
# TUTORIAL: OPENCV & CUDA

**Presented by:**

Ramon Aranda, Francisco Hernandez-Lopez, Francisco Madrigal,

{arac, fcoj23, pacomd}@cimat.mx

Centro de Investigación en Matemáticas, A.C.

Guanajuato, Gto. October 2014

# OUTLINE

- OpenCV & Cuda (Brief Introduction)................ (15 min)
- Image processing in OpenCV ........................ (7.5 min)
- Memory allocation in the GPU....................... (7.5 min)
- Memory passing between OpenCV and CUDA........ (10 min)
- Operation on parallel (GPU management) ............(5 min)
- Operations on GPU: First Examples
  - Addition of Vectors/Matrices.............................. (20 min)
  - Considerations ..............................................(10 min)

# OUTLINE

- Parallel Image processing
  - Compose images ..................................................(20 min)
  - Gradient magnitude...........................................(20 min)
  - Image filtering..................................................(35 min)
  - Corner detector................................................. (20 min)
  - Diffusion image................................................(25 min)
- Native Functions of OpenCV that use CUDA: gpu::mat..(15min)
- Parallel Image processing using multiple GPUs: Examples(20min)
- Conclusions: Potential applications........................(10 min)

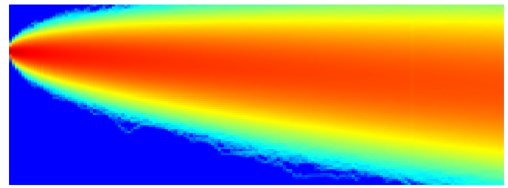
# MOTIVATION:
## COMMON TASKS ON IMAGE PROCESSING

- Image filtering
- Stereo Matching
- Morphology
- HOG
- Segmentation
- Etc.

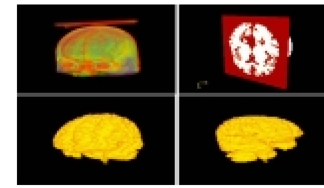
- All **Highly Parallelizable**

# MOTIVATION: OPENCV & CUDA

- You can solve problems:
  - Finance
  - Image processing and Video
  - Linear Algebra, optimization problems
  - Physics, Chemistry, Biology
  - Etc.…

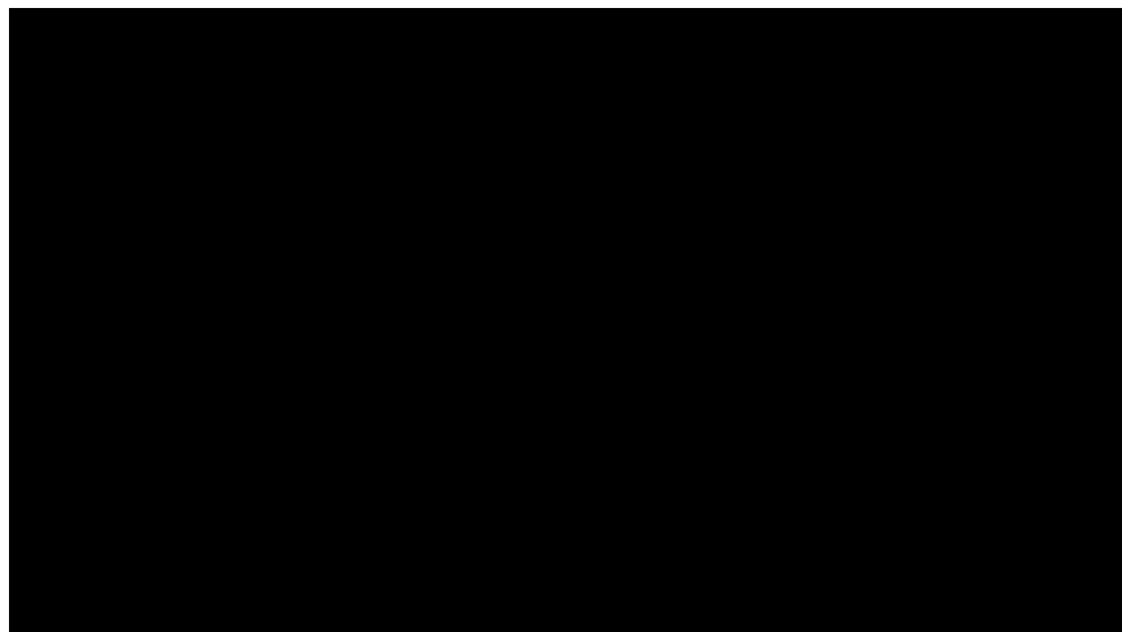


Finite element methods



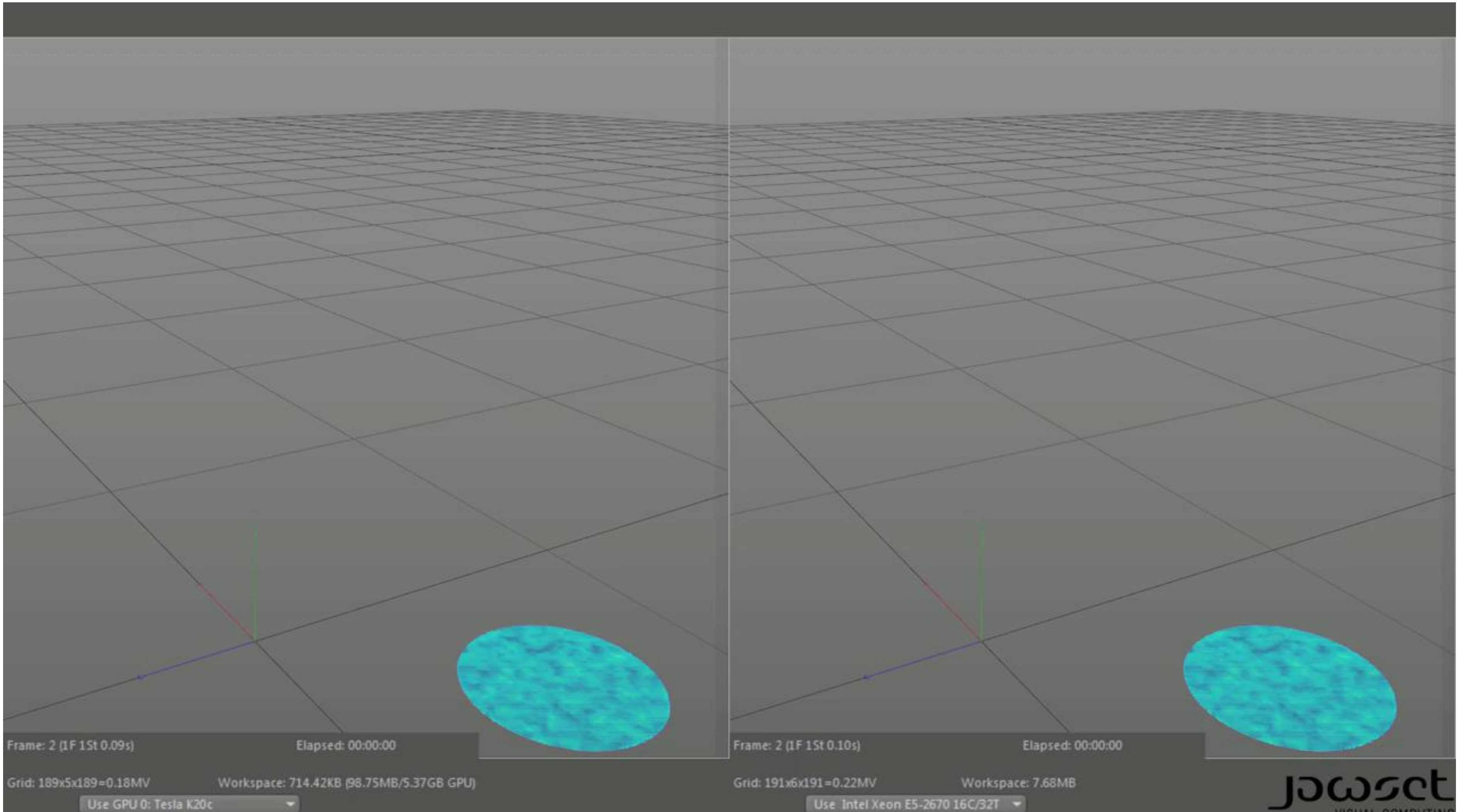
Medial Image Proccessing

Object detection



Protein Simulation

# MOTIVATION: GPU (USING CUDA) VS MULTI-CORE CPU



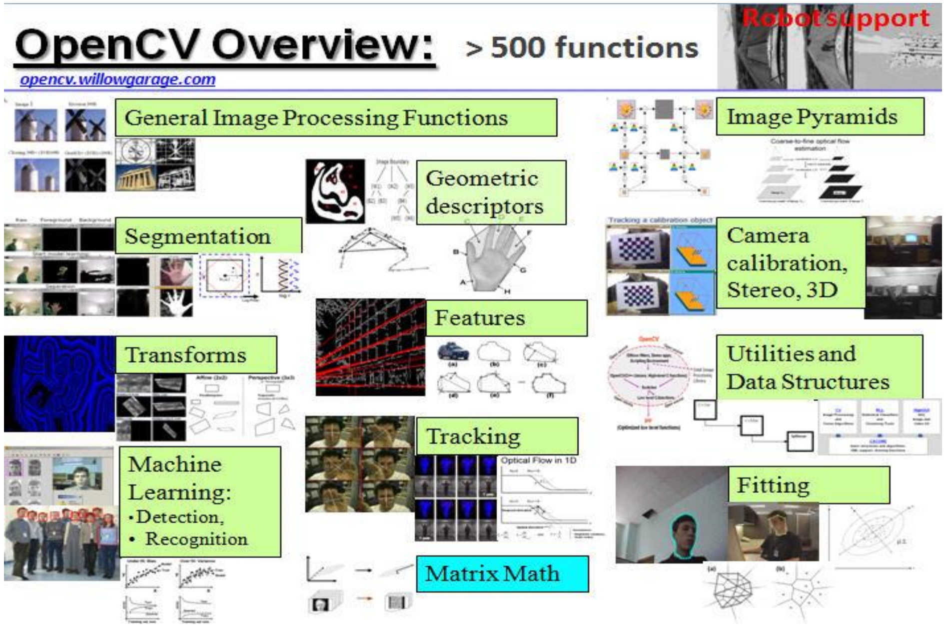
# INTRODUCTION

# INTRODUCTION:
# WHAT IS OPENCV?

- Library of algorithms released under BSD license.
- Interfaces with C++, C, Python and JAVA.
- Can be compiled on Windows, Linux, Android and Mac.
- Has more than 2500 optimized algorithms.
- Support by a big community of users and developers.
- Multiple uses like visual inspection, robotic, etc.

# INTRODUCTION:
# HOW TO INSTALL OPENCV

- http://www.opencv.org/
- http://www.cmake.org/



# INTRODUCTION:
# OPENCV MODULES


General Image Processing


Segmentation


Machine Learning, Detection

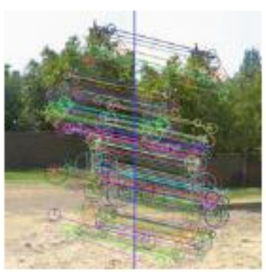

Image Pyramids


Transforms


Fitting

**Video, Stereo, and 3D**


Camera Calibration


Features


Depth Maps


Optical Flow


Inpainting


Tracking

**Source: www.itseez.com**

# INTRODUCTION:
# OPENCV MODULES

- **Contrib:** Miscellaneous contributions
- **Legacy:** Deprecated code
- **Nonfree**:  Algorithms with copyright.
- GPU: GPU functions (**Can use with another CUDA libs**)

# INTRODUCTION:
# PARALLEL COMPUTING

- Running more than one calculation at the same time or "in parallel", using more than one processor.

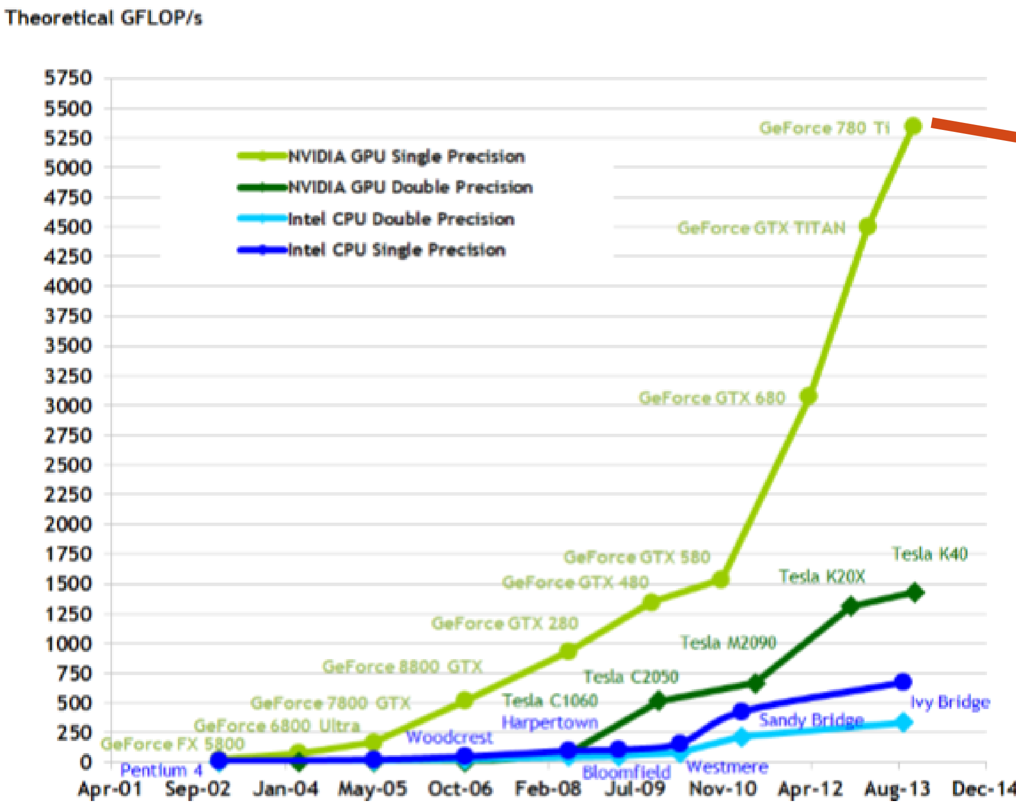

OpenMP

OpenMPI

Cg,
CUDA,
OpenCL

# INTRODUCTION: GPU

- Flexible and powerful Processor
- Handles accuracy of (32/64)-bit in floating point
- Programmed using high level languages
- Offers lots of GFLOPS

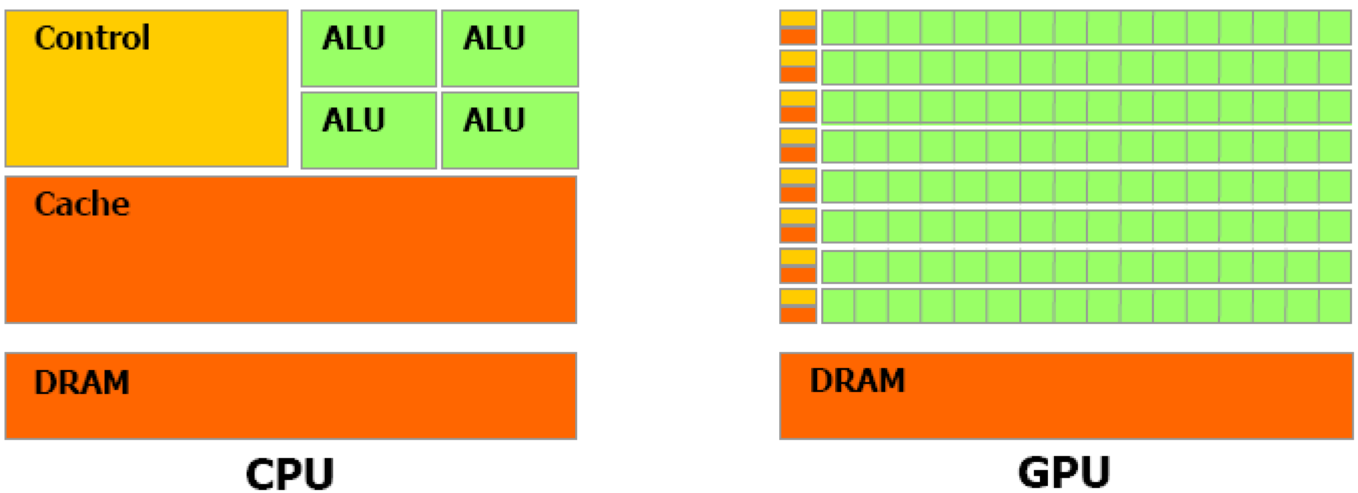



From CUDA_C_Programming_Guide.pdf


GeForce GTX 780 Ti

# Introduction:
# GPU

- Specialized for data parallel computing.
- Uses more transistors to data processing than flow control or data storage.




From CUDA_C_Programming_Guide.pdf

# INTRODUCTION
## CUDA: COMPUTE UNIFIED DEVICE ARCHITECTURE

- GPGPU technology (General-purpose computing on graphics processing units) that lets you use the C programming language to execute code on the graphic processing unit (GPU).
- Developed by NVIDIA.
- To use this architecture it is required to have a GeForce 8 series (or Quadro equivalent), and more recently GPUs.

# INTRODUCTION:
# CUDA FEATURES

- Supports the programming language C/C++, Fortran, Matlab, LabView, etc..
- Unification of hardware and software for parallel computing.
- Supports: Single Instruction, Multiple Data (SIMD).
- Libraries for FFT (Fast Fourier Transform), BLAS (Basic Linear Algebra Subroutines), NPP, TRUSTH, CULA, etc.
- Works internally with OpenGL and DirectX.
- Supports operative systems:
  - Windows, Linux and Mac OS.

# INTRODUCTION:
# CUDA-ENABLED GRAPHIC CARDS


| Architectures | Capability |
|---|---|
| 8-200 series | 1.0 - 1.3 |
| FERMI (400 series) | 2.0 - 2.1 |
| KEPLER (600 series) | 3.0 - 3.5 |

| Next Architectures (2014-2016) | Capability |
|---|---|
| MaxWell | 5.0 - 5.2 |
| Volta-Pascal | -- |

GPU Architectures and Capabilities

# INTRODUCTION: INSTALLING CUDA

- Installing CUDA (http://developer.nvidia.com/cuda/cuda-downloads)

## CUDA Downloads

CUDA 5.5 PRODUCTION RELEASE

| Operating System | Distribution | Architecture x86 64-bit | Architecture x86 32-bit | Architecture ARMv7 | Related Documentation |
|---|---|---|---|---|---|
| Windows | Vista, 7, 8 - Notebook | 64-bit | 32-bit | | Windows Getting Started Guide |
| | Vista, 7, 8 - Desktop | 64-bit | 32-bit | | |
| | XP - Desktop* | 64-bit | 32-bit | | |
| Linux | RHEL 6 | RPM RUN | | | Linux Getting Started Guide<br>RPM / DEB Installation Instructions<br>RUN Installation Instructions |
| | RHEL 5.5 | RUN | | | |
| | Fedora 18 | RPM RUN | | | |
| | OpenSUSE 12.2 | RPM RUN | | | |
| | SLES 11 (SP1 & SP2) | RPM RUN | | | |
| | Ubuntu 12.04 | DEB** RUN | DEB** RUN | DEB | |
| | Ubuntu 12.10 | DEB RUN | DEB RUN | | |
| | Ubuntu 10.04 | RUN | RUN | | |
| Mac OSX | 10.7,10.8 & 10.9 *NEW* | PKG | | | Mac Getting Started Guide |

# QUESTIONS?

?

# IMAGE PROCESSING IN OPENCV

# IMAGE PROCESSING IN OPENCV

- **cv :: Mat**
  - Basic management of matrices

```
// make a 7x7 complex matrix filled with 1+3j.
Mat M(7,7,CV_32FC2,Scalar(1,3));
// and now turn M to a 100x60
// 15-channel 8-bit matrix.
// The old content will be deallocated
M.create(100,60,CV_8UC(15));
```

# IMAGE PROCESSING IN OPENCV

- Class **cv::Mat** is responsible for managing the image
- OpenCV provides functions for reading, showing and saving of images.

```
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>

int main()
{

    // read an image
    cv::Mat image= cv::imread("img.jpg");

    // create image window named "My Image"
    cv::namedWindow("My␣Image");

    // show the image on window
    cv::imshow("My␣Image", image);

    // wait key for 5000 ms
    cv::waitKey(5000);

    return 0;
}
```

# IMAGE PROCESSING IN OPENCV

- **Pixel access**
  - There are different ways to access the pixels within an instance of cv:: Mat. For example, for grayscale images, we can use the member function ".at<type >" (row,col)

```
image.at<uchar>(j,i)= value;
```

  - In the case of more than one channel

```
image.at<cv::Vec3b>(j,i)[channel]= value;
```

# CUDA

# MEMORY ALLOCATION IN THE GPU

# MEMORY ALLOCATION IN THE GPU

- Allocate and free memory
  - **cudaMalloc** `((void**) devPtr, size_t size)`
  - **cudaFree** `(void  *devPtr)`

  - Those are similar to:
    - Malloc()..
    - Free()..

# MEMORY ALLOCATION IN THE GPU

- Copy memory.
  - **cudaMemcpy**`(void *dst, const void *src, size_t count, enum cudaMemcpyKind` **`kind`**`)`
    - ***Kind***:
      - cudaMemcpyHostToHost
      - cudaMemcpyHostToDevice
      - cudaMemcpyDeviceToHost
      - cudaMemcpyDeviceToDevice

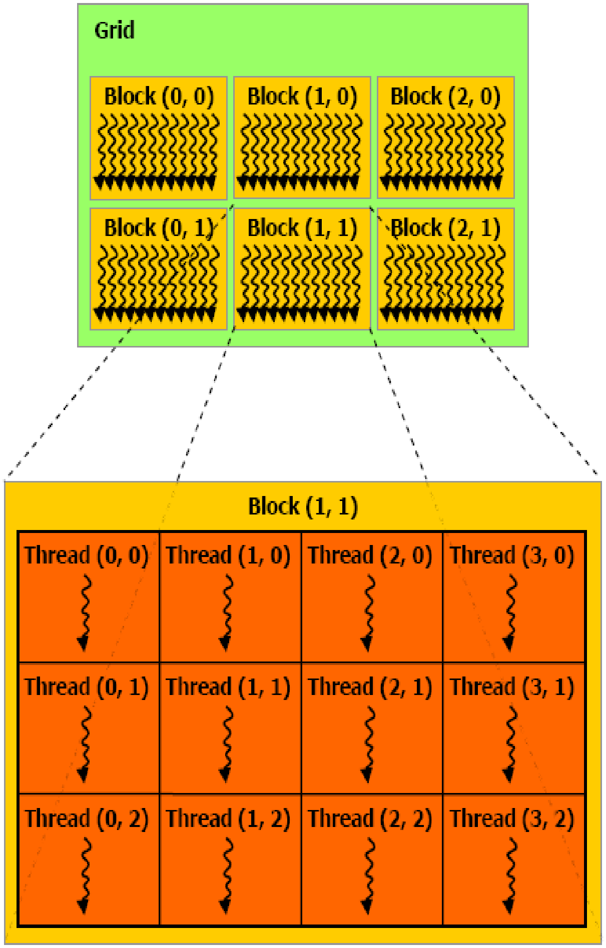
# MEMORY PASSING BETWEEN OPENCV AND CUDA

See example in “MemoryManage.cpp”

# OPERATION ON PARALLEL (GPU MANAGEMENT)

# OPERATION ON PARALLEL: PROGRAMMING MODEL

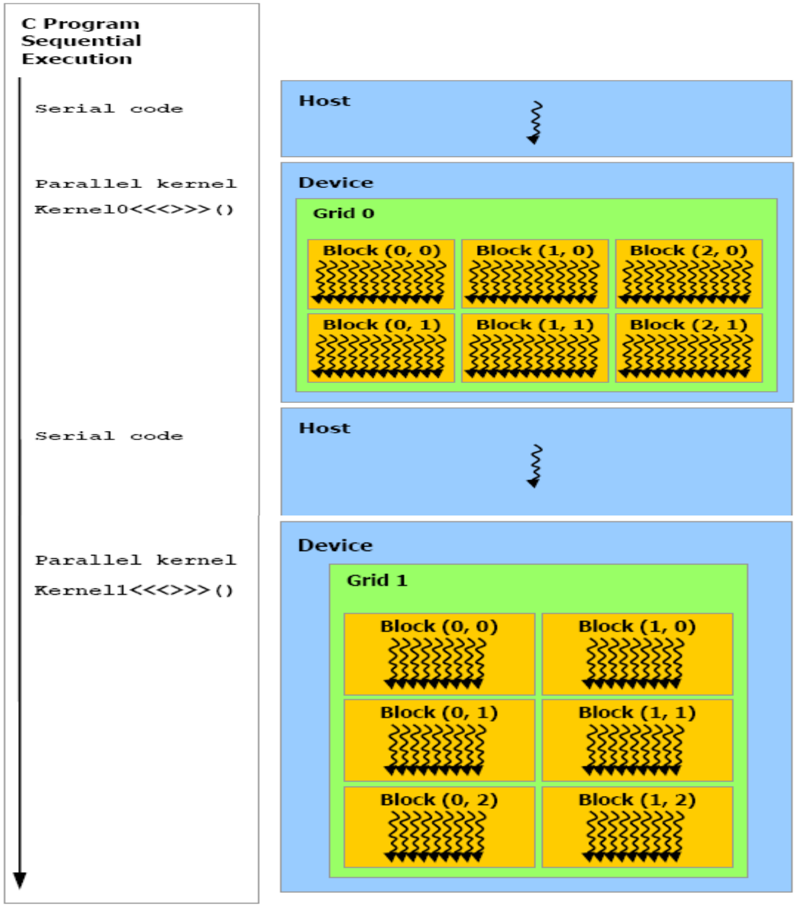
- A program that is compiled to run on a graphics card is called the *Kernel*
- The set of threads that execute a kernel is organized as a **grid** of thread blocks
- A thread block is a set of threads that can cooperate together:
  - Easy access to shared memory
  - Synchronously
  - With a thread identifier ID
  - Blocks can be arranged for 1, 2 or 3 dimensions
- A grid of thread blocks:
  - It has a limited number of threads in a block
  - The blocks are identified by an ID
  - Arrangements can be of 1 or 2 dimensions



# OPERATION ON PARALLEL: PROGRAMMING MODEL

- Running on the Host and Device

Host = CPU
Device = GPU
Kernel = Set of instructions than runs in the device



# OPERATION ON PARALLEL: QUALIFIERS FOR A KERNEL

- **__device__**
  - Runs on the device.
  - Called only from the device.
- **__global__**
  - Runs on the device
  - Called only from the host.

# OPERATION ON PARALLEL: QUALIFIERS FOR VARIABLES

- **__device__**
  - Resides in global memory space.
  - Has the lifetime of an application.
  - Lives accessible from all threads within the grid, and from the host through the library at runtime.

- **Others:**
  - __**constant**__ (Optionally used with __**device**__)
    - Resides in constant memory space.
    - Has the lifetime of an application.
    - Lives accessible from all threads within the grid, and from the host through the library at runtime.
  - __**shared**__ (Optionally used with __**device**__)
    - Lives in shared memory space of a thread block.
    - Has the lifetime of a block.
    - Only accessible from the threads that are within the block.

# OPERATION ON PARALLEL: KERNEL FUNCTION CALLS

- Example function
  - Kernel in the Device:
    - __global__ void NameFunc(float *parameter, …);
  - it must be called as follows:
    - NameFunc <<< Dg, Db, Ns, St >>> (parameter1,…);

- **Dg**: Type *dim3,* dimension and size of the grid.
- **Db**: Type *dim3,* dimension and size of each block.
- **Ns**: Type *size_t,* number of bytes in shared memory.
- **St**: Type *cudaStream_t* that indicates which stream will use the kernel.

  (Ns and St are optional).

# OPERATION ON PARALLEL: AUTOMATICALLY DEFINED VARIABLES

- All __global__ and __device__ functions have access to the following variables:
  - **gridDim** (dim3), indicates the dimension of the grid.
  - **blockIdx** (uint3), indicates the index of the bloque within the grid.
  - **blockDim** (dim3), indicates the dimension of the block.
  - **threadIdx** (uint3), indicates the index of the thread within the block.

# OPERATIONS ON GPU: FIRST EXAMPLES

# OPERATIONS ON GPU: ADD ONE

**CPU C**

```
void add_one_cpu(float *vector, int N)
{
int i;
for (i=0;i<N;i++) {
        vector [j]+=1.0f;
  }
}
void main() {
  .....
  add_one_cpu (a,N);
}
```

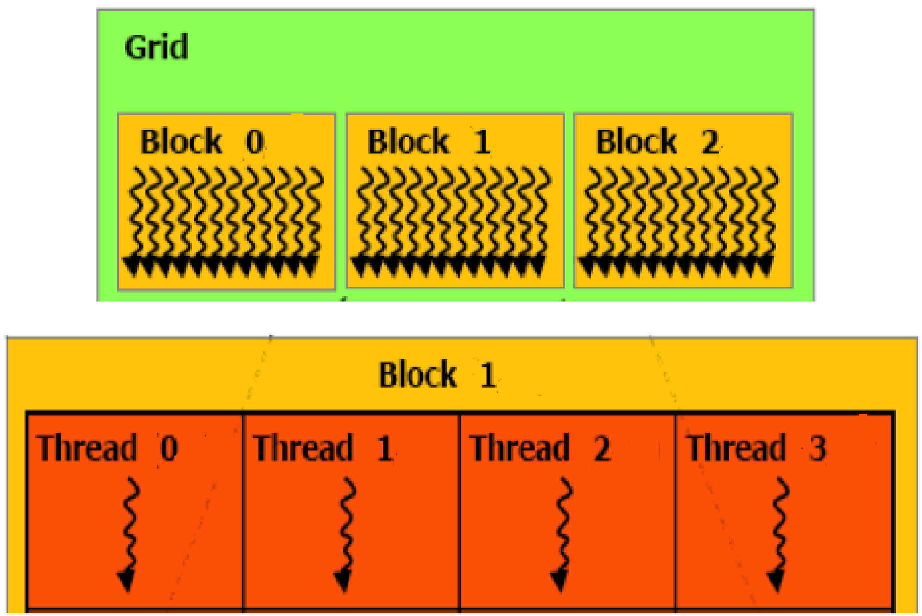
**CUDA C**

```
__global__ void add_one_gpu(float *d_vector, int N)
{
 int i=blockIdx.x*blockDim.x+threadIdx.x;
 if(i <N )
   d_vector[i] += 1.0f;
}

void main() {
 dim3 dimBlock(blocksize, 1, 1);
 dim3 dimGrid(N/dimBlock.x, 1,1);
 add_matrix_gpu<<<dimGrid, dimBlock>>>(a, N);
}
```

# OPERATIONS ON GPU: ADD ONE

- Every element in the vector is processing by every thread in each block



# OPERATIONS ON GPU: ADD VECTORS

- Add two vectors
  - Create host memory: “a_h”, “b_h” and “c_h”
  - Initialize the vectors “a_h” and “b_h”.
  - Create device memory: “a_d”, “b_d” and “c_d”.
  - Copy memory from host to device of vectors a and b.
  - Add vectors a_d and b_d; the result is saved in vector c_d.
  - Copy memory from device to host of vector c.
  - Finally, show the result.
- See “add_vectors.cpp”

# OPERATIONS ON GPU: ADD MATRICES

- Exercise: The code in "add_matrices.cpp" is incomplete; find and correct the mistake.
- Remember:
  - Create host memory: "a_h", "b_h" and "c_h".
  - Initialize "a_h" and "b_h".
  - Crete device memory: "a_d", "b_d" y "c_d".
  - Copy memory from host to device.
  - Add matrix in the device.
  - Copy memory from device to host.
  - Finally, show the result.

# OPERATIONS ON GPU: ADD MATRICES

| | | |
|---|---|---|
| 1,1 | 1,2 | 1,3 |
| 2,1 | 2,2 | 2,3 |
| 3,1 | 3,2 | 3,3 |

→

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

Indexes in Matrix form

Indexes in Vector form

The formula in C/C++ is

Index_vector = i * #cols + j

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

→

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

# OPERATIONS ON GPU: CONSIDERATIONS

- There are some technique to improve the performance of algorithms on GPU.
- Multiple Data, Single Instruction:
  - 32 threads (warp)
  - Avoid use "if".
  - Also, avoid "for" with different stop criteria in each thread

**if()** ← **only 2**
**thread**

**....**

**else** ← **30 trheads**

...

This takes 2 times!

# PARALLEL IMAGE PROCESSING

# PARALLEL IMAGE PROCESSING: EXERCISE: IMAGE COMPOSITION

- Load two images and reserve memory to the output image.
- Create memory on Device (for the 3 images).
- Copy memory of the Host to Device.
- Loop:
  - Kernel (CUDA_Compose_Images)
  - Return the result on the Host
  - Show the result
- Free the memory

# PARALLEL IMAGE PROCESSING: EXERCISE: GRADIENT MAGNITUDE

- Load the original image in host memory.
- Create device memory: Imag_dev, ImagDx_dev, ImagDy_dev, ImagMG_dev.
- Copy the original image from host to device memory.
- Calculate Dx, Dy and GM in the device.

$$D_x(x,y) = I(x,y) - I(x-1,y)$$
$$D_y(x,y) = I(x,y) - I(x,y-1)$$
$$GM(x,y) = \sqrt{D_x^2(x,y) + D_y^2(x,y)}$$

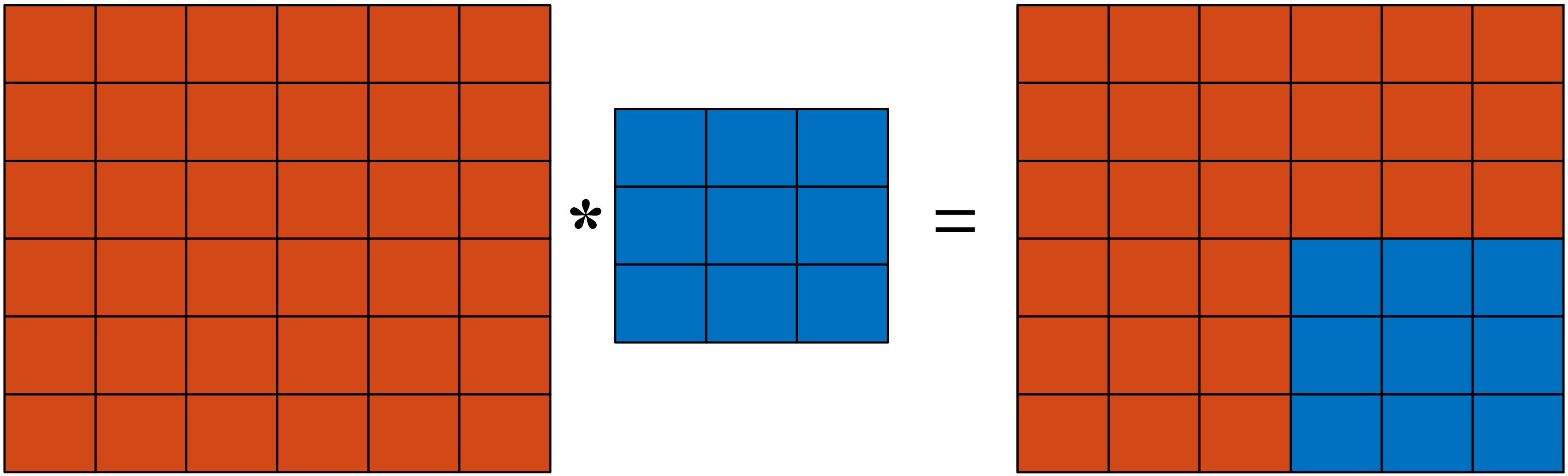
- Copy the result from device to host memory.
- Show the result.

# PARALLEL IMAGE PROCESSING: IMAGE FILTERING

- Example: Mean filter
  - Load the original image in host memory.
  - Create device memory.
  - Copy the original image from host to device memory.
  - Calculate the mean filter.
  - Copy the result from device to host memory.
  - Show the result.

# PARALLEL IMAGE PROCESSING: IMAGE FILTERING

- Mean filter with window size of 3x3:



Image

Convolution Kernel

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

# PARALLEL IMAGE PROCESSING: IMAGE FILTERING

- Exercises: Gaussian and Laplacian filters
  - Load the original image in host memory.
  - Create device memory.
  - Copy the original image from host to device memory.
  - Calculate the Gaussian or Laplacian filter.
  - Copy the result from device to host memory.
  - Show the result

Gaussian Filter:

$$\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}$$

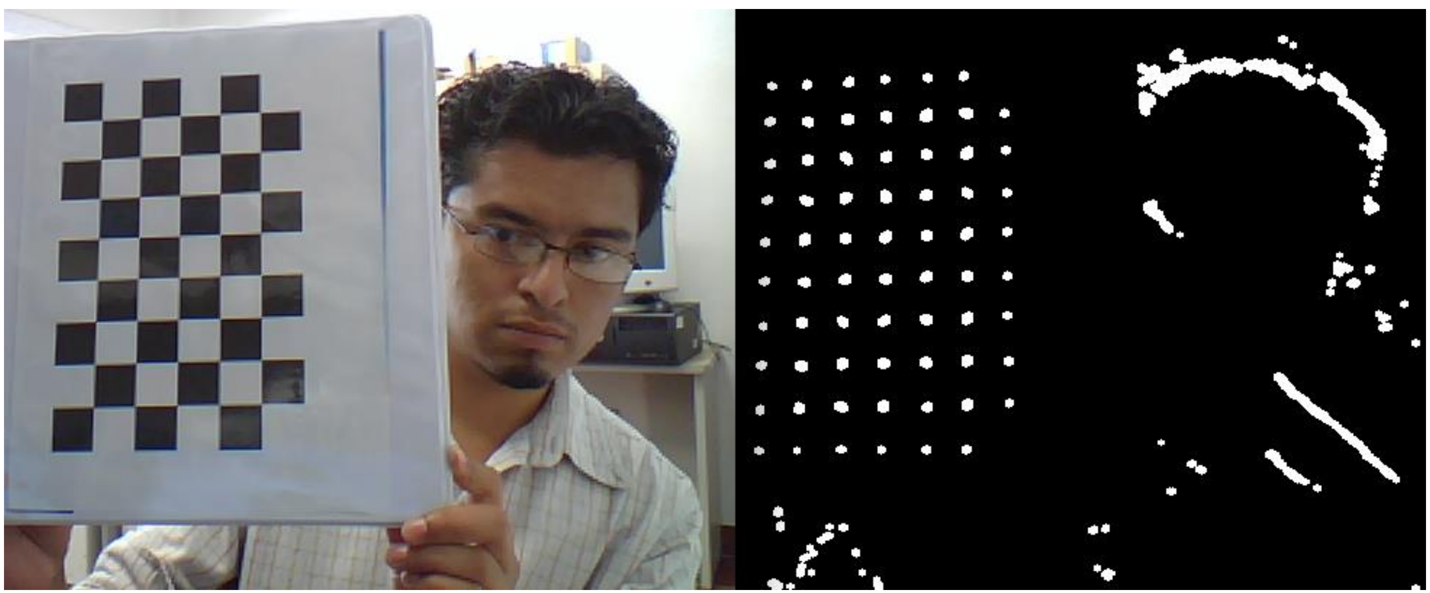
Laplacian Filter:

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

# PARALLEL IMAGE PROCESSING: CORNER DETECTOR

- Exercise: Corner detector with the structure tensor

$$\begin{bmatrix} D_x^2 & D_x D_y \\ D_x D_y & D_y^2 \end{bmatrix}$$



# PARALLEL IMAGE PROCESSING: EXERCISE - DIFFUSION IMAGE

- Given an image g(x) with noise.
- Smooth the image g(x) with the following functional:

$$U[f(x)] = \frac{1}{2}\sum_{x}[f(x) - g(x)]^2 + \frac{\lambda}{2}\sum_{<x,y>}[f(x) - f(y)]^2$$

- Differentiating and equating to zero, we obtain:

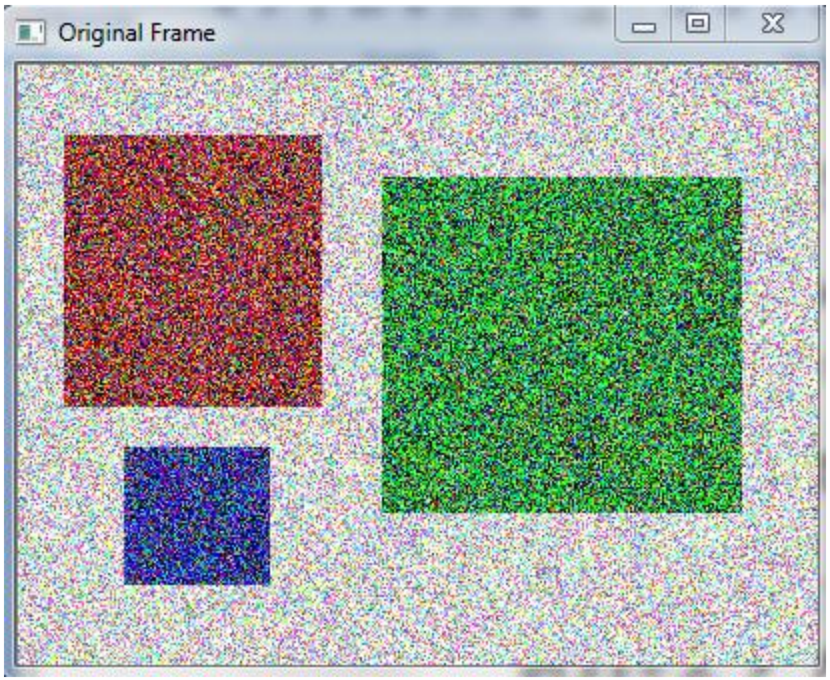
$$f^{k+1}(x) = \frac{g(x) + \lambda \sum_{y \in N_x} f^k(y)}{1 + \lambda |N_x|}$$

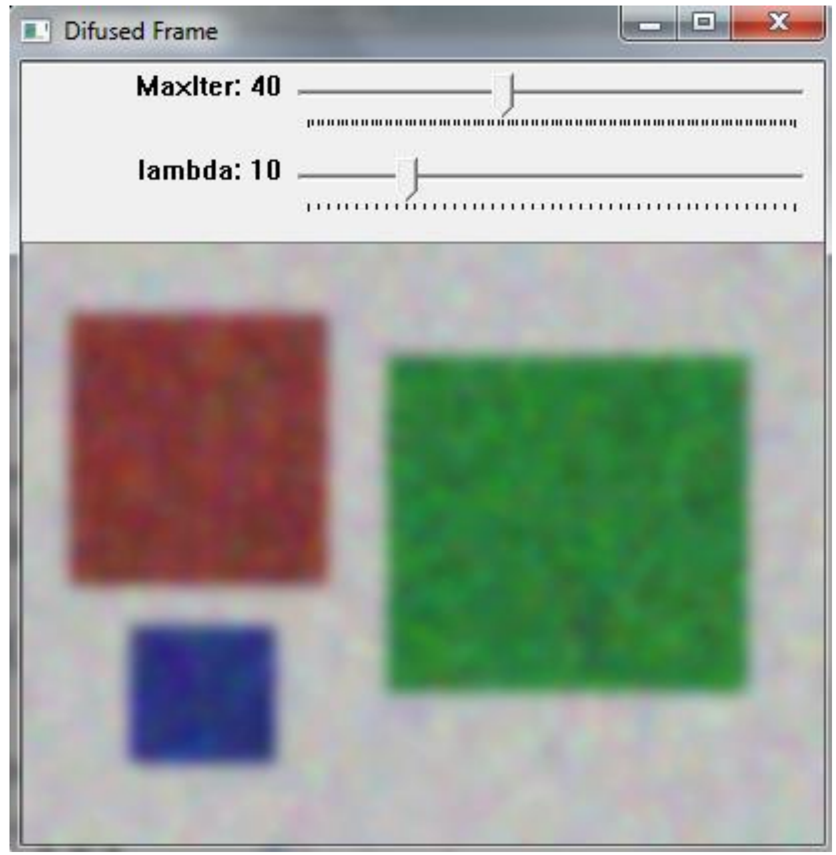
$|N_x| = \#$ neighborhoods of pixel x

$$f^0(x) = g(x)$$

- We can solve by:
  - Jacobi
  - Gauss-Seidel

# PARALLEL IMAGE PROCESSING: EXERCISE - DIFFUSION IMAGE



# PARALLEL IMAGE PROCESSING USING MULTIPLE GPUS: EXAMPLES

- GPUs can be controlled by:
  - A single CPU thread
  - Multiple CPU threads



# PARALLEL IMAGE PROCESSING USING MULTIPLE GPUS: EXAMPLES

- Asynchronous calls (kernels, memcopies) don't block switching the GPU.
- The following code will have both GPUs executing concurrently:
  - cudaSetDevice( 0 );
  - kernel<<<...>>>(...);
  - cudaSetDevice( 1 );
  - kernel<<<...>>>(...);

# PARALLEL IMAGE PROCESSING USING MULTIPLE GPUS: EXAMPLES

- Using multiple GPUs with "OpenMP"

# GPU MODULE DESIGN CONSIDERATIONS

- Key ideas
  - Explicit control of data transfers between CPU and GPU
  - Minimization of the data transfers
  - Completeness
    - Port everything even functions with little speed-up
- Solution
  - Container for GPU memory with upload/download functionality
  - GPU module function take the container as input/output parameters

# GPU Module Design Considerations

- Class GpuMat –for storing 2D (pitched) data on GPU
  - Interface similar to cv::Mat(), supports reference counting
  - Its data is not continuous, extra padding in the end of each row
  - It contains:
    - **data** - Pointer data beginning in GPU memory
    - **step** – Distance in bytes is between two consecutive rows
    - **cols, rows** - Fields that contain image size
    - **upload/download** – Up/down memory from device

# OpenCV GPU Module Example

```
Mat frame;
VideoCapture capture(camera);
cv::HOGDescriptor hog;
hog.setSVMDetector(cv::HOGDescriptor::
      getDefaultPeopleDetectorector());

capture >> frame;



vector<Rect> found;
hog.detectMultiScale(frame, found,
     1.4, Size(8, 8), Size(0, 0), 1.05, 8);
```

```
Mat frame;
VideoCapture capture(camera);
cv::gpu::HOGDescriptor hog;
hog.setSVMDetector(cv::HOGDescriptor::
      getDefaultPeopleDetectorector());

capture >> frame;

GpuMat gpu_frame;
gpu_frame.upload(frame);

vector<Rect> found;
hog.detectMultiScale(gpu_frame, found,
     1.4, Size(8, 8), Size(0, 0), 1.05, 8);
```

Designed very similar!

# CONCLUSIONS:

- **CPU**
  - **Incremental improvements (memory caches and complex architectures)**
  - **Few Multi-core (4/8/16)**
- **GPU**
  - **Highly parallel with 100s of simple cores**
  - **Easier to extend by adding more GPUs**
  - **Continue to grow exponentially!**
  - **Most of the GPUs are cheap!**

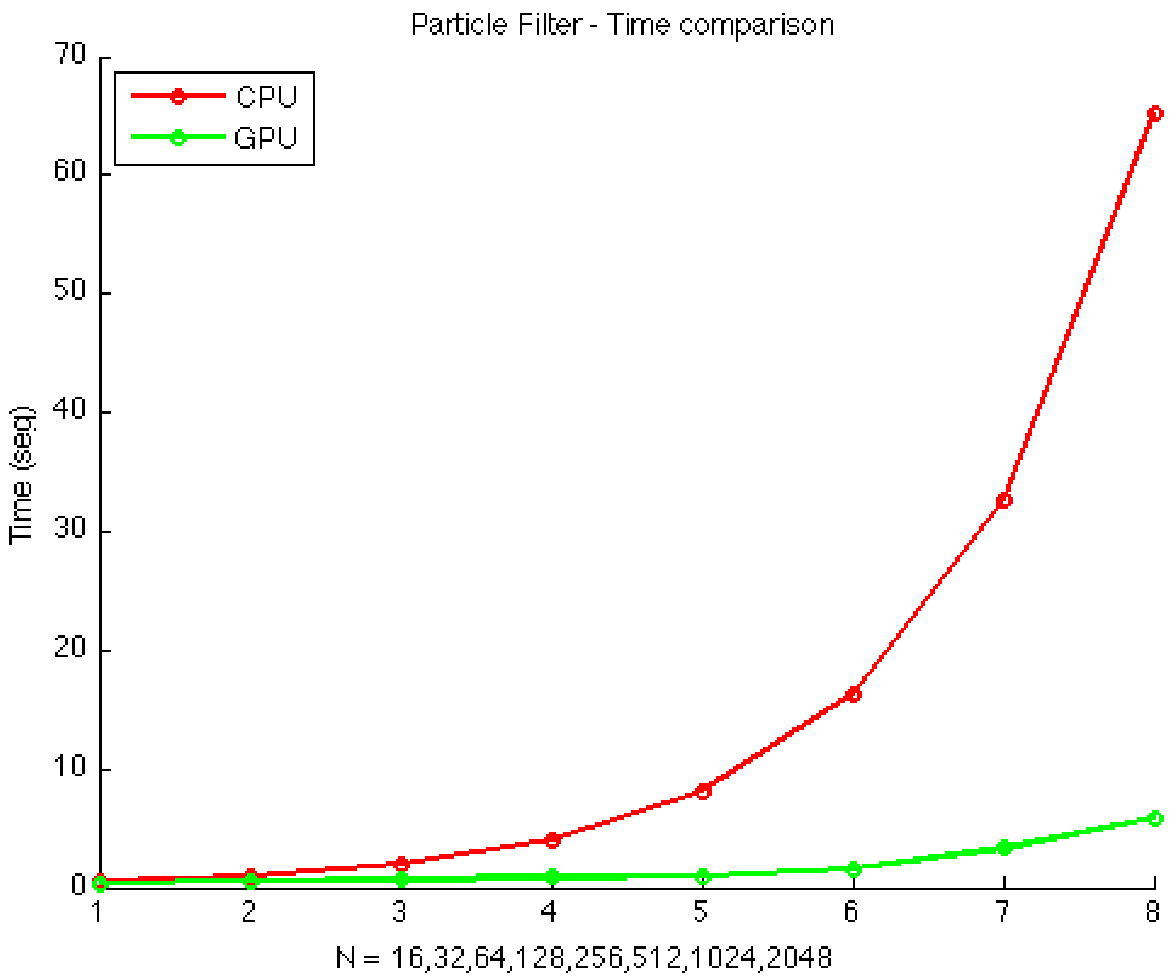
# CONCLUSIONS:

- We presented a small introduction of the parallel processing using GPUs.
- There are many sofistecated  strategies for make up your GPU-code faster.
- Most problems can be parallelized and are suitable to be run on GPUs
- One has to consider the properties of the GPU (shared memory, cache, compute capability) when designing the kernels

# CONCLUSIONS: POTENTIAL APPLICATIONS

Tracking

# CONCLUSIONS:
# POTENTIAL APPLICATIONS

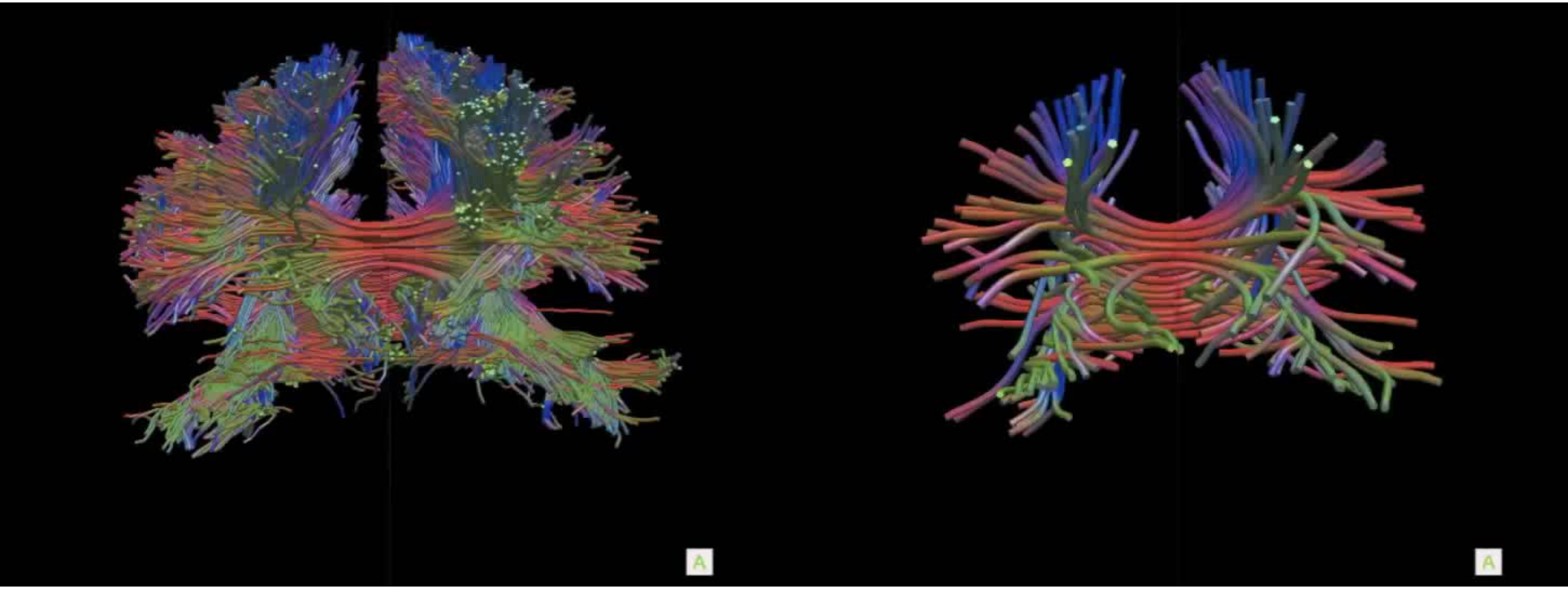



Tracking

# CONCLUSIONS: POTENTIAL APPLICATIONS


VScreen

# CONCLUSIONS: POTENTIAL APPLICATIONS




Tract Estimations from the callosum corpus

**Tractography**

# QUESTIONS?

?

# Thank you!