Hermite Polynomial Characterization of Heartbeats with Graphics Processing Units

Alberto Gil[1], Gabriel Caffarena[1], David G. Márquez[2], and Abraham Otero[1]

[1] University CEU-San Pablo,
Urb. Monteprincipe, 28668, Madrid, Spain
gabriel.caffarena@ceu.es
http://biolab.uspceu.com
[2] Centro Singular de Investigación en Tecnoloxías da Información (CITIUS),
University of Santiago de Compostela, 15782 Santiago de Compostela, Spain

Abstract. In this paper we address the massive parallelization of the characterization of heartbeats by means of Graphics Processors. Heartbeats are represented with Hermite polynomials due to the compactness and robustness of this representation. Both the off-line and on-line characterization of QRS complexes are covered, thus, assessing the capabilities of Graphics Processors for these tasks. The results yield that off-line processing with a GPU can be computed 200× faster than a standard CPU, while on-line processing can be 100× faster.

Keywords: Hermite functions, ECG, QRS, Arrhythmia, GPU, CUDA, Parallelization, Heartbeat Representation

1 Introduction

Automatic ECG classification stands as a powerful tool supporting cardiologists in the task of identifying arrhythmias in a long-term ECG recording. This technique reduces the amount of time that the cardiologist spends performing the visual inspection of ECG.

The QRS complex is the part of the ECG recording that reflects the electrical activity of the ventricles and its characterization with Hermite functions has proved to be a reliable means to perform automatic classification of beats [1]. The two main advantages are the low sensitivity to noise and artifacts, and the ability to represent a QRS complex using a reduced set of parameters (e.g. a 144-sample QRS can be characterized with 7 parameters [2]). These advantages have made the Hermite representation a very common tool for characterizing the morphology of the beats [1–5].

It is common that the implementation of scientific applications requires the use of parallel systems to provide results within a reasonable computation time. Among the different available techniques, the parallelization through Graphics Processing Units (GPU) has thrived in the recent decades. GPUs are cheap, easy to install and, in many cases, they are as powerful in terms of computation as hundreds of microprocessors working in parallel. A single commodity PC with a

GPU attached – via the PCIe connection – can replace a whole cluster with tens and even hundreds of computers. Many biomedical applications have benefited from this technology: MRI reconstruction [6, 7], cardiac tissue simulation [8], biomolecular dynamics [9, 7], bioinformatics [7], etc.

In this paper, we assess the suitability of GPU parallelization for the fast characterization of ECG recordings by means of Hermite polynomials. The MIT-BIH arrhythmia database [10] is used as a benchmark, and two different scenarios are selected: off-line processing and on-line processing.

The paper is divided as follows: Section 2 introduces the characterization of beats based on Hermite functions. Section 3 briefly introduces the GPU programming model. Section 4 explains the parallelization of the characterization algorithms for the off-line and on-line scenarios. The results are presented in Section 5 and, finally, the conclusions are drawn in Section 6.

2 QRS approximation by means of Hermite polynomials

The aim of using the Hermite approximation to estimate beats is to reduce the number of dimensions required to carry out the ECG classification, without sacrificing accuracy. The benchmarks used in this work come from the MIT-BIH arrhythmia database [10] which is made up of 48 ECG recordings whose beats have been manually annotated by at least two cardiologists. Each file from the database contains 2 ECG channels, sampled at a frequency of 360 Hz and with a duration of approximately 2000 beats. In particular, here we are addressing the characterization of the morphology of the QRS complexes since this morphology, together with the distance between each pair of consecutive beats, permits the identification of the majority of arrhythmias.

Firstly, the ECG files are preprocessed to remove baseline drift and high frequency noise. Secondly, the QRS complexes for each beat are extracted by finding the peak of the beat (e.g. the R wave) and selecting a window of 200 ms centered on the beat. Given that all the Hermite functions converge to zero both in $t = \infty$ and $t = -\infty$, the original QRS signal is extended to 400 ms by adding 100-ms sequences of zeros at each side of the complex. Thus, the QRS data are stored in a 144-sample vector $\boldsymbol{x} = \{x(t)\}$. This vector can be estimated with a linear combination of N Hermite basis functions

$$\hat{x}(t) = \sum_{n=0}^{N-1} c_n(\sigma)\phi_n(t,\sigma), \tag{1}$$

with

$$\phi_n(t,\sigma) = \frac{1}{\sqrt{\sigma 2^n n! \sqrt{\pi}}} e^{-t^2/2\sigma^2} H_n(t/\sigma) \tag{2}$$

being $H(t/\sigma)$ the Hermite polynomials. These polynomials can be computed recursively as $H_n(x) = 2xH_{n-1}(x) - 2(n-1)H_{n-2}(x)$, where $H_0(x) = 1$ and $H_1(x) = 2x$. The σ parameter controls the width of the polynomials. In [1] the

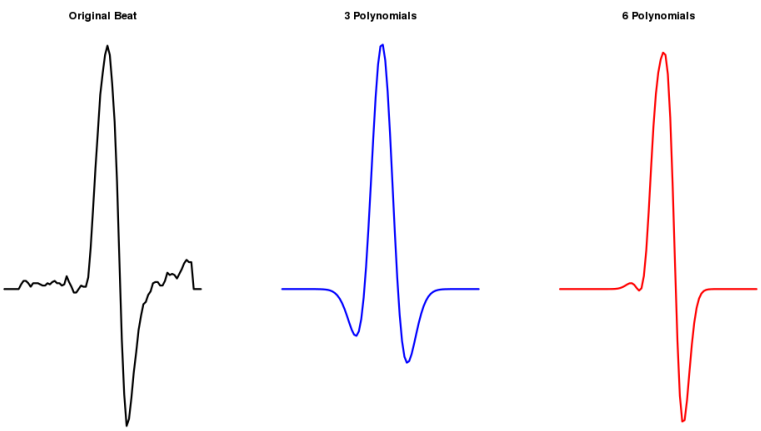

Fig. 1. Representation of heartbeat: left – original beat; center – 3-polynomial representation; right – 6-polynomial representation.

maximum value of σ for a given order n is computed and it holds that the bigger n is the smaller σ_{MAX} becomes.

The optimal coefficients that minimize the estimation error for a given σ are

$$c_n(\sigma) = \sum_t x(t) \cdot \phi_n(t, \sigma) \text{ [1]}. \tag{3}$$

Once the suitable set of σ and $\boldsymbol{c} = \{c_n(\sigma)\}$ $(n \in [0, N-1])$ are found for each beat, it is possible to use only these parameters to reconstruct the beat (see Fig. 1) and, thus, to perform the morphological classification of the beats [1, 4].

3 GPU acceleration

GPUs enable the massive parallelization of algorithms and they reach speedups ranging from 10× to 300× [7] keeping a low power consumption [11]. Internally, they are formed of hundreds of processor cores that work in parallel, executing the same task (*kernel*). They have been welcomed by the scientific community due to their low cost, their relatively programming simplicity and their suitability for floating-point computations – widely adopted in scientific computation. They have been applied to many disciplines, being well accepted among bioengineering research projects [6, 7]. Currently, the most popular GPUs are connected to the PC by means of a PCIe connection, opening the door to low cost high-performance computing. Basically, they are tuned for executing the same task using a huge volume of data. If we move apart from this situation (data dependency, conditional flows, etc.) they do not provide perceptible performance gains. C-like programming languages, such as CUDA [6] (Compute Unified Device Architecture), can be used to program the GPUs. These languages enable

parallel programming and they provide fast compilation and easy integration with traditional programs executed by the CPU. In this work, CUDA was selected since it is specially designed for the GPU devices chosen in this work for the acceleration (Nvidia GPUs).

The programming model of CUDA is intended for encapsulating the inner hardware details of the GPU to the programmer, in order to ease the development process, as well as to facilitate portability to different GPU devices. The GPU is composed of several streaming processors (SP) that possess several cores that can work in parallel. As previously mentioned, the GPU executes the same piece of code (kernel) in parallel using different data sets. A *thread* is a particular execution of the kernel. Each SP handles in parallel a set of *threads* grouped together in the so-called *warps*. The execution of the threads in a warp is parallel as long as there are no conditional branches. If there are different execution paths, the SP executes in parallel all threads that point at the same instruction. This implies, that the SP must first cluster all threads that are in the same execution point, then, execute sequentially each cluster. Thus, the presence of conditional branches can deteriorate performance considerably.

The programmer has some control on the way that threads work in parallel. Threads are grouped in *blocks* using a 1D, 2D, or 3D mesh. As a result, each thread has a 3-dimension identifier (ID). During scheduling, each block is assigned to an SP, and the SP starts the execution of all of its threads (by means of warps). In a similar fashion, blocks are distributed in a 1D/2D mesh, called a *grid*. Thus, the block also has an identifier, and this identifier is visible to the threads belonging to it. Each thread can use the block and thread IDs to generate the memory locations of the data sets that they have to access.

Regarding memory, all threads can access *global memory* (DRAM), all threads within a block access *shared memory* (SRAM), and each individual thread accesses a set of *local registers*. The key point here is that global memory has a high capacity (i.e. 1-6 GB) but it is slow, while shared memory has a small capacity (i.e. 16-48 KB) but it is fast (a couple of orders of magnitude faster than global memory). Global memory must be accessed coalescedly, since the read and write operations work with several consecutive bytes (32, 64, 128, etc.), otherwise, there are prohibitive delays. Shared memory can be accessed randomly.

As a final remark, given that an algorithm is suitable for parallelization, the key to success in acceleration with a GPU are both the wise selection of the block and grid shapes and sizes, and a correct use of the memory hierarchy.

4 CUDA implementation

4.1 Baseline implementation

Algorithm 1 shows the computations involved in the characterization of the ECG beats. The inputs to the algorithm are the ECG data and the maximum polynomial order N, and the output is the set with the $N + 1$ parameters used to characterize the QRS complexes.

Algorithm 1 QRS characterization

Input: ECG data, maximum polynomial order N
Output: Best set of parameters for each beat ($\{\sigma, \boldsymbol{c}\}$)

```
1:  # Loop 1
2:  for all σ and n do
3:     Compute φ_n(t, σ) (2)
4:  end for
5:  Extract QRS complexes from the ECG file (x_i(t))
6:  # Loop 2
7:  err_min = ∞
8:  for all x_i(t) do
9:     for all σ do
10:       for all n do
11:          Compute c_n(σ) # eqn. (3)
12:       end for
13:       Compute x̂_i(t) # eqn. (1)
14:       Compute err = MSE(x̂_i(t), x_i(t)) # eqn. (4)
15:       if err_min > err then
16:          σ_BEST = σ; c_BEST = {c_n(σ)}
17:       end if
18:    end for
19: end for
```

First, the QRS complexes are extracted from the ECG recording, outputting a 144-sample signal $x_i(t)$ for each beat. Then, in the first loop (*Loop1*, lines 2-4), the values of $\phi(t, \sigma)$ are precomputed, aiming at reducing computation time, since these values are used repeatedly during the second loop. The benefit of this precomputation was tested in an Intel i7 leading to a speedup of $10^5\times$. The second loop (*Loop2*, lines 7-19) is devoted to finding the optimal set of σ and coefficients $\boldsymbol{c}$ for each beat. It is composed of two nested loops: the outer one traverses all x_i and the inner one finds the optimal coefficients for the set of σ. A total of S sigmas are tried from the set $\sigma = \{1 \dots \sigma_{max}(N)\}$, where $\sigma_{max}(n)$ is a function of n [1]. Typical values of S are smaller than 100. Thus, for each beat and for different values of σ, the optimal coefficients ($\boldsymbol{c}$) are found and the combination of σ and coefficients that reduces the mean squared error (MSE) between estimation $\hat{x_i}$ and the actual QRS complex x_i is selected to characterize the beat. The MSE is defined as

$$MSE = \sum (x_i(t) - \hat{x}_i(t))^2 \tag{4}$$

4.2 Parallel implementation

The approach taken is to parallelize *Loop1* and *Loop2* (Algorithm 1) using two different kernels: *kernel_φ* and *kernel_Hermite*. Algorithm 2 shows the way

Algorithm 2 Host-side code

Input: ECG data, polynomial order N
Output: Best set of parameters for each beat ($\{\sigma, \boldsymbol{c}\}$)

1: Allocate GPU memory
2: Call *kernel_ϕ*
3: Send all $x_i(t)$ to GPU (Write onto GPU Global memory)
4: Call *kernel_Hermite*
5: Wait for GPU to finish processing
6: Read $\{\sigma_i, \boldsymbol{c_i}\}$ (Write onto Host memory)

that the host (CPU) sends data to the GPU, calls the different kernels to do the processing, and finally, retrieves the data and stores them in the computer's RAM. It is worth noting that while the GPU is executing *kernel_phi* (line 2), the host can send data to the GPU at the same time (line 3). Also, before reading the results (line 6), it is necessary to synchronize with the GPU execution (line 5) to avoid reading inconsistent data. Following, these kernels are explained and also the way to optimize the data transfers for real-time processing as well as for the processing of very long ECG recordings.

Precomputation of $\boldsymbol{\phi}$

The parallelization of *Loop1* in algorithm 1 is straight forward. The Hermite functions $\phi_n(t, \sigma)$ are composed of 144 samples with disregard of the values of n and σ. Thus, we can have 144 threads working in parallel, so that each thread evaluates eqn. (2) at a different time step t. The value t is the same as the thread ID and the values of n and σ are derived from the block ID. Hence, a block contains 144 threads and deals with the computation of all the samples of a function ϕ for a concrete (n, σ) couple. Blocks are arranged in a $S \times N$ mesh. The 2-dimensional block ID has as a first component the index of σ and as a second one the weight of the Hermite polynomials. This scheme results in the parallel computation of as many ϕ functions as SPs are in the GPU. Fig. 2 displays the distribution of threads and blocks.

Search for the optimal coefficients

Loop2 in Algorithm 1 requires a more thorough parallelization. Now, each block is going to handle a different QRS complex $x_i(t)$, so there are as many blocks as beats in the ECG recording (i.e. approximately 2000 for MIT-BIH files). Each block holds 144 threads, since most of the time all the treads are able to work in parallel. A thread can use the block ID to select the beat to work on and the thread ID to know the index of the sample of the beat that is using for the computations.

The pseudocode for *kernel_Hermite* is in Algorithm 3. It must be borne in mind that the kernel is executed by all the threads in a block. The number of the beat (i) and the sample associated to the thread (t) are obtained in lines 1-2

- block(sigma, order of polynomial)
- thread(sample of Φ)

$$\Phi_n[m,\sigma] = \frac{e^{-m^2/2\sigma^2} H_n(m/\sigma)}{\sqrt{\sigma 2^n n! \sqrt{\pi}}}$$

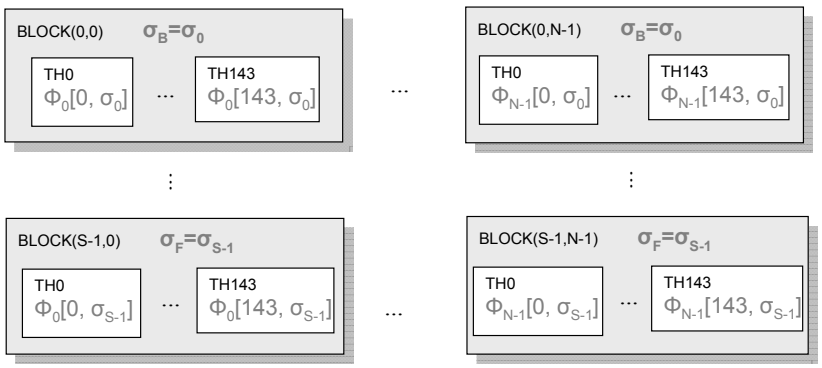

Fig. 2. Thread and block distribution for *kernel_ϕ*

from the block and threads IDs. Then, since the beat data are going to be used several times by the kernel and for each thread, these data are copied onto shared memory (line 3), so from now on, any reference to $x_i(t)$ implies a fast reading from shared memory. As in the original code (Algorithm 1), a loop traversing all values of σ is included (lines 5-21). For each σ, the vector of coefficients $\boldsymbol{c}$ is computed (lines 6-11) and the best one is kept. First, the multiplication between the original signal sample ($x_i(t)$) and the Hermite functions ($\phi_n(t,\sigma)$) are computed in parallel – each thread performs a multiplication on its own (lines 6-8). Then, a reduction technique is applied to carry out the summation [6] (lines 6-8). Unfortunately, it is unviable to perform this with full parallelism, so the performance is deteriorated. In order to compute the MSE, it is necessary to have the estimation of the beat for the current coefficients. Lines 11-14 shows how each thread iterates through the different polynomial orders, computing in parallel the multiplication of the coefficients by the original samples of the QRS complex. The MSE is computed in two steps. The squared error between $x_i(t)$ and $\hat{x}_i(t)$ is computed in parallel and then, the summation is performed by reduction (lines 15-16). Finally, thread 0 updates the best solution if the new MSE computed is the minimum so far.

Data transfer optimization

In the event that the amount of beats is too high that there is not enough memory in the GPU to store them, it is necessary to resort to divide the set of data in subsets that can then be computed sequentially. This idea can also be applied to real-time processing, and, in this case, the size of the subsets must be small (e.g. from 10 to 100 beats). Fig. 3 shows how it is possible to maximize performance by overlapping data transfer with GPU computation. During the computation of *kernel_ϕ* it is possible to also send the first subset of beats (*subset* 0). While *kernel_Hermite* is characterizing *subset* 0, *subset* 1 is being sent to the GPU. During the third kernel call, *subset* 1 is being characterized, *subset* 2 is being transfered to the GPU, and *subset* 0 is being transferred to the computer

Algorithm 3 Pseudocode for *kernel_Hermite* (executed at the GPU)

Input: ECG data, polynomial order N
Output: Best set of parameters for each beat ($\{\sigma, \boldsymbol{c}\}$)

```
1: i = block.ID            # beat index
2: t = thread.ID           # sample index
3: Copy x_i(t) to shared memory # full parallelization
4: err = ∞
5: for all σ do
6:    for all n do
7:       Compute sum_t = x_i(t) · φ_n(t, σ) # eqn. (3) – fully parallel
8:    end for
9:    for all n do
10:      Compute c_n(σ) = Σ sum_t # eqn. (3)) – reduction technique [6]
11:   end for x̂_i(t) = 0
12:   for all n do
13:      Compute x̂_i(t) += c_n(σ) · x_i(t) # eqn. (1)) – fully parallel
14:   end for
15:   Compute err_tmp(t) = (x_i(t) − x̂_i(t))^2 # eqn. (4) – fully parallel
16:   Compute MSE = Σ err_tmp(t) # eqn. (4) – reduction technique
17:   # This only for thread 0
18:   if t = 0 and err > MSE then
19:      σ_best = σ; c_best = c
20:   end if
21: end for
```

memory. The process continues for the rest of subsets. Thus, it is possible to compute in a pipeline fashion and performance is optimized.

5 Results

Algorithm 1 was coded in C language to be executed on a CPU and also in CUDA for the GPU execution. The test platform was a PC with an Intel-i7 (1,6 GHz and 4 GB of RAM) and graphics processor Nvidia TESLA C2050 (448 cores, 4 GB of RAM). The baseline was a single-thread execution of the CPU code. Three different tests were performed:

- **Test A: Off-line processing, short recordings.** It intends to assess the processing of short ECG recordings. The length of the recordings with which the text was carried out was 10, 100, 1000 and 2273 beats. Beats were represented both with Hermite polynomials of order 6 and 9.
- **Test B: Off-line processing, long recordings.** It intends to simulate the offline processing of long ECG recordings, such as Holter recordings. It uses streaming and divides the data in 200 blocks of 5000 beats. The orders of the Hermite polynomials were 6 and 9.

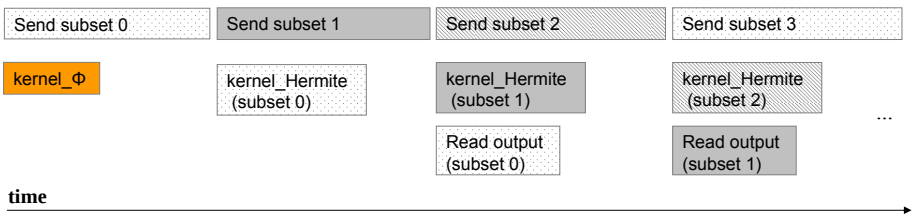

Fig. 3. Streaming processing to optimize performance for real-time processing and the processing of very long ECG recordings

Test C: On-line processing. It intends to simulate the online processing of ECG recordings, such as the real-time processing of the ECG of a patient admitted to a critical care unit. It uses streaming with blocks of short duration and divides the data in 1000 blocks of 10 and 100 beats (two different tests). The polynomial order were 6.

5.1 Off-line processing, short recordings

Table 1 shows the computation time and speedup for **Test A**. The first column indicates the number of beats processed. The second column the number of Hermite polynomials used. The third and fourth columns hold the computation time in ms of the baseline (CPU) and **Test A** (GPU). The last column shows the speedup.

Table 1. Performance results for **Test A**

Beats	N	CPU time (ms)	GPU time (ms)	Speedup
10	6	34	163	0.21×
	9	59	164	0.37×
100	6	175	163	1.11×
	9	258	165	1.56×
1000	6	1603	173	8.67×
	9	2204	169	11.28×
2273	6	3592	187	17.04×
	9	4922	198	23.19×

The results yield that the GPU does not provide a significant benefit for a small number of beats (e.g.$N < 1000$). The maximum speedup obtained is 23×, which might just be in the limit to justify the use of these devices, instead of using a multi-thread implementation with a standard CPU. Also, it is interesting to see that the GPU times for 10 and 100 beats are virtually the same, mainly

because the time needed to allocate GPU memory and to transfer data to the GPU are similar for both 10- and 100-beat blocks and much longer than the kernel computation time.

5.2 Off-line processing, long recordings

Test B is performed processing 200 blocks of 500 beats. Data transfer is optimized following the pipeline scheme from section 4.2. Table 2 shows the computation time and speedup obtained for $N = \{6, 9\}$.

Table 2. Performance results for **Test B**

Beats	Blocks	Beats/block	N	CPU time (msec)	GPU time (msec)	Speedup
10^6	200	5000	6	1986951	11542	171×
			9	2639662	11550	228×

The results show that the benefit of using a GPU for the processing of a high number of beats is significant, since speedups up to 228× are achieved. It is worth noting that the GPU computation times for $N = 6$ and $N = 9$ are virtually the same, while the CPU times increases 30%. Hence, the GPU enables increasing the accuracy of the beat estimation without increasing computation time. To put these results in perspective, for a Holter recording of 24 hours, and 6 leads, the characterization of the beats on the CPU would require approximately 25 minutes, while the GPU would need about 12 seconds.

The big difference between the speedups for **Test A** and **Test B** is due to the overlapping between GPU computation and data transfer carried out in the latter (see subsection 4.2). Even though the classification stage is not included in this study, the current results already show that for off-line processing the use of a GPU is a real asset.

5.3 On-line processing

Test C intends to assess the performance of the GPU for real-time processing. The computation performed and the data transfer scheme are the same as for **Test B**. The only differences are the size and number of blocks used. The size must be small to achieve real-time, and the number of blocks must be very high to simulate a continuous ECG processing. The maximum polynomial order selected was $N = 6$.

Table 3 contains the results for this third experiment. The first column indicates the number of blocks sent to the GPU (and also the number of kernel executions) and the second the number of beats conforming each block. The third column shows the time that a human heart takes to beat as many times as the number of beats processed, considering a heart rate of 60 beats per minute. The

next two columns hold the computation times for the CPU and GPU. Finally, column number six displays the speedup obtained with the GPU.

Table 3. Performance results for **Test C**

Blocks	Beats/Block	Heart time (sec)	CPU time (ms)	GPU time (ms)	Speedup
1000	10	10^4	153083	5474	28.96×
	100	10^5	1582625	15677	102×

First, the results show that the CPU is able to perform real-time processing, since the computation time required to process a single heart beat is much shorter than the beat period (around 1 second). The GPU outperforms the CPU for blocks of both 10 and 100 beats, with speedups of 28.96× and 102×, respectively. For off-line processing, it was clear that a GPU reduces the time that the cardiologist needs to analyze a long ECG recording. As aforementioned, for a Holter recording of 24 hours, and 6 leads, the characterization of the beats on the CPU would require approximately 25 minutes, while the GPU would need about 12 seconds. In the case of real-time processing, both technologies (CPU and GPU) are able to work in real-time, leading to think that the use of a GPU is not justified. However, until the classification stage is not included in the experiment, it is not sensible to make such a statement. Since the GPU is working several orders of magnitude faster than the CPU, everything points at the possibility of applying more complex (and therefore more accurate) classification techniques on real time on the GPU than a CPU will be able to handle.

Let us point out that the speedup obtained for 10-beat blocks is much higher than the one obtained in Table 1, where there was no speedup at all. The reason for that is that the time required for the pipeline processing – that overlaps data transfer and kernel computation – along with the time for GPU memory allocation – that is performed only once – is negligible compared with the computation time of processing a thousand of beats.

6 Conclusions

In this paper, a solution for the GPU parallelization of the characterization of beats by means of Hermite functions was presented. The parallel code, based on CUDA, was explained in detail and performance results were presented. Speedups up to 200× where obtained for both off-line and on-line processing. Regarding the accuracy of the beat characterization, the GPU showed no performance degradation when the order of the Hermite polynomials was increased from 6 to 9, while the CPU computation time increased 30%. The GPU off-line processing of long ECG recordings enables reducing computation time of a 6 leads 24 hours Holter recording from approximately 25 minutes to about 12

seconds. As for on-line processing, both the CPU and GPU are able to work in real-time, although the GPU outperforms the former. It remains to study the impact of using a GPU when also the classification of beats is performed.

As future research lines, the authors propose: i) the addition of a classification stage [12], again for both off-line and on-line processing; and, ii) the assessment of GPU technology for higher orders of the Hermite polynomials representation.

Acknowledgments. We thank Nvidia University Program for the support given to the Laboratory of Bioengineering, University CEU-San Pablo. David G. Márquez is funded by an FPU Grant from the Spanish Ministry of Education (MEC) (Ref. AP2012-5053).

References

1. Lagerholm, M., Peterson, C., Braccini, G., Edenbr, L., Sörnmo, L.: Clustering ECG complexes using Hermite functions and self-organizing maps. IEEE Trans. Biomed. Eng **47** (2000) 838–848
2. Márquez, D.G., Otero, A., Félix, P., García, C.A.: On the Accuracy of Representing Heartbeats with Hermite Basis Functions. In Alvarez, S., Solé-Casals, J., Fred, A.L.N., Gamboa, H., eds.: BIOSIGNALS, SciTePress (2013) 338–341
3. Braccini, G., Edenbrandt, L., Lagerholm, M., Peterson, C., Rauer, O., Rittner, R., Sornmo, L.: Self-organizing maps and Hermite functions for classification of ECG complexes. In: Computers in Cardiology 1997. (1997) 425–428
4. Linh, T.H., Osowski, S., Stodolski, M.: On-line heart beat recognition using Hermite polynomials and neuro-fuzzy network. Instrumentation and Measurement, IEEE Transactions on **52**(4) (2003) 1224–1231
5. Linh, T.H., Osowski, S., Stodolski, M.: On-line heart beat recognition using Hermite polynomials and neuro-fuzzy network. Instrumentation and Measurement, IEEE Transactions on **52**(4) (2003) 1224–1231
6. Kirk, D.B., Hwu, W.m.W.: Programming Massively Parallel Processors: A Hands-on Approach. 1st edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2010)
7. Nickolls, J., Dally, W.: The GPU Computing Era. Micro, IEEE **30**(2) (2010) 56 –69
8. Garcia-Molla, V., Liberos, A., Vidal, A., Guillem, M., Millet, J., Gonzalez, A., Martinez-Zaldivar, F., Climent, A.: Adaptive step ODE algorithms for the 3D simulation of electric heart activity with graphics processing units. Computers in Biology and Medicine **44**(0) (2014) 15 – 26
9. Zhang, Q., García, J.M., Wang, J., Hou, T., Sánchez, H.E.P.: A GPU based Conformational Entropy Calculation Method. In Rojas, I., Guzman, F.M.O., eds.: IWBBIO, Copicentro Editorial (2013) 735–743
10. Moody, G.B., Mark, R.G.: The impact of the MIT-BIH arrhythmia database. Engineering in Medicine and Biology Magazine, IEEE **20**(3) (2001) 45–50
11. Brodtkorb, A., Dyken, C., Hagen, T., Hjelmervik, J., Storaasli, O.: State-of-the-Art in heterogeneous computing. ACM Trans. Des. Autom. Electron. Syst. **18**(1) (2010) 1–33
12. Barbakh, W., Fyfe, C.: Online Clustering Algorithms. Int. J. Neural Syst. **18**(3) (2008) 185–194