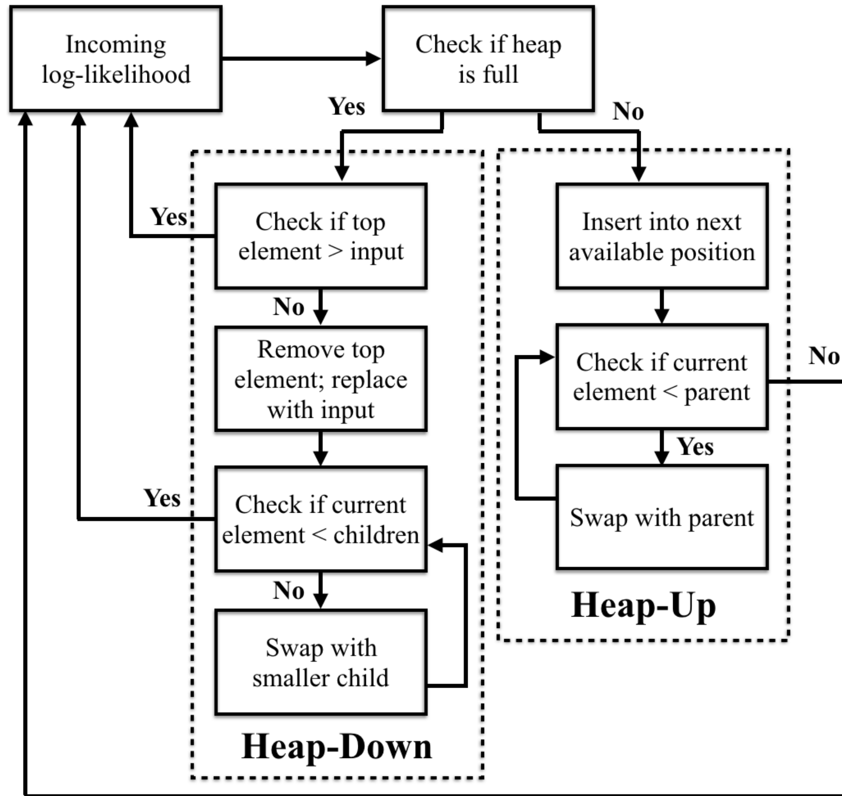

Figure 5: Flow diagram of the state machine implemented in the sorting module.

V.II.Sorting Module

For the sorting module, the Heap Sort algorithm was implemented, as it sorts array elements in-place, thereby reducing memory usage. The module is responsible for extracting the maximum k elements, where k is the number of Gaussians to be selected prior to computing the posteriors using the full GMM. A min-heap of size k, initially empty, that will eventually contain the maximum k elements out of 2048 log-likelihood values computed from the previous diagonal GMM module is maintained. With every incoming log-likelihood, there are two cases. In the first case, the k-element min-heap has not been filled, so the incoming element will be appended to the end of the existing tree, and a heap-up is done. To heap-up, the current node is compared to its parent node. If the current node is greater, the two elements are swapped and the operation is repeated. In the second case, the k-element heap has been filled. The incoming element is compared with

the top element of the heap. If it is smaller than the top element of the heap it will be smaller than all the elements of the heap, since a min-heap is used, in which case the element is discarded and nothing is done. Otherwise, the top element of the tree is replaced with the new incoming element, and a heap-down is done. To heap-down, the current element is compared with both of its children nodes. The minimum is found among the three, and the smallest element is swapped with the current element. If the smallest element is the current element, nothing is done; otherwise, the heap-down is repeated. As a result, the k maximum numbers given all the elements seen so far are kept. Both the heap-up and the heap-down operations take $O(log(k))$ time, giving a final run-time of $O(n*log(k))$, where n is the total numbers of elements; in the present case, n is 2048. Fig. 4 shows a flow diagram of the sorting algorithm.

In hardware, the sorting module was implemented as a finite-state machine (FSM). The inputs to the module are the 2048 log-likelihoods computed by the diagonal GMM selection module and their respective indices for each frame; the module outputs the 20 largest log-likelihoods and their respective Gaussian indices.

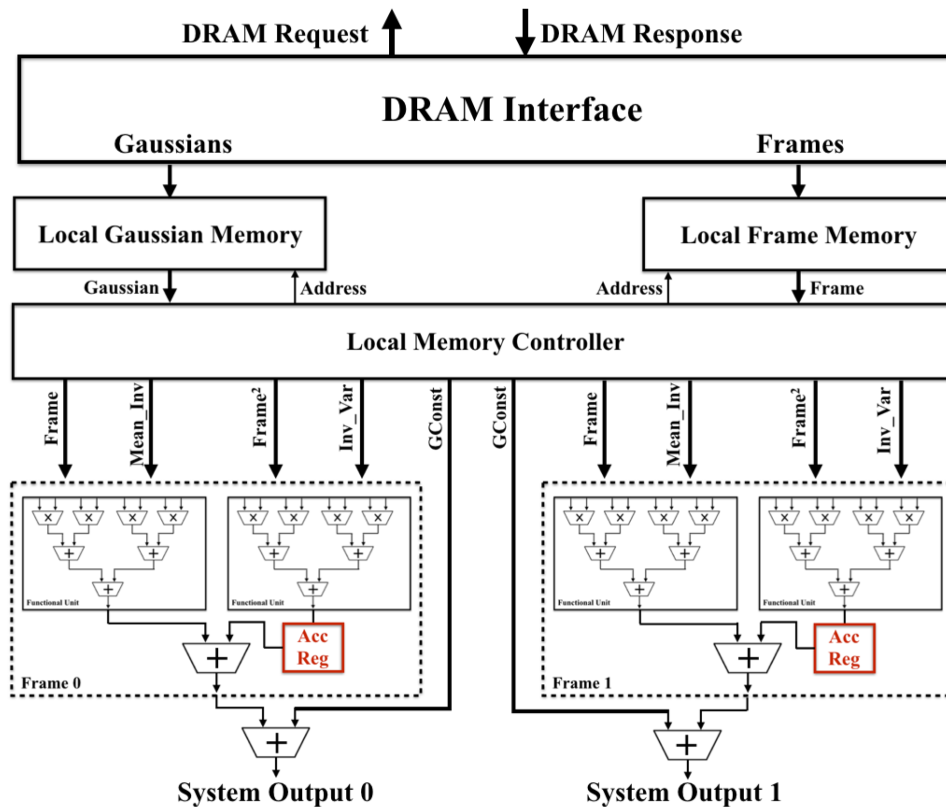

Figure 6: Microarchitecture of the full GMM selection module. The high-level design is similar to that of the diagonal GMM selection module, with the exception of the accumulation register (Acc Reg) highlighted in red. The internal Local Memory Controller, local memories, and DRAM Interface are different from those of the diagonal GMM selection module, since the full GMM model is represented differently than the diagonal GMM model is.

V.III. Full GMM Selection Module

Fig. 4 shows a diagram of the full GMM selection module, labeled "Calculate Posteriors" in Fig. 1; the high-level microarchitecture is similar to that implemented in the diagonal GMM selection module. The DRAM Interface is again responsible for generating requests to and receiving responses from the DRAM, as well as packaging the DRAM responses into a form that is suitable for processing by the functional units. The Local Memory Controller is again designed to generate requests to the local memories and output the responses to the functional units; the functional units themselves are identical to those used in the diagonal GMM module. The primary difference between the to GMM modules is the presence of the accumulation register, which is labeled Acc Reg and highlighted in red in Fig. 4. To compute a log-likelihood for a given frame and Gaussian pair, it is necessary to compute a dot product between the given frame and the product of the mean and inverse covariance of the given Gaussian; the product of the mean and inverse covariance is assumed to be computed in software and stored directly in the DRAM. This dot product is similar to those computed in the diagonal GMM module, as the dot product is between to 60-element vectors of 32-bit numbers. The second part of the computation requires computing a dot product between two 1830-element vectors. In order to reuse the previously designed functional unit, the 1830-element dot product was partitioned into sets of 60-element dot products. Since 1830 is not a multiple of 60, the multiple of 60 that is closest to and larger than 1830 was used, which is 1860; the additional 30 dimensions are set to zero, so that the dot product remains unaffected. Completion of the full 1860-element dot product therefore requires 31 60-element dot products. The accumulation register is used to sum the result of the current 60-element dot product with the accumulated results of the previous dot products. Once the 1860-element dot product is complete, the result is summed with that produced by the the dot product between the frame and product of the mean and inverse covariance; this result is then summed with a constant unique to each Gaussian to produce the final log-likelihood. Mathematically, the second dot product is performed between the flattened lower-triangle of the inverse covariance matrix for a given Gaussian and the flattened lower-triangle of the product of a given frame and its transpose.

VI. Results

One of the primary concerns associated with using fixed-point notation instead of the floating-point notation used in the software implementation of the algorithm is accuracy loss. The system was designed using 32-bit numbers, where the number of bits allocated for the integral and fractional parts was parameterized and could be changed prior to synthesis. For initial testing, 16 bits were allocated to both the integral and fractional parts. The average error between the posteriors generated by the system and those generated by the software was computed by finding the percent difference between the posterior for each Gaussian generated in software and hardware, summing the differences over all of the Gaussians, then dividing by the total number of posteriors generated as output, which was five in the final test; the error was computed individually for each of the two frames tested. The average error for the first frame was found to

be 0.0085% and the average error for the second frame was 0.0078%. These errors are quite small and were considered negligible for most applications of interest; thus, the original choice of a 32-bit fixed-point representation with 16 bits allocated to both the integral and fractional parts was deemed sufficient. The calculated errors for each selected posterior for the first frame are shown in Table 1.

Table 1: Simulation Results and Calculated Errors for the First Frame

Gaussian ID	Hardware Value	Software Value	Error
0	-156.495	-156.5087	0.00875
4	-161.275	-161.2985	0.0145
5	-195.919	-195.9338	0.00755
6	-181.110	-181.1254	0.00850
8	-192.481	-192.4866	0.00290

The design was synthesized, placed, and routed for eventual simulation on a FPGA. Although the design was never successfully run on a FPGA, reports for timing and resource utilization were still made available after synthesis. The design was synthesized with a relatively conservative clock frequency of 20 MHz in order to minimize the number of timing violations generated during synthesis; some salient results of synthesis are summarized in Table 2.

One notable challenge of implementing this design on a FPGA was meeting area requirements. In order to reduce the critical paths in the design, pipelining was used extensively, particularly in the functional units. Pipelining necessarily requires the use of intermediate storage elements between stages, thereby increasing the hardware cost of each module. Since full 32-bit 60-element vectors were often passed between pipeline stages, the storage elements were required to be quite large, which initially prevented the design from fitting on a FPGA board. Most of these storage elements were two-element FIFOs, so area was reclaimed by reducing the size of each FIFO in the functional units to one. The performance penalty incurred by such a change was difficult to measure, but is estimated to be tolerable. Area could have been further reduced by decreasing the number of parallel multipliers and adders in the functional units by folding the computation.

Table 2: Summary of Synthesis Results

Parameter	Value
Slice LUTs	295450 (97.31% utilization)
Slice Registers	260133 (42.84% utilization)
Block RAM Tiles	354 (34.36% utilization)
DSP Blocks	1920 (68.57% utilization)
Clock Frequency	20 MHz
Critical Path	33.405 ns (In DRAM Control)
Total Negative Slack	0.000 ns
Total Negative Slack Failing Endpoints	277973
Worst Negative Slack	0.077 ns

This approach would have increased the number of cycles required to complete each calculation, but would have also likely reduced the critical path length. An alternative approach could have been to use BRAM in place of wide registers, since, according to Table 2, the BRAM utilization is quite low.

VII.Conclusion

Although the design was not successfully simulated on a FPGA, the successful simulation of the implementation indicates that the design performs the desired task sufficiently well as measured by numerical accuracy and performance. The design could be improved by further reducing the area and shortening critical path lengths to reduce the clock period.

Acknowledgements

Glory is due first and foremost to the Lord Almighty, by whose grace the completion of this project was made possible.

The authors would also like to thank Professor Arvind for his guidance throughout the course of the term, as well as Ming Liu and Dr. Murali Vijayaraghavan for their assistance with the implementation of the design. The authors would especially like to thank Dr. Michael Price for generously offering his time and advice during the completion of the project.

References

[1] P. Kenny, “A small footprint i-vector extractor,” *Odyssey 2012*: 1-6.
[2] N. Dehak, P.J. Kenney, R. Dehak, P. Dumouchel, and P. Ouellet, “Front-end factor analysis for speaker verification,” *IEEE Trans. Audio, Speech, Language Proc.*, vol. 19, no. 4, May 2011.

[3] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hanneman, P. Motlicek, Y. Qian, P. Schwarz, J. Silovsky, G. Stemmer, K. Vesely, “The Kaldi speech recognition toolkit,” *IEEE 2011 Workshop on Automatic Speech Recognition and Understanding,* December 2011.