Optimizing co-occurrence matrices on graphics processors using sparse representations

Francisco Igual[1] Rafael Mayo[1]
Timothy Hartley[2] Umit Catalyurek[2]
Antonio Ruiz[3] Manuel Ujaldón[3]

Department of Computer Engineering and Computer Science.
University Jaume I. Castellón (Spain)

Departments of Biomedical Informatics and Electrical and Computer Engineering.
The Ohio State University, Columbus (USA)

Computer Architecture Department.
University of Malaga (Spain)

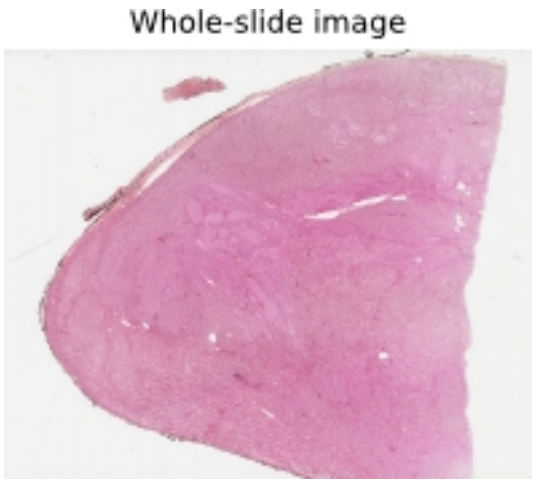
Outline

Biomedical analysis on the GPU

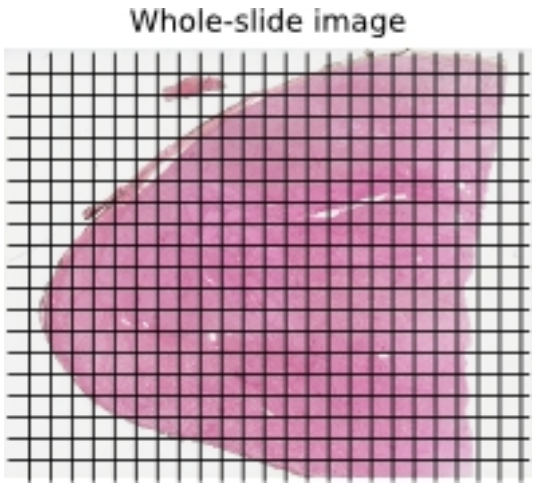


- Cancer prognosis is based on evaluation of tissue samples using a microscope to obtain large-scale images
- The analysis of pathology images is particularly challenging due to the large size of the data
- Image processing on the GPU has attracted many researchers for data mining, segmentation, . . . and fits well within this framework
- Our goal is to optimize the efficient execution of large-scale biomedical image analysis applications on the GPU

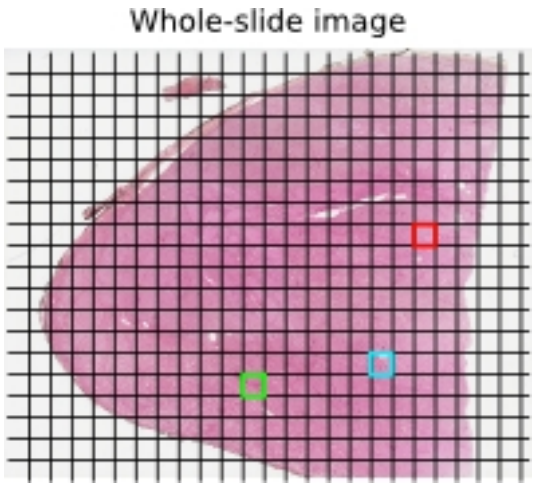
Computation infrastructure

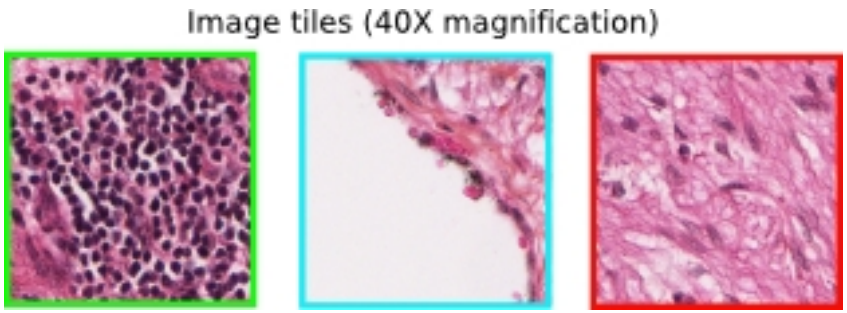


Computation infrastructure

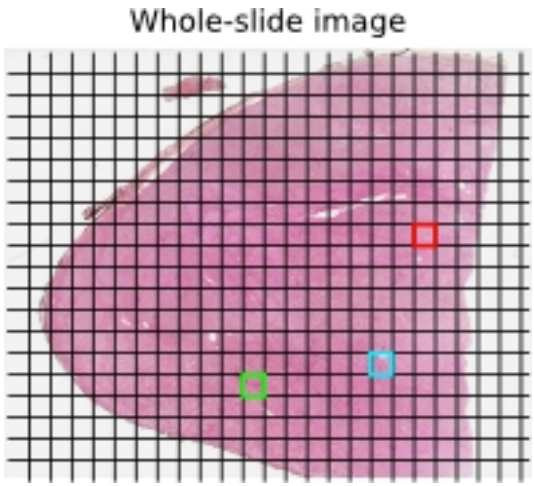


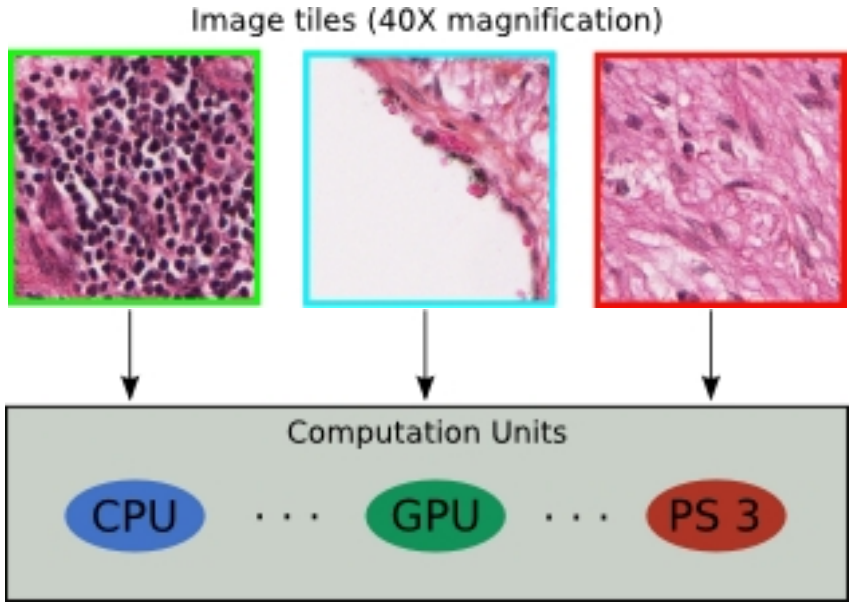
Computation infrastructure



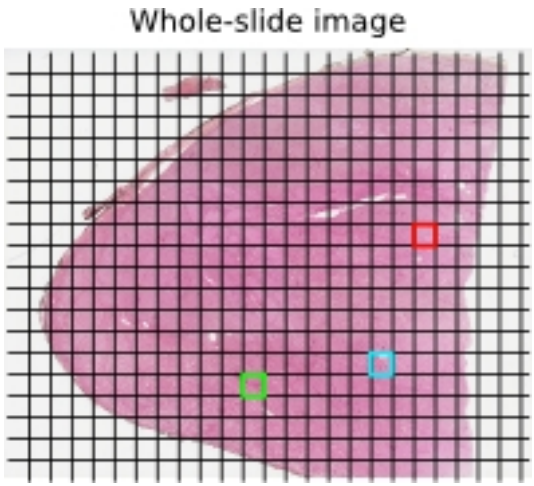


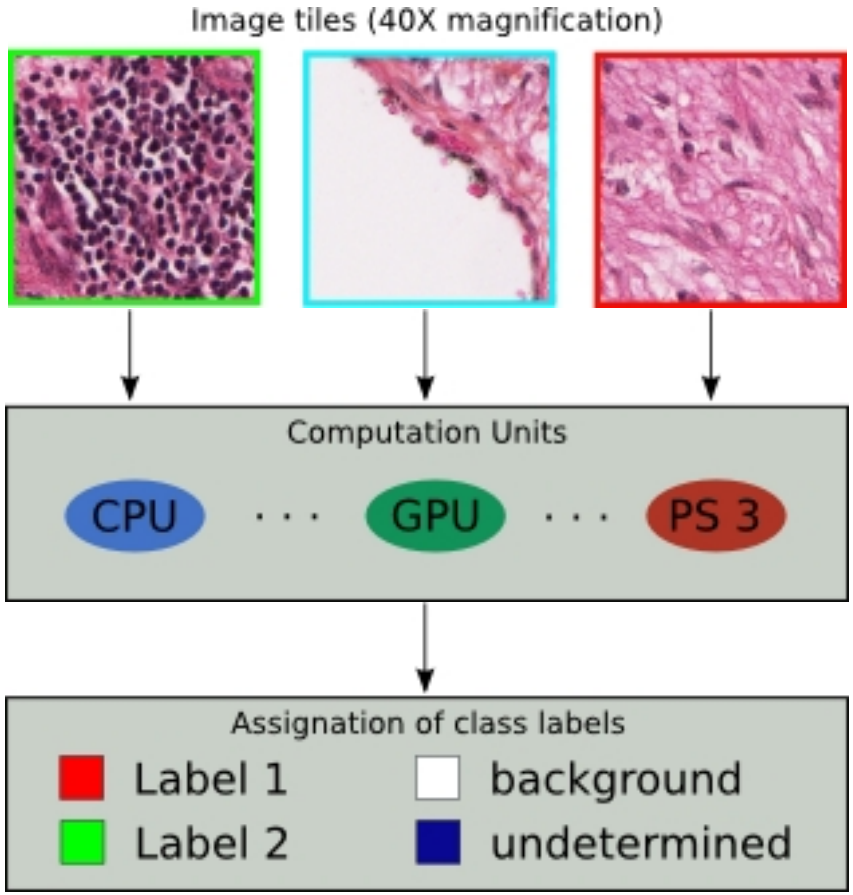
Computation infrastructure



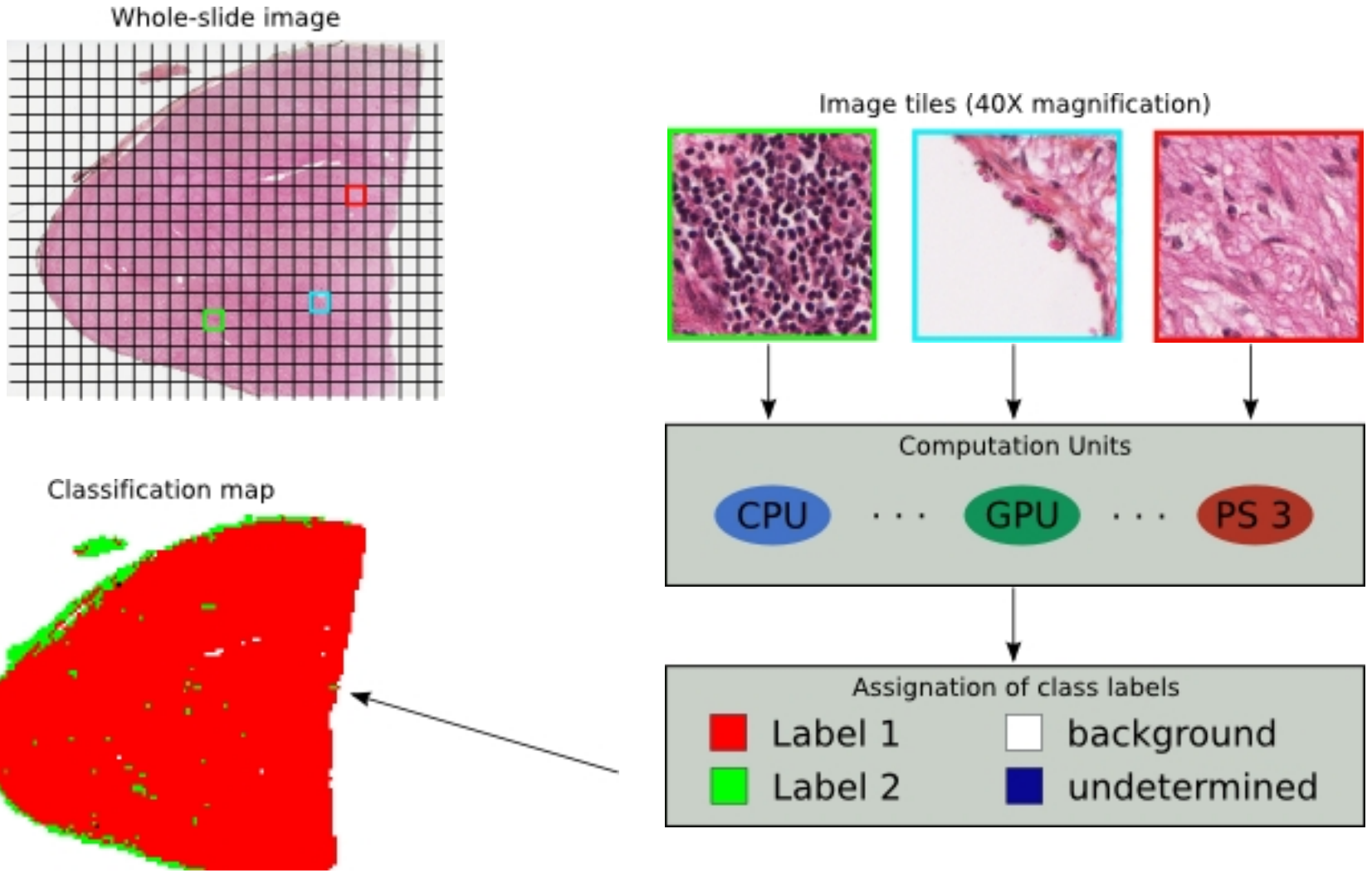


Computation infrastructure

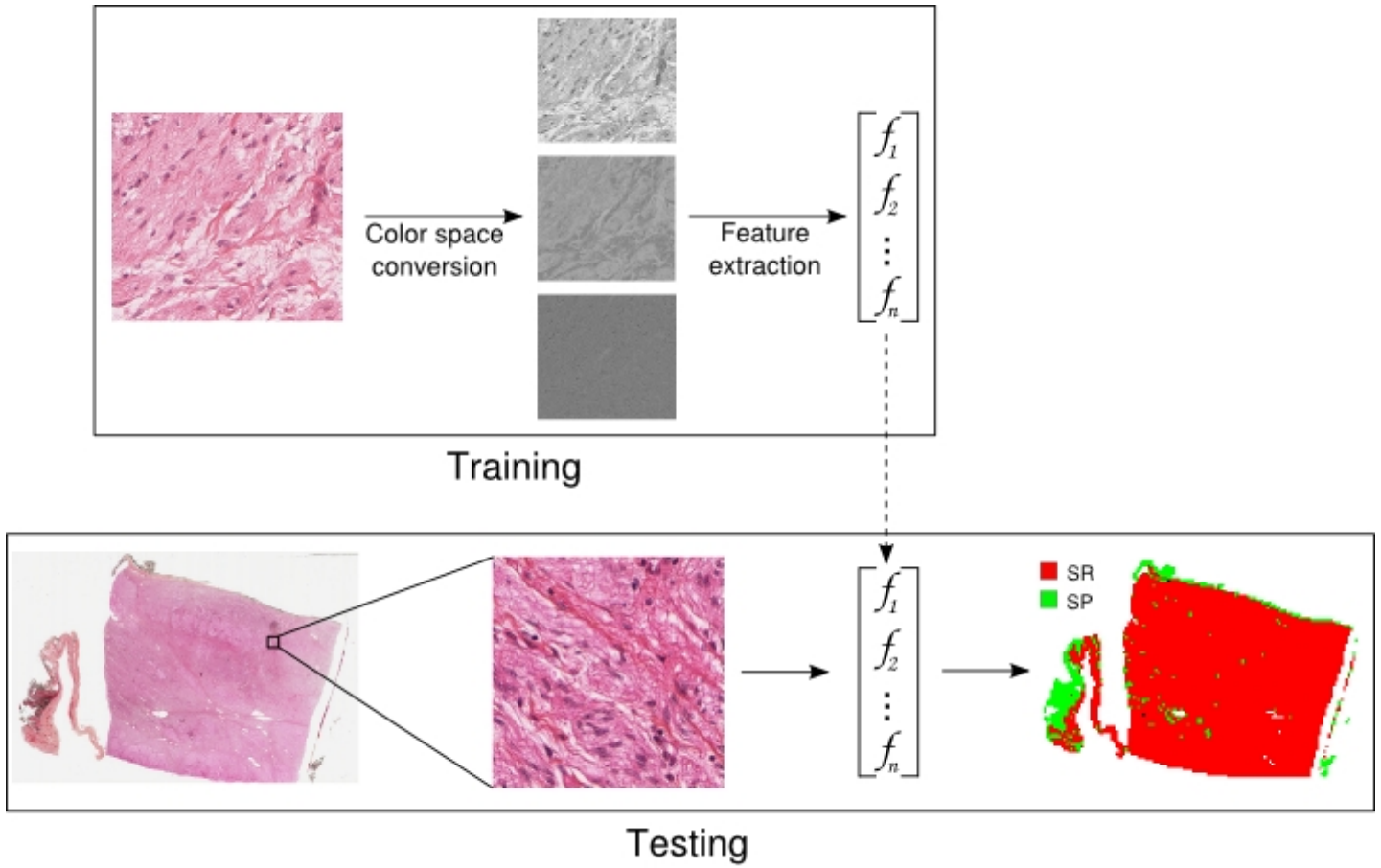




Computation infrastructure



Typical operations on biomedical image analysis



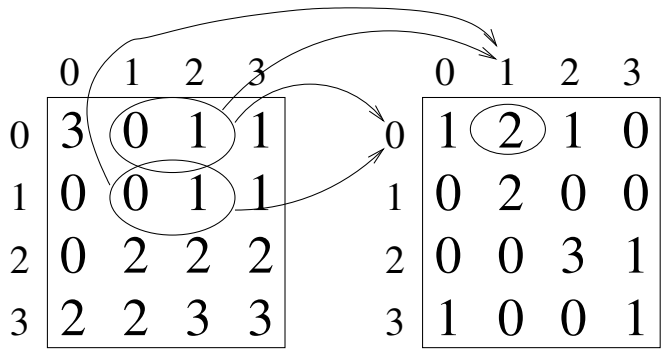
- Color space conversions: Typical GPU-like streaming operators
- Feature extraction: CPU-like operators. Should be optimized

Textural features based on co-occurrence matrices

Introduced by Haralick, 1973

- Joint histogram of intensity levels of a pair of pixels with a given spatial relationship, $[d_x, d_y]$
- Captures the statistics of spatial variation of intensities
- Used as intermediate data structure for calculating features: contrast, correlation, energy, homogeneity...

- Example for a small image with four intensity levels:



Co-occurrence matrix variants

- **For each image pixel** or image tile to classify
- Window size: centered on computed pixel/tile
  - Smallest window: 4×4
  - Largest window: 256×256

  Analyzed the entire set of windows
- Discretized color space or **whole-range** $(0..255)$ in RGB color space
- For each color channel or **using gray scale values**

Experimental setup

- Input images used: taken from real biomedical applications
- Hardware platform: CPU vs GPU comparison
- Data structures: dense vs sparse matrices

Why do we need HPC here? (I)

- Due to the large sizes of whole-slide images
  - A 120K x 120K image digitized at 40x occupies more than 40GB
- Due to the large processing time on a single CPU
  - Using C++ code for a simple image classifier based on statistical features computed through co-occurrence matrices, it takes 850 ms. for an image tile of 1Kx1K and 20 min. for a 50Kx50K slide
- Due to the large number of medical samples
  - In clinical practice, 8-9 biopsy samples are collected per patient, and a hospital may treat around 500 patients per year, consuming months or even years of computational time using C++ on a CPU

In previous studies, we have reduced the time to few days using the GPU, and to few hours using CPU/GPU clusters

Why do we need HPC here? (I)

- Due to the large sizes of whole-slide images
  - A 120K x 120K image digitized at 40x occupies more than 40GB
- Due to the large processing time on a single CPU
  - Using C++ code for a simple image classifier based on statistical features computed through co-occurrence matrices, it takes 850 ms. for an image tile of 1Kx1K and 20 min. for a 50Kx50K slide
- Due to the large number of medical samples
  - In clinical practice, 8-9 biopsy samples are collected per patient, and a hospital may treat around 500 patients per year, consuming months or even years of computational time using C++ on a CPU

In previous studies, we have reduced the time to few days using the GPU, and to few hours using CPU/GPU clusters

Why do we need HPC here? (I)

- Due to the large sizes of whole-slide images
  - A 120K x 120K image digitized at 40x occupies more than 40GB
- Due to the large processing time on a single CPU
  - Using C++ code for a simple image classifier based on statistical features computed through co-occurrence matrices, it takes 850 ms. for an image tile of 1Kx1K and 20 min. for a 50Kx50K slide
- Due to the large number of medical samples
  - In clinical practice, 8-9 biopsy samples are collected per patient, and a hospital may treat around 500 patients per year, consuming months or even years of computational time using C++ on a CPU

In previous studies, we have reduced the time to few days using the GPU, and to few hours using CPU/GPU clusters

Why do we need HPC here? (II)

But we have even harder scenarios...

- Calculation of a co-occurrence matrix **per pixel**
- **Non-discretized** co-occurrence matrix (256×256 size per pixel)
- Variable **window sizes**, from 4×4 to 256×256

Sparse co-occurrence matrices

These conditions typically lead to a sparse co-oc. matrix per pixel:

- 4×4 window
- Non-discretized co-oc. matrix 256×256

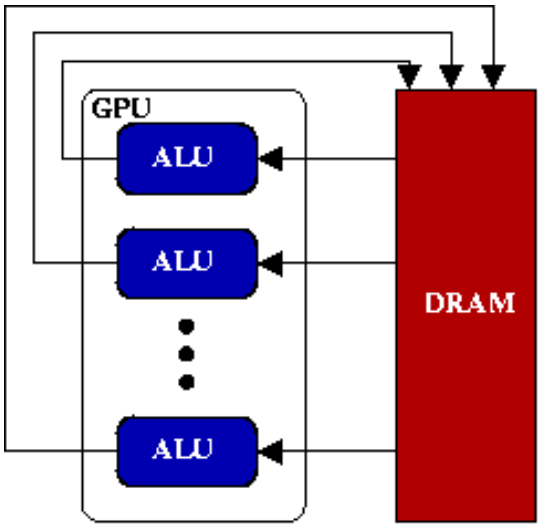
$\Rightarrow$

At most 16 of the 65K elements are nonzeros

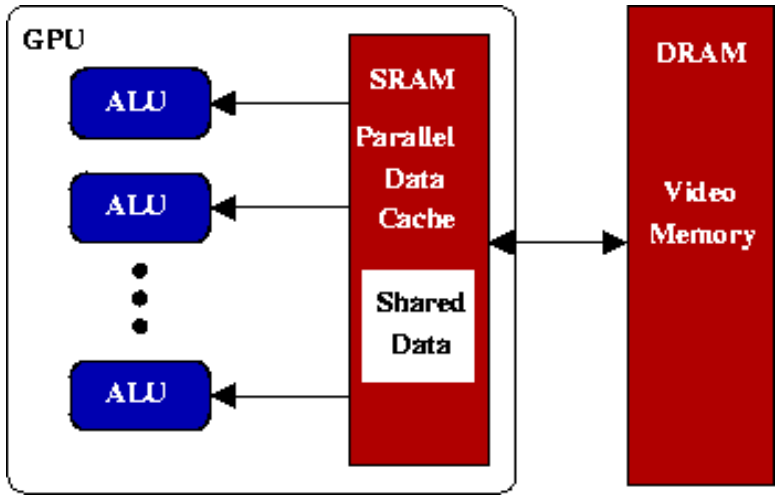
Evolution for the memory paradigms

- CPU (from the 70's):

- GPU (this decade):

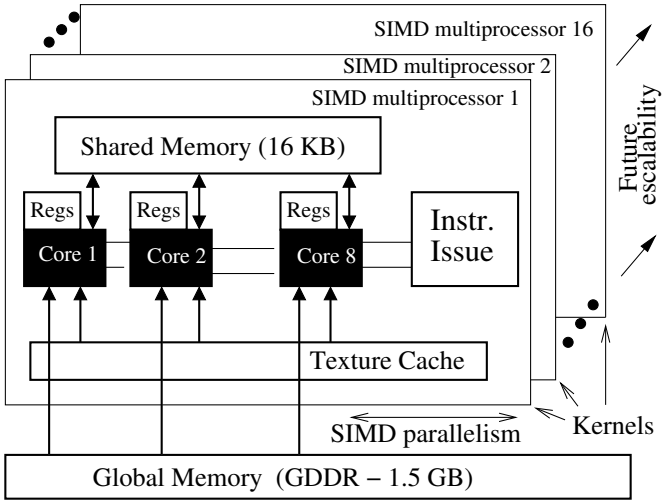


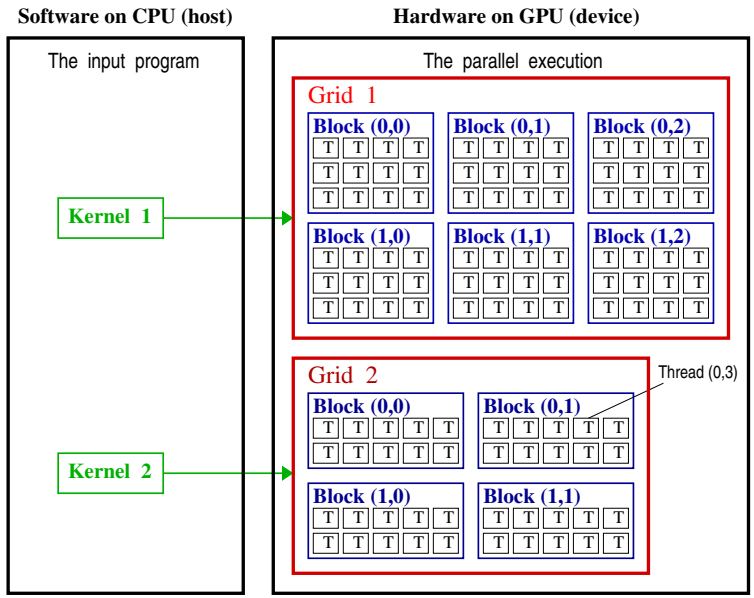
- CUDA (since 2007):



Our goal on the GPU platform

- Overall goal:
  - Fully exploit the parallelism inherent in the application
- Task to achieve this goal:
  - Develop high-quality kernels for use on GPU





Roadblocks to fast GPU kernels

- CUDA exposes better interface than Cg or assembly programming for general-purpose computing
- But architectural idiosyncrasies still need managing:
  - Memory bank conflicts
  - Long latencies to global memory
  - Correct data structure choice
  - Thread deployment shape
  - Kernels with high data reuse
  - Kernels with low computational load
  - Small local memories

Motivation for using sparse matrices and testing different representation formats

Implementation goals

Our scenario

- Calculation of per-pixel co-occurrence matrix
- Variable window sizes (relatively small)
- Non discretized (256×256) co-occurrence matrices

Our goals

- Architectural point of view: use shared memory as much as possible
- Implementation based on sparse formats
- Choose a format simple enough to compute as many threads as possible

Implementation goals

Our scenario

- Calculation of per-pixel co-occurrence matrix
- Variable window sizes (relatively small)
- Non discretized (256×256) co-occurrence matrices

Our goals

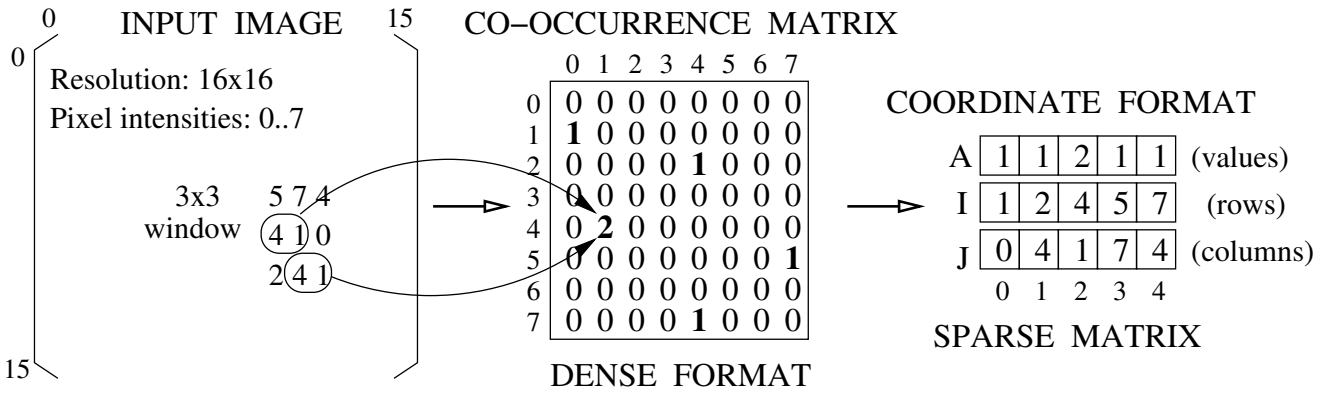
- Architectural point of view: use shared memory as much as possible
- Implementation based on sparse formats
- Choose a format simple enough to compute as many threads as possible

Sparse matrix format

The format is critical for attaining high performance

- Simple to fit the GPU execution model
- Compact to fit in shared memory

Our proposal: coordinate format



Alternative formats for sparse matrices

There are more sparse formats:

Single linked lists

- Uses an ordered linked list per row
- Benefits: efficient insertion

Double linked lists by rows and columns

- Uses a double linked list of columns and rows
- Benefits: allows optimizations for some matrix shapes

Common disadvantages

More storage space

↓

Less threads per block computing co-occurrence matrices

Alternative formats for sparse matrices

There are more sparse formats:

Single linked lists

- Uses an ordered linked list per row
- Benefits: efficient insertion

Double linked lists by rows and columns

- Uses a double linked list of columns and rows
- Benefits: allows optimizations for some matrix shapes

Common disadvantages

More storage space

$\downarrow$

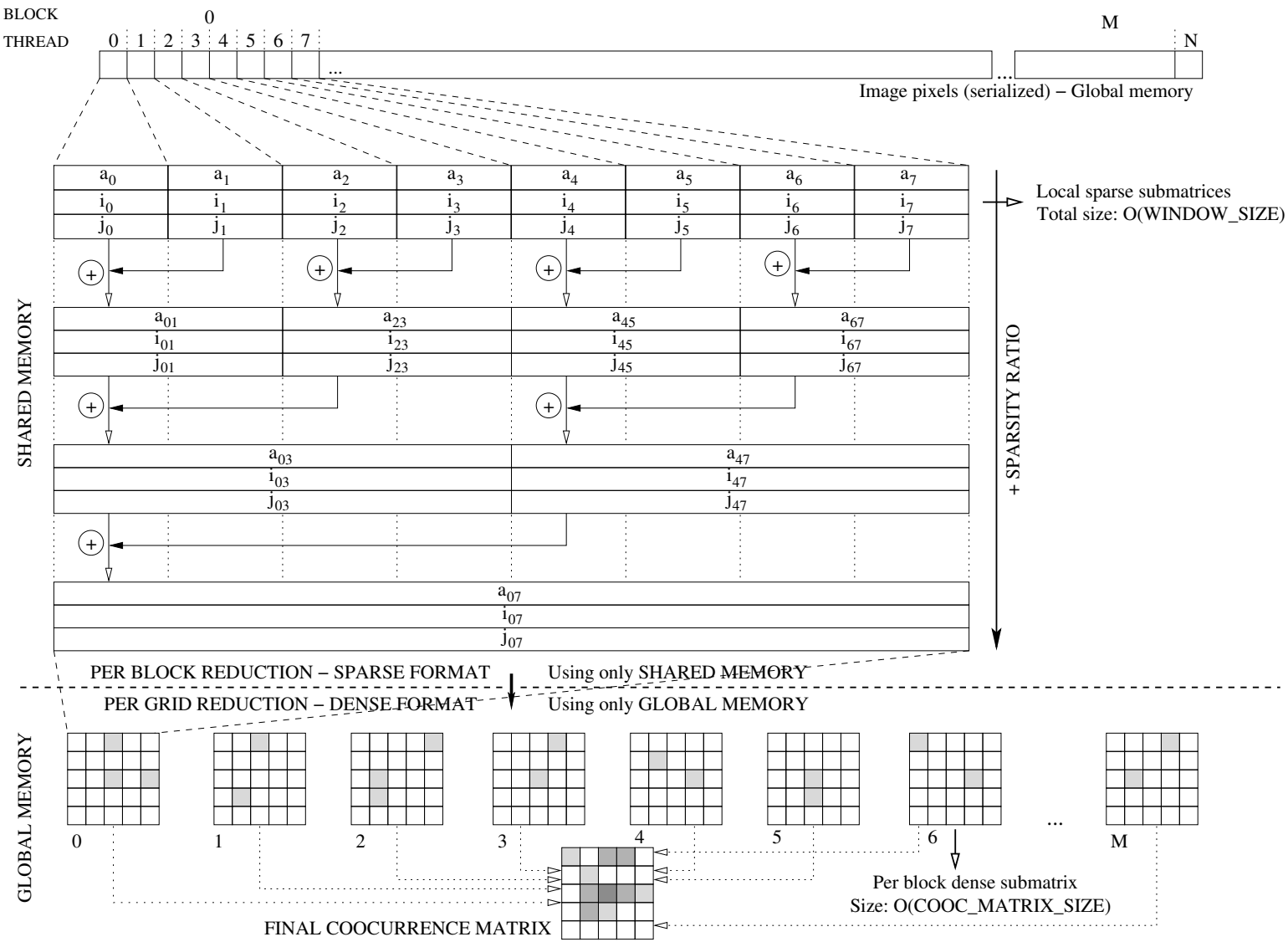
Less threads per block computing co-occurrence matrices

The computational process using CUDA (I)

The algorithm can be shared for all sparse formats and has two phases:

1. Shared memory phase $\Rightarrow$ Sparse format
   - Each thread operates on a chunk of the original window
   - Creates a local sparse representation
   - Sequentially, a reduction process is performed
   - Sparse to dense conversion
   - One dense matrix per block is written to global memory
2. Global memory phase $\Rightarrow$ Dense format
   - The second phase is performed on global memory
   - Global reduction process using shared memory (optimized)
   - Usually attaining good performance results

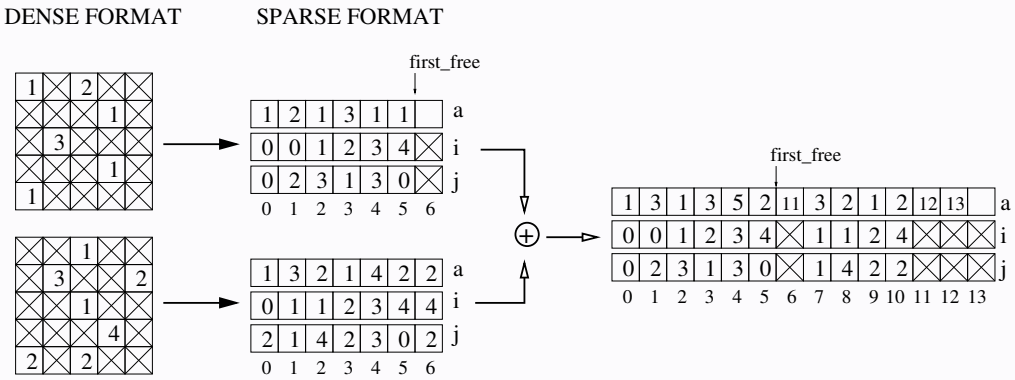
The computational process using CUDA (II)



Dense-sparse conversions and reduction process

- Sparse formats are inefficient in global memory:
  - No data coalescing
  - No regular access pattern
  - Expensive insertion/modification
- Instead, we decided to use dense formats because:
  - Sparse-dense transformation is cheap
  - There are not space restrictions in global memory

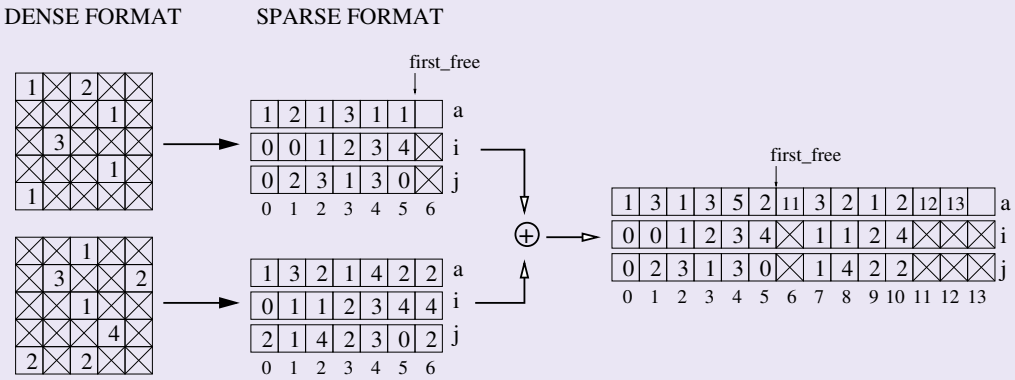
Reduction process



Dense-sparse conversions and reduction process

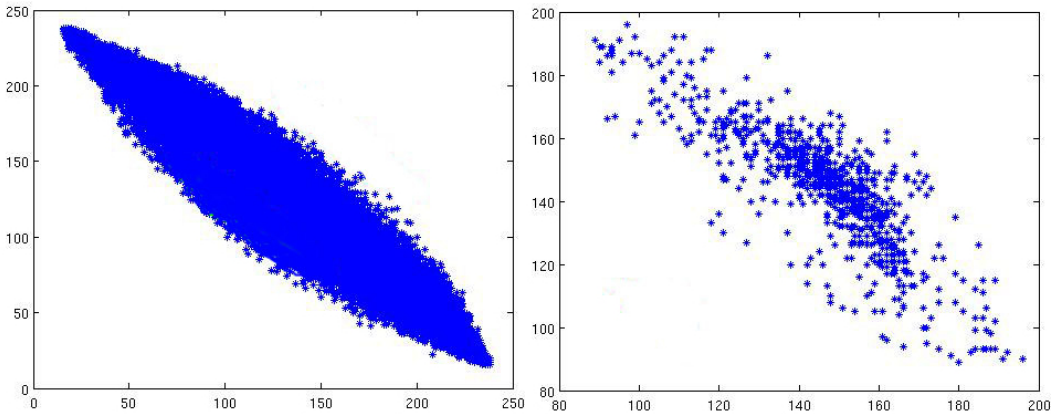
- Sparse formats are inefficient in global memory:
  - No data coalescing
  - No regular access pattern
  - Expensive insertion/modification
- Instead, we decided to use dense formats because:
  - Sparse-dense transformation is cheap
  - There are not space restrictions in global memory

Reduction process

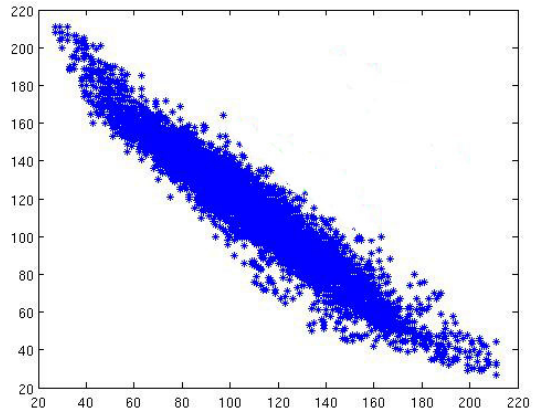


Rationale for optimizations

- Nonzero elements tend to be concentrated in the main diagonal of the matrix
- Some empirical examples from biomedical apps



Cartilage and bone regeneration using different stainings



Neuroblastoma cancer prognosis

Potential optimizations

We have performed some optimizations:

Diagonal shape

It is possible to store some diagonals in a dense format

Improving insertion

- Contiguous pixels of images are related
- It is possible to apply techniques to improve consecutive insertions when they are next
- **Example**: keep the position of the last insertion to accelerate future search

However, most of them do not offer good performance results
Reason: too sparse matrices / too small window sizes

Potential optimizations

We have performed some optimizations:

Diagonal shape

It is possible to store some diagonals in a dense format

Improving insertion

- Contiguous pixels of images are related
- It is possible to apply techniques to improve consecutive insertions when they are next
- **Example**: keep the position of the last insertion to accelerate future search

However, most of them do not offer good performance results
Reason: too sparse matrices / too small window sizes

Our benchmark

- Input images: taken from bone and cartilage regeneration and precomputed to obtain grayscales
- CPU: Intel Core 2 Duo
- GPU: Nvidia GeForce 8800 GPU
- Experimental numbers measured for different window sizes and discretized levels

Window size impact

- Results (in ms.) for different window sizes
- Using non-discretized co-oc. matrix

Window	CPU	GPU dense	GPU sparse	Nonzeros	Speedup	GPU/CPU
4x4	1.36	7.61	0.10	0.024%	76.1x	13.60x
8x8	2.82	7.62	0.16	0.090%	47.62x	17.62x
16x16	2.82	7.58	0.39	0.390%	19.43x	7.23x
32x32	3.04	7.63	0.74	1.560%	10.31x	4.10x
64x64	3.08	7.76	1.74	6.250%	4.45x	1.77x
128x128	2.94	8.54	7.70	25%	1.1x	0.38x
256x256	2.96	9.19	46.49	100%	0.19x	0.32x

- Best results achieved for Sparse Formats on GPU for matrices sparse enough
- Even better results that those on CPU

Encouraging results

Potential optimizations

- Results (in ms.) of different window sizes
- Using non-discretized co-oc. matrix

Window	Coordinate	Simple link	Double link	Optimally
4x4	0.10	0.13	0.30	0.21
8x8	0.14	0.25	0.58	0.37
16x16	0.36	1.02	1.52	1.10
32x32	0.45	2.31	4.91	4.83
64x64	1.13	3.46	6.52	6.90
128x128	6.58	19.85	23.33	24.41
256x256	43.19	65.99	78.19	78.31

- Best results are achieved for the simplest implementation
- Less storage space means more threads computing co-oc. matrices per block

Disappointing results

Discretization impact

- Results (in ms.) of different discretization level of the co-occurrence matrix
- Window size: 16×16

Co. size	CPU	GPU dense	GPU sparse	Nonzeros	Speedup	GPU/CPU
16x16	2.82	0.23	0.21	100%	1.09x	13.42x
32x32	2.82	0.31	0.27	25%	1.14x	10.44x
64x64	2.82	0.67	0.28	6.25%	2.39x	10.07x
128x128	2.82	2.09	0.33	1.56%	6.33x	8.54x
256x256	2.82	7.58	0.39	0.39%	19.43x	7.32x

- Fixing the window size (16×16, very sparse matrix), sparse Representation is the winner
- The best results are achieved for the most sparse matrices
- Particularizing the window size to 16x16, the sparse representation always performs better

Outstanding results

Results discussion

CPU vs. GPU

- Dense representations on GPU did not achieve good results for small window sizes
- With sparse formats, we overcome this deficit versus the CPU, to achieve a two-digit speed-up on GPUs

Sparsity rate

- Sparse representations are more effective on:
  - Co-occurrence matrix per pixel (a tile is too big)
  - Relatively small window sizes $\Rightarrow$ High sparsity
- This situation is **very common** in biomedical imaging

Results discussion

CPU vs. GPU

- Dense representations on GPU did not achieve good results for small window sizes
- With sparse formats, we overcome this deficit versus the CPU, to achieve a two-digit speed-up on GPUs

Sparsity rate

- Sparse representations are more effective on:
  - Co-occurrence matrix per pixel (a tile is too big)
  - Relatively small window sizes $\Rightarrow$ High sparsity
- This situation is **very common** in biomedical imaging

Results discussion

In conclusion

- We achieve speedups of 20x - 50x for typical window sizes over dense representations
- We even achieve interesting speedups over CPU
- Remember: feature extraction was a CPU-like operation...
- Our main goal is to optimize each step of the image analysis process $\Rightarrow$ no CPU use until the GPU process finishes

Conclusions

- Biomedical image analysis is a HPC-demanding application
- There are processes that fit perfectly to the GPU execution model
- . . . but feature extraction is not included among them
- We propose an algorithm to compute co-oc. matrices for the most typical scenarios
- It is necessary to take into account both the architecture and the execution model, which explains why a more complex and efficient format does not lead to better performance
- Optimizing ALL the steps of our process allows us to execute it exclusively on GPU, with no transfers penalty nor CPU occupancy

Future work

- This effort is part of the developing of an image processing library for biomedical applications implemented entirely on the GPU using CUDA
- Other methods already implemented:
  - Color channel conversions
  - LBP operator for neighbour analysis
  - Zernike moments for texture analysis
  - Clustering algorithms such as k-means and EM-LDA
- Our plan includes the implementation into high-performance platforms such as TESLA nodes and CPU/GPU clusters:
  - *Some promising results to appear on the 22nd ACM ICS'08 next month*

Thank you...

For more information...

http://www3.uji.es/~figual
figual@icc.uji.es