Compute & Memory Optimizations for High-Quality Speech Recognition on Low-End GPU Processors

Kshitij Gupta and John D. Owens

Department of Electrical & Computer Engineering, University of California, Davis
One Shields Avenue, Davis, California, USA
{kshgupta,jowens}@ucdavis.edu

Abstract—**Gaussian Mixture Model (GMM) computations in modern Automatic Speech Recognition systems are known to dominate the total processing time, and are both memory bandwidth and compute intensive. Graphics processors (GPU), are well suited for applications exhibiting data- and thread-level parallelism, as that exhibited by GMM score computations. By exploiting temporal locality over successive frames of speech, we have previously presented a theoretical framework for modifying the traditional speech processing pipeline and obtaining significant savings in compute and memory bandwidth requirements, especially on resource-constrained devices like those found in mobile devices.**

In this paper we discuss in detail our implementation for two of the three techniques we previously proposed, and suggest a set of guidelines of which technique is suitable for a given condition. For a medium-vocabulary, dictation task consisting of 5k words, we are able to reduce memory bandwidth by 80% for a 20% overhead in compute without loss in accuracy by applying the first technique, and memory and compute savings of 90% and 35% respectively for a 15% degradation in accuracy by using the second technique. We are able to achieve a 4x speed-up (to 6 times real-time performance), over the baseline on a low-end 9400M Nvidia GPU.

I. Introduction

A paradigm shift is under way in the computing world where both hardware (processor architectures) and software (programming models) are moving away from sequential to parallel computing. This shift is being driven by the need for more compute power required for processing current and emerging compute workloads. Amongst them, an unlikely front-runner has emerged in the form of graphics – used in games for entertainment, and perhaps more importantly as a means of providing a rich user experience – catapulting the GPU to one of the most essential components in any consumer electronic system.

The incorporation of *unified shaders* and programming languages like CUDA [1] and OpenCL [2] has extended the potential use of the GPU to non-graphics workloads that exhibit data- or thread-level parallelism. Over the years, GPGPU programming has made several strides and there are very few application domains that have yet to see benefit from using the GPU [3]. However, moving applications that have been optimized for one processor family (CPU) onto another (GPU) is often times a non-trivial task, requiring a re-design of the application to suit the underlying processor architecture and programming model.

To this end, a majority of the work in literature has to date primarily focused on suggesting modifications for the sole purpose of extracting maximum speedup. The most constrained part in any product today, however, is the memory system. Unlike the past where discrete GPUs were more popular for desktop systems, the recent trend towards widespread deployment of portable electronics is necessitating a move towards integrated GPUs, heterogeneous processors, which are based on a unified memory architecture (CPU and GPU SoC) like the NVIDIA Tegra [4] and AMD Fusion [5]. As the CPU and GPU reside on the same die and share the same physical memory, a higher memory access contention is bound to happen in these systems, which will lead to further exasperation of the memory bottleneck when data is requested simultaneously by these two processors. We therefore believe that in order for the full potential of GPUs to be realized (not just limited as a *co-processor*, but an *applications processor*) in such platforms, a focus on optimizing memory accesses must be made for achieving highly efficient systems in the future.

In this paper we focus on both compute and memory access patterns of an important form of Natural User Interface (NUI) that will play a central role in how we interact with technology – Automatic Speech Recognition (ASR). High-quality, continuous speech recognition is compute- and memory-bandwidth-intensive, requiring 10's of GFLOPs of compute and 100's of MB/sec of bandwidth, consuming significant system resources especially for an 'always on' speech-driven interface. While the computations within the ASR system are a good fit for the data- and thread-level parallelism provided by the GPU, the unmodified CPU formulation of the speech pipeline requires far higher memory accesses, making ASR a good reference application for our exploration. More specifically, we focus on the core compute- and memory-bandwidth-intensive portions in the ASR pipeline related to GMM computations in this paper.

As speech-driven interfaces are best suited in mobile, on-the-go scenarios, our optimizations are aimed at addressing some of the key challenges that are a part of small form-factor devices – limited compute, memory bandwidth, and stringent power consumption requirements. Since ultra-mobile GPGPU-capable processors found in smartphones and tablets are not yet available for mainstream developer consumption, we limit our current focus to the lowest-end GPU processor available, NVIDIA's 9400M processor.

The remainder of the paper is organized as follows. We begin by discussing in greater detail specific architectural challenges GPUs exhibit, especially in the memory context, in Section II. In Section III, we provide a brief overview of ASR followed by our motivation and a detailed description of modifications for fast GMM compute in Section IV. We present implementation details of our GMM compute routines showing significant savings in both compute and memory bandwidth on a 5k-word dictation task in Section V, followed by a discussion along with a comparison of this work with other known GPU implementations in Section VI. We conclude by pointing to future directions of our work.

II. Memory: A Critical Resource

A major distinguishing factor between CPU and GPU processors is the difference in ratio of area devoted to compute and memory resources. While the CPU is designed for handling highly irregular workloads, modern GPUs are architected as throughput machines that are well suited for large workloads that present a high degree of data- and thread-level parallelism. This fundamental difference in the kind of workload these processors handle leads to a distinct architectural choice: unlike the traditional CPU which focuses on latency hiding through a hierarchy of large on-chip caches and out-of-order execution for extracting parallelism from sequential workloads, the GPU architecture is comparatively simple, one that devotes a significant portion of on-chip silicon to compute resources with small caches and special-purpose memories.

Although techniques to hide memory latency differs between the CPU and GPU, it must be noted that due to small caches on the GPU, lack of support for temporal locality requires that data be fetched from global memory[1]. Like other aspects of GPU programming where a greater onus of extracting performance is left to the programmer (rather than the compiler) by providing low-level control to certain elements in the processor, extraction of temporal locality in the case of the GPU is also left to the programmer. This is especially important since access to global DRAM memory is costly both in terms of performance (often the limiting factor in achieving peak throughput performance) and power (contributing a significant percentage of total power drawn in embedded systems).

As the cost of memory access is high, an ideal system would have a higher ratio of compute-to-memory access (FLOPS/Byte). We refer to this ratio as the *arithmetic intensity* factor. The degree of arithmetic intensity is oftentimes beyond the control of the programmer, being dependent on the algorithm instead, which the programmer must once again try to work around, if possible.

[1] The GeForce 9400M GPU we use in this study does not have a L2 cache and therefore has no way of exploiting temporal locality, requiring data to be loaded from DRAM for every frame of data processed; the newer NVIDIA Fermi GPU family, however, does have an L2, albeit a fraction of the size of modern CPUs.

Lastly, as chip architectures and programming models have evolved, it is becoming easier to implement larger portions of an application on the GPU. However, there is still a significant limitation in realizing entire applications as not all portions of an application benefit from GPU-like architectures. The CPU and GPU have traditionally been linked by slow system interfaces like PCI-Express, whose memory throughput is significantly lower, and data transfer latencies quite often larger than the speed-up obtained by offloading a task to the GPU, thus limiting the usefulness of application deployment using the GPU.

We see heterogeneity moving from the board-level to the chip-level as the solution to this problem, with both CPU-GPU on the same die (like NVIDIA's Tegra and AMD's Llano processors), and a dedicated communication link between the two (like in Intel's Sandy Bridge family) as a natural evolution to platform architectures. This change, while addressing communication issues between CPU and GPU, will introduce a different kind of bottleneck – as the number of number I/O pins on a chip is unlikely to change due to economical and practical considerations, these high-performance heterogeneous processors will lead to a greater contention between the CPU and GPU for memory accesses. In effect, the cost of memory transactions is likely to remain constant while the cost of compute gets increasingly cheaper. Application performance will therefore not be limited by compute capabilities, but memory.

Due to these reasons, it is no longer sufficient to design or modify an application/algorithm without studying and optimizing for memory bandwidth and access patterns. In the remainder of this paper, through an example ASR application, we present systematic modifications to the traditional algorithm pipeline relating to the log-likelihood score computations in the Acoustic Modeling phase, and show how these modifications leads to an efficient system, while addressing all aspects discussed in this section.

III. Example Application: Automatic Speech Recognition

Several techniques have been developed over the past few decades for performing computer-based speech recognition, often referred to as Automatic Speech Recognition (ASR). Out of these techniques, Hidden Markov Modeling (HMM) and Artificial Neural Networks (ANN)-based approaches have stood-out for continuous, naturally spoken speech by yielding fairly accurate results. The goal of any ASR system is to classify incoming voice patterns into a sequence of most likely spoken words with the help of trained models that are compiled offline. The problem of obtaining this sequence of words can be represented by the following equation:

$$\widehat{W} = \frac{argmax}{W}\{P(O/W) * P(W)\} \qquad \text{Eq. 1}$$

where, $P(O/W)$ is the posteriori probability of observing the observation vector given a word, and $P(W)$ is the a priori probability of observing a word. Word and sequence-of-word probabilities, $P(W)$, are computed offline on large sets of text corpora that contain all words the system is designed to recognize. In N-gram systems, these probabilities are represented as probabilities of word tuples (*1, 2, .. N*). Since the word and language knowledge sources are built offline, most operations in this part of the system rely on memory look-ups.

The computation of posteriori probability, $P(O/W)$, is performed during run-time giving the probability of speech segments matching one of the acoustic sounds. In a HMM-based system, like the one used in this research, this posteriori probability is modeled by HMMs. Every word could theoretically be represented by its own HMM, but that would yields poor results for dictionaries with more than tens of words due to insufficient training data. Further, as speech can be represented by a sequence of basic sound units, *phonemes*, there would be a lot of redundancy in computation.

For this reason, words in most high-end systems are modeled by their corresponding phonetic sequences, where every phoneme is represented by a HMM, each modeled by a 3- or 5-state Bakis topology. States within the HMM represent sub-phonetic (acoustic) sounds that are statistically modeled by a mixture of Gaussians, often referred to as the Gaussian Mixture Model (GMM). A collection of GMM's is referred to as the Acoustic Model (AM). It should be noted that the granularity of distribution within the GMM plays an important role in determining the robustness, accuracy, and the nature of speech that the system can decipher (discrete or continuously spoken words). Gaussian likelihood probability computation is compute-intensive, and consumes a majority of processing time.

All knowledge sources in the system are laid out in a hierarchical manner: language, words, phonemes, and acoustics. Speech is processed in frames that are produced in 10 msec intervals. The process of generating valid hypotheses consists of two major phases as shown in Fig. 1(a). The first phase involves information flow: lists of speech units that can be hypothesized for the current frame based on information from prior frames are propagated from higher level knowledge sources to lower levels sources. The second phase involves all compute operations related to probability generation, look-up and pruning with a set of all possible hypothesized words for the current frame. This process is performed iteratively for every frame, and at the end of the spoken utterance, the best scoring path of all possible spoken words is output as the final sequence of recognized words.

Although each step in Fig. 1(a) is sequential with either data or information coming from previous stages, there is enormous parallelism within every stage of both phases due to limited data dependencies within each stage. Effectively, both generate and score phases allow for efficient extraction of thread- and data-parallel operations on modern parallel processors like the GPU, and hence are well suited for our application.

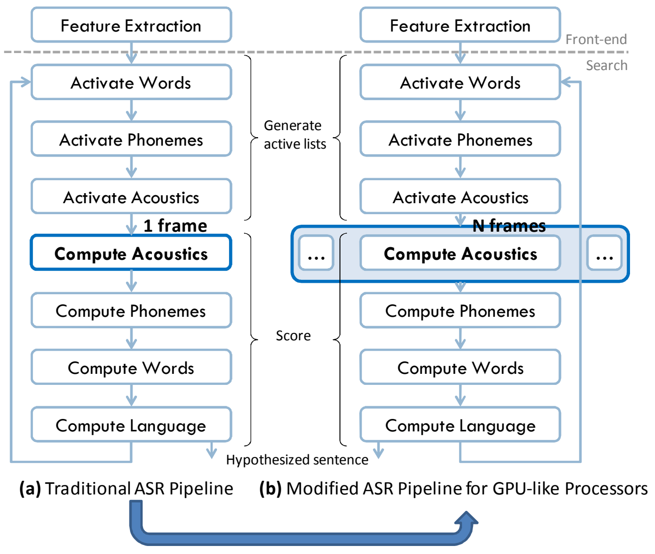

Fig. 1. A conceptual diagram representing a typical ASR pipeline is depicted in (a) while our modified pipeline is shown in (b).

A. GMM Compute Overview

The Acoustic Modeling knowledge source is a collection of all HMM states for a given target vocabulary, with each state being represented by a Gaussian Mixture Model (GMM). Every GMM comprises of a group of multi-variate Gaussians, i.e. multi-dimensional Gaussians. For the ASR case, the dimensions can be approximated to be independent from one another, thereby simplifying the Gaussian co-variance matrix into a diagonal one. The *score* for GMM *g* over *M* mixtures, each with a weight of *W* over *D* dimensions, is computed by Eq. 2.

$$score_g = \sum_{m=1}^{M}\left(\frac{W_{g,m} * e^{\sum_{d=1}^{D}\frac{\left(x_d-\mu_{g,m,d}\right)^2}{2\sigma_{g,m,d}^2}}}{\sqrt{(2\pi)^D\left|\sigma_{g,m,d}^2\right|}}\right) \qquad \text{Eq. 2}$$

This computation can be simplified by taking the log on both sides, thus replacing exponent and division operations by addition and multiplication. Upon pre-computing constant terms that don't rely on speech features generated during run-time, *x*, log-likelihood computation, *score'*, can be reduced to Eq. 3.

$$score'_g = log\sum_{m=1}^{M}\Bigg(W'_{g,m} + \mathrm{K} + \sum_{d=1}^{D}\left(x_d^2 * \sigma'^2_{g,m,d} - x_d * \sigma''^2_{g,m,d}\right)\Bigg) \qquad \text{Eq. 3}$$

where, $W'_{g,m} = log\left(W_{g,m}\right)$

$$\mathrm{K} = log\left(1/\sqrt{(2\pi)^D\left|\sigma_{g,m,d}^2\right|}\right) + \sum_{d=1}^{D}\frac{\mu_{g,m,d}^2}{2\sigma_{g,m,d}^2}$$

$$\sigma'^2_{g,m,d} = 1/2\sigma_{g,m,d}^2$$

$$\sigma''^2_{g,m,d} = 2 * \mu_{g,m,d} / 2\sigma^2_{g,m,d}$$

The summation of scores for all mixtures transforms from an addition to a log-addition (addition in the log domain). As log-likelihood scores have a small dynamic range, they are multiplied by a scaling factor, *f*, to linearly magnify the dynamic range. The final simplified equation is shown in Eq. 4.

$$score'_g = f * log \sum_{m=1}^{M} \left(W'_{g,m} + \mathrm{K} + SUMM\right) \quad \text{Eq. 4}$$

where, $SUMM = \sum_{d=1}^{D} \left(x_d^2 * \sigma'^2_{g,m,d} - x_d * \sigma''^2_{g,m,d}\right)$.

In summary, all operations are reduced to multiplication and summation operations. Further optimizations are possible on the GPU as these operations can be fused into a single multiply-add operation, further reducing the arithmetic intensity. Every memory load therefore corresponds to a single arithmetic instruction.

IV. Optimizations for GMM Compute

A. Motivation

One key challenge for efficient implementation of ASR on lower-end, low-cost devices is memory accesses – both memory size and access patterns. The ASR computing workload has been found to have low IPC, and high L2 miss rates that prove detrimental in realizing efficient systems [7, 8]. While modern desktop systems now have enough area devoted for larger caches (measured in MBytes) that can help mitigate performance hit due to cache misses, it is unlikely that this will happen in low-end processors aimed at small form-factor devices in the near future. Further, workloads on GPU-like processors are likely to continue to necessitate larger area to compute resources and lesser to memory. The only solution for addressing these challenges in the context of ASR was for us to re-visit the traditional pipeline and explore modifications that fit well within the parallel processing paradigm, with little impact to accuracy.

From Fig. 1(a), the flow between stages is inherently sequential, with the current stage requiring information from the previous stage. Theoretically, it is possible to de-couple the sequential nature by ignoring the feedback lists, and following a brute-force approach – computing all units of speech at every stage for every frame of speech input. If this approach was followed, it would be possible to theoretically 'batch-process' frames, processing several frames for every load of data from various knowledge sources. The size of these knowledge sources, however, is non-trivial (of the order of 100's of MB for large systems), and would incur significant overhead in terms of both compute and memory bandwidth if the entire knowledge source was read for every batch of frames to be processed. On lower-end GPU-like processors that are constrained for both memory and compute resources, this is not a feasible solution. Managing active data through the use of feedback lists is therefore an important part for realizing practical systems that can be deployed in real-world scenarios.

In its most simplified form, with pre-computations and re-arrangement of likelihood scores of multi-variate Gaussians with a diagonal co-variance, the operations in GMM compute can be reduced to one arithmetic operation (tens of cycles) per every memory load (hundreds of cycles of latency) as shown in Section III. In order to achieve maximum benefit from GPU-like parallel architectures, the arithmetic intensity of these operations must be improved to address the disparity between memory and compute operations – the more the number of operations that can be performed per memory load, the better would be the utilization of underlying hardware compute resources.

Lastly, as stated earlier, caches play an important role in helping achieve good performance on CPU-based systems. The lack of memories that provide temporal locality of data on-chip in GPU-like processors is perhaps the most important reason for revisiting the traditional ASR pipeline, since for every frame of speech, data needs to be loaded from main memory.

B. Theoretical analysis & suggested modifications

All issues mentioned above can be resolved with one insight: the nature of temporal locality within successive frames of speech input. An outline of a theoretical analysis of this approach [6]. In this section we provide greater details of how the theoretical analysis could be used, through simple modifications showing how techniques that have been successfully employed on the CPU can be successfully extended with minor modifications on the GPU. While certain aggressive optimizations can lead to significant reduction in memory bandwidth requirement by increasing compute overhead significantly, on lower-end processors, both memory bandwidth and compute resources are scarce and therefore one parameter cannot be optimized at the cost of the other. A balance is essential in realizing an efficient system. Our modifications were therefore aimed at addressing these issues without significantly increasing compute overhead from our baseline, while drastically reducing memory bandwidth requirements with minimal impact on final recognition accuracy.

B.1 Modification # 1: Frame-Level

The rate of change in the speech signal over short periods of time can be slow. Successive frames, which represent 10 msec segments of speech, can sometimes be approximated as being similar. These similar frames can be assumed to provide minimal additional information about the voice, and can therefore be ignored. This is the first-level of approximation that is often used in a few speech systems to reduce the computational intensity of operations in the GMM likelihood computation phase. While this down-sampling does help in reducing the compute load, it quickly introduces error beyond skipping every 3rd frame. Amongst the exploration by Chan et al., this was the least promising technique [9]. Skipping is also detrimental in noisy conditions as signal variations are much larger than in clean-speech conditions.

Since our goal is to use GPUs without significantly increasing error rate, we would like to minimize approximations with acceptable degradation in accuracy. We therefore studied the 'lifetime' of every Gaussian in the model to understand the locality pattern. To our surprise, while there is a lot of variation over successive frames as new hypotheses are formed in the word and language models, good scoring states continue to remain active for several frames. Specifically, from our analysis on the Resource Management 1 (RM1) corpus, we found the average lifetime (number of successive frames for which a GMM is active) was greater than 10 frames.

From this observation, we concluded that significant temporal locality existed at the frame level. Given the high average lifetime, we could assume that once a Gaussian is activated, it would be safe to assume it is active for a fixed number of future frames. However, as can be seen in Fig. 2, many states also have a small lifetime, which means a significant compute overhead could be incurred when dealing with Gaussians with smaller lifetimes.

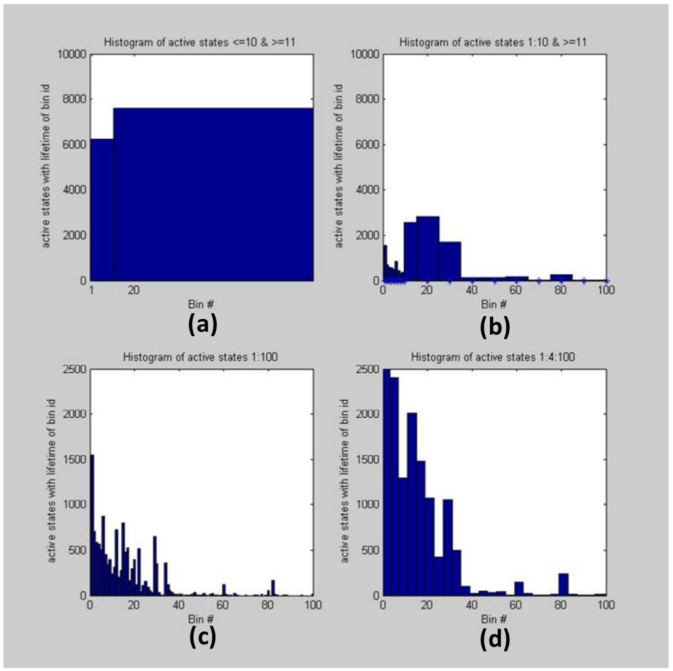

Fig. 2. Histogram of lifetime of GMMs active over successive frames over one utterance consisting of 500 frames of speech over different binning ranges. From (a) & (b), there are many instances when GMMs remain active for more than 10 frames. However, from (c) & (d), many GMMs also have short life-spans and accounting for these GMMs is critical to minimize the compute overhead.

As it is not possible to predict which GMMs are likely to live longer, a balanced approach is required that can account for both small and long lifetimes. Therefore, instead of assigning a fixed lifetime, we use a dynamic chunk-based activation scheme, limiting the frames for which a Gaussians can be active in successive frames to the chunk boundary. At any frame within a chunk if a Gaussian is activated by the higher layers, it is deemed active for the rest of the chunk. We refer to our dynamic, chunk-based look-ahead, as Acoustic Modeling Look-ahead (AML).

A GMM activated at the boundary of the chunk and active throughout represents the best-case scenario for both memory and compute for AML, while a GMM active in the first frame but inactive for the rest represents the worst-case compute scenario as additional frames that would otherwise not be computed would have already been processed. As GMMs are activated closer to the chunk boundary, a lesser savings in memory bandwidth is achieved. The AML process is shown for a few frames in Fig. 3. With this technique we could theoretically obtain 80% savings in memory bandwidth with no loss to the accuracy, at the cost of 20% increase in compute overhead for a chunk size of 8 frames, over the RM1 speech corpus [6].

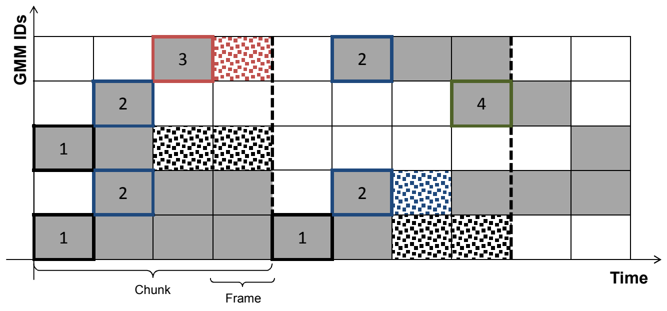

Fig. 3. An example of Acoustic Modelling Look-ahead (AML) is shown, with chunk size of 4 frames. For each frame, an empty block represents an inactive GMM, a greyed block represents an active GMM, and a shaded block represents an inactive GMM that is deemed active due to AML. The GMM is accessed from main memory only during the first frame in which it is active within a chunk, represented by bold outlined blocks. The colours of these blocks represent the frame number within the chunk when they are read from memory.

B.2 Modification # 2: Gaussian-Level

While the incorporation of our AML technique reduces the memory bandwidth requirement, there is a compute overhead associated with this optimization. To address this, we looked at other techniques which are used for reducing compute workload of GMM computations, amongst which one technique stood out. It is based on the nature of GMM computations where-by both the context-independent (CI) and context-dependent (CD) models are combined to form the Acoustic Model. CD-GMMs are specific acoustic instances of their CI-GMMs catering to left and right contexts of phonetic sound, implying that several CD-GMMs correspond to a single CI-GMM. Conversely, every CI-GMM is a generic version of multiple CD-GMMs. This information is used to construct a two-pass approach wherein the first pass is on the coarser, CI-GMM data that help narrow down the CD-GMMs to evaluate in a system based on the score of their CI-GMM computer-parts. If a CI-GMM scores well relative to others for a given frame, CD-GMMs corresponding to it could be worth computing, and if the score is below a certain threshold, CD-GMMs corresponding to those CI-GMMs could safely be approximated to CI-GMM scores without severe repercussions to the accuracy of the system.

In order to extend the AML approach, we analyzed the lifetime of CI-GMMs. We observed that CI-GMMs have significantly smaller lifetimes due to a tight pruning threshold. So the AML approach of a blind look-ahead cannot be used as is, since a CI-GMM tends to be sporadically active leading to a lot of dynamism. In order to regularize the operation, we proposed the use of a simple voting scheme whereby the decision of whether to score or approximate a CD-GMM for a chunk is based on the sum of number of frames for which the CI-GMM scores better than the threshold.

From this modification we were able to reduce the compute workload on the system by over 60% while saving 90% memory bandwidth. The latter savings are comparable to those achieved by AML modifications, with compute savings as an added benefit. Given the promising results of using both these techniques, we implemented our algorithms following these principles.

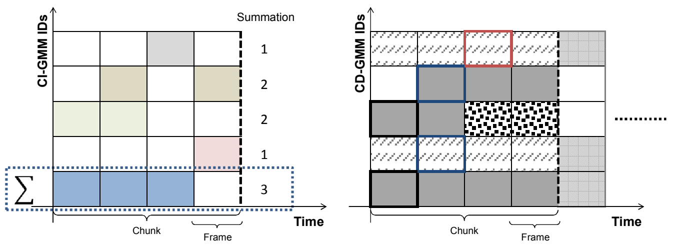

Fig. 4. An example of CI-based Acoustic Modelling Look-ahead (CI-AML) is shown, with chunk size of 4 frames. Left: In phase #1, the CI-GMM scores are computed and the summation of frames for which each CI-GMM is active within a chunk is computed. Right: The result of applying a threshold of 2 frames is shown here. For the sake of simplicity, a one-to-one mapping between CI- & CD-GMMs is shown. All CD-GMMs corresponding to CI-GMMs that are active for less than the threshold value are not computed (shown by dashed fill) and are backed off to base CI-GMM scores instead. All others (rows 1, 3, and 4) are computed just as they would for the regular AML case when the GMM is the first active GMM in the chunk. The squares in bold are shown for reference sake from the base AML case, which indicates the frame when a GMM was computed for that chunk, but CD-GMMs corresponding to rows 2 and 5 are not loaded as they are not computed for this chunk.

V. Implementation & Results

We constructed our system piece-wise, adding additional features to every successive implementation. We began with a brute-force implementation that focused on the computation of the likelihood GMM scores (V-A). Then we incorporated the acoustic active list coming from the Phonetic Modeling block into the Acoustic Modeling phase so as to compute only active GMMs (V-B). We then incorporated our AML technique for chunk-based processing (V-C) and finally included the CI-GMM layer optimization (V-D). Our final implementation is modular and allows us to experiment with each of these settings from within one framework, as required.

We use Sphinx 3, the most widely used open-source ASR system, as the reference system for the work presented in this paper, and use the Wall Street Journal (WSJ0) speech corpus from the Linguistic Data Consortium for our experiments [10, 11]. The WSJ0 test set is a medium-vocabulary dictation test set that contains 5k words extracted from the Wall Street Journal news reports with verbalized and non-verbalized pronunciations. We use the non-verbalized punctuation (nvp) portion of the evaluation set, and selected 80 utterances from the speaker-independent test set to get a reasonable coverage of the entire test suite. Our language model is based on the N-gram model, having over 1.5M bigrams and 2.5M tri-grams. Our Acoustic Models consist of 110k tri-phones, 49 base-phones, 4,000 CD-GMMs and 147 CI-GMMs.

Our implementation was done in OpenCL running on Windows 7, using GPUs from NVIDIA. We chose a low-end system consisting of a GeForce 9400M GPU (with 2 stream multiprocessors) on the NVIDIA ION platform. All data structures were extracted from Sphinx and pre-processed offline for a GPU-friendly layout. Our main GMM knowledge source was converted from a Array-of-Struct layout into a Struct-of-Array format, which is the ideal layout for GPU processing. We compressed all our knowledge sources (mean, variance, pre-computed constant, and mixture weight) into a single linear array. This layout helps us in obtaining fully coalesced memory read accesses for achieving peak bandwidth.

We use four primary metrics while analyzing our results: compute & memory bandwidth overhead or savings (as per context), Word Error Rate (WER) and Real-Time Factor (RTF), where:

$$WER = 100 * \frac{\#\ of\ (insertions + deletions + substitutions)}{\#\ of\ total\ words\ in\ the\ reference\ utterance}$$

$$RTF = \frac{time\ required\ for\ processing\ data\ in\ real\ time}{time\ taken\ by\ GPU\ to\ process\ data}$$

Higher WER figures correspond to lower accuracy. RTF of 1 represents real-time performance, where data is processed (speech recognized) at the same rate at which it is generated (or spoken). RTF greater than 1 represents faster operation, and less than 1, slower than real-time.

When the acoustic feedback list is used, memory accesses become non-contiguous at the GMM-level, but are sequential within the GMM (mean/variance pairs for every Mixture in every Gaussian). Depending on the application and vocabulary size, typical size of the Acoustic Model ranges from G = 2k to 6k, M = 2 to 32, and D = 39. From Section II-B, for the 39-dimension case, every Gaussian requires 80 values (39-mean/variance pairs, the pre-computed constant term, K, and mixture weight, W). So the regularity of memory accesses is pseudo-random, requiring several KBytes of data per GMM to be loaded from memory, enough for having several burst-mode read operations for maximizing memory throughput, and the computations are well suited for wide-SIMD architectures like GPUs. We exploit this regularity in our implementation.

When mapping algorithms to GPU-like massively parallel processors, it is important to clearly structure all operations. Modern parallel programming models like CUDA and OpenCL require the programmer to explicitly split their implementation into memory transfer and compute phases, with producer-consumer memory objects wrapped around execution programs or kernels. The main idea is for data to be first initialized and loaded into a memory space accessible by the GPU; the compute kernel executes a set of pre-defined operations on this data, writes it into GPU-accessible memory, followed by transfer of data to the hosts memory space for the caller to make use of the data. High processing rates can only be achieved when branching within each kernel is minimized, as heavy branching in the worst case can lead to serialization of operations. We take this into account, and regularize operations such that the core compute kernel has no branching through the use of parallel prefix sum (scan) followed by a compact operation.

A. Core GMM-compute routine (Brute-force)

At the core of most modern parallel processors, especially GPU-like processors, is a SIMD engine. The number of SIMD-lanes, or the granularity of every instruction processed every cycle, in NVIDIA GPUs is 32 (which is broken into 2 half-warps of 16 threads each). To support these compute units, on-chip shared memories and caches have an equal number of banks. One of the primary design decisions for our implementation was therefore to take into account the SIMD nature of computations, and the width of these SIMD units. Computations are best mapped as multiples of 16, for which our data layout packs every mixture into 80 values as described previously.

In the OpenCL programming model, a kernel is broken into smaller units referred to as work-groups, each of which consists of several work-items, with each work-item corresponding to an individual thread. Work-items within a work-group can co-operatively execute by sharing data through on-chip, local memory, while work-groups synchronize using global memory, when data sharing is required outside the work-group. GPUs obtain their efficiency by having thousands of threads in flight in order to hide long memory latencies due to the absence of on-chip caches. Every work-group consists of usually 100's of threads, and multiple work-groups are scheduled concurrently onto a single multiprocessor on the GPU, depending on available resources: local memory and registers. The multiprocessor is fully utilized (occupancy = 1) when enough work-groups can be scheduled on the processor so there is no idle time when the processor is waiting for data.

Since register and local memory directly correspond to occupancy, which is an important factor in obtaining peak compute throughput, we organized our core compute kernel accordingly to best fit into the given constraints. To have our kernel map well to 16-wide SIMD architectures, we organized the core GMM likelihood computation to work in multiples of 16. Since the number of bytes of storage required by an entire mixture is very high (80x4 Bytes), breaking the computation into batches of 16 (requiring 5 iterations) is well-suited from local memory usage perspective. Both the feature and knowledge source (Gaussian information) are loaded in multiples of 16. This helped increase our occupancy significantly. Since every memory load corresponds to a single thread, the number of mixtures assigned to a work-group can exceed the number of work-items, there-by requiring another iteration of memory loads. Finally, once data is loaded into local memory, iterations are made over the multiply-add operations that need to be performed for all the 16 values.

```
for all Gaussians in the work-group, in parallel
  for number of iterations in 0 to 4
    score = 0 //init
    xSh = load() //features into local memory
    for all mixtures in 0 to M-1
      kbSh = load() //part. GMM into local memory
      for dimensions in 0 to 13
        score += xSh * kbSh
      end
      if iteration == 4 // sentential, branch for
pre-computed constant and mixture weights
        score = kbSh - score  //constant term
        score *= f            //scaling factor
        score += W            //mixture weight
      else
        score += sSh * kbSh
      end
    end
  end
end
```

Fig. 5. Pseudo-code for 'gmmCompute(act-list)'

B. Feedback lists

Next, we incorporated the acoustic feedback list. This list is an array of flags with true or false values corresponding to active GMMs (and hence referred to as the active-list), indicating which GMMs get scored for the current frame. In order to regularize the compute, we use parallel reduction using prefix sum (scan), followed by a compaction operation based on the approach of Sengupta et al. [12]. This scan-compaction stage is important since it eliminates branching in core compute kernels and helps boost performance significantly. The use of active lists is currently the best known implementation of GMM computations on the GPU [16]. We therefore use this as our baseline for measuring both compute and memory overhead or savings, as the case may be. Since the scan is on very small arrays of a few thousand elements, the overhead incurred is almost negligible.

C. Frame-Layer: AML

AML-based processing is achieved by not operating on the 'in' active list directly, but using two additional arrays – ***buffer*** and ***new***. The 'buf' array acts as a cushion that keeps track of all active GMMs processed in prior frames of the current chunk, while the 'new' array corresponds to active data in the current frame that was not processed in prior frames of the current chunk. Upon generation of the 'new' list for a given frame, 'buf' is updated according to the pseudo-code shown. It can be seen that maintaining both 'buf' and 'new' arrays only requires logical operations and therefore does not consume any significant compute or memory resources. Hence, it is possible to implement dynamic chunk-based processing, that can lead to savings without incurring much overhead.

After the AML phase, the 'new' vector is compacted and GMM likelihood scoring is performed using the *gmmCompute()* routine shown previously, with a slight difference: Instead of processing one frame per invocation of the kernel, scores are computed for all frames ranging from the current frame to the end frame in the chunk. The only overhead at this stage is in terms of re-setting and maintaining the bit-vectors for every frame, which is very small when compared to loading entire GMMs every frame.

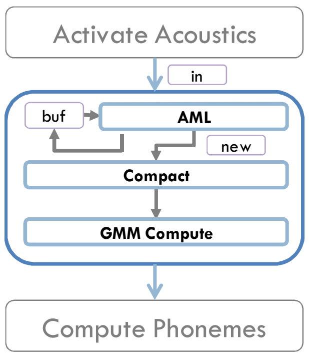

Fig. 6. Data-flow steps for AML

```
for frm in 0 to all frames in the utterance

  // AML processing
  chunkBoundary = frm % CHUNK_SIZE
  if (chunkBoundary == 0) // at the boundary
    reset new-list
    reset buf-list
  end

  new-list = in-list && !buf-list //populate
  buf-list = new-list || buf-list //update

  // Scan & Compaction
  act-list = scan_compact(new-list)

  // GMM Compute over remaining frames in current
     chunk
  score = gmmCompute(act-list)

end
```

Fig. 7. Pseudo-code for ‘aml(in-list)’

TABLE I
FRAME-LEVEL, ACOUSTIC MODEL LOOK-AHEAD RESULTS FOR VARYING CHUNK SIZES. CHUNK SIZE OF 1 FRAME IS THE BASELINE FOR PERCENTAGES OF COMPUTE AND MEMORY CONSUMED

Chunk	*WER*	*Comp. Ovrhead (%)*	*BW Saved (%)*	**RTF 9400 M (ION)**
1	6.86	0	0	1.50
2	6.86	3.46	43.76	2.70
4	6.86	9.76	67.46	3.27
8	**6.86**	**20.64**	**79.90**	**3.96**

Since in the worst case only more GMMs are being scored when they would otherwise not have been, it is important to note that there is no impact on the overall accuracy.

D. GMM Layer: CI-GMM

Our last level of optimization was to incorporate CI-GMM-based computations to the pipeline. The overall structure remains the same as that presented in V-C, but with a couple additional stages that deal with the first-pass on CI GMMs. After the initial round of scores is computed, the CI-GMM Process stage uses the best scores for the number of frames in the chunk to obtain a list of frames for which CI-GMMs pass the threshold when compared to the best-scoring GMMs. These values are summed for all frames, and a simple voting mechanism is used to determine whether the CD-GMMs corresponding to CI-GMMs need to be processed, or can be backed-off with CI-GMM scores. This yields two lists, one compute and one backoff list, processed by AML(c) and AML(b) blocks respectively.

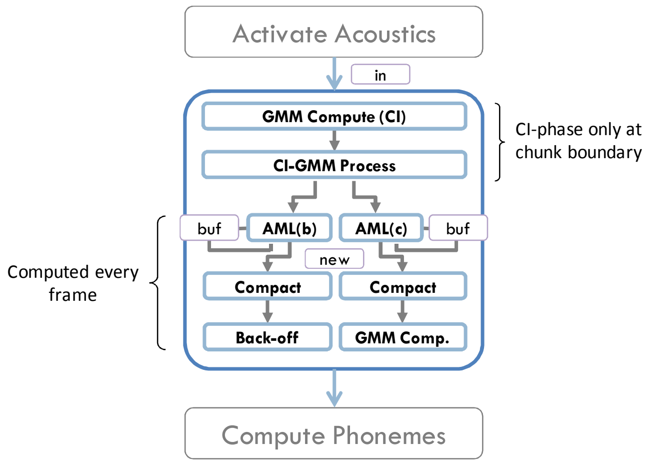

Fig. 8. Data-flow steps for CI-AML optimization

TABLE II
RESULTS OF USING THE TWO-PASS, CI- AML

Chunk	*CI-GMM Thresh*	*WER*	*Comp. Saved (%)*	*BW Saved (%)*	**RTF 9400 M (ION)**
4	1	7.27	24.04	79.47	4.32
4	2	7.72	36.81	82.95	4.85
4	3	8.67	48.81	86.21	5.40
8	1	7.23	11.78	86.05	4.95
8	2	7.31	23.57	87.75	5.37
8	**3**	**7.81**	**34.05**	**89.27**	**6.18**

Amongst all the additional stages added between the brute-force implementation and the CI-GMM optimized implementation, the overhead imposed by CI-GMM is the largest. It has to do with the fact that a data structure with mappings from CI-GMMs to corresponding CD-GMMs (represented by ‘cig2g-list’) is required to activate CD-GMMs in either the compute or backoff arrays.

```
for frm in 0 to all frames in the utterance

  chunkBoundary = frm % CHUNK_SIZE
  if (chunkBoundary == 0) // at the boundary
    reset new-lists {b & c}
    reset buf-lists {b & c}

    for CHUNK frames in parallel
      // Compute CI-GMM scores
      score = gmmCompute(all CI-GMMs)
      maxScore = max(score)

      // CI-GMM process
      tmpVec = score > (maxScore + CIbeam)
      finVec = sum of tmpVec over all frames in
               the chunk for every CI-GMM
    end

    if finVec > CiGThresh //include CI-GMM to CD-
                            GMM de-referencing
      cig2g-c-list = 1
    else
      cig2g-b-list = 1
    end
  end

  // AML processing
  aml(cig2g-c-list && in-list) //compute GMMscore
  aml(cig2g-b-list && in-list) //backoff by
                                 using corresp.
                                 CI-GMM scores

end
```

Fig. 9. Pseudo-code for 'cigmm_aml()'

VI. Discussion

From Tables I & II, the larger the chunk size, the greater is the speed-up achieved. This is perhaps not a very surprising result since as we discussed earlier, likelihood computation of GMM scores is a load-intensive operation, dominated by 100's of cycles of memory access latency compared to 10's of cycles of compute. If speed of processing was the only goal, then one would tend to pick larger chunk sizes that require fewer accesses to memory. For most practical systems, however, speed-up is not the only criterion. Other aspects such as compute & memory bandwidth saved, and the effect of any optimizations for reducing burden on the system resources to accuracy, also need to be considered.

Our results clearly show that there is no one correct answer on what configuration is best. Depending on the constraints, and the order of importance of various parameters, the solution would vary. We have compiled a set guidelines below:

- If compute resources were not a limitation, and power considerations not of paramount importance, using AML would suffice. Savings of 80% memory bandwidth with several times faster than real-time processing speed for no loss of accuracy can be achieved. Further, if, under certain usage scenarios maximum speed-up is not required, smaller chunks could be used to minimize strain on compute resources.
- If compute resources are at a premium, then the CI-GMM technique would need to be used. The first consideration is perhaps tolerance to loss in accuracy. Since scores are approximated when using the CI-GMM technique, the more the approximation by increasing the CI-GMM state threshold, the greater the loss in accuracy. Second, is the chunk size: the greater the chunk size, the lesser the approximation for smaller values of the CI-GMM state threshold. A chunk size of 8 frames gives better accuracy compared to chunk size of 4 for corresponding CI-GMM state thresholds.
- Platform-wise, since low-end processors like the NVIDIA 9400M are always likely to be resource bound, or have tighter operating constraints, the CI-GMM optimization can be quite helpful. Not only do fewer computations need to be performed, but the speed-up obtained can also be of importance as it can free system resources for other applications to use the device for the remaining time. A direct side-effect is loss in accuracy, ranging from 6% to 26.3% relative to the baseline, though in absolute terms for our 5k word dictionary the degradation is marginally worse.

As processing on the GPU becomes increasingly mainstream, with more applications running on the GPU for everyday tasks, a wide range of conditions will be encountered by the system. Therefore, rather than fixing any of the parameters discussed in this paper, the ideal system would dynamically adjust and switch between AML or CI-AML processing, for varying chunk sizes, and CI-GMM thresholds. For example, if the user was dictating a document, then a higher error rate might be tolerable when compared to a command and control task, where accuracy is critical. The system could also autotune these parameters dynamically when battery power goes below a certain threshold, or the ambient temperature of the device is higher than desired.

In comparison with other known implementations of GMM compute on the GPU, a majority of the approaches proposed to date have focused on harnessing the raw compute power of these processors by computing 'all' GMMs for every mixture for every frame [13-15]. The most optimized implementation that we are aware of is by Chong et al. [16], where they use the acoustic active list. Since this is the best known methodology implemented on the GPU, we chose AML for a chunk size of 1 frame, which essentially represents the traditional approach, as our baseline for all numbers presented in this paper. Just by using the acoustic feedback list, even a traditional approach of computing GMM scores for one frame per iteration requires 65.74% less compute than a brute-force approach. As we intend to run our implementation on low-end devices, a brute-force approach is not a feasible option.

VII. Conclusion & Future Work

At the GMM layer, we have validated the results presented in our theoretical analysis over a larger 5k-word dictionary, showing that our proposed modifications can significantly help in reducing compute and bandwidth requirements that are required for ASR running on low-end GPU processors. Looking forward, there are two additional approaches that could be incorporated on top of our implementation to yield even greater savings in memory and compute requirements.

Frist, as vocabulary size increases for more complex tasks, the number of mixtures per GMM will likely increase by a factor of two to four, with a lesser increase in the number of GMMs. The third level of modifications we proposed earlier [6] would be directly applicable for addressing this. Second, it has been shown in prior studies that half-precision floating point or custom fixed-point operations, if crafted carefully, can be used instead of single-precision floating-point operations of GMM likelihood computations. While custom fixed-precision requires several software instructions to implement every operation, with the support for a half-float data type in OpenCL (with hardware-level native support in the near future), half-float implementations might provide a good opportunity to further reduce memory bandwidth requirements by half.

Nonetheless, our current framework lays a strong foundation for further exploration into realizing the goal of using low-end GPU processors for performing speech recognition. To the best of our knowledge, ours is the only work to analyze and present a mechanism of reducing compute and memory bandwidth requirements on GPUs for speech recognition, with a special focus on low-end processors. Looking ahead, as resource-constrained mobile platforms become increasingly prevalent, we believe that researchers will need to broaden their focus from the single-minded goal of achieving maximum possible acceleration today to optimizing systems that include memory bandwidth and power consumption metrics as well. We believe that research efforts that combine these additional metrics will be more relevant in influencing the future direction of processor architecture, software programming models, or algorithm selection for use in practical, deployable products. The work presented in this paper is a step in that direction.

ACKNOWLEDGEMENTS

We thank an Intel Microprocessor Technology Lab grant for funding this project, and the BMW Technology Office for lending us the NVIDIA ION platform.

REFERENCES

[1] *NVIDIA CUDA Compute Unified Device Architecture: Programming Guide* (Version 4.0), Nvidia Corporation, 2011.

[2] A. Munshi, *The OpenCL Specification (Version 1.1, Revision 44),* Khronos Group, 2010.

[3] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," in *Computer Graphics Forum*, vol. 26, pp. 80-113, 2007.

[4] (2011) NVIDIA Tegra website. [Online]. Available: http://www.nvidia.com/page/handheld.html

[5] (2011) AMD Fusion details from Wikipedia. [Online]. Available: http://en.wikipedia.org/wiki/AMD_Fusion

[6] K. Gupta, and J. D. Owens, "Three-Layer Optimizations for Fast GMM Computations on GPU-like Parallel Processors," in *Proc. of Automatic Speech Recognition & Understanding Workshop*, pp. 146-151, Dec. 2009.

[7] K. Agaram, S. Keckler, D. Burger, "A Characterization of Speech Recognition on Modern Computer System," in *Proc. of the 4th IEEE Workshop on Workload Characterization*, pp. 45-53, Dec. 2001.

[8] R. Krishna, S. Mahlke, and T. Austin, "Insights Into the Memory Demands of Speech Recognition Algorithms," in *Proc. of the 2nd Annual Workshop on Memory Performance Issues*, May 2002.

[9] A. Chan, R. Mosur, A. Rudnicky, and J. Sherwani, "Four-layer Categorization Scheme of Fast GMM Computation Techniques in Large Vocabulary Continuous Speech Recognition Systems," in *Proc. of Interspeech*, pp. 689-692, Oct. 2004.

[10] (2011) Sphinx Homepage. [Online]. Available: http://cmusphinx.sourceforge.net/html/cmusphinx.php

[11] J. Garofalo, D. Graff, D. Paul, and D. Pallett, "CSR-I (WSJ0) Speech Corpora," Linguistic Data Consortium, LDC93S6A,

[12] S. Sengupta, M. Harris, Y. Zhang, J. D. Owens, "Scan Primitives for GPU computing," *Graphics Hardware 2007*, pp. 97-106, August 2007.

[13] P. Cardinal, P. Dumouchel, G. Boulianne, and M. Comeau, "GPU Accelerated Acoustics Likelihood Computations," pp. 350-353, in *Proc. of Interspeech*, Sept. 2008.

[14] P. R. Dixon, T. Oonishi, and S. Furui, "Harnessing Graphics Processors for the Fast Computation of Acoustic Likelihoods in Speech Recognition," *Computer Speech & Language*, vol. 23, pp. 510-526, Oct. 2009.

[15] P. R. Dixon, T. Oonishi, and S. Furui, "Fast Acoustic Computations Using Graphics Processors," in *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, pp. 4321-4324, Taipei, Taiwan, 2009.

[16] J. Chong, Y. Yi, A. Faria, N. Satish, and K. Keutzer, "Data-Parallel Large Vocabulary Continuous Speech Recognition on Graphics Processors," in *Proc. of Emerging Applications and Manycore Architecture*, pp. 23-35, Jun. 2008.