Peer group and Fuzzy Metric to Remove Noise in Images using Heterogeneous Computing.

Ma. Guadalupe Sánchez -ITCG ,
Vicente Vidal -UPV ,
Jordi Bataller – UPV.

CONTENT

- Introduction
- A parallel Denoising Algorithm.
- Multi-GPU and multi-core CPU Implementation.
- Experimental Study.
- Concluding remarks.

INTRODUCTION

Image Denoising → Image processing → Affect the performance and accuracy of some processes → Real-time processing → **Optimal implementation**

Noise removal is a very important task in applications related to:

- video communication
- biomedical science
- image post-proccesing
- surveillance

Malfunction

during the process of

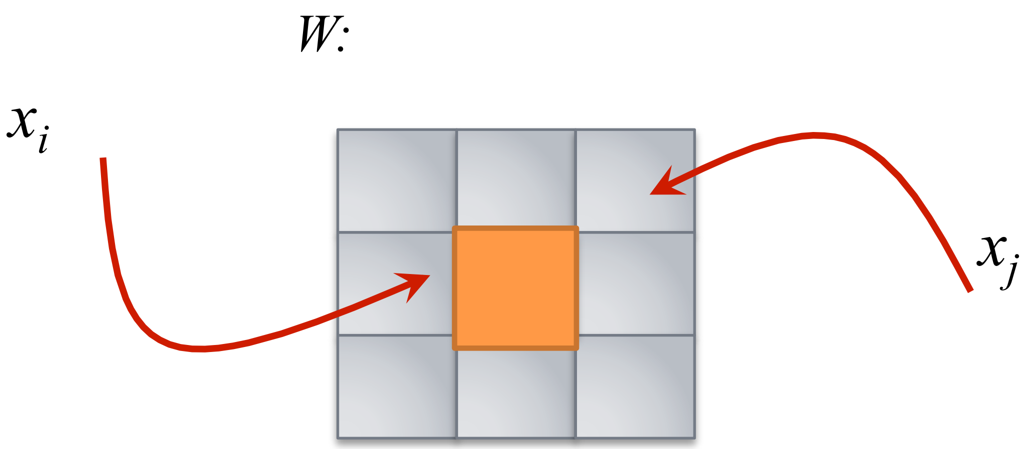
- image formation,
- storage or
- transmission

Impulsive noise

The widespread model of
Impulsive noise

- is the "**Salt and Pepper**" model, or **fixed-value** noise.
- It considers a pixel is wrong, when its value is an **extreme** value within the signal range.
- We assume this model in our work.

Many of the filters to remove impulsive noise in digital images have been designed, some of them are based on the concept of ”peer group”.



Is the set of its neighbors that are similar to it, according to an chosen metric. (fuzzy measure, euclidea measure)

These investigations have shown good results in

Quality

but they don’t seem to be appropriate for <u>real-time processing.</u>

GPU

- Are currently a very popular platform for develop parallel applications, considering price and speed.

- Are another widely used tools for parallel applications.

Our parallel version of filter is based on

- peer group and
- fuzzy metric

<u>Good quality</u> results while trying to <u>improve its performance</u>, making them usable for <u>real-time processing</u>.

The implementation uses:

GPU
.
.
.
GPU

core
.
.
.
core

CPU

When we use GPU, the assignation of the pixels on shared memory or texture memory, is with the purpose of <u>take the most advantage of the hardware.</u>

Shared memory

Texture memory

CONTENT

- Introduction
- A parallel Denoising Algorithm.
- Multi-GPU and multi-core CPU Implementations.
- Experimental Study
- Concluding remarks

A parallel Denoising Algorithm

- Our parallel algorithm uses the peer group and fuzzy metric.
- The fuzzy metric between pixels x_i and x_j in the color image is given by:

$$M(x_i, x_j) = \prod_{l=1}^{3} \frac{min\{x_i(l), x_j(l)\} + k}{max\{x_i(l), x_j(l)\} + k}.$$

- The peer group of a pixel x_i is comprised by the pixels of a window centered in x_i whose distance from x_i exceeds d:

$$P(x_i, d) = \{x_j \in W \quad : M(x_i, x_j) \geq d\}.$$

- The denoising algorithm is as follows:

It has two main steps:

Detection → pixel x_i is declared as corrupted :

$$\text{if } \#P(x_i, d) < (m + 1),$$

where m is the voting threashold.

Filtering → If x_i was previously marked as corrupted, it is replaced using the AMF.
The old value is replaced by the new value.

CONTENT

- Introduction
- A parallel Denoising Algorithm.
- Multi-GPU and multi-core CPU Implementations.
- Experimental Study
- Concluding remarks

Multi-GPU and multi-core CPU Implementations.

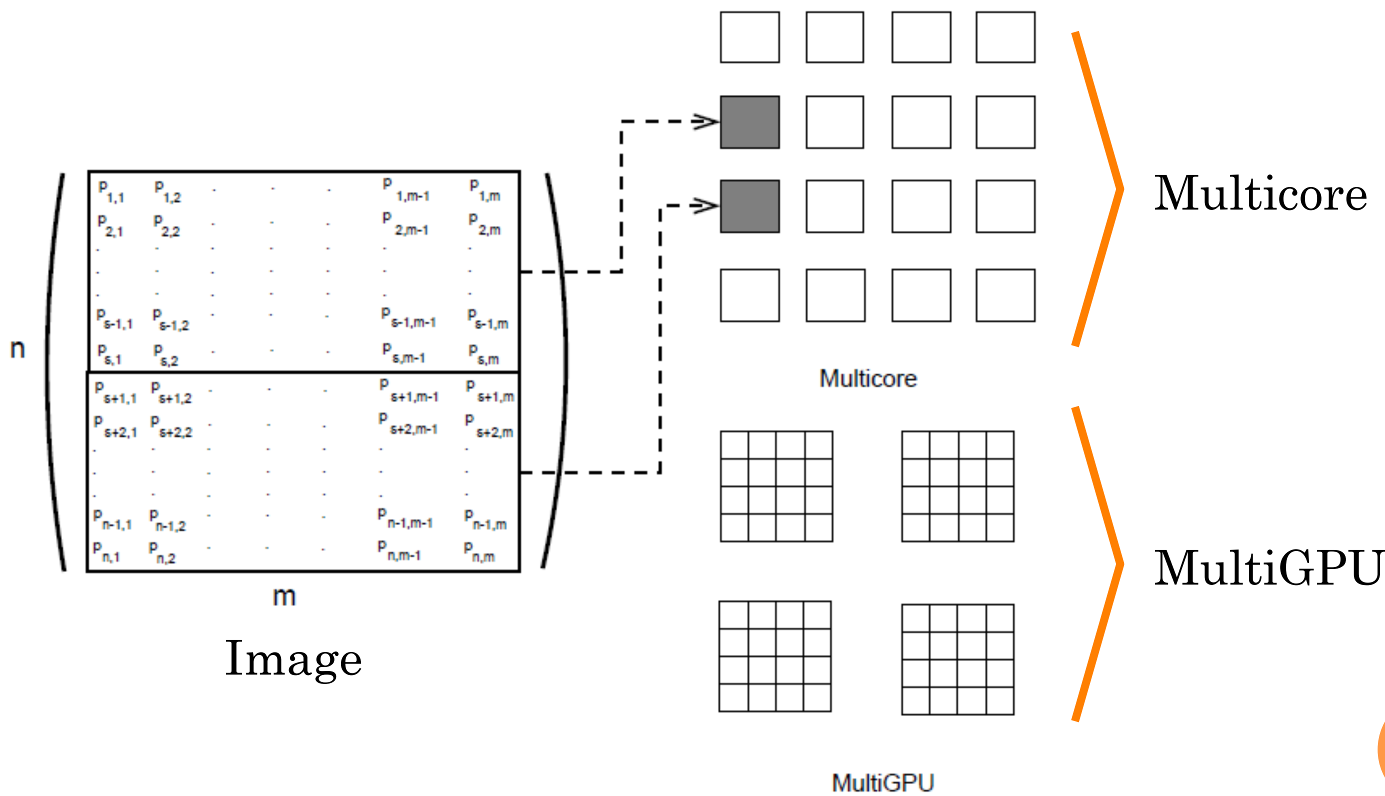
- We have developed three implementations for two parallel architectures.

OpenMP for multi-core CPU

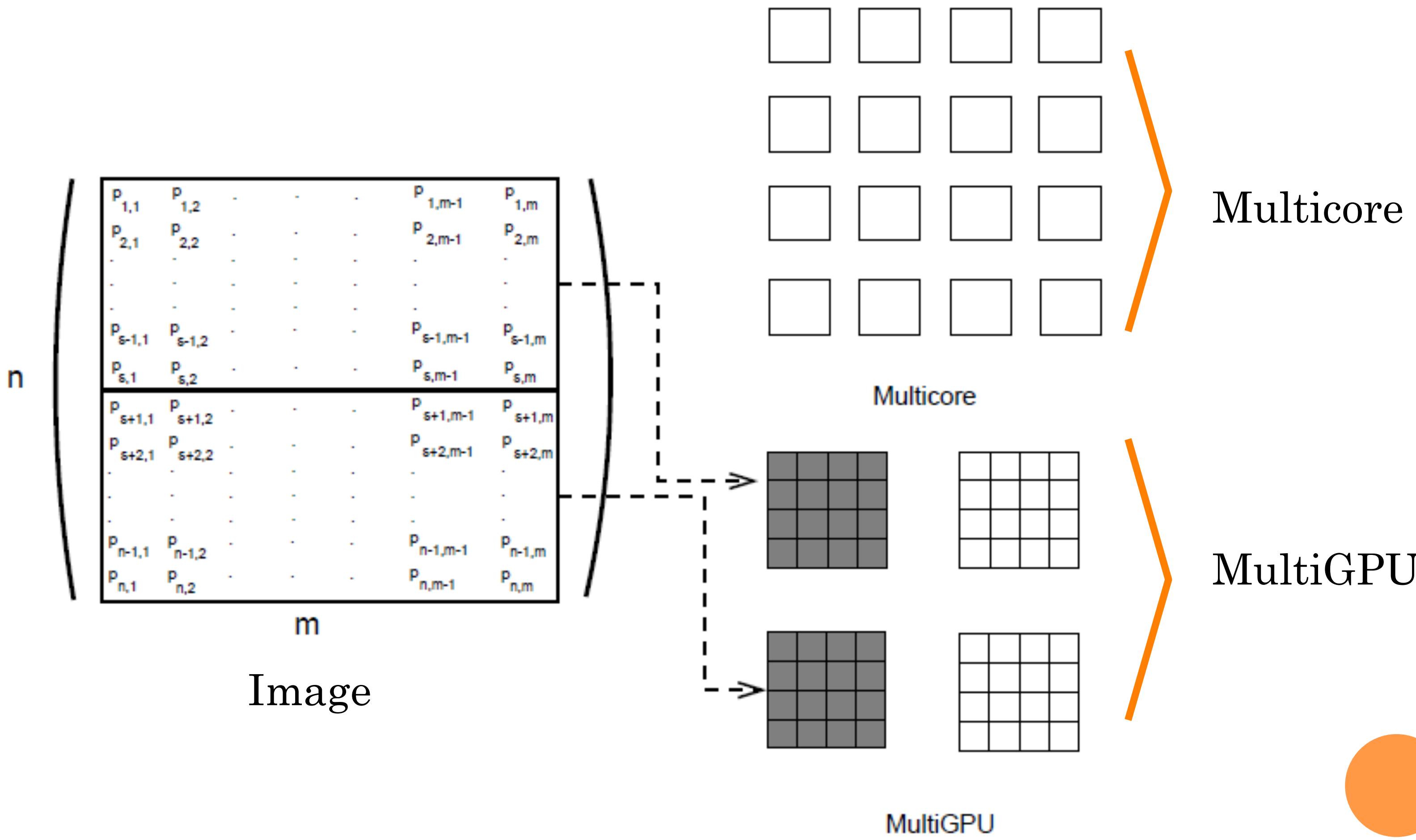
CUDA for multi-GPU

CPU and GPU in combination

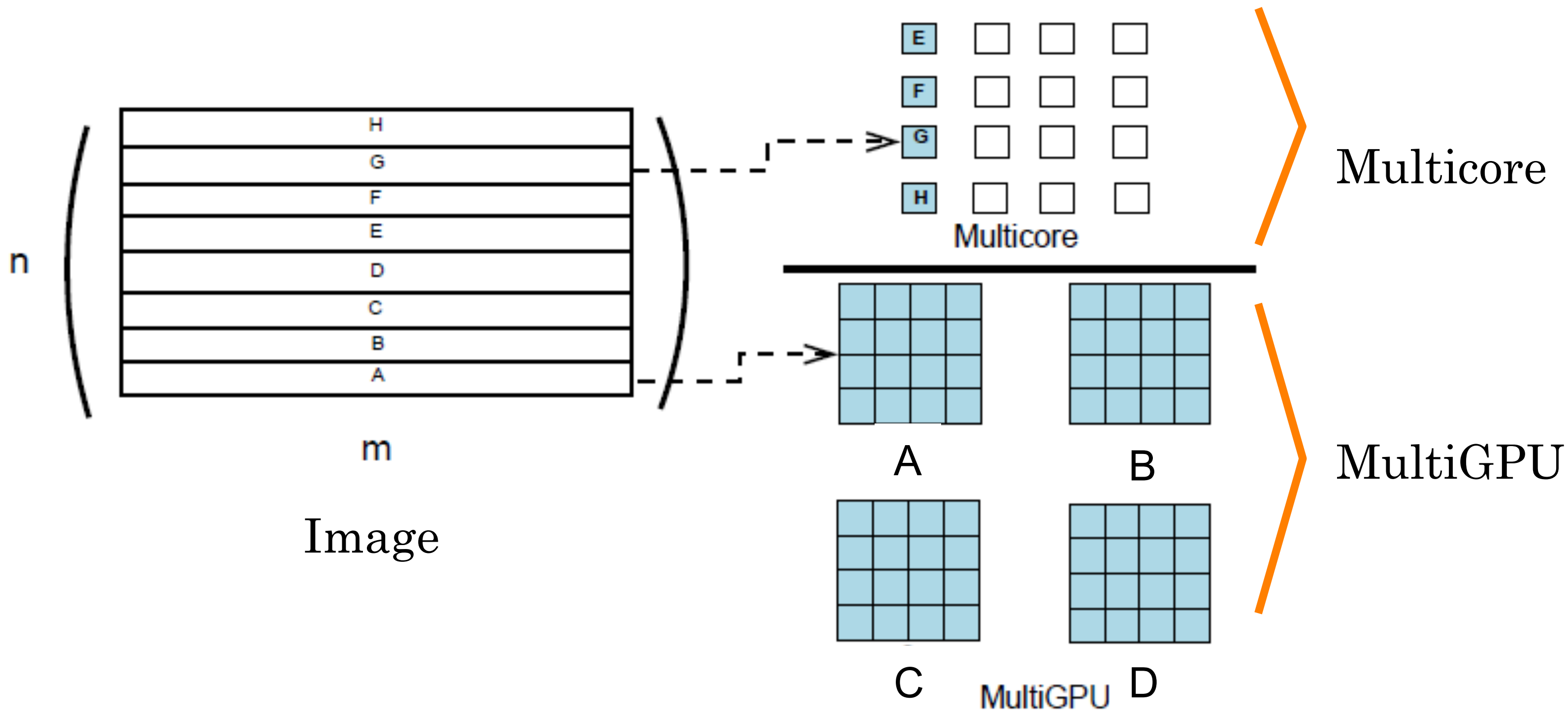
- When only the CPU is used, some pixels are assigned to one core, and the rest to a second core, leaving the remaining cores (and the GPUs) idle.



- When only the GPU is used, the image is divided into two parts, each one will be processed by a different GPU.



- Using both, Cores in CPU and GPU, we made a partition of the image in eight horizontal blocks, to be processed by a combination of GPUs and cores in the CPU.



Main memory (RAM) ← GPU memory

Tasks to be executed by a GPU are coded into functions called: **kernels**

Synchronization

- Step 1. Detection → kernel
- Step 2. Filtering → kernel

won't start until the first ends.

- Therefore, before the actual processing st e CPU control program must select
- which GPU devices will be used,
- copy data to them,
- launch the kernels and
- recover the results.

GPU

Memories with different features (size, speed, access).

For it, is necessary to consider:

- where to put the data and
- how to access them,

once they are on the GPU.

- Shared memory

Global memory GPU → shared memory block

is available only to the set of threads being executed by the same multi-processor in a GPU.

- Texture memory

Global memory GPU ← Thread. (but read-only mode)

CONTENT

- Introduction
- A parallel Denoising Algorithm.
- Multi-GPU and multi-core CPU Implementations.
- Experimental Study
- Concluding remarks

Experimental Study

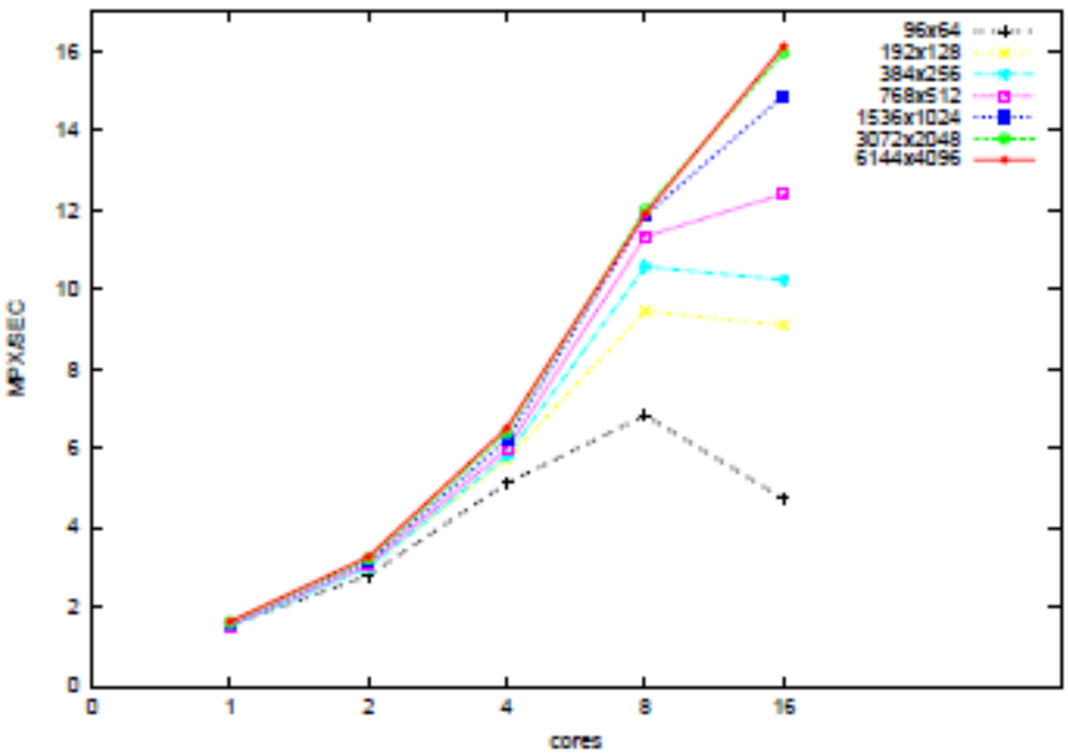
- The first test was to compare the use of shared memory versus texture memory, using 1, 2 and 4 GPUs.

Image size	Texture memory	Shared memory	GPU
6144 x 4096	20.66	17.34	1
6144 x 4096	30.57	29.49	2
6144 x 4096	47.12	41.37	4

In the following tests, the texture memory is always used.

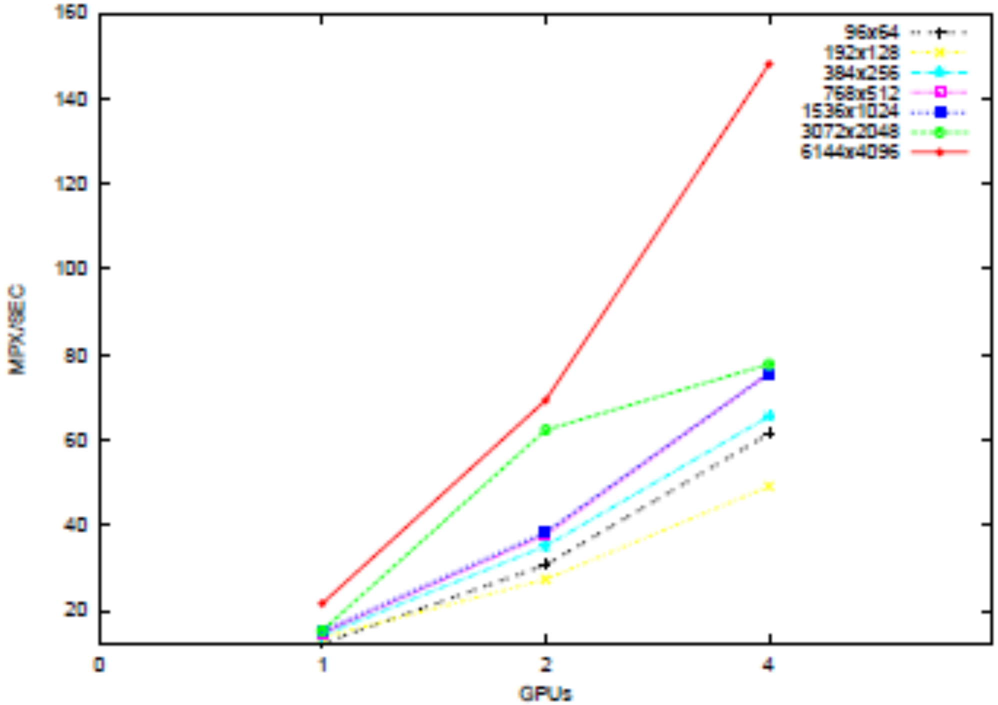
CPU	GPU
Intel Quad-Core Xeon 2 x 2.2 GHz.	GeFroce GT 120, 512 MB. 4 Multiprocesors.

- Performance results using CPU with 16 cores and different image sizes.



For sizes larger than 384x256, the best results are obtained when all the 16 cores are used.
Otherwise, it is better to use 8 cores only.

Using several GPUs….

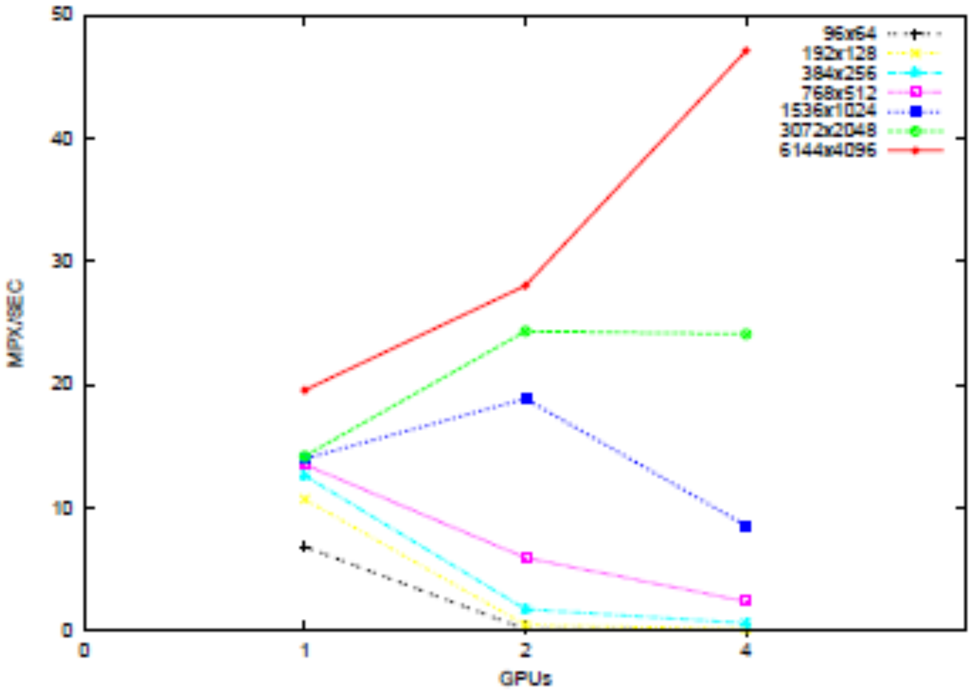


Results using only the GPU, this is,

without transfers RAM-GPU memory.

It is clear that better outcomes occur when all the GPUs are used (4 in our case).

Results including data transfer between RAM & CPU memory.



b

If the image is smaller than 1536x1024 pixels, it's better to use a single GPU because the time used in transfers is not compensated by the use of more GPUs.

Otherwise, for sizes larger than 1536x1024, using more GPUs improves the performance,

- Results when Cores of the CPUs and the GPUs were simultaneously used.

Image size	Multicore	MultiGPU	Multicore and MultiGPU
96x64	6.82	6.83	8.37
192x128	9.45	10.69	15.54
384x256	10.57	12.60	20.01
768x512	12.40	13.51	37.13
1536x1024	14.88	18.86	42.39
3072x2048	15.96	24.34	43.78
3072x2048	15.96	24.10	69.91
6144x4096	16.12	47.12	68.46

In all cases it's best use CPU-GPU in combination.

Results using CPU-GPU in combination. Table showing how many cores and GPUs are used for each image size, and which part of the images is assigned to GPUs.

image size	GPUs	Cores	Size on GPU
96x64	1	16	1/4
192x128	1	11	3/8
384x256	1	7	1/2
768x512	1	9	3/4
1536x1024	2	9	3/4
3072x2048	2	9	3/4
3072x2048	4	11	7/8
6144x4096	4	7	7/8

- for image sizes less than 768x512, it's better to use one GPU.
- for image sizes between 768x512 and 3072x2048 it's better to use 2 GPUs.
- for image sizes greater than 6144x4096 it's better to use 4 GPUs.

The part of the images assigned to GPU varies according to the image size

- It seems that, for greater sizes, more work is to be done on the GPUs.

CONTENT

- Introduction
- A parallel Denoising Algorithm.
- Multi-GPU and multi-core CPU Implementations.
- Experimental Study
- Concluding remarks

Concluding remarks

- Our implementation was developed to be executed either on multi-core CPU, on several GPUs, or using the CPU along with the GPUs.
- The results shown that this latter option (CPU +GPUs) gives the best performance.
- On the way, the use of texture memory is better than the use of shared memory.

- The final conclusion is that implementing image denoising algorithms to be run on multi-core CPUs and GPUs are very advisable. This opens the door to use algorithms for real-time processing.
- In future works, we plan to test our programs on the last generation of GPU cards, and to adress other common problems on images, such as edge detection.

Thanks for your atention!!!!