Asociación Argentina
de Mecánica Computacional

Mecánica Computacional Vol XXX, págs. 3033-3044 (artículo completo)
Oscar Möller, Javier W. Signorelli, Mario A. Storti (Eds.)
Rosario, Argentina, 1-4 Noviembre 2011

CONSIDERING PURE GPU MODEL FOR AN AUDIO FINGERPRINTING SYSTEM

Natalia Miranda[a], Fabiana Piccoli[a] and Edgar Chávez[b]

[a] *Universidad Nacional de San Luis, Ejército de los Andes 950, 5700 - San Luis - Argentina*

[b] *U. Michoacana México and CICESE, México*

e-mail: {mpiccoli}@unsl.edu.ar

Abstract. The demand for protecting, managing and indexing digital audio is growing quickly. As a viable solution for this, fingerprinting is receiving increased attention. An audio fingerprinting system extracts feature vectors (called fingerprint) from a query audio, finds matching in a database (DB), and retrieves the appropriate audio signals associated with the matching fingerprint in the DB.

An audio fingerprint is a compact low level content-based digest of an audio signal. It provides the ability to identify short, unlabeled audio signals in a fast and reliable way. There are several practical requirements which a successful audio fingerprinting system should satisfy. First, it should be able to identify corrupted audio signals in spite of degradations. Second, it should be able to identify the signals of only a few seconds long. Finally, it should be computationally efficient, both in calculating of the fingerprints and in searching for the best match in the DB. Besides, an audio fingerprinting system should be scalable, i. e., it has to operate well with very large DBs. A good option is to apply high performance techniques in the solution.

The Graphics Processing Unit (GPU) provides high performance computing through the threading model. Its main characteristics are high computational power, constant development and low cost and provides a kit of programming called CUDA. It provides a GPU-CPU interface, thread synchronization, data types, among others.

CUDA supports several types of memory that can be used to achieve high execution speeds in applications. The global memory is large but slow and tends to have long access latencies and finite access bandwidth, whereas the shared memory is on-chip memory, small and fast. The variables that reside in this type of memory can be accessed at very high speed in a highly parallel manner. Other memories are constant and texture memory which are read-only.

In this work, we propose implement the whole audio fingerprint system in a pure GPU model, using all properties offered by GPU: shared memory, constant memory, atomic functions, coalescing access, among others. We show different optimizations through the use of CUDA memory hierarchy. We achieve to reduce the total number of accesses to the global memory using shared memory and to improve considerably the performance. Finally, the experimental results are presented.

Copyright © 2011 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

1 INTRODUCTION

Audio indexing and audio identification has received a lot of attention in the last years. Specially the last, that consists in the ability to pair audio signals of the same perceptual nature. The audio identification demands an stable and persistent object representation, it is the same although the audio suffers different natural degradation. Such representation is called an Audio fingerprint (AFP).

A fingerprinting system basically consists of two parts: fingerprint extraction and an algorithm to search for matching in a fingerprint database. In this work, we focus on the first part only and we propose an AFP implementation applying high performance techniques.

An AFP is a compact representation of the perceptually relevant parts of audio content, which can be used to identify an audio (or a segment) file, even it is severely degraded due to compression or other types of signal processing operations. The fingerprints of a large number of audio signals are usually stored in a large database. An audio signal can be identified by comparing the fingerprint of query with each fingerprint in the database.

Well-known applications of AFP are broadcast monitoring, connected audio and filtering for file sharing applications Ibarrola and Chavez (2010), Haitsma and Kalker (2002), Shin et al. (2002), Wang (2003). The use of fingerprints has several advantages. First, the dataset to compare is relatively small, because fingerprints are compact descriptions similar to hash functions. Second, comparing fingerprints can be done efficiently, because the perceptually irrelevant parts have been removed. One audio file, encoded using different coding schemes, gives the same fingerprint or very similar. Fingerprints from two arbitrary selected pieces of audio signal are very different.

In order to employ high performance computing to speedup the process of obtaining the AFP, the Graphics Processing Unit (GPU) represents a good alternative. The GPU is attractive in many application areas by its characteristics, especially their parallel execution capabilities and fast memory access. They promise more than an order of magnitude speedup over conventional processors for some non-graphics computations. The use of GPUs in general-purpose computing is becoming a very accepted alternative.

A GPU computing system consists of two basic components, the traditional CPU and one or more GPUs (Streaming Processor Array). The connection between CPU and GPU is by mean of PCI Express bus. The GPU can be considered as a manycores coprocessor ables to support fine grain parallelism (a lot of threads run in parallel, all of them collaborate in the solution of the same problem) GPU is different than other parallel architectures because it shows flexibility in the local resources allocation (memory or register) to the threads. In general, a GPU multiprocessor consists of several streams, each of them has multiple processing units, records and on-chip memory. Each stream multiprocessor can run a variable number of threads. The programmer decides how many threads and how they will work. These can be adjusted to achieve improvements in the system performance.

Each GPU applies the Single Process-Multiple Data (SPMD) model, all units of computation (thread) running the same code, not necessarily synchronously, over different data. Every thread shares the global memory space.

The CUDA programming model proposes a model for GPU programming. It has two main characteristics: the parallel work through concurrent threads and the memory hierarchy. A CUDA program is written in standard C/C++ extended by several keywords and constructs. The user supplies a single source program encompassing both host (CPU) and *kernel* (GPU) code. Each CUDA program consists of multiple phases that are executed on either the CPU or

Copyright © 2011 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

the GPU. All phases that exhibit little or no data parallelism are implemented in the CPU. In opposition, if the phases present much data parallelism, they are implemented as *kernel* functions in the GPU. A *kernel* function defines the code to be executed by each threads launched in a parallel phase.

There are several restrictions on *kernel* functions, they cannot: be recursive neither have static variables declarations or a variable number of arguments. The communication between CPU and GPU is through API calls. *Kernel* code is initiated performing a function call.

Threads in the CUDA model are grouped into thread blocks. All threads in a block execute on one SM and communicate among them through the shared memory. Threads in different blocks can communicate through global memory. Besides shared and global memory, the threads have their local variables. Thread blocks form a grid. The number of grids, blocks per grid and threads per block are parameters fixed by the programmer. As they can affect the performance of the application, can be adjusted.

Respect of memory hierarchy, CUDA threads may access data from multiple memory spaces during their execution. Each thread has private local memory and each block has shared memory visible to all its threads. These memories have the same lifetime that the *kernel*. All threads have access to the same global memory and two additional read-only memory spaces: the constant and texture memory spaces. The constant and texture memory spaces are optimized for different memory usages. The global, constant and texture memory spaces are persistent across *kernel* launches by the same application. Each kind of memory has own access cost, the global memory accesses are the most expensive.

In previous work Miranda N. (2010), Miranda N. and A. (2010b), Miranda N. and A. (2010a), we presented an audio fingerprint that was implemented through mixed model based on CPU-GPU architecture, some task were solved in GPU and the others in CPU.Its performance was good despite this solution did not take advantages of all GPU resources. In this work, we propose implement the whole audio fingerprint system in a pure GPU model, using all properties offered by GPU: shared memory, atomic functions Sanders and Kandrot (2010), coalescing access, among others.

The paper is organized as follows: in sections 2, we explain the complete process of obtaining a sequential AFP. Next, we show all the process implemented in GPU. In section 4, we analysis the implementation and its results. Finally, the conclusions and future works are exposed.

2 SEQUENTIAL AUDIO FINGERPRINT PROCESS

The first task of an audio-fingerprinting system is to extract features from the signal. The audio signal is processed on a frame by frame basis, i.e. it is split into frames of equal size and the AFP process is applied to each them Ibarrola (2007). A frame of signal is a short segment of audio. The figure 1 shows AFP process. In the *Frame Normalization* stage, the stereo audio signals are converted to mono aural, an amplitude normalization is frequently used to make the AFP robust to changes in volume. When we split the signal, we considered an overlap of 50%, it ensures a slow variation of the extracted features.

Actually, there are systems that extract signal features directly in time domain as in Kurth and Scherzer (2003) where the sign of the time derivative of the signal was found to be robust to lossy compression and low-pass filtering. However, most systems extract signal features in the frequency domain using a variety of linear transforms such as the Discrete Cosine Transform, the Discrete Fourier Transform, the Modulation Frequency Transform Sukittanon and Atlas (2002) and some Discrete Wavelet Transforms like Haar's and Walsh-Hadamard's Subramanya et al. (1999). Therefore in the phase *Computation of FFT and Hanning*, the signal is transformed

Copyright © 2011 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

from time domain to frequency domain using the Fast Fourier Transform (FFT). Previuosly a window, Hanning window, is employed to reduce edge effects and emphasize the signal near the middle of the frame.

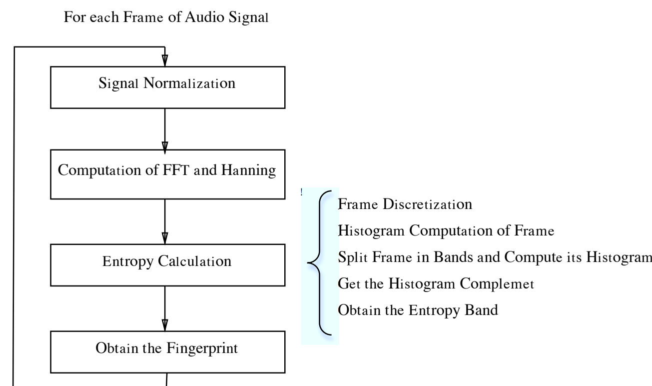

Figure 1: Sequential Audio Fingerprint Process

The *Entropy Calculation* stage includes many tasks. First, the frame is split in two parts, the real and imaginary parts then each part is transformed to a vector of discrete values, i. e. the continuous values are converted to discrete values. To represent each elements of a frame have been used 8 bits, hence each discrete data will take one of value between 0 and 255. After that the histogram of frame is calculated to obtain the estimation of Probability Density Function (PDF).

The histogram is a fast and simple method to estimate the entropy, it is a good method when the online determination of the PDF of an audio stream is done. In this case, the certainty of the histogram method is ensured by the fact that thousands of audio samples will be used at building the histogram.

Besides the histogram, there are other methods like parametric and non-parametric. The first methods are advisable when the distribution is known a priori and the amount of data involved is not large Bercher and Vignat (2000). For non-parametric methods, no assumptions are made about the distribution of the PDF belongs to. The PDF is shaped by the data that, in turn, are smoothed by some kernel. They are computationally expensive and so not frequently used for real time pattern recognition applications.

In the subtask *Split frame in bands and compute its histogram*, the frame is divided in bands according to the Bark scale Zwicker (1961). The Bark scale defines 25 critical bands, the first 24 corresponding to the bands of hearing. The last, 25, is discarded since only the youngest and healthiest ears are able to perceive. For any given band b, the elements of the frame corresponding to b are used to build two histograms, one for the real parts and another one for the imaginary parts of these elements. After that get the histogram complement (it is the difference between the frame histogram and band histogram), it is used to estimate the probability distribution function.

Finally, once the entropy of each band b is obtained (the entropy of b is the sum between the corresponding entropies of real and imaginary parts). Equation (1) states how the bit corresponding to band b and frame n of the AFP is determined using the entropy values of frames n

Copyright © 2011 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

and $n-1$. Only 3 bytes (i.e., 24 bits) are needed for each frame of audio signal.

$$F(n,b) = \begin{cases} 1 & if \ \ [h_b(n) - h_b(n-1)] > 0 \\ 0 & Otherwise \end{cases} \tag{1}$$

The AFP of signal is formed by many AFPs, each belongs to a frame. All of them are calculated in sequence. The next section, we introduce how to compute AFPs using GPU.

3 AUDIO FINGERPRINT PROCESS ON GPU

Figure 2 illustrates the parallel AFP process. The problem is particularly well suited for massive parallel processing. This joins with the GPU benefits: many-cores architecture, memory hierarchy and atomic functions, we implement a faster AFP process using GPU as a parallel computer.

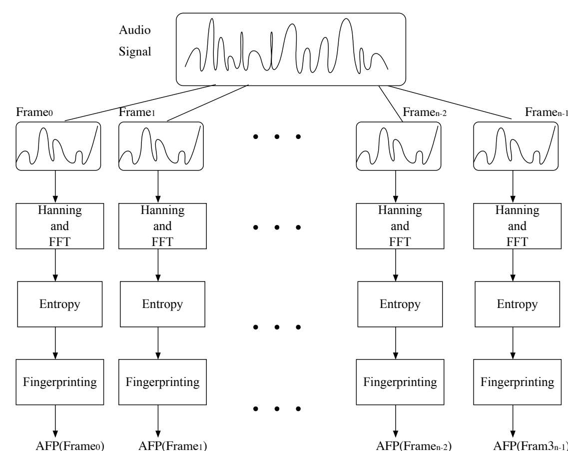

Figure 2: Parallel Audio Fingerprint Process

The GPU AFP process has three main stages, all of these are applied sequentially on every frame of signal. The audio signal is split into frames with fixed length of 16 KB (this size is equivalent to a frame duration of 370ms, which is adequate for entropy computation). Like in the sequential process, all frames are overlapped 50% to ensure a slow variation of the extracted features. Each frame is processed in parallel by a block of threads. A block calculates the AFP of a frame using up to 256 threads. If the audio signal has N frames then N blocks are launched in GPU to get together the AFP (The total number of threads is $N \times 256$). The output of GPU AFP process are $N-1$ vectors of twenty-four bits, the each vector is AFP of a frame.

The AFP process in GPU needs two data transfers between CPU and GPU

1. *From CPU to GPU*: At the beginning of AFP process, the CPU saves the data global memory of GPU. The data are the whole normalized audio signal and auxiliary data needed for different stages of process.

2. *From GPU to CPU*: This transfer is made at the end of AFP process.

The GPU AFP process is implemented through three *kernel*s, each of them corresponding to each stages in figure 2. The *kernels* are executed in sequence and no data movement is necessary between them. For two firsts kernels, there are N blocks each has 256 threads. In the last kernel, there are $N-1$ blocks of one thread each one.

In the next sections, we discuss each stages of the processing to obtain the AFP.

Copyright © 2011 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

3.1 Hanning and FFT Stage

To compute this stage, the computing is divided in two main phases: the first is Hanning window and Bit-reverse order. The second phase is the FFT calculus. Each phase has the next characteristics:

- *Hanning window and Bit-reverse order*: Each frame of audio signal is emphasized near the middle through the applying of Hanning window. This task reduces de edges effects. After, the emphasized frame is arranged according to bit-reverse vector. The bit-reverse vector indicates where the frame component will be placed (Each even index element in the first part of the frame is swapped with its corresponding even index element in the second part of the frame). All frames share the bit-reverse vector, in consequence, it is calculated in CPU and saved in GPU global memory for every threads can access it.

- *FFT*: In this second phase, the FFT computation takes place properly. We implemented the FFT algorithm based on the original algorithm of Cooley and Tukey Cooley and Tukey (1965). The inverse and direct FFT can be computed changing a single parameter. The sample is divided in two subsets of size half the original size, using the Danielson Lanczos theorem Danielson and Lanczos (1942). This process is repeated recursively or iteratively until the trivial problem (The problem cardinality is two). In this case, it is iterative because the CUDA does not allow the recursion.

  In this phase, the GPU needs other auxiliary data: the weight vector. It has same characteristic of bit-reverse vector, it is the same for all frame. Then it is calculated previously by CPU and saved to global memory of GPU.

For each frame, a block of threads solves this stage. We fixed 256 threads to be executed in parallel. As the number of threads is smaller than the vector size (16KB=16384 components), each thread will work on a fraction of the data, it has 64 components (16384/256).

The output of FFT is the same signal frame but in the frequency domain. It is a vector of complex number. The next steps of AFP work with two vectors, the vector of real components and vector of imaginary components.

3.2 Entropy Stage

As we said in section 2, the entropy based in histograms is a good choice, besides the entropy of a signal is a measure of the amount of information that the signal carries Ibarrola (2007). The Shannon's entropy Shannon and Weaver (1949) is a good candidate to identify a signal through an unique value. Small perturbations on the sample values of X produce smaller perturbations on the measured entropy. If the sample values of X are denoted by $\{x_i\}$ then entropy is defined as

$$H(X) = -\sum_{i=0}^{255} p(x_i) ln(p(x_i)) \tag{2}$$

where $p(x_i)$ is the probability that the signal takes value x_i. It is computed using Laplace's formula $p(x_i) = \frac{f(x_i)}{N}$. $f(x_i)$ is the number of times that value x_i occurs in the sequence X and N is the frame size.

As the entropy stage implies many tasks, we call them phases. In order, the phases are: Translation to Discrete Vector, Final Band Histogram and Band Entropy. All phases are applied

Copyright © 2011 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

to two vectors: vector of real components and vector of imaginary components. The figure 3 shows the whole entropy stage and the all its phases.

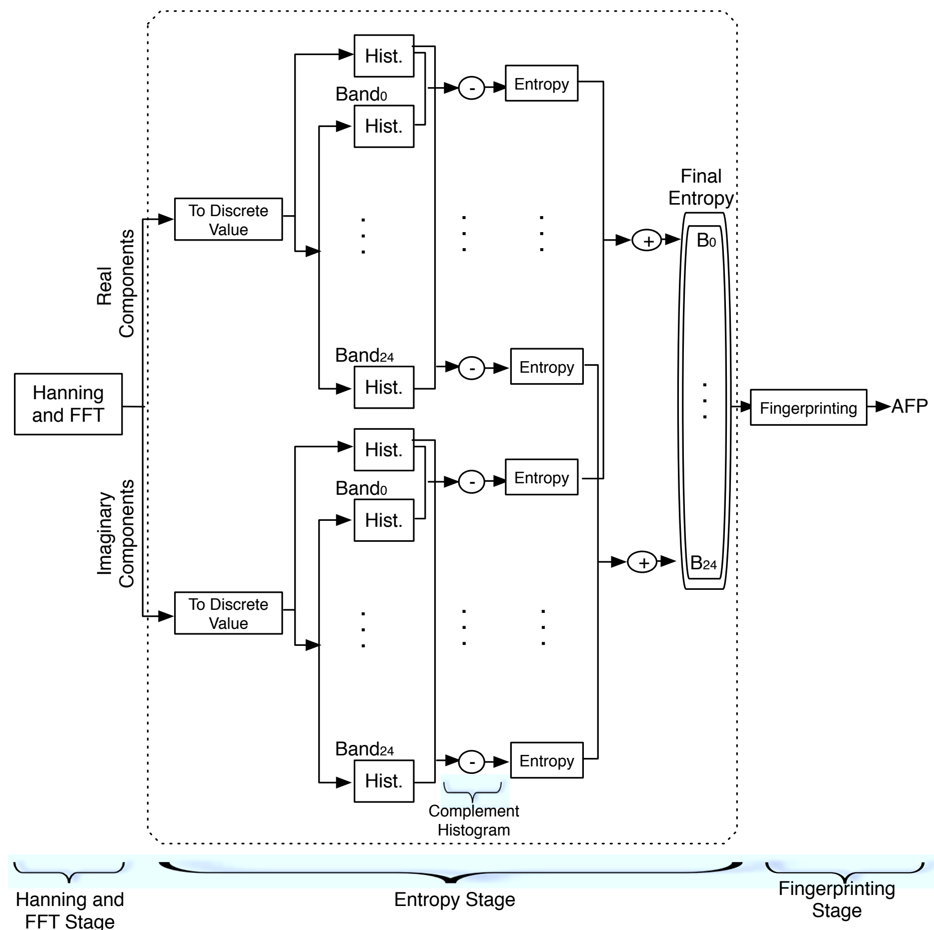

Figure 3: Entropy stage for each frame of signal

Each phase has the next characteristics:

- *Translation to Discrete Vector*: The continuous values have to be converted to discrete values. This implies to obtain the maximum and minimum values to determine the interval between them. The interval is divided in 256 subintervals and each value in a frame is assigned to one of them.

- *Final Band Histogram*: Many steps are needed to calculate the final histogram. Once the frame is converted to two vectors (real and imaginary parts) of discrete values between 0 and 255, the histogram is calculated. It is about 256 different values.

  In the previous implementations, the histograms were calculated using global memory. Although the obtained results are good, we can improve them using shared memory to reduce the global memory accesses (The shared memory is on-chip memory, it is shared by all threads of a block and is faster to access than global memory). In this proposal, the histogram is calculated directly over a 256-element vector that is allocated in shared memory. Each thread accesses directly at its corresponding location, which is defined by current discrete value of frame vector. The threads accesses to shared memory can present

Copyright © 2011 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

conflicts (Two or more threads operate simultaneously over the same memory address). This problem is resolved using the atomic functions Sanders and Kandrot (2010). If two or more threads want to access the same memory address, they are serialized.

After the histogram of the whole frame is calculated, the frame is divided in 24 critical bands according to Bark scale (see section 2) and twenty four histograms are calculated, one for each band. As the bands can be obtained with a standard filter bank tuned with the corresponding frequencies of the bark scale, their limits are the same for every frame, in consequence they can be computed in advance and read by each thread of the grid. In this proposal, the limits are calculated at the beginning and save to the constant memory of GPU (off-chip readonly memory with cached accesses). This design decision allows to improve the application performance.

From the total and band histograms, we calculate the complement histogram.

The whole process is applied twice, one for the real components and the other to the imaginary components, see figure 3. Consequently, the output of this phase is the real and imaginary histograms of each band. These histograms are the input of Band Entropy stage

- *Band Entropy*: Each element i of complement histogram represents the frequency of $i-th$ element ($f(i)$). It is necessary to obtain the probability according to Laplace's formula $\frac{f(i)}{N}$, where N is the total elements of frame (16384) minus the quantity of elements in the current band. Each thread calculates a probability, $p(x_i)$ and the $p(x_i) \times ln(p(x_i))$. Finally the thread results are added. This operation is a sum reduction and it is made in parallel Sanders and Kandrot (2010).

  The output is a twenty-element vector, its component is the sum of real and imaginary entropies of each band.

In this stage, a block computes the frame entropy. Each block has 256 threads, in consequence each of them is responsible of a data subset (The same case that FFT stage). The synchronization points are necessary between two continuous phases.

The output data of Entropy stage are N vectors, one per frame, whose 24 components are the entropy values of each band.

3.3 Fingerprinting Stage

Once entropy is computed for every frame of audio, the AFP can be calculated according to equation 1. The parallel implementation in GPU launches many blocks as frames exist. In this implementation, we define $N-1$ block with one thread. Each block calculates the frame AFP from its entropies.

Finally, all frame AFPs have to be moved to CPU.

4 ANALYSIS AND RESULTS

In the previous GPU AFP implementations, even though we reached good performance, they did not take advantage of every characteristics of GPU. To improve the GPU AFP process, we consider using memory hierarchy of GPU.

In the next section, first, we detail first the GPU used and characteristics of the audio signal: size MB and frame number. Following, we show the results obtained in different GPU for each signal.

Copyright © 2011 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

4.1 Proof Environment

The analysis was made for four GeForce GPU: GT320M, GT330M, GTX 260 and the GTX 470 whose characteristics are:

	GT 320M	GT 330M	GTX 260	GTX 470
Global Memory	265027584 bytes	536150016 bytes	938803200 bytes	1341325312 bytes
SM	6	6	27	14
SP	48	48	216	448
Clock rate	950 MHz	1040 MHz	1242 MHz	1215 MHz
Compute Capability	1.2	1.2	1.3	2.0

The six audio signals are songs of different sizes, the next table details the characteristic of six songs, its size and number of frame.

Audio ID	A-16MB	A-26MB	A-46MB	A-116MB	A-164MB	A-218MB
size MB	16	26	45.7	116.1	164.4	218
Frames	510	831	1462	3540	5015	6654

Each reported value is the averages of many executions of corresponding algorithm that detailed above.

4.2 Experimental Results

The first proposal resolved everything through the global memory and registers. As the GPU has a memory hierarchy with different costs of access, this work presents the results obtained by the use of constant and shared memory in AFP stages. The main changes were in the entropy stage. In this stage, two tasks were improved: the histogram computation and division of a frame according to Bark scale.

The histogram computation is repetitive and very costly, more exactly, it is performed fifty times for each audio frame (twenty five times for real components and twenty five for imaginary parts). Moreover, if it is done using a memory with high latency access, the overall performance will be affected. In this work, the histogram is implemented using shared memory and atomic functions. They are necessary to manage access conflicts. Figure 4 shows the time spent by Entropy stage using memory global (GM) and using shared memory with atomic function ($SM - AF$).

Results are better and the computation time is reduced drastically. These benefits are achieved to expense of portability, GPUs with compute capability 1.2 or grater are required NVIDIA (2008).

Other kind of GPU memory is the constant memory, we use it to save the limits of each band. The limits are the same for all frames, in consequence they are computed at the beginning in CPU and saved in the constant memory of GPU. Every thread of applications can read it faster than the global memory. The corresponding times of two implementations are detailed in the next table, figure 5. In this case we considered the three larger audio signals.

The obtained times by implementation with constant memory are a little better than the implementation with global memory. The difference is not great because the volume of data is small.

Copyright © 2011 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

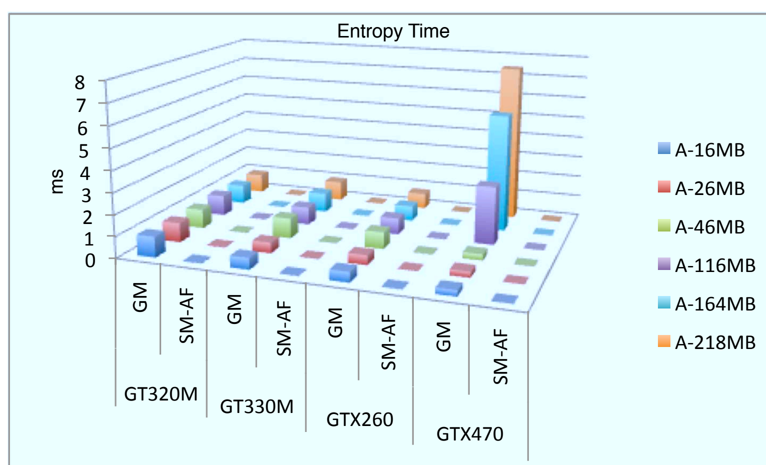

Figure 4: Global Memory Histogram vs Shared Memory Histogram

GPU	Impl. Type	A-116MB	A-164MB	A-218MB
GT320M	GM	0.0133	0.0135	0.0139
	CM	0.0127	0.0124	0.0126
GT330M	GM	0.0203	0.0250	0.0197
	CM	0.0181	0.0154	0.0155
GTX260	GM	0.016	0.0156	0.014
	CM	0.013	0.0116	0.011
GTX470	GM	0.007	0.0077	0.0076
	CM	0.0067	0.0067	0.0068

Figure 5: Time of Entropy stage using Global Memory (GM) and Constant Memory (CM)

Respect to general application, the improvements in entropy stage increased significantly the performance GPU AFP process. The figure 6 displays the total time obtained in both applications.

The table in figure 7 shows the speedup of the pure computation step, without considering data transfers. For massive audio fingerprinting a multiple buffer strategy should be implemented to take care of data transfers between disk, the CPU memory and the GPU memory. In other words, a continuous flow of data should be ensured from the disk to the CPU and from the CPU to the GPU. This will ensure a maximum resource usage in the GPU.

Observe the speedup increase as more of the data zone of the GPU is used.

5 CONCLUSIONS AND FUTURE WORK

In this work, we sketched the basic characteristics of the $GPUAFP$ process. This process obtains the AFP for an audio signal from the parallel processing of its frames. A frame is a signal fragment of 16KB length. All frames are processed simultaneously, if the whole audio signal can be accommodated in the GPU RAM.

Through of the use of GPU memory hierarchy, we optimized and got very well results respects of process performance and resource demand, we reduced the global memory accesses and the transfers between CPU and GPU. We obtained very good speedup, mainly for bigger data size.

At this moment we are working over data transferring between CPU and GPU. The transfer-

Copyright © 2011 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

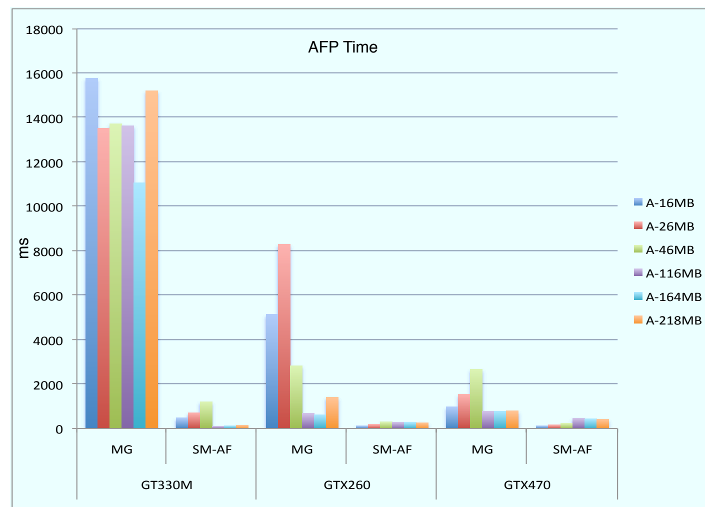

Figure 6: Total time of two GPU AFP implementation

Audio ID	A-116MB	A-164MB	A-218MB
GT320M	443,10	762,95	832,972
GT330M	299.302	354.56	388.941
GTX260	46.03	47.77	196.94
GTX470	67.29	70.84	72.29

Figure 7: Speedup of GPU AFP

ring the WAV file to the data memory in the GPU is costly. It is usual to have the audio available in mp3 or other compression format with size is about one tenth of the WAV file. Transferring the mp3 file to the GPU and there decompress the stream, that probably will increase the speedup, compared to the sequential process. We will also try with a number crunching (double precision) GPU instead of a gamer version to improve the quality of the AFP.

REFERENCES

Bercher J. and Vignat C. Estimating the entropy of a signal with applications. *IEEE Transactions on Signal Processing*, 48(6):1687–1694, 2000.

Cooley J. and Tukey J. An algorithm for the machine calculation of complex fourier series. *Math. Comput.*, 19:297âĂŞ301, 1965.

Danielson G.C. and Lanczos C. Some improvements in practical fourier analysis and their application to x-ray scattering from liquids. *J. Franklin Institute*, 233:365âĂŞ380 and 435âĂŞ452, 1942.

Haitsma J. and Kalker T. A highly robust audio fingerprinting system. In *International Symposium on Music Information Retrieval ISMIR*. 2002.

Ibarrola A.C. and Chavez E. Real time tracking of musical performances. In *MICAI*, volume To appear. 2010.

Ibarrola J.A.C. *Análisis digital de la señal de voz.* Ph.D. thesis, Universidad Michoacana de San Nicolás de Hidalgo, México, 2007.

Copyright © 2011 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

Kurth F. and Scherzer R. A unified approach to content-based and fault tolerant music recognition. In *114th AES Convention, Amsterdam, NL.* 2003.

Miranda N. Piccoli F. C.E. Using gpu to speed up the process of audio identification. In *I2TS 2010 - 9th International Information and Telecommunication Technologies Symposium.* IEEE - R9, 2010. ISBN 978-85-89264-11-2.

Miranda N. Piccoli F. C.E. and A. C.I. Fast gpu audio identification. In *16vo Congreso Argentino de Ciencias de la Computacion (CACIC 2010)*, page 229:242. 2010a. ISBN 978-950-9474-49-9.

Miranda N. Piccoli F. C.E. and A. C.I. Finding audio fingerprinter using gpu. In *IX Congreso Argentino de Mecanica Computacional - XXXI Congreso Iberico Latinoamericano de Metodos Computacionales en Ingenieria (Mecom - Ciilamce 2010)*, page 3107:3126. 2010b. ISSN 1666-6070.

NVIDIA. Nvidia cuda compute unified device architecture, programming guide version 2.0. In *NVIDIA*. 2008.

Sanders J. and Kandrot E. *CUDA by Example, An Introduction to General Purpose GPU Programming*. 2010. ISBN 978-0-13-138768-3.

Shannon C. and Weaver W. *The Mathematical Theory of Communication*. University of Illinois Press, 1949.

Shin S., Kim O., Kim J., and Choil J. A robust audio watermarking algorithm using pitch scaling. In *14th International Conference on Digital Signal Processing*, volume 2, pages 701 – 704. 2002.

Subramanya S., Simha R., Narahari B., and Youssef A. Transform-based indexing of audio data for multimedia databases. In *International Conference on Multimedia Applications*. 1999.

Sukittanon S. and Atlas E. Modulation frequency features for audio fingerprinting. In *IEEE, International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, volume 2, pages 1773–1776. University of Washington USA, 2002.

Wang A. An industrial strength audio search algorithm. In *International Conference on Music Information Retrieval (ISMIR)*. 4th International Conference on Music Information Retrieval, Baltimore, Maryland, USA, October 27-30, 2003, 2003.

Zwicker E. Subdivision of the audible frequency range into critical bands. *The Journal of the Acoustical Society of America*, (33), 1961.

Copyright © 2011 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar