Optimized Acoustic Likelihoods Computation for NVIDIA and ATI/AMD Graphics Processors

Jan Vaněk, Jan Trmal, Josef V. Psutka, and Josef Psutka

University of West Bohemia, Department of Cybernetics, Univerzitni 22, 306 14 Plzen, Czech Republic

vanekyj, jtrmal, psutka_j, psutka@kky.zcu.cz

Abstract—**In this paper, we describe an optimized version of a Gaussian-mixture-based acoustic model likelihood evaluation algorithm for graphical processing units (GPUs). The evaluation of these likelihoods is one of the most computationally intensive parts of automatic speech recognizers, but it can be parallelized and offloaded to GPU devices. Our approach offers a significant speed-up over the recently published approaches, because it utilizes the GPU architecture in a more effective manner. All the recent implementations have been intended only for NVIDIA graphics processors, programmed either in CUDA or OpenCL GPU programming frameworks. We present results for both CUDA and OpenCL. Further, we have developed an OpenCL implementation optimized for ATI/AMD GPUs. Results suggest that even very large acoustic models can be used in real-time speech recognition engines on computers equipped with a low-end GPU or laptops. In addition, the completely asynchronous GPU management provides additional CPU resources for the decoder part of the LVCSR. The optimized implementation enables us to apply fusion techniques together with evaluating many (10 or even more) speaker-specific acoustic models. We apply this technique to a real-time parliamentary speech recognition system where the speaker changes frequently.**

Index Terms—**Automatic speech recognition, parallel algorithms, parallel architectures, software performance.**

I. INTRODUCTION

Large vocabulary continuous speech recognition (LVCSR) is a highly computationally intensive task, with the acoustic model likelihoods computation accounting for the largest portion of the processing task. In a few recent studies [1], [2], and [3], a GPU was employed as a coprocessor to compute these likelihoods. A speed-up of 4× to 6× was achieved for the full acoustic model evaluation itself. This led to a speed-up of 1.3× to 3× for the LVCSR system, depending on the individual task and setup, while maintaining an untouched accuracy. For real-time applications, CPU-only recognizer implementations often have to employ simplified (i.e., small) models, (acoustic model) pruning, computational approximations, or combinations of these techniques to fit into the real-time constraints. Thus, by using hybrid CPU-GPU implementations, an increase in accuracy can be achieved because even low-end GPUs are powerful enough to evaluate significantly larger models in real-time, without the need for any additional acceleration or complexity reduction techniques.

The goal of this work was to develop a fast GPU implementation that fully exploits the computing power of the GPUs of both main manufacturers: NVIDIA and ATI/AMD. This opens the possibility of using real-time LVCSR systems and other speech applications not only on laptops but also on near-future hand-held devices [4]. Further, the GPU can be shared between additional recognizers or other applications. The proposed approach could also be used during the acoustic model training phase. As expected, this reduces the training time significantly, especially for frame-discriminative methods [5].

The parallelization of an entire LVCSR system, together with its implementation on GPUs, was examined by Chong *et al.* [6], [7]. The acoustic model evaluation phase represents one half of the total computing time. With a significant speed-up of this part, the freed resources can be reassigned to other parts of the LVCSR engine: a more precise decoder, larger dictionary, or more complex language model. Alternatively, the same system can be run on a slow computer.

In our department, we focus on real-time speech applications (e.g., generation of automatic captions, dialog systems). The well-utilized GPU power enables us to simultaneously evaluate a set of speaker-specific (or speaker-cluster-specific) acoustic models in real-time and then combine the individual models' likelihoods using a fast fusion method [8]. In addition, a combination of full speaker-cluster-specific models with speaker-specific feature-transformation matrices is used to prepare several models, thus enabling the recognizer to handle large inter- and intra-speaker variability. Only a well-optimized GPU implementation has adequate capacity to deal with such a large number of models in real-time.

The outline of this article is as follows. In section II, a general introduction into GPGPU domain is given, together with some architecture details about NVIDIA and ATI cards. In section III, a thorough review of the current state of research in the discussed task (i.e. computation of acoustic likelihoods using GMM), in section IV and V, the proposed implementations for the NVIDIA architecture and ATI architecture are introduced and discussed in detail. In section VI, the results of the proposed methods are published and comparison to the existing methods from section III is made. Section VII concludes this article and suggests some possible real-world scenarios enabled by the research presented in this article.

II. General Purpose Computing on GPU

Traditionally, GPUs were developed for performing the primary task of graphics processing. With the increasing demand for high quality 3D graphics processing, the GPU's performance and versatility started to grow rapidly. The hunger for rich graphical experiences led GPU vendors and graphical chips designers to implement general programmability in several stages (e.g., vertex and fragment/pixel shaders) of the 3D processing pipeline. Historically, the vertex shader unit has been used for manipulation using the vertices of objects in 3D scenes, while fragment/pixel shader programs have been used to manipulate the color of the rendered fragment or specific pixel. Both these tasks are easily parallelized. Therefore, instead of the high-speed serial processing commonly employed in the domain of CPUs, GPUs were developed to support massively parallel processing.

The increase in raw computational power triggered an interest in whether GPUs could be used for more general computing tasks than just 3D graphics and high fidelity graphics effects.

The tasks suitable for processing on a GPU are characterized by high parallelism, low dependency between individual work elements, and a rather numerical character with minimal branching. Such tasks are commonly known as *data-parallel* algorithms.

Because they deal with the same task, the architectures of the GPUs from both the vendors are remarkably similar. Their characteristics are as follows:

- Support a *very large number of threads* with minimal thread switching overhead;
- *High speed main memory* (approx. 100 GB/s, CPU with about 8 GB/s): However, to achieve this high speed, the memory accesses must follow specific patterns and the accesses suffer from high latency;
- *Low caching capability*: The main memory is not cached at all, but other specific kinds of memories (constant memory and texture memory) are cached. However, the control of these caching capabilities from the viewpoint of programmer/compiler is limited, if not non-existent, and the data elements must be accessed in a predefined manner to achieve the maximum cache-hit ratio;
- *Limited Host (CPU)-to-Device (GPU) communication support:* The communication is usually established through DMA transfers; The DMA transfer request must be first configured and then submitted to DMA controller. This brings an additional latency in communication (hundreds of CPU cycles at minimum [9]). Since the GPU is usually connected via PCI-express bus, the theoretical bandwidth is about 8 GB/s and the achievable bandwidth is about 4-6 GB/s. For small data transfers, the achievable bandwidth is significantly smaller; for example for 1024 byte blocks, the achievable bandwidth is about 100 MB/s. Therefore, the computational task should limit the host-device communication to minimum.
- *Limited thread synchronization capabilities;* global synchronization across all the running threads is problematical (because of the architectural assumption of low dependence across work elements). All the running threads are usually synchronized only at the end of the computational task, so the usual approach is to decompose the computation into two or several computational tasks and each task is executed individually.
- *Limited branching capabilities* as a result of the SIMD (single instruction multiple data) character of the processing units. In general, branching is supported, but it has a significant influence on the computational throughput.

Because GPU architectures have such distinct characteristics, common CPU programming models (and programming languages based on these programming models) are not suitable for this task. In order to achieve a close-to-peak performance, the programmer must consider many low-level specifics of the given target architecture and, therefore, the programming model as well as the programming language must support an explicit expression of the programmer's intentions. During the last few years, several programming concepts have been proposed and slowly abandoned.

NVIDIA's CUDA (Compute Unified Device Architecture) has gained a wide acceptance. However, the CUDA standard is proprietary. Thus, intellectual property concerns led to the development of the open standard OpenCL (Open Computing language). The OpenCL standard was developed in cooperation with teams from ATI/AMD, IBM, Intel, NVIDIA, and others. Both these frameworks operate with similar concepts, usually named in the same way, and migration between them is quite straightforward. The OpenCL standard [10] operates with the following models.

- *The platform model* postulates a *host* connected to one or many OpenCL *computing devices*. Every OpenCL device is divided into one or more *processing elements*.
- *The execution model* postulates that execution of an OpenCL program occurs in two parts: a host program, which operates on the host, and one or more *kernels*, which run on one or more OpenCL devices. Each kernel, when dispatched to run, has a notion of task geometry and is executed within a so-called index space. This means that each instance of a kernel is identified by coordinates in this index space, which provides a global ID (gID) to each kernel instance. The specific kernel instance identified by this gID is called a *work-item*. Work-items are organized into *work-groups*. The work-groups provide a more coarse-grained decomposition of the index space. A work-group is assigned a unique work-group ID with the same dimensionality as the index space used for the work-items. The work-items in a given work-group execute concurrently on the processing elements of a single compute unit.
- *The memory model* specifies that the running kernel has access to four distinct types of memory*:*
  - *Global memory—a* memory region that grants access to read/write (R/W) operations by any work-item within any work-group. The

memory address space is common to all work-items. Depending on the device capabilities, this memory may be cached. In the scope of GPU, the global memory maps to the off-chip main memory (see above).

- *Constant memory*—a segment of memory, whose content was initialized by the OpenCL host and which remains constant during the execution of the kernel. In the context of GPUs, the constant memory is usually on-chip and cached automatically, so that the access latency for consecutive accesses is reduced.
- *Local memory*—a segment of memory local to a work-group. In the context of GPUs, this is an on-chip memory with low access latency. The amount if this memory is significantly limited (thousands of bytes)
- *Private memory*—a segment of memory local to the individual work-item. The private memory can be thought of as a set of registers. The access to this memory is fast and virtually without any additional latency, however the memory volume is very limited (hundreds of bytes).

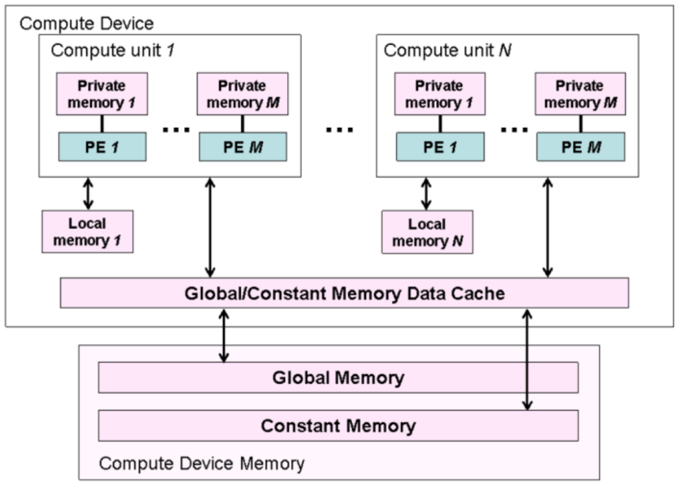

Fig. 1 Memory organization for a conceptual OpenCL device, from [10].

Since the publication of OpenCL v1.0 (June 2008), many HW vendors and SW producers have announced support of OpenCL in their products. Thus, there is a significant chance that OpenCL will become a standard for heterogeneous computing. Unfortunately, this wide acceptance does not dislodge the burden of hand-tuning the computational kernels for individual distinct HW architectures. All the major OpenCL platform producers have released specific "*Best Practices Guides*" [11], [12], which discuss the specific considerations that should be accounted for when programming for a given platform.

The CUDA standard is historically older and its application is limited to NVIDIA devices. However, it postulates the same models as the OpenCL. The platform and memory model are basically the same, while the execution model differs only in terminology. Because of this compatibility, we will use the CUDA terminology mostly even when referring to ATI/AMD implementation. The main terminology differences are following: *Work-item* in OpenCL corresponds with *thread* in CUDA. *Work-group* corresponds with *thread-block* (or *block*), and finally, *Local* memory is called *shared* memory in CUDA.

A. NVIDIA GPU architecture

The NVIDIA GPU consists of many *processing elements* (PEs) called multiprocessors. Older NVIDIA GPUs have 8 stream processors in each PE, together with a 16-kB on-chip local memory (called *shared memory*). A single PE offers 8 k or 16 k 32-bit registers, depending on the GPU series. The Fermi-based PEs are larger and include 32 or 48 stream processors. The 64-kB on-chip memory can be set to two configurations: 16 kB L1 cache and 48 kB local memory or vice versa.

The peak memory bandwidth from global memory can be achieved only via *coalesced* access, where 16 consecutive work-items (half-warp) read/write consecutive addresses (note that full-warp is needed on Fermi-based cards). Another example involves using local memory, where one should omit bank-conflicts, which can significantly degrade the kernel performance. These issues are discussed in more detail in [13] and [14]. The second-mentioned is more focused on kernel optimizations per-se.

B. ATI/AMD GPU architecture

The ATI/AMD GPU also consists of multiple PEs. However, compared to NVIDIA PEs, the internal architecture is different. Each PE contains 16 stream cores, each equipped with five stream processors (four in Cayman based 69XX GPUs). This is why ATI/AMD GPUs have a higher raw computational performance than comparable NVIDIA GPUs. In real life, however, it is difficult to supply the input data sufficiently fast to keep these high-performance multiprocessors fully utilized. Therefore, the maximal performance cannot be achieved in some tasks or badly optimized implementations. The older ATI cards (HD 4000 series and older) do not contain on-chip local memory but use the slower global memory instead. Therefore, a programmer should implement the algorithm in a fashion that does not use the local memory on these cards. The HD 5000 and 6000 series have 32 kB of local memory, but it is not advisable to directly use it extensively because the limited throughput is insufficient to gain the maximum computational performance. Instead of the local memory, a relatively large 256-kB register file is preferred. A more detailed performance optimization guide can be found in [11].

III. Recent Implementations Review

The first use of a GPU for acoustic model likelihoods computation was briefly mentioned by Dixon *et al.* [15]. The following year, a more detailed paper was published by Cardinal *et al.* [1]. Both approaches share a common ground: the computation of likelihoods as dot products. The entire acoustic model is represented as a matrix $\boldsymbol{A}$, in which each row is a log-weighted Gaussian component with a diagonal covariance matrix. The i^{th} component of a J-dimensional mixture model is represented as a vector according to

$$\left\{K, \frac{\mu_{i1}}{\sigma_{i1}^2}, \cdots, \frac{\mu_{iJ}}{\sigma_{iJ}^2}, -\frac{1}{2\sigma_{i1}^2}, \cdots, -\frac{1}{2\sigma_{iJ}^2}\right\}, \tag{1}$$

where K is

$$\log w_i - \frac{J}{2}\log 2\pi - \frac{1}{2}\sum_j \log \sigma_{ij}^2 - \frac{1}{2}\sum_j \frac{\mu_{ij}^2}{\sigma_{ij}^2}. \tag{2}$$

The feature vector is then expanded to

$$\boldsymbol{x}^T = \{1, x_1, \ldots, x_J, x_1^2, \ldots, x_J^2\}. \tag{3}$$

The score of every Gaussian can be found as the matrix-vector multiplication $\boldsymbol{y} = \boldsymbol{Ax}$. The output vector **y** is a vector of the log-weighted scores of every Gaussian component of the mixture (given input vector $\boldsymbol{x}$). An evaluation of the following n feature vectors can be performed as a single matrix-matrix multiplication, $\boldsymbol{Y} = \boldsymbol{AX}$, where each column of $\boldsymbol{Y}$ contains the log-weighted scores for every Gaussian component of every mixture model for the corresponding feature vector in $\boldsymbol{X}$, which is a matrix of n feature vectors

$$\boldsymbol{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n\}. \tag{4}$$

The final mixture likelihoods are obtained by the sum of all the belonging Gaussians. This sum is implemented in the log-domain as a logarithmic addition, which is defined as $\ln(e^x+e^y)$.

TABLE I
CARDINAL'S IMPLEMENTATION RESULTS.

Number of Gaussians	TIME (MS)	RTF	GFLOPS
32	2.27	0.227	10.7
64	4.39	0.439	11.1
128	8.55	0.855	11.4

The evaluation of the likelihoods can be implemented in two ways: the first possibility is to create a single kernel that computes the dot product, as well as the logarithmic addition. This approach can decrease the required memory bandwidth and reduce the overhead of running two separate kernels. On the other hand, it is much more difficult to propose an optimal block/grid architecture for both parts of the kernel. The second way is to create a separate kernel for each part of the algorithm. The dot product part implemented as a matrix-matrix multiplication can be computed very efficiently [16]. The efficient implementation of *sgemm* is in the CUBLAS library [17]. Thus, one does not need to create a new (probably less efficient) kernel. The efficient logarithmic addition implementation can be adopted from the parallel sum algorithm [18]. The native *log* and *exp* functions should be used to get the maximum performance. This two-kernel approach is simple and can help realize a satisfactory performance, especially if the overhead is marginalized by computing a large block of feature vectors simultaneously (i.e., large n).

A. *Cardinal's implementation*

The implementation developed at CRIM and described in [1] is a single-kernel approach. Each block computes the final likelihood of one mixture of Gaussians. As a result, the number of launched blocks is the number of distributions in the acoustic model. Each block contains 256 threads. The main computationally intensive part is the *multiply-add* (MAD) operation in the dual while-loop. From the viewpoint of performance tuning, it is crucial that, together with this single instruction, many other operations need to be performed: the while-loops' stopping conditions need to be evaluated, two operands need to be fetched from the global memory, another additional operand needs to be loaded from the shared memory and, finally, the result needs to be stored back in the shared memory. In this case, the main bottleneck is caused by the fetches from global memory. To perform a single-clock instruction, two 32-bit floats (i.e., 8 bytes) need to be fetched. This reduces the maximum performance of this implementation to less than 1/15 of the peak FLOPS on all current GPUs. In addition, the two tied loops and combined read-store operation with shared memory affect the performance, but they are not the main bottlenecks in this case.

The implementation on a GeForce 8800GTX card performs 5× faster than a CPU-only SSE-based vectorized implementation. The tested acoustic model had 39 dimensions and consisted of 4600 states with 32, 64, or 128 Gaussians. The *real-time factor* (RTF) is estimated on the basis of 100 frames/s. The elapsed times per single feature vector, together with floating-point operations per second (FLOPS), are presented in Table 1. The RTF is the ratio between the computing time and length of the processed speech signal. Note that these numbers are only for acoustic model evaluation, not for full speech decoding. The theoretical peak performance of this card is 346 GFLOPS (for MAD instructions only), and the real achieved performance on matrix-matrix multiplication is about 200 GFLOPS [16]. The main bottleneck of this implementation is described above. In addition, the computation of a single feature vector suffers from the high overhead of frequent CPU-GPU transfers. Therefore, a larger window needs to be used for efficient GPU utilization. A significant redesign of the HMM decoder may be necessary to allow the blocked computation of AM likelihoods.

B. *Dixon's implementation*

In contrast, the implementation developed at the Tokyo Institute of Technology and described in [2] and [19] uses a two-kernel approach. The CUBLAS library is used for matrix-matrix multiplication, and an extra kernel is used to perform the final logarithmic addition. To achieve the maximum performance for the entire decoder, the GPU part runs completely asynchronously. Two buffers for page-locked memory are used; the CPU decoder works with one, while the GPU is preparing the other in parallel. This is a way to hide the CPU-GPU transfer overhead as well as the GPU processing time itself. The proper window length was analyzed in that study, and a length higher than 10 was recommended. Thus, window lengths of 8 or 16 should be used for real-time decoders, and even larger windows can be used for offline tasks. In [19], the use of 16-bit half floats was also tested for storing model parameters. Only a negligible change was reported in the recognition accuracy. The model size in memory is not a significant issue if the GPU is used only for acoustic model evaluation. However, in cases where the GPU is shared between several tasks or is also used for a decoder, model size reduction can be helpful.

The implementation was tested for the recognition of spontaneous Japanese speech on a GeForce 8800GTX GPU. The feature-vector length was 38, and the acoustic model contained 3000 states, with the Gaussian component counting 2, 4, ... , 512. The feature-vector window size was 32. Performance results are presented for the acoustic model with the decoder, not for the model alone. We put forth our best effort to estimate the acoustic-only performance numbers from the presented graphs. These estimated results are shown in Table 2. This implementation is much faster than Cardinal's approach, as previously described, mainly because of the optimized *sgemm* from the CUBLAS library and the large feature-vector window. However, the performance is still far from that of matrix-matrix multiplication alone. Using the general *sgemm* is not an optimal solution for badly shaped matrices and cannot harness all the optimization possibilities of the likelihoods computation task (especially appropriate data re-use). Moreover, storing all the results at the end of the first kernel and re-reading them again during the start of the second kernel is wasteful. A well-optimized single-kernel approach should be able to achieve better performance.

TABLE III
KVETON'S IMPLEMENTATION RESULTS.

GPU	Total # of Gaussians	Approach	RTF	GFLOPS
GTS 250	50k	Full	0.05	17
		Hierarchical	0.03	28
GTX 285	50k	Full	0.04	21
		Hierarchical	0.02	42
GTS 250	150k	Full	0.15	17
		Hierarchical	0.07	36
GTX 285	150k	Full	0.11	23
		Hierarchical	0.05	51

C. Kveton's implementation

In contrast with the previous implementations, an implementation developed at IBM and described in [3] uses a more general OpenCL programming tool than the NVIDIA-specific CUDA tool. This implementation is based on the two-kernel approach. The first dot-product kernel is implemented as a matrix-matrix multiplication of a window of feature vectors (length is 16). This full acoustic likelihood computation is considered to be a baseline for further describing a hierarchical approach. This approach is a Gaussian selection method based on a hierarchical scheme of clustered Gaussians. At run time, only the top scoring clusters are evaluated. The authors used 1024 clusters, with the top ¼ evaluated. Thus, the theoretical speed-up is 4× as compared to full computation. This technique is well suited for implementation on a CPU. The GPU implementation is not trivial and is described in detail in that paper. The actual speed-up is about 2× in comparison to a full GPU approach.

The full and hierarchical approaches were tested on GeForce GTS 250 and GTX 285 GPUs. Two acoustic models with 40 dimensions were tested. The first *small* model had 2 k states and 50 k Gaussians in total. The *large* one had 6 k states and 150 k Gaussians. The number of Gaussians per state was not fixed, but the approach does not support this kind of model. Therefore, *fake*-Gaussians (about 30%) were added to maintain uniform numbers. The results for only the acoustic likelihoods are shown in Table 3. GFLOPS values for the hierarchical approach were evaluated in the same manner as that for full computation (pruned Gaussians are taken as *computed*). The performance was better than Cardinal's but worse than Dixon's, even if the used GPUs were faster (GTS 250 is about 1.35× faster than 8800GTX; GTX 285 is about 2× faster). Even the hierarchical approach did not outperform its competitors, although paying more attention to the optimization of this algorithm would certainly greatly improve its performance. Some of the slowdown could have been caused by the use of OpenCL. We have analyzed the difference between the CUDA and OpenCL implementations of a compatible method in the Results section of this paper.

TABLE II
DIXON'S IMPLEMENTATION RESULTS.

Number of Gaussians	RTF	GFLOPS
128	0.08	77
256	0.15	82
512	0.3	82

D. Chong's implementation

The implementation developed at the University of California at Berkeley and described in [6], [7] is not for the acoustic likelihood computation only but a whole GPU-implemented LVCSR, as noted in the Introduction. The classical Viterbi-based LVCSR and a weighted finite state transducer based LVCSR were implemented. When the acoustic likelihood computation is performed on a CPU, it represents more than 80% of the total computation time. In the GPU LVCSR, the authors reported an occupation of about 50%. This is caused by the better parallelization suitability of the likelihood calculation. The authors used a two-kernel approach to compute these likelihoods. Only the pruned list of states was computed for each time step. It reduced the computation time by 70%. The authors reported that they achieved close-to-peak performance in this phase [7], but no detailed description of the implementation was given. The reported performances on a GeForce GTX 280 for the dot-product and logarithmic addition kernel were 194 and 367 GFLOPS, respectively. The reported time needed for the likelihood computation phase varied from 73 to 178 ms per second of input speech (depending on the pruning settings). However, on the basis of the reported GFLOPS, the likelihood computation should take only about 30 ms per second of speech without any pruning. These numbers do not add-up together very well. Maybe the finding of unique labels and the pruning management limit the high-performance core.

IV. NVIDIA OPTIMIZED IMPLEMENTATION

Our implementation evaluates the likelihoods for all the states and for a window of feature-vectors in advance in asynchronous way -even for states that are not needed by decoder because of pruning. This approach is much more suitable for GPU than an on-demand selection of computed states and the overall performance of entire recognizer is better.

The core of implementation is based on the single-kernel approach to avoid storing and re-reading the intermediate data. Each block manages all the Gaussians of 64 states, together with 8 feature vectors. Therefore, the grid is 2D. Columns are composed of stripes of 8 feature vectors and rows are stripes of 64 states. The number of rows is given by the model states number, and the number of columns depends on the feature-vector window length, which can be controlled by the decoder (according to the real-time/offline scenario). The number of threads per block is equal to the number of evaluated states. The optimal number is 64 in the most cases. To ensure memory address alignment (required for coalesced memory access) all the dimensions (model as well as feature vectors) are padded to be multiples of 4. This padding also enables the use of float4 textures, which are the fastest solution for read-only memory. All the data are rearranged in advance to be in the order they will be read. This step ensures the maximal cache-hit ratio.

All the 8-feature-vector data are loaded into a shared memory buffer in the beginning of the kernel, together with a squares calculation, according to equation (3). This maximizes data reuse. The shared memory buffer size is defined just before the kernel is executed, according to the feature-vector aligned dimension. During the computation, only the model parameters are fetched through the texture cache.

A. *Kernel pseudocode*

Algorithm 1 shows the pseudocode of the kernel. The kernel consists of an initialization part, where the feature vectors are loaded and squares are calculated. In addition, likelihood registers for all 8 feature vectors are defined and set to "log-zero," which means a predefined big-enough negative value. Thereafter, a loop for all the Gaussians begins. First, a set of 8 accumulators is defined and set to zero. Then, an access address for the texture memory is computed using the grid, block, and thread built-in variables. Pre-fetching the model parameters is a good way to at least partially hide the global memory latency. The memory latency is hidden not only by the other running blocks but also by the current block. The pre-fetching technique is favorable in the case where only a few concurrent blocks are running (because of shared memory or the register file limit). We use temporary float4 registers (*_u4*, *_v4*) to store the pre-fetched model values. *u4* and__*u4* contain four consecutive dimensions of μ_{ij}/σ_{ij}^2. *v4* and *_v4* likewise contain $-1/2\sigma_{ij}^2$.

The most computationally-intensive part of the kernel is the body. The body is composed of an inner loop that iterates through all the dimensions. This loop is unrolled by factor 4. Unrolling improves the body's algorithmic intensity and works nicely with the used float4 texture data type. The body of this loop begins with loading management. The pre-fetched values are copied into another register (*u4*, *v4*) and the next values are pre-fetched into the original variables. The texture address shift is given by the number of threads in the block. This is possible because of a careful reorganization of the model parameter memory layout. The optimized memory layout ensures good texture cache utilization and maximizes the data throughput. We bind textures as a linear memory of the float4 data type. The rest of the body of the inner loop consists of 64 MAD instructions that accumulate four dimensions from 8 feature vectors multiplied by the appropriate model parameters *u4* and *v4*. The creation of a large block of computation-only instructions is a key approach for achieving good performance. The large block also ensures an efficient hiding of global memory latency. In our case, one MAD operand is loaded from shared memory. The same 32-bit word is loaded by all the threads. Therefore, no bank-conflicts can arise. An instruction with shared-memory operands is slower than a registers-only instruction. In our

ALGORITHM I
OPTIMIZED KERNEL PSEUDOCODE.

```
1:  fetch all entire 8 feature-vectors to shared memorybuffercompute squares to the second half of the buffer
2:  __syncthreads()
3:  set 8 likelihood registers to "log-zero"

4:  loop for all Gaussians

5:      set 8 accumulators to zero
6:      compute address into model texture memory
7:      pre-fetch model parameters for the first 4 dimensions (_u4, _v4)

8:      loop for all dimensions/4 – unrolled by factor 4

9:          copy pre-fetched _u4 and _v4 to another registers u4, v4
10:         adjust address
11:         pre-fetch next _u4, _v4

12:         compute unrolled block of 64 MAD instructions
                8 vectors × 4 dimensions × 2 (u,v) = 64 MAD instructions
                use #pragma unroll or manual unrolling

13:     end of loop for dimensions

14:     fetch K constants for actual Gaussians (a float per thread)
15:     finalize all 8 accumulators and do addLog()

16: end of loop for Gaussians

17: store final likelihoods
```

case, the MAD instruction takes 6 clocks instead of 4 clocks for the registers-only variant. This reduces the maximal throughput to 2/3 in the MAD part of the kernel, but there was no other faster solution without using shared memory because register space is a scarce resource. A similar architecture is used in Volkov's optimized matrix-matrix multiplication algorithm [16].

After the inner loop, the accumulators for the actual Gaussian are finalized with the addition of fetched constant *K*, which is defined by equation (2) above. Then, the likelihoods for all the feature vectors are updated using a logarithmic addition function. We implemented the addLog function in the following way:

```
mx = max(x1, x2);
y = mx + __logf(1.0f + __expf(min(x1, x2) – mx));
```

where *max*, *mix*, *__logf*, and *__expf* are fast GPU-native functions. This addLog variant is accurate and fast enough. Therefore, no approximation, which is often used in CPU implementation, is needed. At the end of the kernel, the final likelihoods are stored back into the global memory. The writes should be coalesced, but the performance of this stage is actually not very important.

To achieve the maximum performance, the input model parameters in the memory layout must match the fetching order. The memory ordering is the same for the μ_{ij}/σ_{ij}^2 texture (designated as *u4* in kernel pseudocode) as for $-1/2\sigma_{ij}^2$ (designated as *v4*). The order is schematically illustrated in Fig. 1 via a bottom-up schema. The basic building block (Fig. 1A) consists of four 32-floats. It is a vector of four consecutive dimensions. This vector is read by the single thread as a float4 data type. The next upper block (Fig. 1B) consists of 64 float4 vectors. They are read by all 64 threads running in the actual block. These threads access consecutive memory addresses; therefore, the memory access is coalesced. These blocks are read in the inner loop sequentially for all the dimensions; therefore, they must be stored in memory consecutively according to the dimension (Fig. 1C). One block contains a complete set of parameters of one 64-states Gaussian. The next upper block consists of all the Gaussian blocks according to the outer loop (Fig. 1D). The memory-block is read by all the blocks belonging to the same row of the grid. The entire texture memory is composed of all these Gaussian memory-blocks for all the states of the acoustic model (Fig. 1E).

B. *Variable number of Gaussians per state*

Our implementation can also easily support acoustic models with a variable number of Gaussians per state. Only minor changes are required. Because the 64-state blocks are computed independently, a constant number of Gaussians per state has to be ensured only within the individual blocks. Therefore, the model states are sorted in advance according to their Gaussians per state numbers and divided into 64-state blocks. The state-blocks are padded, if necessary. The parameter textures for both models are composed in the same way as in Fig. 1. Only an additional memory-offset vector needs to be passed into the kernel because the 64-state

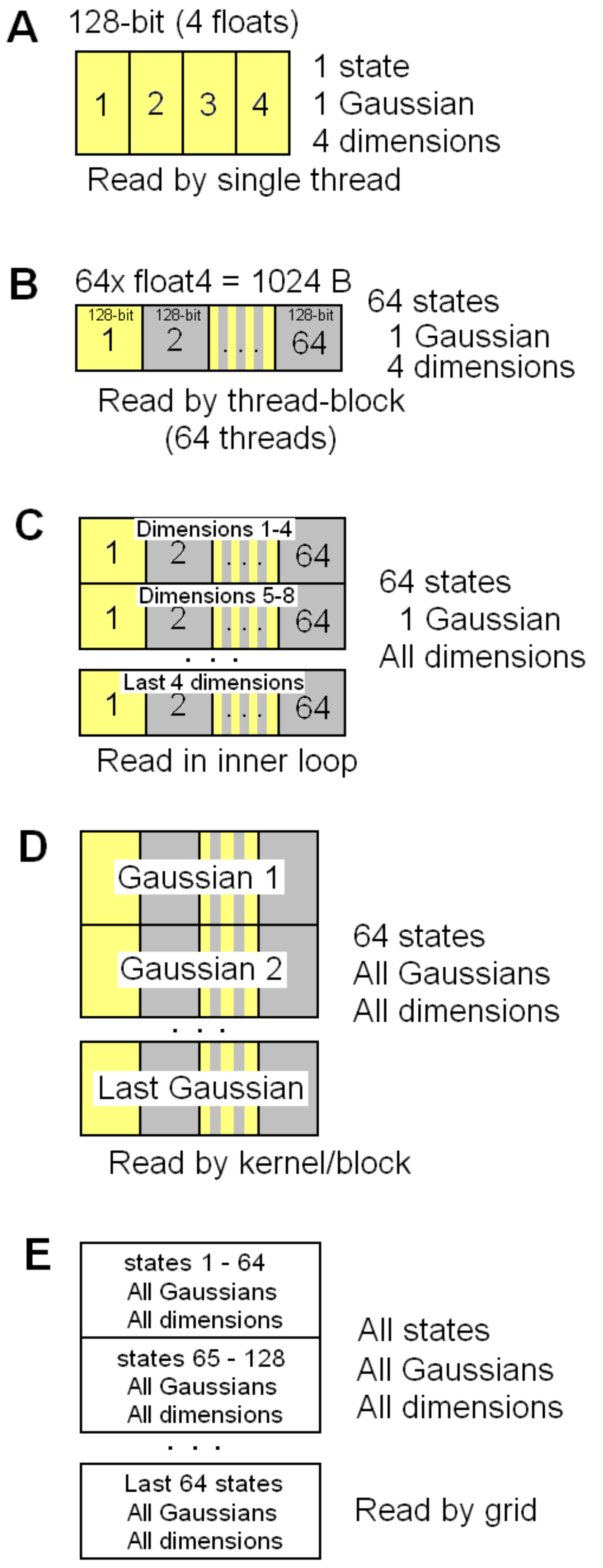

Fig. 2 Model parameters texture memory arrangement bottom-up schema. Upper memory blocks consist of consecutively laid-down bottom blocks. This memory arrangement fits the kernel read order and ensures maximum memory reading performance.

memory-blocks (Fig. 1D) do not have a constant size. Moreover, a vector with state-indexes can be passed into the kernel, and then the final likelihoods can be stored in the original order if necessary.

C. *Final tuning*

The number of threads in a block (number of evaluated states) can be tuned. A low number of threads (one warp) exhibits a low memory-latency hiding ability. In contrast, too large a number of threads (128 and more) limits the number of active blocks per multiprocessor because of the limited number of registers. An overly large thread-block can increase the number of evaluated virtual (padded) Gaussians and states, especially for models with variable numbers of Gaussians per state. Therefore, the state-block size should be tuned for the individual acoustic model's shape and hardware resources because different GPU models vary greatly in their architecture–they differ in register-file size as well as computation/memory performance ratio, for example. According to our experience, the 64-thread-block is optimal in most cases.

Although the pre-fetching technique helps to better hide memory latency in cases where very low number of blocks is running, it consumes additional registers. From our experience, Fermi-based cards obtain better results without the pre-fetching technique.

Fermi-based cards also suffer from a lower memory bandwidth to computing power ratio. This means the total kernel performance is limited by the memory bandwidth, even if the memory latency is well hidden. The limit can be overpassed by higher data reuse. Therefore, we have also implemented a 16-vector kernel version. This kernel computes 16 feature vectors simultaneously, where double model parameter reuse is in place. The 16-vector kernel over-performs the 8-vector version by 15–20% on Fermi-based cards. On other cards, the performance does not improve.

D. *Kernel asynchronous calls*

Our kernel management is almost identical to Dixon's, which was described in a previous section. We also use two page-locked host memory buffers. One is used for a decoder, and the other is a destination for likelihoods of the next feature-vector window. The complete GPU call is done in an asynchronous fashion. Therefore, no CPU thread is blocked during the GPU activity. The only exception is when a small acoustic model is used. If a kernel call is very short, we found that the asynchronous call management overhead can exceed the positives and a synchronous approach is faster. This is the case when the kernel processing time is shorter than about 4 ms. However, in this case, the acoustic model computation is fast enough anyway.

V. ATI/AMD OPTIMIZED IMPLEMENTATION

Because OpenCL is an open standard supported by several manufacturers, it theoretically enables the use of the same code on ATI/AMD GPUs. However, because of the different architecture, the performance of code optimized for one kind of device or one manufacturer will generally be poor when used on a different device. If support for both manufacturers is necessary, it is possible to either use CUDA for NVIDIA GPUs and OpenCL for ATI GPUs or to use the OpenCL framework for both, together with kernels specifically implemented and tuned for the given device architecture.

The general structure of our ATI/AMD OpenCL kernel is the same as the CUDA kernel structure (see Algorithm 1). It consists of the same two loops. The outer goes through the Gaussians, while the inner unrolled loop goes through the feature-vector dimensions. The outer loop is also unrolled by factor 4 to improve the algorithmic intensity of the kernel core. Further, in contrast to the CUDA kernel, no local/shared memory is used. All the needed data are loaded at the beginning of the inner loop using 2D float4 textures. The computing kernel body inside the inner loop simultaneously processes 8 feature vectors, 4 consecutive dimensions, and 4 consecutive Gaussians of 64 states. The kernel body consists of two blocks of 32 float4 MAD instructions. The first one performs the calculations with μ_{ij}/σ_{ij}^2 model parameters. Before the second, the squares of the feature-vector data are calculated. Then, the other 32 float4 MAD instructions are performed on the $-1/2\sigma_{ij}^2$ model parameters. This large block of float4 instructions utilizes the entire multiprocessor well and helps to hide the texture memory latency. In addition, the packed-float4 instructions are used in the logarithmic addition section of the kernel. The number of running threads per block is 64, which is also the warp-size at the ATI/AMD GPUs.

VI. RESULTS

In addition to using the described implementations together within the speech decoder, we prepared a stand-alone application for benchmarking the AM likelihoods evaluation only. It randomly generates the input data as well as the model parameters, allowing the performance with various model sizes and shapes to be easily evaluated. We use RTF and FLOPS measures for a performance comparison. The elapsed time is measured, including the host-device memory transfers. We define the total number of float-operations needed to correctly compute the FLOPS measure as the sum of the dot-product part and logarithmic addition part. The number of operations in the dot-product part is 4 per dimension per Gaussian. The number of addLog() operations per Gaussian, according to our implementation, is 9. The real number of operations/clocks is implementation and hardware specific, and the throughput of the *log* and *exp* functions is usually much worse than multiplication or addition. The total number of operations depends on the number of evaluated feature vectors, the dimension, and the total number of Gaussians.
CUDA toolkit 3.2 and 263.06 drivers were used for the NVIDIA cards. ATI Stream SDK v2.3 and an ATI Catalyst 10.12 driver were used for the ATI cards. All the tests were run under Windows XP 32-bit.

A. *Performance comparison of various GPUs*

First, we tested a subset of the GPUs available to us using a very large model, together with a large feature-vector window. This setup suppressed the CPU-GPU communication overhead and examined both the maximum GPU performance and implementation performance/quality. We chose a 5000-state model with 256 Gaussians per state and 36 dimensions. This model had 1,280,000 Gaussians in total. The feature-vector window length was 256, and the total number of 2560 vectors was computed during the benchmark. RTFs were calculated on a 100 vectors per second basis.

The results are shown in Fig. 2 and Fig. 3. The green bars denote NVIDIA GPUs, where the CUDA implementation was used. The red bars denote ATI GPUs, where OpenCL was used. The performance scores in GFLOPS are shown in Fig. 2. The real-time factors (RTFs) are shown in Fig. 3. The measured performance is very close to well-optimized matrix-matrix implementations. The remaining gap is mainly caused by the lower throughput of the *log* and *exp* functions during the logarithmic addition phase. The measured RTFs show that even a laptop GPU is able to process this very large model in less than half of the real-time. Desktop models are much faster and achieve elapsed times that are 7 to 50 times shorter than real-time. The ATI GPUs power is successfully utilized. Thus, the ATI cards achieved much better results than the comparable NVIDIA cards because of higher raw computational throughput. The results indicate that practically any GMM-based acoustic model can be used in real-time applications, even with a low-end, mainstream, or even laptop GPU. In addition, the offline recognizers can be significantly speeded-up if the decoder part is powerful enough.

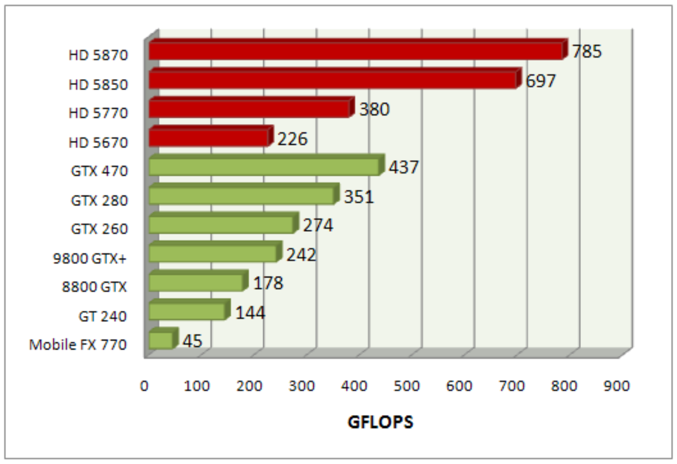

Fig. 3 Performance in GFLOPS of our optimized implementation for various ATI (red/dark) and NVIDIA (green/light) GPUs.

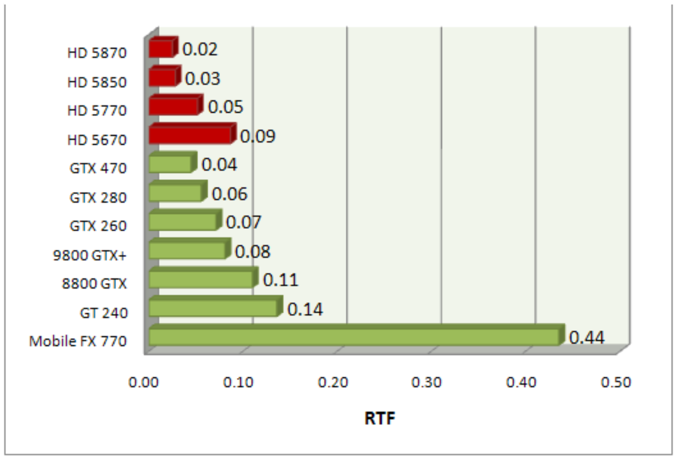

Fig. 4 Real-time factors (RTFs) of our optimized implementation for various ATI (red/dark) and NVIDIA (green/light) GPUs. Tested on model with 1,280k Gaussians in total. RTF calculation is based on 100 vectors per second rate.

B. *CUDA and OpenCL comparison*

In this subsection, we compare the CUDA and OpenCL implementations. We proposed two optimized OpenCL implementation: one for NVIDIA GPUs and the other for ATI/AMD GPUs. A cross-test was also performed. The results are shown in Fig. 4 for the NVIDIA GT 240 and ATI HD 5670 cards. The test setup and model metrics are the same as in the previous subsection. There is almost no difference between the CUDA and equivalent OpenCL implementations. The only difference is that 2D textures are used in the OpenCL implementation because texture-cached linear memory is not supported. The cross-test showed that the optimization techniques are really architecture-specific and at least two architecture-specific variants are necessary.

A comparison of the overheads for the implementations was also very interesting. We compared the performances of the CUDA and OpenCL implementations for various feature-vector lengths. The results are shown in Fig. 5. Six window sizes ranging from 8 to 256 were tested on a smaller acoustic model with 16 Gaussians per state and 5000 states. The results show that OpenCL is a little slower for longer window-sizes but the overhead is significantly smaller, which causes lower elapsed times for small window-sizes. The distinct part of the overhead is not caused solely by the CPU-GPU memory transfers. The kernel-only times are also significantly higher, more than double in our 8-vectors case. In our case, the total overhead varied from 0.3 to 1 millisecond per kernel run. In the case of real-time speech recognition, the overhead is not a major problem anyway because the decoder part is significantly slower, and the overhead is therefore hidden with a large margin.

The overhead size and composition depend on the individual hardware, and probably even software, setup. Using the GPU during computation as a system display for windows-based operating systems can also play a role (the GPU in Fig. 5 was used as the display). The conclusions drawn from these results lead us to recommend the use of kernels that are as long as possible for maximum performance. On the other hand, if the GPU is used as the system display, kernel computations that are too long cause the display response to "freeze." The kernel computing time should vary between 10 and 50 ms to reduce the freezing as well as the overhead.

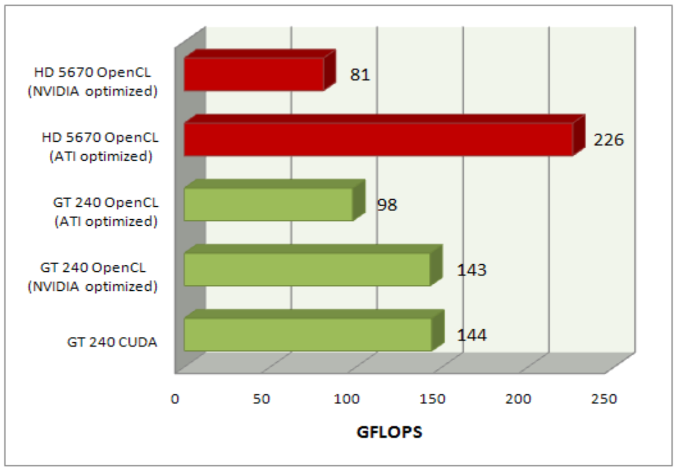

Fig. 5 Performance in GFLOPS of various optimized implementations for ATI HD 5670 (red/dark) and NVIDIA GT 240 (green/light) GPUs.

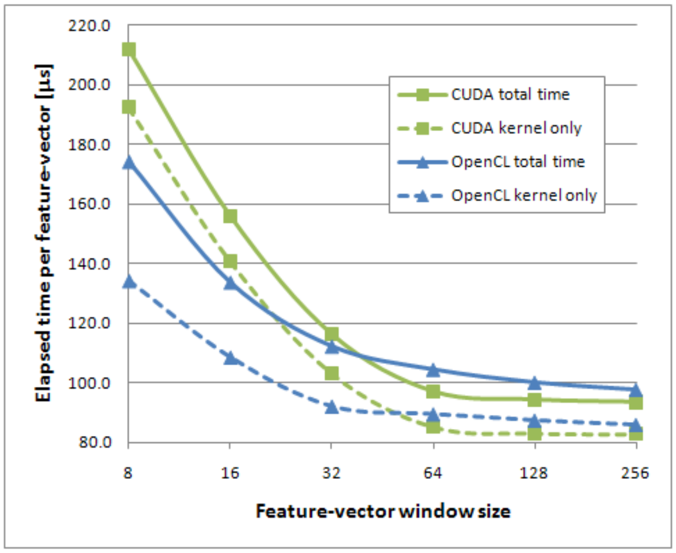

Fig. 6 Elapsed time per feature vector on GT 240 GPU for various vector-window sizes. Both CUDA and OpenCL implementations were tested and the total as well as kernel-only elapsed times were measured.

C. *Evaluation with recognizer*

In this subsection, practical experiments with a real recognizer are presented. The recognizer was designed for both off-line and real-time applications. For the evaluation, we used the same data, which was used for the automatic captioning of parliamentary sessions [20]. The training data for the acoustic model consisted of 200 hours of parliament speech records. The digitization of an analogue signal was carried out at a 44.1-kHz sample rate in a 16-bit resolution format. We used PLP features with delta and delta-delta coefficients. The feature vector had a total of 36 dimensions. Feature vectors were computed at 10 ms intervals (100 vectors per second). The acoustic model consisted of 5385 states, and each state had 16 or 36 Gaussians. The total numbers of Gaussians were 81 k and 194 k. When evaluated on a CPU, the smaller model was the largest one that fit into the real-time constraints with some margin if a Gaussian-pruning fast evaluation algorithm was used. The large model was the best performing model. A higher number of Gaussians did not bring a significantly better recognition performance.

The test set consisted of an hour of parliament speech. A trigram language model was trained using about 20 M tokens of normalized Czech Parliament transcriptions. The dictionary size was 186 k of words. The recognition accuracy and RTF were evaluated for four different pruning settings of the recognizer. The 8-feature-vector window was used. This window size is commonly used for real-time applications.

In our experiment, we tested both acoustic models. Three approaches were evaluated for the small model. Two were computed with a CPU. The first reference approach computed all the model Gaussians (referred to as *16G_CPUfull*) on a single CPU core. The second approach was based on Gaussian pruning and referred to as *16G_CPUfast*. The third approach was the GPU-CUDA implementation described in this paper. The GPU-only approach was used for both the small and large models (referred to as *16G_GPU* and *36G_GPU*). The experiments were performed on an Intel Core2 Quad 2.83 GHz CPU together with a GTX 260 GPU. The decoder part of the recognizer used all four cores of the CPU. The results are shown in Fig. 6. The *CPUfull* approach achieved good accuracy but was a long way from real-time performance. In contrast, the *CPUfast* algorithm was much faster and ensured the real-time constraints with a margin, but at a price of about a 1% drop in accuracy. The GPU-implementations had no problem with speed, and the RTF difference between the small (16G) and large (36G) models was not significant. The large model also had a small improvement in accuracy. Therefore, employing the GPU opens two sources of accuracy improvement in real-time systems. The first source is the full acoustic model processing without the need for any pruning or approximations. The second source is the possibility of using much larger models. In many speech recognition tasks, it is now possible to process even bigger model in real-time with the aid of the GPU than we are able to robustly train because of the lack of data.

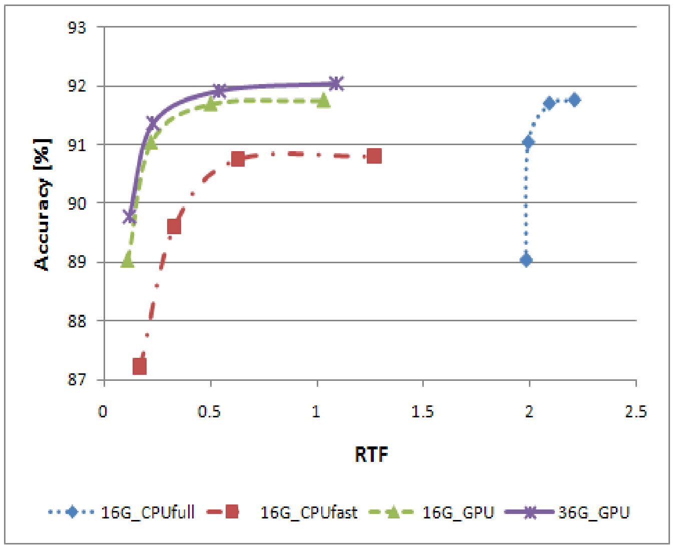

Fig. 7 Parliament speech recognition experiment. Dependence of the recognition accuracy on the recognizer speed for four different decoder pruning settings and four acoustic model evaluation variants.

VII. Conclusion

In this paper, we have described our GPU implementation of acoustic model likelihoods computation; it shows close to the peak performance on many GPUs and is significantly faster than the previously published implementations. We presented and compared CUDA and OpenCL implementations optimized for NVIDIA GPUs. In addition, the OpenCL implementation optimized for ATI/AMD GPUs has been described and the results are presented. The ATI GPUs performed better than the comparable NVIDIA GPUs in this task. The results of tests with a recognizer suggest that during speech recognition, it is now possible to use any large acoustic model that can be reliably trained. Moreover, fusion techniques for the simultaneous evaluation of a large set of models can now be applied to real-time recognition.

ACKNOWLEDGMENT

The authors would like to thank their colleagues (Aleš Pražák, Lukáš Machlica, Zbyněk Zajíc, and Marek Hrůz) for testing the presented implementations on their hardware. The authors would also like to thank members of the BOINC distributed computing community - Czech National Team for the testing performed on their hardware: Dušan Vykouřil (forest), Miloslav Machát (Indy), Frenk, Martin Krahulík (krahulik), and Jakub Sedláček.

This research was supported by the Grant Agency of the Czech Republic, project No. GAČR P103/12/G084.

REFERENCES

[1]P. Cardinal, P. Dumouchel, G. Boulianne, and M. Comeau, "GPU accelerated acoustic likelihood computations," in *Proc. of INTERSPEECH 2008*, pp. 964-967. Brisbane, Australia, September 23-26, 2008.

[2]P. R. Dixon, T. Oonishi, S. Furui. "Harnessing graphics processors for the fast computation of acoustic likelihoods in speech recognition," *Computer, Speech and Language,* pp. 510-526,

[3]P. Kveton and M. Novak, "Accelerating hierarchical acoustic likelihood computation on graphics processors," In *Proc. of INTERSPEECH 2010*, pp. 350-353. Makuhari, Japan, September 26-30, 2010.

[4]K. Gupta and J. D. Owens, "Three-layer optimizations for fast GMM computations on GPU-like parallel processors," in *Proc. of IEEE ASRU*, pp. 146-151, Merano, Italy, 2009.

[5]J. Vanek, "Discriminative training of acoustic models," Ph.D. dissertation, Dept. of Cybernetics, Univ. of West Bohemia, Pilsen, Czech Republic, 2009. (in Czech)

[6]J. Chong, Y. Yi, A. Faria, N. Satish, and K. Keutzer, "Data-parallel large vocabulary continuous speech recognition on graphic processors," *Tech. Rep.* UCB/EECS-2008-69, EECS Department, University of California, Berkeley, 2008.

[7]J. Chong, E. Gonina, Y. Yi, and K. Keutzer, "A fully data parallel WFST-based large vocabulary continuous speech recognition on a graphics processing unit," In *Proc. of INTERSPEECH 2009,* pp. 1183-1186. Brighton, United Kingdom, September 6-10, 2009.

[8]J. Vanek and J. V. Psutka, "Gender-dependent acoustic models fusion developed for automatic subtitling of parliament meetings broadcasted by the Czech TV," *Text, Speech and Dialogue, Lecture Notes in Computer Science*, Volume 6231/2010, 431-438. Springer, Berlin, 2010.

[9]S. Padalikar, G. Diamos, "Exploring The Latency and Bandwidth Tolerance of CUDA Applications," NFinTes Tech Report, December 2009.

[10] Khronos Group Std., "The OpenCL specification, version 1.1," http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[11] AMD Company, "AMD Accelerated Parallel Processing, OpenCL Programming Guide," http://developer.amd.com/gpu.

[12] NVIDIA Corporation, "The OpenCL best practices guide," http://developer.nvidia.com/object/cuda_3_2_downloads.html

[13] NVIDIA Corporation, "CUDA C Programming Guide 3.2," http://developer.nvidia.com/object/cuda_3_2_downloads.html

[14] NVIDIA Corporation, "CUDA C Best Practices Guide 3.2," http://developer.nvidia.com/object/cuda_3_2_downloads.html

[15] P. R. Dixon, D. A Caseiro, T. Oonishi, and S. Furui, "The titech large vocabulary WFST speech recognition system," in *Proc. of IEEE ASRU*, pp. 443-448, Dec, Kyoto, Japan, 2007.

[16] V. Volkov and J. W. Demel, "Benchmarking GPUs to tune dense linear algebra," in *Proc. of ACM/IEEE Conference on Supercomputing (SC08*), Austin, Texas, 2008.

[17] NVIDIA Corporation, "CUBLAS user guide 3.2," http://developer.nvidia.com/object/cuda_3_2_downloads.html

[18] D. B. Kirk and W. W. Hwu, "Programming massively parallel processors: A hands-on approach," Morgan Kaufmann, San Francisco, 2010.

[19] P. R. Dixon, T. Oonishi, and S. Furui, "Fast acoustic computations using graphics processors," in *Proc. of IEEE ICASSP*, pp. 4321-4324, Apr, Taipei, Taiwan, 2009.

[20] A. Prazak, J. Psutka, J. Hoidekr, J. Kanis, L. Muller, and J. Psutka, "Automatic online subtitling of the Czech parliament meetings," *Text, Speech and Dialogue*, *Lecture Notes in Artificial Intelligence,* pp. 501-508, Springer, Berlin, 2006.

Jan Vaněk received the M.Sc. degree equivalent in cybernetics in 2003 and the Ph.D. degree in cybernetics in 2010, both from the University of West Bohemia, Plzeň, Czech Republic.

He is currently a Research Assistant at the Department of Cybernetics, University of West Bohemia, since 2010. He was working also at the Institute of Physical Biology in Nové Hrady, since 2006 to 2011. His research interests include speech and speaker recognition, acoustic modeling, signal and image processing and GPGPU programming.

Jan Trmal has received the M.Sc. degree equivalent from the University of West Bohemia, Department of Cybernetics in the field of automatic speech recognition.

He is currently pursuing his PhD at the same department. His interests include GPGPU programming, artificial neural networks and development of captioning systems.

Josef V. Psutka received the M.Sc. degree equivalents in cybernetics in 2001 and in mathematics in 2005, and the Ph.D. degree in cybernetics in 2007, all from the University of West Bohemia, Plzeň, Czech Republic.

He was a Research Assistant in the Department of Cybernetics, University of West Bohemia, from 2001. He is currently an Assistant Professor at the same department. His research interests include mainly speech signal parameterization and acoustic modeling methods for automatic speech recognition.

Josef Psutka received the M.Sc. degree equivalent in electrical engineering and the Ph.D. degree in cybernetics from the Czech Technical University, Prague, Czech Republic, in 1974 and 1980, respectively.

He worked as an Assistant Professor in the Technical Institute, Plzeň, Czech Republic, from 1978 to 1991. In 1991, he joined the Department of Cybernetics, University of West Bohemia, Plzeň, as an Associate Professor, and became a Full Professor in 1997. His research interests include speech signal processing, acoustic modeling, large-vocabulary ASR, speech synthesis, and pattern recognition.