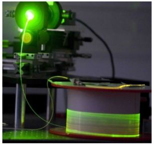
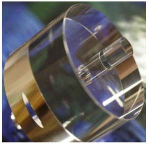
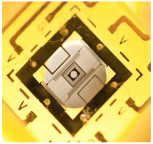
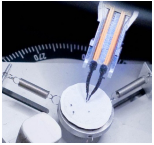
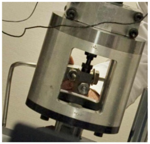
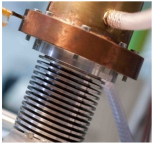
INSTITUT DE RECHERCHE FEMTO-ST

GPU-accelerated snake.

Implementation of a region-based segmentation algorithm (snake).

Gilles Perrot

Image segmentation – Definition, goals

- Dividing an image in two homogeneous regions.
- Reducing the amount of data needed to code information.
- Helping the human perception in certain cases.

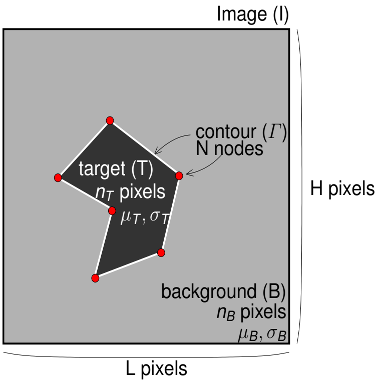
Image segmentation – Definition, goals

- Dividing an image in two homogeneous regions.
- Reducing the amount of data needed to code information.
- Helping the human perception in certain cases.

Images of our interest – Characteristics

- 16 bit-coded gray levels,
- From 1 Mpixels to more than 100 Mpixels,
- Corrupted by additive white Gaussian noise.

Algorithm basics – The criterion



- The goal is to find the most likely contour Γ (number and positions of nodes).
- The criterion used is a *Generalized Likelihood* one .
  In the Gaussian case, it is given by

$$GL = \frac{1}{2}\left[n_B.log\left(\widehat{\sigma_B}^2\right) + n_T.log\left(\widehat{\sigma_T}^2\right)\right]$$

where $\widehat{\sigma_\Omega}$ is the estimation of the deviation σ for the region Ω.

Algorithm basics – Parameters estimation

- In the Gaussian case, Probability Density Function (PDF) p_Ω has two parameters, average μ_Ω and standard deviation σ_Ω, which are estimated by maximum likelihood.
  If z is the gray level of the pixel of coordinates (i, j) :

$$\left(\begin{array}{l} \widehat{\mu_\Omega} = \frac{1}{n_\Omega} \sum_{(i,j)\in\Omega} z(i,j) \\ \widehat{\sigma^2_\Omega} = \frac{1}{n_\Omega} \sum_{(i,j)\in\Omega} (z(i,j) - \widehat{\mu_\Omega})^2 \end{array} \right.$$

- These estimations have to be computed for each test state of the contour Γ : time-consuming.

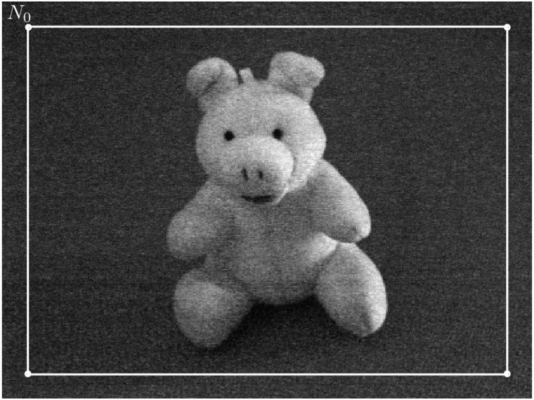
Algorithm basics – Parameters estimation

- In the Gaussian case, Probability Density Function (PDF) p_Ω has two parameters, average μ_Ω and standard deviation σ_Ω, which are estimated by maximum likelihood.
  If z is the gray level of the pixel of coordinates (i, j) :

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

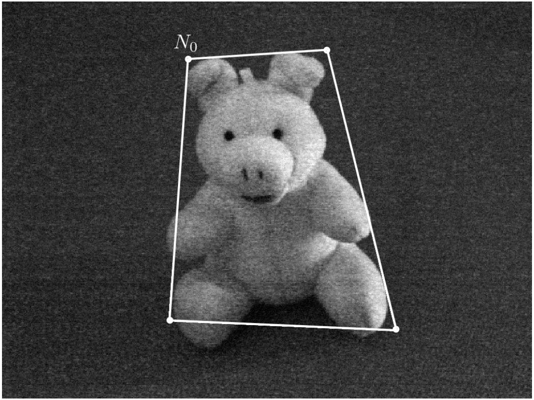
- These estimations have to be computed for each test state of the contour Γ : time-consuming.
- Based on the Green-Ostogradsky theorem, Chesnaud has shown how to replace those 2-dimensions sums inside the contour by 1-dimension sums along the contour.
- This optimization implies the precomputation of three cumulated images, each one containing a single parameter needed to compute the corresponding pixel's *contribution* to the above sums.

Algorithm basics – Illustration



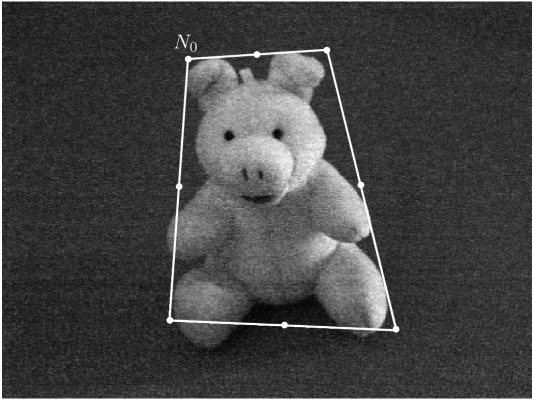
- 15 Mpixels image (SSE implementation limit).
- Initial contour : 4 nodes.

Algorithm basics – Illustration



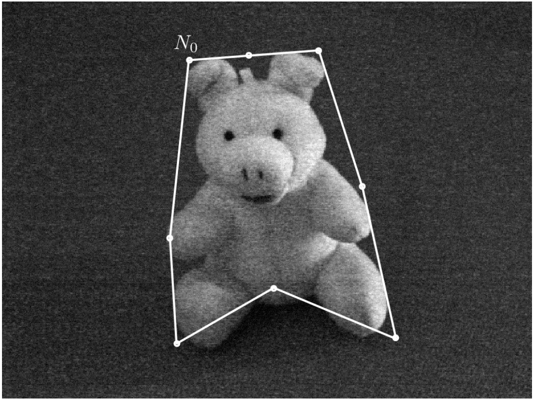
- End of first iteration : no more move can be of interest.

Algorithm basics – Illustration



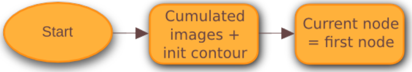
- Nodes added in the middle of segments.

Algorithm basics – Illustration

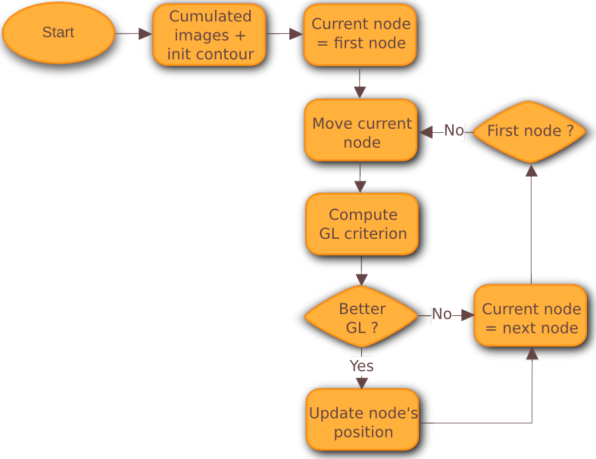


- End of second iteration.

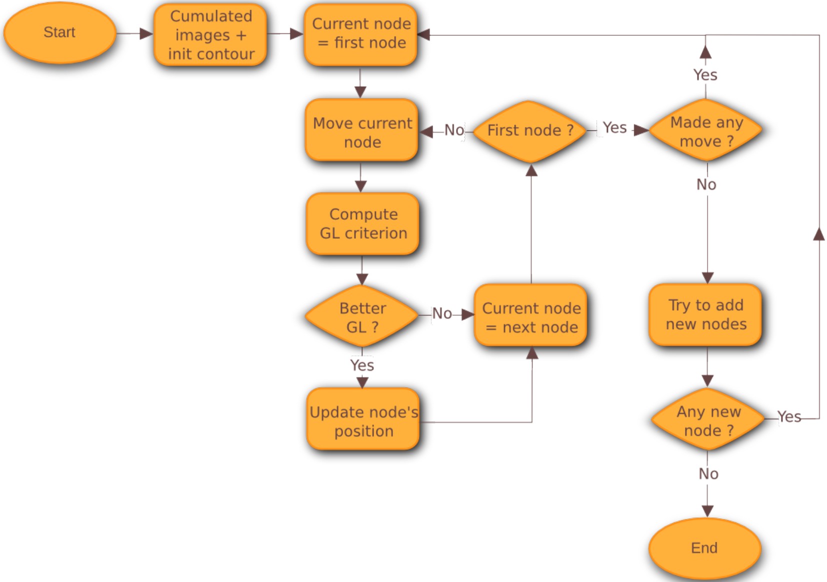
Algorithm basics – Simplified flowchart



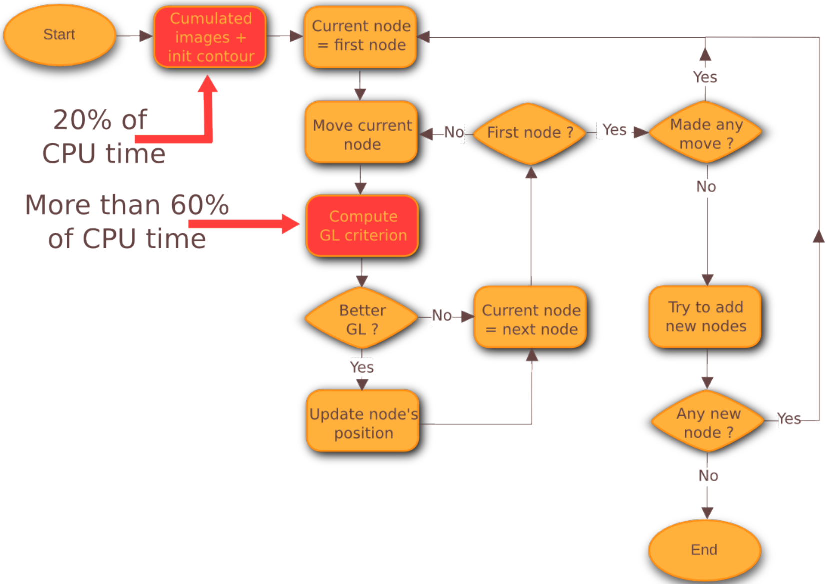
Algorithm basics – Simplified flowchart



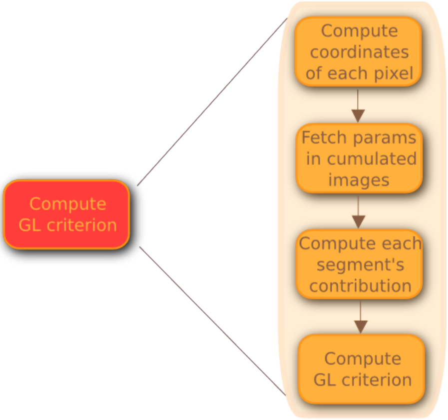
Algorithm basics – Simplified flowchart



Algorithm basics – Simplified flowchart



Algorithm basics – GL criterion computation



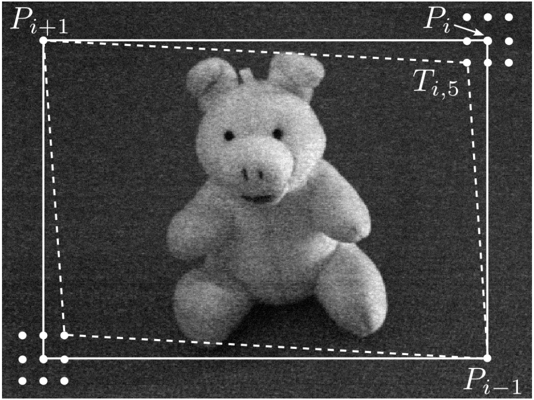
GPU implementation

Design facts

- Parallelism needs reside essentially in two clearly identified code blocks : GL criterion (60%) and cumulated images (20%)
- Keeping data in GPU memory avoids costly transfers. Thus, the whole computation is being performed by the GPU.
- Parallelism level is set to one thread per pixel of the contour. Alternate choices lead to lower performance (1 thread/segment, 1 thread/contour).
- The innermost loop, among contour nodes, is being parallelized.
- The two other loops are CPU driven. Each loop only requires a single byte of data to be transferred from GPU to CPU at each step.
- The precomputations of two cumulated images are being parallelized. The element values of the third cumulated image are being computed on the fly.

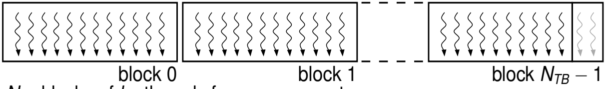
GPU implementation

GL criterion – Parallelism level



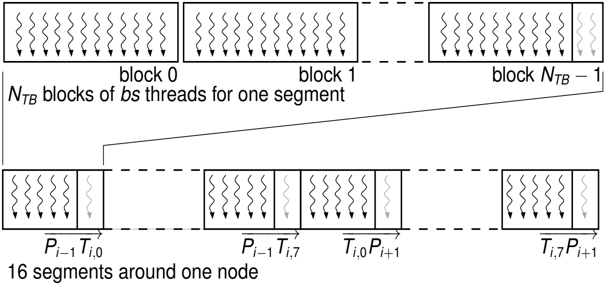
- 8 test positions around each node P_i, denoted $T_{i,0}$ to $T_{i,7}$.
- Each test position defines a pair of segments.
- Each combination of test positions defines a contour.
- GPU implementation evaluates in parallel every possible combination of test positions
- Even nodes and odd nodes are moved independently.

GL criterion – Data structure

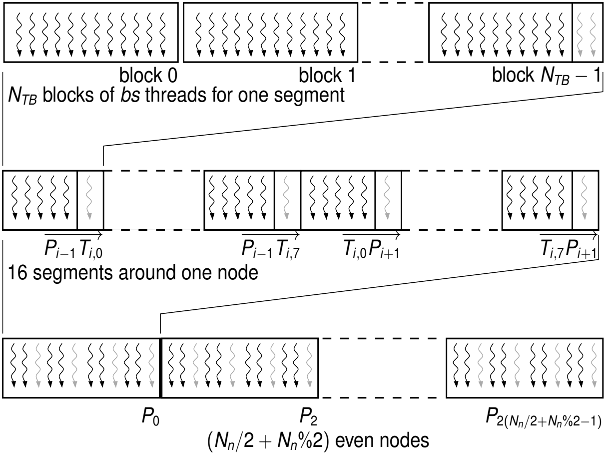


N_{TB} blocks of *bs* threads for one segment

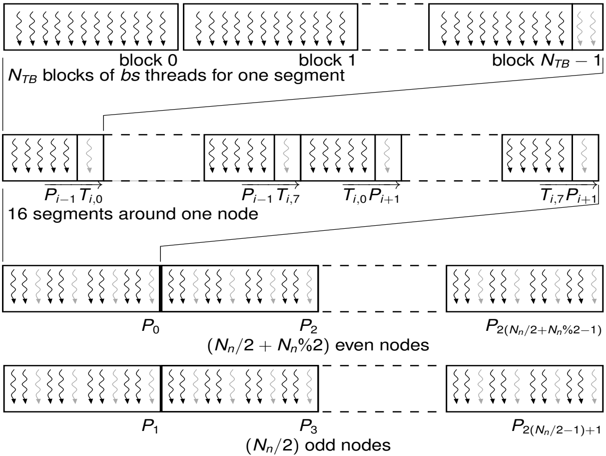
GL criterion – Data structure



GL criterion – Data structure



GL criterion – Data structure



GPU implementation

GL criterion – Parallel computations

- Each active thread corresponds to a pixel of a segment. There are inactive threads.
- The main kernel (kernel1) computes coordinates of every pixels, fetches parameter values from cumulated images and achieves the first reduction stage at block level.
- Another kernel (kernel2)achieves a second reduction stage, providing individual segments contributions.
- A third kernel (kernel3)computes the GL criterion value of each evaluated contour and returns the best.

Focus on kernel1 – Coordinates of pixels

```
__global__ void kernel1(int nb_nodes, uint32 npix_max, int l, uint2 *liste_pix, bool pairs)
    // indexes of elements
    int blockSize = blockDim.x ;
    int tib = threadIdx.x ;
    int nblocs_noeud =  gridDim.x / (nb_nodes/2 + (nb_nodes%2)*pairs) ;
    int nblocs_seg = nblocs_noeud / 16 ;
    int id_interval = blockIdx.x / nblocs_noeud ;
    int id_segment = ( blockIdx.x - id_interval*nblocs_noeud )/nblocs_seg ;
    int tis = idx - ( id_interval*nblocs_noeud + id_segment*nblocs_seg )*blockDim.x ;

    //in registers
    uint2 p ;
    int dx=x2-x1;
    int dy=y2-y1;
    int abs_dx = ABS(dx);
    int abs_dy = ABS(dy);
    int nb_pix = abs_dy>abs_dx?(abs_dy+1):(abs_dx+1);
    int incx, incy ;

    if (tis < nb_pix){
       if (abs_dy > abs_dx){
        //1 thread per row
        double k = (double)dx/dy ;
        p = make_uint2( y1 + incy*tis , x1 + floor((double)incy*k*tis+0.5) ) ;
       } else {
        //1 thread per column
        double k=(double)dy/dx ;
        p = make_uint2( y1 + floor((double)(incx*k*tis)+0.5) , x1 + incx*tis ) ;
      }
    }
    __syncthreads();
```

GPU implementation

Focus on kernel1 – Coordinates of pixels

- A Bresenham algorithm is not efficient here to discretize segments into a set of individual pixels. It would generate too much branches in the code.
- As contour is always processed counterclockwise, we use a simpler but more efficient method.
- There are only few branches in the code but they do not lead to any overhead compared to sequential Bresenham.
- Not enough computation to perform. Unable to hide arithmetic operations latency.
- Use of thread registers for better throughout.

Focus on kernel1 – Segments contributions, fetching pixel's parameters

```
//shared memory vectors
extern __shared__ t_sum_1 scumuls_1[] ;
t_sum_x * scumuls_x = (t_sum_x*) &scumuls_1[CFI(blockDim.x)] ;
t_sum_x2 * scumuls_x2= (t_sum_x2*) &scumuls_x[CFI(blockDim.x)] ;

if ( (tis >0) && (tis < nb_pix-1)
   && ( ((abs_dy <= abs_dx) && ( ( xprec > p.x) || ( xsuiv > p.x)))
    || (abs_dy > abs_dx) ) )
{
  //fetch two parameters in the cumulated images, the third is computed on the fly.
  int pos = p.x * l + p.y ;
  scumuls_1[ CFI(tib)] = 1 + p.y ;
  scumuls_x[ CFI(tib)] = cumul_x[ pos ] ;
  scumuls_x2[CFI(tib)] = cumul_x2[ pos ];
} else {
  //pixel with null contribution and padding in the last block of each segment
  scumuls_1[ CFI(tib)] = 0;
  scumuls_x[ CFI(tib)] = 0;
  scumuls_x2[CFI(tib)] = 0;
}
__syncthreads();
```

Focus on kernel1 – Segments contributions, fetching pixel's parameters

- No possible coalescence for global memory accesses as segment's geometry always vary.
- Two ways shared memory bank conflicts exists as shared data is 64 bits-coded.
- But shared memory is still the best choice because of the reduction to be done.

Focus on kernel1 – Segments contributions, first reduction stage

```
uint offset;
#pragma UNROLL
for (offset=1024; offset>32; offset/=2) {
  if (blockSize >= 2*offset) {
    if (tib < offset) {
      scumuls_1[ CFI(tib)] += scumuls_1[ CFI(tib + offset) ];
      scumuls_x[ CFI(tib)] += scumuls_x[ CFI(tib + offset) ];
      scumuls_x2[CFI(tib)] += scumuls_x2[CFI(tib + offset) ];
    }
    __syncthreads();
  }}

if (tib < 32) {
  #pragma UNROLL
  for (offset=32; offset>0; offset/=2 )
  {
    scumuls_1[ CFI(tib)] += scumuls_1[ CFI(tib + offset) ];
    scumuls_x[ CFI(tib)] += scumuls_x[ CFI(tib + offset) ];
    scumuls_x2[CFI(tib)] += scumuls_x2[CFI(tib + offset) ];
  }}

if (tib == 0) {
  gsombloc[ blockIdx.x ] = scumuls_1[0] ;
  gsombloc[ blockIdx.x + gridDim.x ] = scumuls_x[0] ;
  gsombloc[ blockIdx.x + 2*gridDim.x ] = scumuls_x2[0] ;
}
```

Focus on kernel2 – Segments contributions, second reduction stage

```
__global__ void somsom_full(uint64 * somblocs, int nb_nodes, unsigned int nb_bl_seg,
                            uint64 * somsom, bool pairs){
  //registers
  uint64 sdata[3];
  unsigned n = nb_nodes/2 + pairs*(nb_nodes%2) ;
  unsigned int seg = blockIdx.x ;
  unsigned int nb_seg = 16*n ;

  //1 thread per segment
  sdata[0] = 0;
  sdata[1] = 0;
  sdata[2] = 0;

  for (int b=0; b < nb_bl_seg ; b++){
    sdata[0] += somblocs[seg*nb_bl_seg + b];
    sdata[1] += somblocs[(seg + nb_seg)*nb_bl_seg + b];
    sdata[2] += somblocs[(seg + 2*nb_seg)*nb_bl_seg + b];
  }

  //sums ~ segment contribution --> global memory
  {
   somsom[3*seg] = sdata[0];
   somsom[3*seg + 1] = sdata[1];
   somsom[3*seg + 2] = sdata[2];
  }
}
```

GPU implementation

Results and analyse

- On images of size between 10 and 150 Mpixels : speedup around x7 compared with SSE2 implementation.
- The first iteration is fast, while the followings are sometimes lower than the reference CPU implementation. It is due to larger segments and thus less inactive threads in the grid.
- Algorithm that do not fit very well GPU architecture.
- Numerous lines of code. For example 200 lines vs 20 lines to compute cumulated images.

Conclusion

- Speedups are not so impressives.
- Another data structure may be more suited.
- A 2D process would be far easier to code, but would not bring such a speedup that the 2D→1D transform brought.
- When designing an algorithm, the targetted host properties should be taken into account.
- Some processes may actually not be suited to GPUs.