Title
Three-Layer Optimizations for Fast GMM Computations on GPU-like Parallel Processors

Authors
Gupta, Kshitij
Owens, John D.

Publication Date
2009

DOI
10.1109/ASRU.2009.5373410

Peer reviewed

Three-Layer Optimizations for Fast GMM Computations on GPU-like Parallel Processors

Kshitij Gupta, John D. Owens

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

***Abstract*—In this paper we focus on optimizing compute and memory-bandwidth-intensive GMM computations for low-end, small-form-factor devices running on GPU-like parallel processors. With special emphasis on tackling the memory bandwidth issue that is exacerbated by a lack of CPU-like caches providing temporal locality on GPU-like parallel processors, we propose modifications to three well-known GMM computation reduction techniques. We find considerable locality at the frame, CI-GMM, and mixture layers of GMM compute, and show how it can be extracted by following a chunk-based technique of processing multiple frames for every load of a GMM. On a 1,000-word, command-and-control, continuous-speech task, we are able to achieve compute and memory bandwidth savings of over 60% and 90% respectively, with some degradation in accuracy, when compared to existing GPU-based fast GMM computation techniques.**

I. Introduction

Several challenges have hindered ubiquitous deployment of Automatic Speech Recognition (ASR) as a means of HCI. Apart from difficulty in achieving high accuracy under real-world conditions, from an implementation stand-point, ASR workloads prove challenging on three fronts: compute, memory bandwidth, and irregular memory access patterns. With a shift in the compute paradigm towards commodity parallel processing architectures (IBM's Cell, modern graphics processors [GPU], Intel's Larrabee), these processors can be used for overcoming the abovementioned challenges, ultimately leading to the implementation of fast ASR systems.

For optimally utilizing these new processor architectures, either new techniques need to be invented, or traditional techniques that have worked well on single-core (scalar) CPU need to be revisited and altered to better suit the parallel processing model. Since a major portion of silicon area on modern parallel processors is reserved for compute resources, the compute bottleneck is relatively easier to solve when compared to memory issues (bandwidth and access patterns). In contrast to most speech papers that entirely focus on exploring techniques from a purely computation-reduction perspective, and GPU implementation papers to-date that have focused on obtaining the best possible speedup without analyzing memory bandwidth-related issues, we primarily focus on the memory bandwidth issue in this paper while leveraging well-established computation-reduction techniques that are known to simultaneously reduce the compute burden.

Among the various stages in HMM-based ASR recognizers, Acoustic Modeling is known to be a dominant part of the system, accounting for between 60-90% of all processing cycles in most systems. Each state within a continuous-density HMM is modeled by a mixture of multivariate Gaussians, also referred to as the Gaussian Mixture Model (GMM). LVCSR systems have thousands of GMMs, each with up to 128 mixtures, requiring several tens of GFLOPS of compute and several hundreds of MB of memory bandwidth per second.

Recently, there has been a shift towards adopting GPUs, which provide massive amounts of data and thread-level parallelism, for general-purpose (GPGPU) compute: implementing and accelerating non-graphics kernels and applications [1]. All research efforts for implementing ASR on GPUs to-date have focused on utilizing high-end desktop processors with the primary goal of maximally utilizing the underlying processor resources for obtaining the best possible speedup [2]-[4]. Given the several orders of magnitude difference between required versus available compute and memory resources offered by these high-end platforms, reasonable speedup could be obtained merely by focusing on wisely laying out data and applying rudimentary optimizations.

Our approach to ASR on GPU-like processors is from a different perspective. Since a majority of GPUs sold on the market today are low-end (mostly integrated graphics processors), and this trend is only likely to accelerate as adoption of mobile and handheld devices increases, we believe that for widespread adoption of ASR as a ubiquitous form of HCI, it is critical to optimize ASR algorithms to map efficiently onto these lower-end processors running on mobile/smaller form-factor devices. Targeting low-end GPU-like processors presents a very different set of constraints and challenges – fewer compute units are available, memories are smaller, memory bandwidth is significantly less, and power is a major concern. Suffice to say that optimizations at every level are required in order to achieve efficient implementation on such platforms.

In this paper we focus on the compute intensive part of ASR – GMM computations within Acoustic Modeling. We focus on three of the four layers presented by Chan et al. [5] for realizing fast GMM computations, and propose three modifications for making these algorithms map well onto GPU-like parallel programmable architectures, addressing both compute, and memory bandwidth issues without significantly impacting accuracy.

This is an implementation-agnostic paper where we focus on three primary parameters: accuracy, memory bandwidth consumed, and computation reduced. We show that a considerable amount of temporal locality exists at various

levels of GMM computations, which, when exploited, can help in achieving our stated goals, leading to a highly optimized implementation on low-end GPU-like processors. We focus on the modifications necessary and leave GPU-specific implementation details and optimizations to be presented in a future paper.

II. Motivation: Memory Efficiency is Critical

The goal of an ASR system is to pick the most likely sequence of words that have the highest probability of occurring for a given speech segment. While doing so, the system has to maintain a set of hypotheses for every frame, which if done in an unconstrained manner can grow the search space to unmanageable proportions. Keeping the search space to a manageable size is therefore an important aspect for any practical system. Managing the search space is critical at higher levels than it is at lower levels since every speech unit at a lower level is shared by multiple units in the next higher level.

The system can be constrained by keeping track of active speech units (or states) throughout the knowledge-base hierarchy, and computing scores for only the states that are active. Conceptually, this process can be divided into two phases as shown in Fig. 1(a). In the first phase, a set of *active lists* corresponding to *active states* are generated and propagated from higher-level models to corresponding speech units at the next lower levels. In the second phase, scores of these active states are computed and propagated to the next higher level until the highest level is reached. This process is repeated for every frame.

A typical ASR pipeline is shown in Fig. 1, from which it is evident that ASR systems have heavy data-dependency internally, making execution of each stage sequential in nature by requiring active state information from the previous stage. In order to simplify the system, the innermost computation (Compute Acoustics in Fig. 1(a)) can be de-coupled from the rest of the system, leading to a brute-force approach whereby all GMMs in the Acoustic Model are scored. This approach is used by Cardinal et al. [2], enabling them to map dense-linear GMM computations with high SIMD (Single Instruction Multiple Data) utilization on a GPU.

Since GMM data to be loaded from memory is fixed in a brute-force approach and is irrespective of active word hypotheses, frames could be batch-processed whereby multiple speech frames are computed for every load of the model as suggested by Mathew et al. [6], greatly helping address the memory bandwidth issue. However, such an approach comes at a huge overhead of computing inactive state scores that would otherwise be ignored by higher layers, doing little to enhance accuracy. Consequently, despite a 10x difference in CPU v/s GPU compute capability, the speed-up was limited to 5x for GMM computations [2]. Further, while a brute-force approach is possible on high-end processors, it is not practical on low-end processors with significantly less compute resources and power consumption constraints.

Incorporating active lists to guide which GMMs get scored every frame typically saves between 50–75% of computations over the brute-force approach and is therefore a necessary optimization. By using acoustic active lists, Chong et al. report a speed-up of 19x over their reference CPU implementation [3]. While incorporating active lists for GMM reduces the compute load, it introduces a point of serialization due to the dependency of next-frame computation on previous frames' active states, thereby constraining the memory sub-system since computations can no longer be batch-processed.

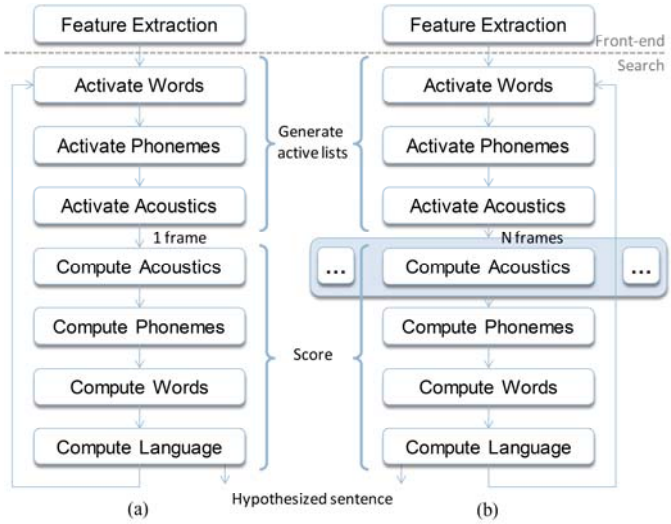

Fig. 1. A conceptual two-phase data-flow diagram of a typical ASR pipeline showing: (a) the traditional approach, and (b) the proposed chunk-based processing of GMM likelihood computations on the GPU.

Parallel architectures tend to have smaller caches compared to the large workloads they process. Modern GPU architectures do have several different kinds of caches, albeit extremely small in size and mostly special-purpose in nature (targeted towards graphics workloads). GPUs using NVIDIA's CUDA programming model feature a user-managed shared memory for sharing data across a SIMD lane for non-graphics applications [7]. These memories, however, are small (16 kB), and restrict data sharing to within a thread block on each SIMD core, not providing CPU cache-like temporal locality. Due to this nature of the architecture, utilizing active lists introduces a level of non-determinism that requires accessing active GMMs from main memory every frame, exacerbating the memory bandwidth requirement.

Most importantly, as processors evolve towards a unified memory architecture model similar to today's integrated GPUs or AMD Fusion-like processors (CPU & GPU on the same chip die, sharing the same address space and main memory), inefficient accesses to main memory can lead to severe performance bottlenecks. Reducing and optimizing all memory accesses is therefore an important aspect for maximizing performance on these architectures, especially for UI workloads that have the potential to be always-on.

From all these observations we were motivated to draw the following conclusion: make use of active lists for reducing the amount of compute while finding ways of reducing memory bandwidth requirements for GPU-like parallel architectures by exploring and subsequently exploiting temporal locality at

various levels within GMM computations, modifying the ASR pipeline to the one shown in Fig. 1(b). In later sections we show that considerable locality exists, and that it is possible to batch-process several frames every time a model is loaded from main memory without de-coupling the feedback mechanism, thereby helping address all goals identified above simultaneously.

III. Design Attributes for Parallel-Friendly ASR Algorithms

We compiled a set of design attributes that would be critical in achieving a highly optimized, parallel processor-friendly, fast GMM implementation on low-end GPU-like parallel processor architectures. Since well-established computation reduction techniques exist, we compare and contrast traditional approaches on scalar CPUs with those of attributes suitable for many-core, wide-SIMD equipped parallel architectures, and use these attributes as the driving principle for our subsequent exploration.

A. Dynamism

- **CPU:** Scalar CPU processors are targeted for a wide range of applications with a core focus on making the common case(s) faster for irregular workloads, utilizing typically less than 1% of the silicon area for compute purposes. Optimizing every instruction to minimize the number of compute operations at all costs is therefore critical for achieving speedup in most cases, even if this requires introducing significant irregularities in compute or memory accesses (as is the case with most GMM compute reduction techniques).
- **GPU:** GPU-like parallel processors are based on wide-vector SIMD units which are well suited for throughput applications that have considerable data-level and thread-level parallelism. Algorithms that can maximally benefit from these architectures have regular, well-defined compute and memory access patterns. An important design goal is therefore to regularize algorithms as much as possible, even if such a modification comes at the cost of system resource utilization overhead.

B. Caches

- **CPU:** Since many applications have considerable temporal locality, the goal of CPU-like scalar processors is to reduce memory access latency by storing reusable data in various levels of caches. Access to memory staggered over time, like in the case of processing several frames of speech, does not incur a high memory access cost.
- **GPU:** As discussed in Section II, GPUs have small caches, and the per-block shared memory available for compute workloads does not provide temporal locality, so GPUs require significantly more accesses to main memory. Taking the CPU approach to optimization does not conserve memory bandwidth, so optimizing for cache-less architectures is therefore an important aspect for future GPU-like processors.

C. Computational Complexity

- **CPU:** Computational complexity, or the number of operations required for every load from memory, is not as important a point for cache-based CPU architectures, where data access cost is minimal if the workload can achieve high cache-hit rates.
- **GPU:** Workloads with a low compute-to-memory ratio make poor use of the GPU's computational resources. For example, GMM log-likelihood evaluations require only two arithmetic operations per 4-byte word loaded from memory, making these computations highly load-intensive. Since data is invariably loaded from main memory, it is vital to amortize the cost of every load by processing multiple speech frames, indirectly increasing the computational complexity.

D. Branch Support

- **CPU:** A significant amount of silicon area on CPUs is devoted to providing support for branch prediction and handling. Introducing irregularities for optimizing algorithms to the lowest level possible for achieving maximum computation reduction on such architectures does not have an adverse impact.
- **GPU:** The modern GPU, at its core, is a SIMD processor in which all threads are run in lockstep. The more divergent the branches in SIMD execution, the larger the runtime penalty to process such data. In the worst case, all SIMD lanes diverge differently, leading to serialization equal to the width of the SIMD lane. It is therefore important to reduce branches in the application as much as possible, to reduce the number of times branch-able code is executed, and if branches are necessary, to attempt to minimize the divergence in those branches.

E. Compute-Memory Tradeoff

- **CPU:** CPUs tend to be compute-bound since a large portion of the processor is devoted to various levels of on-chip memories. The primary goal therefore is to minimize compute operations to the fullest extent possible.
- **GPU:** Conversely, GPUs have abundant compute resources but limited on-chip memory. The goal therefore should be to trade memory accesses for extra compute if that can prove to be more efficient. In the long-term, since computation costs decrease with every new generation of hardware while communication costs remain roughly constant, compute should be preferred over communicate.

We hand-picked three techniques that, based on the nature of GMM scoring in ASR, are known to provide the best possible reduction in computation. By exploiting temporal locality at various stages and modifying these techniques to fit within the attributes discussed in this section, we are able to achieve significant compute and memory bandwidth savings.

IV. Observations

In the following two sections we present our observations and suggested modifications to three well-established GMM computation reduction techniques at the *frame*, *GMM* and *mixture* layers in order to make them parallel-friendly. All our experiments were performed on CMU's open-source Sphinx 3 recognizer [8] running the 1,000-word Resource Management corpora task comprising of 1600 utterances in the continuous speech, speaker-independent test-set [9]. Our knowledge-base was derived from open-source Sphinx models consisting of a trigram language model, continuous-density HMMs, with tied states modeled by a set of 1935 GMMs having 8 mixtures each [10].

Sphinx uses several layers of optimizations for fast GMM compute primarily based on the work presented by Chan et al. [5], [11]. We bypass all these optimizations for each of the experiments described in the following sections, effectively using our own GMM compute routines, performing un-approximated full-precision scoring and incorporating approximations from only the optimization layer being discussed.

A. Frame-Layer

Frame-layer optimizations are the highest level of fast GMM compute optimizations. Since for certain parts of speech the input signal can vary slowly, two coarse-grain approximation techniques which entirely depend on the input data can be used. One approach is based on naively down-sampling the frequency at which GMM scores are evaluated, computing scores only every *N*th frame. As *N* increases, overhead in the search module and error rate increase rapidly without a significant reduction in computations, limiting the usefulness of this approach. The other, more accurate approach is to find successive frames over which the input signal has not changed significantly, and use GMM likelihood scores from the first frame for all similar successive frames. Chan et al. report only modest compute reduction savings from this approach [5].

Our approach to frame-layer compute reduction is somewhat different. As motivated in Section II, since active lists are an important aspect for implementing a highly optimized design, and as noted in Section III, with the need for regularizing compute and memory access patterns on GPU-like parallel processors, we focus on addressing these aspects at the frame layer.

One of the biggest hurdles we found with incorporating the GMM active list was a large variation in the number of active GMMs over any speech segment, varying from very few to almost the entire set. Although it seems logical to conclude that there is little locality over time, we wanted to analyze the true nature of the locality and ran a few tests focusing on the *lifetime* of active GMMs, measured by the number of successive frames for which a they are active. We ran this experiment for three beam-setting combinations – relaxed, medium, and very tight – directly impacting accuracy rates. The Word Error Rate (WER) and active GMM lifetimes averaged over the entire test-set for each of the three beam settings are shown in Table I.

TABLE I
Average Lifetime of Active GMMs Over the RM1 Test-Set

Beam	WER	Avg. Lifetime (# frames)
Relaxed	2.68	13.95
Medium	4.21	12.50
Tight	8.03	11.32

Surprisingly, despite the number of active states varying over time, a state once activated remains active for on average 11–14 frames for each of the three beam settings. This shows that significant temporal locality exists regardless of beam pruning constraints, and if exploited, could lead to substantial savings in memory bandwidth. In order to obtain the best possible accuracy, we use *relaxed* beam settings for all experiments discussed in this paper.

B. GMM-Layer

GMM-layer optimizations are the next lower level of fast GMM optimizations that are possible. This approach is built on the nature of ASR knowledge-bases, which relies heavily on context-dependent models for handling co-articulation effects in continuous speech. Since context-dependent (CD) models are refined versions of context-independent (CI) models, a two-pass approach can be employed whereby CD states are scored only if their corresponding CI state score lies within a certain threshold, and approximated with CI scores if they fall beyond this threshold. It has been shown that significant savings for a small decrease in accuracy is possible by following this approach [5]. Similar to the previous analysis, Table II shows the WER and lifetimes of CI GMM that lie within beams averaged over the entire test-set.

TABLE II
Average Lifetime of Active CI-GMMs Over the RM1 Test-Set

-ci_pbeam	WER	Avg. Lifetime (# frames)
1e-7	2.95	5.38
1e-5	3.09	3.75
1e-3	3.29	2.60

Unlike the analysis in Table I, CI-GMM lifetimes are much smaller due to the narrower pruning threshold applied. This might lead one to conclude that temporal locality is too small to be practically beneficial. However, in the following section we show that this information can still be useful and can aid in a parallel-friendly approach with significant savings. For the remainder of CI-GMM experiments, we use the medium ci_pbeam (=1e-5) setting.

C. Mixture-Layer

The next layer of fast GMM optimizations are at the within-GMM, mixture level. In order to more accurately model data within continuous-density HMMs, weighted mixtures of multivariate Gaussians are used. Since any input can be close to only a small subset of mixtures, the GMM score could be approximated by computing only mixtures that are close to the input, while ignoring the others. A two-pass

coarse-grain/fine-grain approach is followed: the first-pass on Vector Quantized (VQ) or Sub-Vector Quantized (SVQ) version of the original GMM data results in the generation of a list of high-scoring mixtures, and the second pass is used for computing likelihood scores of these mixtures from the original GMM model. In our experiments we observed that the topmost mixtures in a frame remain amongst the top few mixtures over successive frames, showing that some locality over frames exists.

V. Results

From these observations we conclude that considerable locality exists at all layers of GMM compute. We propose three modifications to the traditional CPU approach, one for each of the three layers discussed in the previous section, to make them better suited for GPU-like parallel architectures by satisfying all attributes of parallel-friendly algorithms identified in Section III. All results presented in this section focus on three aspects: accuracy (WER) and the percentage of compute overhead and bandwidth saved for our proposed modifications w.r.t. the baseline setup.

We chose our baseline as the approach followed in the most optimized GMM compute implementation to-date on the GPU [3], utilizing the GMM active list of Fig. 1(a) to decide which GMMs get scored every frame. Since this implementation is not known to make use of any GPU-specific on-chip memories (or caches) for exposing temporal locality, data for every active GMM must be loaded from main memory every frame. The number of active GMMs can therefore directly be used for obtaining both the number of compute operations and the memory bandwidth required, which we use as our baseline for all savings presented in this section.

A. Modification # 1: Frame-Layer

From the discussion presented in Sections II & III, we concluded that the only approach for reducing memory bandwidth is to process frames in groups or *chunks*, computing Gaussian likelihood scores over multiple frames for every load of the GMM model while simultaneously supporting the acoustic active list (effectively moving the implementation from Fig. 1(a) to Fig. 1(b)). From our frame-level analysis of active state lifetimes, since states are active for more than 10 frames on average, we decided to use a "blind look-ahead" approach whereby once a state is active, it is deemed active for the rest of the chunk, irrespective of when in the chunk it ends-up being deactivated. This approach allows GMM models to be loaded only once per chunk, and likelihood scores are computed for all frames from the first frame in the chunk the state becomes active to the last frame in the chunk.

Since in the worst case we only compute scores for states that could otherwise have been de-activated earlier than at the chunk boundary, the only side effect of this approach is an increase in compute overhead, while providing significant bandwidth savings. Importantly, since scores of states de-activated by the search module are ignored anyway by the higher levels, it does not affect the accuracy. We refer to this technique as Acoustic Modelling Look-ahead (AML).

TABLE III
Frame-Layer Look-ahead Results. Compute Overhead represents extra computations needed due to the look-ahead, while Bandwidth Saved represents fewer memory accesses compared to the baseline (chunk=1).

Chunk	WER	Compute Ovrd (%)	BW Saved (%)
1	2.68	0	0
2	2.68	3.40	46.42
4	2.68	9.69	70.04
8	**2.68**	**20.55**	**82.07**
16	2.68	38.06	89.31

Since SIMD units within GPU-like parallel processors have an even number of lanes, we only focus on even multiples of chunk sizes. Chunk=1 effectively implies no look-ahead, and is functionally equivalent to the approach by Chong et al. [3]. We use this as our baseline for comparing all compute and bandwidth savings. Results in Table III show that a significant percentage of bandwidth can be saved as chunk sizes increase at the cost of a more modest increase in compute, without affecting accuracy. The promising results led us to incorporate AML for all our subsequent experiments.

B. Modification # 2: GMM-Layer

While the AML technique helps save considerable bandwidth, it requires significant compute overhead for larger chunk sizes. We incorporate GMM-layer optimizations to address this. In Section IV-B we noted that lifetimes of CI states are much smaller than those of active states. Following a frame-by-frame approach of which frames in a chunk should get scored and which frames should be backed-off depending on CI activity again leads to indeterminism, making it difficult to utilize the AML approach.

We propose a simple solution to solve this indeterminism. Instead of considering individual frames, we consider frame chunks as a whole in order to decide whether active CD states get scored. If the sum of frames for which the CI state passes the ci_pbeam threshold is greater than our threshold (CI State Threshold), CD states are scored for all frames in the chunk; otherwise all scores in the chunk are backed-off to CI scores. Since the lifetime of CI states was 3.75, we consider the threshold of both 3 and 4 frames.

TABLE IV
CI GMM-Layer Look-ahead Results. The CI-GMM Optimization helps reduce the required compute workload and memory bandwidth simultaneously.

Chunk	CI State Threshold	WER	Compute Saved(%)	BW Saved(%)
1	1	3.09	46.16	46.16
4	3	3.08	60.66	82.27
4	**4**	**3.03**	**67.97**	**90.18**
8	3	3.03	47.59	90.26
8	4	2.97	54.92	91.89

From Table IV it can be seen that by using the CI-GMM approach considerable savings in compute can be obtained. As scores are backed-off, the accuracy degrades slightly due to the approximations introduced, while simultaneously reducing the bandwidth and compute burden.

C. Modification # 3: Mixture-Layer

Although the CI-GMM approach reduces both the compute and bandwidth requirements significantly, we still explore the mixture-level optimization since in cases of GMM with a larger number of mixtures, this technique also contributes significantly to savings by requiring the computation of only the top mixtures. Instead of computing SVQ codeword scores and selecting top mixtures every frame, we compute them only at the start of a chunk, selecting the top N scoring mixtures. Whenever a state becomes active within a chunk, only the mixtures selected at the start of the chunk are computed. While this approach makes a bigger assumption of the locality and presence of top-scoring mixtures at chunk beginnings, our results in Table V show reasonable accuracy can still be achieved by selecting 3 or 4 of 8 mixtures in our GMM model. The overhead is not significant considering that the traditional beam-selection approach in Sphinx uses approximately 2.5 mixtures on average.

TABLE V
MIXTURE-LAYER LOOK-AHEAD RESULTS

Chunk	Top Mixtures	WER	Compute Saved(%)	BW Saved(%)
4	3	3.57	36.61	85.53
4	**4**	**2.95**	**23.56**	**81.96**
8	3	5.48	39.76	91.50
8	4	3.92	25.50	89.41

D. Piecing it all together: Frame + GMM + Mixture-Layers

Finally, since each of the three techniques analysed in this paper provide compute reductions that are orthogonal to each other, we conclude this section by presenting results obtained on combing these techniques. Due to the approximations introduced by mixture-level optimizations, the results are not as promising as those obtained by using just CI-GMM (Section V-B). Notably, although a comparable compute and memory bandwidth savings rate is achieved, it comes at the cost of degradation in accuracy. We believe that the combined savings will be better for GMMs with more mixtures.

TABLE VI
RESULTS FROM COMBINING ALL THREE LAYERS OF OPTIMIZATIONS

Chunk	CI State Threshold	Top Mix.	WER	Compute Saved(%)	BW Sv(%)
4	4	3	4.00	69.11	93.94
4	**4**	**4**	**3.29**	**65.06**	**92.69**
8	4	3	6.21	72.77	95.58
8	4	4	4.40	67.09	94.56

VI. CONCLUSION & FUTURE WORK

In this paper we focused on GMM computations in Acoustic Modeling with a special emphasis on reducing the memory bandwidth requirement in low-end, GPU-like parallel processors targeted towards mobile and handheld devices. We proposed three modifications to popular fast GMM techniques, making them parallel-friendly, which enabled us to reduce the compute and memory bandwidth requirement by over 60% and 90% respectively for a relative accuracy degradation ranging between 13.05% (Table IV) to 22.76% (Table VI) over our baseline system. The results show that there is no one solution, with the correct choice depending on several factors. We shall present the details of our GPU implementation along with an in-depth analysis of trade-offs for techniques presented here in a future paper.

ACKNOWLEDGEMENTS

Thanks to Intel Corporation for primary financial support for this project, with additional support appreciated from the National Science Foundation (Award 0541448) and the SciDAC Institute for Ultrascale Visualization.

REFERENCES

[1] 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.

[2] P. Cardinal, P. Dumouchel, G. Boulianne, and M. Comeau, "GPU Accelerated Acoustics Likelihood Computations," in *Proc. of Interspeech*, pp. 964-967, Sept. 2008.

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

[4] 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.

[5] 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.

[6] B. Mathew, A. Davis , and Z. Fang, "A Low-power Accelerator for the SPHINX 3 Speech Recognition System," in *Proc. Of Conference on Compilers, Architecture and Synthesis for Embedded Systems*, Oct. 2003.

[7] *NVIDIA CUDA Compute Unified Device Architecture: Programming Guide* (Version 2.2), NVIDIA Corporation, 2009.

[8] (2009) Sphinx Homepage. [Online].. Available: http://cmusphinx.sourceforge.net/html/cmusphinx.php

[9] D. Pallett, "A Look at NIST's Benchmark ASR Tests: Past, Present, and Future," in *Proc of IEEE Workshop on Automatic Speech Recognition and Understanding*, 2003.

[10] (2009) Sphinx Open Source Models website. [Online]. Available: http://www.speech.cs.cmu.edu/sphinx/models/

[11] A. Chan, R. Mosur, and A. Rudnicky, "On Improvements to CI-based GMM Selection," in *Proc. of Eurospeech*, pp. 565–568. Sept. 2005.