Journal of Supercomputing manuscript No.
(will be inserted by the editor)

An Efficient Musical Accompaniment Parallel System for mobile devices

Pedro Alonso · P. Vera-Candeas · Raquel Cortina · José Ranilla

Received: date / Accepted: date

Abstract This work presents a software system designed to track the reproduction of a musical piece with the aim at match the score position into its symbolic representation on a digital sheet. Into this system, known as automated Musical Accompaniment System, the process of score *alignment* can be carried out real-time. A real-time score alignment, also known as *score following*, poses an important challenge due to the large amount of computation needed to process each digital frame and the very small time slot to process it. Moreover, the challenge is even greater since we are interested on handheld devices, i.e. devices characterized by both low power consumption and mobility. The results presented here show that it is possible to exploit efficiently several cores of an ARM® processor, or a GPU accelerator (presented in some SoCs from NVIDIA) reducing the processing time per frame under 10 ms in most of the cases.

Keywords Audio-to-Score Alignment · Score Following · Musical Accompaniment · Parallel Computing · Real-time Computing

1 Introduction

The task of synchronizing an audio recording of a musical piece with the corresponding sheet music is known as *alignment* [1]. Alignment is a process that can

Pedro Alonso
Depto. de Sistemas Informáticos y Computación
Universitat Politècnica de València, Spain
E-mail: palonso@upv.es

Raquel Cortina · José Ranilla
Depto. de Informática
Universidad de Oviedo, Spain
Raquel Cortina E-mail: raquel@uniovi.es
José Ranilla E-mail: ranilla@uniovi.es

P. Vera-Candeas
Telecommunication Engineering Department
Universidad de Jaén, Spain
P. Vera-Candeas E-mail: pvera@ujaen.es

be carried out *offline* or *online*. *Online alignment*, also called *score following*, is the real-time synchronization of a live musician playing a score (music live performance) with the symbolic score itself and decoding of expressive parameters of the musician on the fly. The interactivity between the performance and the score following (Interactive Music System) is associated with an electronic equipment that "listens" and "responds" to the performer's input. To conduct this tracking, the system software running on this equipment acquires a digital signal and performs the estimation from the current and previous signals. There are many useful applications for an efficient score following system featuring, for instance, automatic page turning [2], automated computer accompaniment of a live soloist [3], synchronization of live sound processing algorithms for instrumental electroacoustic composition, and the control of visual effects synchronized with the music.

Before triggering the score following, it takes place an *offline preprocessing* stage. Under this stage, all the information contained in the MIDI score is organized so that it can be used for alignment purposes. At this stage, for instance, a set of combinations of concurrent notes and the transitions between them is defined. Afterwards, the score following, which works real-time, divides the processing on each upcoming digital frame in several stages. The first one is the *feature extraction*, where the features that characterize some specific information about the musical content are extracted from the audio signal. The second stage, named *Module Distortion*, is in charge of calculating a cost matrix that will be used as input by the following stage; it is the most expensive in terms of floating point operations (flops). The third stage performs the matching by finding the best match between the feature sequence and the score. Of the two main methods used for this matching, i.e. the Hidden Markov Model (HMM) [8–11] and the Dynamic Time Warping (DTW) [4, 12–14], the last one is the one which has been integrated in the system proposed in this contribution.

In this work, we aim at closing the whole online alignment for score following system by proposing a fast implementation of the most expensive module, the second stage, i.e. the *Module Distortion*. Our implementation exploits one of the most commonly found multicore processors in tablets and smart phones, such as it is the ARM® processor. Moreover, our application is also able to use the embedded GPU contained into some NVIDIA ARM® processor chips, allowing thus to achieve a high performance for very long musical scores.

The structure of the paper is as follows. In Section 2, we explain the architecture of the proposed system and go through details of the computational aspects of the application from a theoretical point of view. Section 3 exposes the implementation, in particular, for the GPU since is one of the most important highlights of the paper. The experimental results are shown in Section 4. The paper is closed with a conclusions section.

2 The Musical Accompaniment System

Our aim is to address the score following problem through the development of an automated computer Musical Accompaniment System that is mainly based on the framework proposed in [4]. Of this system, the matching stage (the third one), which has been implemented using the DTW method, was already tackled in [15], where it was designed a parallel version suitable for several processor cores and

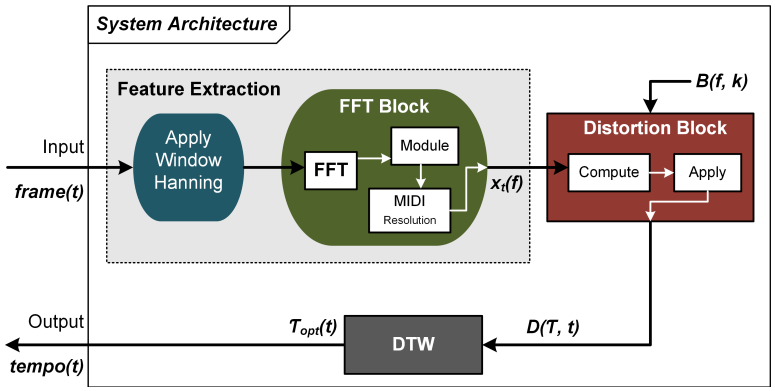

Fig. 1 The Musical Accompaniment System.

for GPUs. The software solution proposed for the *online alignment* has been built satisfying the following two basic requirements: *real-time* and *mobility*. Accordingly, this design should be aware of the low computational power of the *handheld* devices, specially the cheapest ones, and should deeply use the possibilities offered by some heterogeneous parallel architectures that provide the processor with a low consumption GPU embedded into the same chip (SoC).

Figure 1 outlines the main blocks of the Musical Accompaniment System. Our audio signals (*frame*) are monaural with 44.1 kHz sampling frequency and 16 bits per sample. When a new frame arrives, *frame(t)*, the process starts at the **Feature Extraction** block, where it is obtained a low-level spectral representation of the audio data (Time-Frequency representation). First, a windowed Fast Fourier Transform (FFT) is applied to the signal. A Hanning window with a size of 128 ms and a hop size of 10 ms is used (for both synthetic and real-world signals). Note that, according to these values the frame length (N_{frame}) is 5700 samples. We chose this value as in [17] to have enough frequency resolution for low frequency sounds. However, the transform length (N_{FFT}) is 16384 (zero padding is used) in order to obtain at least one frequency bin to each MIDI interval.

Performed the FFT, the time-frequency representation is converted from linear frequency to MIDI resolution. In order to obtain this computation, first the magnitude spectrogram (inset *Module* in Fig. 1) is computed from the complex output of the FFT (in real and imaginary parts). Afterward, the frequency bins belonging to the same MIDI interval is summed up (inset *MIDI Resolution* in Fig. 1) resulting in $x_t(f)$, that is the time-frequency representation of the input frame in MIDI resolution. Note that the number of MIDI pitches (N_{MIDI}) corresponds with the range of notes that a piano can play in MIDI scale.

The inputs of the **Distortion Block** (DB) are $x_t(f)$, in addition to the spectral patterns $B(f, k)$ obtained in the preprocessing stage. Note that the maximum value of k is N_{BASES}, which corresponds to the number of units in the score. A score unit represents the occurrence of concurrent or isolated notes in the score (see [17]). The Distortion Block calculates $D_t(\tau)$, which measures the suitability of each unit to be active at each frame t (referenced to the signal input) by analyzing the likelihood between the spectral patterns $B(f, k)$ and the time-frequency representation of the input signal $x_t(f)$. Note that, in $D_t(\tau)$, τ represents the synthetic signal frames

Algorithm 1 Module Distortion

Let $\mathbf{B}(f,k)$ the matrix learned in preprocessing stage.
{/* Compute Block */}
$\Phi_t(k) = 0$.
for k=1 to N_{BASES} **do**
  for f=1 to N_{MIDI} **do**
    $\Phi_t(k) = \Phi_t(k) + Dist_\beta(x_t(f),\ g_t(k)B(f,k))$, where $g_t(k) = \frac{\sum_f x_t(f)B(f,k)^{(\beta-1)}}{\sum_f B(f,k)^\beta}$.
  end for
end for
{/* Apply Block */}
for τ=1 to DTW_{size} **do**
  $D_t(\tau) = \Phi_t(k_\tau)$, where k_τ is the unit played at time τ at the score.
end for

or score positions. To obtain the optimum unit at each frame we use, as a cost function, the Beta-divergence (see (1)), which includes in its definition the most used costs functions in the state-of-art (see [16]). One component per unit is used so that, single non-zero restriction can be imposed to the gains allowing thus the use of the efficient signal decomposition method described in [16], among others. Finally, the length of $D_t(\tau)$ at frame t is given by the DTW_{size} parameter, that is the duration in frames of the score (MIDI time) to be aligned. Remember that, as in [17], a hop size of 10 ms is used for frame duration in MIDI time.

As it can be seen in Algorithm 1, the computational intensity of the Distortion Block is larger than in Feature Extraction module. This is mainly because it depends on the number of units of the composition. Consequently, the higher the number of units of the score, the larger the computational intensity.

$$D_\beta(x|\hat{x}) = \begin{cases} \frac{1}{\beta(\beta-1)}\left(x^\beta + (\beta-1)\hat{x}^\beta - \beta x\hat{x}^{\beta-1}\right) & \beta \in (0,1)\cup(1,2] \\ x\log\frac{x}{\hat{x}} - x + \hat{x} & \beta = 1 \\ \frac{x}{\hat{x}} + \log\frac{x}{\hat{x}} - 1 & \beta = 0\ . \end{cases} \quad (1)$$

Finally, in order to perform the alignment we used the **DTW** to match the score position with each input signal frame. In this stage and in order to reduce latency, no backtracking is allowed, that is, the decision is made directly from the information contained into frame t. The simplest online approach is obtained by matching the performance position at frame t with the score position τ associated to the minimum value of the accumulated cost at frame t. A thorough explanation of the theoretical aspects of the problem, as well as the *multi-tier* architecture designed for the DTW functional block, can be found in [4,16,17]. Our parallel implementation of this algorithm, presented in [15], has been used here with some little changes, including some optimizations carried out to avoid the downside of a reduction operation in parallel.

From the theoretical point of view, the sequential cost of the computation applied to each frame can be approximated by

1. **Feature Extraction**.
   - *Hanning Window*. Element-wise product of two vectors of size N_{frame}.
   - *FFT*. One-dimensional FFT algorithms have computational complexity of $O(n\log(n))$ when n is a power of 2. We have used the high performance

Algorithm 2 Distortion matrix computation method

```
1  for k = 1 to N_BASES do
2    for f = 1 to N_MIDI do
3      α = α + f_f T_{k,f}
4    end for
5    α = α/n_t
6    c = α^{β−1}
7    for f = 1 to N_MIDI do
8      d = d + (f_f^β + c(α(β − 1)S_{k,f} − βf_f)T_{k,f})/(β(β − 1))
9    end for
10   DS_k = d
11 end for
```

libraries `fftw` [18] for the processor and `cuFFT` [19]) for the GPU. The vector size N_{FFT} has been selected as a power of 2 so the FFT cost can be approximated by $N_{FFT}\log_2(N_{FFT})$ flops.
- *Module*. Performs $N_{FFT}/2$ times three basic operations: $3N_{FFT}/2$ flops.
- *MIDI Resolution*. Computes N_{MIDI} squares and the sum of the elements calculated in *Module*, thereby giving a cost of $N_{MIDI} + N_{FFT}/2$ flops.

2. **Module Distortion**.
   - *Compute*. The computational cost depends on the value of β (Eq. 1). However, for all the cases is $O(N_{bases} \times N_{MIDI})$.
   - *Apply*. Performs $2DTW_{size}$ operations so $\in \theta(DTW_{size})$.
3. **DTW**. Following [15], the cost of this operation is $O(DTW_{size})$, being for the parallel version $O\left(\frac{plog(p)+DTW_{size}}{p}\right)$.

In summary, the overall cost is

$$O(N_{frame} + N_{FFT}\log_2(N_{FFT}) + 2N_{FFT} + N_{MIDI} + N_{bases} \times N_{MIDI} + 2DTW_{size}) \approx$$
$$O(N_{frame} + N_{FFT}\log_2(N_{FFT}) + N_{FFT} + N_{MIDI} + N_{bases} \times N_{MIDI} + DTW_{size}) \ .$$

for the sequential version. When parallel computing is used in multi-core systems and following [15], the theoretical computational complexity of DTW is $O\left(\frac{plog(p)+DTW_{size}}{p}\right)$. The rest of the steps are mainly conformed by loops with independent iterations and, thereby, easily divided between the CPU cores (excluding FFT where an external optimal code is used). Consequently, we can state that the efficiency is near to the optimal, specially when the number of cores of the CPU is low, which is the case of the ARM® CPUs.

3 The Module Distortion

We depart in our analysis form a refined version of Algorithm 1, in particular of lines 3–8, that we can inspect in Algorithm 2. This algorithm works mainly walking on two matrices, T and S of size $N_{BASES} \times N_{MIDI}$, to produce an array DS of size N_{BASES} of distortion states. Based on this fact we have developed a CUDA kernel that performs this computation in parallel using as many threads as possible to execute all possible operations in parallel.

To this end, we arrange computations in 2D thread blocks of size 32×16 so that parameter N_{MIDI} (which is 114) is partitioned in chunks of size 32 and parameter

Listing 1 Kernel for the distortion matrix computation.

```
__global__ void kernel_CD(double* DS, const double* f, const double* n,
                const double* S, const double* T, const double* p,
                const double beta, const int MIDI, const int BASES) {
  unsigned int      i =  blockIdx.x * blockDim.y + threadIdx.y, j, k;
  unsigned int stride = i * 128;
  unsigned int th_row = threadIdx.y;
  unsigned int th_col = threadIdx.x;
  unsigned int    row = i + threadIdx.x;
  bool          guard = th_row == 0 && row < BASES && th_col < blockDim.y;
  double a = 0.0, b, c = beta-1.0;

  __shared__ double s_a[16], s_b[16];

  if ( i < BASES ) {
     for( j=th_col, k=stride+th_col; j<MIDI; j+=32, k+=32 )
       a += f[j] * T[k];
     a = warpReduceSum(a);
     if( th_col == 0 ) s_a[th_row] = a;
     __syncthreads();
     if( guard ) {
       a = s_a[th_col] / n[row];
       b =  pow(a, c);
       s_b[th_col] = beta * b;
       s_a[th_col] = b * a * c;
     }
     __syncthreads();
     j = th_col; k = stride+th_col;
     for( j=th_col, k=stride+th_col; j<MIDI; j+=32, k+=32 )
       a += (p[j]+T[k] * (S[k]*s_a[th_row]-f[j]*s_b[th_row])) / (beta*c);
     a = warpReduceSum(a);
     if( th_col == 0 ) s_a[th_row] = a;
     __syncthreads();
     if( guard ) DS[row] = s_a[th_col];
  }
}
```

N_{BASES} is partitioned in chunks of 16. This way we can get that each thread access exclusively one position of matrices T and S. Previously to the call to the kernel we need to managed data adequately to make more efficient the performance of the kernel. Matrices T and S are fit into memory within an allocated array of size $N_{BASES} \times 128$ for alignment purposes. Consequently, each row of T or S has 128 elements though only the first 114 positions are referenced in the computations.

Listings 1 shows a snippet of the kernel for the computation of the distortion array of states that results from the translation of Algorithm 2 into CUDA. The listing shows the double precision version, existing also the simple precision one. For simplicity and coherence between Algorithm 1 and Listing 1, it must be noted that we assume vector `p` has been previously computed by another kernel to power each entry of array `f` to β.

The kernel fills vector `DS` with the computed distortion states. We have focused mainly onto the optimization of the memory access pattern, with the help of the CUDA profiler, since this fact is one of the most that usually influences on efficient executions of CUDA kernels, specially of memory bound codes. Variable `i` addresses the ith row of matrices T and S. (Be aware that we store these rectangular matrices in a linear array.) Note that `blockDim.x` is 32 and `blockDim.y` is 16 so the threads within the same warp access consecutive memory locations (access

entries within the same row). The goal is that loops in lines 15–16 and 28–29 are executed by all the threads within the block. Upon termination of these loops, each thread contains a partial result stored into the local variable `a` that must be combine by a reduction operation. We aimed to manage operations so that they can take place within the active warp. Hence, the reduction can be carried out using warp shuffle instructions to exchange variables among threads. This instruction set is available since CUDA compute capability 3.x, and is available in the target machine used for our experiments. The device kernel `warpReduceSum` performs a reduction sum over variables `a` of all the threads within the active warp, leaving the final result into the variable `a` of the first thread of that warp. This routine is easily accessible on Internet[1].

We used a boolean variable `guard`, computed at line 9, that is true only for the first warp of the thread block. This warp is in charge of computing some instructions like, for instance, those of the `if` clause at lines 20–25. Otherwise, only the first thread of the warps would execute this snippet of code thus loosing a lot of performance. Furthermore, we can guarantee this way that the access to memory (in line 33) is always to consecutive allocations.

4 Evaluation and experimental results

For the experimental evaluation we have focused our interest only on a light-weighted device like it is the NVIDIA Jetson, in particular, the TX1 model. This development kit features a Quad-core ARM® Cortex®-A57 MPCore Processor, and a NVIDIA Maxwell™ GPU with 256 NVIDIA® CUDA® Cores. The TX1 version of this board represents a significant step forward over the previous model (TK1) that allows to address the alignment of scores of longer duration in less time. Note that this micro-architecture is similar to that used in many tablets and smartphones (e.g. Jiayu S3s, with a more powerful CPU, or NVIDIA Shield, with the same chip), thus fulfilling with one of the established requirements: mobility. In addition, the improvement we have made in the software developed, in particular, in the GPU version produced as a result a very good performance. In order to help improving the final output we used the high performance libraries `fftw` and `cuFFT`.

Figure 2 summarizes the most important results we can expect from the NVIDIA Jetson executing our high performance online score matching software to follow a musical score by using either the ARM cluster of cores (left graphics) or the embedded GPU (right graphics). The results are presented in terms of AveRage Time per Sequence (ARTpS) and WoRst Time per Sequence (WRTpS) in milliseconds. The first is the most likely time that we may need to process a frame, while the second one represents an upper bound. The system can be considered quite stable according to the similarity obtained between the two times in most of the cases. The complexity per frame of the algorithm depends on the length of the score. We used synthetic audio files of duration: 150 sec., 5 min., 10 min., 15 min., and 30 min. to test and explore the limits of the system. The implemented software makes use of double precision real numbers (64 bits) and simple precision (float numbers of 32 bits), and all the parameters of the score alignment algorithm

[1] `https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler`

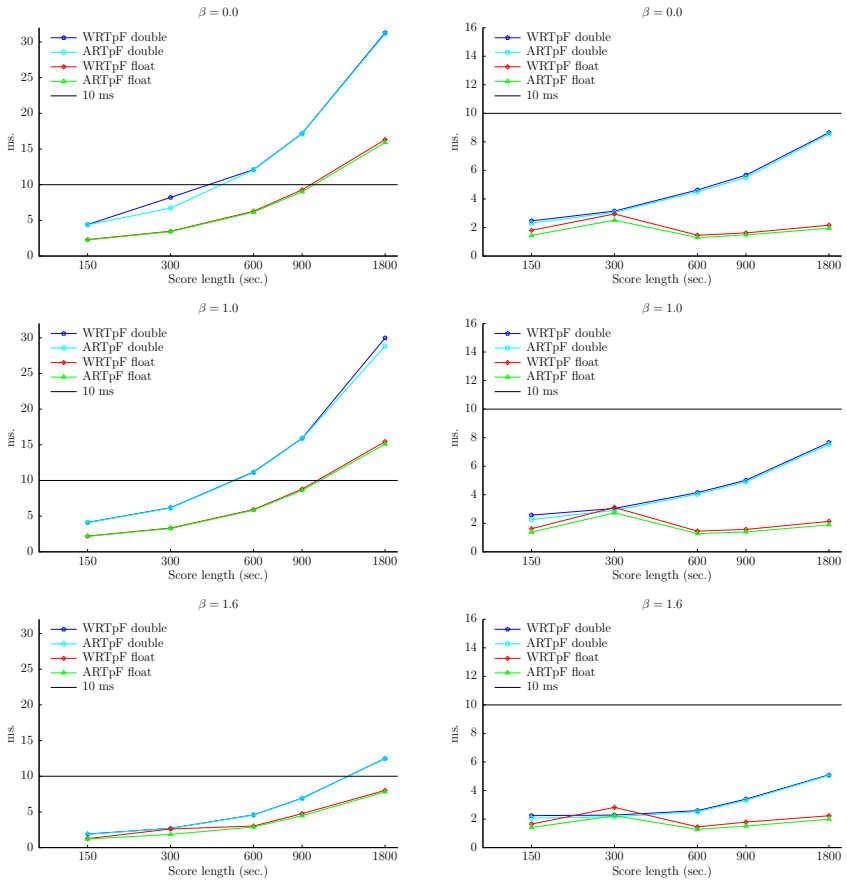

Fig. 2 Experimental results of Online Score Following. AveRage (ARTpS) and WoRst (WRTpS) Time per Frame (in ms.) using the four ARM cores (left figures) and the GPU (right figures) for $\beta = \{0, 1, 1.6\}$.

were set to the same values given in [17]. The flat line that intersects the y-axis at 10 ms. in the figure corresponds to what we judge is the maximum acceptable time to process a frame in real-time (real-time threshold) (with the hop size between frames used in [17]).

As it is shown in the graphics on the left, the implemented score matching is able to execute in real-time for all cases except for the largest score in double precision in the general case for β, i.e. for $\beta \notin \{0, 1\}$. When β is 0 or 1 the system responds in real-time for scores of ≤ 15 minutes in simple precision, and for scores of ≤ 5 minutes. Different values of β may imply the use of the *log* function, only multiplications, or power functions (see (1)) so, consequently, may highly impact on the frame processing time. The existence of a GPU coprocessor embedded in the Jetson device in addition to the implementation of the score following algorithm presented in this contribution for the GPU allows to fit the frame processing time always below the real-time threshold for all the combinations of β and score

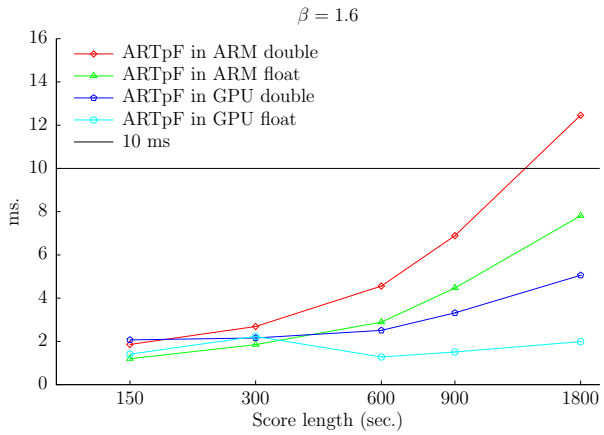

Fig. 3 Comparison of the frame processing time between the use or not of the GPU coprocessor of the NVIDIA Jetson TX1.

Table 1 Average time in seconds of 50 repetitions of the main blocks of the proposed system for a 30 minutes audio file using double precision real numbers.

	ARMx1	ARMx4	GPU
Feature Extraction	38.34	13.02	36.18
Distortion Block	530.29	142.64	39.77
Dynamic Time Warping	343.14	88.80	14.90

lengths, whatever the precision used, as it can be seen on the right hand size graphics of Fig. 2.

Finally, and for comparison purposes, we show in Fig. 3 the relationship among the times achieved when using or not the GPU, varying the precision and the score length, using the general case for the β parameter as example. Also, we show in Table 1 the average time of the main blocks of the Musical Accompaniment System depicted in Fig. 1 using double precision for a 30 minutes audio file. Combining the information of Fig. 3 and Table 1 it can be appreciated the substantial growth in GPU performance with regard to the ARM processors if the Feature Extraction stage (FS) is not considered. It is important to note that the computational weight of FS, which is quite small when compared with the other two blocks, comes from the FFT module and we used there libraries `fftw` and `cuFFT`. Library `fftw` is very optimized for the ARM® processor and performs very well in parallel. Table 1 also shows that the parallelization, implemented with OpenMP [20] for the CPU, is efficient, i.e. the empirical efficiency obtained is close to one, as it was expected according to the theoretical estimation presented in Section 2.

5 Conclusions

In this work we have addressed the score following problem developing an automated computer Musical Accompaniment Parallel System. We managed to use

high performance computing techniques including tools like OpenMP for multi-cores or CUDA for GPUs with the aim of solving the problem in real-time using hadheld/mobile devices. The low computational power of these devices and the small time window (10 ms) to process each digital frame are both very strong and critical requirements enforced by many applications.

To demonstrate the high performance achieved with our proposed software system, we carried out the tests using the most commonly multi-core processors it can be found in modern tablets and smartphones, i.e. the ARM® processor, and also the embedded GPU contained into some NVIDIA ARM® processor chips. In this context, we have analyzed the limits of a real-time response in terms of score duration. As a result, we conclude that the system always achieves good score alignments within the real-time constraint of $10ms$ for short/medium score durations. For long, and very long, scores it is necessary to use our GPU based approach.

Acknowledgments

This work was supported by the Ministry of Economy and Competitiveness from Spain (FEDER) under projects TEC2015-67387-C4-1-R, TEC2015-67387-C4-2-R and TEC2015-67387-C4-3-R, the Andalusian Business, Science and Innovation Council under project P2010-TIC-6762 (FEDER), and the Generalitat Valenciana PROMETEOII/2014/003.

References

1. A. Cont, D. Schwarz, N. Schnell, and C. Raphael. 2007. "Evaluation of real- time audio-to-score alignment" *in Proc. of the International Conference on Music Information Retrieval (ISMIR) 2007*, Vienna, Austria.
2. A. Arzt. 2008. "Score Following with Dynamic Time Warping. An Automatic Page-Turner" *Master's Thesis, Vienna University of Technology*, Vienna, Austria.
3. C. Raphael. 2010. "Music Plus One and Machine Learning" *in Proc. of the 27 th International Conference on Machine Learning*, Haifa, Israel, 21-28.
4. J.J. Carabias-Ortí, F.J. Rodríguez-Serrano, P. Vera-Candeas, N.Ruiz-Reyes, F.J. Cañadas-Quesada. 2015. "An Audio To Score Alignment Framework Using Spectral Factorization And Dynamic Time Warping," *in Proc. of the International Conference on Music Information Retrieval (ISMIR)*, pp. 742-748, Málaga, Spain.
5. A. Cont. 2010. "A coupled duration-focused architecture for real-time music-to-score alignment," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 32, no. 6, pp. 974-987.
6. N. Montecchio, and N. Orio. 2009. "A discrete filterbank approach to audio to score matching for score following," *in Proc. of the International Conference on Music Information Retrieval (ISMIR)*, pp. 495-500.
7. M. Puckette. 1995. "Score following using the sung voice," *in Proc. of the International Computer Music Conference (ICMC)*, pp. 175-178.
8. Z. Duan, and B. Pardo. 2011. "Soundprism: An Online System for Score-informed Source Separation of Music Audio," *IEEE Journal of Selected Topics in Signal Processing*, vol. 5, no. 6, pp. 1205-1215.
9. A. Cont. 2006. "Realtime audio to score alignment for polyphonic music instruments using sparse non-negative constraints and hierarchical hmms," *In Proc. of IEEE International Conference on Acoustics Speech and Signal Processing (ICASSP)*, Toulouse. France.
10. P. Cuvillier, and A. Cont. 2014."Coherent time modeling of Semi-Markov models with application to realtime audio-to-score alignment". *In Proc. of the 2014 IEEE International Workshop on Machine Learning for Signal Processing*, 16.

11. C. Joder, S. Essid, and G. Richard. 2013. "Learning optimal features for polyphonic audio-to-score alignment." *IEEE Transactions on Audio, Speech, and Language Processing*, 21, 10, 2118-2128.
12. S. Dixon. 2005. "Live tracking of musical performances using on-line time warping," *in Proc. International Conference on Digital Audio Effects (DAFx)*, Madrid, Spain, pp. 92-97.
13. N. Hu, R. B. Dannenberg, and G. Tzanetakis. 2009 "Polyphonic audio matching and alignment for music retrieval," *in Proc. of the IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)*, pp. 185-188.
14. N. Orio, and D. Schwarz. 2001. "Alignment of monophonic and polyphonic music to a score," *in Proc. International Computer Music Conference (ICMC)*.
15. P. Alonso, R. Cortina, F. J. Rodríguez-Serrano, P. Vera-Candeas, M. Alonso-Gonzalez, and J. Ranilla. 2016. "Parallel online time warping for real-time audio-to-score alignment in multi-core systems," *The Journal of Supercomputing*, published online DOI 10.1007/s11227-016-1647-5.
16. J.J. Carabias-Ortí, F.J. Rodríguez-Serrano, P. Vera-Candeas, F.J. Cañadas-Quesada, and N. Ruiz-Reyes. 2013 "Constrained non-negative sparse coding using learnt instrument templates for realtime music transcription," *Engineering Applications of Artificial Intelligence*, Volume 26, Issue 7, August 2013, pp. 1671-1680.
17. J.J. Carabias-Ortí, F.J. Rodríguez-Serrano, P. Vera-Candeas, and D. Martínez-Muñoz. 2016. "Tempo Driven Audio-to-Score Alignment using Spectral Decomposition and Online Dynamic Time Warping," *ACM Trans. Intell. Syst. Technol.*, accepted.
18. FFTW. `http://www.fftw.org`. *Last accessed on July, 2016.*
19. NVIDIA CUDA Fast Fourier Transform library (cuFFT). `http://https://developer.nvidia.com/cufft`. *Last accessed on July, 2016.*
20. The OpenMP API specification for parallel programming. `http://openmp.org`. *Last accessed on July, 2016.*