Analysis of parallel computing strategies to accelerate ultrasound imaging processes

D. Romero-Laorden, *Student Member, IEEE,* J. Villazón-Terrazas, *Student Member, IEEE,* O. Martínez-Graullera, *Member, IEEE,* A. Ibáñez, M. Parrilla and M. Santos Peñas

Abstract

This work analyses the use of parallel processing techniques in synthetic aperture ultrasonic imaging applications. In particular, the Total Focussing Method, which is a $O(N^2 \times P)$ problem, is studied. This work presents different parallelization strategies for multicore CPU and GPU architectures. The parallelization processes on both platforms are discussed and optimized in order to achieve real-time performance.

Index Terms

Ultrasound imaging, GPGPU, multicore, signal processing, parallel computing, CUDA

1 Introduction

Ultrasonic imaging is currently one of the most popular visualization methods to examine the interior of opaque objects[1]. It is mainly based on the pulse-echo response, where the propagation of an induced mechanical acoustic disturbance inside an object is analysed through the echoes caused by discontinuities in the material. Ultrasonic imaging is especially relevant for medical diagnosis, as well as in *non-destructive testing* (NDT)[2], [3].

Most ultrasonic imaging systems are based on transducers arrays. They manage hundreds of independent elements that yield multiple input data signals which are processed by different

• *D. Romero-Laorden, J. Villazón-Terrazas, O. Martínez-Graullera A. Ibáñez and M. Parrilla are with the Instituto de Tecnologías Físicas y de la Información (ITEFI), Spanish National Research Council (CSIC), C/Serrano 144, 28006, Madrid (Spain) - Email: david.romero@csic.es, M. Santos Peñas is with Dpto. Arquitectura de Computadores y Automática, Facultad de Informática, Universidad Complutense de Madrid*

This work has been supported by the Spanish Government under projects DPI2010-19376 and FIS2013-46829R.

beamforming techniques [4]. This is in general a computationally demanding task, so nearly all commercial systems have costly tailored hardware designs with parallel processing capabilities which can attain a high enough frame rate. Nevertheless, systems are pushed to the limit by real time applications of computationally intensive techniques such as the one under study: Synthetic Aperture Focusing Techniques (SAFT) with the Total Focussing Method (TFM). As multicore systems and General-Purpose Computing on Graphics Processing Units (GPGPU) have become more popular, parallel signal processing can nowadays be developed on these platforms at affordable prices. Several works on the use of conventional hardware for ultrasonic beamforming, such as CPU cluster [5] or GPU [6], [7], [8] have been published.

In SAFT, beamformig uses signals independently acquired from pairs of emission-reception array elements. For an array of N elements this set of signals can be composed by up to $N \times N$ signals, which is referred to as the Full Matrix Capture (FMC). As for the TFM, it performs beamforming at each image point with every signal in the FMC. This process produces high quality images with all its points focused both in emission and reception [9], but the computational cost is high. The TFM cost for P image points is $O(N^2 \times P)$ and this is why it is mostly only used in laboratories.

In recent years, some research has addressed how to implement FMC + TFM using Field Programmable Gate Arrays (FPGA)[10], [11], [12]. As a result, the company M2M has recently launched a real-time TFM system with 64 sensors (4096 signals FMC) which is capable of achieving up to 25 images per second for a 256×256 image [10].

The concept of real-time is closely linked to the specific particularities of the field of application. In NDT this is generally related to the response time since the transducer is placed until an image is obtained and the synchronization time with the movements of the sensor on the system under study. In this context, it is generally assumed that 25 img/sec is real-time (as in classic imaging systems). In medicine or in any field of application where there are moving elements, this time is determined by the speed of the changes occurring in the medium. The required frame rates may be much higher than 25 img/sec, specially for cardiac applications as Papadacci et. al. describe in [13]; ultrafast applications rendering up to 1000 img/sec [14].

Other studies have explored parallel computing strategies on CPUs and GPUs to address the same problem. Thus, CPU beamforming parallelization became a research field in 2000 as cluster computing turned widespread and gained importance [5] in improving 2D real-time, as well as 3D ultrasound imaging systems. Moreover, with the arrival of faster and more powerful PCs the requirements in dedicated hardware were reduced and replaced by software processing. J. A. Jensen et. al describe in [15] and [16] a Delay-and-Sum (DAS) library that implements specific functions for the beamforming. All these previous works were implemented for a single core of

CPU and thus they developed parallel versions of these libraries in order to exploit multicore CPUs. Munk et al. show in [17] toolboxes for beamforming computation over multi-core CPUs, where the time consumption is 142.3 seconds for a $300,000$ pixel TFM image from an array of 192 sensors ($36,864$ signals) running in a 16-core high standard workstation.

Initial works on GPGPU based implementations of SAFT focused on the use of minimum redundancy coarrays [18] for 128 array elements (255 signals) and 256 image points which rendered a frame rate of 30 TFM images per second in real time [19]. In succeeding works, the frame-rate increased to 150 images per second [20]. More recently some other research has explored using the FMC in post-processing but without the TFM. In [21], the FMC (16 emission and 64 receptor elements) is considered at a fixed focus; a frame rate of 40 img/sec for a 512×512 image (NVIDIA Fermi with 512 cores) is obtained. There are also some other works on the use of GPGPU to implement more complex beamforming techniques, such as Capon Beamforming [22].

More specific works on FMC + TFM can also be found. In [8], a GPGPU implementation (NVIDIA Fermi, 384 cores) obtains, for a 60×60 image, 188 img/sec (for a 16 element array), 117 img/sec (for a 32 element array) and 8 img/sec (for a 64 element array). Furthermore, the computational power of the GPU has been exploited to find a solution to the focus law correction when an irregular interface lies between the array and the region of interest. In this sense, in [23] a solution to time-efficient auto-focusing of unknown geometry through dual-layered media is presented. The procedure described in this paper it is able to obtain up to 30 frames per second (32 elements and with an image size of 240×80 pixels). Another solution to this problem, based on iterative numerical methods to solve the Fermat's principle, is presented in [24]. In this work the pre-calculated compensation is applied to the beamforming process in order to generate real-time images on a fixed position. In doing so, the computational cost is very high but, once the compensation is computed, a 20 img/sec frame rate is obtained for an array of 32 elements and a FMC + TFM image of 120×80 pixels. Finally, some works compare the capabilities of several technologies in terms of obtaining obtain real images. In [25] several systems are compared (Xeon X5690 with 6 cores, GPU NVIDIA Fermi with 512 and 448 cores, and GPU ATI 1536 cores) with a simulation application (CIVA) and different developing tools (OpenCL, CUDA). In this work, GPGPUs obtained up to 5 images per second for a 200×200 image with an array of 128 elements.

In line with these works, the present paper analyses real-time implementation of the TFM on multi-core CPUs and many-thread GPUs. In both cases, we started implementing the same basic algorithm which was then incrementally enhanced. This approach helped us identify bottlenecks. In next section, we review the theory behind the Total Focusing Method algorithm

and beamforming in SAFT systems. Next, a parallel beamformer design is carried out on both architectures. Finally, experimental results are presented along with a performance evaluation of both platforms and from which our conclusions are drawn.

2 TOTAL FOCUSSING METHOD

Figure 1 depicts the general procedure to compose an image with the FMC + TFM technique. It shows the linear transducer array of N elements, the Region of Interest (ROI) discretized in a grid of points $x[k,l]$, the final image $a[k,l]$, and an ideal reflector located at $\vec{x}_p$ ($x[k_p,l_p]$ in the discretized space).

Thus, assuming that e_i is the emitter transducer and e_j and e_{j+1} are two of the receptors transducers, the presence of a reflector at $\vec{x}_p$ introduces an echo at time $t_i + t_j$ for the signal $s_{i,j}(t)$ (echo-pulse response where i-index is the emitter element and the j-index is the receptor), and at time $t_i + t_{j+1}$ for the signal $s_{i,j+1}(t)$. Then, for any $x[k,l]$ space point, we can obtain from each signal $s_{i,j}(t)$ the echo response originated at that given point, and sum it all together to the image point $a[k,l]$. If this value becomes significant compared to other image points, we can confirm the presence of a reflector on it.

2.1 The beamformer

Giving this description and assuming that the acquisition system has produced the FMC of sampled signals, a generic signal can be described as:

$$s_{i,j}[n] = s_{i,j}(t) \cdot \delta(t - n\tau_s) \tag{1}$$

where $s_{i,j}[n]$ is the signal which corresponds to the e_i emitter and e_j receptor and $1/\tau_s$ is the sampling rate.

To obtain the echo response of a virtual reflector at $x[k,l]$ point, the flight time from e_i to e_j is:

$$t_{i,j}[k,l] = t_i[k,l] + t_j[k,l] - t_0 = \frac{|\vec{x}_i - x[k,l]| + |\vec{x}_j - x[k,l]|}{c} - t_0 \tag{2}$$

being c the medium velocity, $\vec{x}_i$ and $\vec{x}_j$ the locations of elements e_i and e_j, and t_0 the initial acquisition time. To fit this value in the sampled signal $s_{i,j}[n]$ they are mapped to memory factored as:

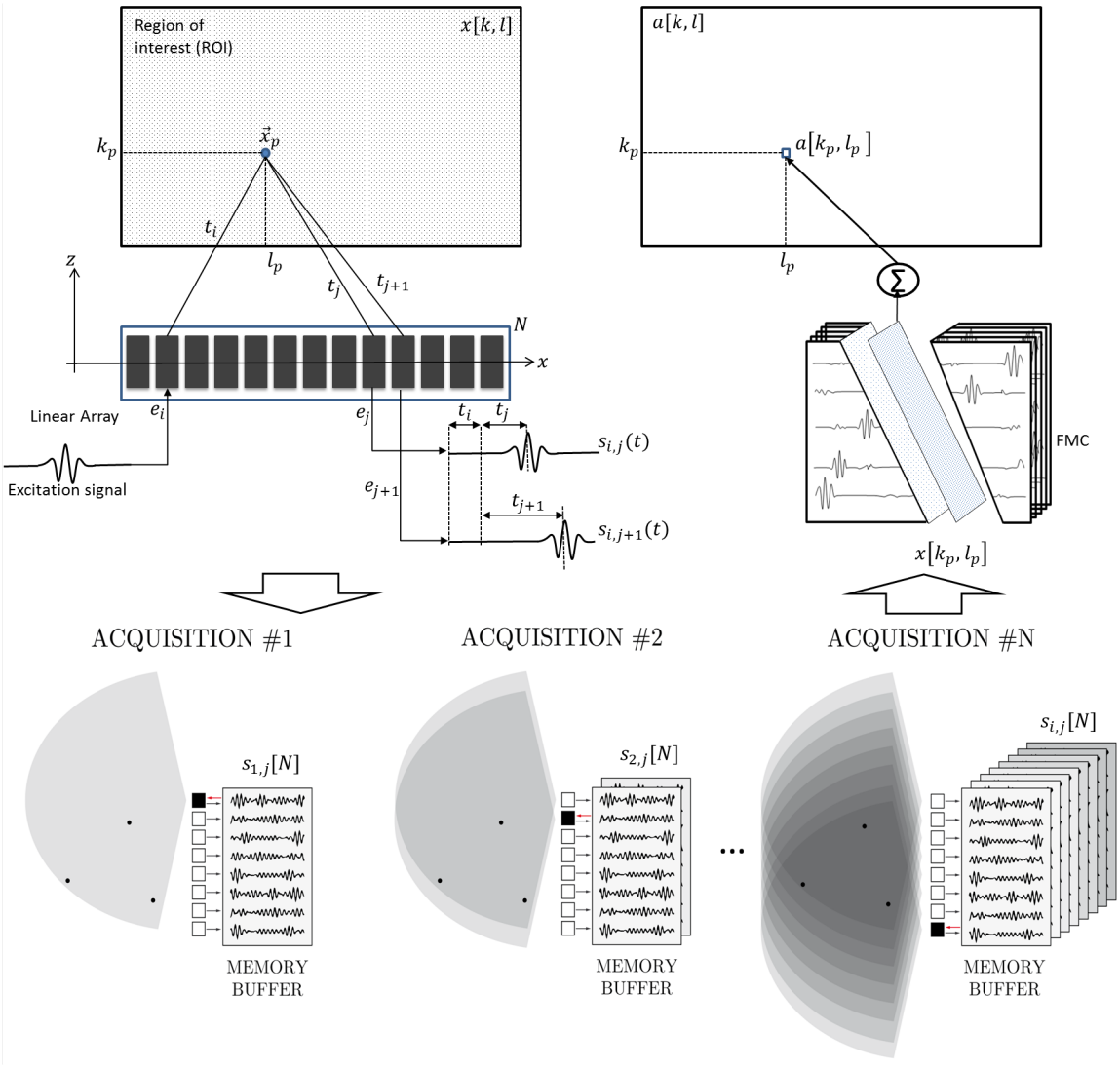


Fig. 1. Schematic diagram illustrating the TFM with a FMC acquisition procedure. A linear array of N elements, a reflector point $\vec{x}_p$ and the travel time to several array elements e_i, e_j and e_{j+1} from the reflector point are shown. The echo signals received at both elements is also presented. The FMC is composed through the acquisition process where a matrix of N received signal is obtained for each emitter. In order to focus the data at $x[k_p, l_p]$ a slice of data from the FMC can be selected and summed to compose the $a[k, l]$ value in the image.

$$m[i,j,k,l] = floor\left(\frac{t_{i,j}[k,l]}{\tau_s}\right) \tag{3}$$

$$\Delta m[i,j,k,l] = \frac{t_{i,j}[k,l]}{\tau_s} - floor\left(\frac{t_{i,j}[k,l]}{\tau_s}\right) \tag{4}$$

where $m[i,j,k,l]$ is an index value and $\Delta m[i,j,k,l]$ corresponds to an interpolation factor. So the echo response for the $x[k,l]$ point in the signal $s_{i,j}$ can be obtained by linear interpolation as:

$$S[i,j,k,l] = s_{i,j}[m[i,j,k,l]] + \Delta m[i,j,k,l]\,(s_{i,j}[m[i,j,k,l]+1] - s_{i,j}[m[i,j,k,l]]) \tag{5}$$

If all $s_{i,j}$ are considered the image value at the point of interest is obtained as:

$$a[k,l] = \sum_{i=1}^{N}\sum_{j=1}^{N} b_{i,j} S[i,j,k,l] \tag{6}$$

where $b_{i,j}$ are the coefficients of the spatial filter that is implemented by the array.

The beamformed points $a[k,l]$ are computed in radiofrequency. For this reason the image is rectified by envelope filtering so that it can be smoothed. This process can be done by means of the Hilbert Transform [26] which gives the analytic representation of $a[k,l]$:

$$a[k,l] \rightarrow a_I[k,l] + j a_Q[k,l] \tag{7}$$

where $a_I[k,l]$ and $a_Q[k,l]$ are the phase and quadrature components. Then the envelope can be computed as:

$$A[k,l] = \sqrt{a_I^2[k,l] + a_Q^2[k,l]} \tag{8}$$

and this value is used to determine the colour of each pixel on the image.

2.2 Practical implementation

Before studying the implementation of Equation 8 two practical issues need to be addressed.

The first relates to reducing the size of the FMC. This is done by means of the *half-matrix* [27], [28], [8] technique. Analysing flight times for two reciprocal signals, it is easy to observe that both signals are beamformed by the same time, that is $t_{i,j}[k,l] = t_{j,i}[k,l]$. Then, since $b_{i,j} = b_{j,i}$ it is unneccessary to keep both signals as we can build a new one $\hat{s}_{i,j}[n]$ as:

$$\hat{s}_{i,j}[n] = s_{i,j}[n] + s_{j,i}[n] \tag{9}$$

obtaining a new data set formed by $N(N+1)/2$ signals -almost a half of those in the FMC. If Equation 9 is performed by the system front-end or by the CPU in the GPU beamformer case, data movements are reduced and also beamforming operations which are halved.

The second relates to envelope computation. The Hilbert Transform may introduce artifacts on the border of the image [26] because of the periodicity condition in the FFT. In order to avoid those artifacts the analytic decomposition can be done in the acquired signals (now $\hat{s}_{i,j}[n]$). Then, distortion is produced in the extreme edge of the signals, usually far away from the ROI. If that was not the case the signals would be smoothed in the beamforming process. Then, in terms of data size, a complex data matrix is created and $s_{i,j}(t)$ can now be expressed by its analytic form:

$$\hat{s}_{i,j}[n] \rightarrow \hat{s}_{Ii,j}[n] + j\hat{s}_{Qi,j}[n] \tag{10}$$

Then the echo values at $x[k,l]$ point for each signal in the analytic decomposition is:

$$Iv_{i,j}[k,l] = \hat{s}_{Ii,j}[m_{i,j}[k,l]] + \Delta m_{i,j}[k,l]\,(\hat{s}_{Ii,j}[m_{i,j}[k,l]+1] - \hat{s}_{Ii,j}[m_{i,j}[k,l]]) \tag{11}$$

$$Qv_{i,j}[k,l] = \hat{s}_{Qi,j}[m_{i,j}[k,l]] + \Delta m_{i,j}[k,l]\,(\hat{s}_{Qi,j}[m_{i,j}[k,l]+1] - \hat{s}_{Qi,j}[m_{i,j}[k,l]]) \tag{12}$$

The ultrasonic image is now obtained as:

$$A_I[k,l] = \sum_{i=1}^{N}\sum_{j=i}^{N} b_{i,j} Iv_{i,j} \tag{13}$$

$$A_Q[k,l] = \sum_{i=1}^{N}\sum_{j=i}^{N} b_{i,j} Qv_{i,j} \tag{14}$$

$$A[k,l] = \sqrt{A_I[k,l]^2 + A_Q[k,l]^2} \tag{15}$$

2.3 The algorithm

The **Algorithm 1** describes the process and presents the first implementation (CPU1). As we can see, there are significant data-read operations, computational operations for the delays, conversion to indexes, indexations into the FMC matrix to recover signal values, interpolations, sum and write for pixel values and every combination of emitter and receptor, etc. Two parts of this process can be parallelized: the analytic decomposition (pre-processing) and the beamforming itself.

Algorithm 1 CPU Beamforming. CPU1

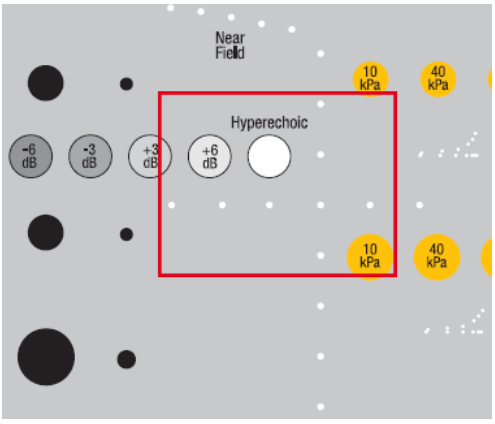
Pre-processing
$\hat{s}_{i,j}[n] \leftarrow s_{i,j}[n] + s_{j,i}[n]$ — Data reduction
$\hat{s}_{Ii,j}[n] + j\hat{s}_{Qi,j}[n] \leftarrow \hat{s}_{i,j}[n]$ — Analytic signal
Beamforming
for $k = 0$ **to** R_H **do**
  for $l = 0$ **to** R_V **do**
    $A_I = 0$, $A_Q = 0$ — Initialize pixel
    for $i = 1$ **to** N **do**
      for $j = i$ **to** N **do**
        m, Δm — Compute index
        if $0 \leq m \leq L$ **then**
          $I \leftarrow (1 - \Delta m)\hat{s}_{Ii,j}[m] + \Delta m\hat{s}_{Ii,j}[m+1]$ — Interpolated sample
          $Q \leftarrow (1 - \Delta m)\hat{s}_{Qi,j}[m] + \Delta m\hat{s}_{Qi,j}[m+1]$ — Interpolated sample
          $A_I \leftarrow A_I + b_{i,j}I$ — Multiply by $b_{i,j}$ and accumulate sum in A_I pixel
          $A_Q \leftarrow A_Q + b_{i,j}Q$ — Multiply by $b_{i,j}$ and accumulate sum in A_Q pixel
        end if
      end for
    end for
    $A[k,l] \leftarrow \sqrt{A_I^2 + A_Q^2}$ — Envelope calculation
  end for
end for
return A — Final image

3 EXPERIMENTAL SET-UP

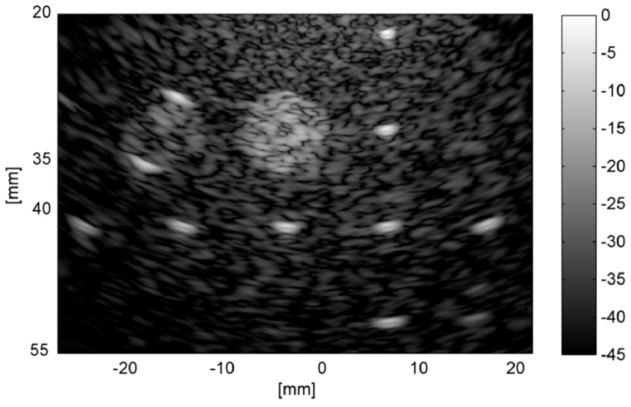
To illustrate the process a medical phantom (physical model which simulates the scattering and attenuation properties of biological tissue) is used. Specifically, $040GSE$ model by CIRS Inc. company (Figure 2(a)). In the figure we also delimit an area to compose our reference image (Figure 2(b)). The image is composed by three different image sizes: 256×256, 512×512 and 1024×1024.

The array transducer used is a PA 2.75/64-1093 model from VERMON company [29]. It is a linear array of 64 elements with an elementary pitch of $d = 0.28mm$. Each element emits a Gaussian pulse with a center frequency of $2.6MHz$ and a relative bandwidth of 65%. The acquisition equipment is a SITAU system [30] by DASEL S.L. company (www.daselsistemas.es). The data sets of $64 \times 64 \times 4096$ samples that form the FMC are acquired with a data resolution of 12 bits. The tables below show processing time but time for data transfer from the acquisition system is not been included.

We chose a variety of computing platforms as imaging systems (Table 1). Platform #1 is a laptop, platform #3 and #4 are low-level workstations and platform #2 and #5 are medium-level workstations. The GPUs of platforms #1, #2 and #3 are based on Fermi architecture [31] (GeForce 540M, Quadro 2000 and Quadro 4000 with 96, 192 and 256 cores respectively). Platforms #4



(a)



(b)

Fig. 2. 2(a) Multi-Tissue ultrasound phantom scenario 2(b) TFM image from Multi-Tissue ultrasound phantom.

and #5 are based on latest Nvidia's architecture, Kepler [31] (Quadro K2000 and Quadro K5000 equipped with 384 and 1536 cores each). In this work, we do not attempt to draw a comparison between the architectures. Rather we try to expand our experience base and give the reader a better evaluation of the results.

All platforms run Microsoft® Operating System Windows® 7. In all cases the FMC data and the output ultrasonic image are stored as single-precision floating-point numbers (4 bytes). We could have chosen more general tools, such as OpenCL, for our GPU development but used CUDA instead as it allows better adaptation to the specificities of the different platforms (which are all NVIDIA). Furthermore, CUDA provides libraries with highly optimized implementations, particularly faster speed of execution for FFT and provides tools that allow a more refined analysis and debug information on the occupancy of multi-processors and registers used by the algorithms.

TABLE 1
Imaging computing platforms. *DDR: Double Data Rate; GDDR: Graphics Double Data Rate.

Computing Platform	CPU & GPU	Memory*	# Cores
1	Intel Core i7 3632QM	DDR3, 6 Gbytes	8
	GeForce 540M (Fermi optimus)	GDDR3, 1 Gbytes	96
2	Intel Xeon E51650v2	DDR3, 32 Gbytes	12
	Quadro 2000 (Fermi)	GDDR5, 1 Gbytes	192
3	Intel Core 2 Quad Q9450	DDR3, 4 Gbytes	4
	Quadro 4000 (Fermi)	GDDR5, 2 Gbytes	256
4	Intel Core 2 Quad Q9450	DDR3, 4 Gbytes	4
	Quadro K2000 (Kepler)	GDDR5, 2 Gbytes	384
5	Intel Xeon E51650	DDR3, 8 Gbytes	12
	Quadro K5000 (Kepler)	GDDR5, 6 Gbytes	1536

We do include several platforms in our final analysis but for reasons of readability this section focuses on computing platform #3. This is a low-level workstation (Intel Core 2 Quad Q9450 with 4 cores) with a professional NVIDIA GPU (Quadro 4000 graphics card with 256 cores, the best of our Fermi boards).

4 Analytic Decomposition

The parallelization of the analytic decomposition and that of the FFT through the Hilbert transformation are closely linked. This is a very well studied process whose implementation has been optimized in several libraries. In this work, we implement it on the CPU and for that purpose we chose the FFTW3 [32] in float variabletype. In this case the parallelization is done creating as many threads as available cores, signals are then distributed across them. Single core implementation is computed as a reference.

In the GPU case, the Hilbert Transform uses the fastest FFT algorithm provided by CUFFT libraries [31]. Furthermore, CUDA provides fast intrinsic maths routines which provide better performance at the price of IEEE compliance. In our case, since our data is 12 bit integer type and current converters handle up to 16 bits, the use of these routines makes no significant numerical difference that might have an impact on the quality of the final output ultrasound image. On the other hand, performance is increased by 150% and for that reason we decided to use them throughout all the algorithms implemented for the GPU.

Table 2 shows the computing time of these analytic decomposition based on the three FFT implementation in all platforms. Although these times are not indicative of the final performance,

TABLE 2
Computing times in x64 OS measured in seconds for the FFT in CPU and GPU over the all platforms.

Computing Platform	FFT Single CPU	FFT Multi CPU	FFT GPU
# 1	0.234	0.056	0.016
# 2	0.185	0.025	0.011
# 3	0.312	0.081	0.006
# 4	0.324	0.095	0.009
# 5	0.187	0.026	0.003

they show the upper limits that each platform can achieve.

The improvement in the CPUs due to the parallelization is clearly shown; the maximum frame rate increases from 5 to 40 images per second. For those platforms that have four cores (#3 and #4) the speed-up due to parallelization is almost 4 times, close to the number of cores. However, platform #1 (increasing from 1 to 8 cores) shows a similar speed-up than that obtained for platforms #3 and #4 (4 cores); and for 12 cores (platforms #2, #5) the improvement achieved is only 7.4 times faster.

In the case of GPU, analytic decomposition limits the frame rate to 333 in the best platform (#5, Kepler) and to 65 in the worst case (#1, Fermi Optimus). As it was expected, better results are obtained by GPU platforms, that are at least 8 times faster than the CPUs. In the tables below we include the cost of this FFT calculation.

5 CPU BEAMFORMING

The implementation of the beamforming algorithm on a multicore CPU is based on a common multi-threading scheme as described in Figure 3. This is a *pixel-oriented* parallelization where each CPU thread is responsible for calculating the value of a set of pixels. Thus, image pixels are divided by the number of CPU cores N_{CPU} giving a total of N_{SP} pixels per CPU thread, where:

$$N_{SP} = \frac{N_H \times N_V}{N_{CPU}} \tag{16}$$

N_H and N_V are the number of pixels in both dimensions of the image ($N_I = N_H \times N_V$).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2544312, IEEE Transactions on Parallel and Distributed Systems

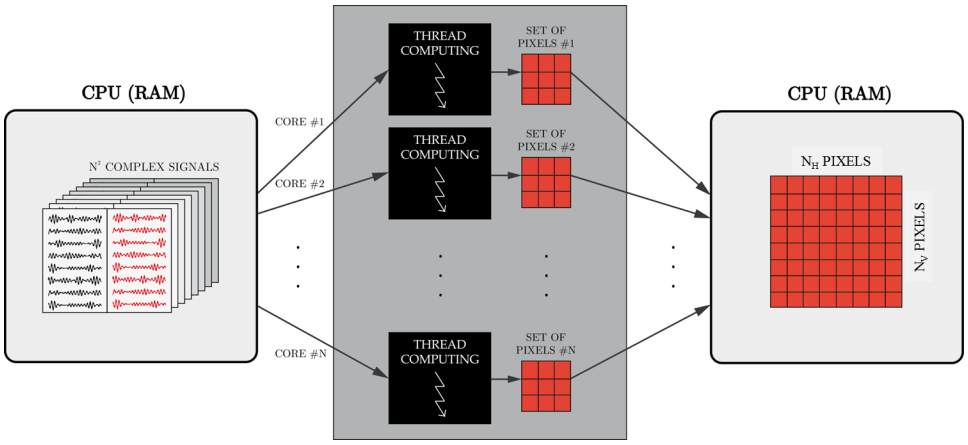


Fig. 3. CPU multi-threading implementation scheme.

5.1 First implementation

The first implementation follows **Algorithm 1**. That means that each pixel $[k, l]$ of the image can be solved by one isolated thread that iterates over all the array elements, jumping from signal to signal over the FMC structure following the sample index provided by $m[i, j, k, l]$. Two separate variables are maintained to beamform *in phase* and *in quadrature* values that are finally used to obtain $A[k, l]$.

Time consumption (Table 3) for different images sizes, 256×256, 512×512 and 1024×1024 pixels, is 3.638, 8.68 and 23.262 seconds respectively; which is far from real time requirements.

The main advantage of this process is that each pixel is solved using a register-oriented implementation. In this way we can avoid writing temporary results to memory. Its main drawback is that read memory accesses to data are not coalesced because sample values are retrieved from non-contiguous memory spaces. This fact is considered to be the main reason for this poor performance.

5.2 Second implementation

The solution to this problem is to change the direction of the DAS beamformation process. Thus, the second implementation first iterates over the array elements and then over every image pixel as shown in **Implementation 2**. Each core solves different sets of N_{SP} points of the image.

Computational costs in the different implementations are compared in Table 3. In CPU2 the computing time is reduced by half when compared to CPU1. CPU2 times for the different image sizes are 0.869, 4.576 and 12.062 seconds respectively. Nevertheles they stand in the same relation to the number of image points as in CPU1. Now each $A_I[k, l]$ and each $A_Q[k, l]$ needs

Implementation 2 CPU Beamforming. CPU2/CPU2SSE	Notes
$A_I[k,l]=0$, $A_Q[k,l]=0$	Initialize subimage region
for $i=1$ **to** N **do**	
for $j=i$ **to** N **do**	
for $[k,l]=0$ **to** N_{SP} **do**	
m, Δm	$\_mm\_loadu\_ps, \_mm\_sqrt\_ps, \_mm\_mul\_ps, \_mm\_add\_ps$
if $0 \leq m \leq L$ **then**	
$I \leftarrow (1-\Delta m)\hat{s}_{Ii,j}[m] + \Delta m \hat{s}_{Ii,j}[m+1]$	$\_mm\_loadu\_ps, \_mm\_mul\_ps, \_mm\_add\_ps$
$Q \leftarrow (1-\Delta m)\hat{s}_{Qi,j}[m] + \Delta m \hat{s}_{Qi,j}[m+1]$	$\_mm\_loadu\_ps, \_mm\_mul\_ps, \_mm\_add\_ps$
$A_I[k,l] \leftarrow A_I[k,l] + b_{i,j} I$	$\_mm\_loadu\_ps, \_mm\_mul\_ps, \_mm\_add\_ps$
$A_Q[k,l] \leftarrow A_Q[k,l] + b_{i,j} Q$	$\_mm\_loadu\_ps, \_mm\_mul\_ps, \_mm\_add\_ps$
end if	
end for	
end for	
end for	
$A[k,l] = \sqrt{A_I[k,l]^2 + A_Q[k,l]^2}$	$\_mm\_loadu\_ps, \_mm\_sqrt\_ps, \_mm\_mul\_ps, \_mm\_add\_ps$
return $A[k,l]$	Final subimage region

to be written and read several times but the cost is less than in the direct implementation of **Algorithm 1**. This is because we can benefit now from cache memory and the spatial arrangement which substantially improve computational time.

In order to make a faster implementation, we considered using *Streaming SIMD Extensions 2* (SSE2) [33]. Thus an optimized version of CPU2, was designed integrating SSE2 instructions. In CPU2SSE, on every cycle the operation register works with the *in phase* and *in quadrature* components of two pixels at the same time. The instructions for this version can be found on the right side of **Implementation 2**.

TABLE 3
Parallelization of TFM algorithm running on Platform #3. CPU strategies comparison for different image sizes.

#3	256×256	512×512	1024×1024
CPU1	3.63	8.68	23.26
CPU2	0.86	4.57	12.06
CPU2SSE	0.46	2.245	5.95

With SSE instructions results are drastically improved. CPU2SSE attains 0.469, 2.245 and 5.95 seconds respectively - that is ≈ 5 times faster than implementation CPU1.

5.3 Third implementation

Another consideration can be made to improve performance. If our platform has enough memory resources and the region of interest is well defined, time-delay computation and index transformation can be done previously and stored as a table rather than computed on-the-fly. The total space needed is $N_V \times N_H \times N^2$. In the case of large images this might exceed memory capacity. That is the reason why it has only been tested on platform #2 (equipped with 32GB memory). This new implementation is referenced as CPU3SSE and it is equivalent to **Implementation 2**. The only difference with CPU2SSE is that m and Δm are now read from a table using $\_mm\_loadu\_ps$ instruction. CPU1, CPU2 and CPU2SSE have also been computed so as to obtain a measurement of performance.

Results are presented in Table 4. The most significant result is that CPU3SSE is able to produce more than 13 images per second in the 256 case - a 168% increase in performance compared to CPU2SSE. In the 512 case, there is a 140% increase. However in the 1024 case there is only about an 8% improvement. This decline is due to the increase in the number of transactions with the memory that use a maximum of 4 access channels to serve the 12 threads.

#2	256×256	512×512	1024×1024
CPU1	0.42	1.27	4.75
CPU2	0.26	0.95	3.64
CPU2SSE	0.12	0.36	1.35
CPU3SSE	0.07	0.26	1.24

TABLE 4
Parallelization of TFM algorithm running on Platform #2. CPU strategies comparison for different image sizes.

6 GPU BEAMFORMING

The GPGPU development involves a paradigm shift caused by the many-thread SIMD model of GPUs. Furthermore, the integration of different memory accesses allows alternative implementations for **Algorithm 1** which are not feasible with the CPU. Additionally, GPUs memory capacity is more limited than that of CPUs and delay precomputation would only be possible for small images and for that reason it is not considered for this platform.

6.1 First implementation

The starting point to improve the beamforming process performance is **Algorithm 1**. The **Implementation 3** shows how it can be adapted to the GPU architecture. GPU1 is carried

out by launching a thread per image pixel. To this end, a computational grid with $B_X = \lceil \frac{N_H}{T_{B_X}} \rceil$ and $B_Y = N_V$ blocks of T_{B_X} threads is defined to launch the kernel. This implementation keeps all intermediate results in registers of the scalar processors and thus write to memory operations are avoided.

Implementation 3 GPU1 kernel	GPU optimization resources and notes. GPU1op
$s_{Ii,j}[n], s_{Qi,j}[n]$	Signal stored in textures memory
$x[k,l]$	Space coordinates stored in textures memory
$b_{i,j}$	Spatial filter stored in constant memory
x_i	Sensor coordinates stored in shared memory
$k \leftarrow threadIdx.x + blockIdx.x * blockDim.x$	Calculate k coordinate by thread index
$l \leftarrow blockIdx.y$	Calculate l coordinate by block index
1: **for** $i = 1$ **to** N **do**	
2: **for** $j = i$ **to** N **do**	
3: $\hat{m} = (t_j[k,l] + t_i[k,l] - t_0)/\tau_s$	Compute interpolation factor
4: **if** $0 \leq m(x,z) \leq L$ **then**	
5: $I_v \leftarrow texture\{\hat{s}_{Ii,j}, \hat{m}\}$	Interpolation by GPU texture hardware
6: $Q_v \leftarrow texture\{\hat{s}_{Qi,j}, \hat{m}\}$	Interpolation by GPU texture hardware
7: $A_I \leftarrow A_I + b_{i,j} I_v$	
8: $A_Q \leftarrow A_Q + b_{i,j} Q_v$	
9: **end if**	
10: **end for**	
11: **end for**	
12: $A[k,l] \leftarrow \sqrt{A_I^2 + A_Q^2}$	
13: **return** $A[k,l]$	

Computing times for GPU1 implementation, where there is no resource optimization, are 0.325, 0.68 and 1.95 seconds for 256×256, 512×512 and 1024×1024 pixels images respectively. It shows non-coalescing data reads and it does not maximize multiprocessors occupancy, nevertheless performance is better than in the CPU implementation.

Performance can be improved by optimizing the resources in the GPU. Once identified, the data elements of the process can be arranged across the different memory resources according to their use. Generally, this task can be approached with different mechanisms: shared memory which is small in size but has fast access speed as it is on-chip; texture memory which is cached and can be used for write and read operations ensuring all data reads are coalesced when it is defined as surface memory; and constant memory which is fast when all threads use it (broadcast) but can only be used for data read operations [31].

Bearing in mind these considerations, the first implementation can be optimized in GPU1op. In GPU1op the data matrix and space coordinates are stored in texture memory which is used as surface memory, whereas apodization values are stored in constant memory. Additionally,

the interpolation operation is no longer explicitly needed as texture mechanisms provide linear interpolations, at no cost, directly by hardware when the sample value is retrieved. All changes are shown in GPU1op (see **Implementation 3**). This data organization improves processing times, running times being 0.18, 0.36 and 1.10 seconds respectively for each image size.

6.2 Second implementation

The previous implementation gives good results. There is, however, a more suitable approach for the GPU model. This approach is based on the Nikolov SAFT implementation recommended for multiple FPGAs [11]. This implementation can be easily parallelized and optimized for a GPU.

The basic principle is simple. The data retrieved from all receptors corresponding to a single emitter can be used for making a low resolution image (LRI). Then a high resolution image (the final image) with full dynamic focusing at all points is obtained by combining (adding together) the N low resolution images. More memory is needed to store the partial results but the coalescence problem identified in the previous implementation is solved. The process is illustrated in Figure 4.

Therefore, GPU2 parallelization strategy is composed by two different kernels (see **Implementation 4**). The **Implementation 4 kernel one** is responsible for creating the low resolution image for each element. To this purpose, a thread per image pixel and array element is launched, and a three-dimensional computational grid is defined in the kernel, with $B_X = \lceil \frac{N_H}{T_{B_X}} \rceil$, $B_Y = \lceil \frac{N_V}{T_{B_Y}} \rceil$ and $B_Z = N$ blocks of $T_{B_X} \times T_{B_Y}$ threads in each dimension. It is important to remark that we are now creating 3D blocks. The block size must be equal to the number of elements in the array so as to cover each element on emission. Each thread within a block is in charge of calculating the partial sum of each emission-reception combination.

Once all low resolution images have been computed, the second kernel is in charge of combining all LRI images together (**Implementation 4 Kernel two**). To achieve this, a second grid with $B_X = \lceil \frac{N_H}{T_{B_X}} \rceil$ and $B_Y = \lceil \frac{N_V}{T_{B_Y}} \rceil$ blocks of $T_{B_X} \times T_{B_Y}$ is defined where each thread is responsible for calculating the sum for a given pixel across the multiple LRI images.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2544312, IEEE Transactions on Parallel and Distributed Systems

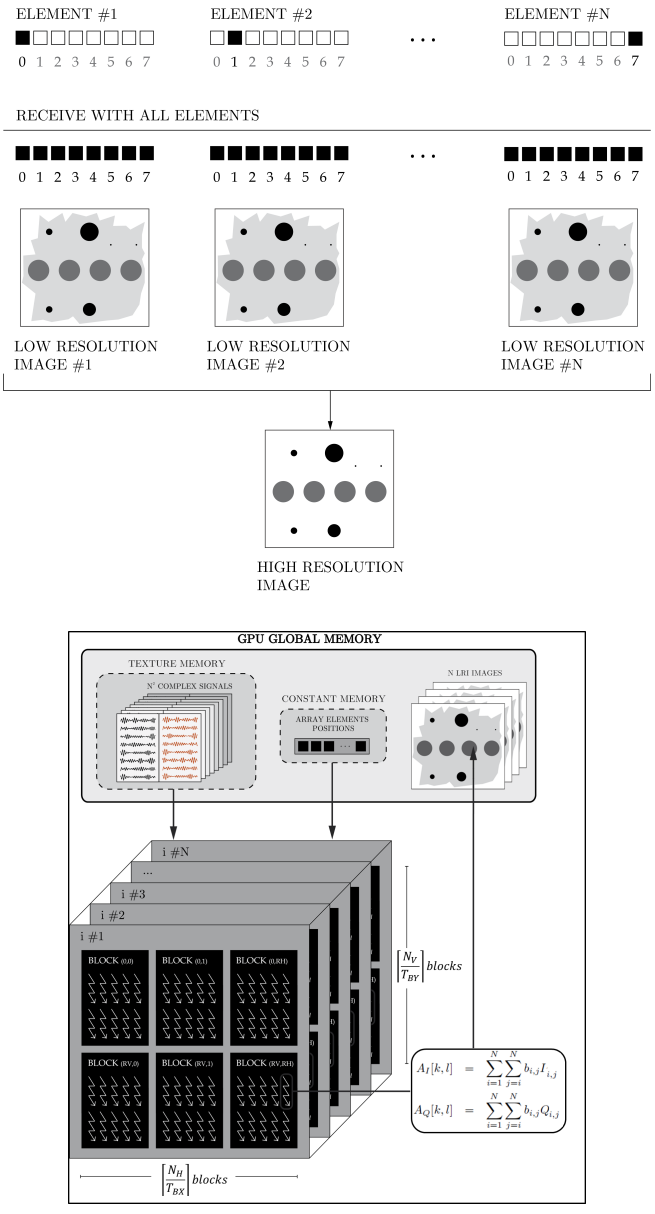


Fig. 4. Each combination of emitter and all receptors is used to create N LRI images which are combined to compose the final high resolution image. One thread is responsible for a pixel and an element of the array producing a set of low resolution images which belongs to each array element.

Implementation 4 GPU2 Kernel one

$s_{Ii,j}[t], s_{Qi,j}[t], x[k,l]$	Store in textures memory
$b_{i,j}$	Store in constant memory
$k \leftarrow threadIdx.x + blockIdx.x * blockDim.x$	Calculate k coordinate by thread/block index
$l \leftarrow threadIdx.y + blockIdx.y * blockDim.y$	Calculate l coordinate by thread/block index
$i \leftarrow threadIdx.z + blockIdx.z * blockDim.z$	Calculate i coordinate by thread/block index

1: $LRI_I[i,k,l] = 0$, $LRI_Q[i,k,l] = 0$ — Initialization
2: **for** $j = 1$ **to** N **do**
3: $\hat{m} = (t_j[k,l] + t_i[k,l] - t_0)/\tau_s$ — Compute interpolation factor
4: **if** $0 \leq m(x,z) \leq L$ **then**
5: $I_v \leftarrow texture\{\hat{s}_{Ii,j}, \hat{m}\}$ — Interpolation by GPU texture hardware
6: $Q_v \leftarrow texture\{\hat{s}_{Qi,j}, \hat{m}\}$ — Interpolation by GPU texture hardware
7: $LRI_I[i,k,l] \leftarrow LRI_I[i,k,l] + b_{i,j} I_v$ — Store as GPU texture
8: $LRI_Q[i,k,l] \leftarrow LRI_Q[i,k,l] + b_{i,j} Q_v$ — Store as GPU texture
9: **end if**
10: **end for**
11: **return** LRI_I, LRI_Q — Low resolution images

Implementation 4 GPU2 Kernel two

$k \leftarrow threadIdx.x + blockIdx.x * blockDim.x$	Calculate k coordinate by thread/block index
$l \leftarrow threadIdx.y + blockIdx.y * blockDim.y$	Calculate l coordinate by thread/block index

1: $A_I[k,l] = 0$, $A_Q[k,l] = 0$ — Initialization
2: **for** $i = 1$ **to** N **do**
3: $A_I[k,l] \leftarrow A_I[k,l] + texture\{LR_I[i,k,l]\}$ — Read from texture memory
4: $A_Q[k,l] \leftarrow A_Q[k,l] + texture\{LR_Q[i,k,l]\}$ — Read from texture memory
5: **end for**
6: $A[k,l] \leftarrow \sqrt{A_I[k,l]^2 + A_Q[k,l]^2}$
7: **return** A — Final image

Figure 5 shows the evolution in performance for platform #3 in all implementations. Resource optimization can double performance but the most significant increase however is obtained when the coalescence problem is solved. In fact, for the last implementation computing times are 0.028, 0.09 and 0.355 seconds for each image size. These results, depending on the image size, are between 6 and 3 times better than the ones previously obtained.

6.3 Analysis on the GPU

Our approach for the development of GPU implementations allows us to assess both strategies. The main grid and block dimensions were chosen by means of *Nvidia Occupancy Calculator* tool [34], [31], so as to attain the best performance. Thus, in the GPU1op case, the block-size was set to 256 threads (16×16 threads per block) which gives a grid of 32×32 blocks for an image of

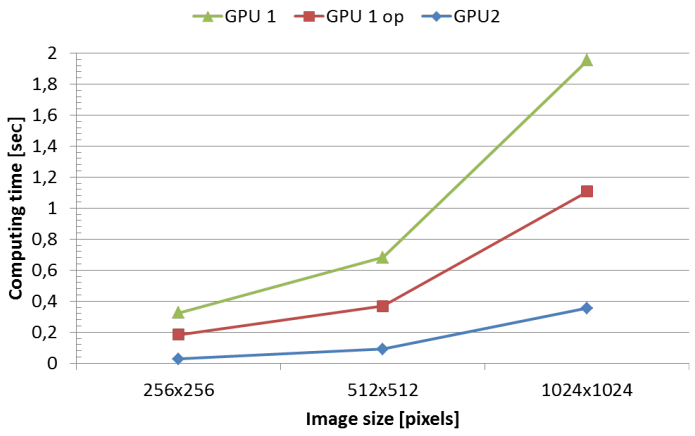


Fig. 5. Parallelization of TFM algorithm running on Platform #3. GPU strategies comparison for different image sizes.

256×256 pixels. For the GPU2 strategy optimum block-size was 512 threads ($8 \times 8 \times 8$ threads per block), grid dimensions being $32 \times 32 \times N$. Table 5 shows these configuration parameters.

The assessment enables us to see that both optimizations give a higher level of occupancy (over 90%) but also that the better use of memory resources and optimization in GPU2 give a better performance (close to 100%). Additionally in this implementation, the distribution in 3D blocks gets more benefit from spatial data locality in each multiprocessor and reduces the number of registers used by each thread. All these factors allow GPU2 to be twice as fast as GPU1op.

TABLE 5
Configuration parameters on GPU for the two strategies analysed. Image case of 256×256 pixels.

	GPU1op	GPU2
Kernel grid	$B_X = B_Y = 32,\ B_Z = 1$	$B_X = B_Y = 32,\ B_Z = N$
Kernel blocksize	$T_{BX} = T_{BY} = 16$	$T_{BX} = T_{BY} = T_{BZ} = 8$
Threads per block	256 threads	512 threads
Registers/Thread	18	12
Occupancy per SM	93.99%	99.24%

7 Performance Evaluation

In order to keep our comparative view on the platforms, we also consider implementations CPU2SSE and GPU2. The time needed for generating each image in each platform is shown in

Table 6. They were computed using an average of 16 measurements. It should be mentioned that GPU times also include copy time from CPU RAM to GPU RAM. The analysis of performance in the GPGPU algorithm with *NVIDIA profiler* showed that almost 40% of the computational cost is due to data transfer from the acquisition system to the GPU, via the CPU.

As it can be observed, in all cases the time consumed by the GPU is shorter than that required by the CPU. In terms of performance, the best CPU (platform #2) outperforms 9 times the best GPU (platform #5), and this is so for all image sizes. If different combinations between GPU and CPU platforms are considered, we see that moving the beamforming process from the CPU into the GPU can improve performance between 1.6 to 60 times, depending on the platform. This is also to show the diversity of platforms and the need for a previous analysis of the hardware capabilities before going into any optimization process.

TABLE 6
Computing times in x64 OS measured in seconds.

Platform		256×256	512×512	1024×1024
#1	CPU	0.29	1.02	3.94
	GPU	0.072	0.23	0.851
#2	CPU	0,13	0,37	1,35
	GPU	0.046	0.133	0.479
#3	CPU	0.45	2.25	5.95
	GPU	0.028	0.09	0.355
#4	CPU	0.49	2.35	5.99
	GPU	0.042	0.103	0.420
#5	CPU	0.15	0.51	1.72
	GPU	0.015	0.038	0.14

As for frame rates (Figure 6), CPU frame rates are up to 8 img/s (notice a 13 frame rate is attained with CPU3opt in #3), whereas all GPU-based systems rates are above 12 frames per second.

Platforms #3 GPU and #5 GPU need to be highlighted as they both generate 35 and 65 frames per second for 256×256 images in real time. As for CPUs, it is interesting to remark, that when platforms with the same number of cores are used (#2 CPU and #5 CPU) changes in the SSE instructions provide a more efficient improvement (Ivy vs Sandy Intel architectures). This fact is more noticeable as the size of the images increases.

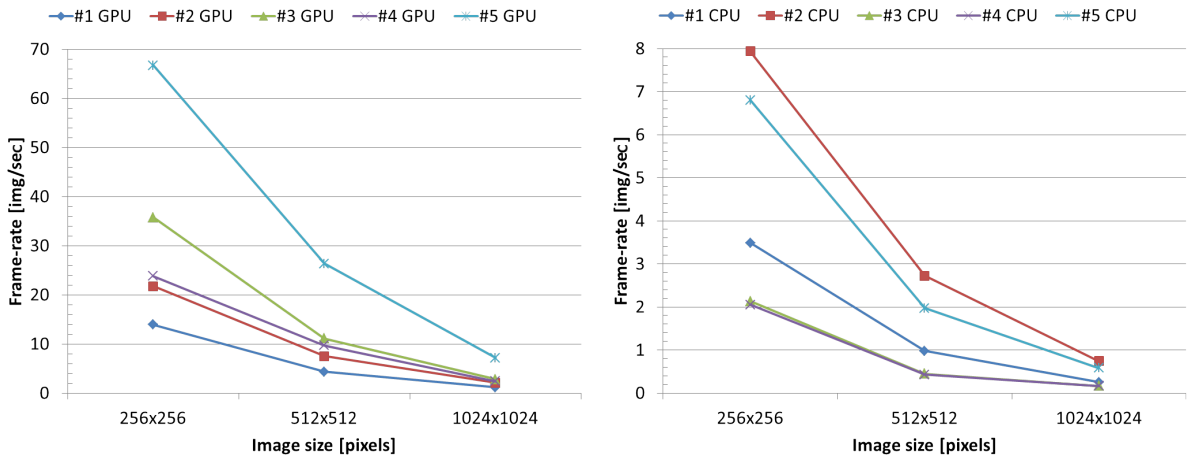


Fig. 6. Frame-rate in GPU an CPU.

8 Conclusions and Future Work

This work has focused on evaluating whether parallel computing techniques are mature for the implementation of a real time software beamforming system. We have studied CPU and GPU architectures, with the latter giving good performance more than 25 img/sec. We have developed in two stages the TFM algorithm and its execution has been parallelized.

The first stage has focused on obtaining the analytic signal. This operation is based on Fourier transform, which has a very well known parallelization and an optimized model with low computational costs on both architectures. The GPU case is especially significant, as the cost is two orders of magnitude below the one in the multicore implementation. The analytical signal procedure leads to an increment in the computational cost of the TFM imaging algorithm as two process flows need to be maintained. However this increase in computational cost is not excessive as both flows can share focal laws calculation. The main drawback is the decrease in multicore systems performance due to the fact that SSE instructions capacity to parallelize pixels is halved. Still we have maintained this approach because of the benefits from the quadrature and in-phase signals and their positive impact on image quality. Additionally artifacts caused by envelope's conversion are avoided and it is possible to perform more sophisticated beamforming processes, such as phase coherence imaging, that complement conventional DAS. In this sense, this is a line for future research.

The second stage has dealt with the beamforming process, which comprises two steps: focal law calculation and extracting and summing samples. In order to calculate focal laws on multicore systems two options have been considered. On the one hand, dynamic delays calculation,

which minimizes memory requirements of the algorithm. It has been implemented on all platforms considered. On the other hand, pre-calculation, which offers a significant increment in speed for small images. By contrast, its computational cost for large images becomes too high and performance is diminished. In the GPU case we have focused on dynamic calculation of focal laws whose cost is negligible compared to the cost of data memory transactions when these are precomputed. The analytical signal together with the non-use of filters and dynamic calculation of the focal law allows the algorithm to zoom the image in/out without affecting performance. This fact offers advantages in terms of algorithm usability.

In multicore systems, the extraction of samples and summation has been achieved by allowing each process to work on a subimage and resolving each signal sequentially. In GPUs this is done with a kernel, which for each emitter works over a single pixel, and creates separate images which are finally combined to form the final image. This system gives a good encapsulation of the memory space in each block. Thus, interference at the accesses is limited and the number of registers in the kernel is reduced in comparison with other optimization strategies.

Beyond the absolute values obtained in the different platforms, it is possible to conclude that GPU systems performance is in general an order of magnitude above that of multicore systems. In our view, this is not only because of the high number of computing cores as it is also mainly determined by memory accesses. The GPU provides more tools to optimize algorithms than the multicore model. However, we believe that the multicore model can also offer viable solutions if the requirements needed in the potential field of application are not very demanding.

Notwithstanding the above, we can also state that the main difficulty in the GPU is data transfer from the acquisition system into the GPU (via the CPU) which accounts for around 40% of the computational cost. In the NVIDIA case this obstacle can now be overcome by Kepler GPUDirect RDMA technology that connects to another device on a GPU placed on the same PCI bus. Our current instrumentation does not support that feature.

The expected technological progress for both CPU and GPU systems and the emergence of new technology integrating both platforms, lead us to believe that the development of beamformers on software is a very promising future line of research as it will provide new and increasingly sophisticated beamforming and, in turn, better quality of ultrasound imaging. Definitely, that will be key to new fields of application such as ultrafast imaging.

ACKNOWLEDGMENTS

This work has been supported by the Spanish Government under projects DPI2010-19376 and FIS2013-46829R.

REFERENCES

[1] M. Lauckner, *Manual of diagnostic ultrasound*, World Health Organization, Ed., 2011.

[2] T. L. Szabo, *Diagnostic Ultrasound Imaging: Inside Out*. Elsevier, 2004.

[3] J. Krautkramer and H. Krautkramer, *Ultrasonic Testing of Materials*, Springer Verlang, Ed., 1990.

[4] K. Thomenius, "Evolution of ultrasound beamformers," in *Ultrasonics Symposium, 1996. Proceedings., 1996 IEEE*, vol. 2, Nov 1996, pp. 1615–1622 vol.2.

[5] F. Zhang, A. Bilas, A. Dhanantwari, K. N. Plataniotis, R. Abiprojo, and S. Stergiopoulos, "Parallelization and Performance of 3D Ultrasound Imaging Beamforming Algorithms on Modern Clusters," in *Proceedings of the 16th international conference on Supercomputing*, 2002, pp. 294–304.

[6] C.-I. C. Nilsen and I. Hafizovic, "Digital beamforming using a GPU," in *IEEE International Conference on Acoustics, Speech and Signal Processing*. Ieee, Apr. 2009, pp. 609–612.

[7] H. K. H. So, J. Chen, B. Y. S. Yiu, and A. C. H. Yu, "Medical Ultrasound Imaging: To GPU or not to GPU," *IEEE Micro*, vol. 31, no. 5, pp. 54–65, 2011.

[8] M. Sutcliffe, M. Weston, B. Dutton, P. Charlton, and K. Donne, "Real-time full matrix capture for ultrasonic non-destructive testing with acceleration of post-processing through graphic hardware," *NDT & E International*, vol. 51, pp. 16–23, 2012.

[9] C. Holmes, Bruce W. Drinkwater, and P. D. Wilcox, "Post-processing of the full matrix of ultrasonic transmitreceive array data for non-destructive evaluation," *NDT & E International*, vol. 38, no. 8, pp. 701–711, Dec. 2005.

[10] "Gekko: Advanced phased-array ut by m2m." [Online]. Available: http://www.m2m-ndt.com/products/Gekko_features.htm

[11] S. I. Nikolov, "Synthetic aperture tissue and flow ultrasound imaging," Ph.D. dissertation, Technical University of Denmark, 2001.

[12] M. Birk, A. Guth, M. Zapf, M. Balzer, N. Ruiter, M. Hübner, and J. Becker, "Acceleration of image reconstruction in 3D ultrasound computer tomography: An evaluation of CPU, GPU and FPGA computing," in *Conference on Design and Architectures for Signal and Image Processing (DASIP)*, Tampere, 2011, pp. 1–8.

[13] C. Papadacci, M. Pernot, M. Couade, M. Fink, and M. Tanter, "High-contrast ultrafast imaging of the heart," *Ultrasonics, Ferroelectrics, and Frequency Control, IEEE Transactions on*, vol. 61, no. 2, pp. 288–301, February 2014.

[14] M. Tanter and M. Fink, "Ultrafast imaging in biomedical ultrasound," *IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency Control*, vol. 61, no. 1, pp. 102–19, Jan. 2014. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/24402899

[15] J. r. A. Jensen and S. I. Nikolov, "Fast simulation of ultrasound images," *2000 IEEE Ultrasonics Symposium. Proceedings. An International Symposium (Cat. No.00CH37121)*, vol. 2, pp. 1721–1724, 2000.

[16] J. Kortbek, S. I. Nikolov, and J. r. A. Jensen, "Effective and versatile software beamformation toolbox Jacob," in *Medical Imaging 2007: Ultrasonic Imaging and Signal Processing*, S. Y. Emelianov and S. A. McAleavey, Eds., Mar. 2007.

[17] J. M. Hansen, M. C. Hemmsen, and J. r. A. Jensen, "An object-oriented multi-threaded software beamformation toolbox," in *SPIE Medical Imaging: Ultrasonic Imaging, Tomography, and Therapy*, J. D'hooge and M. M. Doyley, Eds., Mar. 2011, pp. 79 680Y–79 680Y–9.

[18] C. J. Martín-Arguedas, "Técnicas de apertura sintética para la generación de imagen ultrasónica," Ph.D. dissertation, Universidad de Alcalá, 2010.

[19] D. Romero-Laorden, O. Martínez-Graullera, C. J. Martín-Arguedas, R. Tokio Higuti, and A. Octavio, "Using GPUs for beamforming acceleration on SAFT imaging," in *IEEE International Ultrasonics Symposium*. Rome, Italy: IEEE, 2009, pp. 1334–1337.

[20] D. Romero-Laorden, O. Martínez-Graullera, C. J. Martín-Arguedas, M. Pérez-Lopez, and L. Gómez-Ullate,

"Paralelización de los procesos de conformación de haz para la implementación del Total Focusing Method," in *12 Congreso Español de END*, Valencia, 2011.

[21] J. M. Hansen, D. Schaa, and J. r. A. Jensen, "Synthetic Aperture Beamformation using the GPU," in *IEEE International Ultrasonics Symposium*, Orlando, Florida, 2011.

[22] J. Asen, J. Buskenes, C.-I. C. Nilsen, A. Austeng, and S. Holm, "Implementing capon beamforming on a GPU for real-time cardiac ultrasound imaging." *IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency Control*, vol. 61, no. 1, pp. 76–85, Jan. 2014.

[23] S. M., W. M., C. I., and D. K., "Full matrix capture with time efficient auto focussing of unknown geometry through dual-layered media," *Insight. British Journal of NDT*, vol. 55-6, 2013.

[24] M. Weston, "Advanced Ultrasonic Digital Imaging and Signal Processing for Applications in the Field of Non-Destructive Testing," Ph.D. dissertation, University of Manchester, 2011.

[25] G. Rougeron, J. Lambert, E. Iakovleva, L. Lacassagne, and N. Dominguez, "Implementation of a GPU Accelerated Total Focusing Reconstruction Method within CIVA Software," *40th Annual Review of Progress in Quantitative Nondestructive Evaluation*, vol. 1581, no. 1, pp. 1983–1990, 2013.

[26] A. V. Oppenheim, W. R. Schafer, R. W. Schafer, and J. R. Buck, *Discrete-Time Signal Processing*, 2nd ed. Upper Saddle River, New Jersey: Prentice-Hall, 1989, vol. 23, no. 2.

[27] C. Holmes, B. W. Drinkwater, and P. D. Wilcox, "Advanced post-processing for scanned ultrasonic arrays: application to defect detection and classification in non-destructive evaluation," *Ultrasonics*, vol. 48, no. 6-7, pp. 636–42, Nov. 2008.

[28] A. J. Hunter, B. W. Drinkwater, and P. D. Wilcox, "The wavenumber algorithm for full-matrix imaging using an ultrasonic array." *IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency Control*, vol. 55, no. 11, pp. 2450–62, Nov. 2008.

[29] "Vermon: Manufacture composite piezoelectric transducers for medical and ndt applications." [Online]. Available: http://www.vermon.com/vermon/Medical_Products.php

[30] J. Camacho, O. Martinez, M. Parrilla, R. Mateos, and C. Fritsch, "A strict-time distributed architecture for digital beamforming of ultrasound signals," in *Intelligent Signal Processing, 2007. WISP 2007. IEEE International Symposium on*, Oct 2007, pp. 1–6.

[31] NVIDIA, *CUDA C Programming Guide 6.0*, 2014, no. February 2014. [Online]. Available: www.nvidia.com

[32] "Fftw website." [Online]. Available: www.fftw.org

[33] "Intel intrinsics guide - sse instructions." [Online]. Available: https://software.intel.com/sites/landingpage/IntrinsicsGuide/

[34] NVIDIA, "NVIDIA Developer CUDA zone," 2014. [Online]. Available: https://developer.nvidia.com/cuda-education-training

David Romero-Laorden was born in Talavera de la Reina, Toledo, Spain, in 1984. He received his M.S. degree in computer science from the Complutense University of Madrid, Spain, in 2008. In 2016 he received the PhD degree by the Universidad Complutense de Madrid. His research interests include ultrasonic imaging systems, synthetic aperture systems, digital signal processing and parallel programming on heterogeneous architectures.

Javier Villazon-Terrazas was born in Cochabamaba (Bolivia). He belongs to Ultrasonic Evaluation Group of the Applied Acoustic and Nondestructive Evaluation Department of the Spanish National Research Council (CSIC). He graduated with honours in Electronic Engineering from Universidad Del Valle, Bolivia in 2002. In 2015 he received the PhD degree in the System and Radio communications Department at Universidad Politécnica de Madrid. His research activities include Evolutionary Algorithms, Acoustic Field, Ultrasonic Array design and Computer Science.

Oscar Martínez-Graullera was born in Valencia, Spain. He received the Telecommunication Engineering degree in 1995 from the Polytechnic University of Valencia, and his Ph.D. degree from the Polytechnic University of Madrid in 2000. Since 2004, he is working at the Spanish National Research Council (CSIC, Madrid) as a Tenured Scientist. He is involved with ultrasonic imaging, ultrasonic arrays, digital signal processing, and real-time architectures.

Alberto Ibañez-Rodriguez was born in Benavente, Zamora (Spain). In 1984 he received his Degree in Physics from the University of Valladolid and his Ph.D. in Electronics from the Complutense University of Madrid (Spain). Since 1984, he is working at the Spanish National Research Council (CSIC, Madrid) as a Tenured Scientist. His research interests are in the field of integration of sensors, real-time systems, signal processing and its applications in nondestructive evaluation.

Montserrat Parrilla Romero was born in Ciudad Real, Spain. She received her B. Tech. and M.S. degrees in information technology form the Universidad Politcnica de Madrid in 1990 and 1992, respectively. She also received a Ph.D. degree in computer science from the same university at the Departamento de Arquitectura y Tecnologa de Sistemas Informticos (Computer Architecture and Information Technology Department) in 2004. Her research interests include 2-D and 3-D real-time ultrasonic imaging for clinical and industrial applications and automatic defect detection and characterization in ultrasonic nondestructive testing applications.

Matilde Santos Peñas was born in Madrid, Spain. She received her B.Sc. and M.Sc. degrees in Physics (Computer Engineering) and her Ph.D in Physics in 1994, from the University Complutense of Madrid (UCM). Since 1986 she has been with the Department of Computer Architecture and Systems Engineering at the UCM, where she is currently a Full Professor in System Engineering and Automatic Control. Her main research interests are: Intelligent Control, Signal Processing, Modelling and Simulation.