Preliminary project proposals, autumn 2015

All projects can be taken by one or two students. In case of two students, each student takes a different approach to solve the problem (different ways of solving the problem, different flow graphs, using DPS, MPI, OpenMP or CUDA for GPU to parallelize the problem). Unless indicated otherwise, we expect at least one implementation per person. Two different DPS flow graphs correspond to two different implementations. Each group must have at least one distributed memory or GPU implementation, using MPI, DPS or CUDA. Further implementations can be created for multi-cores with OpenMP.

Project report

Every project will be accompanied by a detailed report, which must at least contain the following elements:

Theoretical part

1. A brief description of the application and its computational complexity (O(n), O($n$log$n$), …). Provide an upper bound of the achievable speedup using Amdahl's law by considering the serial and parallelizable parts of the considered application. In order to have a more accurate prediction, you may also use Amdahl's law to estimate the potential speedup of your parallel application by considering as serial time the time needed to distribute the initial data to the slaves which may then start their computation and the time for collecting the results back on the master node. Knowing the amount of data to be sent in terms of data size and number of messages, the latency (Gigabit Ethernet: 300μs) and the throughput of the network (Gigabit Ethernet: 66 MB/s), you can predict the time required for the initial data distribution.
2. Consider different parallelization strategies, and describe the pros and cons of each one of them. Establish the computational complexity of their computation to communication ratio.
    a. If you use DPS, draw the DPS flow graph that implements your solution: flow graphs, data object contents, thread collections.
    b. If you use MPI, draw a message-passing graph that describes your implementation, and specify the content of the transferred messages.
    c. If you use OpenMP, describe the flow of your program as a sequence of serial and parallel sections, possibly with the type of data decomposition and scheduling used.
    d. Provide a detailed theoretical analysis of one or more of your strategies using a *timing diagram* taking into account the computing and communication times, the pipelining of operations, etc. From the critical path, derive a theoretical model enabling predicting the *speedup* of the application on a variable number of nodes (or cores or wars). Some applications may have data dependent behaviors preventing a general theoretical model from being derived. In such cases, make some assumptions and derive a model for these particular cases. Compare the results with your initial guess using Amdahl's law.

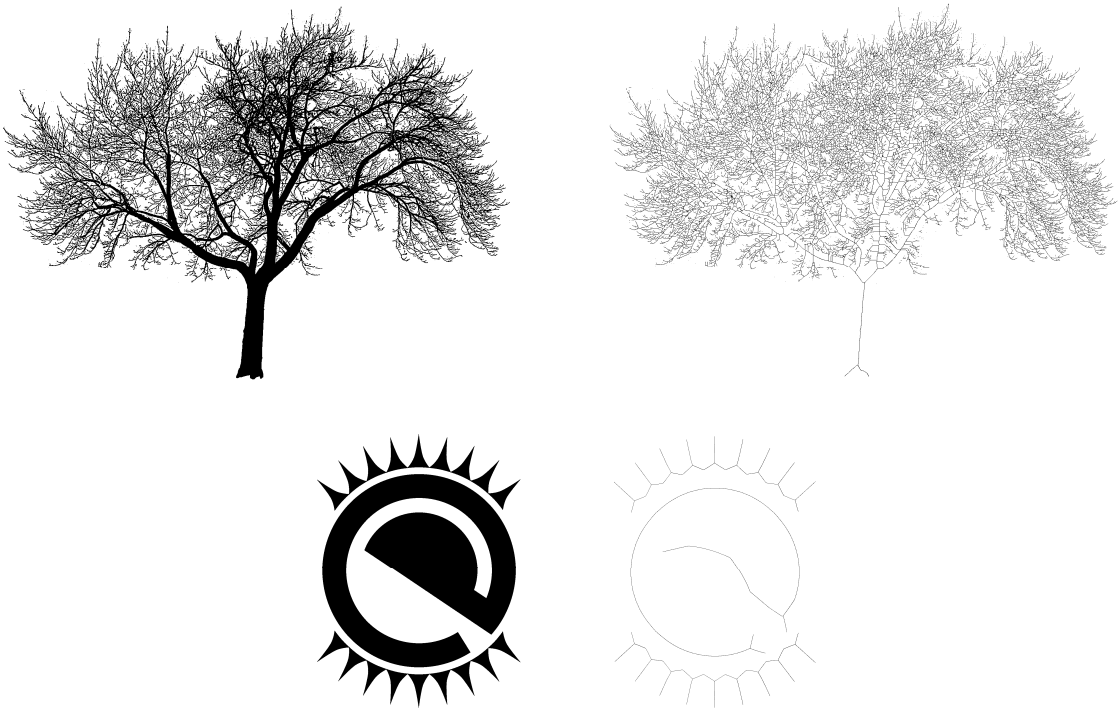
       If you use CUDA for running on GPU, give a timing diagram indicating the different program phases such as transferring from the PC memory to the GPU global memory, transfer of data from GPU global memory to GPU shared memory, running a part of the program in parallel on the GPU, transfer of results from GPU shared memory to GPU global memory and possibly to the PC main memory, terminating the program on the PC host. Consider starting with 32 cores (1 warp) and varying the number of threads in multiples of 32 (multiple warps) until you reach the maximal number of cores. You may also predict the speedup in respect to the serial execution time on a normal PC (see course slides about GPU's and CUDA on Moodle).
3. If applicable, discuss possible optimizations of the parallel implementation and of the algorithm.
4. Depending on the complexity of the application, implement one or several of the described strategies, and describe the related issues.

5. Evaluate the performance of the application on at least 1, 2, 4 and 8 distributed memory nodes or with CUDA, with 32, 64, 96, 128, 160, 192 data threads (cores). Draw a speedup graph combining the practical absolute (in respect to a serial execution), relative (in respect to a parallel execution on a single node (or warp)) and the theoretical speedup, as well as Amdahl's law. Mention the reference execution time. Discuss any difference between your original prediction and the actual measured speedup and provide a corrected model if needed. If applicable, compare practical results for different sizes of inputs, the impact of the different optimizations, etc.

The *theoretical prediction report* must contain parts 1 to 2. The final report is an extension of the theoretical prediction report that includes parts 3 to 6 and may include corrections of the original prediction model.

The report should be fairly brief and include figures and graphs. Please hand in the printed report together with a CD-ROM/DVD containing the report files as well as all program sources, binaries and result files.

Image skeletonization



Algorithm: We decide for each pixel P1 whether it is deleted by considering information about its 8 neighbour pixels P2, P3, …, P9. Let $\boldsymbol{TR(P_1)}$ be the number of **background (white=0)** to **foreground (black=1)** transitions in the neighbourhood of P_1, and $\boldsymbol{NZ(P_1)}$ the number of foreground (black) neighbours of P_1. P_1 is erased, i.e. it becomes background (white) if all the following conditions are true:

P_3	P_2	P_9
P_4	P_1	P_8
P_5	P_6	P_7

$$\begin{cases} 2 \le NZ(P_1) \le 6 \\ TR(P_1) = 1 \\ P_2 \cdot P_4 \cdot P_8 = 0 \ \text{ or } \ TR(P_2) \ne 1 \\ P_2 \cdot P_4 \cdot P_6 = 0 \ \text{ or } \ TR(P_4) \ne 1 \end{cases}$$

The algorithm is repeated until the image does not change any more.

References:

Dynamic load balancing of parallel cellular automata
http://diwww.epfl.ch/w3lsp/publications/gigaserver/dlbopca.pdf

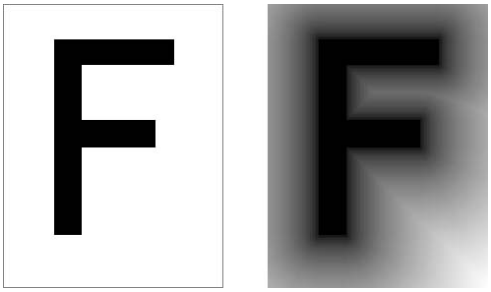
and
A.K. Jain, Fundamentals of Digital Image Processing, Prentice-Hall, 1989, 382-383

Use the helper files lspbmp.h and lspbmp.c to manipulate images.

Assistant: Petar Pjanic, R.D. Hersch (presenter)

Distance transform

Transforms a binary image into a greyscale image, where the value of each pixel is given by the distance between the current pixel and the nearest black pixel. Distances may be computed as Manhattan, chessboard, quasi-Euclidian (Chamfer) or Euclidean distances.



Algorithm:

- Initialize binary image shape pixels with 0 and all other pixels with a large number (larger than the largest possible distance in the image, according to the selected distance metric).
- Choose your distance metric $D(p,q)$, where p is the current pixel location (p_x, p_y) and q is one of the neighbour pixel locations (4 neighbours or 8 neighbours), (q_x, q_y).

  For the *cityblock* (or *Manhatten*) metric: $D^4(p,q)=1$ where q is one of the 4 direct neighbours of p.

  For the *chessboard* metric : $D^8(p,q)=1$ where q is one of the 8 direct or indirect neighbours of p.

  For the *Chamfer* metric: direct neighbours have a distance of 3 and indirect neighbours a distance of 4.

4	3	4
3	x	3
4	3	4

  For the *Euclidian* metric : D(p,q) = $D(p,q)=\sqrt{(p_x-q_x)^2+(p_y-q_y)^2}$

- Two passes across the image:

  For the 1st pass, traverse the image from left to right scanline after scanline, from top to bottom. For each pixel, compute its new intensity $F(p)$ according to the upper and left neighbours as follows

q	q	q
q	**p**	

  $F(p) = \min [F(p), D(p,q) + F(q)]$

  For the 2nd pass, traverse the image from right to left scanline after scanline, from bottom to top. For each pixel, compute its new intensity $F(p)$ according to the lower and right neighbours as follows

	p	q
q	q	q

  $F(p) = \min [F(p), D(p,q) + F(q)]$

References:

http://en.wikipedia.org/wiki/Distance_transform
http://people.cs.uchicago.edu/~pff/dt/ (Provides a serial implementation)

Use the helper files lspbmp.h and lspbmp.c to manipulate images.

Assistant: Romain Rossier

Hough transform

Find the equations of lines present in an image. Use Sobel (or another edge detection operator) to identify pixels that lie on an edge. The Hough space is then built as follows: for every pixel plot the distance to the origin and the angle of every line that goes through the pixel. Once all pixels have been processed, locate local maxima within the Hough space. The coordinates of each maximum identifies provides the equation of a line.

For more information, print file "HoughTransform_Hlavac_212_221.pdf" from the book of M. Sonka, V. Hlavac and R. Boyle, Image Processing, Analysis and Machine Vision, Thomson, pp. 212-221.

Use the helper files lspbmp.h and lspbmp.c to manipulate images.

Assistants: Petar Pjanic, R.D. Hersch (presenter)

Noise reduction in images by averaging over a selected neighborhood

Noise is assumed to be an independent random variable with zero mean and standard deviation and is assumed to be additive to the original intensity values. We consider the distribution of mean values over several image realizations. The mean values computed with n samples have a standard deviation of $\sigma/\sqrt{n}$ where σ is the standard deviation of the population. Therefore, averaging intensity values in small regions where intensity is quasi-uniform reduces the noise.

In order to select the most uniform region around one pixel, one may consider 3x3 masks, where, for a given mask orientation, the current pixel is one of the boundary pixels. One may compute the dispersion σ^2 for each mask orientation and calculates the mean over the mask orientation with the lowest dispersion.

See file "AveragingRotatingMask_Hlavac_127-129.pdf.

Create noisy images by taking a standard grayscale image and adding to each k^{th} pixel (e.g. k=3) random noise with mean value zero and a given standard deviation (e.g. σ= 20% of the maximum intensity).

Use the helper files lspbmp.h and lspbmp.c to manipulate images.

Assistants: R. Rossier, R.D. Hersch (presenter)

Noise reduction in images by median filtering

The median separates the higher half of a probability distribution from the lower half. Median filtering filters noise and at the same time reduces the blurring of the edges. It replaces the intensity of the current point in the image by the median of the intensities of its neighbourhood.

Algorithm:
For each pixel in the image, establish an ordered list of its own and neighbour pixel intensities (e.g. 8-neighbours) and select as the current pixel intensity the one in the middle of the ordered list.
A more efficient version of the algorithm is given in the file below.
See file "AveragingMedianFiltering_Hlavac_129-132.pdf.

Create noisy images by taking a standard grayscaling image and adding to each k^{th} pixel (e.g. k=3) random noise with mean value zero and a given standard deviation (e.g. σ= 20% of the maximum intensity).
Use the helper files lspbmp.h and lspbmp.c to manipulate images.

Assistants: R. Rossier, R.D. Hersch (presenter)

Boundary detection

Maxima of the first derivative of a function indicate steep locations (strong slope) and zero crossings of the 2nd derivative indicate locations of extrema. By combining a smoothing function (Gaussian) and the Laplacian (Laplacian of a Gaussian), one obtains the intermediate image $p(x,y)$ from which the zero crossings can be extracted.

Laplacian of Gaussian : $p(x,y) = \nabla^2 \left[G(x,y,\sigma) * f(x,y) \right] = \left[\nabla^2 G(x,y,\sigma) \right] * f(x,y)$

Gaussian : $G(r) = e^{-r^2/2\sigma^2}$ where $r^2 = x^2 + y^2$

First derivative of Gaussian : $G'(r) = -\frac{1}{\sigma^2} r \cdot e^{-r^2/2\sigma^2}$

Second derivative, which is the Laplacian of a Gaussian: $G''(r) = \frac{1}{\sigma^2}\left(\frac{r^2}{\sigma^2} - 1 \right) \cdot e^{-r^2/2\sigma^2}$

Therfore, the convolution mask becomes

$$h(x,y) = c\left(\frac{x^2 + y^2 - \sigma^2}{\sigma^4} \right) \cdot e^{-(x^2+y^2)/2\sigma^2}$$ where c normalizes the sum of mask elements.

Here is an example of a 5 x 5 discrete approximation of such a convolution mask:

$$\begin{bmatrix} 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & -2 & -1 & 0 \\ -1 & -2 & 16 & -2 & -1 \\ 0 & -1 & -2 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{bmatrix}$$

In order to extract the zero crossings, go over the image with a 2x2 window. Assign the edge label to the upper left corner. If within the 3 other window locations, there is a positive and a negative number, label the upper left corner pixel as "edge".
Try to find the boundaries with kernels of different sizes.

Reference: M. Sonka, V. Hlavac and R. Boyle, Image Processing, Analysis and Machine Vision, Thomson, pp. 138-142.
Assistants: R. Rossier, R.D. Hersch (presenter)

Bilateral filtering on GPU

The bilateral filter is used mainly for image denoising and enables keeping high frequency transitions (edges) intact in the filtered image.

An image filtered by Gaussian Convolution is given by:

$$GC[I]_{\mathbf{p}} = \sum_{\mathbf{q}\in\mathcal{S}} G_\sigma(\|\mathbf{p}-\mathbf{q}\|)\, I_{\mathbf{q}},$$

where $G_\sigma(x)$ denotes the 2D Gaussian kernel (see Figure 2.1):

$$G_\sigma(x) = \frac{1}{2\pi\sigma^2}\exp\left(-\frac{x^2}{2\sigma^2}\right).$$

When working on a grid with (u,v) coordinates, x^2 is calculated as follows:

$$x^2 = (p_u - q_u)^2 + (p_v - q_v)^2$$

The bilateral filter, denoted by $BF[\cdot]$, is defined by:

$$BF[I]_{\mathbf{p}} = \frac{1}{W_{\mathbf{p}}}\sum_{\mathbf{q}\in\mathcal{S}} G_{\sigma_s}(\|\mathbf{p}-\mathbf{q}\|)\, G_{\sigma_r}(|I_{\mathbf{p}}-I_{\mathbf{q}}|)\, I_{\mathbf{q}},$$

where normalization factor $W_{\mathbf{p}}$ ensures pixel weights sum to 1.0:

$$W_{\mathbf{p}} = \sum_{\mathbf{q}\in\mathcal{S}} G_{\sigma_s}(\|\mathbf{p}-\mathbf{q}\|)\, G_{\sigma_r}(|I_{\mathbf{p}}-I_{\mathbf{q}}|).$$

Conceive an algorithm and a program on CUDA for parallelizing the bilateral filter.

Use the brute force approach, but consider different image sizes and kernel sizes.
The standard kernel size is between $2\sigma \times 2\sigma$ and $5\sigma \times 5\sigma$.

Reference:
S. Paris, P. Kornprobst, J. Tumblin, F. Durand, Bilateral filtering: Theory and Application, Foundations and trends in Computer Graphics and Vision, Vol. 4, No. 1, 1-73 (2008)

http://people.csail.mit.edu/sparis/bf_course/

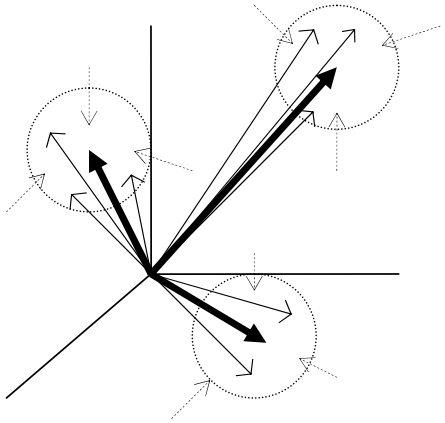
Responsible: R. D. Hersch

Vector Quantization for creating colour lookup tables

We want to build a small set of colours that can be used to effectively display a true colour image. To construct this set of colours, we use the following algorithm:

1. Start with a desired number of colors (set of 16, 32, 64 or 256 colors) with which your image will be represented. Evenly distribute all pixels of the image onto the colour set, assigning pixels as 'belonging' to a certain colour of the set. Once all pixels have been assigned, for each colour of the set, set its representative value to the average colour of all the pixels that have been assigned to it.
2. For each pixel of the image, compute the Euclidean distance that separates it from each representative colour in the set. Assign it to the colour to which it is closest.
3. For each colour in the set, recompute its average representative colour based on all the pixels that have been assigned to it.
4. Continue from step 2, until no pixels have moved from one set to another.

Create the color image with the resulting color lookup table.



Use the helper files lspbmp.h and lspbmp.c to manipulate images.

R. Rossier (presenter), R.D. Hersch

Sieve of Eratosthenes

Finds prime numbers according to the following algorithm:

1) Write a list of numbers from 2 to the largest number you want to test for primality. Call this List A.
2) Write the number 2, the first prime number, in another list for primes found. Call this List B.
3) Strike off 2 and all multiples of 2 from List A.
4) The first remaining number in the list is a prime number. Write this number into List B.
5) Strike off this number and all multiples of this number from List A. The crossing-off of multiples can be started at the square of the number, as lower multiples have already been crossed out in previous steps.
6) Repeat steps 4 through 6 until no more numbers are left in List A.

Variant: The sieve of Atkin (which is computationally more efficient) can be considered instead.

References:

http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes
http://en.wikipedia.org/wiki/Sieve_of_Atkin

Assistants: V. Keller, R.D. Hersch (presenter)

Fast Fourier Transform

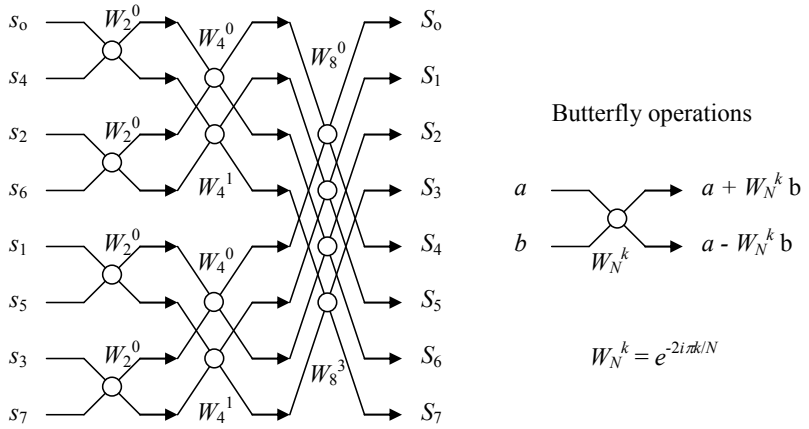
Each term of the discrete Fourier transform (DFT) of a complex vector x of size N is computed as

$$X_k = \sum_{n=0}^{N-1} x_k e^{-(2i\pi/N)nk}$$

Computing the N values of X therefore requires $O(N^2)$ operations. If N is even, we may split the vector x into two vectors e and o containing the even-indexed and the odd-indexed numbers of x. The formula may then be rewritten as

$$X_k = \begin{cases} E_k + e^{-2i\pi k/N} O_k & \text{if } k < N/2 \\ E_k + e^{-2i\pi(k-N/2)/N} O_k & \text{if } k \geq N/2 \end{cases},$$

where E is the DFT of e and O the DFT of o. If N is a power of 2, we may recurse until no DFT needs to be computed, reducing the complexity of the computation to $O(N\log N)$. The figure below illustrates an example with $N = 8$:



The project can be extended to a 2D FFT.

References:

Numerical recipes online: http://www.nrbook.com/a/bookcpdf/c12-2.pdf
http://en.wikipedia.org/wiki/Fast_Fourier_transform
http://en.wikipedia.org/wiki/Cooley-Tukey_FFT_algorithm

Use the random number generator found in the LU application in the DPS distribution to initialize your vector.
Assistants: V. Keller, R.D. Hersch (presenter)

Jacobi iterative linear equation solver

The Jacobi iterative method solves a linear system $Ax = b$ using the following recursion:

$$\begin{cases} x_0 = \text{random vector} \\ Dx_k = (D - A)x_{k-1} + b \\ \text{stop when } \left|x_k[i] - x_{k-1}[i]\right| < e, \ \forall i \end{cases}$$

where D stores the diagonal of A. Typical values for e are 10^{-6} or 10^{-8}.

Use the random number generator found in the LU application in the DPS distribution to initialize your matrices and vectors.
Assistants: V. Keller, R.D. Hersch (presenter)

Gaussian elimination with back-substitution

Given a linear system $Ax = b$ matrix, build the triangular matrix A' so as to obtain $A'x=b'$ by performing the following steps:

1. Divide the first row (i=1) as well as b_1 by a_{11};
2. for all following rows: starting with j=i+1:
   $\text{row}_j = \text{row}_j + \text{row}_i \,.\, (-a_{ji}/a_{ii})$;
   $b_j = b_j - b_i \,.\, (-a_{ji}/a_{ii})$;
3. i =i+1; continue at 2.

With partial pivoting (exchange of rows), make sure that diagonal element a_{ii} is largest or large enough within the row.

$$\begin{bmatrix} a'_{11} & a'_{12} & a'_{13} & a'_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ 0 & 0 & a'_{33} & a'_{34} \\ 0 & 0 & 0 & a'_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1' \\ b_2' \\ b_3' \\ b_4' \end{bmatrix}$$

References:

Numerical recipes online:

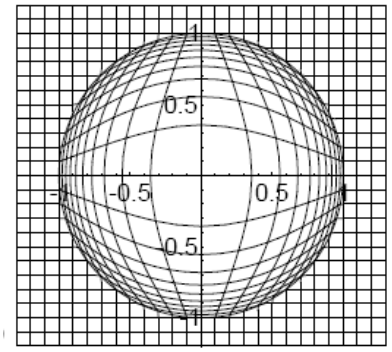
http://www.nrbook.com/a/bookcpdf/c2-1.pdf (Gauss-Jordan elimination)
http://www.nrbook.com/a/bookcpdf/c2-2.pdf (Gaussian elimination with back-substitution)

Use the random number generator found in the LU application in the DPS distribution to initialize your matrices and vectors.

Assistants: V. Keller, R.D. Hersch (presenter)

Fish-eye transformation with bilinear interpolation

The objective is to transform a bitmap image according to a geometric transformation mapping the target image space back into the original space. For each pixel (x,y) in the destination image, we can compute the corresponding coordinate in the source image (x',y'):



The forward transformation is given in polar coordinates by

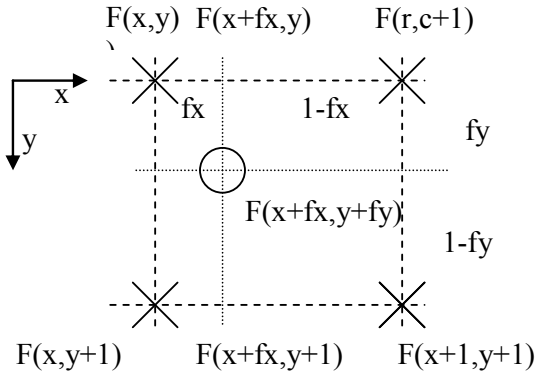
$$\theta' = \theta$$

$$r = \begin{cases} \dfrac{m\frac{r'}{1-r'}}{1+m\frac{r'}{1-r'}} & \text{if } r' < 1 \\ r' & \text{otherwise} \end{cases}$$

where m is a magnifying factor (e.g 2, 3 or 4).

The backward transformation, to be used to create the destination image is $\theta = \theta'; \quad r' = \begin{cases} \dfrac{1}{m \cdot \frac{1-r}{r} + 1} & \text{if } r < 1 \\ r & \text{otherwise} \end{cases}$

The result of the transformation will not necessarily be an integer. The simplest approach to resampling the source image is to simply select the nearest neighbour by rounding x_{source} and y_{source} to the nearest integer. This will result in a very blocky image when scaling up. A better solution is to use bilinear interpolation, where the four nearest pixels are considered, and weighted according to their distance from x_{source} and y_{source}. The following figure illustrates the process of bilinear interpolation, where x_{source} has been split into integer and fractional components x and fx, and y_{source} has been split into integer and fractional components y and fy. The interpolation has been split into horizontal and vertical components.



: pixel center of the original image

: point corresponding to a pixel center in the zoomed image

Interpolations:

F(x+fx,y) = (1-fx)·F(x,y) + fx·F(x+1,y)

F(x+fx,y+1) = (1-fx)·F(x,y+1) + fx·F(x+1,y+1)

F(x+fx,y+fy) = (1-fy)·F(x+fx,y) + fy·F(x+fx,y+1)

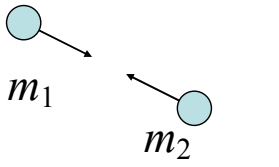
There are many other interpolation techniques that yield slightly higher image quality.

Use the helper files lspbmp.h and lspbmp.c to manipulate images.

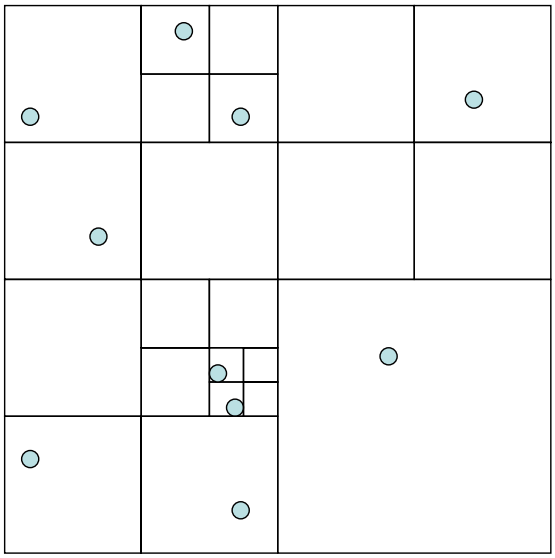
Assistants: R. Rossier (presenter)

N-body simulation

Simulate the gravitational attraction of multiple bodies. The gravitational attraction between two bodies is given by $F=G*(m_1*m_2)/(r*r)$, where m_1 and m_2 are the masses of the two bodies, and r is the distance separating them. The trajectory of motion of multiple bodies can be computed by using this equation and a simple numerical integrator. When the number of bodies to simulate is very large, it is useful to optimize the computation by clustering distant bodies into a quad-tree. Individual distant bodies may then be represented by the cluster they belong to. The mass of the cluster is given by the sum of the masses of the bodies, and its position is the center of gravity.



$$F = G \cdot \frac{m_1 \cdot m_2}{r^2}$$



Assistant: V. Keller, R.D. Hersch (presenter)

,

Convex hull in two dimension

Several algorithms exist for creating the convex hull of a set of points in two dimensions. For the sake of simplicity, we assume that 3 points are never collinear.

a) Giftwrapping algorithm

The so-called Gift wrapping algorithm starts with the lowest point as a first anchor point *j* and a horizontal line. We then compute the angular rotation of the line necessary to reach each of the points within the point set. This associates one angle α_{i1} to each point $i <> j$. The next candidate line segment is the line segment *ji* with the smallest angular deviation α_i .With the new line given by *ji*, we recompute the angular rotation α_{i2} to reach each non-processed point *i* and select to next point *i* with the smallest non-negative angular rotation

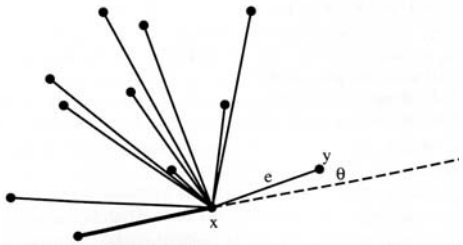


Fig 1: The next edge makes the smallest angle in respect to the previous edge.

Algorithm: GIFT WRAPPING
Find the lowest point (smallest y-coordinate).
Let i_0 be its index, and set $i \leftarrow i_0$.
repeat
  for each $j \neq i$ do
    Compute counterclockwise angle θ from previous hull edge.
  Let k be the index of the point with the smallest θ.
  Output (p_i, p_k) as a hull edge.
  $i \leftarrow k$
until $i = i_0$

The algorithm runs in $O(n^2)$ time: O(n) for each hull edge.

b) Quickhull algorithm

The quickhull algorithm concentrates on the points that lie exterior to a preliminary bounding polygon. We start we a quadrilateral given by the 4 extremal points (Fig. 2).

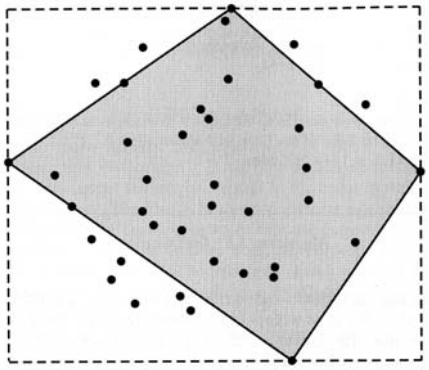


Fig. 2. First iteration of quickhull: bounding quadrilateral (current polygon)

For each segment *ab* of the current polygon, we compute the external point *c* which is furthest away (largest distance). The current segment *ab* is replaced by the corresponding two segments *ac* and *cb*.

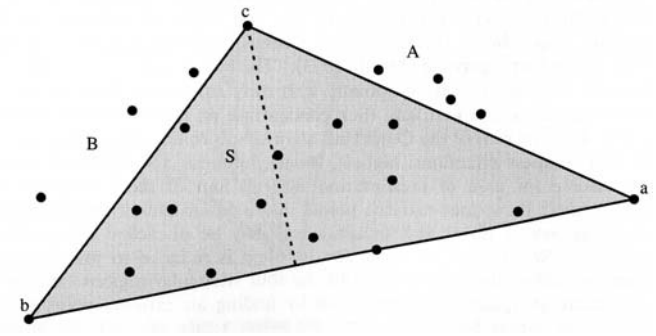


Fig. 3. Expansion of polygon boundary segment *ab* into *ac* and *cb*.

The quickhull algorithm runs in O(*n* log *n*) time for a well behaved point set, i.e. when the perpendicular segment to the furthest point divides the remaining points into two sets of points of similar size. In the worst case, the algorithm runs in $O(n^2)$ time.

c) Graham's algorithm

There exists a further algorithm called Graham's algorithm which is guaranteed to run in O(*n* log *n*) time.

We here also start from the lowest point and a horizontal line segment. We then sort the other points according to their turning angle (counterclockwise) in respect to the horizontal segment. We then try to create the convex hull segments from the current point to the next point in the list. We also maintain a stack and push each successive hull point onto the stack. We verify that each new point represents a left turn. If this is not the case, we pop from the stack the previous point and create the segment with the new point and verify that this segment represent a left turn with the prevous segment. If this is not the case, we again pop a point from the stack and create the segment with the new point, etc…

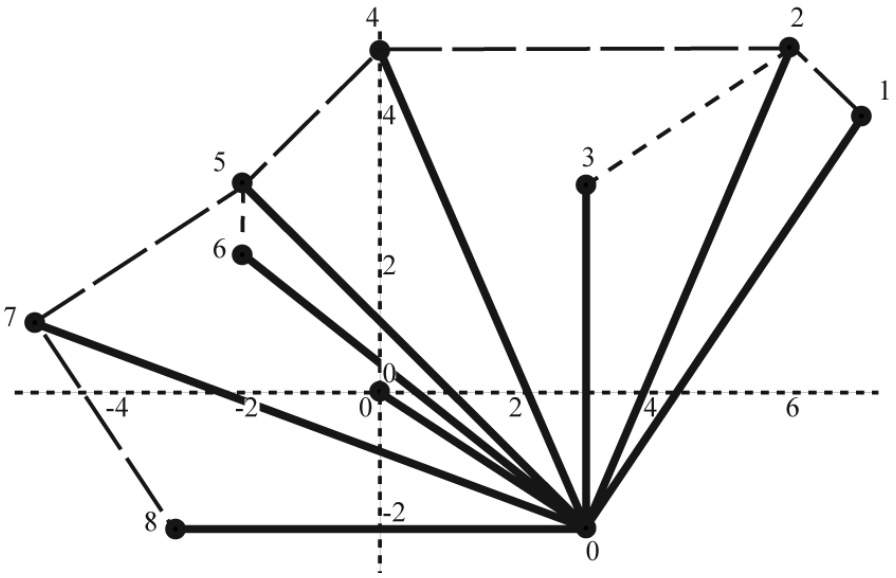


Fig. 4. Sorting the points according to their angle in respect to the horizontal line

d) Merging two spatially distinct hulls *A* and *B* into single hull

Two distinct hulls are merged by finding their mutual lower and upper tangents. We start with the right most point *a* of hull A and the leftmost point *b* of hull B. We then walk along successive points *a* of A and *b* of B, clockwise on A and counterclockwise on B so as to go down until the two neighbouring segments of *a* and of *b* are both above line *ab*.

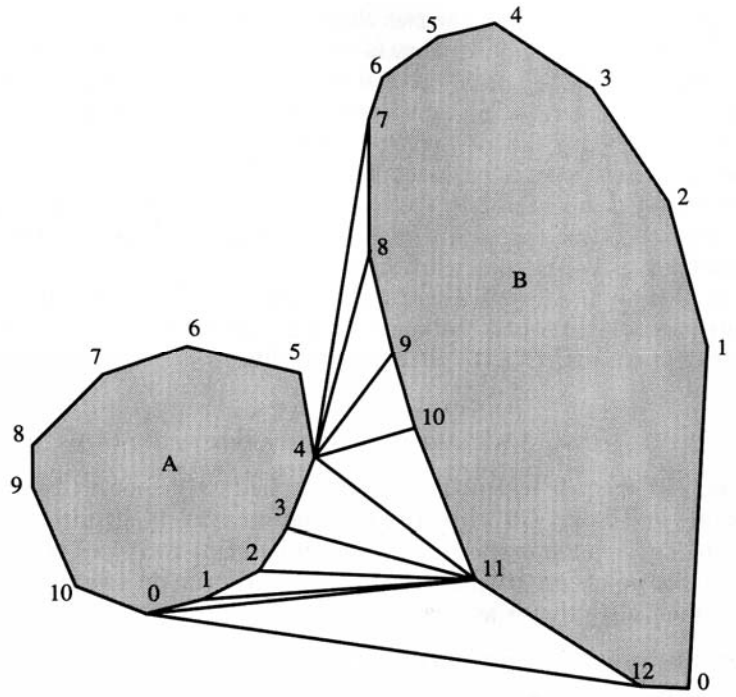


Fig. 5. Finding a common tangent of the two hull by progressing along successive hull points.

Possible parallelization strategies

i) For the gift wrapping algorithm: subdivide the total angle (180 degrees) into as many sub-angles as processors, create sub-hulls and merge and correct the sub-hulls into the final hull.

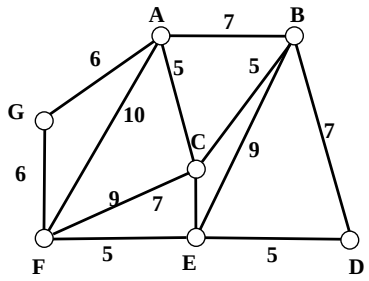
ii) For the quickhull algorithm, run on a two or four processor system by distributing the quadrilateral segments and their associated set of points to the individual processors. Then, the sub-hulls are merged. For more processors, consider extrema points at other orientations (+- 45^{o} , maximal distance by projection into corresponding oblique line through the center).

iii) For all algorithms: Subdivide your set of points into spatially distinct sets of subpoints. Each processor handles one subset of points and creates a corresponding subset hull. Subset hulls are then successively merged two by two, as explained in paragraph (d), and the final convex hull is finally obtained.

Assistant Petar Pjanic, R. D. Hersch (presenter)

Computing the minimal spanning tree

We consider a connected undirected graph G with vertices V and edges E : G=(V,E).
The minimal spanning tree is a subgraph T of G that comprises all vertices of G and which has no cycles.
We look for the miminal spanning tree, i.e. the tree whose total cost represented by the summation of the costs associated to the individual edges is minimal. The problem is typical for communication networks, transportation networks, energy pipelines, VLSI chip routing, etc..



Spanning tree construction with successive snapshots of the spanning tree edge set :
{ A}, {AC}, { AC,CB}, {AC,CB,AG},
{AC,CB,AG,GF}, {AC,CB,AG,GF, FE},
{AC,CB,AG,GF, FE, ED},

Fig. 1 (a) An undirected connected graph, (b) the corresponding spanning tree

Prim's algorithm for computing the minimal spanning tree is extremely simple. It has a high resemblance with Dijkstra's shortest path algorithm.

Here is an outline of the algorithm, given a graph G = (V,E, W), were W is the symmetric weighted adjacency matrix, whose elements give the cost of an edge between two vertices. Where no edge exists, the weight is ∞.

(a) We start with any root vertex *r* and put it in the empty set of vertices V_T which forms the spanning tree.
(b) We compute the distance between each vertex $v \in (V - V_T)$ and the vertices within V_T.
(c) Find the vertex $u \in (V - V_T)$ with the smallest distance to a vertex of V_T.
(d) Add that vertex and the connecting edge to the tree
(e) Continue at (b) until all vertices are within the set V_T .

```
procedure PRIM_MST(V, E, w, r)              // find the minimal spanning tree T
begin
   V_T : (r) : //initialize spanning tree vertices V_T with vertex r, the designated root
   d( r) := 0;                                      //compute d[:], the
   for all v ε(V- V_T) do                           //weight between
      if edge (r, v) exists, set d[v]: = ω(r, v);   // r and each
      else set d[v]: = ∞;                           // vertex outside V_T
   while V_T ≠ V do // while there are vertices outside T
   begin
      Find a vertex u ε(V- V_T) such that d[u] : = min {d[v] | v ε(V- V_T)} ;
      V_T : = V_T ∪ {u} :
      for all v ε (V- V_T) do                       //recompute d[:] to the new vertex u
          d[v]: = min { d[v], ω (u, v)} :           //which is now in T
   endwhile
```

//use d[:] to find *u*,
//vertex closest to T

source: Vivek Sarkar, Parallel graph algorithms, Rice University (slides on the Internet)

Regarding the parallelization, one may, by partitioning the adjacency matrix into the different compute nodes, compute in parallel the distance between each vertex v ε (V- V_T) and the vertices within V_T and within each node, find the minimal distance and then, on the master, the overall minimal distance.

Responsible: R. D. Hersch

Computing the shortest path in a graph (Dijkstra's algorithm)

We consider a connected undirected graph G with vertices V and edges E : G=(V,E).
We look for the shortest path from a source node to a destination node. The weight of a path p =(v_1, v_2, .. v_k) is the sum of the weights associated to each individual edge of the path. The problem is typical for network packet routing, route selection for transportation purposes, etc..

Here is an outline of the algorithm, given a graph G = (V,E, W), were W is the symmetric weighted adjacency matrix, whose elements give the cost of an edge between two vertices. Where no edge exists, the weight is ∞.

(a) We start with the source vertex *s* , label it as last path vertex *u* and put it in the empty set of vertices V_T which forms the shortest path.

(b) We compute the distance d(*s*,*v*) between each neighbouring vertex *v* of last path vertex *u* and the source vertex s :
d(*s*,*v*)= minimum { d(*s*,*v*), d(*s*,*u*), + d_{uv} }.

(c) Find the vertex *w* of the set of neighbouring vertices *v* of V_T with the smallest distance to the source vertex *s*.

(d) Mark that vertex *w* as last path vertex *u*, insert it and its connecting edge into the shortest path.

(e) Continue at (b) until all vertices are within the set V_T .

As illustration, consider figure 1 below, where node 1 is the source node. .

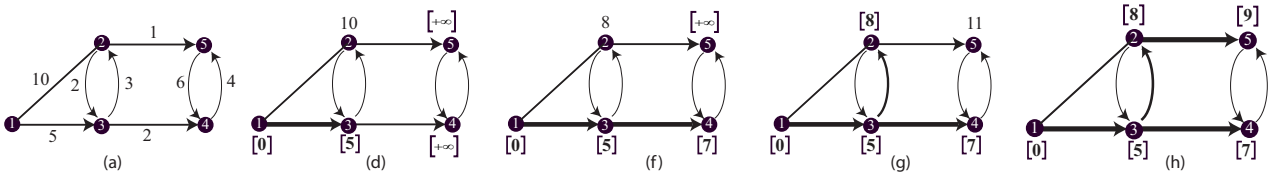


Fig. 1. The execution of Dijkstra's shortest path algorithm: Fig 1a gives the initial directed graph.

Regarding the *parallelization*, one may distribute vertical bands of the adjacency matrix to the processing nodes and allocate subgraphs formed by vertices comprising the source vertex as well as a subset of the graph vertices in each node. Then, at each iteration, the distance of neighbours from the last path vertex can be computed in parallel (b) and exchanged, as well as vertex *w* neighbour of vertices *v* of V_T with minimal distance to the source can be computed (c).

Assistants: R. Rossier (presenter), R.D. Hersch

Variant: Computing the shortest path between any source node and any other node

Apply in parallel the shortest path algorithm by designing each time a different source node. This yields the full set of shortest paths between any two nodes
Assistants: R. Rossier (presenter), R.D. Hersch

Image binarization (Otsu's method), for GPU implementation + OpenMp

From Wikipedia, the free encyclopedia

An example image thresholded using Otsu's algorithm

In computer vision and image processing, **Otsu's method** is used to automatically perform clustering-based image thresholding,[1] or, the reduction of a graylevel image to a binary image. The algorithm assumes that the image contains two classes of pixels following bi-modal histogram (foreground pixels and background pixels), it then calculates the optimum threshold separating the two classes so that their combined spread (intra-class variance) is minimal.[2] The extension of the original method to multi-level thresholding is referred to as the Multi Otsu method.[3] Otsu's method is named after Nobuyuki Otsu (大津展之 *Ōtsu Nobuyuki*?).

Method

In Otsu's method we exhaustively search for the threshold that minimizes the intra-class variance (the variance within the class), defined as a weighted sum of variances of the two classes:

$$\sigma_w^2(t) = \omega_1(t)\sigma_1^2(t) + \omega_2(t)\sigma_2^2(t)$$

Weights ω_i are the probabilities of the two classes separated by a threshold t and σ_i^2 variances of these classes.

Otsu shows that minimizing the intra-class variance is the same as maximizing inter-class variance:[2]

$$\sigma_b^2(t) = \sigma^2 - \sigma_w^2(t) = \omega_1(t)\omega_2(t)\left[\mu_1(t) - \mu_2(t)\right]^2$$

which is expressed in terms of class probabilities ω_i and class means μ_i.

The class probability $\omega_1(t)$ is computed from the histogram as t:

$$\omega_1(t) = \sum_0^t p(i)$$

while the class mean $\mu_1(t)$ is:

$$\mu_1(t) = \left[\sum_0^t p(i)\,x(i)\right] / \omega_1$$

where $x(i)$ is the value at the center of the ith histogram bin. Similarly, you can compute $\omega_2(t)$ and μ_2 on the right-hand side of the histogram for bins greater than t.

The class probabilities and class means can be computed iteratively. This idea yields an effective algorithm.

Algorithm

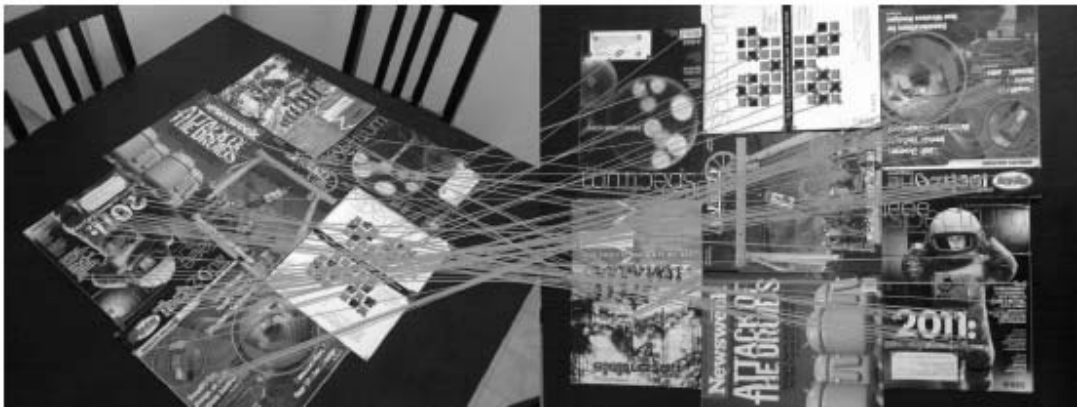
1. Compute histogram and probabilities of each intensity level
2. Set up initial $\omega_i(0)$ and $\mu_i(0)$
3. Step through all possible thresholds $t = 1 \ldots$ maximum intensity
    1. Update ω_i and μ_i
    2. Compute $\sigma_b^2(t)$
4. Desired threshold corresponds to the maximum $\sigma_b^2(t)$
5. You can compute two maxima (and two corresponding thresholds). $\sigma_{b1}^2(t)$ is the greater max and $\sigma_{b2}^2(t)$ is the greater or equal maximum
6. Desired threshold = $\dfrac{\text{threshold}_1 + \text{threshold}_2}{2}$

Consider a large input image taken with an illumination that varies from one side of the image to the other side. Segment it into 64x64 or 128x128 pixel areas and apply in parallel to each area Otsu's binarization algorithm.

Responsible: Sami Arpa (algorithms), R.D. Hersch (presenter)

Matching of feature points between two images (e.g. GPU-CUDA)

For many image processing and recognition tasks, it is important to match feature points located in one image to the feature points located in a second image.



(figure from paper : Rubllee et al., ORB an efficient alternative to SIFT and SURF, ICCV 2011)

The first step consists in identifying in the two images feature points, e.g. corners. The second step consists in characterizing these feature points by a signature vector. The third step consists in matching feature points in one image to feature points in the other image by comparing the proximities of their signature vectors.
The first, second and third steps may be parallelized on CUDA as independent tasks, each by one student.

A. Identification of corner points

See E. Rosten, R. Porter, and T. Drummond. Faster and better: A machine learning approach to corner detection. IEEE Trans. Pattern Analysis and Machine Intelligence, 32:105–119, 2010.

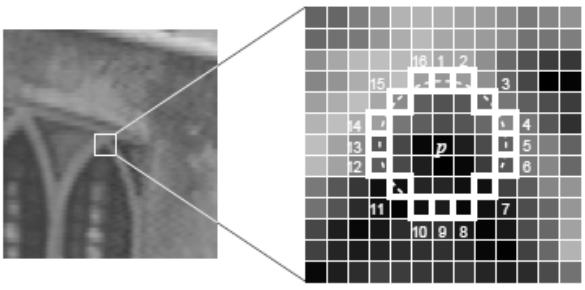


Fig. 1
12 POINT SEGMENT TEST CORNER DETECTION IN AN IMAGE PATCH. THE HIGHLIGHTED SQUARES ARE THE PIXELS USED IN THE CORNER DETECTION. THE PIXEL AT p IS THE CENTRE OF A CANDIDATE CORNER. THE ARC IS INDICATED BY THE DASHED LINE PASSES THROUGH 12 CONTIGUOUS PIXELS WHICH ARE BRIGHTER THAN p BY MORE THAN THE THRESHOLD.

III. HIGH-SPEED CORNER DETECTION

A. FAST: Features from Accelerated Segment Test

The segment test criterion operates by considering a circle of sixteen pixels around the corner candidate p. The original detector [95], [96] classifies p as a corner if there exists a set of n contiguous pixels in the circle which are all brighter than the intensity of the candidate pixel I_p plus a threshold t, or all darker than $I_p - t$, as illustrated in Figure 1. n was originally chosen to be twelve because it admits a high-speed test which can be used to exclude a very large number of non-corners. The high-speed test examines pixels 1 and 9. If both of these are within t if I_p, then p can not be a corner. If p can still be a corner, pixels 5 and 13 are examined. If p is a corner then at least three of these must all be brighter than $I_p + t$ or darker than $I_p - t$. If neither of these is the case, then p cannot be a corner. The full segment test criterion can then be applied to the remaining candidates by examining all pixels in the circle.

After smoothing the image by a low pass filter (e.g. Gaussian filter with σ=2 pixels or uniform 3x3 kernel filter) one checks the 4 horizontal and vertical neighbours of pixel *p*. If 3 of 4 are all lower or higher than I_p then one verifies that all intermediate pixels are also all lower or higher than I_p. If this is the case, pixel *p* is a corner pixel. Once a corner pixel *p* is detected, all pixels at and inside the corresponding circle around *p* are masked. This mask prevents the detection of further corner pixels within the area spanned by that circle.
As a result of corner detection, a map can be created with all corner pixels *p* activated.

B. Characterization of corner points by a signature vector

M. Calonder, V. Lepetit, M. Özuysal, T. Trzcinski, C. Strecha, and P. Fua, BRIEF: Computing a local binary descriptor very fast, IEEE PAMI, vol. 34, num. 7, p. 1281-1298

Many possible characterizations exist. The one proposed by Calonder et al. (from EPFL) is the following.

More specifically, we define test τ on patch $\mathbf{p}$ of size $S \times S$ as

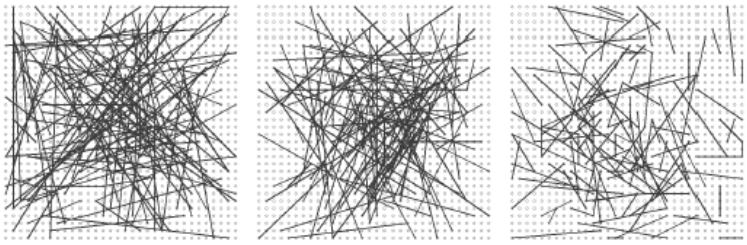
$$\tau(\mathbf{p};\mathbf{x},\mathbf{y}) := \begin{cases} 1 & \text{if } I(\mathbf{p},\mathbf{x}) < I(\mathbf{p},\mathbf{y}) \\ 0 & \text{otherwise} \end{cases} , \quad (1)$$

where $I(\mathbf{p},\mathbf{x})$ is the pixel intensity in a smoothed version of $\mathbf{p}$ at $\mathbf{x} = (u,v)^\top$. Choosing a set of n_d $(\mathbf{x},\mathbf{y})$-location pairs uniquely defines a set of binary tests. We take our BRIEF descriptor to be the n_d-dimensional bit string that corresponds to the decimal counterpart of

$$\sum_{1 \le i \le n_d} 2^{i-1}\, \tau(\mathbf{p};\mathbf{x}_i,\mathbf{y}_i) \ . \quad (2)$$

In this paper we consider $n_d = 128$, 256, and 512 and will show in the Results section that these yield good compromises between speed, space, and accuracy.

Sampling within the considered SxS window is performed as follows:



G I G II G III

I) $(\mathbf{X},\mathbf{Y}) \sim$ i.i.d. Uniform$(-\frac{S}{2},\frac{S}{2})$: The $(\mathbf{x}_i,\mathbf{y}_i)$ locations are evenly distributed over the patch and tests can lie close to the patch border.

II) $(\mathbf{X},\mathbf{Y}) \sim$ i.i.d. Gaussian$(0,\frac{1}{25}S^2)$: The tests are sampled from an isotropic Gaussian distribution. This is motivated by the fact that pixels at the patch center tend to be more stable under perspective distortion than those near the edges. Experimentally we found $\frac{S}{2} = \frac{5}{2}\sigma \Leftrightarrow \sigma^2 = \frac{1}{25}S^2$ to give best results in terms of recognition rate.

III) $\mathbf{X} \sim$ i.i.d. Gaussian$(0,\frac{1}{25}S^2)$, $\mathbf{Y} \sim$ i.i.d. Gaussian$(\mathbf{x}_i,\frac{1}{100}S^2)$: The sampling involves two steps. The first location $\mathbf{x}_i$ is sampled from a Gaussian centered around the origin, like the previous, while the second location is sampled from another Gaussian centered on $\mathbf{x}_i$. This forces the tests to be more local. Test locations outside the patch are clamped to the edge of the patch. Again, experimentally we found $\frac{S}{4} = \frac{5}{2}\sigma \Leftrightarrow \sigma^2 = \frac{1}{100}S^2$ for the second Gaussian performing best.

C. Matching of feature points : they are matched by considering the Hamming distance between their signature vectors, i.e. the relative number of positions where the zeros and ones of the vectors don't match.

Responsible: Sami Arpa (algorithms), R.D. Hersch (presenter)