GPU implementation of a region based algorithm for large images segmentation

Gilles Perrot, Stéphane Domas, Raphaël Couturier
Distributed Numerical Algorithmics team (AND), Laboratoire d'Informatique de Franche-comté
Rue Engel Gros, 90000 Belfort, France
forename.name@univ-fcomte.fr

***Abstract*—Image segmentation is one of the most challenging issues in image computing. In this work, we focus on region-based active contour techniques (snakes) as they seem to achieve a high level of robustness and fit with a large range of applications. Some algorithmic optimizations provide significant speedups, but even so, execution times are still non-neglectable with the continuing increase of image sizes. Moreover, these algorithms are not well suited for running on multi-core CPU's. At the same time, recent developments of Graphical Processing Units (GPU) suggest that higher speedups could be obtained by use of their specific design. We have managed to adapt a specially efficient snake algorithm that fits recent Nvidia GPU architecture and takes advantage of its massive multithreaded execution capabilities. The speedup obtained is most often around 7.**

***Keywords*-GPU; segmentation; snake;**

I. Introduction

Segmentation and shape detection are still key issues in image computing. These techniques are used in numerous fields ranging from medical imaging to video tracking, shape recognition or localization. Since 1988, the active contours (snakes) introduced by and Kass et al. [**?**], have proved to be efficient and robust, especially against noise, for a wide range of image types.

The main shortcoming of these algorithms is often their high dependence on the initial shape, though several contributions have lowered this dependency and also brought more accurate segmentation of non convex shapes [**?**] [**?**].

The information that drives a snake model comes either from the contour itself or from the characteristics of the regions it defines. For noisy images, the second option is often more suitable as it takes into account the statistical fluctuations of the pixels. One approach [**?**], [**?**] proposes a geometric (polygonal) region-based snake driven by the minimization of the stochastic complexity. One significant advantage is that it runs without any free parameter which can be helpful when dealing with image sequences or slices (3D).

An important issue of image processing, especially segmentation, has always been the computation time of most algorithms. Over the years, the increase of CPU computing capabilities, although quite impressive, has not been able to fulfill the combined needs of growing resolution and real-time computation. Since having been introduced in the early 1980's, the capabilities and speed of graphics accelerators have always been increasing. So much so that the recent GPGPU (General Purpose Graphic Processing Units) currently benefit by a massively parallel architecture for general purpose programming, especially when dealing with large matrices or vectors. On the other hand, their specific design obviously imposes a number of limitations and constraints. Some implementations of parametric snakes have already been tested, such as [**?**]. However, a similar solution (computation per small tile) is not suited for the algorithm we have implemented.

Our goal, in collaboration with the PhyTI team[1], was to propose a way to fit their algorithm to the Nvidia© Tesla GPU architecture. The remainder of this paper presents the principles of the algorithm and notations in section II. In section III, the details of the sequential CPU implementation are explained. Section IV summarizes Nvidia's GPU important characteristics and how to deal with them efficiently. Then sections V and VI detail our GPU implementation and timing results. Finally, the conclusion of section VII evaluates the pros and cons of this implementation and then gives a few direction to be followed in future works.

II. Sequential algorithm : outlines

The goal of the active contour segmentation (snake) method we studied [**?**] is to distinguish, inside an image I, a target region T from the background region B. The size of I is L x H pixels of coordinates (i, j) and gray level $z(i, j)$. We assume that the gray levels of T and B are independent random vectors, each with a distribution p^{Ω} of its components $(\Omega \in \{T; B\})$. The present implementation uses a Gaussian distribution, but another one can easily be used as Gamma, Poisson,...(Cf. [**?**])...

The *active contour* S, which represents the shape of T is chosen as polygonal. The purpose of the segmentation is then to determine the shape that optimizes a pseudo log-likelihood-based criterion (PLH). This is done by a very simple iterative process which is initialized with an arbitrary shape, then at each step :

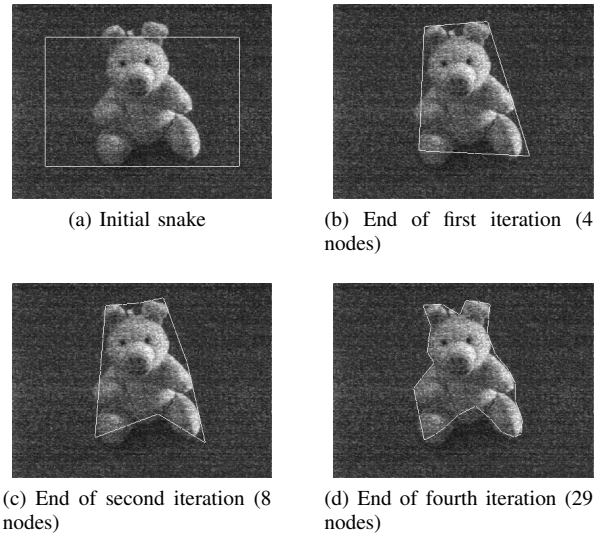
[1]Physics and Image Processing Group, Fresnel Institute, Ecole Centrale de Marseille (France)

1) it modifies the shape
2) it estimates the parameters of the Gaussian functions for the two regions and evaluates the criterion.
3) it validates the new shape if the criterion has a better value.

A simplified description of it is given in *Algorithm 1* which features two nested loops : the main one, on iteration level, is responsible for tuning the number of nodes ; the inner one, on step level, takes care of finding the best shape for a given number of nodes. *Figure 1* shows intermediate results at iteration level. Sub-figure *1a* shows the initial rectangular shape, *1b* shows the best four-node shape that ends the first iteration. Sub-figures *1c* and *1d* show the best shape for an eight-node snake (resp. 29-node) which occurs at the end of the second iteration (resp. fourth).

Algorithm 1: Sequential algorithm : outlines

```
1: begin with a rectangular 4 nodes snake;
2: repeat                          /* iteration level */
3:     repeat                      /* step level */
4:         Test some other positions for each node, near
           its current position;
5:         Find the best PLH and adjust the node's
           position;
6:     until no more node can be moved;
7:     Add a node in the middle of each long enough
       segment;
8: until no more node can be added;
```



(a) Initial snake

(b) End of first iteration (4 nodes)

(c) End of second iteration (8 nodes)

(d) End of fourth iteration (29 nodes)

Figure 1. segmentation of a noisy image

III. Sequential algorithm : details

A. Criterion

For p^{Ω} a Gaussian function, Θ_{Ω} ($\Omega \in \{T; B\}$) has two components, the average value μ and the deviation σ which are estimated by

$$\widehat{\Theta_{\Omega}} \left(\begin{array}{l} \widehat{\mu} = \frac{1}{N_{\Omega}} \sum\limits_{(i,j)\in\Omega} z(i,j) \\ \widehat{\sigma^2} = \frac{1}{N_{\Omega}} \sum\limits_{(i,j)\in\Omega} z^2(i,j) - \mu^2 \end{array} \right.$$

The likelihood of a region is given by

$$P[I|S_{n,l}, \Theta_T, \Theta_B] = P(\chi_T|\Theta_T)P(\chi_B|\Theta_B)$$

where

$$P(\chi_{\Omega}|\Theta_{\Omega}) = \prod_{(i,j)\in\Omega} p^{\Omega}[z(i,j)] \quad (\Omega \in \{T; B\})$$

And then the log-likelihood by

$$-N_{\Omega} \log\left(\sqrt{2\pi}\right) - N_{\Omega}.log\left(\sigma\right) - \frac{1}{2\sigma^2} \sum_{(i,j)\in\Omega} \left(z(i,j) - \mu\right)^2$$

Considering the two regions, the criterion to be optimized is then :

$$C = \frac{1}{2}\left(N_B \log\left(\widehat{\sigma_B}^2\right) + N_T \log\left(\widehat{\sigma_T}^2\right)\right)$$

B. CPU implementation

Let $S_{n,l}$ be the snake state at step l of iteration n, and $S^i_{n,l}$ the node i of $S_{n,l}$ ($i \in [0; N_n]$). Each segment of $S_{n,l}$ is considered as an oriented list of discrete points. Chesnaud & Refregier [**?**] have shown how to replace the 2 dimensions sums needed to estimate Θ_{Ω} by 1 dimension sums along $S_{n,l}$. However, this approach involves weighing coefficients for every single point of $S_{n,l}$ which leads to compute a pair of transformed images, at the very beginning of the process. Such images are called cumulated images and will be used as lookup tables. Therefore, beyond this point, we will talk about the *contribution* of each point to the 1D sums. By extension, we also talk about the *contribution* of each segment to the 1D sums.

A more detailed description of the sequential algorithm is given by *Algorithm 2*. The process starts with the computation of cumulated images ; an initialization stage takes place from line 3 to line 9. Then we recognize the two nested loops (line 10 and line 11) and finally the heart of the algorithm stands on line 15 which represents the main part of the calculations to be done :

1) compute the various sums without the contributions of both segments connected to current node $S^i_{n,l}$.
2) compute the contributions of both segments, which requires :
   - To determine the coordinates of every discrete pixel of both segments connected to $S^{i,w}_{n,l}$.

- To compute every pixel contribution.
- To sum pixel contributions to obtain segment contributions.

3) compute the PLH given the contribution of each segment of the tested snake.

Algorithm 2: Sequential simplified algorithm

```
1:  read image from HDD;
2:  compute_cumulated_images();
3:  iteration n ← 0;
4:  N_0 ← 4;
5:  S_{n,l} ← S_{0,0};
6:  step d ← d_max = 2^q;
7:  current node S^i_{0,0} ← S^0_{0,0};
8:  l ← 0;
9:  compute PLH_ref, the PLH of S_{n,0};
10: repeat                          /* iteration level */
11:     repeat                      /* step level */
12:         for i = 0 to N_n do
13:             S^{i,w}_{n,l} (w ∈ [0;7]) are the neighbors of S^i_{n,l}
                by distance d;
14:             for w = 0 to 7 do
15:                 compute PLH_w for S_{n,l} when S^{i,w}_{n,l}
                    replaces S^i_{n,l} ;
16:                 if PLH_w is better than PLH_ref then
                        PLH_ref ← PLH_w;
17:                     move node S^i_{n,l} ← S^{i,w}_{n,l};
18:             end
19:         end
20:         l ← l + 1;
21:     until no node move occured;
22:     add new nodes, N_n ← N_n + N_newnodes;
23:     if d > 1 then d ← d/2 else d = 1 ;
24:     n ← n + 1;
25:     compute PLH_ref, the PLH of S_{n,0} ;
26: until no new node added;
```

The profiling results of the CPU implementation shown in *Figure 2* display the relative costs of the most time-consumming functions. It appears that more than 80% of the total execution time is always spent by only three functions :

- `compute_segment_contribution()` which is responsible for point 2 above,
- `compute_cumulated_images()` which computes the 3 lookup tables at the very beginning,
- `compute_pixels_coordinate()` which is called by `compute_segment_contribution()`.

Measurements have been performed for several image sizes from 15 MPixels (about 3900 x 3900) to 144 MPixels (about 12000 x 12000). On the one hand, we can notice that function `compute_segment_contribution()` always lasts more than 45% of the total running time, and

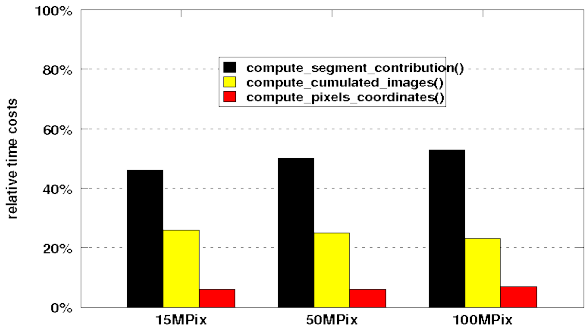


Figure 2. the three most-consumming functions for various image sizes

even more when the image gets larger. On the other hand, the function `compute_cumulated_images()` costs more than 23%, decreasing with image size, while function `compute_pixels_coordinate()` always takes around 6%. It confirms that the need for parallelization resides in line 15 and line 2 of Algorithm 2 as they contain every call to those three functions.

The following sections detail how we managed to implement these time-consumming functions in parallel, but a brief reminder on GPU's recent architecture is presented first.

IV. NVIDIA'S GPU ARCHITECTURE

GPUs are multi-core, multi-threaded processors, optimized for highly parallel computation. Their design focuses on SIMT model by devoting more transistors to data processing rather than data-caching and flow control [**?**].

For example, Figure 3 shows a Tesla C1060 with its 4GB of global memory and 30 SM processors, each including :

- 8 Scalar Processors (SP)
- a Floating Point Unit (FPU)
- a parallel execution unit (SIMT) that runs threads by warps of 32.
- 16KB of shared memory, organized in 16 banks of 32 bits words

Nvidia uses a parameter called the *compute capability* of each GPU model. Its value is composed of a major number and a minor number ; for example the C1060 is a sm13 GPU (major=1 minor=3) and C2050 is a sm20 GPU.

The recent Fermi cards (eg. C2050,) have improved performances by supplying more shared memory in a 32 banks array, a second execution unit and several managing capabilities on both the shared memory and level 1 cache memory ([**?**], [**?**], [**?**]. However, writing efficient code for such architectures is not obvious, as re-serialization must be avoided as much as possible. Thus, when designing, one must keep a few key points in mind :

- CUDA model organizes threads by a) threads blocks in which synchronization is possible, b) a grid of blocks with no possible synchronization between blocks.

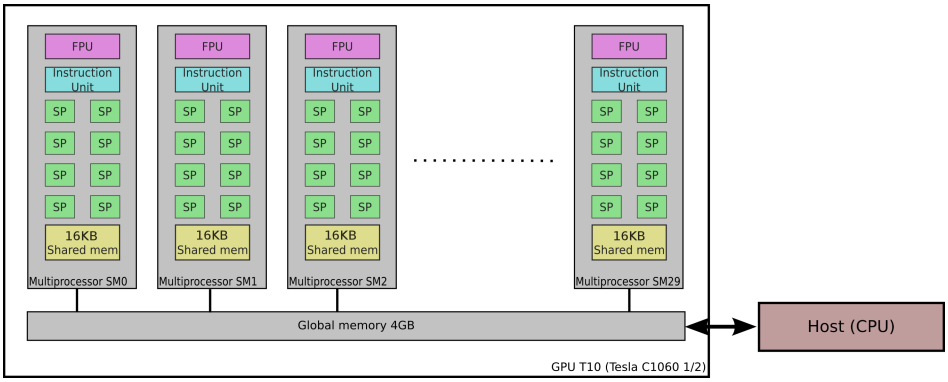


Figure 3. schematic diagram of GPU's internal architecture

- there is no way to know in what order the blocks are to be scheduled during one single kernel execution.
- data must be kept in GPU memory, to reduce the overhead due to copying between CPU and GPU memories.
- the total amount of threads running the same computation must be maximized.
- the number of execution branches inside a block should be reduced as much as possible.
- global memory accesses should be coalescent, *ie*. memory accesses done by physically parallel threads (16 at a time) must be consecutive and contained in a 128 Bytes range.
- shared memory is organized by 16 x 32 bits wide banks. To avoid bank conflicts, each parallel thread (16 at a time) must access a different bank.

All the above charasteristics make it always a quite constrained problem to solve when designing a GPU code. Moreover, a non suited code would probably run even slower on GPU than on CPU due to the automatic serialization which would be done at run time.

V. GPU IMPLEMENTATION

In the implementation described below, pre-computations and proper segmentation are discussed separately. To keep data in GPU memory, the whole computation is assigned to the GPU. CPU still hosts :

- data reading from HDD
- data writing on HDD if needed
- main loops control (corresponding to lines 10 and 11 of Algorithm 2)

It must be noticed that controlling these loops is achieved with only a very small amount of data being transferred between host (CPU) and device (GPU), which does not produce high overhead.
Morever, the structures described below need 20 Bytes per pixel of the image to process (plus an offset of about 50 MByte). It defines the maximum image size we can accept : approximately 150 M Pixels.

A. Pre-computations

To replace 2D sums by 1D sums, Chesnaud *et al.* [**?**] have shown that the three matrices below should be computed :

$$C_1(i,j) = \sum_{k=0}^{k=j}(1+k)$$

$$C_z(i,j) = \sum_{k=0}^{k=j} z(i,k)$$

and

$$C_{z^2}(i,j) = \sum_{k=0}^{k=j} z^2(i,k)$$

Where $z(i,k)$ is the gray level of pixel of coordinate (i,j), so that C_1, C_z and C_{z^2} are the same size as image I.

First, we chose not to generate $C_1(i,j)$, which requires that values should be computed when needed, but saves global memory and does not lead to any overhead. The computation of C_z and C_{z^2} easily decomposes into series of *inclusive prefixsums* [**?**]. However, by keeping the *1 thread per pixel* rule, as the total number of threads that can be run in a grid cannot exceed 2^{25} (Cf. [**?**]), slicing is necessary for images exceeding a size threshold which can vary according to the GPU model (e.g. 33 MPix for sm13 GPU, eg. C1060). It's quite easy to do, but it leads to a small overhead as the process requires multiple calls to one kernel. Slicing can be done in two ways :

- all slices are of the same size (balanced)
- slices fit the maximum size allowed by the GPU, leaving one smaller slice at the end of the process (full-sized).

The balanced slice option has proved to run faster.
For example : if a given image has 9000 lines and the GPU can process up to 4000 lines at a time, it's faster to run 3 times with 3000 lines rather than twice with 4000 and once with 1000.

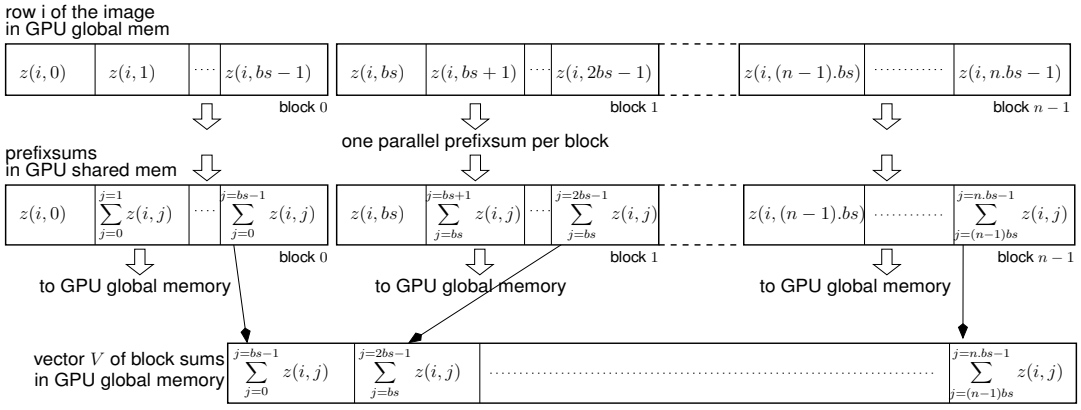


Figure 4. `compute_blocks_prefixes()` details.

As the sums in C_z and C_{z^2} are row-wide, it is easy to see that every block-wide sum will be needed before being able to use it in the global sum. But as mentioned earlier, the scheduling of blocks must be considered as random. So, in order to ensure synchronizations, each row of the original image is then treated by three different kernels :

- `compute_blocks_prefixes()`.
- `scan_blocksums()`.
- `add_sums2prefixes()`.

Figures 4, 5 and 6 show relevant data structures for a given row i of I. We assume that each thread block runs bs threads in parallel and each row of C_z needs n blocks to cover its L pixels.

Figure 4 shows the details of the process for row i of the original image I, already stored in GPU global memory. Operands are first copied into GPU shared memory for efficiency reasons. An inclusive prefixsum is then performed inside each independant thread block. At this point, only the first shared memory block contains the final values. Its last element contains the sum of all elements in the corresponding block of I. In order to obtain the right values for the row i of C_z, every element value in the other blocks must then be summed with an offset value. This offset value is the sum of all element values in every corresponding previous block of row i.

As the scheduling of blocks is fully unpredictable, the necessary intermediate results have to be stored in GPU global memory before exiting from kernel. Each element of the prefixsums in GPU shared memory has been stored in its corresponding position in C_z (GPU global mem), along with the vector of block sums which will be passed later to the next kernel `scan_blocksums()`.

The kernel `scan_blocksums()` (Figure 5) only makes an exclusive prefixsum on the vector of block sums described above. The result is a vector containing, at index x, the value to be added to every element of block x in each line of C_z.

This summing is done in shared memory by kernel `add_sums2prefixes()` as described by Figure 6.

The values of C_{z^2} are obtained together with those of C_z and in exactly the same way. For publishing reasons, figures do not show the C_{z^2} part of structures.

With this implementation, speedups are quite significant (Table I). Moreover, the larger the image, the higher the speedup is, as the step-complexity of the sequential algorithm is of $O(N^2)$ and $O(N\log(N))$ for the parallel version. Even higher speedups are achieved by adapting the code to specific-size images, especially when the number of columns is a power of 2. This avoids inactive threads in the grid, and thus improves efficiency. However, on sm13 GPUs, these computations are made with a 2-way bank conflict as sums are based on 64-bit words, thus creating overhead.

B. Segment contributions

The choice made for this implementation has been to keep the *1 thread per pixel* rule for the main kernels. Of course, some reduction stages need to override this principle and will be pointed out.

As each of the N_n nodes of the snake $S_{n,l}$ may move to one of the eight neighbor positions as shown in *Figure 7*, there is $16N_n$ segments whose contribution has to be estimated. The best combination is then chosen to obtain $S_{n,l+1}$ (Figure 7). Segment contributions are computed in parallel by kernel `GPU_compute_segments_contrib()`.

The grid parameters for this kernel are determined according to the size of the longest segment $npix_{max}$. If bs_{max} is the maximum theoritical blocksize that a GPU can accept,

- the block size bs is taken as
  - $npix_{max}$'s next power of two if $npix_{max} \in [33; bs_{max}]$
  - 32 if $npix_{max} < 32$
  - bs_{max} if $npix_{max} > 256$
- the number of threads blocks assigned to each segment, $N_{TB} = \frac{npix_{max}+bs-1}{bs}$

Our implementation makes intensive use of shared memory and does not allow the use of the maximum theoritical

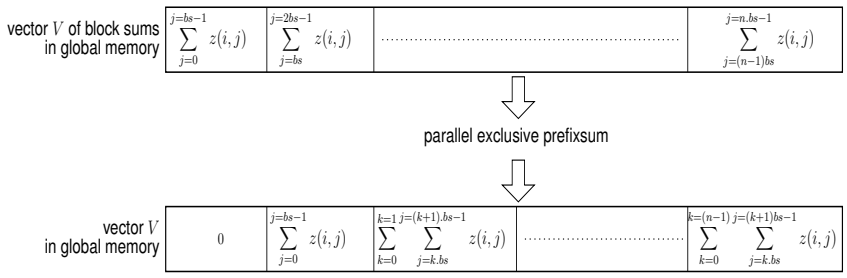


Figure 5. `scan_blocksums()` details.

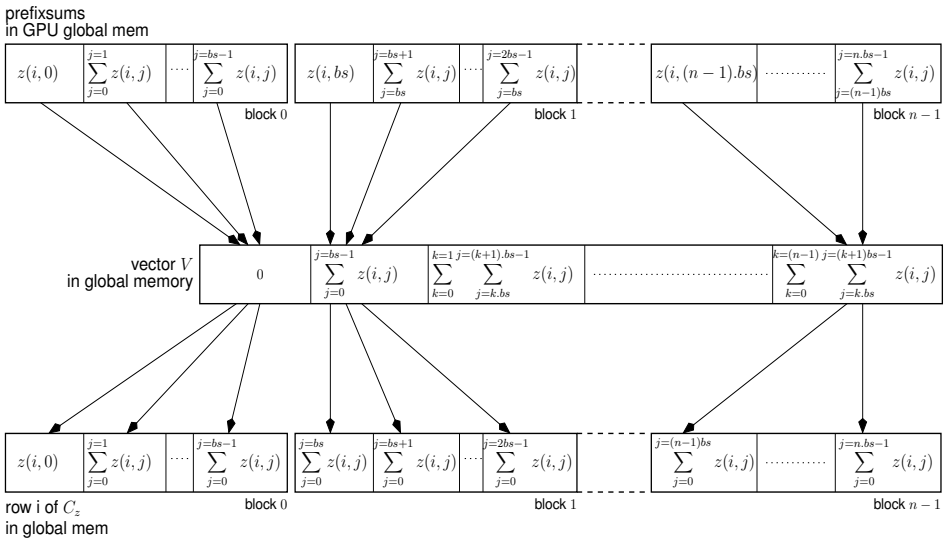


Figure 6. `add_sums2prefixes()` details.

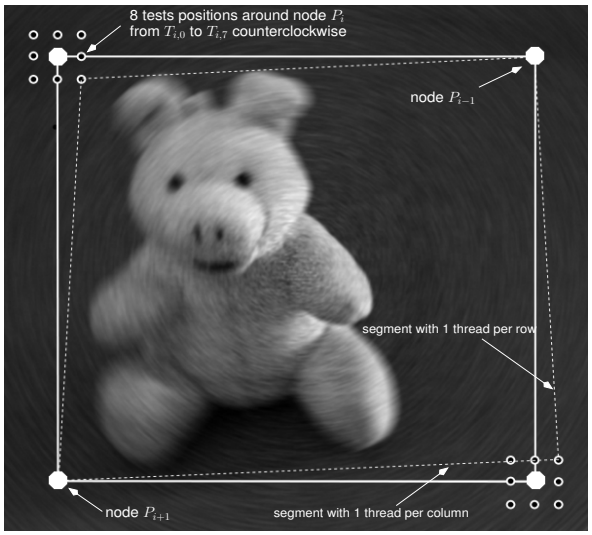


Figure 7. topology around nodes

blocksizes (512 for sm13, 1024 for sm20, see [**?**] and [**?**]). Instead we set $bs_{max}^{sm13} = 256$ and $bs_{max}^{sm20} = 512$. Anyway, testing has shown that most often, the best value is 256 for both *sm13* and *sm20* GPU's.

Then `GPU_compute_segments_contrib()` computes in parallel :

- every pixel coordinates for all $16N_n$ segments. Since the snake is only read in one direction, we have been able to use a very simple parallel algorithm instead of Bresenham's. It is based on the slope k of each segment : one pixel per row if $|k| > 1$, one pixel per column otherwise.
- every pixel contribution by reading the corresponding values in the lookup tables.
- every thread blocks sums of individual pixel contributions by running a *reduction* stage for each block.

The top line of Figure 8 shows the base data structure in GPU shared memory which is relative to one segment. We concatenate the single segment structure as much as necessary to create a large vector representing every pixel of every test segment. As each segment has a different size (most often different from any power of two), there is a non-neglectable number of inactive threads scattered in the whole structure. Two stages are processed separately : one

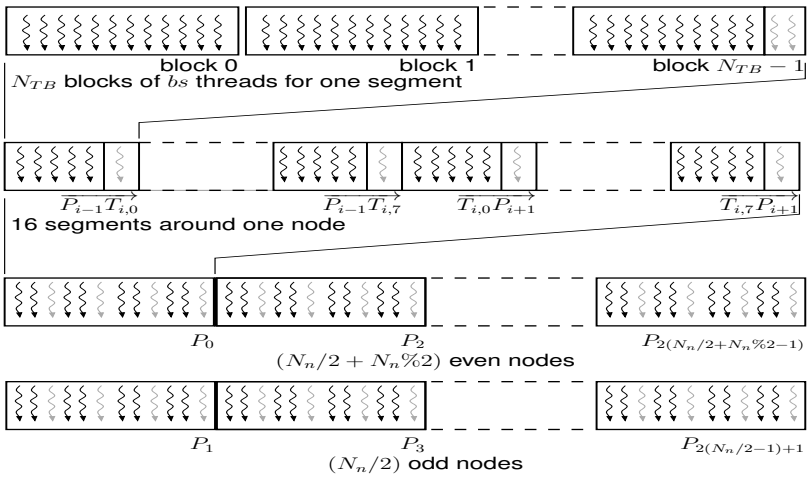


Figure 8. structure for segments contributions computation. Gray symbols help to locate inactive threads as opposed to black ones that figure active threads.

for all even nodes and another one for odd nodes, as shown in the two bottom lines of Figure 8.

The process is entirely done in shared memory ; only a small amount of data needs to be stored in global memory for each segment :

- the coordinates of its middle point, in order to be able to add nodes easily if needed.
- the coordinates of its first and last two points, to compute the slope at each end of the segment.

The five values above are part of the weighing coefficients determination for each segment and node.

The `GPU_sum_contribs()` takes the blocks sums obtained by `GPU_compute_segments_contrib()` and computes a second stage parallel summing to provide the $16N_n$ segment contributions.

C. Segments with a slope k such as $|k| \leq 1$

Such a segment is treated with 1 thread per column and consequently, it often has more than one pixel per row as shown by Figure 9. In an image row, consecutive pixels which belong to the target define an interval which can only have one low and one high ends That's why, on each row, we choose to consider only the contributions of the innermost pixels. This selection is also done inside `GPU_compute_segments_contrib()` when reading the lookup tables for each pixel contribution. We simply set a null contribution for pixels that need to be ignored.

D. Parameters estimation

A `GPU_compute_PLH()` kernel computes in parallel :

- every $8N_n$ vector of parameters values corresponding to each possible next state of the snake. Summing is done in shared memory but relevant data for these operations are stored in global memory.
- every associated pseudo likelihood value.

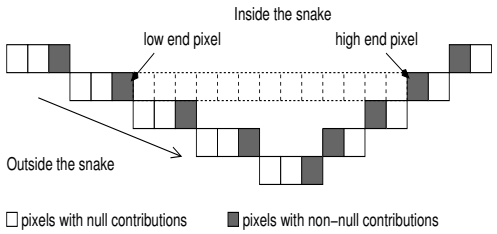


Figure 9. Zoom on part a of segment with $|k| < 1$, at pixel level.

- node substitutions when better PLH have been found and if it does not lead to segments crossing.

E. End of segmentation

Segmentation is considered achieved out when no other node can be added to the snake (Algorithm 3). A very simple GPU kernel adds every possible node and returns the number it added.

VI. Speedups

The CPU (SSE) implementation by N. Bertaux from the PhyTI team, based on [**?**] has been our reference to ensure segmentation's quality and to estimate speedups. Results are given in Table I. CPU timings were measured on an Intel Xeon E5530-2.4GHz with 12Go RAM (LIFC cluster). GPU timings were obtained on a C2050 GPU with 3GB RAM (adonis-11.grenoble.grid5000.fr).

Execution times reported are means on ten executions. The image of figure 1a (scaled down for printing reasons) is a 16-bit gray level photo from PhyTI team, voluntarily noisy for testing reasons. The contrast has been enhanced for better viewing.

We separately give the timings of pre-computations as they are a very general purpose piece of code. Segmentations have been performed with strictly the same parameters

Algorithm 3: Parralel GPU algorithm : outlines. <<<...>>> indicates a GPU kernel parallel process.

load images;
transfer image from CPU to GPU;
<<<compute the 2 cumulated images>>>;
<<<initialize the snake>>>;
repeat /* iteration level */
  repeat /* step level */
    <<<find best neighbor snake>>>;
    <<<adjust node's positions>>>;
    transfer the number of moves achieved from GPU memory to CPU memory.
  until *no more node can be moved*;
  <<<Add nodes>>>;
  transfert the number of nodes added from GPU memory to CPU memory.
until *no more node can be added*;

		CPU	GPU	Speedup
Image 15MP	**total**	**0.51 s**	**0.06 s**	**x8.5**
	pre-comp.	0.13 s	0.02 s	x6.5
	segment.	0.46 s	0.04 s	x11.5
Image 100MP	**total**	**4.08 s**	**0.59 s**	**x6.9**
	pre-comp.	0.91 s	0.13 s	x6.9
	segment.	3.17 s	0.46 s	x6.9
Image 150Mp	**total**	**5.7 s**	**0.79 s**	**x7.2**
	pre-comp.	1.4 s	0.20 s	x7.0
	segment.	4.3 s	0.59 s	x7.3

Table I
GPU (C2050, SM20) VS CPU TIMINGS.

(initial shape, threshold length). The neighborhood distance for the first iteration is 32 pixels. It has a slight influence on the time process, but it leads to similar speedups values of approximately 7 times faster than CPU.

Though it does not appear in Table I, we observed that during segmentation stage, higher speedups are obtained in the very first iterations, when segments are made of a lot of pixels, leading to a higher parallelism ratio.
Several parameters prevent from achieving higher speedups :

- accesses in the lookup tables in global memory cannot be coalescent. It would imply that the pixel contributions of a segment are stored in consecutive spaces in C_z and C_{z^2}. This is only the case for horizontal segments.
- the use of 64-bit words for computations in shared memory often leads to 2-way bank conflicts.
- the level of parallelism is not so high, ie. the total number of pixel is not large enough to achieve impressive speedups. For example, on C2050 GPU, a grid can run about 66 million of threads, but a snake in a 10000 x 10000 image would be less than 0.1 million pixel long.

VII. CONCLUSION

The algorithm we have focused on is not easy to adapt for high speedups on GPGPU, though we managed to make it work faster than on CPU. The main drawback is clearly its relative low level of parallelism. Nevertheless, we proposed different kernels that allowed us to take advantage of the computation power of GPUs. In future works, we plan to try and manage to benefit from larger computing grids of thread blocks. Among the possible solutions, we plan to work on:

- slicing the image and proceeding the parts in parallel. This is made possible since sm20 GPU provide multi kernel capabilities.
- slicing the image and proceeding the parts on two different GPUs, hosted by the same CPU.
- translating the parallelism from pixel level (*1 thread per pixel*) to snake level (*1 thread per snake*), at least during the first iteration, which is often the longest lasting one.
- designing an algorithm, in a GPU way of thinking, instead of adapting the existing CPU-designed algorithm to GPU constraints as we did.