Multi-GPGPU Based Medical Image Processing in Hip Replacement

Anca Morar, Florica Moldoveanu, Victor Asavei, Alin Moldoveanu, Alexandru Egner

** The Faculty of Automatic Control and Computers, University "Politehnica" of Bucharest, Romania, (e-mail: anca.morar@cs.pub.ro, florica.moldoveanu@cs.pub.ro, victor.asavei@cs.pub.ro, alin.moldoveanu@cs.pub.ro, alexandru.egner@cs.pub.ro)*

Abstract: Image processing techniques are widely used in medical software applications. For instance, feature extraction methods such as Canny edge detector and Hough transform can be applied to radiographic images for the purpose of computing certain parameters. The large number of pixels in a radiographic image leads to slow computing times for such algorithms. GPGPU Implementations of these algorithms that reduce the computing times already exist. The novelty of the proposed approach is in taking advantage of the power of multi-GPGPU accelerated computers and other mechanisms of the CUDA architecture. The particularities of orthopedic radiographic images are also taken into account to further reduce the computing times. The new implementations were tested on a computer with two graphic cards and were compared to CPU and other GPGPU implementations. The comparison between the CPU and existing GPU implementations show that multi-GPGPU applications can add a significant performance gain to image processing applications.

Keywords: Image analysis, Image processing, Parallel algorithms, Multi-GPGPU

1. INTRODUCTION

In recent years, the use of GPU cards for other purposes other than rendering has increased considerably. The new GPGPU (General Purpose Computing on Graphics Processing Units) technology enables the implementation of various parallel algorithms that run a lot faster than on the CPU. Most of the image processing algorithms can be parallelized because, usually, the processing steps are the same for every pixel (in case of 2D images) or voxel (for 3D images). This is the case with the Canny edge detector and the Hough transform.

In the previous work by (Morar et al. (2010a, b)) a system that automatically identifies parameters important in Arthroplasty was designed by observing the similarities between different salient parts of bones in radiographic images and simple curves like lines or circles. The parameter extraction method gives quite accurate results, but is time consuming. This is the reason behind investigating the use of the GPGPU paradigm for the parallel implementation of certain feature extraction methods.

The remainder of this paper is organised as follows: the second section discusses the state of the art regarding image processing algorithms on the GPU and computed assisted analysis of radiographic images in the hip arthroplasy field. The third section presents the Canny edge detector optimised to run on multi-GPGPU computers. The forth section describes the Hough transform for lines, also implemented using a multi-GPGPU approach. The forth and the fifth section focus on methods that take advantage of the particularities of radiographic images in the hip arthroplasty field. Thus, the search space for the parameters of the Hough transforms for lines and circles can be considerably reduced. The sixth section shows a comparison of the new implementations with other existing implementations on the CPU and GPU. In the last section a series of conclusions are drawn, and some future research directions are presented.

2. STATE OF THE ART

2.1. Image processing with CUDA

According to (Nvidia Co. (2011a)), the programmable GPU has recently evolved into a parallel processor, with many threads and cores that determine an impressive computational power, as illustrated in Figure 1.

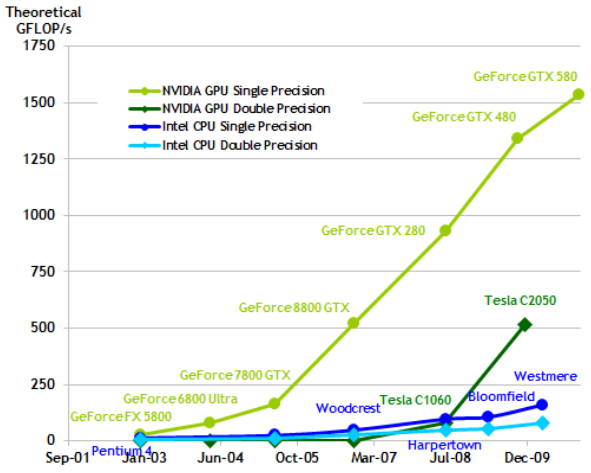

Fig. 1. Floating point operations per second on the CPU and GPU (image courtesy of (Nvidia Co. (2011a)))

CUDA (Compute Unified Device Architecture) represents a parallel architecture in NVIDIA cards that is accessible to

software developers through variants of standard programming languages like CUDA C. It can solve complex problems in a more efficient way than on the CPU. At the core of the CUDA C language there are three main abstractions, a hierarchy of threads, a hierarchy of memories and barrier synchronization, which are exposed to the programmers through a set of minimal extensions of the C language.

Parts of the Canny edge detector have been successfully implemented on the GPU with NVIDIA's Cg shading language by (Fung (2005)). Only the last part of the filter, i.e., the hysteresis thresholding, was not approached in his paper. (Luo et al. (2008)) developed a new implementation of the Canny edge detector, using the GPGPU paradigm. It follows all the steps of the algorithm, including the hysteresis thresholding.

In their paper, (Braak et. al. (2011)) describe three new implementations of the Hough transform for lines. They concentrate on the advantages and disadvantages of each approach. Other research directions presented by (Chen et al. (2011)) follow ways of accelerating the Hough transform both for lines and circles.

2.2. Image processing in hip arthroplasty

According to (Kennon (2008)), athroplasty represents a surgical procedure of inserting an artificial implant (prosthesis) in place of an arthritic joint. (Morar et al (2010a, b)) describe the important parameters in hip replacement and a series of algorithms that lead to the automatic/semi-automatic extraction of these parameters. For the understanding of the current paper, only the salient parts of the bones that can be approximated by simple curves are presented. Further explanations regarding computer assistance in Hip Arthroplasty decision making can be found in the previously mentioned papers.

As can be observed in Fig. 2, the contour of the femoral body can be approximated by two straight lines. Also, the ischiadic tuberosities, the femoral head and the lesser trochanter have similarities with parts of circles. This is the idea behind identifying the position of the salient parts of the bones. After the extraction of the parameters presented in Fig 2 all the other automatic measurements described by (Morar et al. (2010a, b)) can be easily accomplished.

Although there are some papers on GPGPU based image processing and others on automatic measurements in arthroplasty, none of them use the power of multi-GPGPU computers. The computing times are further reduced by taking into account the particularities of special images, like the radiographic images at the level of the hip.

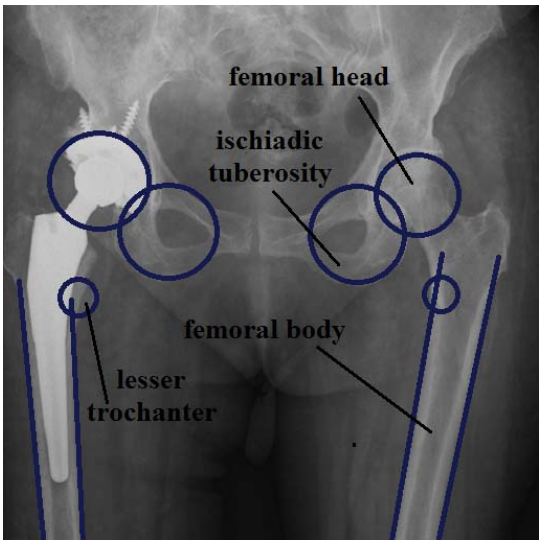

Fig. 2. Parts of bones that can be approximated by curves

3. CANNY EDGE DETECTOR WITH CUDA

3.1. General presentation of the Canny edge detector

Canny edge detector, proposed by (Canny (1986)), is a widely used contour extraction technique.

After many experiments, it was decided that the last step in the Canny edge detector, i.e., the hysteresis thresholding, does not lead to significant improvements in finding the contour of the bones in radiographic images. The hysteresis thresholding step is quite difficult to implement in CUDA because it is not intuitively parallel. Thus, the step that connects the pixels of the contour has been removed from the current implementation of the Canny edge detection. Instead, a single threshold T that divides the pixels into background and contour pixels is introduced.

This new version of the Canny edge detection method is based on the CUDA SDK example of the Sobel filter described by (Nvidia Co. (2011b)). The first five steps and the alteration of the sixth step of the detector are briefly described:

1. The image is filtered with a 3x3 convolution mask that approximates the Gaussian filtering, described in Gonzales et al. (2002).

2. The output is then filtered with the Sobel operator. The gradient is approximated with the following masks:

-1	0	1
-2	0	2
-1	0	1

Dx

1	2	1
0	0	0
-1	-2	-1

Dy

The matrix of the gradient magnitudes is obtained by using the following expression:

$$D(x,y) = \sqrt{D_x^2 + D_y^2} \ . \quad (1)$$

3. The direction of the gradient is determined:

$$\theta = \arctan\left(\frac{D_y}{D_x}\right) \text{ for } D_x \neq 0 \, . \quad (2)$$

4. The direction of the gradient is discretized in the following manner:

$$\theta = \begin{cases} 0^0, if \quad 0^0 \le \theta \le 22.5^0 \quad or \quad 157.5^0 \le \theta \le 180^0 \\ 45^0, if \quad 22.5^0 \le \theta \le 67.5^0 \\ 95^0, if \quad 67.5^0 \le \theta \le 122.5^0 \\ 135^0, if \quad 122.5^0 \le \theta \le 157.5^0 \end{cases} \quad (3)$$

5. Non-maximum suppression: Having a pixel *A*, the two neighbours, *B* and *C*, on the direction of the gradient, are detected. If *Intensity* (*A*) < *Intensity* (*B*) or *Intensity* (*A*) < *Intensity* (*C*), then *Intensity* (*A*) = 0.

6. Instead of applying two thresholds and connecting the border pixels, only one threshold *T* is applied.

3.2. Canny edge detector with CUDA

This section presents a simplified implementation of the Canny edge detector with CUDA. It is based on the method proposed by (Luo et al. (2008)).

The parallel implementation uses three kernels that are executed for all the pixels in the image. Before these kernels are run, the intensities of the image are copied in the GPU texture memory.

The first step of the detector, i.e., the Gaussian filtering, is implemented in the first CUDA kernel. The kernel reads the intensities of the pixels located in a 3x3 neighbourhood of the current pixel and updates the intensity of that pixel based on the Gaussian mask. Each thread block handles an image row, as illustrated in Fig. 3. If the size of the block is less than the image width, a thread handles more than one pixel.

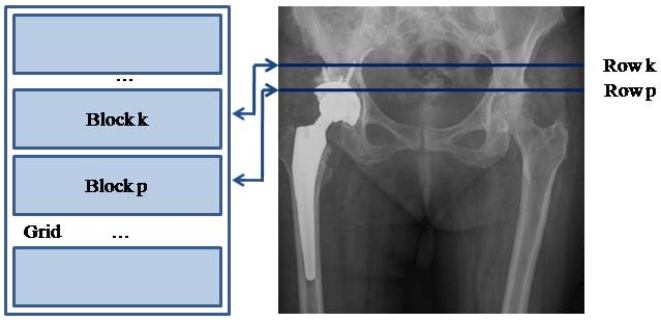

Fig. 3. Each block of threads handles one row in the image

The possibility of using the shared memory has been explored, and is presented in a few lines. Each thread block copies parts of the image by loading data from texture memory to shared memory. Through barrier synchronization, each thread waits until the other threads have finished loading the corresponding data from texture to shared memory. Using the faster access to the shared memory a thread updates in one iteration four pixels instead of one (as in Fig. 4). Again, after updating the pixels, the threads of a block synchronize.

Although the access to the shared memory is faster than the access to texture memory, the transfer between these two memory spaces leads to a lag and determines an insignificant difference between the two implementations.

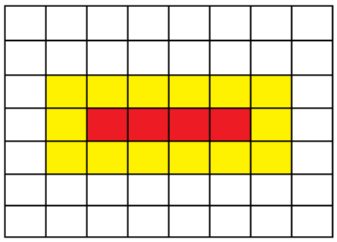

Fig. 4. In one iteration, a thread updates four pixels (the red pixels), based on their neighbours (the yellow pixels).

For simplicity, the next steps of the Canny algorithm are described using only texture memory. It has to be mentioned that they can be implemented using shared memory, but without significant improvement.

The result of the Gaussian filter is saved into texture memory, because it can be accessed faster than the global memory.

The next three steps of the Canny edge detector, i.e. the computation of the gradient magnitude with the Sobel convolution kernels, the computation of the gradient direction and the discretization of the gradient direction, are defined in a single CUDA kernel. Within the kernel, the pixels in a 3x3 neighbourhood of the current pixel are read from the texture memory. For each pixel the values of *Dx* and *Dy* are determined, the gradient amplitude is computed by using (1) and the gradient direction is calculated based on (2). The gradient direction is discretized by using (3). Similar to the Gaussian filtering, each thread block handles an image row.

The last two steps of the Canny edge detector can be defined in a single CUDA kernel. Within the kernel, the discretized direction of the gradient and the intensity of the current pixel are read. The intensities of the pixel's neighbours on the gradient direction are determined. If the intensity of a neighbour is greater than the intensity of the current pixel, this pixel is removed from the contour pixels. The modified sixth step uses a single threshold *T*. All the pixels with intensity value greater than T are considered contour pixels. All the other pixels are background pixels. Each thread block handles one image row.

The pseudo code of the CUDA Canny edge detector is given below.

```
Pseudo code 1. Canny edge detector
Stage 1. Gaussian filtering
copy image from CPU to texture memory on the GPU
for every image pixel p, do (in parallel)
   read values of p from a 3x3 neighbourhood
   compute intensity based on the Gaussian mask
end for
Stage 2. Gradient magnitude and direction + discretization of gradient direction
copy output of Stage 1 into texture memory
for every pixel p, do (in parallel)
   read values of p from a 3x3 neighbourhood
   determine Dx and Dy
   compute gradient magnitude
   compute gradient direction
   discretize gradient direction
end for
Stage 3. Non-maximum suppression and hysteresis
copy the gradient magnitudes into texture memory
for every image pixel A, do (in parallel)
   read gradient direction (discretized)
   determine neighbour pixels C and B along gradient direction
   read values vA, vB and vC of A, B and C
   if vA<vB or vA<vC, then
      vA=0
   end if
   //thresholding
   if vA>T (threshold), then
      vA=255 (border pixel)
   else
      vA=0 (background pixel)
   end if
end for
copy final image from GPU to CPU
End of pseudo code
```

3.3. Improvements to the Canny edge detector with CUDA

This section describes the main improvements of the Canny edge detector implementation with CUDA. One of the features recently introduced in the CUDA architecture is "zero-copy". This feature refers to direct device access to host memory. Before this, a simple CUDA program was structured into three stages:

- Copy data from CPU to GPU
- Process data on the GPU
- Copy data back from the GPU to the CPU

Now, the GPU devices can have access to the data on the host, if the data resides into page locked memory. The data can be allocated on page locked memory via `cudaHostAlloc()`. The instruction `cudaHostGetDevicePointer()` passes back the device pointer corresponding to the mapped, pined host buffer allocated by `cudaHostAlloc()`. This eliminates the copies from host to device and from device to host. However, the access to page locked memory is slower than the access to device memory. So, if a kernel needs to access the data frequently, a compromise should be made:

- Allocate CPU data on page locked memory
- Copy data from CPU to GPU
- Process data on the GPU
- Copy data back from GPU to page locked memory

This approach is faster than the normal flow in a CUDA program because the transfer between page locked memory and device memory is faster than the transfer between pageable memory and device memory. Even if it can reduce the memory transfer duration, page locked memory is limited and can also degrade system performance, since it reduces the amount of memory available to the system for paging. These problems are further discussed in the Results section.

The first improved version of the CUDA implementation of the Canny edge detector takes advantage of the “zero-copy” feature. The data, i.e., the radiographic image, is allocated on page locked memory with `cudaHostAlloc()`. A pointer to this data that can be accessed by the device is obtained with the instruction `cudaHostGetDevicePointer()`. The remainder of the algorithm is similar to the one in pseudo code 1, without the last instruction that copies the data from the GPU back to the CPU.

The second improved version uses page locked memory, but still copies the data between CPU and GPU before and after applying the Canny edge detector.

The last proposed version of the Canny edge detector with CUDA offers the possibility to run the algorithm on multiple graphic cards. If a problem can be divided into several sub-problems to be solved independently on different GPUs, the computing times can be significantly reduced. Let N be the number of available graphic cards on a computer. The initial image can be divided on the y axis into N sub-images that are processed independently, as illustrated in Fig. 5.

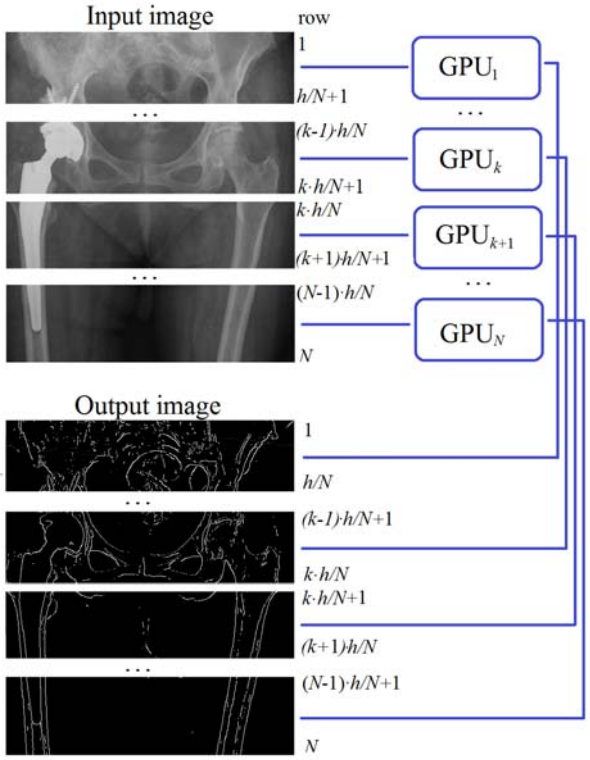

Fig. 5. Division of input and output images for multi-GPGPU processing

Let w and h be the width, respectively the height of the image. It can be assumed without loss of generality that h is a multiple of N.

The output of each GPU is a sub-image of width w and height h/N. The output image of GPU_k starts with the $[(k-1)\cdot h/N+1]^{th}$ row and ends with the $(k\cdot h/N)^{th}$ row. Each pixel is processed based on the pixels from a 3x3 neighborhood. Thus, for the correct processing of the border pixels, the input of each GPU is a sub-image of width w and height $h/N+2$. The input sub-image of GPU_k starts with the $[(k-1)\cdot h/N]^{th}$ row and ends with the $(k\cdot h/N+1)^{th}$ row. Exceptions are the first and the last GPU, which have input images of height $h/N+1$.

Data is processed on different GPUs concurrently through streams. The sub-images are copied from page locked memory on the CPU to GPU and back asynchronously with `cudaMemcpyAsync()`. The pseudo-code of the multi-GPGPU Canny edge detector is given below.

```
Pseudo code 2. Multi-GPGPU Canny edge detector
Divide the data according to the number of
available GPUs in the system

for k=0 to N, do
   set active device GPUk
   allocate memory on CPU for sub-image k (with
cudaHostAlloc)
   copy from initial image to input sub-image k
on the CPU
   allocate memory for sub-image k on GPUk
   create stream for k
end for

Copy data to/from each GPU asynchronously and
apply Canny edge detector on each GPU

for k=0 to N, do
   set active device GPUk
   copy input data from CPU to GPUk (with
cudaMemcpyAsync)
   apply Canny on sub-image of k
   copy output data from GPUk to CPU (with
cudaMemcpyAsync)
end for

Synchronize streams

for k=0 to N, do
   set device GPUk
   wait for all operations of stream k to finish
   delete stream k
   copy data from output sub-image on the CPU to
final image on the CPU
end for
End of pseudo code
```

Arithmetic operations on N GPUs are N times faster than the same arithmetic operations on a single GPU. However, the division of data and memory operations to/from multiple GPUs introduce a lag that can eliminate the multi-GPGPU performance gain. This issue is further discussed in the Results section.

4. HOUGH TRANSFORM FOR LINES WITH CUDA

4.1. General presentation of the Hough transform for lines

A straight line can be described by the expression $y=mx+b$. The main idea behind the Hough transform for lines according to Hough (1962) is to define a line not as a set of points (x,y), but in terms of parameters, based on m and b. Hence, the straight line $y=mx+b$ can be represented by the point (m,b) in the parametric space. There is another pair of parameters, r and θ, better suited to define a line:

$$y=\left(-\frac{\cos\theta}{\sin\theta}\right)\cdot x+\left(\frac{r}{\sin\theta}\right), \text{ or}$$

$$r=x\cdot\cos\theta+y\cdot\sin\theta, \tag{4}$$

where $r\in\left[1,\sqrt{w^2+h^2}\right]$ is the distance from the origin to the line and $\theta\in[0,360]$ is the angle of the vector from the origin to the closest point on the line. The Hough algorithm uses an accumulator that stores the number of white pixels corresponding to each pair (r,θ). The pairs (r,θ) where the accumulator's value is greater than a given threshold describe straight lines in the image.

4.2. CUDA implementation of the Hough transform for lines

The intuitive implementation described in Luo et al. (2011) follows the CPU implementation of the algorithm. However, in the parallel implementation, all threads have concurrent access to data. In order to avoid memory read/write hazards, the incrementing of the accumulator at location (r,θ) has to be done using an atomic operation, `atomicAdd()`. This operation serializes the access of the threads to the accumulator, representing at a first glance an inefficient use of the parallel architecture.

The updating of the accumulator is defined in a CUDA kernel that is executed for every image pixel. The intensity of the current pixel is read from a binary image, where the important pixels are white and the background pixels are black. In the proposed radiographic image processing method, the binary image is the output of the Canny edge detector. If the current pixel is white, the algorithm searches for all the candidate lines that go through that pixel. For all θ, the second parameter r is computed by using (4). If $r\in\left[1,\sqrt{w^2+h^2}\right]$, the accumulator at location (r,θ) is incremented. Every thread block handles an image row.

According to (Luo et al. (2011)), the atomic operation that updates the accumulator introduces a considerable lag. However, for radiographic images, where the contour pixels represent less than 3% of the total number of image pixels and the straight lines are very sparse, the atomic operation does not affect the performance of the CUDA Hough transform.

The pseudo code of the CUDA Hough transform for lines is described below.

```
Pseudo code 3. Hough transform for lines
transfer image data from CPU to GPU texture
memory
Run kernel to initialize accumulator
for every pair (r,θ), do (in parallel)
   accum[r,θ]=0
end for
Run kernel to update accumulator
for every pixel p, do (in parallel)
   if p is white, then
      get x and y coordinates of p
      for θ = 0 to 360, do
         compute r
         if r>1 and r<sqrt(w*w+h*h), then
            increment accumulator accum at
location [r,θ](with atomicAdd)
         end if
      end for
   end if
end for
Determine the lines in the image based on the
value of the accumulator
for every θ, do
   for every r, do
      if accum[r,θ]>T (threshold), then
         (r,θ) is a line
      end if
   end for
end for
End of pseudo code
```

4.3. Improvements to the Hough transform for lines

The first change to the described implementation is to allocate memory for the image data to page locked memory with `cudaHostAlloc()`. If the CPU data is accessed directly by the device with `cudaHostGetDevicePointer()`, the algorithm is slower than the initial one. The reason is the frequent access of the GPU to the accumulator.

The other version that uses page locked memory without eliminating the copies between CPU and GPU is a faster alternative. The computing times for the proposed implementations are described in the Results section.

The next alteration to the initial implementation is the multi-GPGPU Hough transform for lines. It divides the image in a similar way as to the one described in the multi-GPGPU Canny edge detection method. The input and output sub-images of GPU_k for the Hough transform for lines have the same properties as the output sub-image of GPU_k for the Canny edge detector. Here there is no need to duplicate the border rows for the input sub-images because the accumulator is not processed based on neighbouring pixels.

If the data that represents the image can be divided based on the number of available GPUs, this cannot be accomplished with the accumulator. For each GPU, a sub-accumulator of the same size as the initial one is allocated on the CPU on page locked memory and on the current GPU. After the accumulator of each GPU is updated, and the data is copied back to the CPU on page locked memory, the final accumulator is built based on the following expression:

$$accum\ (r,\theta) = \sum_{k=1}^{N} accum\ _{GPU_k}(r,\theta) \tag{5}$$

Another alteration to the initial algorithm is to change the structure of the CUDA grids and blocks.

In the classic implementation, the threads are structured into one-dimensional blocks and the blocks are structured into a one-dimensional grid. The size of the grid is equal to the height of the image. The size of a block is 256. Each block handles a single image row. If the width of the image is greater than the number of threads in a block, a thread processes more than one pixel.

The new approach structures the thread based on the size of the image, but also on the size of the parameter θ. The blocks are structured into two-dimensional grids, with the width and height equal to the image width and image height, respectively. The size of a block is equal to the number of possible values of parameter θ. Thus, a thread does not handle all the candidate straight lines that go through a pixel, but a single line defined by the parameter θ which goes through the current pixel. This implementation is faster than the first one if the search space, i.e., the size of the image and the size of parameters r and θ, is relatively small. Otherwise, there will not be enough GPU cores to handle all the data concurrently, and the CUDA driver will serialize the process. The lag introduced by this serialization can eliminate the performance gain obtained by the different structuring of the CUDA threads.

The initial interval of the parameter θ is [0,360]. If the particularities of the orthopedic radiographic images are taken into account, this interval can be significantly reduced. For instance, if the application searches for straight lines that approximate the contour of the femoral body in images at the level of the hip, only the lines that are almost vertical should be considered. The search space for the parameter θ is reduced to the interval $[0,15]\cup[165,195]\cup[345,360]$. Thus, a block has 45 threads instead of 360, one thread for each θ. A comparison between the existing and proposed implementations is described in section 7.

5. HOUGH TRANSFORM FOR CIRCLES WITH CUDA

5.1. General presentation of the Hough transform for circles

A circle is described by the expression

$$(x-a)^2 + (y-b)^2 = r^2, \tag{6}$$

where (a,b) are the coordinates of the centre of the circle, and r, its radius. The parametric space of the Hough transform for circles is described by the triplet (a,b,r). Usually, $r \in [1, \sqrt{w^2+h^2}]$, where w and h represent the width and the height of the image, while $a \in [0, w]$ and $b \in [0, h]$.

5.2. CUDA implementation of the Hough transform for circles

This section describes the intuitive implementation of the CUDA Hough transform for circles. The accumulator is updated in a CUDA kernel that is executed for all the image pixels. If the current pixel is of interest, r is computed for every pair (a,b) based on (6). If r is within range, the accumulator is incremented at location (a,b,r) with atomic operations. All the triplets (a,b,r) where the accumulator has the value greater than a given threshold T describe a circle.

Similar to the observation in section 4.2 regarding computing times for the implementation of the Hough transform for lines with atomic operations, there is a small number of circles extracted from radiographic images. This is the reason why atomic operations are not considered to increase significantly the computing times in this particular case.

The pseudo code of the CUDA Hough transform for circles is given below.

```
Pseudo code 4. Hough transform for circles
transfer image data from CPU to texture memory
on the GPU
for every pixel p, do (in parallel)
   if p is white, then
      get x and y coordinates of p
      for a = 1 to w, do
         for b = 1 to h, do
            compute r (based on (6))
            if r>1 and r<sqrt(w*w+h*h), then
               increment accumulator accum at
location [a,b,r](with atomicAdd)
            end if
         end for
      end for
   end if
end for
for every a, do
   for every b, do
      for every r, do
         if accum[a,b,r]>T (threshold), then
            (a,b,r) is a circle
         end if
      end for
   end for
end for
End of pseudo code
```

5.3. Improvements to the Hough transform for circles

This section describes the changes made to the classic implementation of the algorithm with CUDA, which are similar to the ones proposed in 4.3.

The parameter space of the Hough transform for circles is three-dimensional, and occupies large amounts of memory. This represents an impediment for using page locked memory. For an image of size 512x512, the accumulator occupies $size(r) \cdot size(a) \cdot size(b) \cdot sizeof$ (int) $= 724$ MB of memory. On the computer used for testing it was impossible to allocate such amount of pinned memory. Also, the direct access to the data that resides on page locked memory via `cudaHostGetDevicePointer()` introduces a lag that reaches the maximum run-time for the kernel launches. These are the reasons for not using the implementations that allocate page locked memory

Another change made to the classic implementation is the different structuring of the CUDA threads. The new approach structures the thread based on the size of the image, but also on the size of the circle radius r. The blocks are structured into two-dimensional grids, with the width and height equal to the image width and image height, respectively. The size of a block is equal to the number of possible values of parameter r. Thus, a thread does not handle all the possible straight circles that go through a pixel, but a single circle defined by parameter r that goes through the current pixel. The current thread computes for every $b \in [\max(0, y-r), \min(y+r, h)]$ the value of parameter a, based on (6). If a is within range, the accumulator is incremented at location (a,b,r). In the classic implementation, a and b loop through their whole search space. In this new implementation, the search space of b is reduced based on the radius of the circle r. The comparison between the classic implementation and the proposed one is described in the Results section.

The parameter search space can be further reduced if the particularities of the processed images are known. For example, if the application searches for circles that approximate the ischiadic tuberosities, the lesser trochanters and the femoral heads in radiographic images, the radius of these circles can be set within certain intervals. The circles representing the ischiadic tuberosities and the femoral heads have a radius smaller than the width of the femoral body; the circles approximating the lesser trochanters have a radius smaller than half the width of the femoral body.

Thus, after determining the femoral body with the Hough transform for lines, the intervals for the radii of the circles can be set.

6. RESULTS

The tests regarding the automatic extraction of parts that can be approximated by simple curves from X-rays were made on a computer with i7-2600K 3.40 GHz processor with 8GB RAM and two Nvidia GeForce GTX 590 GPU cards with 1.5 GB RAM.

The CPU classic implementation of the Canny edge detector was compared to the implementation with CUDA described in section 3.2 and the new implementations that were proposed in section 3.3. Table 1 shows the computing times in ms for each method applied on radiographic images of different sizes. Each value represents the average of 10 tests. The third column in the table represents the intuitive implementation of the Canny edge detector with CUDA, as described in section 3.2. The forth column presents the computing times for the implementation that uses the "zero-copy" ("0copy") feature of the CUDA architecture. The fifth column denotes the implementation that allocates page locked memory ("pmem") on the CPU, but also transfers the data to/from the GPU. The forth GPU implementation is the multi-GPGPU Canny edge detector.

Table 1. Computing times of Canny edge detector on the CPU and on the GPU

Image size (in pixels)	CPU implem (ms)	GPU classic implem (ms)	GPU "0copy" implem (ms)	GPU "pmem" implem (ms)	Multi GPGPU implem (ms)
256^2	15	0.341	0.345	0.279	0.624
512^2	39	0.701	0.662	0.606	0.783
1024^2	153	2.149	1.931	1.867	1.392
2048^2	566	7.668	6.877	6.793	4.104
4096^2	2118	30.481	44.617	27.254	15.98

As can be observed, the implementation that allocates page locked memory on the CPU and transfers it via `cudaMemcpy()` to/from the GPU relatively improves the computing times as compared to the classic CUDA implementation. The implementation where the GPU accesses the data directly from page locked memory decreases performance for large images because of the frequent access to memory. The fastest implementation, i.e., the multi-GPGPU one, reduces significantly the computing times. For an image of size 256x256 the lag introduced by the division of data into sub-images and their transfer to/from the GPU leads to slower computing times than for other implementations. But for an image of size 4096x4096, two graphic cards perform the Canny edge detector almost two times faster than a single graphic card.

Fig. 6 presents the output image after applying the Canny edge detector on a radiographic image.

Fig. 6. The result of applying the Canny edge detector on a radiographic image

Table 2 shows the computing times for the implementations of the Hough transform for lines, both on the CPU and GPU (the average of 10 tests for each image). The first GPU implementation is the one described in section 4.2. The second and third GPU implementations allocate page locked memory with `cudaHostAlloc()`. In the second implementation the data is accessed directly by the GPU via `cudaHostGetDevicePointer()` and in the third one, the data is transferred to/from the GPU via `cudaMemcpy()`. The last column in the table shows the computing times for the multi-GPGPU implementation of the Hough transform for lines.

Table 2. Computing times of the intuitive Hough transform for lines on the CPU and on the GPU

Image size (in pixels)	CPU implem (ms)	GPU classic implem (ms)	GPU "0copy" implem (ms)	GPU "pmem" implem (ms)	Multi GPGPU implem (ms)
256^2	152	1.955	10.209	1.779	2.088
512^2	218	4.615	23.836	4.277	3.784
1024^2	1995	19.344	113.879	18.695	12.642
2048^2	9890	89.494	538.171	87.312	52.979
4096^2	13830	333.472	1768.36	326.143	118.771

As can be observed in Table 2, the "zero-copy" implementation is very slow because of the frequent access of the GPU to the page locked memory for the computing of the accumulator. The implementation that stores the accumulator on the CPU on page locked memory and transfers it to/from the GPU is faster than the other single-GPGPU implementations. But the real performance gain can be observed for the multi-GPGPU implementation.

The next table shows the computing times for similar implementations of the Hough transform. The only difference is in the structuring of the CUDA threads. Instead of searching for all the lines that go through the current pixel, a thread searches for the line that is defined by a certain parameter θ and goes through the current pixel.

Table 3. Computing times of the Hough transform for lines on the CPU and on the GPU - with the changed structuring of the CUDA threads

Image size (in pixels)	GPU classic implem (ms)	GPU "0copy" implem (ms)	GPU "pmem" implem (ms)	Multi GPGPU implem (ms)
256^2	3.184	11.201	3.028	2.715
512^2	7.995	20.181	7.536	6.305
1024^2	19.842	89.541	19.151	14.926
2048^2	54.907	439.531	52.502	38.875
4096^2	139.502	1284.097	132.478	101.326

The different thread structuring leads to faster computing times than the classic one, for images of size up to 1024x1024.

Table 4 shows the computing times of the Hough transform for lines, if the search space of parameter θ is reduced from the interval [0.360] to $[0,15] \cup [165,195] \cup [345,360]$.

Table 4. Computing times of the Hough transform for lines on the CPU and on the GPU - with reduced search space for parameter θ

Image size (in pixels)	CPU implem. (ms)	GPU classic implem (ms)	GPU "0copy" implem (ms)	GPU "pmem" implem (ms)	Multi GPGPU implem (ms)
256^2	33	0.701	4.857	0.549	1.088
512^2	39	1.996	11.744	1.696	2.175
1024^2	337	7.475	55.912	6.553	5.732
2048^2	1656	29.272	240.334	27.235	18.647
4096^2	2342	114.051	810.173	106.784	64.839

Fig. 7 presents the lines extracted with the Hough transform for lines.

Fig. 7. The result of applying the Hough transform for lines in order to extract the contour of the femoral body (and the contour of the prostheses' body, if it exists).

Table 5 shows the computing times for the CPU and GPU implementations of the Hough transform for circles. The first GPU implementation is the one described in section 5.2. The second implementation changes the structuring of the CUDA threads so a thread does not handle all the circles that contain the current pixel, but only one circle defined by a certain radius that contains the current pixel. The third CUDA implementation shows the impact of reducing the search space of the radius parameter from the interval $[1, \sqrt{w^2 + h^2}]$ to $[1, \sqrt{w^2 + h^2}/10]$, based on the properties of bones that can be approximated by circles, which are described in section 5.3. The first CPU implementation is the intuitive one, and the second one reduces the search parameter of r to $[1, \sqrt{w^2 + h^2}/10]$.

The application structured in a manner that a thread handles all the circles that contain the current pixel is considerably slower than the one that is structured in a manner that a thread handles only one circle defined by a certain radius that contains the current pixel. If the search space for the radius is reduced according to the particularities of the radiographic images and the investigated parameters, the computing times are further improved.

Table 5. Computing times of the Hough transform for circles on the CPU and on the GPU

Image size (in pixels)	CPU classic implem. (ms)	CPU "reduced space" implem. (ms)	GPU classic implem. (ms)	GPU "diff struct" implem. (ms)	GPU "reduced spaced" implem. (ms)
256^2	11622	57	389.831	24.154	2.907
512^2	157275	453	2787.51	186.429	20.169
1024^2	1019555	4798	X	X	149.995

Fig. 8 presents the parameters extracted with the Hough transform for circles. As can be observed, the lesser trochanter on the right side could not be extracted because it does not have a circular shape.

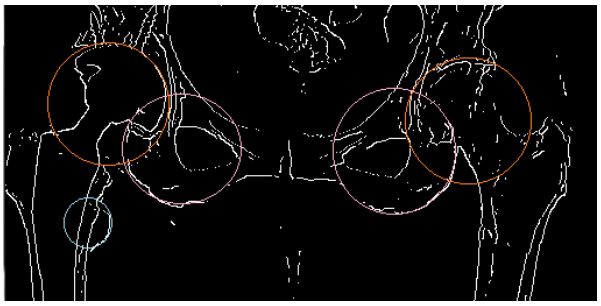

Fig. 8. The result of applying the Hough transform for circles in order to determine the position of the ischiadic tuberosities, the femoral heads and the lesser trochanters.

7. CONCLUSIONS

Even though the differences in GPU implementations (with or without reducing the search space) are small, it must be mentioned that the GPU memory is quite limited as compared to the CPU memory. This is why, for large images, the implementation that does not reduce the ranges of the parameters cannot be run on any hardware.

The tests were made on radiographic images at the level of the hip for the extraction of certain parameters important in hip arthroplasty. However, the proposed multi-GPGPU implementations of the Canny edge detector and the Hough transform for lines can be used on any kind of images. The performance gain is quite impressive. For the Canny edge detector, the application that runs on two graphic cards is almost two times faster than the one that runs on a single GPU. The multi-GPGPU implementation of the Hough transform for lines also reduces the computing times considerably.

The other changes made to the classic implementations show that the particularities of certain images and of the searched objects in the image can also add a performance gain by reducing the search space for the parameters of the Hough transforms.

A future research direction would be to explore the possibility of using the CUDA architecture for accelerating other analysis, processing and visualization algorithms in 2D and 3D medical images.

ACKNOWLEDGEMENTS

Part of the work has been funded by the Sectoral Operational Programme Human Resources Development 2007-2013 of the Romanian Ministry of Labour, Family and Social Protection through the Financial Agreement POSDRU/88/1.5/S/61178.

REFERENCES

Braak G.J., Nugteren C., Mesman B., Corporaal H. (2011). Fast Hough Transform on GPUs: Exploration of Algorithm Trade-offs. *Proceedings of Conference on Advanced Concepts for Intelligent Vision Systems (ACIVS' 11)*.

Canny J.F. (1986). A computational Approach to Edge Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence.* Vol. 8, issue 6, pp. 679-698.

Chen S., Jiang H. (2011). Accelerating the Hough Transform with CUDA on Graphics Processing Units. *Proceedings of 2011 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*.

Dicom Standard (2009) – available at http://dicom.nema.org

Fung J. (2005). *GPU Gems,* Computer Vision on the GPU, pp. 649-664.

Gonzales, R.C., Woods R.E. (2002). *Digital Image Processing,* pp. 222-224. Prentice Hall.

Hough P.V.C. (1962). *Method and Means for Recognizing Complex Patterns*. US Patent Publication No. US 3069654.

Kennon R. (2008). *Hip and Knee Surgery: A Patient's Guide to Hip Replacement, Hip Resurfacing, Knee Replacement & Knee Arthroscopy.* Middlebury.

Luo Y., Duraiswami R. (2008). Canny Edge Detection on NVIDIA CUDA. *Proceedings of IEEE Computer Vision and Pattern Recognition Workshops*.

Morar, A., Moldoveanu F., Moldoveanu A., Asavei V., Egner A. (2010a). Computer Assisted Analysis of Orthopaedic Radiographic Images". *Proceedings of the 9th WSEAS International Conference on Signal Processing*.

Morar, A., Moldoveanu F., Moldoveanu A., Asavei V., Egner A. (2010b). Medical Image Processing in Hip Arthtroplasty. *WSEAS Transactions on Signal Processing.* Vol. 6, issue 4.

Nvidia Co. (2011a). *NVIDIA CUDA C Programming Guide* (Version 4.0) – available at www.nvidia.com

Nvidia Co. (2011b). *NVIDIA CUDA Computing SDK* – available at www.nvidia.com