GPU Accelerated Acoustic Likelihood Computations

Patrick Cardinal[1,2], *Pierre Dumouchel*[1,2], *Gilles Boulianne*[1], *Michel Comeau*[1]

[1]Centre de Recherche Informatique de Montréal (CRIM),Montréal,Canada
[2]École de Technologie Supérieure, Montréal,Canada
{patrick.cardinal,pierre.dumouchel,gilles.boulianne,michel.commeau}@crim.ca

Abstract

This paper introduces the use of Graphics Processors Unit (GPU) for computing acoustic likelihoods in a speech recognition system. In addition to their high availability, GPUs provide high computing performance at low cost. We have used a NVidia GeForce 8800GTX programmed with the CUDA (Compute Unified Device Architecture) which shows the GPU as a parallel coprocessor. The acoustic likelihoods are computed as dot products, operations for which GPUs are highly efficient. The implementation in our speech recognition system shows that GPU is 5x faster than the CPU SSE-based implementation. This improvement led to a speed up of 35% on a large vocabulary task.

Index Terms: Speech recognition, GPU

1. Introduction

Large vocabulary automatic speech-recognition is a computationally intensive task. Nevertheless, quick response times are required in real-world applications. To outperform the speeds offered by modern CPUs, one can turn to alternate processors specialized in parallel computations.

Modern graphic cards incorporate a specialized processor called Graphics Processing Unit (GPU). A GPU is mainly a Single Instruction, Multiple Data (SIMD) parallel processor that is computationally powerful, while being quite affordable. Over the years, the GPU has evolved into a flexible processor.

A noteworthy technological advance was achieved in 2007, when NVidia and ATI introduced the unified architecture which eliminated the graphical pipeline. This greatly enhanced the flexibility and usability of the GPU, to the extent that it is becoming a mainstream alternative for general purpose calculations.

The technology used in GPUs is the 90 nm transistor, as compared to the 45 nm transistor now used in Intel Core 2 processors. However, the latter have more transistors dedicated to memory caching and branch prediction, while GPU transistors are mainly used for building arithmetic units. Consequently, a GPU such as the NVidia GeForce 8800GTX can run at more than 300 GFLOPS while an Intel Core2 Duo performance is limited to 30 GFLOPS [1].

These capabilities could be exploited in order to improve the performance of a speech recognition system. This paper explores how acoustic likelihood computations can be implemented in a GPU. The paper is organized as follows. In section 2, we first make a small wrap-up of applications that take advantage of the GPU power. In the third section, we describe how acoustic computations have been implemented in the GPU while the fourth section shows the experimental results. We conclude by discussing what comes next in our GPU implementation of a speech recognition system.

2. Related Work

In their experimentation of using external hardware for improving a speech recognition system, Nedveschi and al. [2] implemented a 30 word system for recognizing numbers in a FPGA or ASIC. Their implementation was shown to be very efficient in terms of energy consumption and gave results similar to software implementation. Lin and al. [3, 4] implemented a 1000-word speech recognition system in a FPGA that was 7x faster than their software implementation (SPHINX) and resulted in a real-time speech recognizer. However, we have not found any related work applied to large vocabulary speech recognition.

The use of GPU has been explored for feature extraction by Bremer and al. [5]. The implementation showed a speed-up of 7.1x over the software implementation. To the best of our knowledge, [5] is the only work integrating GPU in speech recognition. However, GPUs have been used to improve computations in a wide variety of fields more or less related to graphical applications.

In computer vision, the GPU has been used by Fung[6] to accelerate signal processing algorithms such as blurring, low-pass filtering and downsampling. They obtain a speed-up of 3.5x over a CPU. In image processing, Erra [7] implemented fractal image compression algorithms on a NVidia GeForce FX 6800 and obtained a speed-up of 280x over the equivalent CPU-based algorithm. In computational geometry, GPU has been successfully used in distance fields, collision detection, transparency computation and particle tracing. These, and many other applications implemented on GPUs are discussed in [8].

In signal processing, the FFT has been efficiently implemented in a GPU considering that the FFT is clearly memory bound. An openGL implementation, described in [9], yields a gain of 1.9x. More recently, a RapidMind[10] implementation showed a gain of 2.7x over a highly optimized CPU implementation running on the fastest CPU at the time [11].

These results motivated the use of GPU in our speech recognition system.

3. Implementation

A GPU is a SIMD parallel processor that is specialized in graphical rendering, which involves a high rate of linear algebra operations. Thus, using linear algebra-based acoustic likelihood calculations should lead to a more efficient use of the GPU hardware. The acoustic likelihood for a Gaussian mixture model is defined as :

$$b_j(\vec{o}_t) = \sum_{c=1}^{C_j} \alpha_{jc} \frac{1}{\sqrt{(2\pi)^d |\Sigma_{jc}|}} e^{-\frac{1}{2}(\vec{o}_t - \vec{\mu}_{jc})' \Sigma_{jc}^{-1} (\vec{o}_t - \vec{\mu}_{jc})}$$

Copyright © 2008 ISCA

where $b_j(\vec{o}_t)$ is the probability that distribution j generates the d-dimensional observation vector $\vec{o}$ at time t, C_j is the number of Gaussians in the distribution j, α_{jc} is the weight of Gaussian c in distribution j, μ_{jc} and Σ_{jc} are the mean vector and the covariance matrix of Gaussian c in distribution j. The natural logarithm likelihood of one Gaussian can be expressed as:

$$\begin{aligned} b_{jc}(\vec{o}_t) &= \ln \alpha_{jc} - \frac{1}{2}\ln((2\pi)^d|\Sigma_{jc}|) - \frac{1}{2}\vec{\mu}'_{jc}\Sigma_{jc}^{-1}\vec{\mu}_{jc} + \\ & \vec{\mu}'_{jc}\Sigma_{jc}^{-1}\vec{o}_t - \frac{1}{2}\vec{o}'_t\Sigma_{jc}^{-1}\vec{o}_t \end{aligned}$$

The first three terms are independent of the observations and can be considered a Gaussian-specific constant that can readily be pre-computed. Denoting this term by h_{jc}, it is:

$$h_{jc} = \ln \alpha_{jc} - \frac{1}{2}\ln((2\pi)^d|\Sigma_{jc}|) - \frac{1}{2}\vec{\mu}'_{jc}\Sigma_{jc}^{-1}\vec{\mu}_{jc}$$

Excluding the observations, the last two terms can be denoted by:

$$\begin{aligned} u_{jc} &= \vec{\mu}'_{jc} \cdot \Sigma_{jc}^{-1} \\ v_{jc} &= Diag(-\frac{1}{2}\Sigma_{jc}^{-1}) \end{aligned}$$

where Σ_{jc}^{-1} is the diagonal covariance matrix. The likelihood for a single Gaussian can thus be expressed as:

$$b_{jc}(\vec{o}_t) = (h_{jc} + u_{jc}\vec{o}_t + v'_{jc}\vec{o}_t^2)$$

This computation can be accomplished by a dot-product of the following two vectors in which subscripts designating the distribution component have been omitted for clarity:

$$\begin{aligned} \vec{obs} &= (\bar{1}, o_1, o_2, \cdots, o_n, o_1^2, o_2^2, \cdots, o_n^2) \\ \vec{M} &= (h, \mu_1\sigma_{11}^{-1}, \cdots, \mu_n\sigma_{nn}, -\frac{1}{2}\sigma_{11}^{-1}, \cdots, -\frac{1}{2}\sigma_{nn}^{-1}) \end{aligned}$$

where $\bar{1}$ is the identity element of multiplication. The likelihood of a distribution is defined as :

$$\ln b_j(\vec{o}_t) = \bigoplus_{c=1}^{C_j}(\vec{obs} \cdot \vec{M}_{jc})$$

where $\bigoplus$ is the logarithmic addition and is defined as $\ln(e^x + e^y)$. In this form, the computation of acoustic probabilities is perfectly suitable for a GPU since each distribution can be independently computed in parallel, and the results rest upon basic dot product operations.

3.1. CUDA development framework

We have implemented the acoustic computation module in CUDA, a development framework for NVidia graphic cards[12]. The CUDA framework shows the graphic card as a parallel coprocessor for the CPU. The development language is C with some extensions.

A program in the GPU is called a kernel and many of them can be concurrently launched. A kernel is made up of configurable amounts of blocks, each of which consists in a configurable amount of threads as shown in Figure 1. Built-in variables allow the programmer to know which thread of which block is currently executed.

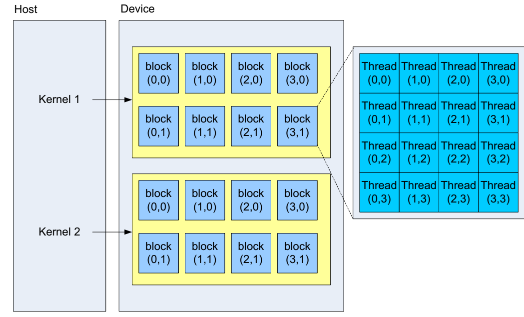

Figure 1: *Overview of CUDA thread batching. [1]*

At execution time, each block is assigned to a multiprocessor. More than one block can be assigned to a given multiprocessor. Blocks are divided in groups of 32 threads called warps. In a given multiprocessor, 16 threads (half-warp) are executed at the same time. A time slicing-based scheduler switches between warps to maximize the use of available resources.

There are two kinds of memory. The first is the global memory which is accessible by all multiprocessors. Since this memory is not cached, it is important to ensure that the read/write memory accesses by a half-warp are coalesced in order to improve the performance. The texture memory is a small part of the global memory which is cached. The texture memory can be efficient when there is locality in data.

The second kind of memory is the shared memory which is internal to multiprocessors and is shared within a block. This memory, which is a lot faster than the global memory, can be seen as user-managed cache. This memory is divided in banks in such a way that succesive 32-bit words are in successive banks. To be efficient, it is important to avoid conflicting accesses between threads. Conflicts are resolved by serializing accesses which lead to a performance drop proportional to the number of serialized accesses.

3.2. Reduction algorithm

The reduction algorithm is an important building block in parallel computing. This algorithm involves a reduction operator which takes two or more arguments and returns some combination of them. The addition and the maximum operators are such operators. The reduction operator is iteratively applied until only one element remains. Figure 2 shows an example of reduction using the maximum operator.

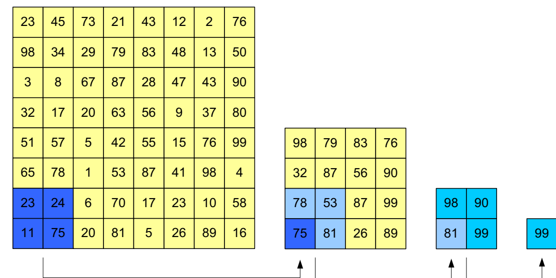

Figure 2: *Reduction algorithm [13].*

This operation can be implemented very efficiently on a GPU since all reductions are independent and can thus be executed in parallel.

3.3. Kernel for acoustic calculation

As described before, the likelihood of a given mixture is the logarithmic addition of dot-products for each component of the mixture. This operation can be implemented as a reduction algorithm which uses the addition as reduction operator, excepted for the C_j last operation for which the logarithmic addition is used to complete the reduction.

In our implementation, the computation of a mixture likelihood is computed by one block of threads. Consequently, the number of launched blocks is the number of distributions in the acoustic model. Each block contains 256 threads.

For efficiency, the observation vector $\vec{obs}$ is copied C_j times. As a result, it is the same length as a distribution vector. There is thus a direct correspondence between its elements and those of $\vec{M}$, thus avoiding index calculations.

Moreover, to ensure efficiency of the reduction process and coalescing access to the global memory during the reduction process, the model vector $\vec{M}$ is reorganized at the distribution level. It's organized such that the C_j firsts elements are the constants, followed by the $\mu_1\sigma_{11}^{-1}$ of each component and so on. Figure 3 shows an example of the vector layout for a 2 dimension, 2-Gaussian model. In this figure, u_{xc} and v_{xc} denote $\mu_x\sigma_{xx}^{-1}$ and $-\frac{1}{2}\sigma_{xx}$ of component c.

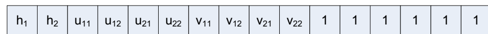

Figure 3: *Example of model vector.*

The observation vector is reorganized in the same way to ensure consistency.

Algorithm 1 shows the implementation of the cross-product in the GPU for the case of a 64 component mixture. The DISTSIZE value is a template parameter and denotes the dimension of a distribution in number of floats. This value must be a power of 2. This algorithm uses optimizations specific to the CUDA architecture shown in [14].

The algorithm works as follows. The shared array declared at line 2 contains the results of the sucessive reduction. This array can be seen as a user managed cache. The baseIndex variable contains the position of the distribution according to the block id. Recall that each block computes the likelihood of one distribution. The block at index 1 works on distribution 1, the block at index 2 on distribution 2, and so on.

The nested loops at lines 5-11 compute all multiplications and perform the first reductions to reduce the data size at 512 elements. The function $syncthreads()$ at line 12 ensures that all the intermediate computations are completed. The rest of the algorithm, lines 13-28, completes the reduction process with the exception that the last 32 steps use the logarithmic addition as reduction operator.

This section could be implemented with a simple loop. However, as the reduction progresses, the number of required threads decreases. In a loop implementation, many threads will just pass through the loop without doing any operations. By using an unrolled implementation, these threads become available for other blocks running in the multiprocessor.

The last thing the algorithm does, at lines 29-30, is to save the reduction result in the results array at the right position according to the block index.

The LogAdd function implements the logarithmic addition. Our implementation is an approximation to avoid the computation of the two exponentials. The same algorithm is used in both the CPU and GPU implementations.

Algorithm 1 Kernel for acoustic calculation

gpuDotProduct(Obs,M,Results)

1: $tid \leftarrow threadIdx.x$
2: __*shared*__ float aux[512]
3: $baseIndex \leftarrow blockIdx.x * DISTSIZE$
4: $x \leftarrow tid$
5: **while** $x < 512$ **do**
6: $\quad aux[x] = M[baseIndex + x] * Obs[x]$
7: $\quad i \leftarrow tid$
8: $\quad$ **while** $i < DISTSIZE$ **do**
9: $\quad\quad aux[x] += (M[baseIndex + x + i] * Obs[x + i])$
10: $\quad\quad i \leftarrow i + 512$
11: $\quad x \leftarrow x + blockDim.x$
12: syncThreads()
13: **if** $tid < 256$ **then**
14: $\quad aux[x] \leftarrow aux[x] + aux[x + 256]$
15: syncThreads()
16: **if** $tid < 128$ **then**
17: $\quad aux[x] \leftarrow aux[x] + aux[x + 128]$
18: syncThreads()
19: **if** $tid < 64$ **then**
20: $\quad aux[x] \leftarrow aux[x] + aux[x + 64]$
21: syncThreads()
22: **if** $tid < 32$ **then**
23: $\quad aux[x] \leftarrow LogAdd(aux[x], aux[x + 32])$
24: $\quad aux[x] \leftarrow LogAdd(aux[x], aux[x + 16])$
25: $\quad aux[x] \leftarrow LogAdd(aux[x], aux[x + 8])$
26: $\quad aux[x] \leftarrow LogAdd(aux[x], aux[x + 4])$
27: $\quad aux[x] \leftarrow LogAdd(aux[x], aux[x + 2])$
28: $\quad aux[x] \leftarrow LogAdd(aux[x], aux[x + 1])$
29: **if** $tid = 0$ **then**
30: $\quad Results[blockIdx.x] \leftarrow aux[0]$

4. Experimental Results

In this section, we describe two experiments that were conducted in order to evaluate GPU performance in a context of large vocabulary speech recognition.

4.1. Experimental setup

Experiments have been conducted with a FST-based speech recognition system developped at CRIM and tuned for speaker independent transcriptions of broadcast news.

The acoustic model has been trained with 171 hours coming from French television programs in Quebec. The programs are a mix of weather, news, talk shows, etc. which have been transcribed manually at CRIM. The acoustic parameters consist in 12 MFCCs, the energy; the corresponding first and second derivatives, for a total of 39 features. The model contains 4600 distributions with diagonal covariance matrices. The test set is made up of 4 hours of similar audio.

The language model has been trained with text from a local newspaper (La Presse, 93 million words) and the acoustic training set's textual transcripts (2.1 million words). The vocabulary size is 59624 words.

The CPU implementation of acoustic calculations uses the SSE registers and runs on a Intel Core 2 duo at 2.66GHz. However, only one core is used. Required probabilities are computed on-demand, which amount to approximately 40% of all probabilities.

The GPU used is the NVidia GeForce 8800GTX which contains 8 multiprocessors for a total of 128 stream processors and

Number of Gaussians	time (ms) CPU	GPU	speed-up
32	12.39	2.37	5.22
64	22.35	4.39	5.09
128	38.82	8.55	4.54

Table 1: *Speedup in acoustic likelihood calculation.*

Number of Gaussians	time (ms/frame) CPU	GPU	Speed up (%)	WER (%)
32	14.74	10.61	28%	35.55%
64	19.00	12.63	33.5%	34.94%
128	25.12	16.78	33.2%	33.94%

Table 2: *Speed-up in a large vocabulary ASR system.*

has 768MB of RAM. The implementation of GPU functions in our speech recognition system is straightforward. Computing likelihood on-demand is very costly in the GPU which works better on big sets of data. Consequently, it is more efficient to compute all acoustic likelihoods at each frame. The current implementation involves a busy waiting since the CPU does nothing useful while the GPU works on likelihood computations and vice-versa.

4.2. Acoustic computations

In the first experiment, we compared performances of acoustic computations only. The task was to compute the likelihood for all the distributions of our acoustic model. The process was repeated 2000 times to ascertain average CPU and GPU times.

We have experimented with 32, 64 and 128 Gaussians per mixture. The results in Table 1 show that the GPU is 5x faster than the optimized CPU implementation. However, the results show that the speed-up decreases when the number of Gaussians increase. We think that this behavior is caused by the fact that the transfer of observations is not hidden by other computations, as that will be the case in a real application. However these results could be improved by having a power of 2 for the parameter vector length. In our case, we have 39 parameters. Consequently, some padding is necessary to complete the vector. This leads to a calculation overhead of 39%. By setting a better parameter vector size or dropping some features, the anticipated speed-up could reach 7x.

4.3. Large vocabulary speech recognition

We also studied the speed-up impact of GPU-integration of a large vocabulary speech recognition system. Table 2 shows the total computation time per frame for both the CPU and GPU implementations and the WER obtained with each model.

The results show a significant gain in recognition speed for this application. However, we had originally expected a greater speed-up in going from 64 to 128 Gaussians. This unexpected behavior may be traced back to the increased accuracy of the 128-Gaussian models, resulting in a smaller number of active states at each frame. This situation advantages the CPU implementation. In spite of that, the GPU still outperforms the latter.

Note that the use of a 128 Gaussian model is barely 13.8% slower than the 32 Gaussian CPU implementation but leads to an improvement of 1.61% in word error rate.

5. Discussion and Future work

We have shown that the use of a GPU for acoustic calculations can speed up the speech recognition system by 33.5%. These results can yet be improved by optimizing the feature vector length. Another room for improvement is to increase the parallelism by ensuring that the CPU performs search-related tasks while the GPU computes the acoustic likelihoods. For example, a set of observations can be sent to the GPU at a given time interval. During the time taken for these calculations to complete, the CPU can work on the previously calculated frames. Moreover, this approach will also decrease the overhead of memory transfer since larger memory chunks will be transfered.

The next step will be to implement the Viterbi decoder in the GPU, which is achievable since different states at time t can be expanded independently.

6. Acknowledgements

This project is made possible with the support of the Canadian Heritage New Media Research Networks Fund, which promotes innovation in new media or interactive digital content that pertain to the Canadian cultural sector.

7. References

[1] NVidia, *NVidia CUDA Compute Unified Device Architecture: Programming Guide*, 2007.

[2] S. Nedevschi, R. Patra, and E. Brewer, "Hardware Speech Recognition for User Interfaces in Low Cost, Low Power Devices," *Design Automaton Conference (DAC)*, 2005.

[3] E. Lin, K. Yu, R. A. Rutenbar, and T. Chen, "Moving Speech recognition from Software to Silicon: the In Silico Vox Project," *Interspeech*, 2006.

[4] E. Lin, K. Yu, R. A. Rutenbar, and T. Chen, "A 1000-Word Vocabulary, Speaker Independent, Continuous Live-Mode Speech Recognizer Implemented in a Single FPGA," *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2007.

[5] D. Bremer, J. Johnson, H. Jones, Y. Liu, D. May, J. Meredith, and S. Veydia, "Application Kernels on Graphics Processing Units," in *Workshop on High Performance Embedded Computing*, 2005.

[6] J. Fung and S. Mann, "Computer Vision Signal Processing on Graphics Processing units," in *IEEE International Conference on Acoustics, Speech and Signal Processing*, 2004.

[7] U. Erra, "Toward Real-Time Fractal Image Compression Using Graphics Hardware," in *International Symposium on Visual Computing*, vol. 3804 of Lecture Notes in Computer Science. Springler-Verlag, 2005.

[8] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. Lefhn, and T. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum*, vol. 26, no. 1, 2007.

[9] T. Sumanaweera and D. Liu, "Medical Image Reconstruction with the FFT," in *GPU Gems 2*. Addison Wesley, 2005.

[10] RapidMind, "http://www.rapidmind.net."

[11] M. McCool, K. Wadleigh, B. Henderson, and H. Lin, "Performance Evaluation of GPUs Using the RapidMind Development Platform," in *Proceedings of the 2006 ACM/IEEE conference on supercomputing*, 2006.

[12] CUDA, "http://www.nvidia.com/object/cuda_home.html."

[13] M. Harris, "Mapping Computational Concepts to GPUs," in *GPU Gems 2*. Addison Wesley, 2005.

[14] M. Harris, "Optimizing parallel reduction in CUDA," NVidia, Tech. Rep., 2007. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/1_1/Website/Data-Parallel_Algorithms.html