FPGA Implementation of a Pipelined Gaussian Calculation for HMM-Based Large Vocabulary Speech Recognition

Veitch, R., Aubert, L-M., Woods, R., & Fischaber, S. (2011). FPGA Implementation of a Pipelined Gaussian Calculation for HMM-Based Large Vocabulary Speech Recognition. International Journal of Reconfigurable Computing, 2011, [697080]. DOI: 10.1155/2011/697080

Published in:
International Journal of Reconfigurable Computing

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 697080, 10 pages
doi:10.1155/2011/697080

Research Article

FPGA Implementation of a Pipelined Gaussian Calculation for HMM-Based Large Vocabulary Speech Recognition

Richard Veitch, Louis-Marie Aubert, Roger Woods, and Scott Fischaber

Electronics, Communications and Information Technology (ECIT), Queens University Belfast, Northern Ireland Science Park, Belfast BT3 9DT, UK

Correspondence should be addressed to Richard Veitch, r.veitch@ecit.qub.ac.uk

Received 1 June 2010; Revised 19 September 2010; Accepted 27 September 2010

Academic Editor: Gustavo Sutter

A scalable large vocabulary, speaker independent speech recognition system is being developed using Hidden Markov Models (HMMs) for acoustic modeling and a Weighted Finite State Transducer (WFST) to compile sentence, word, and phoneme models. The system comprises a software backend search and an FPGA-based Gaussian calculation which are covered here. In this paper, we present an efficient pipelined design implemented both as an embedded peripheral and as a scalable, parallel hardware accelerator. Both architectures have been implemented on an Alpha Data XRC-5T1, reconfigurable computer housing a Virtex 5 SX95T FPGA. The core has been tested and is capable of calculating a full set of Gaussian results from 3825 acoustic models in 9.03 ms which coupled with a backend search of 5000 words has provided an accuracy of over 80%. Parallel implementations have been designed with up to 32 cores and have been successfully implemented with a clock frequency of 133 MHz.

1. Introduction

Automated Speech Recognition (ASR) systems can revolutionize the way that we interact with technology. Large vocabulary speaker independent systems have potential in all forms of computing, from hand held mobile devices to personal computing and even large scale data centres. A low power, real-time embedded system could dramatically impact our daily interactions with digital mobile technology [1] while a faster than real-time multi-stream batch decoder could be used in server applications for distributed systems [2] or data-mining [3, 4].

There are a range of open source software ASR systems available [5, 6]. These tools employ Hidden Markov Models and Viterbi decoding to provide a speech decoder that can be configured for a variety of implementations. Over the last 5 years, however, the research concerning high performance ASR has been more focused on hardware implementations and as such, many FPGA-based speech recognition systems have been implemented, although systems have generally been limited by small vocabulary [7, 8] or have relied on custom hardware to provide the necessary resources required for a large vocabulary system [9]. The approach of pairing a softcore processor with a custom IP peripheral is popular and has been proposed in a number of papers [8, 10] but a system operating on large vocabularies at real-time is yet to be demonstrated. This is, in part, due to the low operating frequencies of softcore processors but another problem is the interfacing with off-chip, high capacity RAM which can introduce large delays that cripple a high bandwidth system like speech recognition.

A real-time speech recognition system is being designed at Queens University Belfast which uses a Weighted Finite State Transducer (WFST) [11] to precompute much of the information needed. This design decision allows for a great degree of flexibility since the WFST can be compiled to suit the available system storage, bandwidth, and processing potential. The decoder forms the basis of the recognition algorithm and has been designed to be scalable to run either in an embedded system where real-time performance at low power will be the goal or alternatively in a server or desktop situation, where higher processor frequencies will allow better than real-time performance for batch or multi-stream decoding.

A software implementation has been created as a proof of concept and tested successfully on a PC. This system

is capable of running at better than real-time, using a vocabulary of 5000 words with an accuracy of 80%, which is consistent with the current state of the art, but does require the full effort of a 3 GHz Intel processor with 3 GB of DDR3 RAM. In order to provide a mobile implementation of the system, work has been carried out to design a multicore architecture comprising of embedded processors and custom IP. The first step in the creation of this mobile architecture is an FPGA-based demonstrator based around a Microblaze softcore processor.

Through profiling of the software system, the Gaussian calculation has been identified as a performance bottleneck. For this reason, a custom Gaussian core was developed with a simple interface that allows it to be implemented either as an embedded peripheral in a system-on-chip speech recognition system or as an FPGA hardware accelerator for use with a desktop or server software system. By decoupling the control and data flow components of the core from the RAM interface, parallel implementations can be achieved in multiple different configurations which can be tailored to make the most of the available resources.

This work, originally presented at the 6th Southern Programmable Logic Conference [12], outlined the Gaussian core as a hardware peripheral capable of real-time operation; dual cores were implemented in order to achieve this. Improvements to that paper that have been included here are given below.

(i) The single core is now capable of running at real time. This increase was achieved by streamlining the loading of acoustic model data from the SDRAM.

(ii) The Software proof of concept system has been optimized to include SIMD instructions and multithreading.

(iii) A number of multicore parallel implementations have been designed and implemented.

(iv) Both the software proof of concept and the multicore implementation of the Gaussian core have been demonstrated to run faster than real-time.

The paper is organized as follows. Section 2 describes the HMM/WFST speech recognition system. Section 3 describes the performance of a software proof of concept implementation and gives some details of optimizations. Section 4 details the design decisions made to allow efficient implementation of the Gaussian calculation as an FPGA-based custom IP core and the options for parallelization. Section 5 gives details of the FPGA implementation of the Gaussian core along with speed and area results for various parallel architectures. Finally, section 6 discusses the implications of the pipelined Gaussian core for both real-time embedded mobile systems and server-based distributed systems.

2. HMM-Based Continuous Speech Recognition

The speech recognition system outputs a textual transcription of the input speech. To do so, the system uses three main knowledge sources prebuilt from speech statistics, which are illustrated in Figure 1: the *language model*, or grammar, gives statistical information on how sentences are broken up into sequences of words; the *lexicon* describes how words are broken up into sequences of sounds, or phonemes; these *phonemes* are represented by Hidden Markov Models as sequences of states with transition probabilities [13].

Each state in the HMM is associated with an acoustic model. Acoustic models, sometimes referred to as Gaussian Mixture Models (GMMs), are Gaussian-based functions used to compute the probability that the input speech frame belongs to the corresponding phoneme [7, 14]. By precomputing these knowledge sources off-line, we are able to reduce the processing power required by the speech recognition system as a whole, at the expense of increasing the memory storage and bandwidth requirements.

At the implementation level, the decoding of the input speech can be separated into three blocks, as shown in Figure 2. These are feature extraction, Gaussian calculation, and backend search. The Gaussian calculation calculates probabilities according to the input speech which are passed to the backend search. Note that it does not work on the audio waveform itself but on a series of Acoustic Observation Vectors (AOVs) created from that waveform by the process of feature extraction.

2.1. Acoustic Observation Vectors. Each AOV consists of typically 39 Mel-Frequency Cepstral Coefficients (MFCCs) which are created from a frame of input speech by applying a series of transforms [15, 16] as shown in Figure 3. The MFCCs contain all the necessary acoustic information of one frame of input speech. A frame represents a 25 ms Hamming window of speech. Frames overlap such that a new AOV is available every 10 ms of speech as shown in Figure 4. Hence, in order to offer real-time performances, the Gaussian calculation block must process each AOV in less than 10 ms.

2.2. Gaussian Calculation. In this system, the most computational intensive task is the Gaussian calculation. It must compute the probability for the acoustic models which are expressed as a mixture of multi-dimensional Gaussians as shown in following equation [7]:

$$p_j = \sum_{m=1}^{M} \frac{w_{j,m}}{2\pi \prod_{k=1}^{K} \sigma_{j,m,k}} \exp\left(\sum_{k=1}^{K} \frac{\left(x_k - \mu_{j,m,k}\right)^2}{2\sigma_{j,m,k}^2} \right), \quad (1)$$

where j is an index for the current acoustic model and M denotes the number of mixtures in the model. Typically there are 8 mixtures in each model although that may vary between models depending on their complexity. The coefficients, x_k, are the MFCCs of the input AOVs. As seen in (1), acoustic models are fully characterized by their mean (μ), variance (σ^2), and weight coefficients (w). This strict implementation of the mixture of Gaussian models is complex as it involves highly nonlinear operations. However, it has been proven that taking the maximum of all of the mixtures, instead of the sum, is a very good approximation of the result [17].

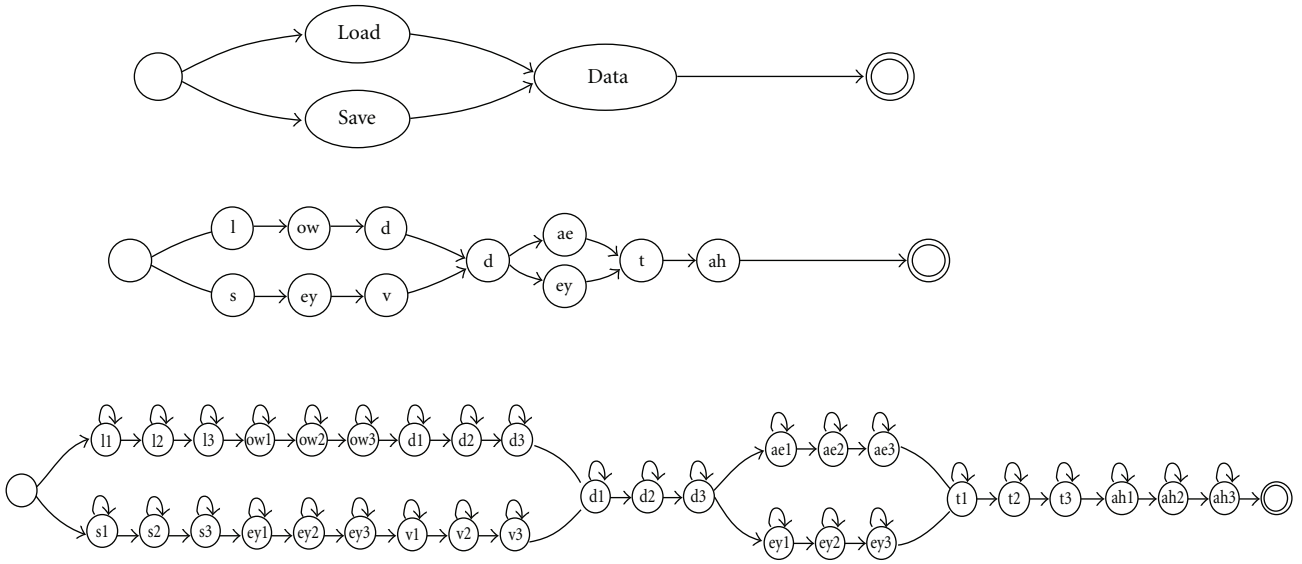

Figure 1: Language, lexicon, and phoneme sources combined.

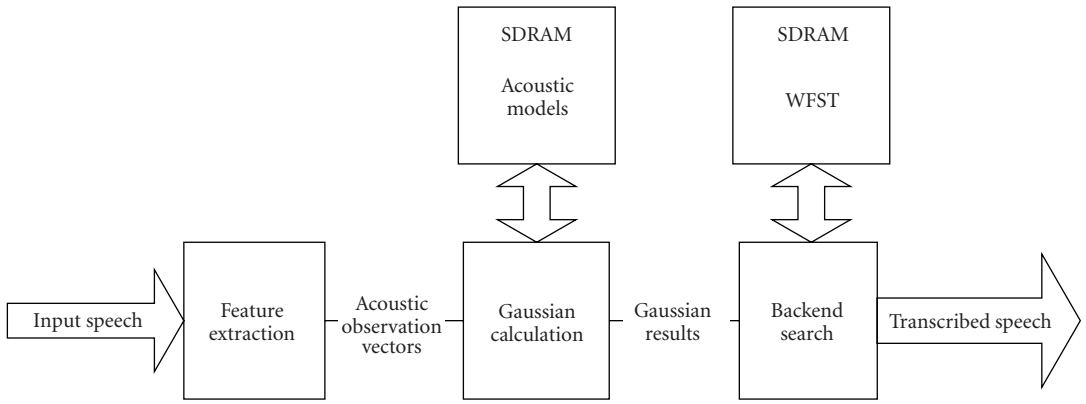

Figure 2: Overview of functional blocks.

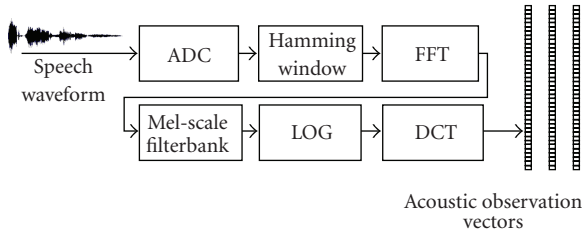

Figure 3: Translation of AOVs from input audio.

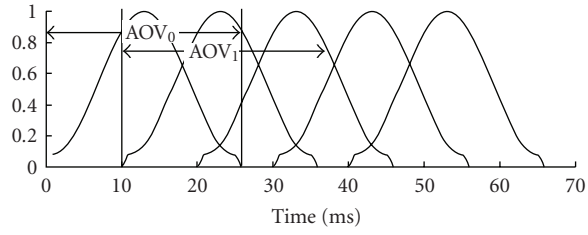

Figure 4: Overlapping of acoustic observation vectors.

Hence, by taking the negative of the logarithm of (1) in order to convert probabilities into costs and applying the previous approximation, the acoustic cost can be evaluated by

$$c_j = \min_{m=1}^{M} \left(\alpha_{j,m} + \sum_{k=1}^{K} \frac{\left(x_k - \mu_{j,m,k}\right)^2}{2\sigma_{j,m,k}^2} \right), \tag{2}$$

where α is a coefficient per mixture that encompasses all constants and parameters outside of the exponential function in (1).

In order to further simplify the computation of the Gaussian block, the variance σ^2 is replaced by a new variable ν' as described in (3). By storing this precomputed value instead of the variance, the calculation can be carried out by (4) which defines the calculation of a Gaussian result for one

AOV and one Acoustic Model.

$$v' = \frac{1}{\sqrt{2\sigma^2}}, \tag{3}$$

$$c_j = \min_{m=1}^{M} \left(\alpha_{j,m} + \sum_{k=1}^{K} \left(\left(\mu_{j,m,k} - x_k \right) \times v'_{j,m,k} \right)^2 \right). \tag{4}$$

2.3. Backend Search. The backend search portion of the system is based on a token propagation algorithm which uses a single WFST network to combine all static knowledge sources of the recognizer: language model, lexicon, and HMM state sequence. The WFST consists of nodes and weighted arcs which are stored in memory and loaded as needed. The token propagation algorithm begins by setting a single token at the initial node of the network. The token is replicated and propagated along every arc leaving the node. Each propagated token has two costs added to its running total. The first is a fixed cost which is the transitional cost of the HMM state represented by the arc [13]. The second cost is calculated by the Gaussian calculation and is dependent on the current AOV and the Acoustic Model associated with the arc [11]. This propagation process is repeated for all existing tokens once per frame. Since the number of tokens within the system increases with each iteration, pruning techniques are required to discard least probable tokens with the highest costs. At the end of a predefined period, the system outputs the most probable transcriptions by tracing back the paths through the WFST network taken by the token with the lowest costs.

2.4. Test Data. In order to recognize continuous speech, the HMMs used do not model isolated phonemes but context dependent triphones, that is, phonemes altered by the preceding and following phonemes. There is an extremely large number of possible tri-phones but a tying-up technique allows the system to work effectively with only 4 400 distinct HMMs [13]. The system uses mainly 3-state HMMs and so would require a theoretical number of $3 \times 4\,400 = 13\,200$ acoustic models. However, a pool of only 3 825 distinct acoustic models is enough to address all HMM states. With a set of acoustic models of this kind, the system is capable of accurate recognition, even with very large vocabulary of up to 50 000 words.

The current demonstrator uses a WFST built from a 5000-word vocabulary which embeds all static knowledge sources, from the bi-gram language model down to all 4 400 HMMs. However, the system has been designed with flexibility as the utmost priority. In the final system, it is intended to increase the vocabulary to 50,000 words. There is also the flexibility to use larger sets of acoustic models. This could be particularly useful for other languages requiring a larger quantity of component sounds.

3. Software Performance

A software implementation of the WFST/HMM speech recognition system has been implemented and tested as a proof of concept. The software was coded in C++ and has been tested on a PC with an Intel Core i7 CPU running at 2.8 GHz with 3 GB of RAM. The system performance is listed in Table 1.

Table 1: Timing results for software implementation.

	Frames/s	Speed (× real time)	Percentage of Clock cycles spent on gaussian calc.
Sequential C++	114	0.88	80%
C++ with SSE intrinsics	902	0.11	48%
8-thread parallel C++ with SSE	2756	0.036	48%

Performance figures are given for two implementations. The first case, which was originally presented in [12], has been implemented in C++ compiled with Microsoft compilers via Visual Studio 2008. This implementation achieves speeds of 114 frames per second on average which is better than the real-time figure of 100 frames per second. Profiling of the software shows that the Gaussian calculation takes up a large majority of the execution time; for this reason, an effort was made to increase the efficiency of this calculation.

In an alternative implementation, the Streaming SIMD Extensions (SSE2) of Intel Processors are used to accelerate the Gaussian calculation by exploiting data parallelism within the arithmetic operations of the Gaussian core. The other functions of the program remain sequential and unchanged.

SIMD architectures provide multiple processing elements that perform the same operation on multiple data simultaneously. SSE2 works on 128-bit registers, that are interpreted in our implementation as 2×64-bit, 4×32-bit, 8×16-bit, or 16×8-bit fixed point integers (signed or unsigned). The effect of using fixed point rather than floating point will be discussed further in Section 4.1. Figure 5 gives a simplified data flow diagram of the SIMD implementation of the Gaussian calculation. Note that numbers in brackets give the fixed-point details on data. The memory organization of the acoustic models was also reviewed to get 16-byte memory-aligned data optimally transferred to SSE2 registers. The use of SSE2 intrinsics with optimized memory access allows the Gaussian calculations to run more than 13 times faster, which translates to the whole speech recognition systems running at 8 times faster than unoptimized code or 9 times faster than real time. However, note that even with the SIMD instructions, the Gaussian core is responsible for 48% of the program's execution time.

The third implementation in Table 1 exploits parallelism at thread level. The optimized software implementation is modified with OpenMP pragmas in order to decode different utterances in parallel on multiple threads. More than the speed increase, this OpenMP implementation illustrates the suitability of the WFST-based backend search for parallelization across multiple threads without significant memory overhead. The static WFST network is shared between

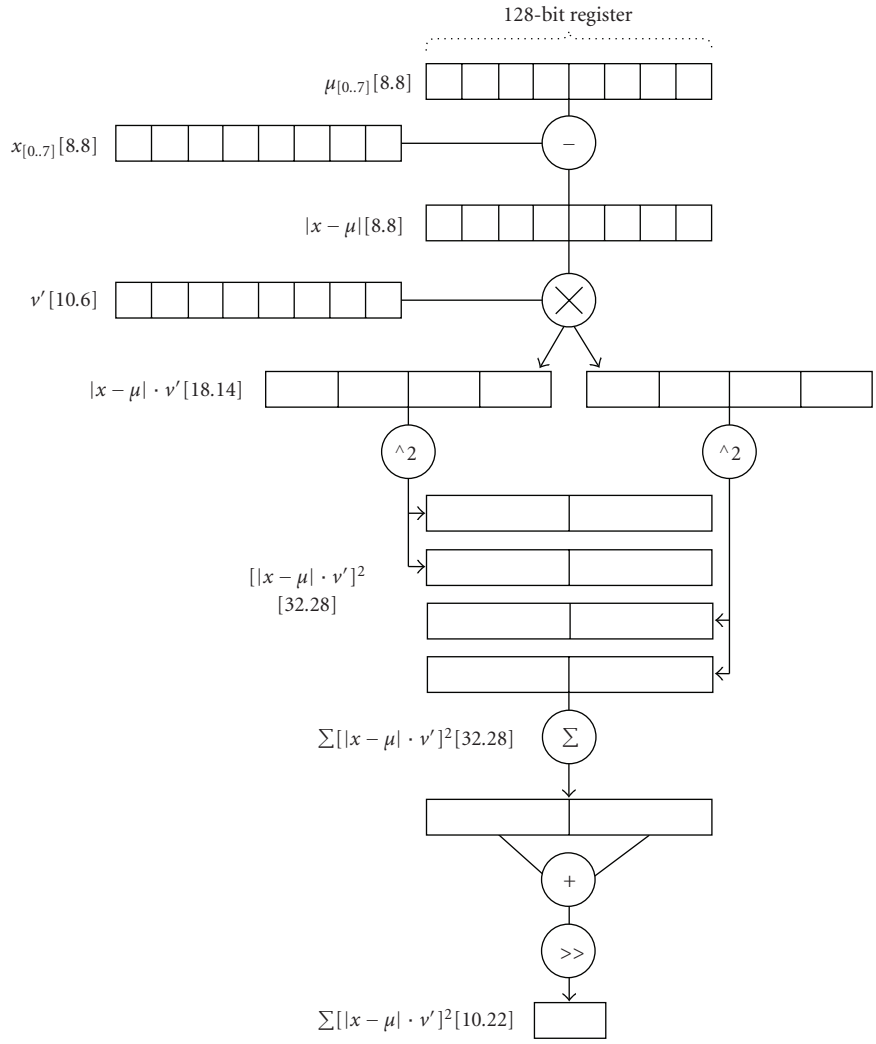

Figure 5: SIMD implementation.

threads, which compensates for its large size. This typically addresses server applications where multiple, independent utterances are decoded in parallel.

So our requirement for our IP core is to be able to address embedded applications, where real-time, low-latency speech recognition must be achieved at low power, and server applications where overall speed is of primary concern. This is why the Gaussian core has been designed with flexibility in mind, in order to fulfill both these roles with one piece of IP. The hard coded custom IP peripheral must be capable of handling the Gaussian calculation in a pipelined and parallel manner if we are to achieve real-time operation with memory bandwidth and clock frequency reductions.

4. FPGA Design Decisions

Even with the SIMD instructions, the Gaussian core is responsible for 48% of the program's execution time. An FPGA-based hardware accelerator capable of operating faster than the software implementation would allow faster decoding of speech. Also, this performance will not be reproducible in an embedded system since processor frequencies are substantially lower and in the case of softcore processors such as the Microblaze, SIMD instructions are not available. In this case, a hard coded custom IP peripheral capable of handling the Gaussian calculation in a pipelined and parallel manner is necessary if we are to achieve real-time operation. In this section, the design decisions involved in FPGA implementation of the Gaussian calculation are discussed.

4.1. Data and Storage. The data required, in order to calculate Gaussian results, comprises acoustic models, which are predefined and stored locally, and acoustic observation vectors which are discrete representations of input speech. During software testing, the dynamic range of each variable was assessed, and a fixed point representation was chosen in order to reduce storage and bandwidth requirements without loss of accuracy in the results. Acoustic observation vectors are stored as 16 bit (8.8) fixed point per coefficient. Acoustic

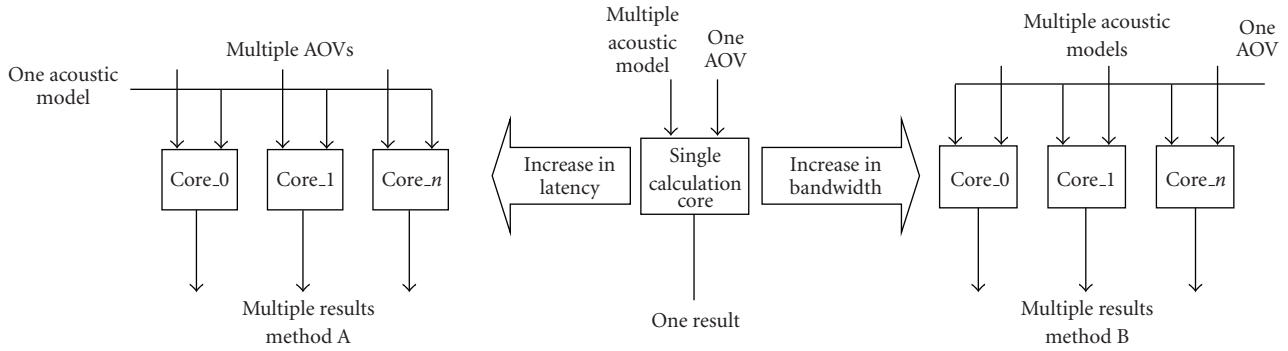

Figure 6: Option for parallelization of the Gaussian calculation.

Models use 16 bits (8.8) for each mean coefficient and 8 bits (2.6) for each variance coefficient with 32 bit weighting per model. Applying these width constraints to the test data results in a mean error of 0.85% in the Gaussian results when compared to the optimum values calculated from a double precision floating point implementation. This error does not measurably affect the word error rate of the decoded speech.

The current fixed point representation of the acoustic models requires 4 500 KB storage space. This is too large to be stored in on-chip Block RAM which for Virtex 5 FPGAs is limited to 2000 KB [18]. Therefore, off-chip RAM is necessary. Although the current set of acoustic models could be stored in SRAM in order to allow for the possibility of increased quantities of acoustic models, the decision was made to use SDRAM for storage. The SDRAM controller used in this design is a custom IP core provided by the board manufacturer, Alpha Data, and was chosen as it was already proven to work successfully with our test hardware. Other solutions are available but were not investigated as this is not the main focus of the work.

4.2. Partitioning. In order to successfully partition the system, communication must be minimized between the Gaussian calculation block and the backend search. This allows the Gaussian calculation to be efficiently implemented as a coprocessor in an embedded system or as an FPGA-based hardware accelerator without the risk of creating a communications bottleneck. In theory, it is not necessary to calculate every Gaussian result for every AOV since results will only be needed for the active arcs in each iteration. For the software implementation, results are calculated as needed resulting in an average number of calculations of 678 per frame from a theoretical maximum of 3 825. In practice, however, if it is possible to calculate all results for each frame, this can be beneficial since this means that there is no need for the backend search to request specific results; instead, the Gaussian block can work independently as long as a full set of results is available at the beginning of the corresponding iteration. Another benefit of this brute force approach is that it simplifies the loading of acoustic models. Since all are being computed, so they can be loaded in the order in which they are stored and in bulk rather than addressing individual models as would be necessary if they were computed on demand. The requirement of the Gaussian block is simple and clearly defined. For each period of 10 ms, one Gaussian result must be calculated for each acoustic model with respect to the current AOV.

4.3. Parallel Calculation. Once we have moved from a request-based system where Gaussian results are calculated individually on demand to a brute force system where results are calculated in bulk ahead of time, we have the opportunity to parallelize the calculation of Gaussian results. This can be done in two different ways, each with its own associated cost as shown in Figure 6. The first approach, method A, involves buffering the input speech of the system. This allows the use of a single acoustic model against multiple observation vectors to calculate multiple results in parallel. The alternative, method B, is to use a single observation vector with multiple acoustic models in parallel. This approach does not require buffering of the input speech but would require an increase in bandwidth used for loading acoustic models to achieve similar results.

The most appropriate choice will depend on the goal of the system and could involve a combination of the two. For example, input buffering will increase the overall latency from input speech to transcription output. Since each AOV represents 10 ms of input speech, this is the amount of buffering that is required for each parallel calculation in method A. For real-time applications, a small amount of buffering is acceptable but the threshold must be kept low in order to ensure usability. On the other hand, a server system running at an order of magnitude faster than real time on multiple streams will be a lot less sensitive to delays caused by buffering.

5. FPGA Implementation

The Gaussian core has been designed with efficiency and flexibility in mind. The calculation itself is fairly simple but must be carried out on a large amount of data at high speed. Since the same calculation is carried out on multiple models independently, a high degree of parallelism can be exploited. For this reason, the approach taken in designing the Gaussian core was to first build a single, efficient pipeline with minimal control and then to build a parallel architecture containing

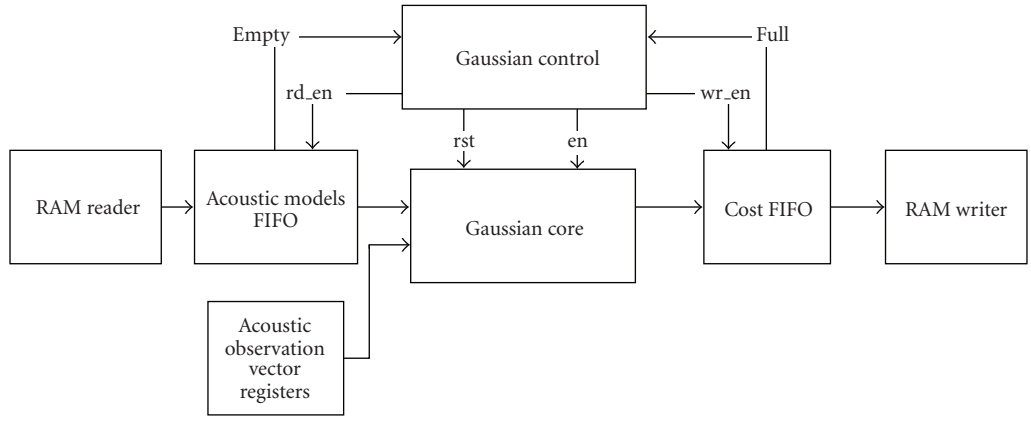

Figure 7: Single core implementation of Gaussian core.

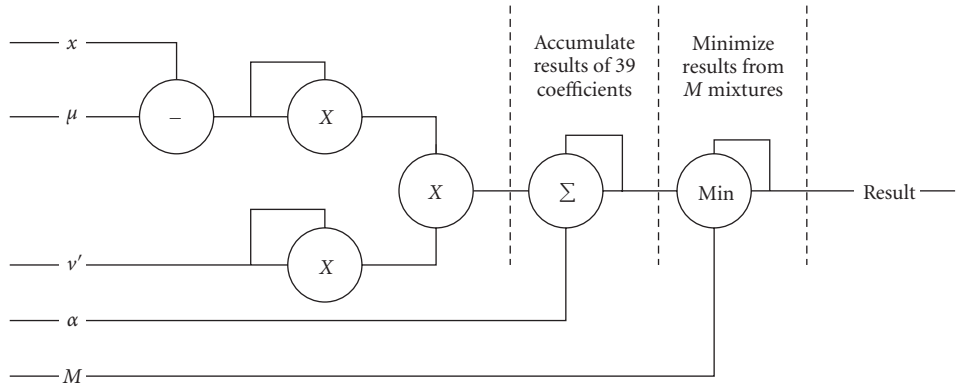

Figure 8: Data flow of Gaussian core.

multiple pipelines which could be configured to achieve specific design goals. In this section, we will first present the single core implementation which was first proposed in the previous paper [12] and then describe a number of new multi-core architectures that have been implemented since the original publication, in order to provide a solution tailored to the required performance of a range of speech recognition systems.

5.1. Single Core. The Gaussian block is the main processing core of the Gaussian calculation. It consists of control and data flow elements and is designed to read inputs from a set of FIFOs and write outputs to another FIFO. The use of FIFOs reduces the complexity of the control logic since it is not required to run the SDRAM interface. Separate modules have been implemented which read acoustic models from SDRAM and write them into the FIFOs and similarly, results are read from the output FIFO and written to SDRAM. This decoupling provides the benefit that the core could be implemented on a completely different platform with a different storage solution without the need to alter the control logic. Figure 7 shows the architecture of a single Gaussian core implementation.

The control component is an FSM which monitors the level of the input FIFOs in order to ensure that enough data is available to start a calculation. It then provides suitable read enable signals to the input FIFOs and coordinates the reset and enable signals to the data core. The control unit also monitors the done signal of the data core and provides a corresponding write enable signal to the output FIFO. The data flow of the Gaussian block implements (4) as a pipeline which can be split into three sections as shown in Figure 8.

$$a = \left((\mu - x) \times v'\right)^2. \tag{5}$$

The first section calculates (5) for every acoustic model Gaussian coefficient. The results of the arithmetic section are then accumulated over the 39 values of a that correspond to one mixture and added to the weight, α, of that mixture. The final stage of the pipeline finds the minimum of a set of results from the accumulation which correspond to one model.

The device utilization results of the single core implementation can be seen in Table 2. It should be noted that the majority of the FPGA resources used in this implementation are accounted for by the SDRAM controllers; this is evident by the small increase in utilization between the single and double core implementations. The single core has been tested and requires 314 clock cycles on average to compute the result of a single acoustic model consisting of 8 mixtures against a single AOV; this requires a minimum

Table 2: Method A. device utilization on Virtex 5 SX95T FPGA (+did not meet timing for 266 MHz.).

	Single core	2 cores	8 cores	16 cores	32 cores	64 cores+
Occupied Slices	8104(55%)	8782(59%)	9460(64%)	10,485(71%)	12,889(87%)	14,557(98%)
Slice LUTs	18,902(32%)	19,306(32%)	21,479(36%)	24,729(41%)	33,188(56%)	42,976(72%)
Slice Registers	14,669(24%)	15,082(25%)	16,976(28%)	20,005(33%)	28,653(48%)	37,325(63%)
BlockRam (Kb)	252(2%)	288(3%)	504(5%)	864(9%)	1512(17%)	2808(31%)
DSP48es	8(1%)	14(2%)	50(7%)	98(15%)	194(30%)	386(60%)

clock frequency of 120 MHz for real-time reconstruction. The design has been successfully placed and routed using Xilinx ISE 11.5 and has passed timing with a clock frequency of 133 MHz for the Gaussian core.

5.2. Multi-Core-FPGA Accelerator. In this section, two different multi-core implementations are presented. The first is based on a buffered input system and is new for this paper. The second was first proposed in [12] and uses parallel acoustic models on a single Acoustic Observation Vector.

5.2.1. Method A Buffered AOVs. The simplest implementation is the buffered AOVs where the same data FIFOs and control logic can be used with multiple data flows. This is illustrated in Figure 9. This option requires the same external memory bandwidth from the RAM reader component and does not significantly increase the on-chip block RAM storage. This design has been implemented with a range of parallel cores, the results of which are presented in Table 2. There is obviously a large overhead in terms of slice utilization due to the RAM controllers and PCI interface logic, which is evident by the fact that the growth in slice utilization with the increase in number of cores is very small, compared to the figures for slice utilization of the single core. It is also evident that DSP slices are mostly used by the Gaussian core itself, due to the almost linear increase of 6 DSPs per core with very little overhead. Block RAM usage in this design is low and does not significantly increase with the number of cores. In fact, the block RAMs used per core actually decrease with the number of cores. This makes sense as the block RAMs are mostly used for the input FIFOs which are not replicated for additional cores. The increase in total block RAMs seen in the highly parallelized designs is due to increased complexity of the output FIFO which is implemented with asymmetric input/output widths in order to combine the results of multiple cores into a single output data stream.

All designs have been synthesized, mapped and placed and routed using Xilinx ISE 11.5. With the exception of the 64-core implementation, all designs passed timing for a clock rate of 133 MHz for the Gaussian core and 266 MHz for the SDRAM controllers. The 64 core implementation can be run with a Gaussian clock of 60 MHz and 120 MHz for the SDRAM controllers.

As noted in Section 5.1, a single core is capable of calculating a full set of Gaussian results in 10 ms at a clock frequency of 120 MHz. The speed increase due to parallel cores in this implementation is linear and directly proportional to the number of parallel cores, since the input data is the same for all cases. The results calculated by the Gaussian core are 16 KB per core per frame meaning that the output bandwidth will be 1.5 MBs^{-1} per core. Assuming that this bandwidth does not exceed that of the output bus, the linear speedup will be maintained; hence the 32-core implementation running at 120 MHz will provide 32 times real-time performance.

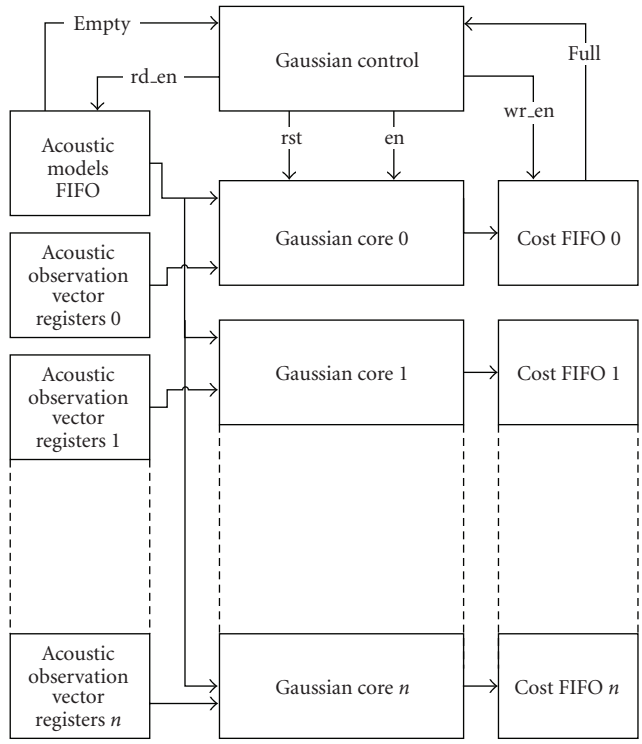

Figure 9: Method A. parallel implementation of Gaussian core.

5.2.2. Parallel Acoustic Models. An option which allows parallelization without input buffering is to use the same input data on multiple acoustic models concurrently. This is illustrated in Figure 10 which shows that the system will require more on-chip FIFOs to store acoustic models along with replicated control logic for each data flow. Although this system does not increase the buffering of input speech to the system, there is a limit to the degree of parallelization available due to the increase in bandwidth.

The results of implementing up to 4 parallel Gaussian cores using this method are presented in Table 3. The main

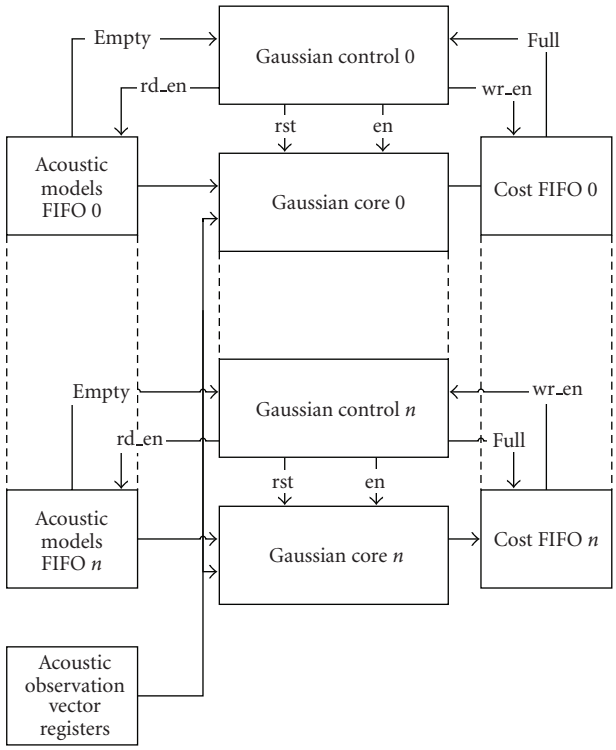

Figure 10: Method B. parallel implementation of Gaussian core.

difference in the results of this design compared to that shown in Figure 9 is the almost linear increase in block RAMs used. This is due to the fact that input FIFOs which are implemented in block RAMs are replicated for each core. The limit of the FPGA used for this project is approximately 8 Kb which suggests that up to 64 cores could be implemented, although the real limit of this design will be memory bandwidth which will be reached much sooner.

The test data used for our proof of concept system uses 1 152 bytes per model. This equates to a total data size of 4 500 KB for 4000 models. This means that real-time operation of a single core at 100 fps will require 439 MBs^{-1} and means each subsequent core will increase the bandwidth by this same amount. A 64 bit DDRSDRAM chip running at 266 MHz will provide approximately 2 GBs^{-1} and so the limit would be, at best, 4 cores per SDRAM chip.

As with the buffered AOV approach described in Section 5.2.1, a linear speed increase, directly proportional to the number of parallel cores, is theoretically possible; however, in this case, the speed increase will be dependent on the bandwidth at the input of the system. All the designs in this section have been placed and routed and have passed timing at the full system speed of 133 MHz with the SDRAM clock running at 266 MHz.

6. Conclusions

We have presented a versatile pipelined Gaussian core along with multiple implementation architectures. The single core implementation has been proven as a real time solution suitable as an embedded peripheral for low power mobile systems. Parallel implementations providing better than real-time have been demonstrated with two degrees of flexibility. This flexible design has the potential to either accelerate or reduce the power consumption of any speech recognition system which uses Gaussian Mixture Models for acoustic modeling.

Table 3: Method B. Device utilization on virtex 5 SX95T FPGA.

	Single core	2 cores	4 cores
Occupied Slices	8,601(58%)	9,117(61%)	9,516(64%)
Slice LUTs	18,618(31%)	19,673(33%)	21,288(36%)
Slice Registers	14,212(24%)	15,597(26%)	16,895(28%)
BlockRam (Kb)	288(3%)	540(6%)	1,044(11%)
DSP48es	8(1%)	14(2%)	26(4%)

As detailed in Section 2, the Gaussian core is part of a larger speech recognition system and must be paired with a feature extraction component and a backend search. For the purposes of testing and proof of concept, these components have been implemented as software running on the host PC. It is intended that in the final system these components will be carried out using embedded processor such as the softcore Microblaze or the dual-core Arm processors provided by the Xilinx Extensible Processing Platform. Pairing the Gaussian core presented here with an ARM based backend search has the potential to provide a fast and accurate low power portable speech recognition system. The main challenge of such a system will be the efficient utilization memory channels when loading the WFST data. This will be addressed in a future publication.

Acknowledgments

The authors would like to thank the Engineering and Physical Sciences Research Council for the support provided under grant number EP/D048605/1, Xilinx for generously donating the FPGAs and software used in this project, and Alpha Data for help and advice.

References

[1] O. Viikki, I. Kiss, and J. Tian, "Speaker- and language-independent speech recognition in mobile communication systems," in *Proceedings of IEEE Interntional Conference on Acoustics, Speech, and Signal Processing*, pp. 5–8, May 2001.

[2] A. Bernard and A. Alwan, "Low-bitrate distributed speech recognition for packet-based and wireless communication," *IEEE Transactions on Speech and Audio Processing*, vol. 10, no. 8, pp. 570–579, 2002.

[3] N. Leavitt, "Let's hear it for audio mining," *Computer*, vol. 35, no. 10, pp. 23–25, 2002.

[4] S. Douglas, D. Agarwal, T. Alonso et al., "Mining customer care dialogs for 'daily news'," *IEEE Transactions on Speech and Audio Processing*, vol. 13, no. 5, pp. 652–660, 2005.

[5] W. Walker, P. Lamere, P. Kwok et al., "Sphinx-4: a flexible open source framework for speech recognition," Sun Microsystems Whitepaper, 2004.

[6] S. J. Young, D. Kershaw, J. Odell, D. Ollason, V. Valtchev, and P. Woodland, *The HTK Book Version 3.4.*, Cambridge University Press, Cambridge, Mass, USA, 2006.

[7] E. C. Lin, K. Yu, R. A. Rutenbar, and T. Chen, "A 1000-word vocabulary, speaker-independent, continuous live-mode speech recognizer implemented in a single FPGA," in *Proceedings of the 15th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '07)*, pp. 60–68, February 2007.

[8] O. Cheng, W. Abdulla, and Z. Salcic, "Hardware-software co-design of automatic speech recognition system for embedded real-time applications," to appear in *IEEE Transactions on Industrial Electronics*.

[9] E. C. Lin and R. A. Rutenbar, "A multi-FPGA 10x-real-time high-speed search engine for a 5000-word vocabulary speech recognizer," in *Proceedings of the 7th ACM SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '09)*, pp. 83–92, February 2009.

[10] K. You, H. Lim, and W. Sung, "Architectural design and implementation of an FPGA softcore based speech recognition system," in *Proceedings of the 6th IEEE International Workshop on System on Chip for Real Time Applications (IWSOC '06)*, pp. 50–55, December 2006.

[11] M. Mohri, F. Pereira, and M. Riley, "Weighted finite-state transducers in speech recognition," *Computer Speech and Language*, vol. 16, no. 1, pp. 69–88, 2002.

[12] R. Veitch, L.-M. Aubert, R. Woods, and S. Fischaber, "Acceleration of hmm-based speech recognition system by parallel fpga gaussian calculation," in *Proceedings of the 6th Southern Conference on Programmable Logic*, 2010.

[13] L. R. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989.

[14] J. Chong, Y. Yi, A. Faria, N. Satish, and K. Keutzer, "Data-parallel large vocabulary continuous speech recognition on graphics processors," in *Proceedings of the 1st Annual Workshop on Emerging Applications and Many Core Architectures*, June 2008.

[15] S. Molau, M. Pitz, R. Schlüter, and H. Ney, "Computing mel-frequency cepstral coefficients on the power spectrum," in *Proceedings of IEEE Interntional Conference on Acoustics, Speech, and Signal Processing*, pp. 73–76, May 2001.

[16] B. Milner and X. Shao, "Clean speech reconstruction from MFCC vectors and fundamental frequency using an integrated front-end," *Speech Communication*, vol. 48, no. 6, pp. 697–715, 2006.

[17] E. Bocchieri and D. Blewett, "A decoder for LVCSR based on fixed-point arithmetic," in *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP '06)*, pp. 1113–1116, May 2006.

[18] Xilinx, "Virtex-5 Family Overview," http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf.