BIPGPU

Una biblioteca optimizada para el procesamiento de imágenes biomédicas sobre procesadores gráficos

Francisco Igual[1] Rafael Mayo[1]
Antonio Ruiz[2] Manuel Ujaldón[2]

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

Computer Architecture Department.
University of Málaga (Spain)

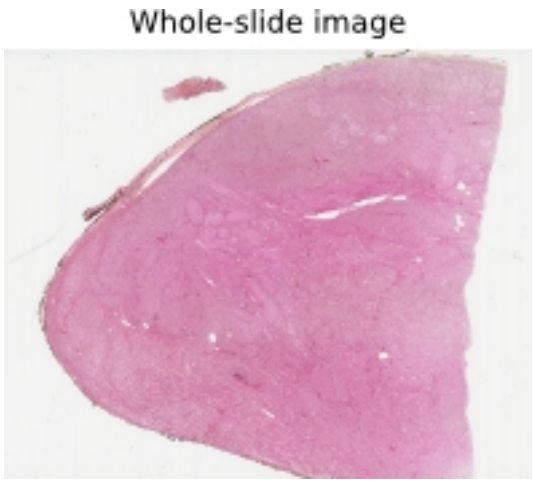
Outline

Biomedical analysis on the GPU

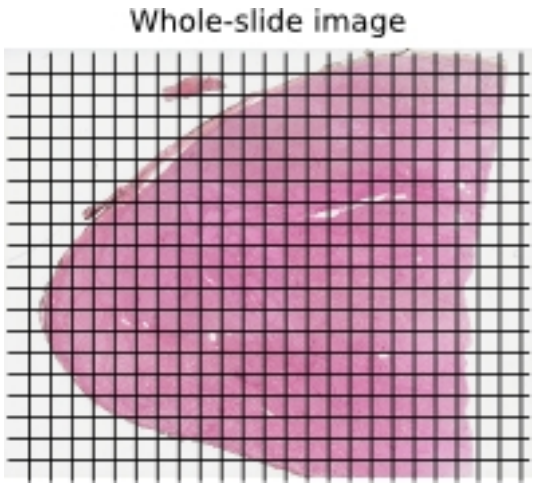


- Cancer prognosis is based on evaluation of tissue samples using a microscope to obtain large-scale images
- The analysis of pathology images is particularly challenging due to the large size of the data
- Image processing on the GPU has attracted many researchers for data mining, segmentation, . .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

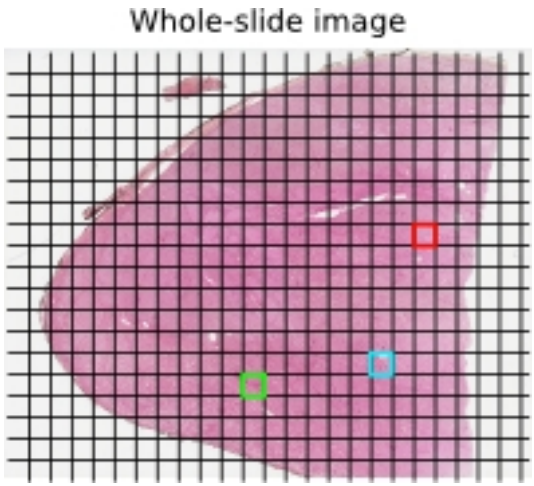
Computation infrastructure

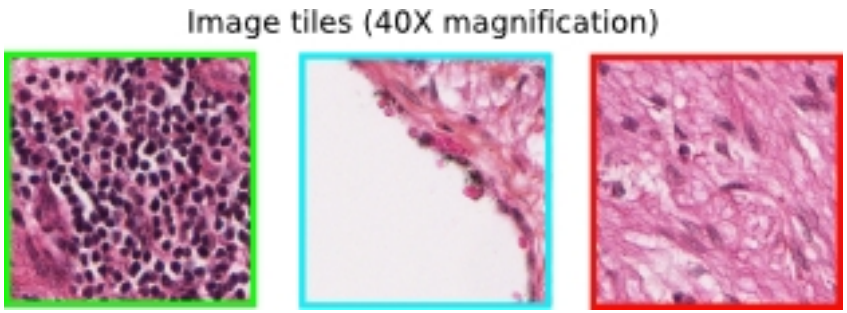


Computation infrastructure

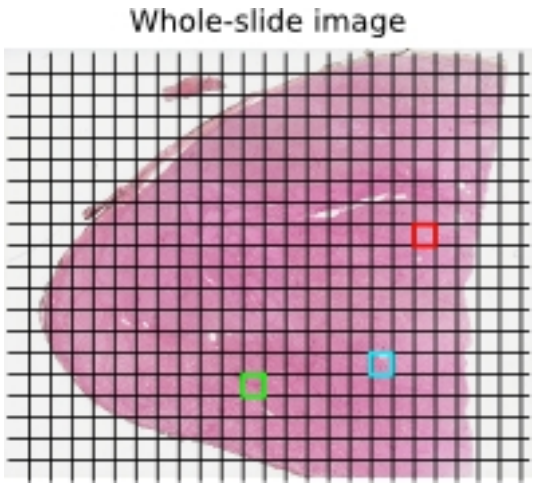


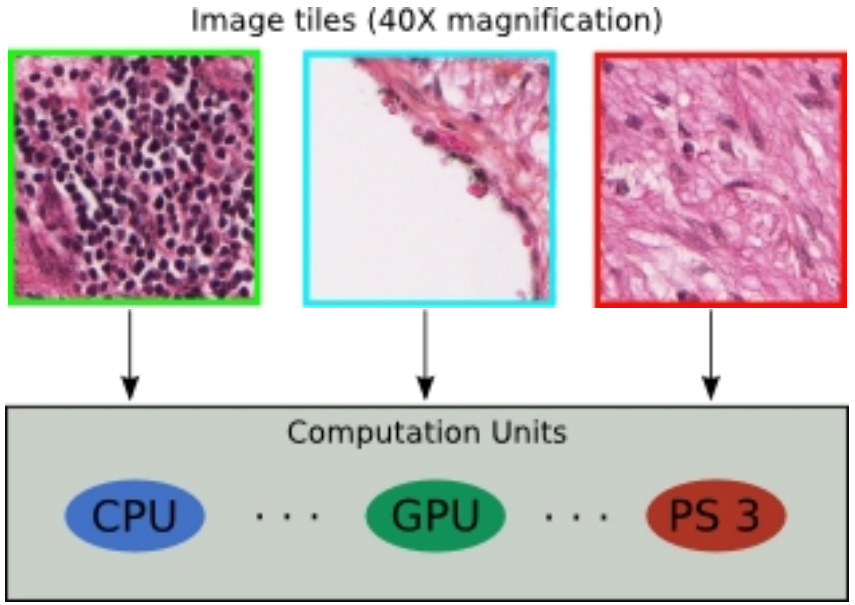
Computation infrastructure



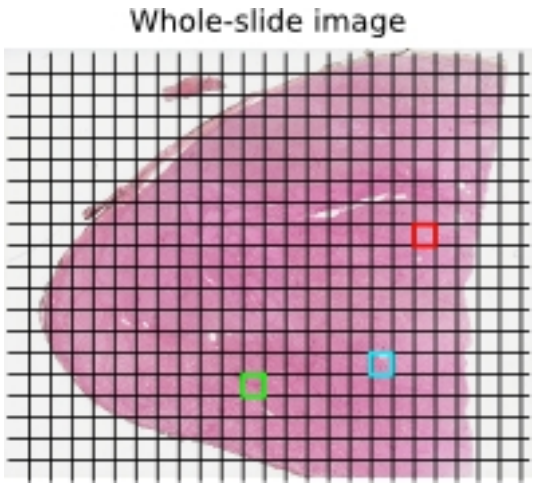


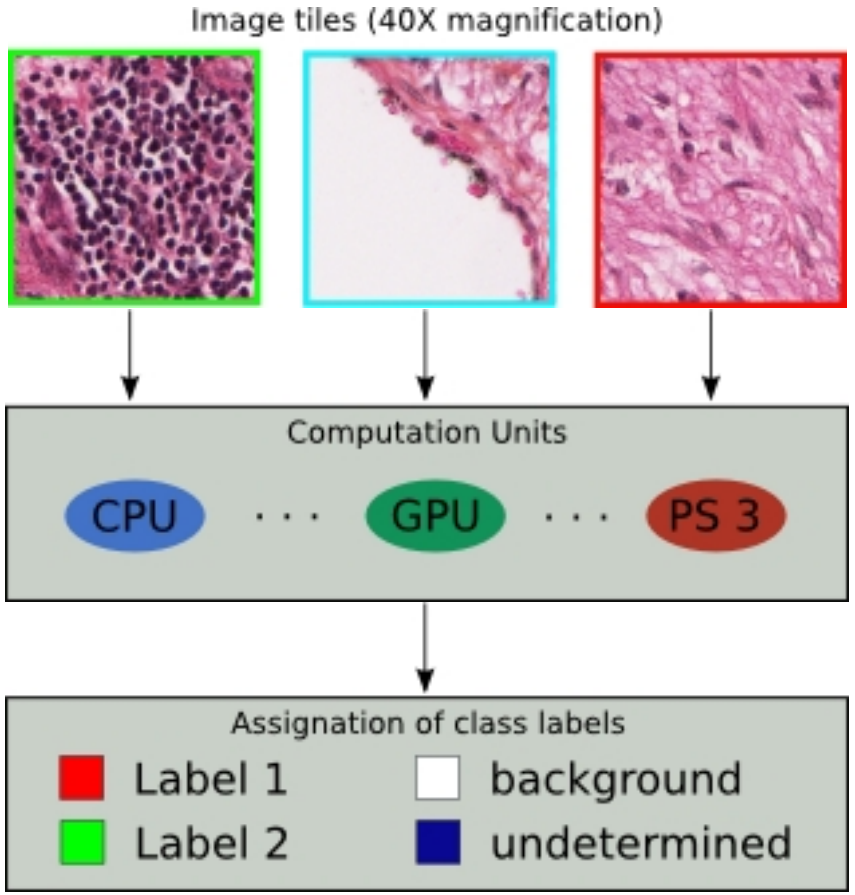
Computation infrastructure



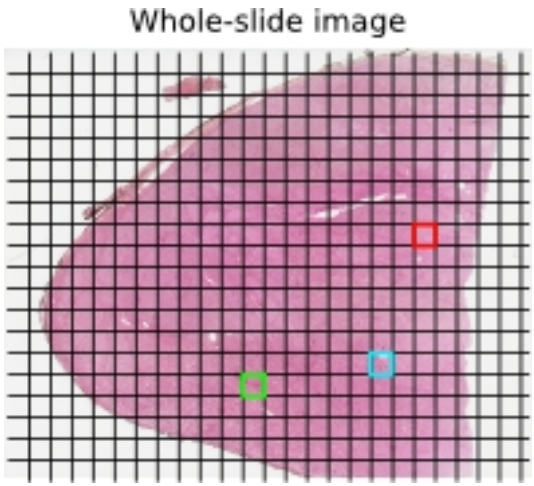


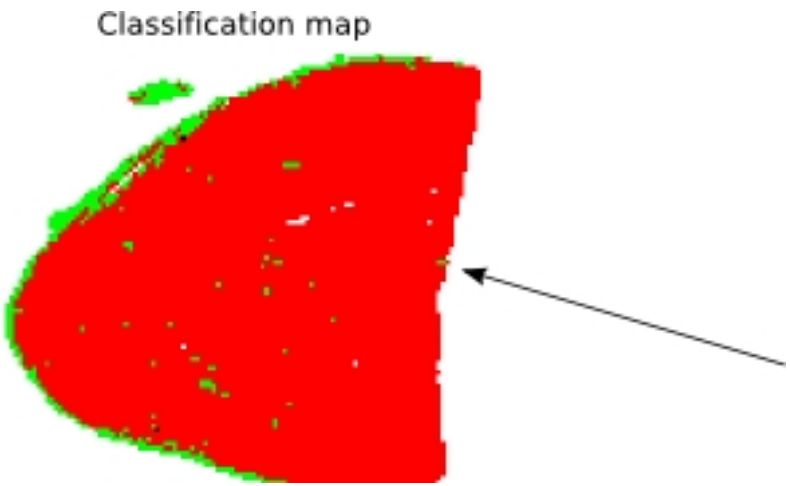
Computation infrastructure

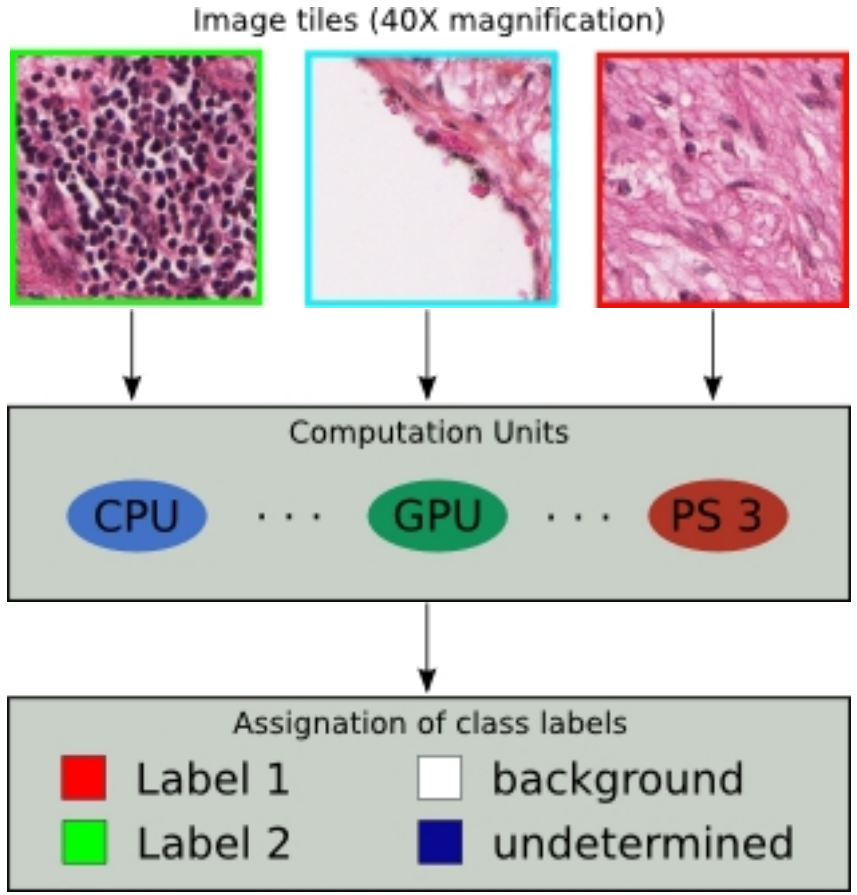




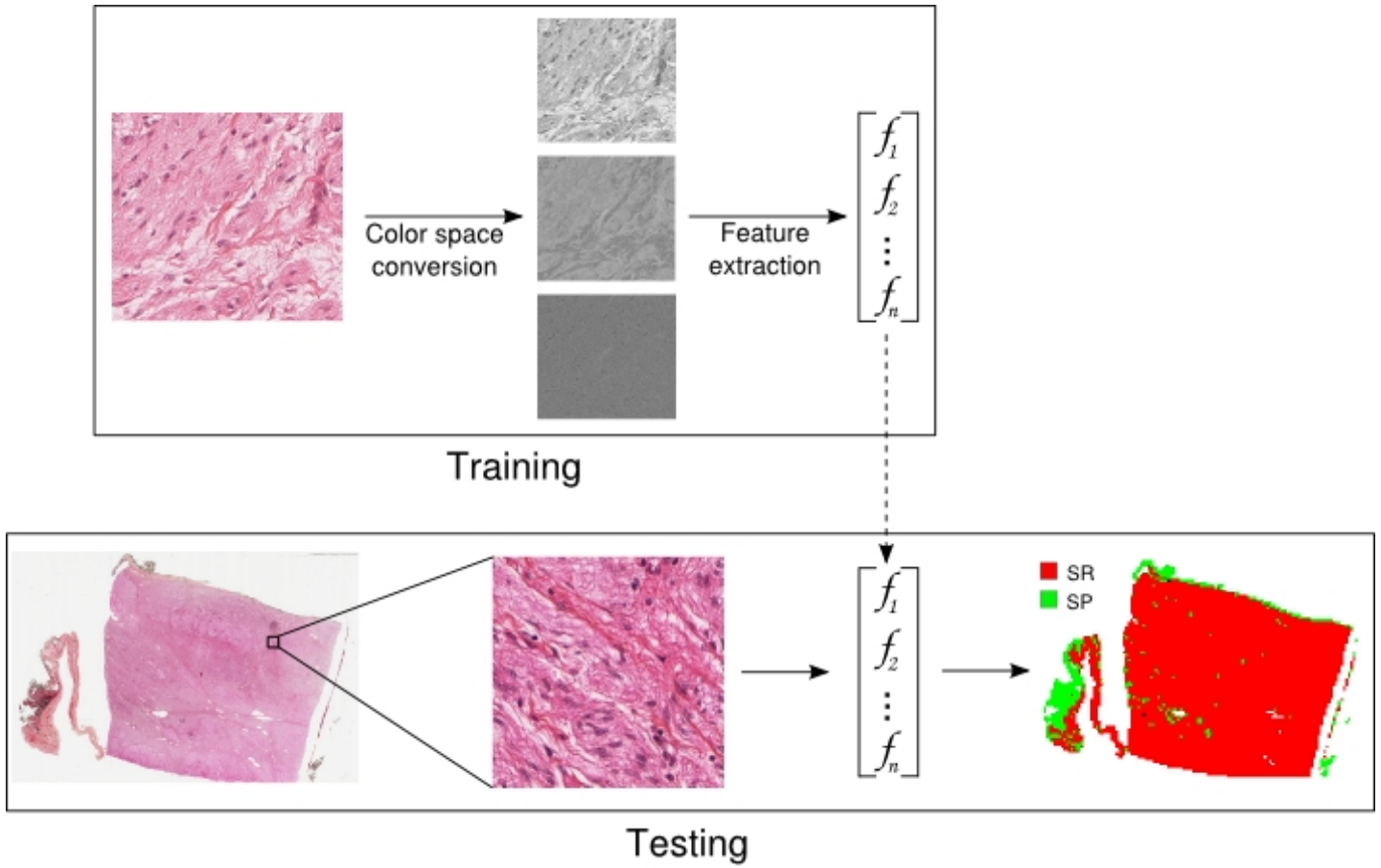
Computation infrastructure







Typical operations on biomedical image analysis



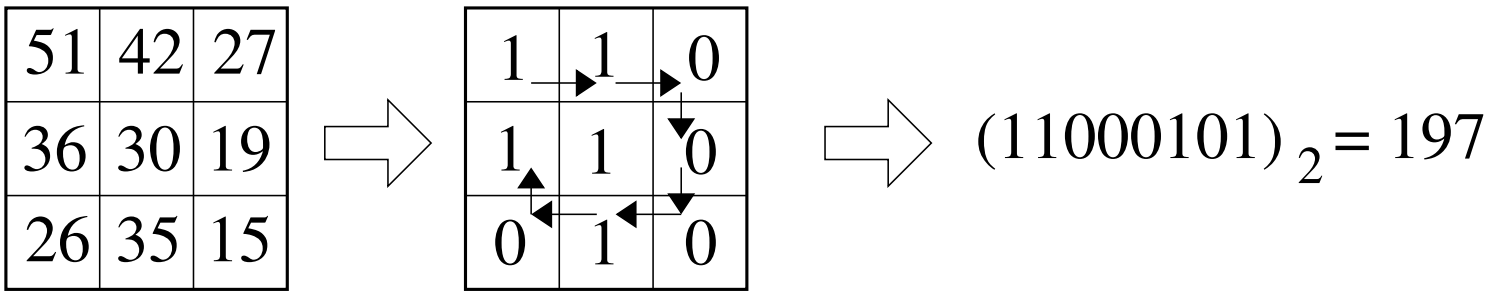
- Color space conversions: Typical GPU-like streaming operators
- Feature extraction: CPU-like operators. Should be optimized

Color conversions

- Typical streaming operations
- Used for a high-fidelity representation depending on the framework within image processing, like:
  - Color addition (light treatment in monitors): RGB, XYZ.
  - Color substraction (printing devices in paper): CMYK.
  - Color treatment/handling: Luv, LA*B*, HSV.
  - Digital storage/potography (sRGB, JPEG, PNG).
  - Textures representation for rendering (bitmaps).
- A conversion is typically required from the source format to the desired scope of application.

LBP operator

- Widely used functional feature (e.g. facial expression recognition)
- LBP: invariant to rotation and local or global intensity variations
- Pixel neighborhood from the original image (small windows)

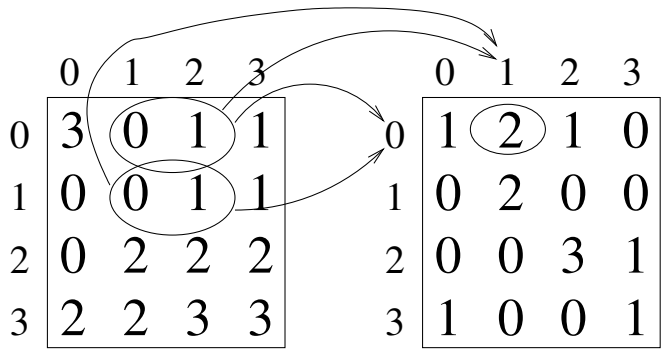


Textural features based on co-occurrence matrices

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

Image descriptors based on Zernike moments

- Spatial domain filters as a direct way to capture texture properties
- Legendre and Zernike polynomials represent an image by a set of mutually independent descriptors.
- The moment within a window centered at a given pixel can be interpreted as a convolution of the image with a mask.
- The more computed moments, the better reconstructed image.

$$V_{pq}(x,y) = R_{pq}(\rho) * e^{jq\theta} \tag{1}$$

- Optimized recursive implementations on CPU (Hwang, Al-Rahi)

Why do we need HPC here?

- 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?

- 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?

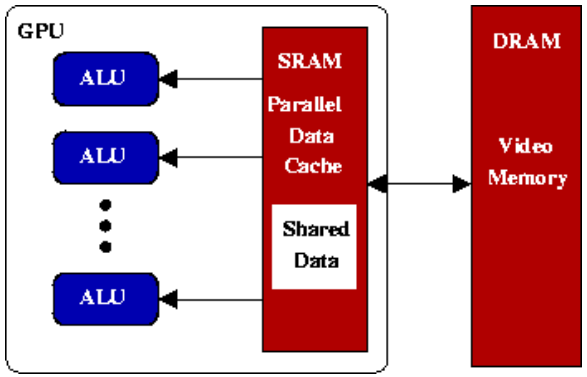
- Due to the large sizes of whole-slide images
  - A 120K x 120K image digitized at 40x occupies more than 40GB
- Due to the large processing time on a single CPU
  - Using C++ code for a simple image classifier based on statistical features computed through co-occurrence matrices, it takes 850 ms. for 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

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

- CUDA (since 2007):



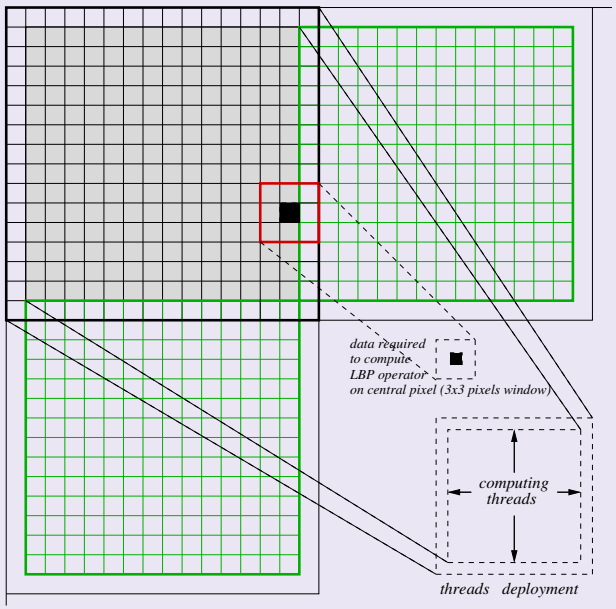
Color conversions and LBP implementation

Color conversions

- Fully streaming operator: without optimizations
- Data coalescing: RGB $\Rightarrow$ padding to 32 bits

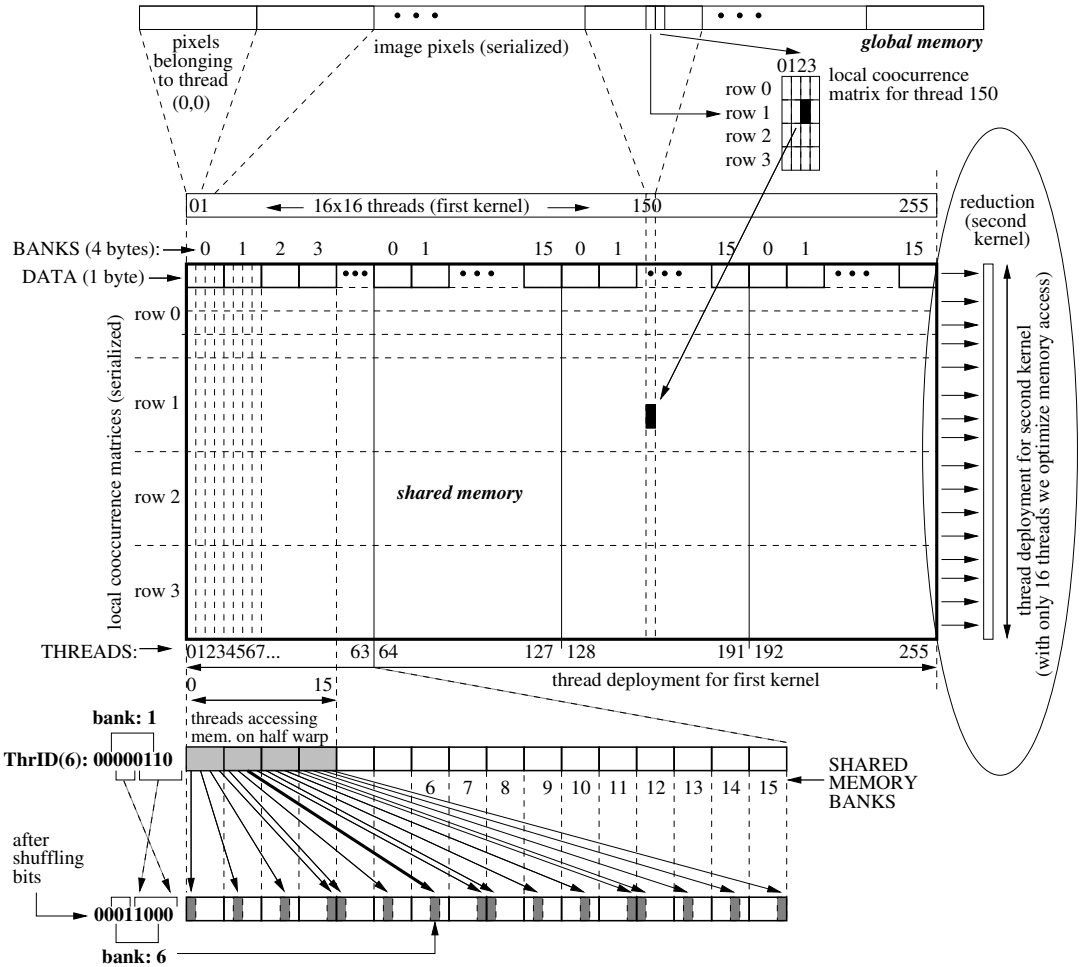
LBP

- Full GPU resource utilization requires heavyweight, complex kernel
- Simpler kernel overlaps windows by one pixel
- Transfers redundant data
- Is still 1.25x faster than heavyweight version



Co-occurrence matrices: discretization and optimizations

- Non-discretized matrices: 256×256 elements. No Sh. Memory
- Solution: discretization

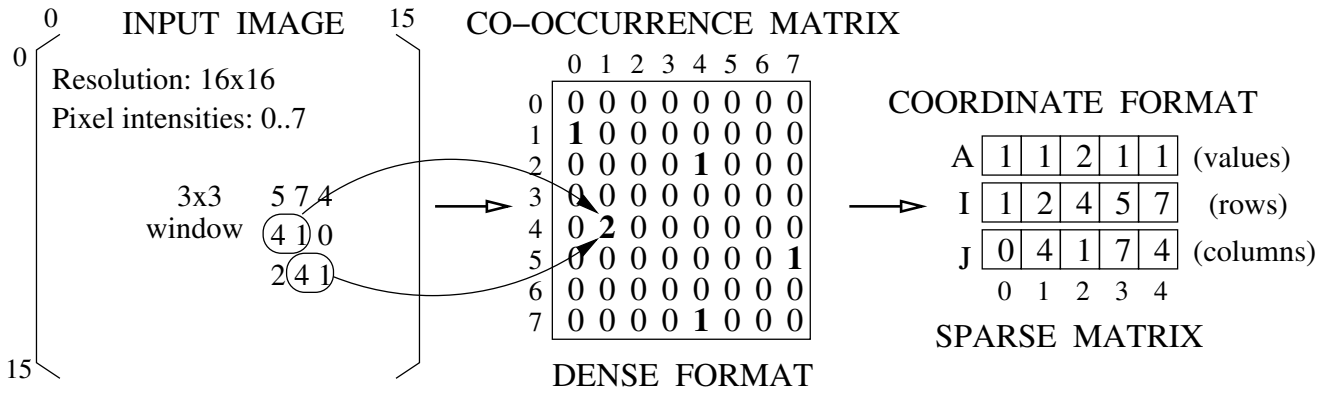


Sparse matrix format

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

$\downarrow$

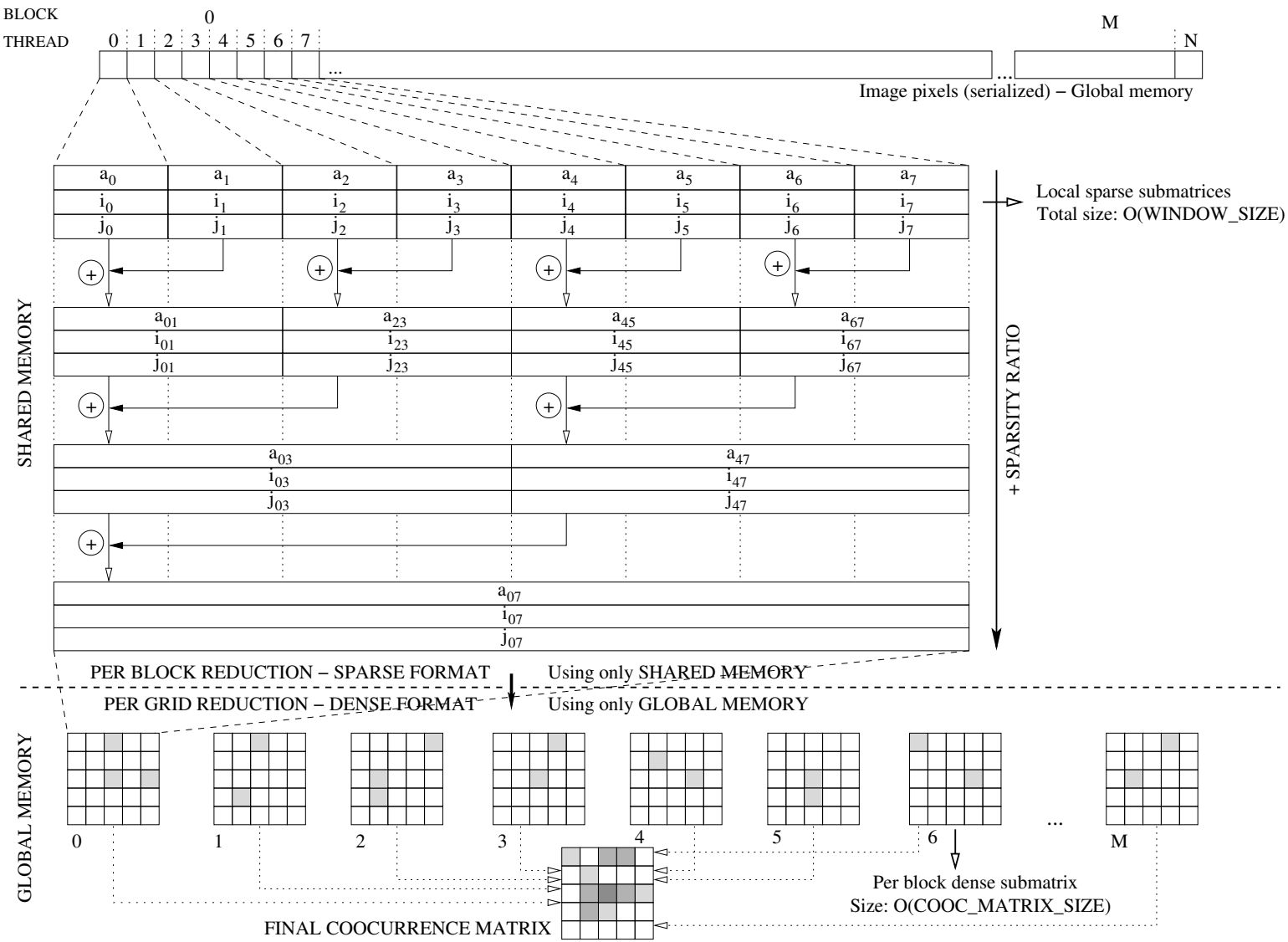
Less threads per block computing co-occurrence matrices

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)



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

Color conversions

Color conversions: fully stream-oriented operation

Format conversion	CPU time	GPU time	GPU Speedup
RGB to XYZ	140.01 ms	1.27 ms	**109.47x**
RGB to Luv	273.83 ms	1.42 ms	**191.62x**
RGB to L*A*B*	267.92 ms	2.23 ms	**119.66x**
RGB to HSV	16.60 ms	0.57 ms	**28.98x**
RGB to sRGB	123.51 ms	1.23 ms	**99.84x**
sRGB to XYZ	16.50 ms	0.43 ms	**37.59x**
sRGB to Luv	150.31 ms	0.57 ms	**263.25x**
sRGB to L*A*B*	144.41 ms	1.29 ms	**111.68x**

LBP operator

- “Simple” GPU implementation outperforms CPU
- Even better results than with Cg

Image size	CPU C++	GPU Cg	GPU CUDA	GPU speedup
128×128	3.95	1.01	0.072	54.86x
256×256	17.83	1.09	0.140	127.35x
512×512	76.70	1.92	0.415	184.81x
1024×1024	310.65	6.88	1.564	198.62x
2048×2048	1234.96	23.91	6.114	201.98x

Co-occurrence matrices: 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

Co-occurrence matrices: 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

Co-occurrence matrices: 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

Zernike moments

- Comparison CPU-GPU (ms.):

All moments of an order	Exec. times 1024x1024 image			Speed-up GPU vs:	
	HWANG (2006)	AL-RAWI (2008)	Direct on GPU	HWANG (2006)	Al-Rawi (2008)
$A_{4,*}$ (3)	258.0	62.5	19.0	13.57x	3.28x
$A_{8,*}$ (5)	859.0	54.5	36.6	23.47x	1.48x
$A_{12,*}$ (7)	1 969.0	62.5	50.5	38.99x	1.23x
$A_{16,*}$ (9)	3 836.0	78.0	68.2	56.24x	1.14x
$A_{20,*}$ (11)	6 586.0	93.5	90.0	73.17x	1.03x

- Optimizations (ms.):

Moments of order	Direct method (GPU)	Optimiz. 1	Optimiz. 2	Optimiz. 3	Speed up
$A_{4,*}(3)$	19.0	14.92	14.03	13.79	1.37x

1. Using `sincosf`
2. Maximizing threads per block (384 optimally)
3. No temporary data structures

Some recommendations

- Implement data parallel, arithmetically intensive, regular codes
- Adapt the algorithm: lightweight and simple threads
- Redisign the algorithm: no recursion, no divergences...
- Choose the correct data structure: compact and simple
- Relax some restrictions. **Goal: use shared memory as much as possible**
- Apply CUDA specific tricks: coalescing, avoid conflicts in shared memory...

Conclusions

- Biomedical image analysis is a HPC-demanding application
- There are processes that fit perfectly to the GPU execution model
- . . . but not all
- It is necessary to take into account both the architecture and the execution model, which explains why a more complex format or algorithm does not lead to better performance
- CUDA-specific improvements can boost performance
- Optimizing ALL the steps of our process allows us to execute it exclusively on GPU, with no transfers penalty nor CPU occupancy