
aspire invent achieve
imec

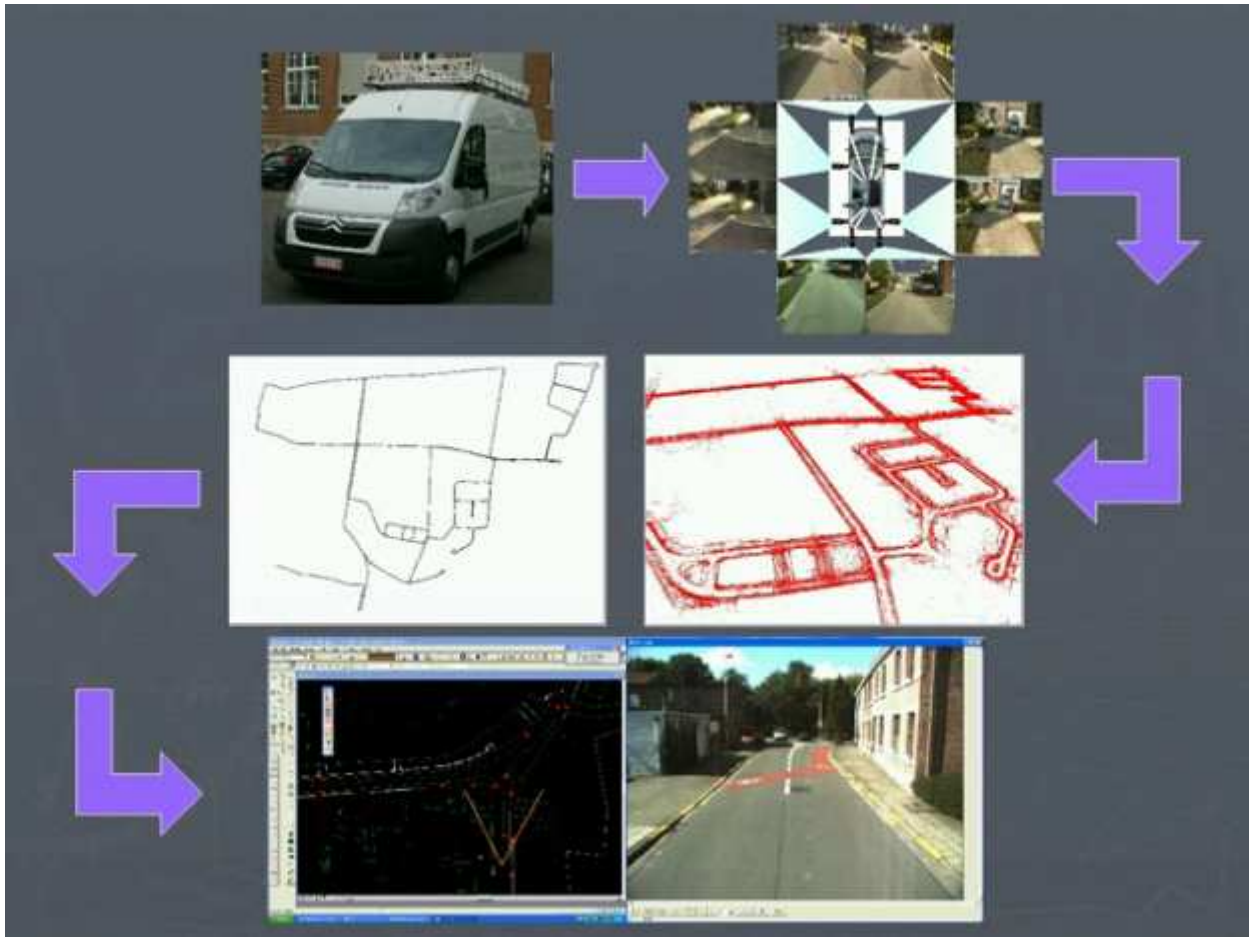
URBAN

WP1 Digital Surveying



URBAN

WP 1 : Digital Surveying



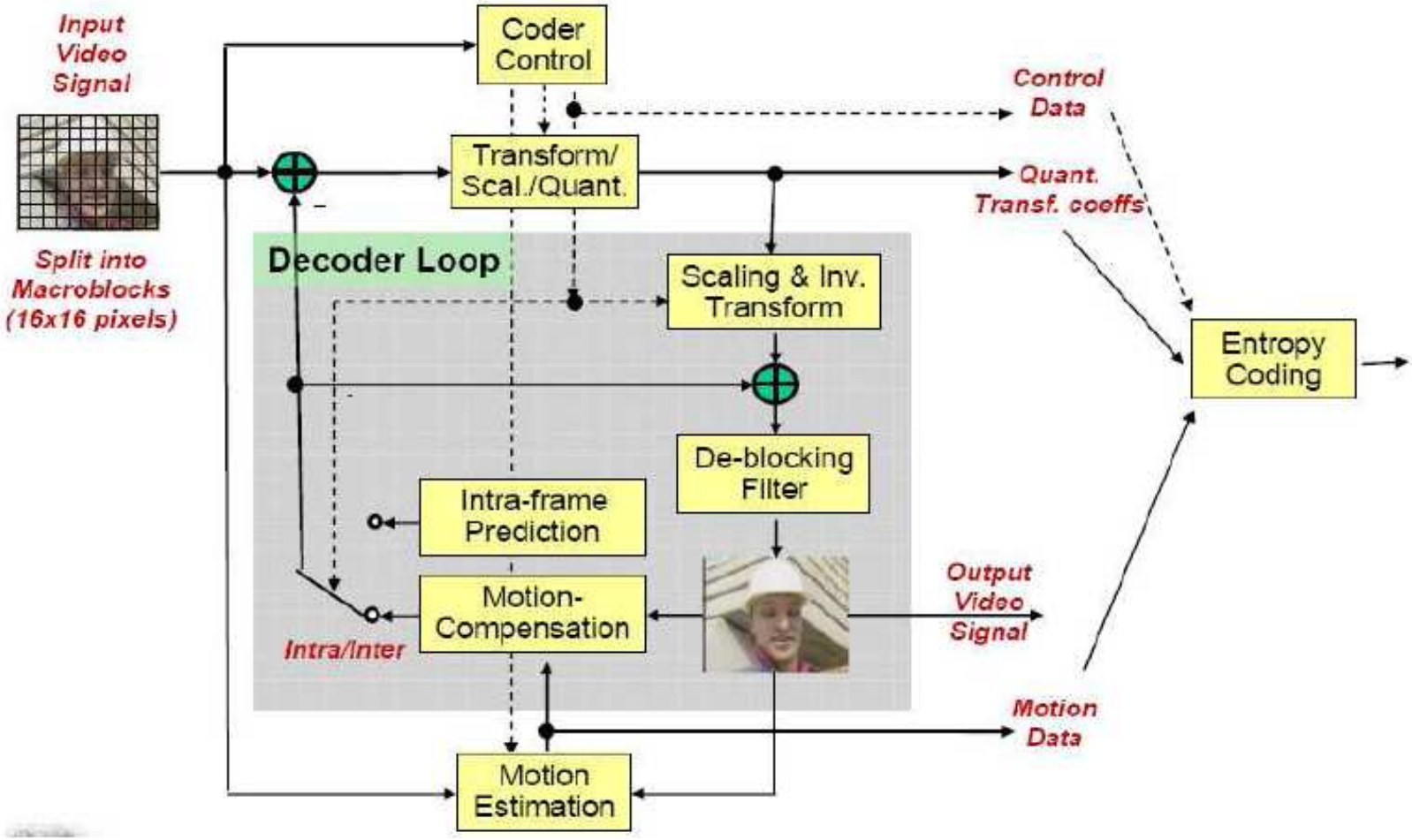
URBAN

WP1 Digital Surveying

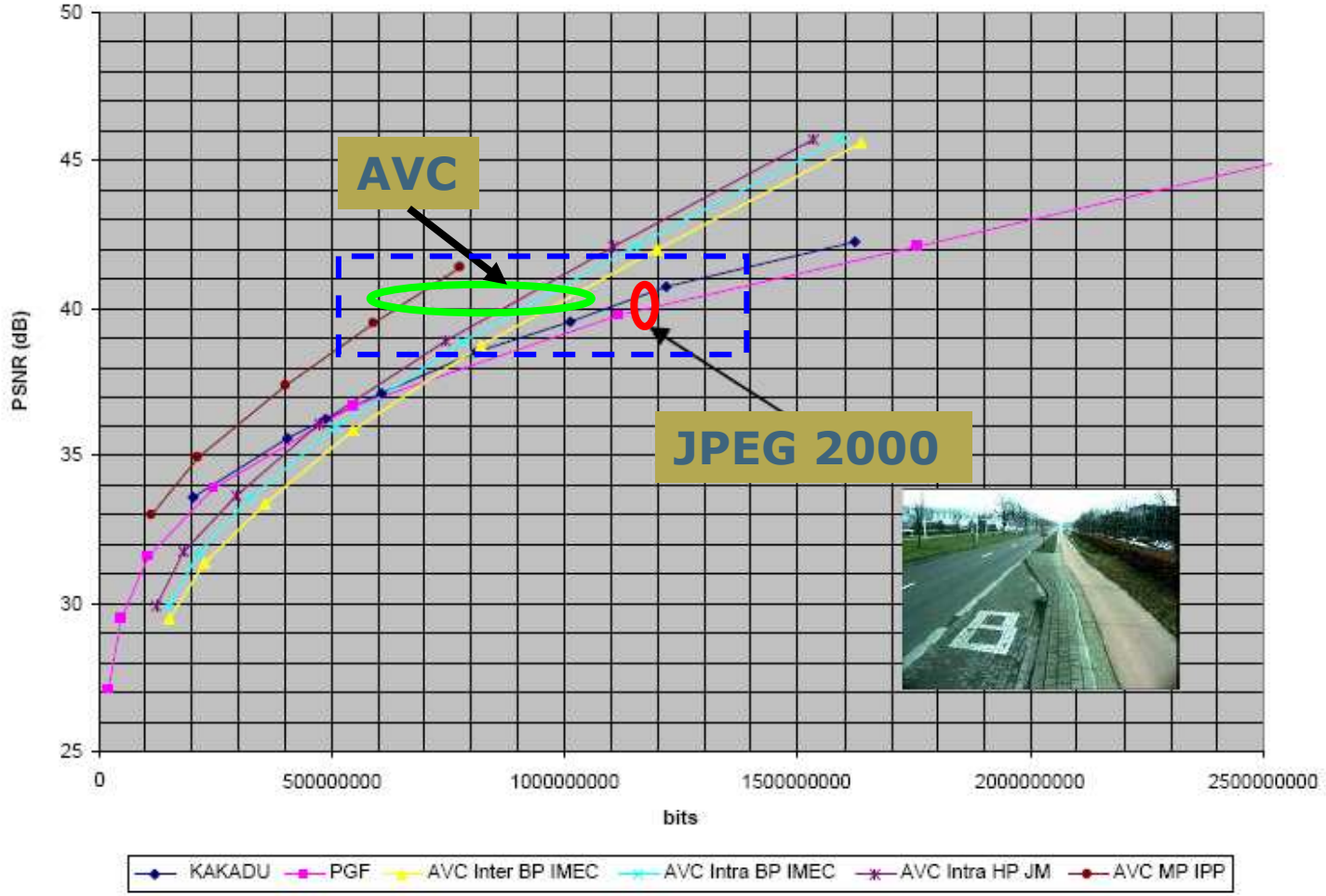
Task 1 : Processing on the van



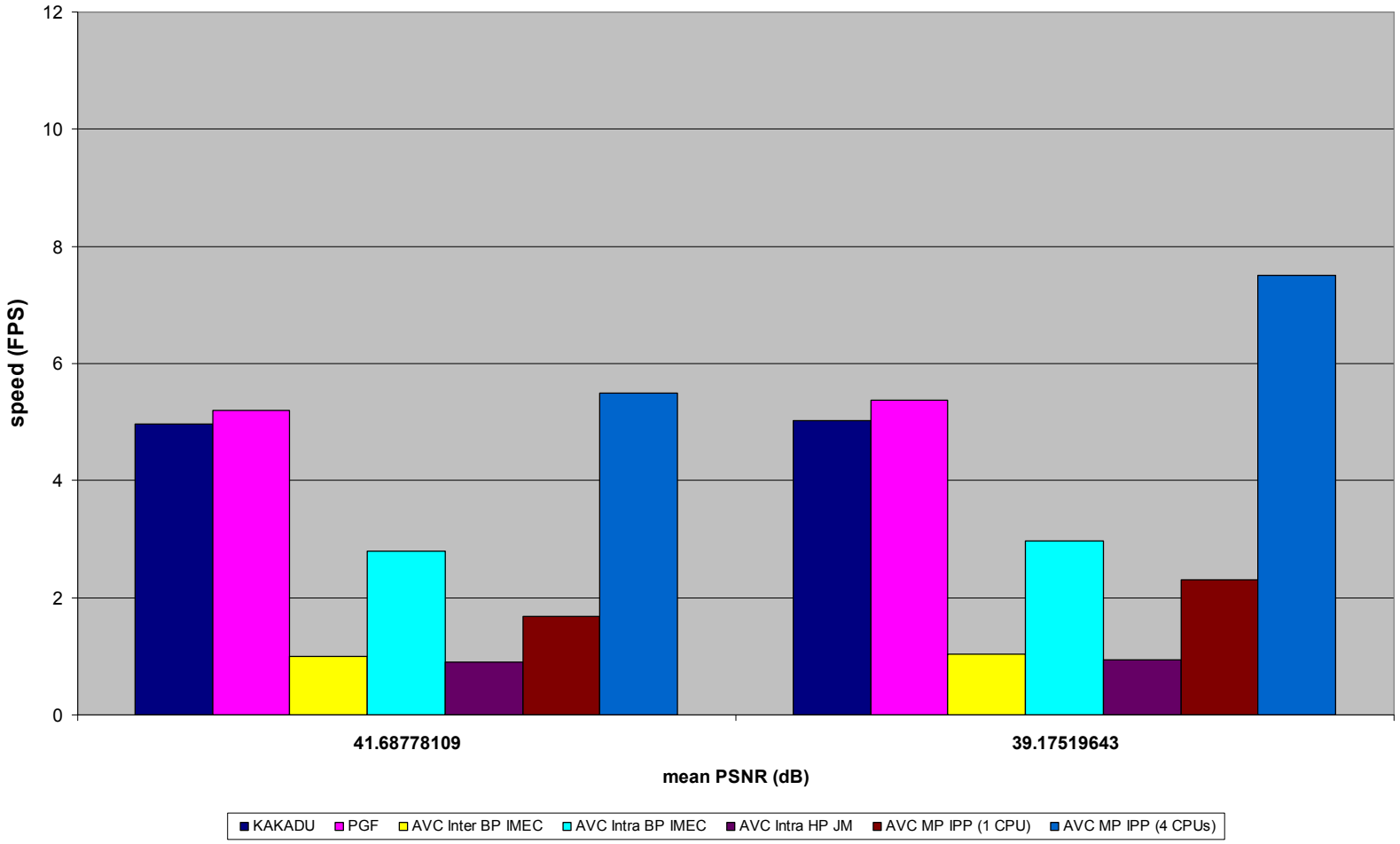
H.264 encoding



H.264 encoding:
Motivation in URBAN – Rate Distortion



H.264 encoding:
Motivation in URBAN – Encoding Speed



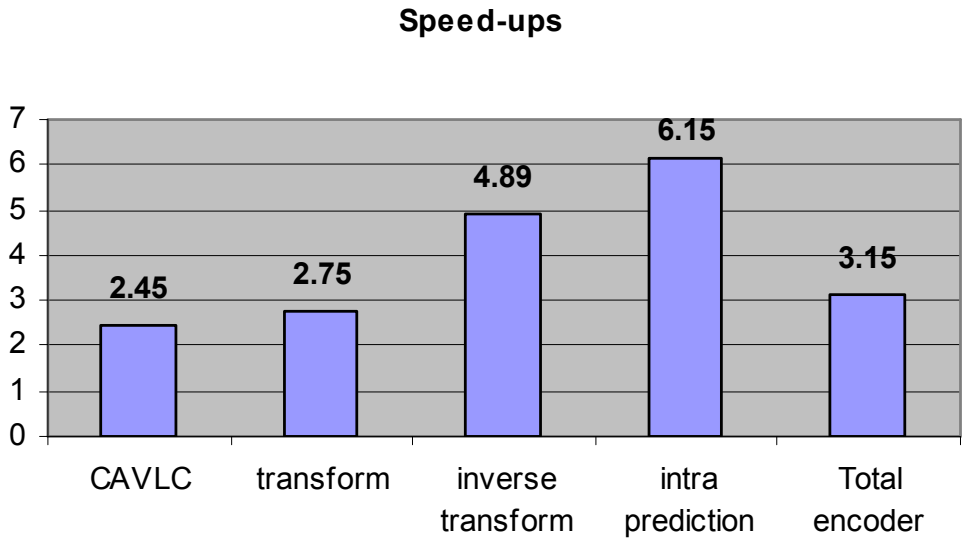
H.264 encoding:
Motivation in URBAN – Quality Measurement

- Quality assessment of the final 3d reconstruction based on the decoded video
  - Test on 20.000 images JPEG2000 @ 295KB and H.264 @ ~350 KB
  - Required accuracy 15 cm in 3D

	JPEG 2000	H.264
features tracks	101000	92000
reprojection error (pixels)	1.126	1.141
error in 2D (cm)	5.97	6.96
error in 3D (cm)	11.3	12

H.264 encoding optimization steps

- Initial version with algorithmic tuning
  - Intra 16x16 and Intra 4x4 modes only
  - *same algorithm as the one used in the quality test*
- Processor specific optimizations
  - SSE optimizations, Loop unrolling in CAVLC, Transform, Inv. Transform and Prediction
  - Use of openMP for a dual core CPU: frames partitioned into two independent slices
  - measured on Intel Core 2 Duo CPU, 2.20GHz, 2GB RAM



Target: 12fps

Final version: 13-15fps

Conclusion for Task 1

- Deliverables D113 and D114:
  - H.264 Intra-only Baseline Profile
  - Encodes URBAN images at ~14 fps > 12 fps (real-time)
  - Acceptable quality and bitrate
    - lower than JPEG 2000 but still acceptable
  - GPU acceleration is not possible
    - Too many dependencies
    - quality degradation using a GPU with CUDA for intra prediction
  - Enables fast image decoding @ 20-25 fps
  - Executables or libraries available on request

URBAN

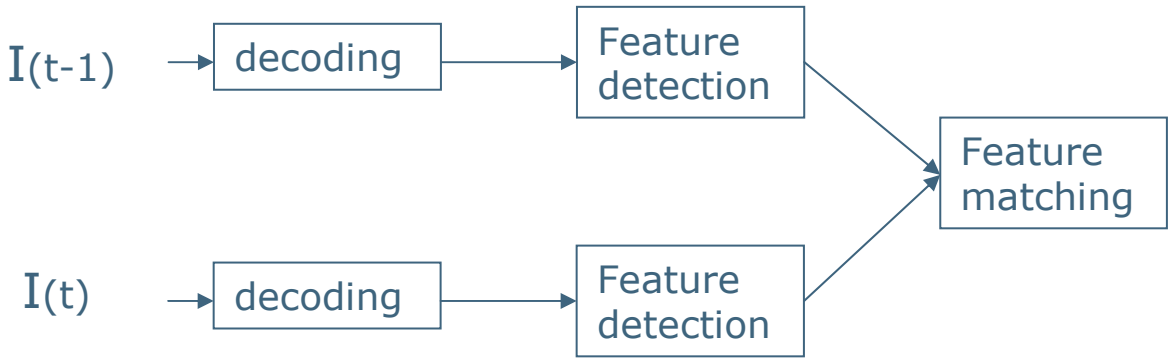
WP1 Digital Surveying

Task 2 : Processing on the servers



Deliverable D122 : optimized implementation of the selected components on a CPU-GPU hybrid system

- Selected components
  - Debayering
    - (actually takes place in the van, see D111, D112, D113)
  - 3D reconstruction components:
    - H.264 decoding
    - Feature (corner) detector
    - Feature matching with NCC (Normalized Cross Correlation)





Hybrid CPU-GPU System

- Hardware used
  - CPU
    - Intel Core i7
    - 5.06 GHz
    - 8 MB Intel smart cache
    - 4GB RAM
  - GPU
    - Nvidia's GeForce GTX 280
    - 1.3 GHz clock speed
    - 240 CUDA cores
    - 65535 threads
    - 1 GB global memory
    - 16 KB shared memory per core
    - **Memory bandwidth 147 GB/sec**
    - **CPU-GPU bandwidth 1.4 GB/sec**
    - Compute capability 1.3





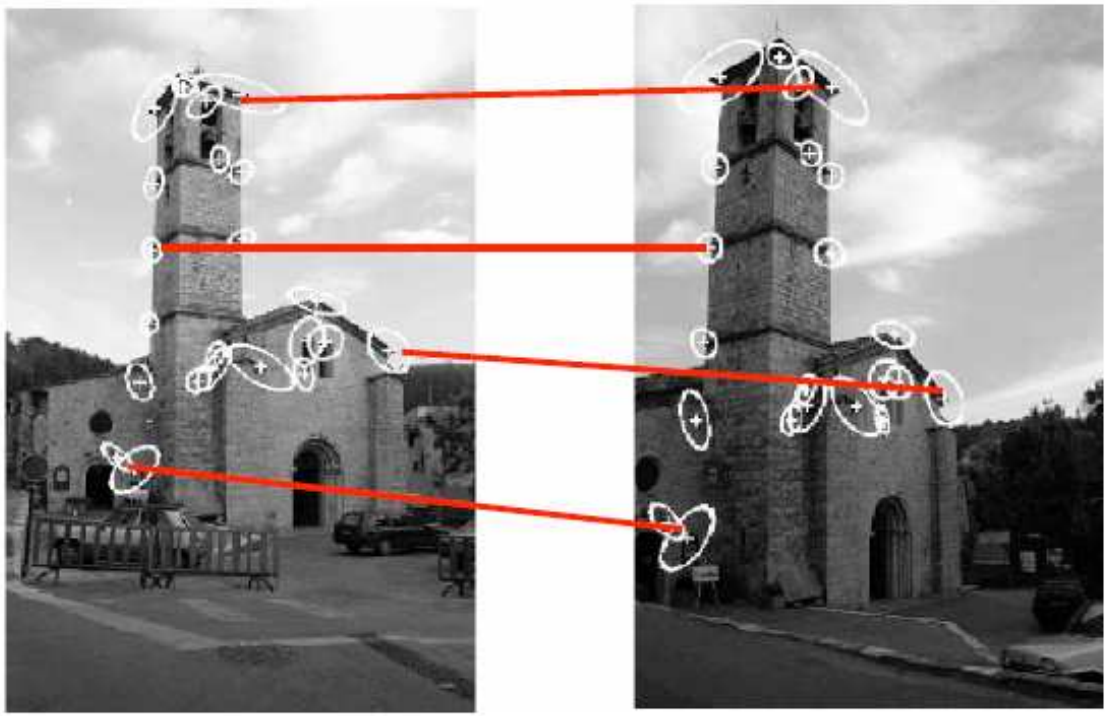
1. H.264 decoding

- Hardware acceleration (video processor)
  - Nvidia GeForce GTX 280 supports Pure Video® HD
  - Intra-coded bitstream decoding
    - 1080p @ 25-30fps
    - 1632x1248 @ 20 fps
  - Using OpenGL-CUDA interoperability



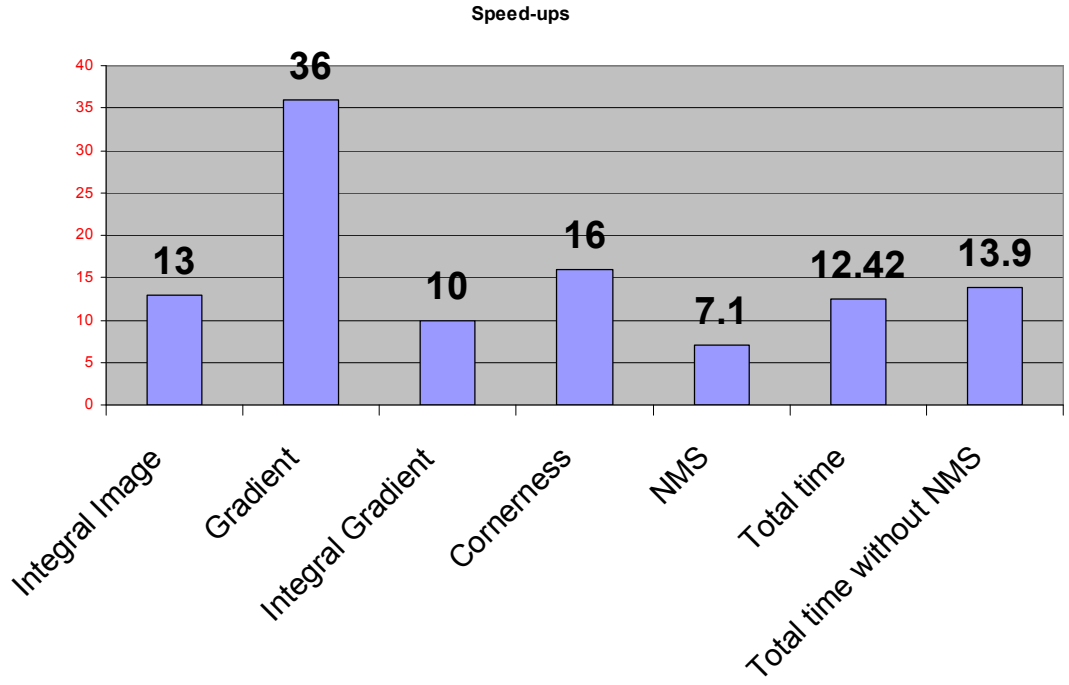
2. Feature (Corner) point detector

- Choice of the algorithm
  - Corner detectors
    - state-of-the-art is based on mainly two papers: Harris and KLT
    - Motivation: matching problem
  - Use of a low complexity corner detector for higher performance



2. Feature (Corner) point detector

- Low Complexity Harris detector
  - Use integral image to calculate sum of pixels/gradients
  - Approximate the Gaussian filter by box filter
- GPU vs. CPU: 10x speed-up



P. Mainali, Q. Yang, G. Lafruit, R. Lauwereins and L. Van Gool, "LOCOCO: LOW COMPLEXITY CORNER DETECTOR", submitted to ICASSP 2010.

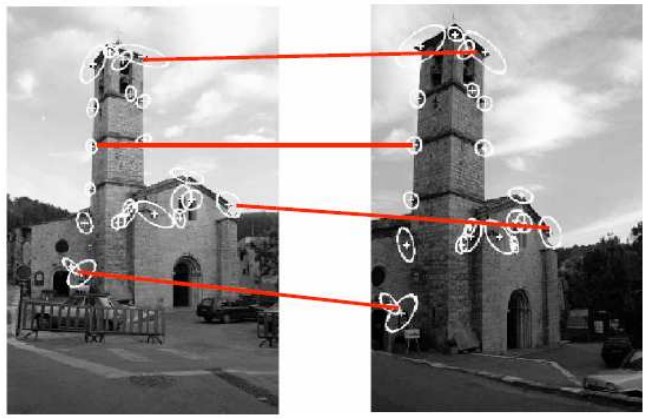
Corner detector: Sample Image



3. Feature matching

- Test possible feature pairs
  - Between stereo views at time *t* — 4 image pairs
  - Between each view at time *t* and *t-1* — 8 image pairs
  - Use a metric to compare their 9x9 neighborhoods
- Normalized Cross Correlation (NCC)
  - Spatial method

$$\frac{\sum_{x,y}[f(x,y)-\bar{f}_{u,v}][t(x-u,y-v)-\bar{t}]}{\left\{\sum_{x,y}[f(x,y)-\bar{f}_{u,v}]^2\sum_{x,y}[t(x-u,y-v)-\bar{t}]^2\right\}^{0.5}}$$



  - Spectral method (approximated)

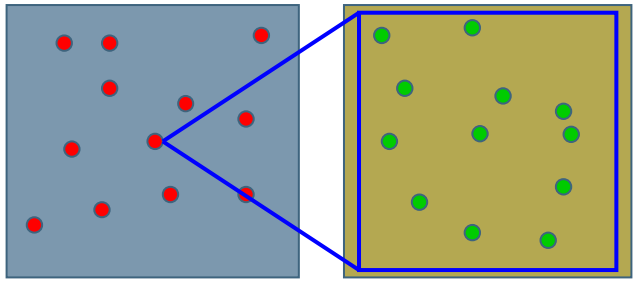
$$\mathcal{F}^{-1}\{\mathcal{F}(f')\mathcal{F}^{*}(t')\}$$

3. Feature matching

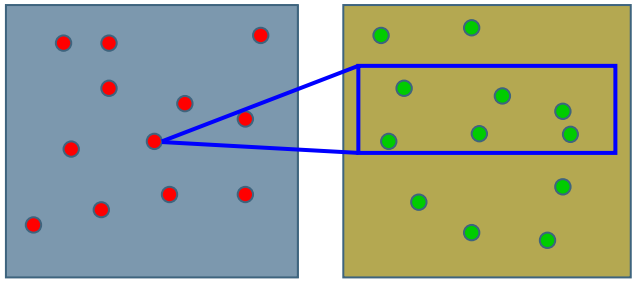
- CUDA implementation
  - Spatial method:
    - Computationally expensive on CPU
    - Can be efficiently run in parallel on GPU as a convolution
    - NCC can be estimated for many window pairs in parallel
  - Spectral method:
    - Approximation but lower complexity on CPU
    - cuFFT library proposes optimized direct and inverse FFT
    - However it is optimized for large window sizes (here we have 9x9)
    - Several windows FFT transformations cannot be launched in parallel on the GPU

Feature matching: CUDA implementation

- Feature pairs selection



Global



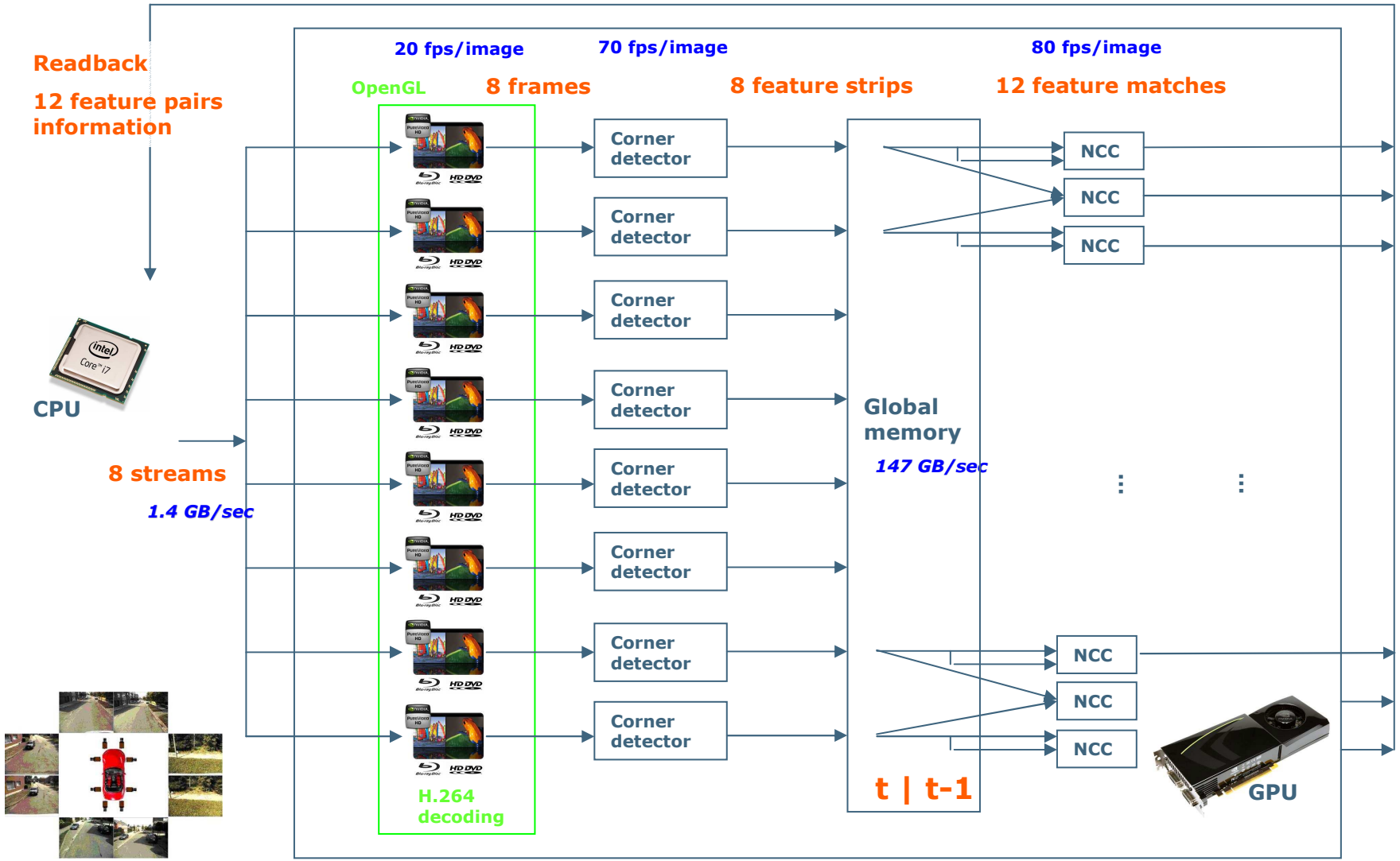
Region of interest

- CUDA implementation
  - Sorting NCC scores done in shared memory

Table 3. Timing measures for the feature matching kernel

Feature matching (NCC)	CPU (ms)	GPU (ms)	Speed-up
1000 x 1000 features	398	12.5	31.84
1000 x 16 features	6.05	0.11	54.5
16x16 features	0.11	0.010	10

Summary: Pipeline



Conclusion for Task 2

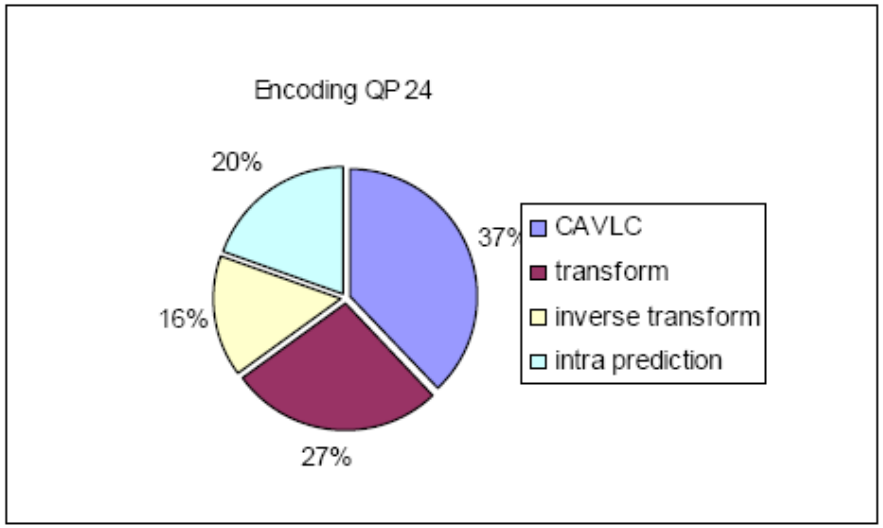
- Deliverables D122 and D123
  - CPU-GPU programming allows for significant acceleration but algorithmic tuning is mandatory
  - Bottleneck: CPU-GPU communication
    - The amount of data transferred between the CPU and GPU depends on the application
      - CPU -> GPU : Optional information on regions of interest for features matches
      - GPU -> CPU : Optional readback of decoded images to the CPU

	CPU (fps)	GPU (fps)
H.264 decoding	5.5	20
Feature detection	5.6	70
Feature matching	2.5	80

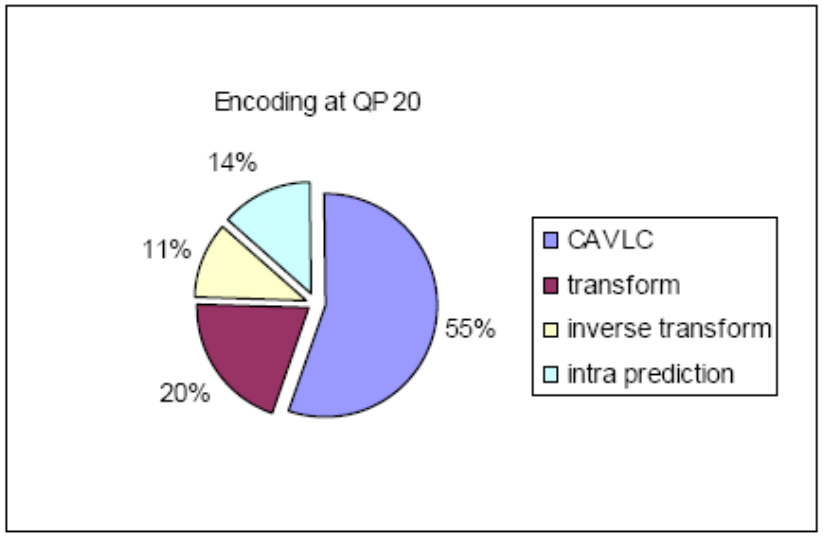

aspire invent achieve
imec

H.264 encoding optimization steps

- Initial version with algorithmic tuning
  - Intra 16x16 and Intra 4x4 modes only
  - *same algorithm as the one used in the quality test*
  - measured on Intel Core 2 Duo CPU, 2.20GHz, 2GB RAM



4.84 fps

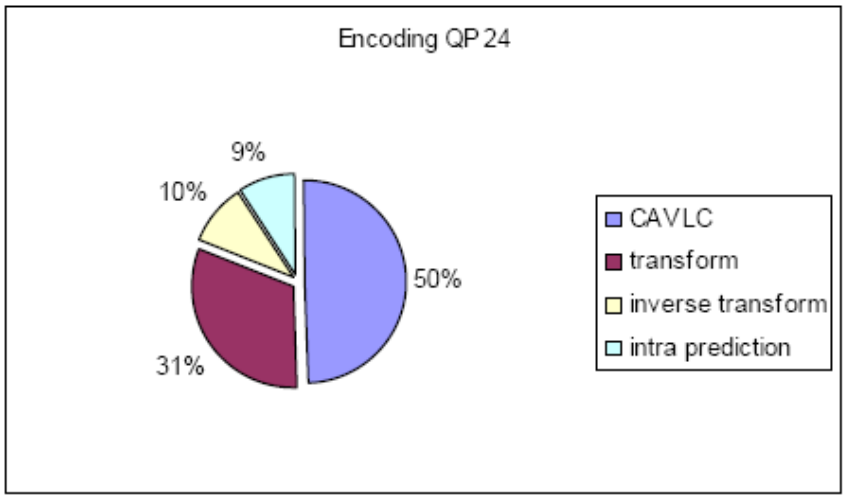


4.29 fps

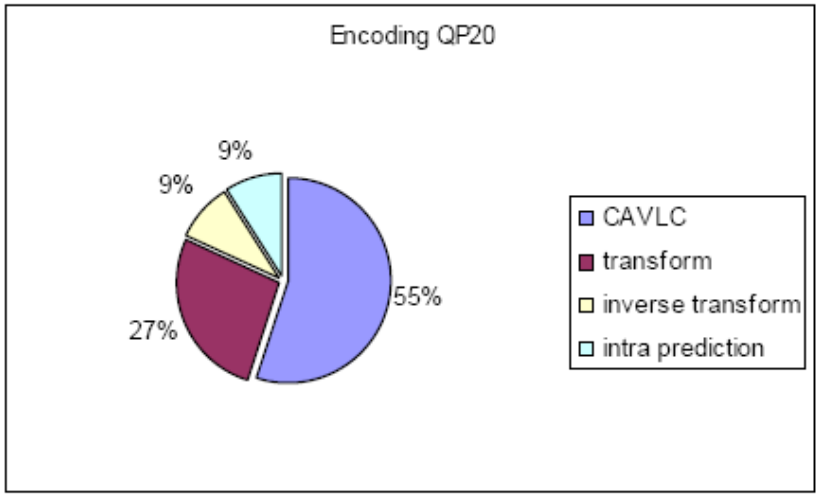
Target: 12 fps

H.264 encoding optimization steps

- DTSE and Vectorization and Multi-threading on several cores
  - SSE optimizations, Loop unrolling in CAVLC, Transform, Inv. Transform and Prediction
  - Use of openMP for a dual core CPU: frames partitioned into two independent slices
  - measured on Intel Core 2 Duo CPU, 2.20GHz, 2GB RAM



15.3 fps



13.7 fps

Target: 12 fps

2. Feature (Corner) point detector

P. Mainali, Q. Yang, G. Lafruit, R. Lauwereins and L. Van Gool, "LOCOCO: LOW COMPLEXITY CORNER DETECTOR", submitted to ICASSP 2010.

- **Harris detector**

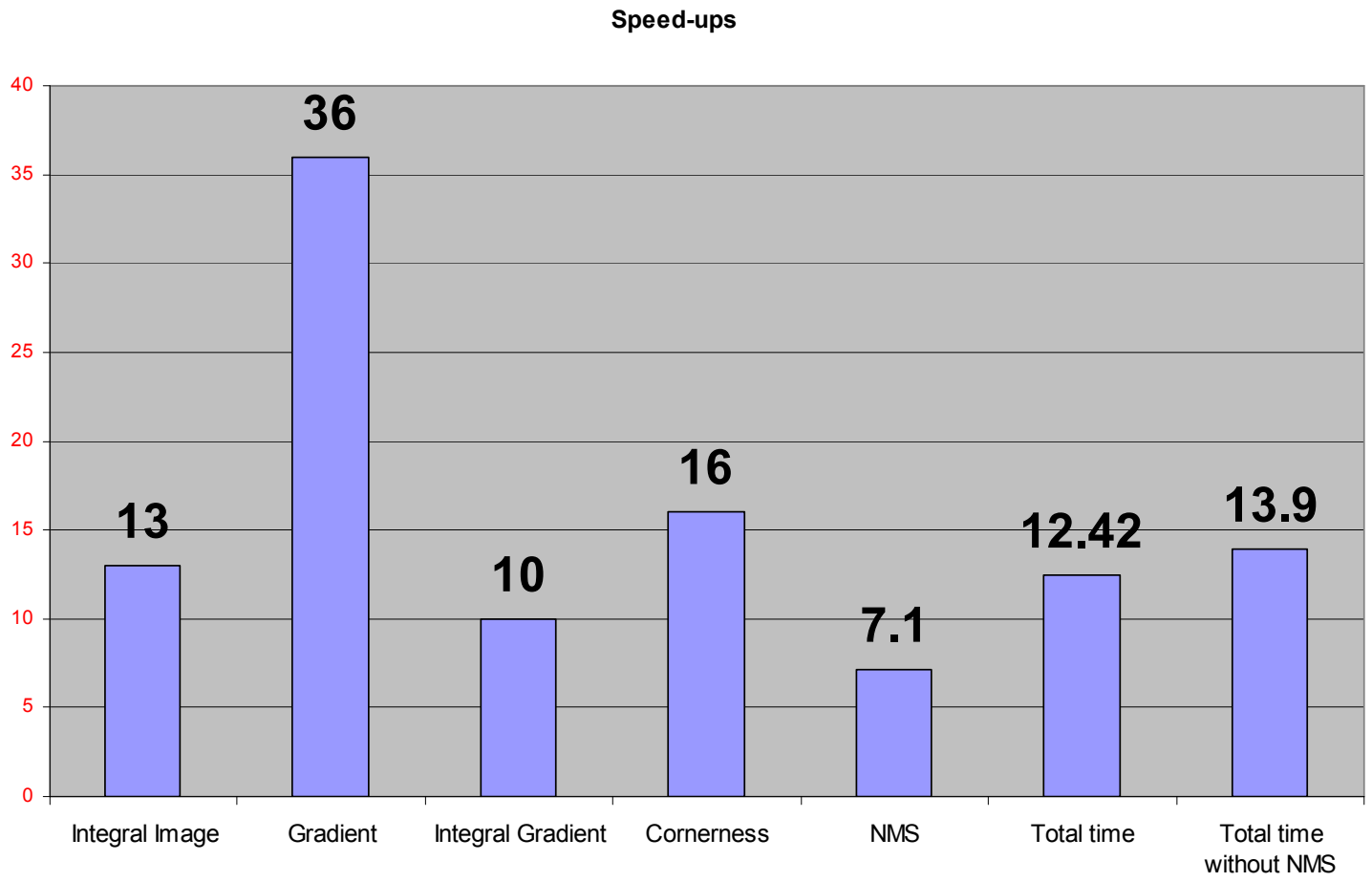
1. Compute x and y derivatives of image filtered by a gaussian Gx, Gy
2. Compute product of derivatives Gxx, Gxy, Gyy
3. Compute weighted averages of these products Sxx, Sxy, Syy
4. Compute the matrix
   H =[Sxx, Sxy; Sxy, Syy]
   and estimate cornerness
   $R = \det(H) - k\,(\mathrm{trace}(H))^2$
5. Non-maximum suppression

- **Low Complexity Harris det.**

1. *Approximate the gaussian derivative filter by the box filter of the integral image and compute G'x, G'y*
2. Compute product of derivatives Gxx, Gxy, Gyy
3. *Compute the sums of these products S'xx, S'xy, S'yy*
4. Compute the matrix
   H =[S'xx, S'xy; S'xy, S'yy]
   and estimate cornerness
   R = det(H)-k (trace(H))2
5. Non-maximum suppression

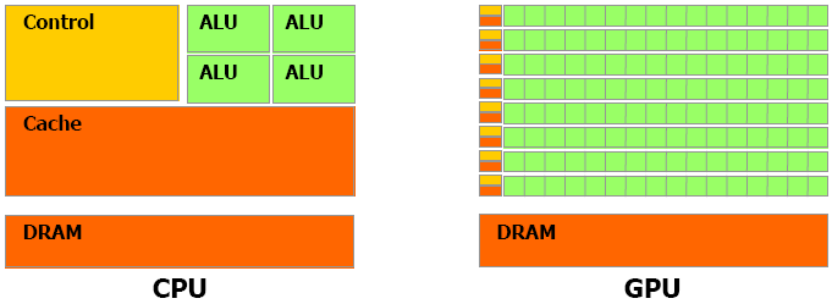
Corner detector: CUDA implementation

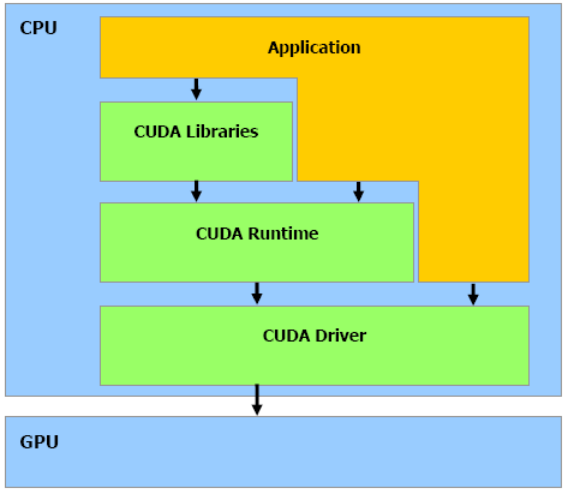
- Speed-up: GPU vs. CPU



Hybrid CPU-GPU programming

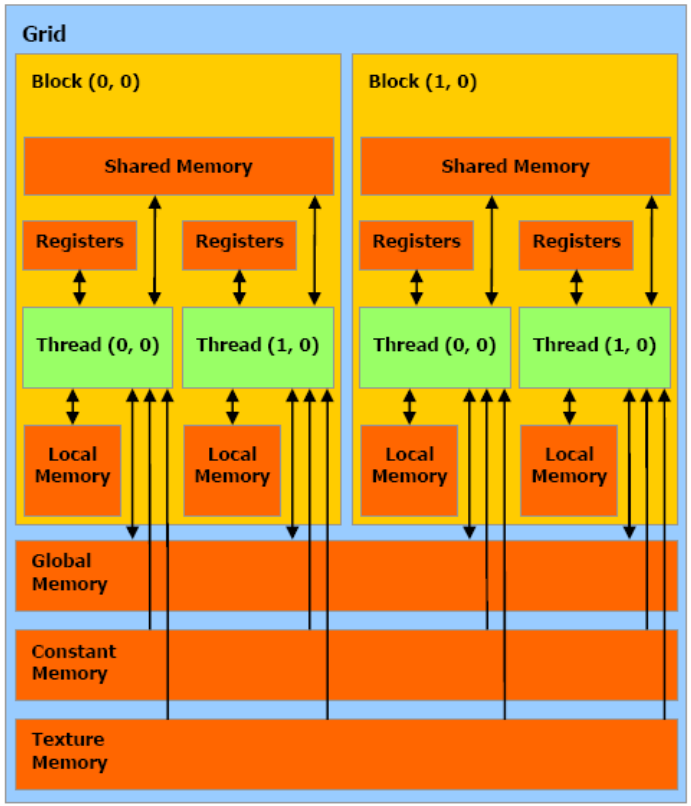
- Comparison of a CPU and GPU implementation
  - CPU (Single CPU with no hyperthreading) : optimization through data locality exploration (efficient data cache)
  - GPU : optimization by parallelization exploration (SIMD)
- CUDA (NVIDIA)
  - Compute Unified Device Architecture
  - "Extended C"





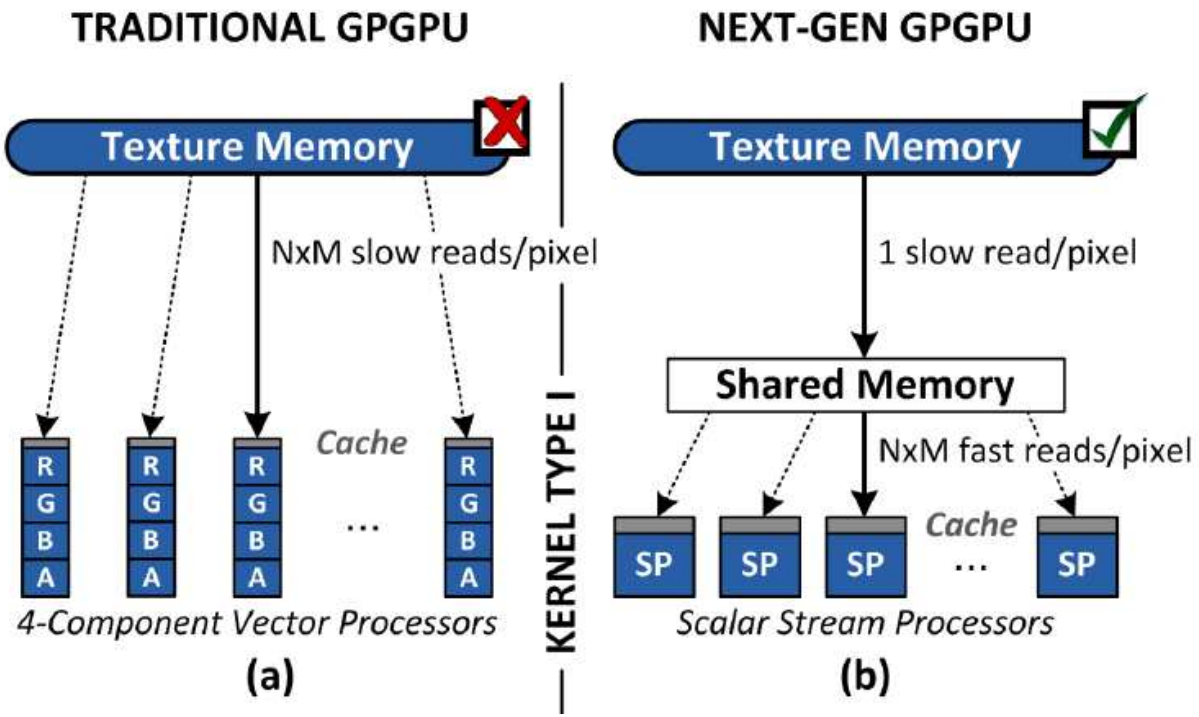
Hybrid CPU-GPU programming

- Memory Model
  - Grid
    - Global Memory
    - Constant Memory (read only)
    - Texture Memory (read only)
  - Block
    - ***Shared Memory***
    - Registers
    - Local Memory
  - SIMD parallelism
    - Hundreds of cores
    - 512 threads per core



CUDA vs Traditional GPGPU

- Shared memory allows for user controlled data cache management
  - Example: **N** x **M** Convolution kernels



Corner detector: CUDA implementation

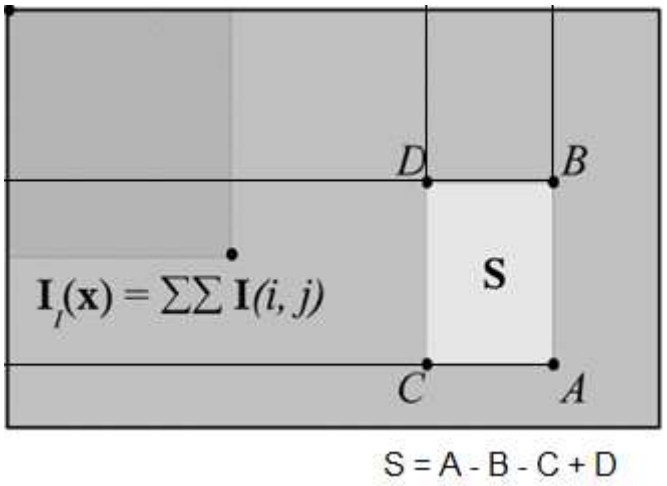
1. Integral image

   - Mostly sequential algorithm but …
     1. Prefixed-sum parallel algorithm to compute the sum of rows
     2. Transpose the result using shared memory and block pre-fetches
     3. Re-operate the prefix-sum on the rows

   - The transpose step is needed in order to optimize the memory reads

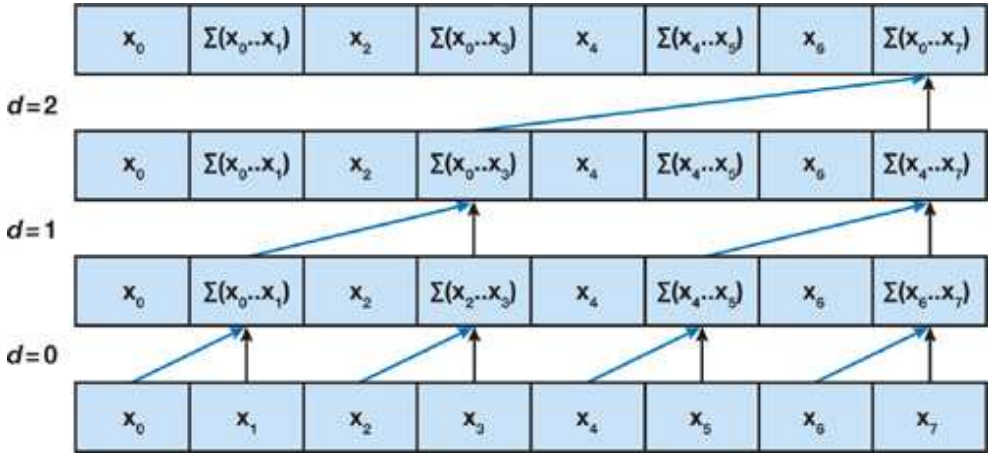
   - CPU: 18.3 ms GPU: 1.41 ms for 1632x1248 images

   - Speed-up = 13

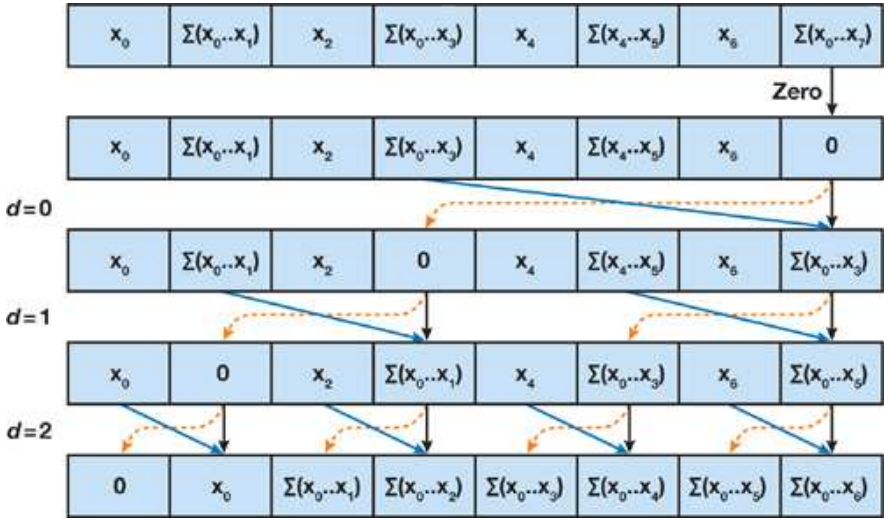


Corner detector: CUDA implementation

- Pre-fixed sum parallel implementation
  - Scan: ***Mark Harris, Shubhabrata Sengupta, and John D. Owens. "Parallel Prefix Sum (Scan) with CUDA". In Hubert Nguyen, editor, GPU Gems 3, chapter 39, pages 851–876. Addison Wesley, August 2007***



Up-sweep (reduction)



Down-sweep

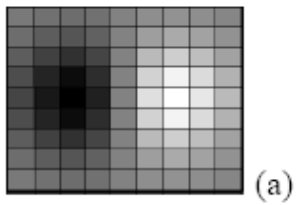
Corner detector: CUDA implementation

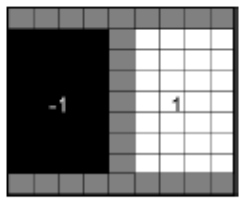
2. Approximate gaussian derivative filter by the box filter of the integral and compute Gx, Gy

- Boxfilters can be computed easily with CUDA. Options:
  1. Compute from global memory directly with memory fetches at x+-4, y+-4 positions
  2. Pre-store the input from the global memory into the shared memory for optimized reads

– CPU: 45.4 ms GPU: 1.26 ms (for two gradients)

– Speed-up = 36


(a)



(b)

Corner detector: CUDA implementation

3. Compute the products Gxx, Gxy, Gyy
4. Compute their sums Sxx, Sxy, Syy

- Combination of the two first kernels. Options:
  1. Pre-compute Gxx, Gyy, Gxy, store them in global memory and then compute ***integral images*** of the results
  2. Fuse both kernels by avoiding a pre-store of the three gradient products in global memory
     - But bandwidth is high…
     - Extra computations lower the optimality of the scan algorithm implementation by using more resources per thread
     - Results are similar
- CPU: 75.1 ms GPU: 7.5 ms (three kernel launches)
- Speed-up = 10

Corner detector: CUDA implementation

5. Evaluate the cornerness R from matrix H

- Pixel-wise evaluation of a simple expression from Sxx, Sxy, Syy
- Optimized reads by coalescing the pointers

- CPU: 17.6 ms GPU: 1.1 ms
- Speed-up = 16

6. Non-maximum suppression

- Sequential algorithm but a Quick Sort can be launched in parallel using the Radix Sorting
  - ***Nadathur Satish, Mark Harris, and Michael Garland. "Designing Efficient Sorting Algorithms for Manycore GPUs". Proc. 23rd IEEE International Parallel & Distributed Processing Symposium, May 2009.***

– CPU: 22.7 ms GPU: 12.8 ms

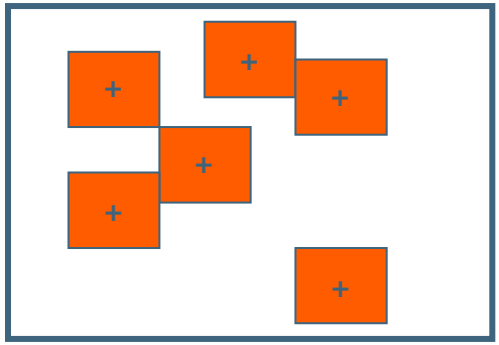
– Speed-up = 1.8

Feature matching: CUDA implementation

- Spatial method

$$\frac{\sum_{x,y}[f(x,y)-\bar{f}_{u,v}][t(x-u,y-v)-\bar{t}]}{\left\{\sum_{x,y}[f(x,y)-\bar{f}_{u,v}]^2\sum_{x,y}[t(x-u,y-v)-\bar{t}]^2\right\}^{0.5}}$$

  - Computing the mean of the windows $\bar{f}_{u,v}$ and $\bar{t}$ can be done by
    - Pre-fixed sum
    - Integral images value fetches
  - Memory fetches are not optimized for CUDA
    - Feature windows are generally not well aligned
    - Storing the windows in shared memory does not bring enough data re-use

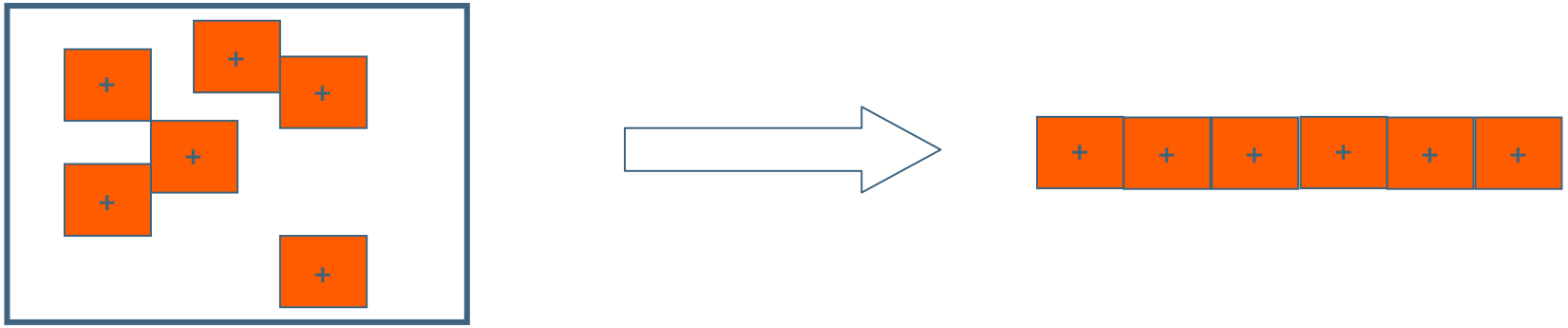


Feature matching: CUDA implementation

- Spatial method

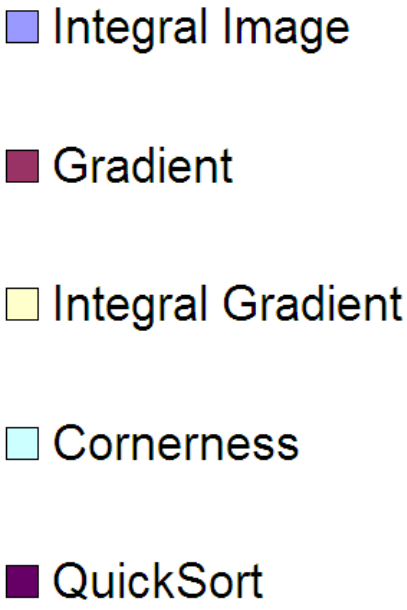
$$\frac{\sum_{x,y}[f(x,y)-\bar{f}_{u,v}][t(x-u,y-v)-\bar{t}]}{\left\{\sum_{x,y}[f(x,y)-\bar{f}_{u,v}]^2\sum_{x,y}[t(x-u,y-v)-\bar{t}]^2\right\}^{0.5}}$$

  - Pre-center (subtract the mean) and align the feature neighborhoods in the feature detection kernel (integral image is already used)
  - These feature neighborhood strips are stored in global memory for each image
  - Minimal extra computation for the feature detection kernel
  - GTX 280 global memory = 1GB enables to avoid many CPU-GPU communications

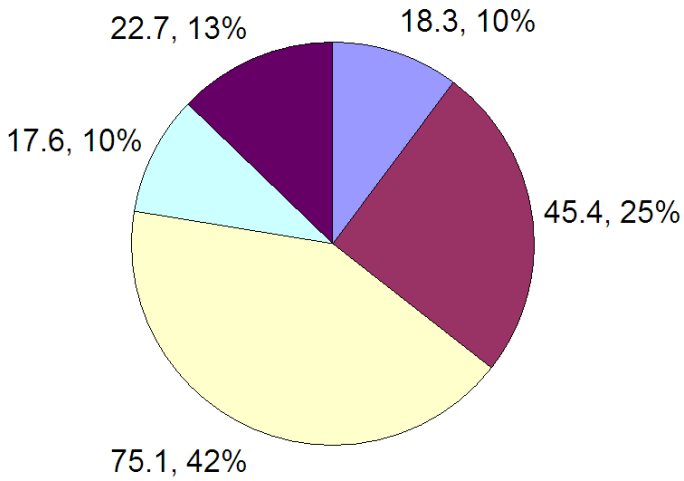


Corner detector: CUDA implementation

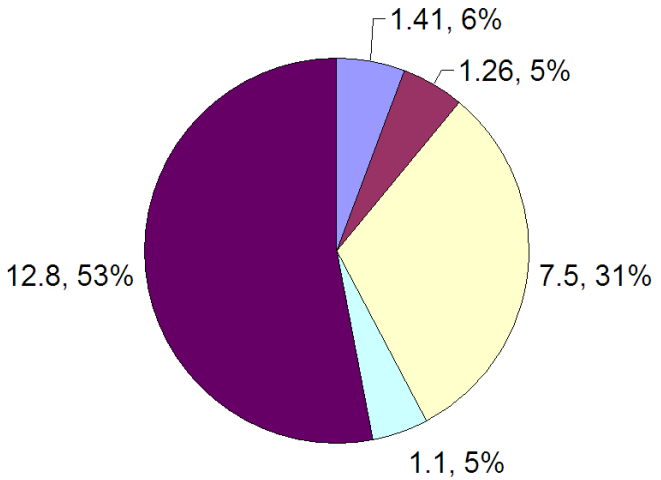
- Summary



CPU



GPU



Feature matching: CUDA implementation

- Feature pairs selection
  - For each feature f_i in image **A**
    - NCC is computed between f_i and g_k in image **B**, with $k \in [\ 0, K]$
    - Select the best correlated pair (f_i, g_k)
    - K can be fixed or dependent (*e.g.* features in a region of interest)
  - CUDA implementation
    - Sorting NCC scores done in shared memory

Table 3. Timing measures for the feature matching kernel

Feature matching (NCC)	CPU (ms)	GPU (ms)	Speed-up
1000 x 1000 features	398	12.5	31.84
1000 x 16 features	6.05	0.11	54.5
16x16 features	0.11	0.010	10