Optimization of the Gaussian Mixture Model Evaluation on GPU

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

Department of Cybernetics, University of West Bohemia
Univerzitni 8, 306 14 Plzen, Czech Republic

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

Abstract

In this paper we present a highly optimized implementation of Gaussian mixture acoustic model evaluation algorithm. Evaluation of these likelihoods is one of the most computationally intensive parts of automatics speech recognizers but it can be well-parallelized and offloaded to GPU devices. Our approach offers significant speed-up compared to the recently published approaches, since it exploits the GPU architecture better. All the recent implementations were programmed either in CUDA or OpenCL GPU programming frameworks. We present results for both; CUDA as well as OpenCL.

Results suggest that even very large acoustic models can be utilized in real-time speech recognition engines on computers and laptops equipped with a low-end GPU. Optimization of acoustic likelihoods computation on GPU enables to use the remaining GPU resources for offloading of other compute-intensive parts of LVCSR decoder.

Other possible use of the freed GPU resources is to evaluate several acoustic models at the same time and use fusion techniques or model selection techniques to improve the quality of resulting conditional likelihoods under diverse conditions.

Index Terms: CUDA, GPU, OpenCL, Gaussian Mixture Models

1. Introduction

Large vocabulary continuous speech recognition (LVCSR) is a highly computationally intensive task. The acoustic model likelihoods computation accounts for the largest processing part. In a few recent papers [1], [2], and [3] a GPU was employed as a coprocessor to compute these likelihoods. The speed-up $4\times$ – $6\times$ was achieved for the full acoustic model evaluation itself. This leads to LVCSR system speed-up by factor $1.3\times$ – $3\times$, depending on individual task and setup, while keeping the accuracy untouched. For real-time applications, CPU-only recognizer implementations have to often employ simplified (i.e. smaller) models, (acoustic model) pruning, and computational approximations, or combinations of these techniques to fit into the real-time constrains. Thus, by using a hybrid CPU-GPU implementations, an increase of accuracy can be achieved since even low-end GPUs are powerful enough to evaluate significantly larger models in real-time; without necessity of any additional acceleration or complexity reduction techniques.

We are interested in real-time speech applications (e.g. generation of automatic captions, dialog systems). The well-utilized GPU power enables us to evaluate a set of speaker-specific (or speaker-cluster-specific) acoustic models at once in real-time and combine the individual models likelihoods together using a fast fusion method [4]. In addition, a combination of full speaker-cluster-specific models with speaker-specific feature-transformation matrices gives us an opportunity to prepare several tens of models, which in turn enables the recognizer to handle large inter- and intra- speaker variability. Only the well-optimized GPU implementation has enough performance to deal with such large amount of models in real-time.

2. General Purpose Computation on GPU

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

Due to the distinctive characteristics of GPU architecture (high speed, high latency main memory, limited caching capabilities, limited communication with CPU, minimal thread switching and planning overhead), the common CPU programming models (and programming languages based on this programming models) are not suitable for GPU programming. In order to achieve 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 explicit expression of programmer's intentions.

NVIDIA's CUDA (Compute Unified Device Architecture) has gained a wide acceptance, however the CUDA standard is proprietary and the intellectual property concerns led to development of an open standard OpenCL (Open Computing Language). The OpenCL standard was developed in cooperation with teams from ATI/AMD, IBM, Intel, NVIDIA and others and many HW vendors and SW producers announced support of OpenCL in their products. Unfortunately, this wide acceptance does not dislodge the burden of hand-tuning the computational kernels for individual distinct HW architectures.

The NVIDIA GPU consists of many processing elements (PEs) called multiprocessors. Older NVIDIA GPUs have 8 stream processors in each PE together witch 16kB on-chip local memory (called shared memory). Single PE offers 8k or 16k 32-bit registers, depending on GPU series. The Fermi-based PEs are bigger and contain 32 or 48 stream processors. The 64kB on-chip memory can be set to two configurations: 16kB L1 cache and 48kB local memory or vice versa. The peak memory bandwidth from global memory can be achieved only via coalesced access, where consecutive 16 work-items (half-warp) access consecutive addresses[1]. Another example is using of local memory where one should omit bank-conflicts which can significantly degrade the kernel performance. These issues are discussed in more detail in [5].

[1] full-warp is needed on Fermi-based cards

3. Implementations Review

The first use of a GPU for acoustic model likelihoods computation was briefly noticed in [6]. The following year a more detailed paper was published ([1]). Both approaches share the same common ground – computation of likelihoods performed by means of matrix dot-products. The entire acoustic model is represented as a matrix $\mathbf{A}$ in which each row is a log weighted Gaussian component with diagonal covariance matrix. The i-th component of J dimensional mixture model is represented as a vector $\mathbf{a}_i^{\mathrm{T}}$

$$\mathbf{a_i}^{\mathrm{T}} = \left\{ K_i, \frac{\mu_{i1}}{\sigma_{i1}}, \ldots, \frac{\mu_{ij}}{\sigma_{ij}}, -\frac{1}{\sigma_{i1}^2}, \ldots, -\frac{1}{\sigma_{ij}^2} \right\} \quad (1)$$

where

$$K = \log w_i - \frac{J}{2}\log 2\pi - \frac{1}{2}\sum_j \log \sigma_{ij}^2 - \frac{1}{2}\sum_j \frac{\mu_{ij}^2}{\sigma_{ij}^2} \quad (2)$$

The feature vector $\mathbf{x}^{\mathrm{T}}$ is then expanded to

$$\mathbf{x}_e^{\mathrm{T}} = \left\{1, x_1, \ldots, x_J, x_1^2, \ldots, x_J^2\right\} \quad (3)$$

The evaluation of score of every Gaussian can be performed as the matrix-vector multiplication $\mathbf{y} = \mathbf{A}\mathbf{x}_e$. The output vector $\mathbf{y}$ is a vector containing log weighted scores of every Gaussian component of the mixture (given the input vector $\mathbf{x}$ of the feature-vector for every Gaussian component). An evaluation of n following feature vectors can be performed as a single matrix-matrix multiplication $\mathbf{Y} = \mathbf{A}\mathbf{X}_e$, where the columns of the $(2J+1) \times n$ matrix $\mathbf{X}_e$ are the expanded feature vectors. The final mixtures likelihoods are obtained by logarithmic summation of all the relevant Gaussians.

The evaluation of 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 required memory bandwidth and reduce the overhead of executing two separate kernels. On the other hand, it is much more difficult to propose optimal block/grid architecture for both parts of the kernel. The second way is to create a standalone kernel for each part of the algorithm. The dot product part (i.e. matrix multiplication) can be evaluated efficiently using the *sgemm* from the CUBLAS library from CUDA SDK[2]. The efficient logarithmic addition implementation can be adopted from parallel sum algorithm. The native log and exp functions should be used to get maximum performance. This two-kernel approach is simple and can achieve a satisfactory performance, especially, if the overhead is marginalized by using of a large block of feature-vectors at once (i.e. large n).

4. Description of the proposed implementation

Our implementation is based on the single-kernel approach to avoid storing and re-reading the intermediate data. Each block manages all Gaussians of 64 states together with 8 feature-vectors. Therefore, the grid is 2D. Columns are composed by stripes of 8 feature vectors and rows are stripes of 64 states. Number of rows is given by the model states number and number of columns depends on feature-vector window length, which can be controlled by the decoder (according to real-time/offline scenario). Number of threads per block is equal to the number of evaluated states. Optimal number is 64 in most cases. To ensure memory addresses alignment (required to coalesced memory access) all dimensions (model as well as feature-vectors) are padded to be multiples of 4. This padding also enables usage of float4 textures that 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. Entire 8-feature-vector data are loaded into a shared memory buffer in the beginning of the kernel together with squares calculation according to eq. (3). It maximizes the data reuse. The shared memory buffer size is defined just before the kernel is executed according to the feature-vector dimension aligned by 4. During the computation, only the model parameters are fetched through the texture cache.

```
1  fetch all entire 8 feature-vectors to shared memory buffer
2  compute squares to the second half of the buffer
   according to eq. (3)
3  __syncthreads()
4  set 8 likelihood registers to "log-zero"
5  for each Gaussian do
6      set 8 accumulators to zero
7      compute address into model texture memory
8      pre-fetch model parameters for the first 4
       dimensions (_u4, _v4)
9      for all dimensions do
10         copy pre-fetched _u4 and _v4 to another
           registers u4, v4
11         adjust address
12         pre-fetch next _u4, _v4
13         compute unrolled block of 64 MAD instructions
           8 vectors x 4 dimensions x 2 (u,v) = 64 MAD
           instructions use #pragma unroll or manual
           unrolling
14     end
15     fetch K constants for actual Gaussians (a float per
       thread)
16     finalize all 8 accumulators and do addLog()
17 end
18 store final likelihoods
```

Algorithm 1: *An optimized kernel for GMM likelihoods computation*

The pseudo-code of the kernel is depicted as Algorithm 1. The kernel consists from an initialization part where the feature-vectors are loaded and $\mathbf{x}_e$ is calculated. Also, the likelihood registers for all 8 feature-vectors are defined and set to "log-zero" which means predefined big-enough negative value. Thereafter, a loop for all Gaussians begins. At first, the set of 8 accumulators is initialized (set to zero). Then, access address to texture memory is computed using the grid, block, and thread build-in variables. Pre-fetching the model parameters is a good way how to conceal the global memory latency. The computational loop of the body is unrolled by factor 4 to improve the algorithmic intensity and it fits nicely to the used float4 texture data-type. The part of the loop consists of 64 MAD (multiply-and-add) instructions that accumulate four dimensions from 8 feature-vectors multiplied by the appropriate model parameters u4 and v4.

The large block also ensures an efficient hiding of global memory latency. In our case, one MAD operand is loaded from

[2]`http://developer.nvidia.com/object/cuda_3_2_downloads.html`

shared memory. The same 32-bit word is loaded by all threads. Therefore, no bank-conflicts can arise. An instruction with shared-memory operands is slower than registers-only instruction. In our case, the MAD instruction takes 6 clocks instead of 4 clocks for registers-only variant. This of course reduces the maximal throughput to 2/3 in the MAD part of the kernel but there is no other faster solution than using shared memory, since register count is highly limited. A similar architecture is used in an optimized matrix-matrix multiplication algorithm [7]. After the inner loop, accumulators for actual Gaussian are finalized with addition of fetched constant K from eq. (2) above. Then, the likelihoods for all feature-vectors are updated by logarithmic addition function.

At the end of the kernel, the final likelihoods are stored back into the global memory. To achieve the maximum performance, the input model parameters memory layout must match the fetching order. The memory ordering is the same for $\frac{\mu_{ij}}{\sigma_{ij}^2}$ texture (marked as u4 in kernel pseudo-code) as well as for values $-\frac{1}{2\sigma_{ij}^2}$ (marked as v4). Each thread fetches four 32-bit floats as single float4 data-type. It is a vector of consecutive four-dimensions. The model parameters are read by all 64 threads running in actual block. These threads access a 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 dimension. This larger blocks are also ordered in the way that match the fetching order.

4.1. Variable number of Gaussians per state

Our implementation can easily support also acoustic models with variable number of Gaussians per state. Only a minor changes need to be done. Because the 64-states blocks are computed independently, a constant number of Gaussians per state has to be ensured within the individual blocks only. Therefore, model states are sorted in advance according to theirs Gaussians per state numbers and divided into 64-state blocks. Only an additional memory-offset vector needs to be passed into the kernel because 64-states memory-blocks do not have a constant size.

4.2. Final tuning

Number of threads in block (number of evaluated states) can be tuned. Low number of threads (one warp) exhibit low memory-latency hiding ability. In contrast, too large number of threads (128 and more) limits the number of active blocks per multiprocessor because of limited number of registers. According to our experience, the 64-thread-block is optimal in most cases.

The pre-fetching technique helps to hide the memory latency but consume additional registers. Therefore, it performs better on cards with higher register count per stream processor. From our experience, Fermi-based cards obtain better results without pre-fetching technique.

The Fermi-based cards also suffer from lower memory bandwidth to computing power ratio. It means that total kernel performance is limited by memory bandwidth even if the memory latency is hidden well. This limit can be elicited by increased data reuse. We have implemented a 16-vectors kernel version. This kernel computes 16 feature-vectors at once so one phase of model parameters reading is eliminated. The 16-vector kernel outperforms the 8-vector kernel by 15–20 % on Fermi-based cards. The performance does not change on other cards.

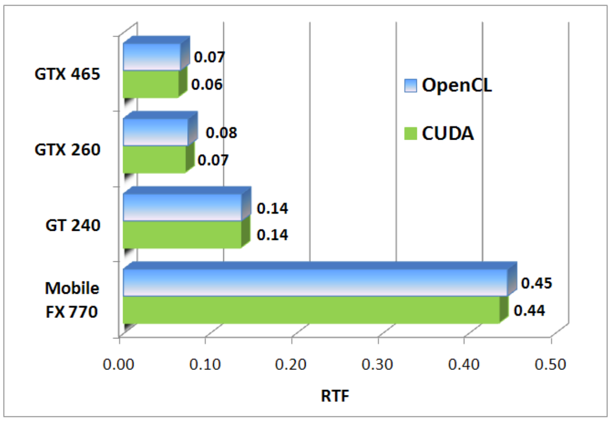

Figure 1: *Real-time factors (RTFs) of our optimized implementation for several NVIDIA GPUs with CUDA and OpenCL implementations. Tested on model with 1,280k Gaussians in total. RTF calculation is based on 100 vectors per second rate.*

5. Results

In addition to analysis of the described implementations within the speech decoder, we measured the speed of AM likelihoods evaluation only. For performance comparison, we use RTF measure which is a ratio between elapsed time and length of processed speech. Elapsed time includes the host-device memory transfers.

At first, we tested CUDA as well as OpenCL implementation with several GPUs available to us. We've choosen a very large model together with a large feature-vector window to suppress the CPU-GPU communication overhead during the implementation performance evaluation. We have chosen a 5000-states model with 256 Gaussians per state (i.e. $\approx$ 1.3M Gaussians in total) and feature vector dimension 36. The feature-vector window length was 256. RTFs were calculated on 100 vectors per second basis. Results are shown in Fig. 1.

The measured RTFs suggest that even a laptop GPU is able to process this very large model in less than a half of the real-time. Desktop models are much faster and achieved elapsed times from 7 to 50 times shorter than real-time. Results indicate that practically any GMM-based acoustic model can be used in real-time applications even with a low-end office PC GPU or even a laptop GPU. Also, the offline recognizers speed-up can be significant, if the decoder part will be powerful enough. The results also shows that OpenCL implementation is a little slower for this high window-size than CUDA.

We compared performance of CUDA and OpenCL implementations in more detail for various feature-vector lengths on GeForce GT 240 GPU. Results are shown in Fig. 2. Six window sizes in range from 8 to 256 were tested on a smaller acoustic model with 16 Gaussians per state and 5000 states. The results shows us 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 varies between 0.3 and 1 millisecond per kernel run. In the case of real-time speech recognition, the overhead is not a major problem anyway as the bottleneck in this case is the decoder part.

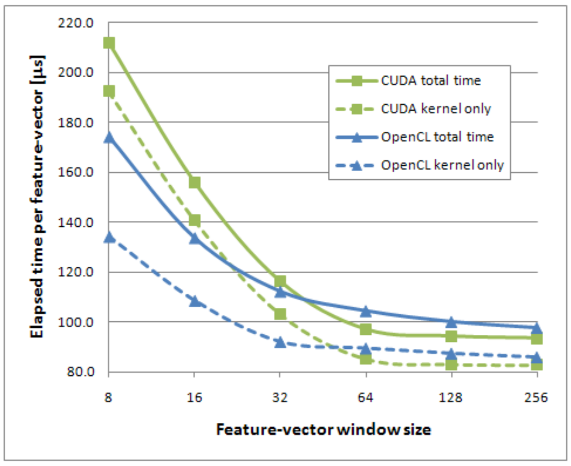

Figure 2: *Elapsed time per feature-vector on GT 240 GPU for various vector-window sizes. Both CUDA and OpenCL implementations were tested and total as well as kernel-only elapsed times were measured.*

5.1. Comparison of the Existing Implementations

Direct comparison of the aforementioned implementations is problematic. Each implementation was developed and benchmarked on different devices, different model complexities and different feature vector sizes, etc. The implementation [1] evaluates only one feature vector at a time, therefore the performance is hampered by communication overhead. The implementation [3] was intended primarily for completely different task. The fastest competing implementation ([8], [2]) achieves $\mathrm{RTF} = 0.15$, whereas our implementation benchmarked on similar device and with about $1.5\times$ larger acoustic model achieves $\mathrm{RTF} = 0.08$.

5.2. Real-life task LVCSR experiments

We have done also experiments with real recognizer. The recognizer was designed for both off-line as well as real-time applications. For evaluation, we used data that are used for automatic captioning of parliamentary sessions. The task itself as well as the LVCSR system setup is described in [9].

For our experiments we used acoustic models with 81k and 194k Gaussians (5k states, each consisting of 16 or 36 Gaussians). The smaller model is the largest one that, when evaluated on CPU using a fast Gaussian prunning algorithm, fits into the real-time constraints. When no prunning method is used, its RTF is about 2. The large model performs about 1 % abs. better (approx. 91 % vs. 92 %), however its RTF is about 5.

In our experiment, we tested the both acoustic models. The experiments were performed on Intel Core2 Quad 2.83GHz CPU together with GTX 260 GPU. The decoder part of the recognizer uses all four cores of the CPU. Using the GPU during the recognition, we were able to fit within the RTF constraints even with the big model.

Therefore, employing of GPU opens two sources of accuracy improvement in real-time systems. The first source is the full acoustic model processing without need of any pruning or approximations. The second source is the possibility to use of much larger models. In many speech recognition tasks, it is possible now to process in real-time bigger model with the aid of GPU than we are able to train reliably.

6. Conclusion

In this paper we described our GPU implementation of acoustic model likelihoods computation that achieves close to peak performance on tested GPUs and is significantly faster than the previously published implementations. We presented CUDA and OpenCL implementations optimized for NVIDIA GPUs and we compared them to each other. The results of tests with the recognizer suggest that during speech recognition it is now possible to use as large acoustic models as can be reliably trained. Also fusion techniques together with evaluation of a large set of models are now possible even in real-time recognition.

7. Acknowledgements

This research was supported by the Ministry of Education of the Czech Republic, project No. 2C06020, by the Technology Agency of the Czech Republic, project No. TA01011264 and by the University of West Bohemia, project No. SGS-2010-054. The access to the MetaCentrum computing facilities, provided under the programme LM2010005 funded by the Ministry of Education, Youth, and Sports of the Czech Republic, is highly appreciated.

8. References

[1] P. Cardinal, P. Dumouchel, G. Boulianne, and M. Comeau, “GPU accelerated acoustic likelihood computations,” in *Proceedings of Interspeech 2008*. Causal Production, Ltd., 23–26 September 2008, pp. 964–967.

[2] 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, no. 4, pp. 510 – 526, 2009.

[3] P. Kveton and M. Novak, “Accelerating hierarchical acoustic likelihood computation on graphics processors,” in *Proceedings of Interspeech 2010*. Causal Production, Ltd., 26–30 September 2010, pp. 350–353.

[4] J. Vaněk and J. Psutka, “Gender-dependent acoustic models fusion developed for automatic subtitling of parliament meetings broadcasted by the Czech TV,” in *Text, Speech and Dialogue*. Springer Berlin / Heidelberg, 2010.

[5] (2010, August) The CUDA C best practices guide, version 3.2. NVIDIA Corporation. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf

[6] P. R. Dixon, D. A. Caseiro, T. Oonishi, and S. Furui, “The Titech large vocabulary WFST speech recognition system,” in *Proc. ASRU Automatic Speech Recognition & Understanding IEEE Workshop*, 2007, pp. 443–448.

[7] V. Volkov and J. W. Demmel, “Benchmarking GPUs to tune dense linear algebra,” in *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis SC 2008*, 2008, pp. 1–11.

[8] P. R. Dixon, T. Oonishi, and S. Furui, “Fast acoustic computations using graphics processors,” in *Proc. IEEE Int. Conf. Acoustics, Speech and Signal Processing ICASSP 2009*, 2009, pp. 4321–4324.

[9] A. Pražák, J. V. Psutka, J. Hoidekr, J. Kanis, L. Müller, and J. Psutka, “Automatic online subtitling of the czech parliament meetings,” in *Text, Speech and Dialogue*. Springer Berlin / Heidelberg, 2006.