Computational Graphics: Lecture 15 SpMSpM and SpMV, or, who cares about complexity when we have a thousand processors?

The CVDLab Team
Francesco Furiani

Università degli Studi Roma TRE

Tue, April 3, 2014

Problem

Informations, informations, informations (lot of).

We need techniques to work with it:

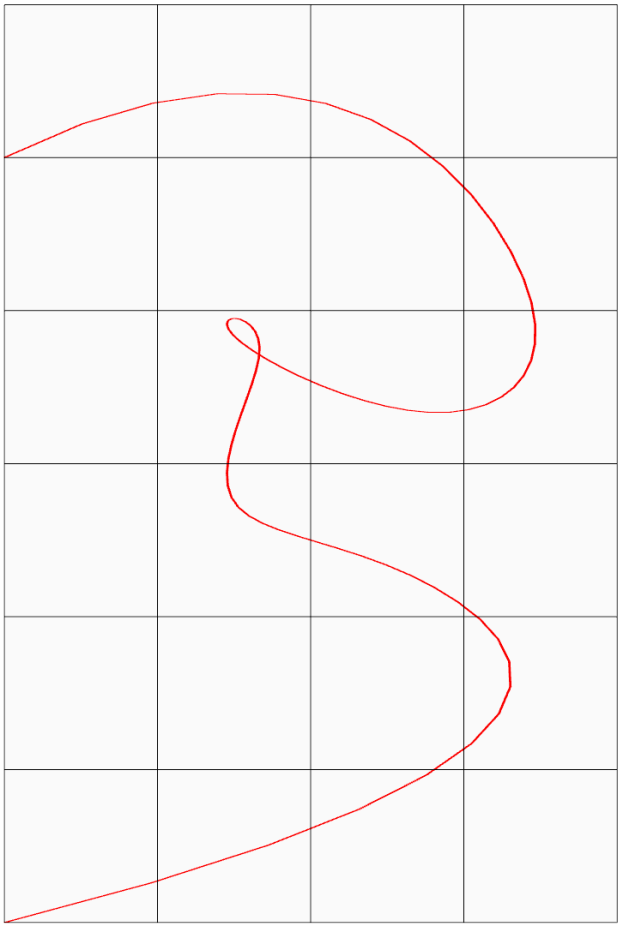
- scalability;
- speed;
- portability.

Geometric models

We need similar techniques in a field called "big geometric data"

- Actual models doesn't scale well with increase of informations
- Ad-hoc solutions, not reusable

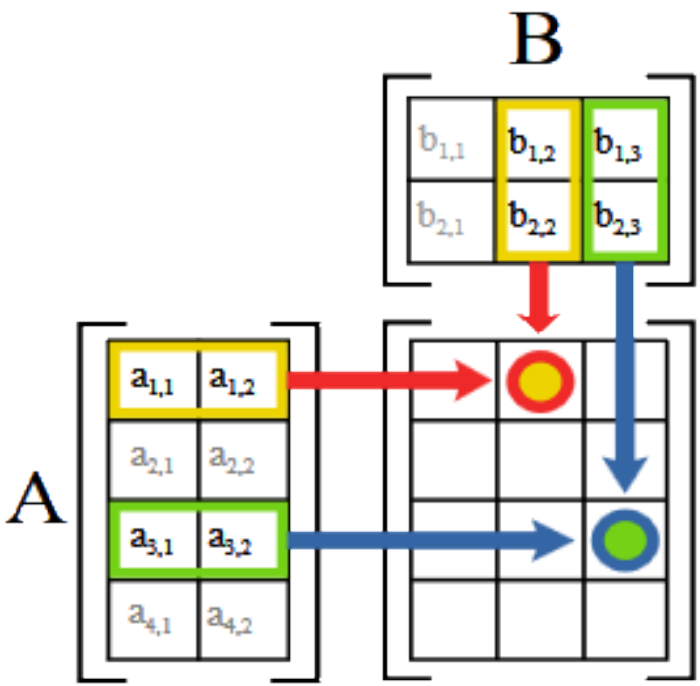
LAR



LAR (Linear Algebric Representation) is a representation scheme (Di Carlo, Paoluzzi, Shapiro) in which the mathematical model to represent topological structures are co-chains complexes:

1. representation through sparse matrices;
2. topological computations done through matrix algebra.

Matrix multiplication



There are different way to multiply two matrices A,B but in the naive mode that everyone should know the complexity is $\mathcal{O}(m * p * n)$ or $\mathcal{O}(n^3)$ with $m = p = n$.

Other algorithms have been created (on paper and tested) that reduce complexity to $\mathcal{O}(n^{2.3729})$ although we can never go more down than $\Omega(n^2)$.

Matrix multiplication and memory layout

Let's say we want to multiply a matrix with a vector with a program on a PC. The pseudo code would be like how we do it on paper, going on a row and accumulating the result in the area reserved the result.

If we think in how we could memorize the informations, array of arrays, the layout in the memory would be helpful for our operations since consecutive elements needed in the operation are consecutive even in memory avoiding jumping in different areas that could slow down operation.

Matrix multiplication and memory layout

What if we want to multiply two matrices?

We'll do it like we do on paper, and iterate for each row of the left-handed matrix with every column of the right-handed matrix.

If we memorize both the matrices by row when we've to do the operations we'll make the PC jump in different memory areas because we need to iterate by column on the right-handed matrix!

Well can we fix this? Sure let's write the second matrix ordered by column!

Matrix multiplication and memory layout

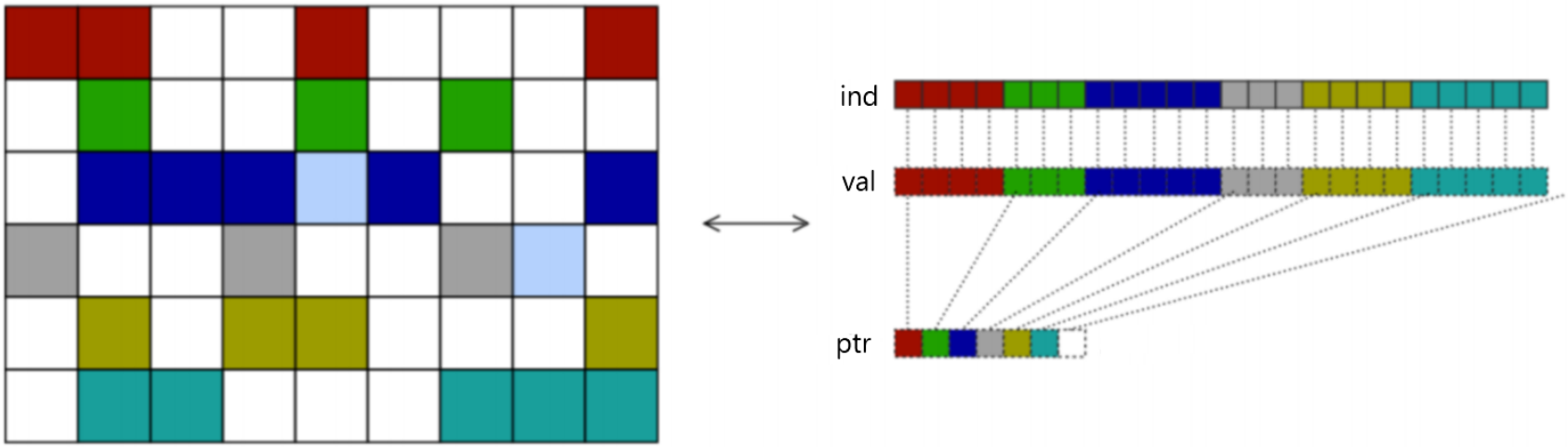
Are we forgetting something?

Well yeah, the matrices you multiply on paper are tiny, tiny, tiny! Usually in geometric models matrices are of billions elements (just think of the vertices cardinality in a complex model).

We need a way to work with big matrices that, in LAR, are sparse!

Sparse matrices computation

With sparse matrices we don't need to store all elements, thus reducing the memory footprint.
This permits to increase the dimension of models (increasing again the memory footprint -_-").
It complicates the multiplication procedures, since we store elements in a different way.



Sparse matrices computation

Still missing anything?

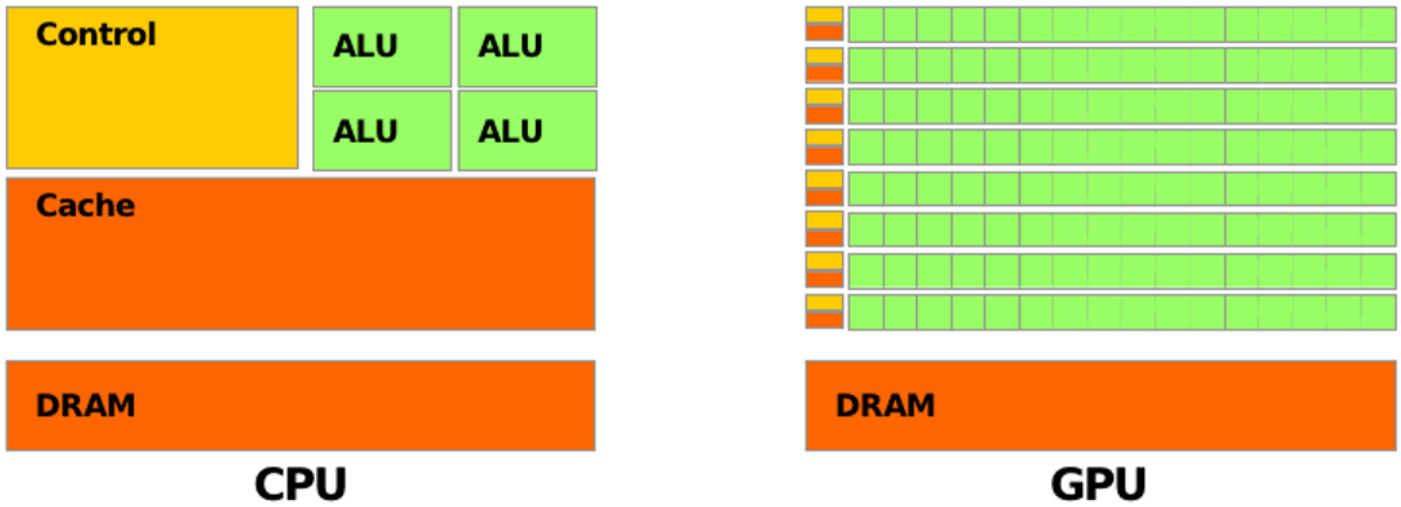
Billions of rows, means a lot of iterations. We'll probably get old waiting for the result!

What can we do? Well we can exploit the thousand computing units that most of the PC nowadays have.

Yes you probably have a thousand computing units on your PC and you never use them!

GPU

Graphic Processing Unit, specialized in graphic processing, are different from a CPU:



- less main memory, but extremely fast
- more computation units
- SIMD paradigm ... (Single Instruction Multiple Data) ...
- ... and SPMD (Single Program Multiple Data)

OpenCL™

C99-like library that can be executed con many platform (CPU, GPU, FPGA, ...).

- Low-level
- High-performance
- Vendor abstract
- Heterogeneous environment

Install

Windows

NVIDIA Install the CUDA toolkit

AMD Install the OpenCL SDK

Intel Install the OpenCL SDK

Linux

NVIDIA Install the CUDA toolkit

AMD Install the OpenCL SDK

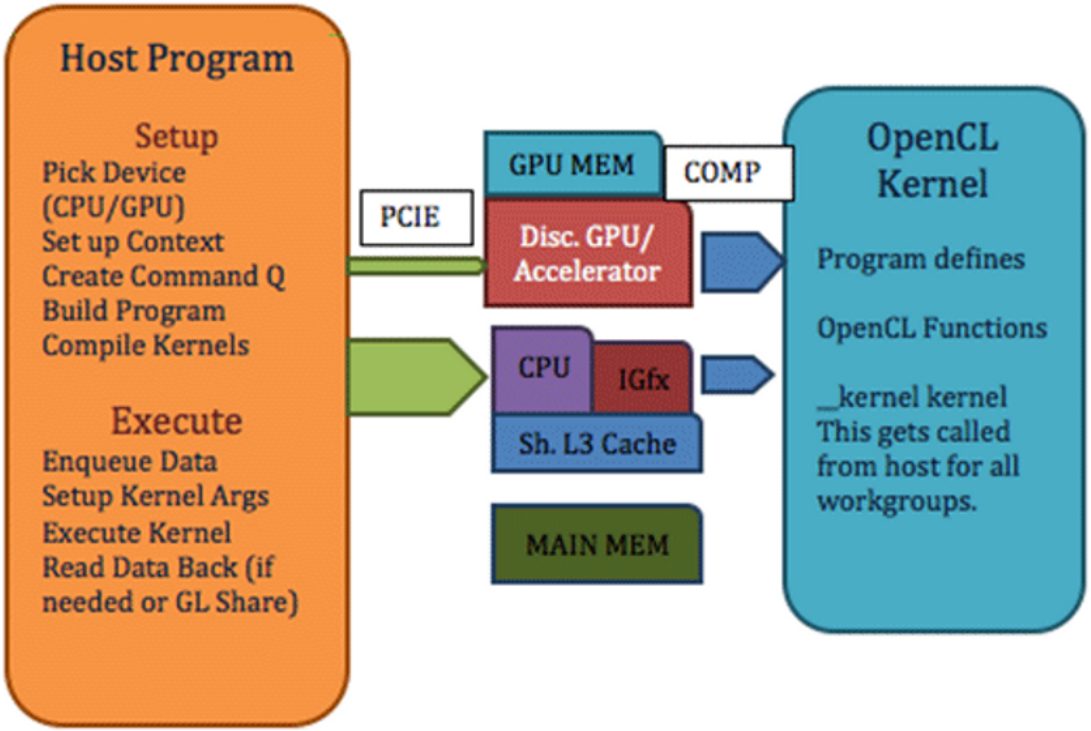
Intel Install the OpenCL SDK

MacOSX

Everything already present. OpenCL was created by Apple :)

How it works (1/2)

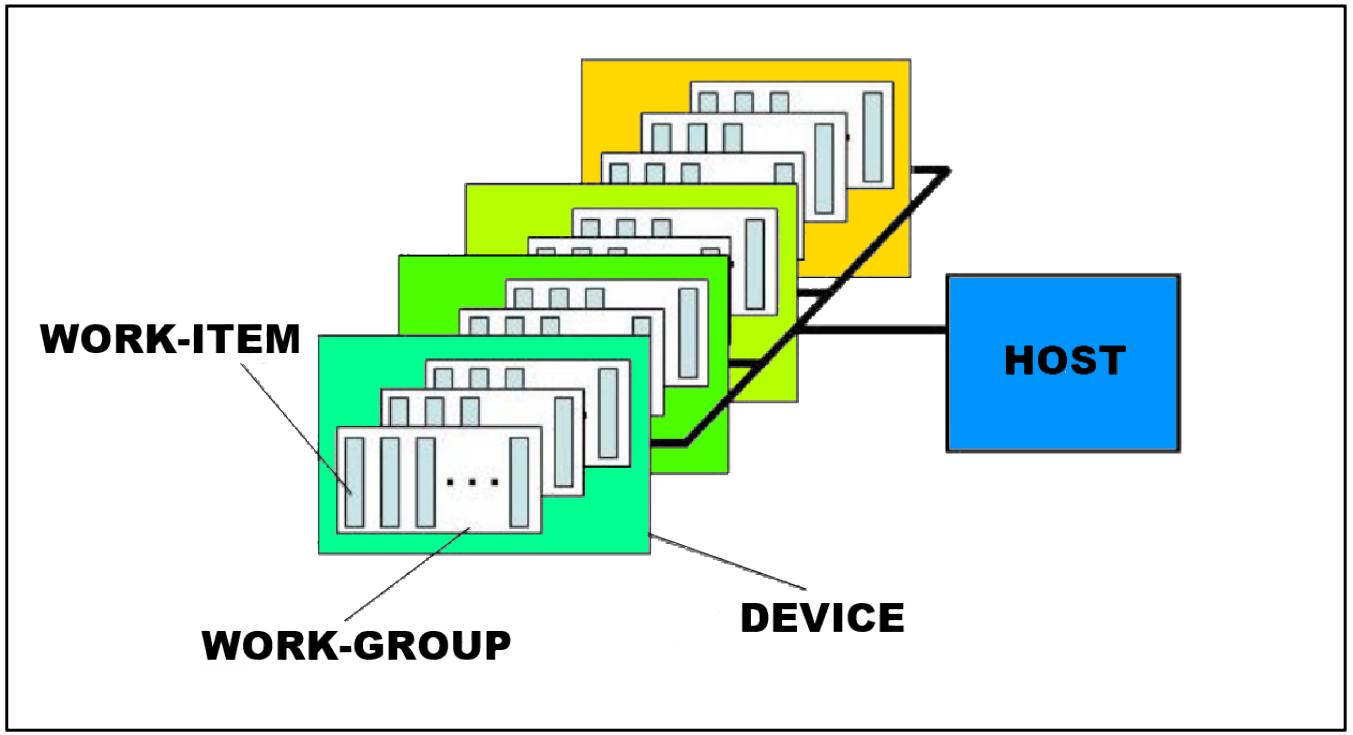
A program on the host coordinates computation and memory allocation.



Another program (kernel) is executed on device cores.

How it works (2/2)

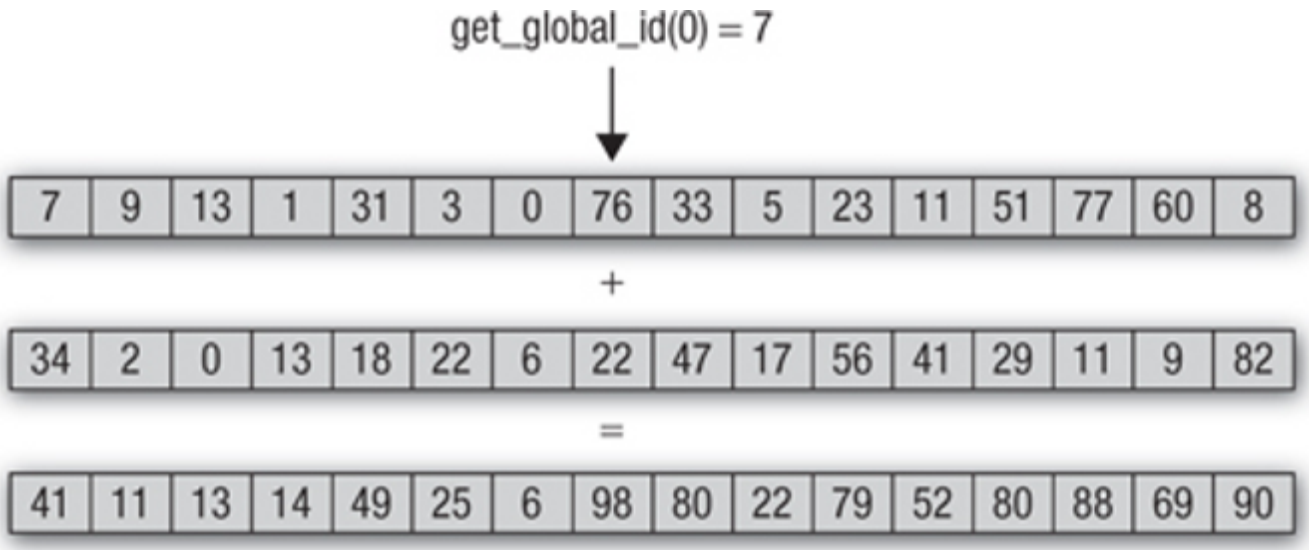
A kernel executing on a core is called a work-item.



Groups of work-item are a work-group.

What is a kernel?

A work-item executes the same instructions changing the flow using an index that is defined by the host.



We execute the same logic but on different memory areas.

3SAC

3SAC kernel multiplies two sparse matrices.

It's the combination of three different kernels:

- K1 : NNZ cardinality calculation of the result
- K2 : Offset vector creation
- K3 : Final computation

3SAC [K1]

K1 kernel calculates NNZ cardinality for each result row and stores them.

Example

$$\begin{bmatrix} 1 & 2 & 0 & 0 \\ 0 & 0 & 9 & 3 \\ 0 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} \checkmark & \times \\ \checkmark & \checkmark \\ \times & \times \end{bmatrix} \rightarrow \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix}$$

$A \times B =$ Different form 0 elements $\rightarrow |\text{NNZ}|$ per row

3SAC [K1]

K1 kernel calculates NNZ cardinality for each result row and stores them.

Example

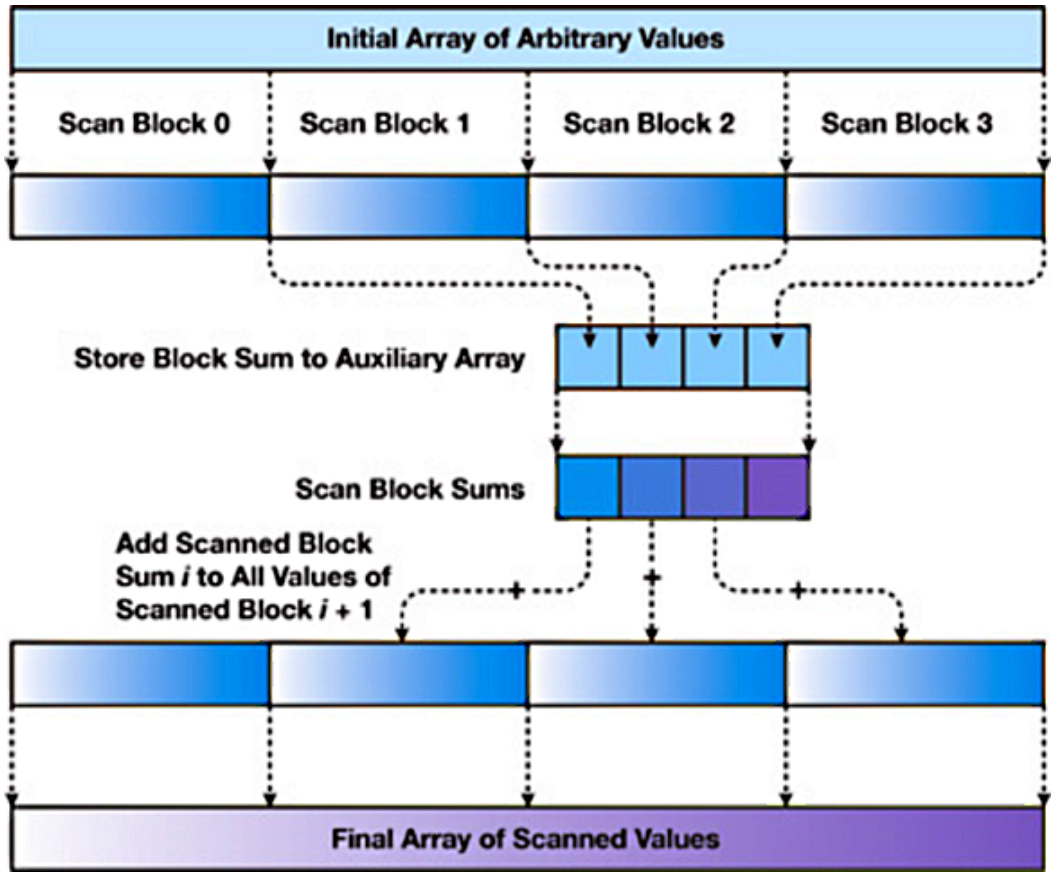
$$\begin{bmatrix} 1 & 2 & 0 & 0 \\ 0 & 0 & 9 & 3 \\ 0 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} \checkmark & \times \\ \checkmark & \checkmark \\ \times & \times \end{bmatrix} \rightarrow \begin{bmatrix} 1 \\ 2 \\ 0 \\ 0 \end{bmatrix}$$

$A \times B =$ Different form 0 elements $\rightarrow |\text{NNZ}|$ per row

The second-last element, at the end, will be copied for the K2 kernel execution.

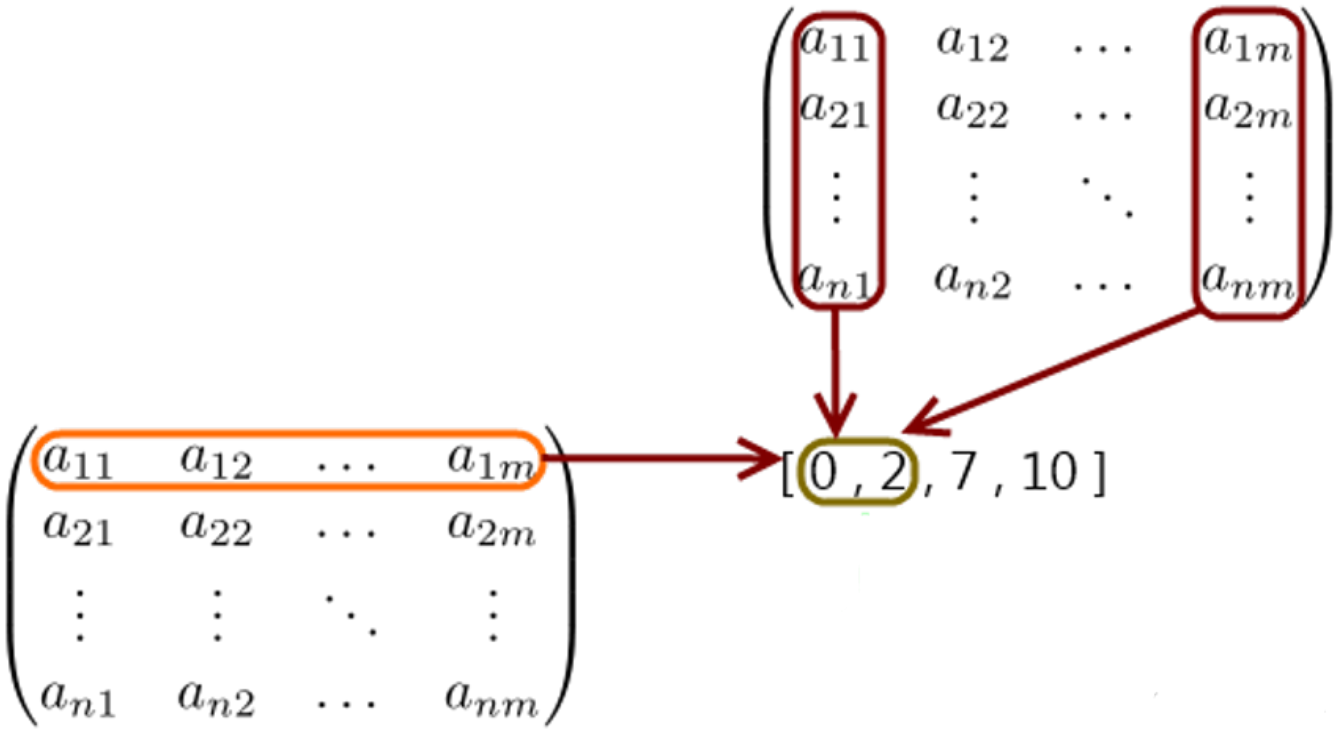
3SAC [K2]

K2 kernel executes a prefix sum on the output vector of K1.



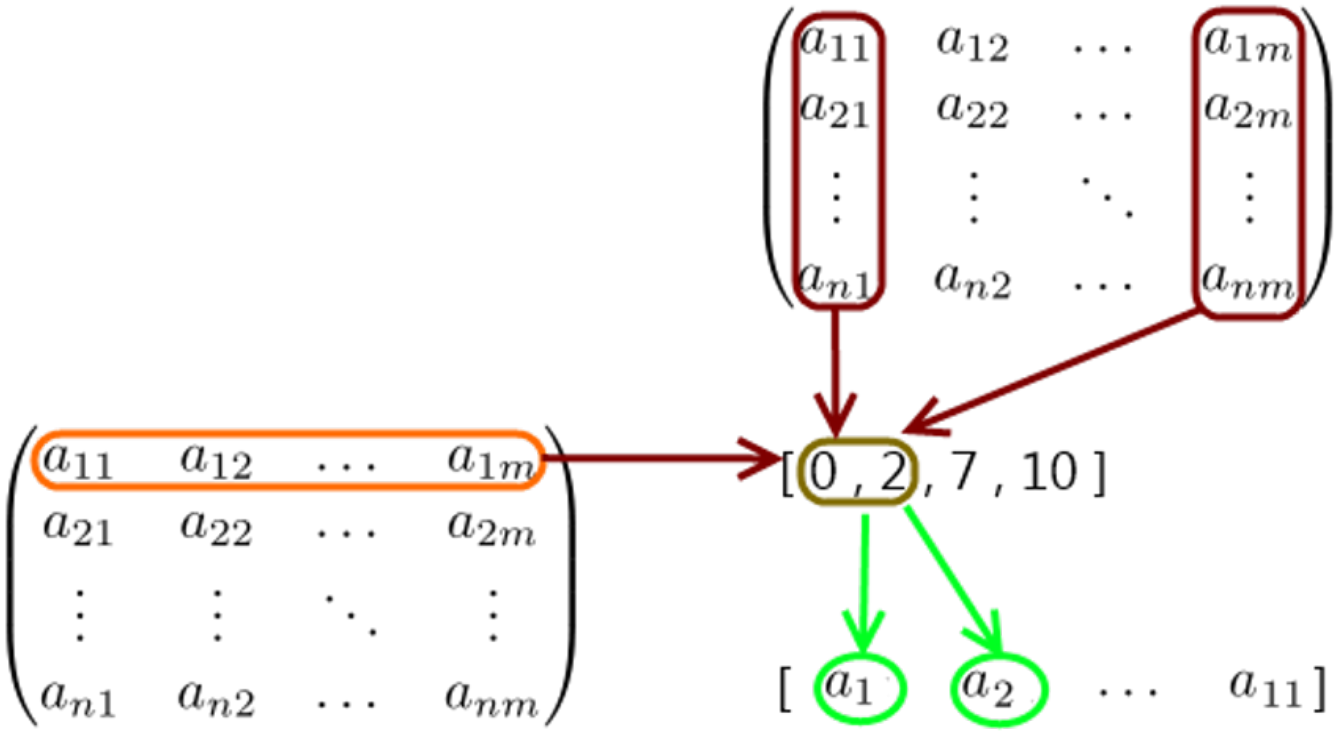
3SAC [K3]

K3 kernel executes the multiplication between the two matrices.



3SAC [K3]

K3 kernel executes the multiplication between the two matrices.



The vector calculated by K2 is used ad the positional index for the memory area in which we'll write the result.

SpMMV (1/3)

SpMMV kernel multiplies a sparse matrix with many dense binary vector.

Many multiplication executed in parallel.

SpMMV (2/3)

The binary vectors are compressed.

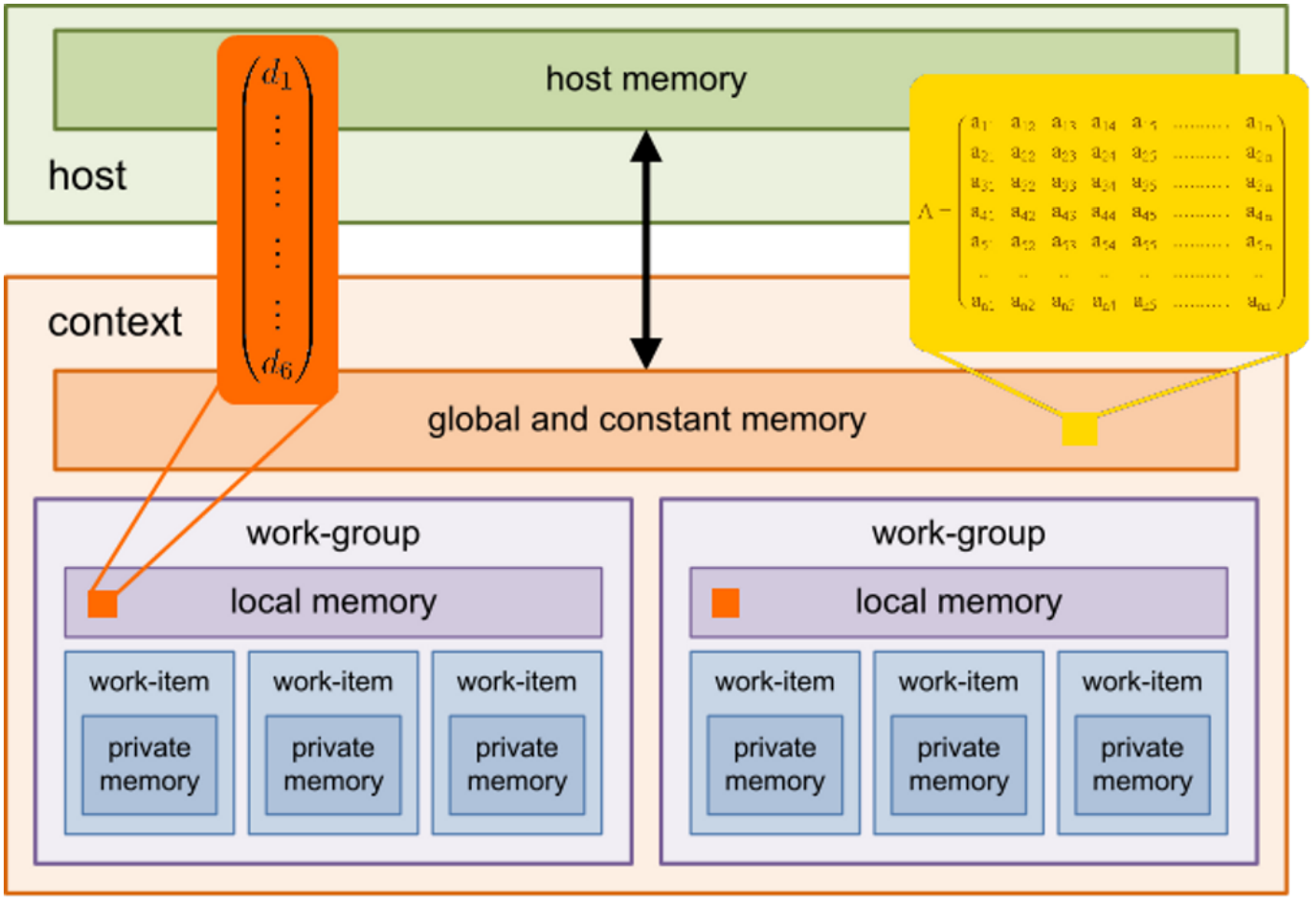
Example

The vector $[1, 1, 0, 1, 0, 1, 0]$ becomes 53

Memory compression, more vectors, more multiplications in parallel.

SpMMV (3/3)

A work-group does the multiplication of the matrix with a vector.



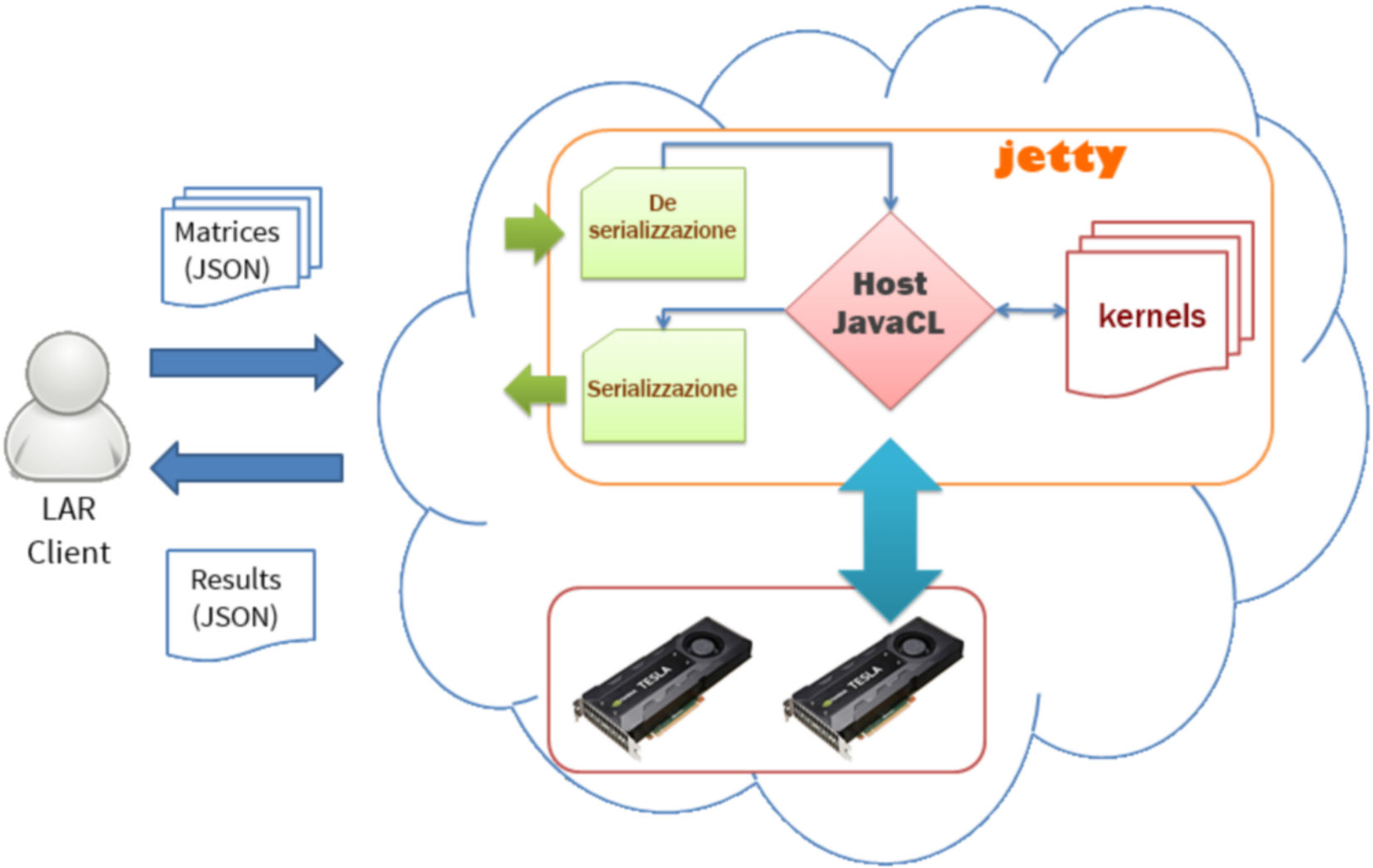
On your PC

Go to GitHub and clone `https://github.com/furio/lar-cc-cl` (this is an accelerated version of LAR, but it doesn't use all of the kernels described here).

Start the acceleration service in the `clservice` subdirectory, using the `start-project.sh` script. You need Java, Maven and OpenCL installed (for a more detailed guide check the `README.md` in the repository directory `lar-cc-cl/clservice`).

Once the service is running you can import the `larcc.py` in your project and enjoy. You'll might need some extra modules for Pyhton like: `simplejson`, `requests`, `termcolor`, `logging`.

jetty e JavaCL



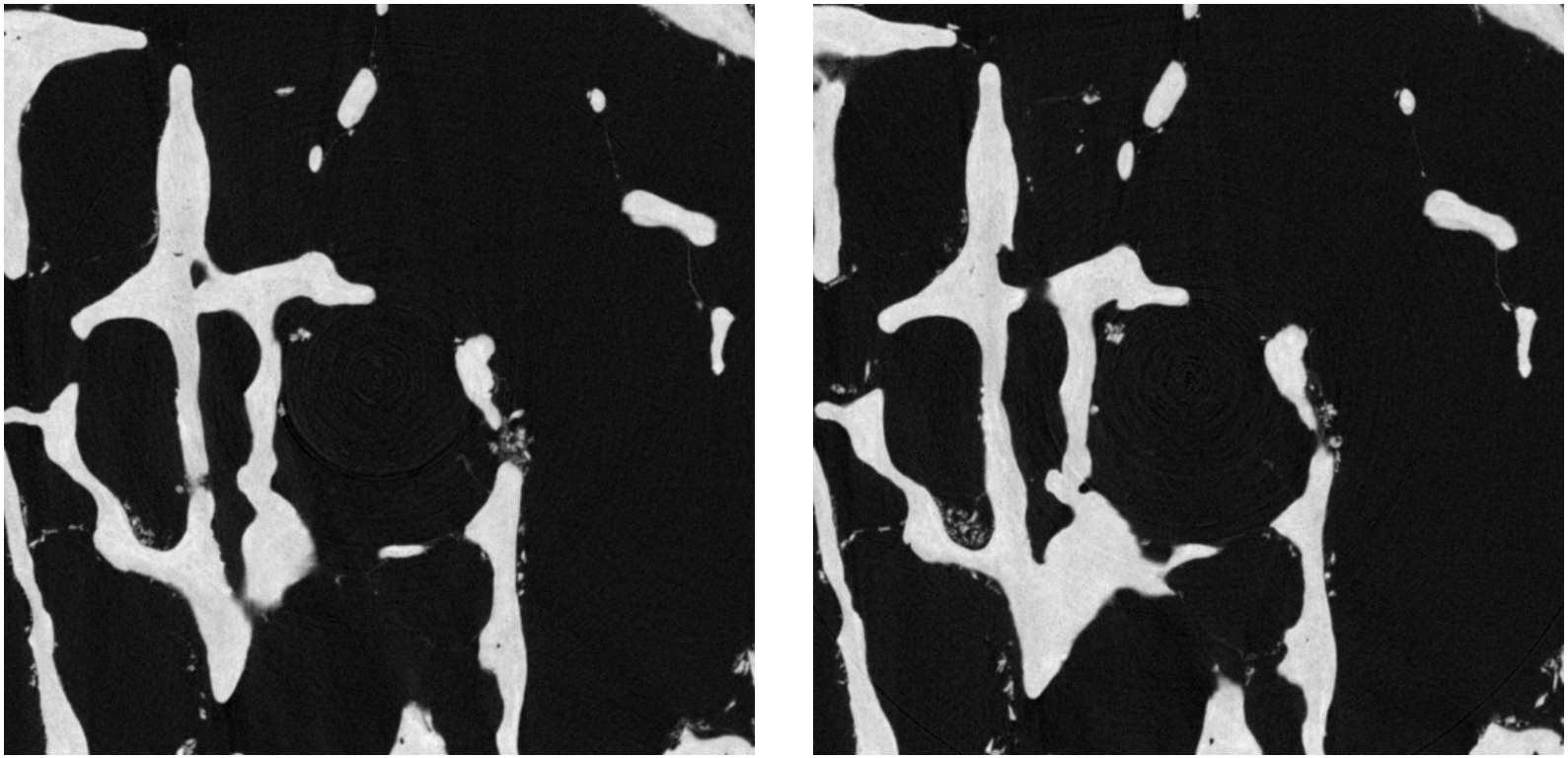
Pipeline application

Extraction of models from medical images. It uses all the kernels described in here (it is the latest updated version).

You can get it from here:
`https://github.com/cvdlab/lar-running-demo`

`README` is in the repository explains how to use it.

Biomedical images



(a) (b)

Figure: MRI of a spongy bone (2 of 1024 slices)

Border extraction

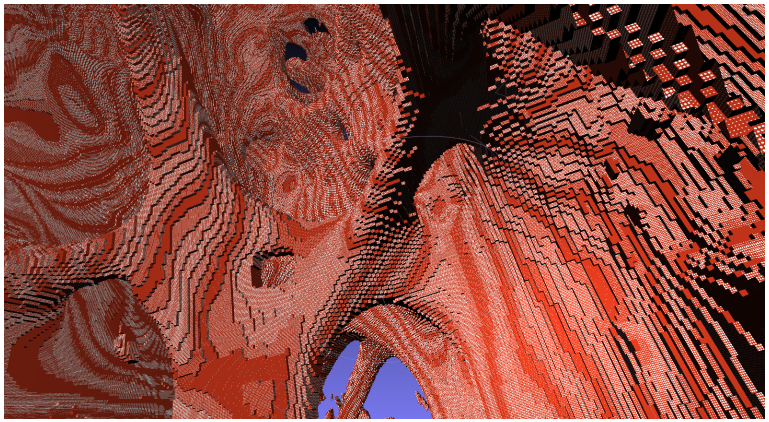
SpMMV kernel execution.

"Host" program computes memory/iterations.

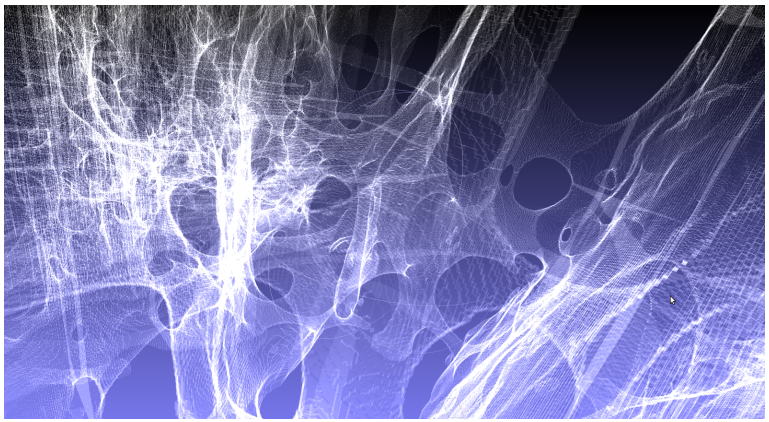
```
RunJob [Java Application] /usr/lib/jvm/java-7-oracle/bin/java
PTR Buffers
CL Buffers
Allocable memory: 268345344
Computable vectors: 335
Will allocate memory: 267571200Kernel source

Create program
Create kernel
Create kernel [0][335][1024]
No local cache
Memory sync
WgSize: 64320 (335) - LocalSize: 192
EnqueueND Range - wgSize+locSize
Copying back memory from GPU
Kernel execution in: 1369 millis. Return code: 0
Copying back memory to struct
Release kernel
Create kernel [335][335][689]
Add in CPU memory
Copying back memory from GPU
No local cache
WgSize: 64320 (335) - LocalSize: 192
EnqueueND Range - wgSize+locSize
Kernel execution in: 1341 millis. Return code: 0
Copying back memory to struct
Release kernel
Add in CPU memory
Create kernel [670][335][354]
No local cache
WgSize: 64320 (335) - LocalSize: 192
EnqueueND Range - wgSize+locSize
Copying back memory from GPU
```

Visualization



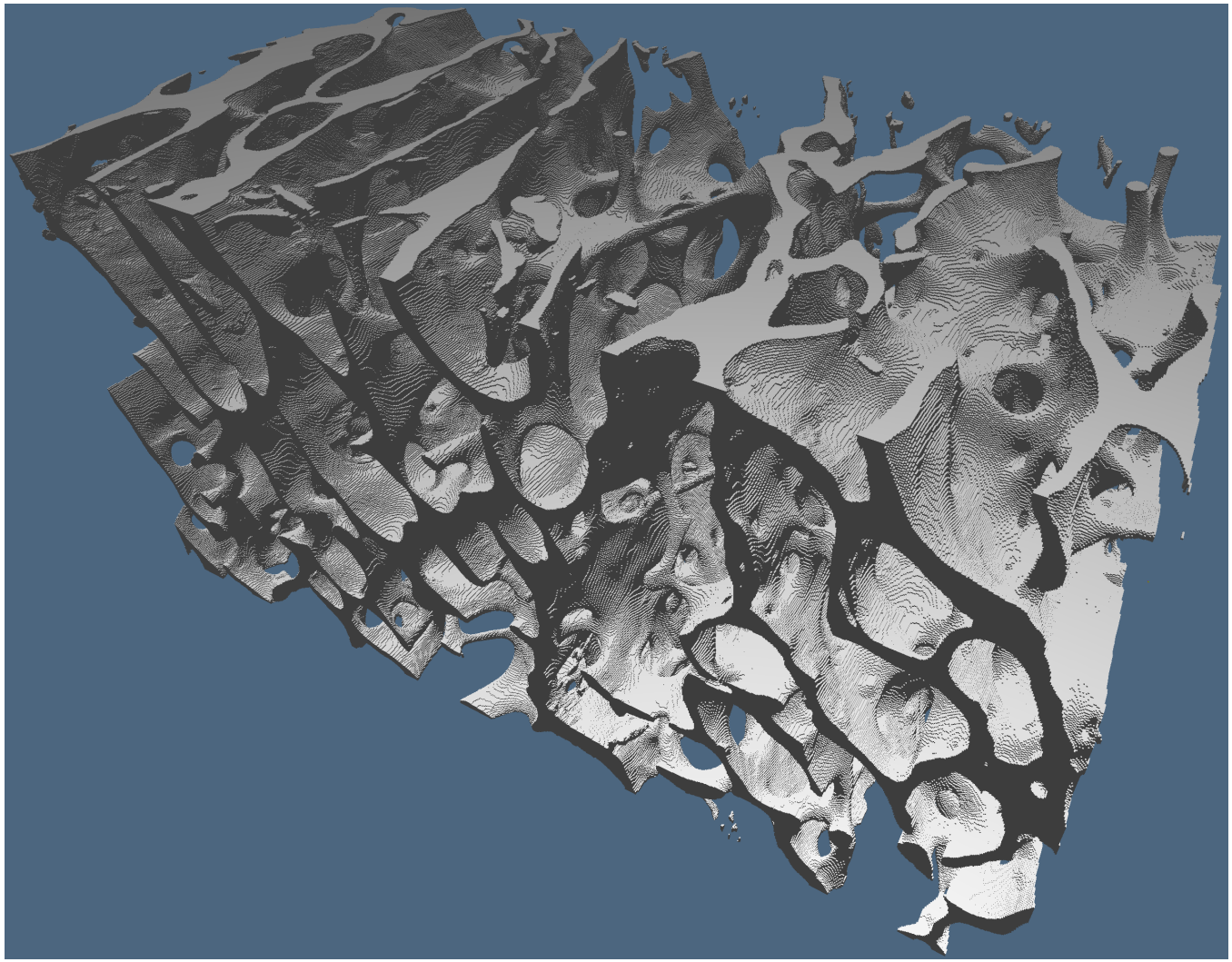
(a)



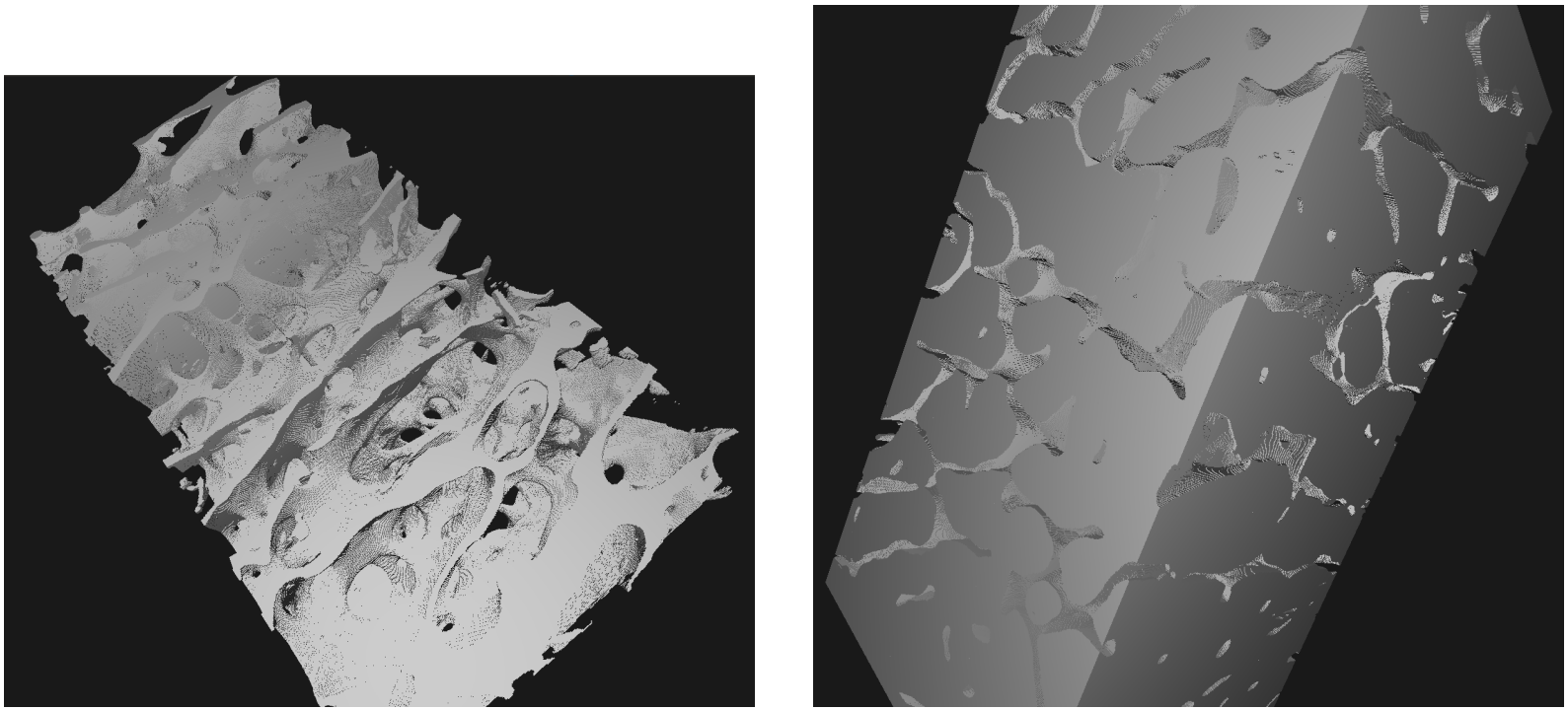
(b)

Figure: MeshLab rendering with shaders: (a) z stripe (b) x-ray.

Visualization



Visualization

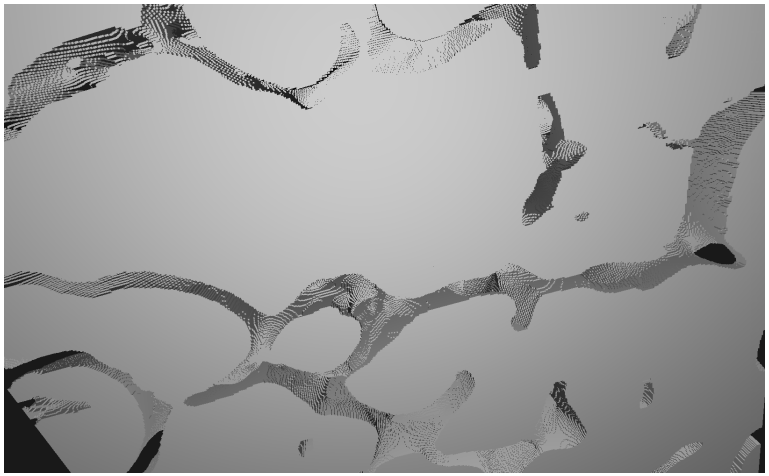


(a) (b)

Figure: Manta raytrace: (*a*) spongy bone (*12065847* vertices and *24472222* faces) (*b*): complement (*21344953* vertices and *61975236* faces).

Visualization

(a)



(b)

Figure: Manta raytrace of a detail: (a) spongy bone (b) complement.

Interested?

Want to play with cutting edge technology?
Algorithm programming on a GPU could interest you?
Drop us (teacher mailbox :)) a line for a project or a thesis!

paoluzzi@dia.uniroma3.it

Practical
demo
now!