High Performance Computing & Big Data Services

hpc.uni.lu

hpc@uni.lu

@ULHPC

# Uni.lu HPC School 2020

## PS7: Introduction to GPU programming with CUDA (Part II)

## Image Convolution with GPU and CUDA


**Uni.lu High Performance Computing (HPC) Team**

**F. Pinel, L. Koutsantonis**

University of Luxembourg (UL), Luxembourg

http://hpc.uni.lu


## Latest versions available on Github:

UL HPC tutorials: https://github.com/ULHPC/tutorials

UL HPC School: http://hpc.uni.lu/hpc-school/

PS7 tutorial sources: ulhpc-tutorials.rtfd.io/en/latest/cuda/exercises/convolution/

SCHOOL 2020

# Objectives of this session

Complementary Hands On with CUDA:

1. GPU Global Memory Allocation
2. Dynamic Shared Memory Allocation
3. Thread Indexing
4. Thread Synchronization

Application to **Image Convolution implementation on GPU**

# Summary

# Laplacian of Gaussian (LoG)

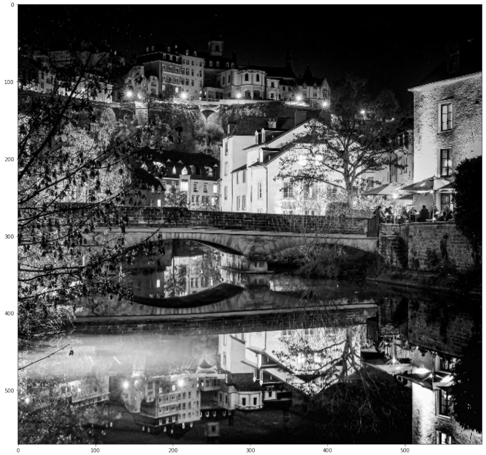
- Derivative Filter used to find rapid changes in signals and especially images
- Used for edge detection and noise detection
- Mathematical Formula:

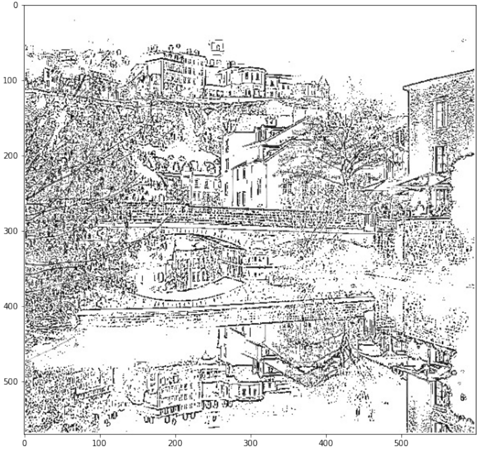
$$H(x,y) = \frac{-1}{\pi\sigma^4}\left(1 - \frac{x^2+y^2}{2\sigma^2}\right)e^{-\frac{x^2+y^2}{2\sigma^2}}$$

$\hookrightarrow$ Discrete convolution kernel (of a finite size W = 3):

$$H_{ij}(\sigma) = \begin{pmatrix} 1 & 2 & 1 \\ 2 & -16 & 2 \\ 1 & 2 & 1 \end{pmatrix}$$

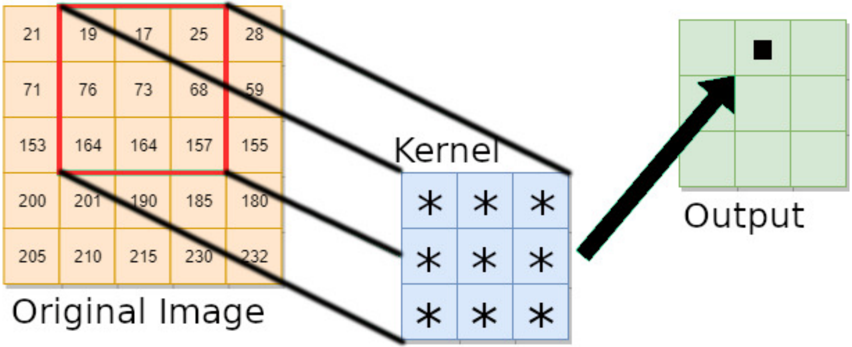
# Convolution (Example with LoG)



# Convolution (Operator)

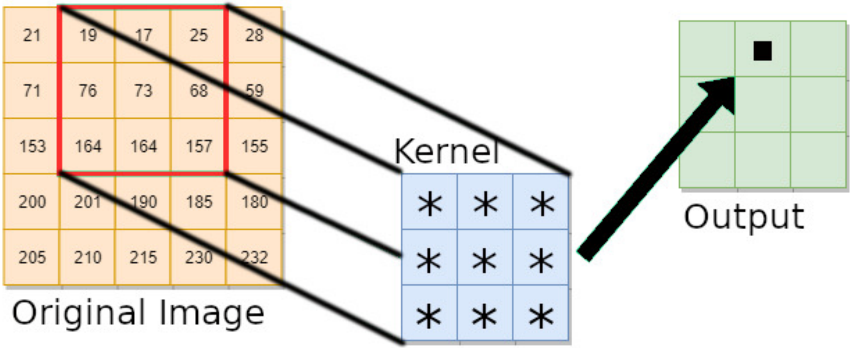
$$G_{m,n} = F * H = \sum_i \sum_j F_{m-i,n-j} H_{i,j}$$



# Convolution (Operator)

Input Image:
$Ny \times Nx$

Output Image:
$(Ny - W + 1) \times (Nx - W + 1)$



# Summary

# CPU Implementation

```
//CPU function: conv_img_cpu
//Parameters: float *img, float *kernel, float *imgf, int Nx, int Ny, int kernel_size
//center: center of kernel
  for (int i = center; i<(Ny-center); i++)
    for (int j = center; j<(Nx-center); j++){
        sum = 0;
        for (int ki = 0; ki<kernel_size; ki++)
           for (int kj = 0; kj<kernel_size; kj++){
               ii = j + kj - center;
               jj = i + ki - center;
               sum+=img[jj*Nx+ii]*kernel[ki*kernel_size + kj];
               }
        imgf[i*Nx +j] = sum;
        }
```

# Summary

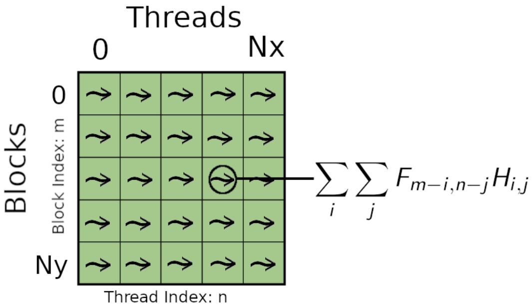
# Exercise: GPU Implementation

Your Turn!

## GPU Implementation

▶ url ◀ | github | src

- implement and launch a CUDA kernel performing the image convolution
- Each GPU block performs the convolution of a single image row.
  - ↪ Each block thread computes the convolution of a single element.



# Exercise: GPU Implementation

1. Access to the Data

```
# Clone  Tutorials github repository
(access)$> cd ~/git/github.com/ULHPC/tutorials/
(access)$> git pull
(access)$> cd cuda/exercises/convolution/
```

***Source File:*** *LoG_gpu_exercise.cu*

***Input File:*** *lux_bw.dat* *(Grayscale Image in Ascii Format)*

# Exercise: GPU Implementation

2. Write a GPU function (kernel) implementing the image convolution

```
__global__ void conv_img_gpu(float *img, float *kernel, float *imgf,
    int Nx, int Ny, int kernel_size)
```

**Hint!**

You can dynamically allocate shared memory and synchronize block threads:

```
extern __shared__ float myvect[];
//code
__syncthreads();
```

# Exercise: GPU Implementation

3. In main, allocate GPU memory (`d_kernel`) and transfer the filter coefficients .
   ↪ Allocate GPU memory for the resulting image (`d_imgf`)
   ✓ e.g. the original image is preloaded in `d_img`

## Hint!

- To **allocate** global memory, **copy from/to**, and **de-allocate** use:

```
cudaMalloc(void** devPtr, size_t count);

cudaMemcpy(void* dst, const void* src, size_t count, cudaMemcpyHostToDevice);

cudaFree(void* devPtr);
```

# Exercise: GPU Implementation

4
a. Configure the execution (`Number_of Blocks, Number_of_Threads`)
b. launch the kernel (`mykernel<<<Number_of_Blocks, Number_of_Threads>>>()`) to compute the convoluted image.

**Hint!**

- To dynamic allocate the shared memory lauch you kernel with:

```
mykernel<<<Number_of_blocks, Number_of_Threads, memory_size>>>(... myparameters...);
```

# Exercise: GPU Implementation

5. Plot your results on your local machine
   ↪ use Python or any other scripting language

**Hint!**

- To copy your result to your local PC use:

```
# prefer rsync -- don't forget trailing '.' means 'here'
rsync -avzu  iris-cluster:path/to/fname .
```

- A readily available jupyter notebook can be used to plot your results!
  ↪ see `show_images.ipynb`

# Last remarks

- Reserve an **interactive GPU job**

```
### ... either directly - dedicate 1/4 of available cores to the management of GPU card
$> si-gpu -c7
# /!\ warning: append -G 1 to really reserve a GPU
# srun -p interactive --qos debug -C gpu -c7 -G 1 --mem-per-cpu 27000 --pty bash

### ... or using the HPC School reservation 'hpcschool-gpu'
srun --reservation=hpcschool-gpu -p gpu --ntasks-per-node 1 -c7 -G 1 --pty bash
```

- Load CUDA and GCC modules

  `module load compiler/GCC system/CUDA`

- Compile CUDA program

  `nvcc -arch=compute_70 -o exe src`

# Summary

# Device function Implementation (1/4)

```
__global__ void conv_img_gpu(float *img, float *kernel, float *imgf,
                             int Nx, int Ny, int kernel_size)
{

  //local ID of each thread (withing block)
  int tid = threadIdx.x;

  //each block is assigned to a row of an image, iy index of y value
  int iy = blockIdx.x + (kernel_size - 1)/2;

  //each thread is assigned to a pixel of a row, ix index of x value
  int ix = threadIdx.x + (kernel_size - 1)/2;

  //idx global index (all blocks) of the image pixel
  int idx = iy*Nx +ix;
...
```

# Device function Implementation (2/4)

```
//total number of kernel elements
 int K2 = kernel_size*kernel_size;

 //center of kernel in both dimensions
 int center = (kernel_size -1)/2;

 //Auxiliary variables
 int ii, jj;
 float sum = 0.0;

/*
Define a vector (float) sdata[] that will be hosted in shared memory,
*extern* dynamic allocation of shared memory:
*/
 extern __shared__ float sdata[];
...
```

# Device function Implementation (3/4)

```
/*Transfer data frm GPU memory to shared memory
tid: local index, each block has access to its local shared memory
e.g. 100 blocks -> 100 allocations/memory spaces
Eeah block has access to the kernel coefficients which are store in shared memory
Important: tid index must not exceed the size of the kernel*/

  if (tid<K2)
    sdata[tid] = kernel[tid];

  //Important. Syncronize threads before performing the convolution.
  //Ensure that shared memory is filled by the tid threads

  __syncthreads();

...
```

# Device function Implementation (4/4)

```
/*
Each thread computes the one pixel value of the output image;
number of computations per thread = size_kernel^2
The result is stored to imgf
*/

if (idx<Nx*Ny){
  for (int ki = 0; ki<kernel_size; ki++)
    for (int kj = 0; kj<kernel_size; kj++){
        ii = kj + ix - center;
        jj = ki + iy - center;
        sum+=img[jj*Nx+ii]*sdata[ki*kernel_size + kj];
        }
  imgf[idx] = sum;
  }
}
```

# GPU memory Allocation and Data Transfer

```
//Allocate GPU memory

cudaMalloc(&d_img,Nx*Ny*sizeof(float));
cudaMalloc(&d_imgf,Nx*Ny*sizeof(float));
cudaMalloc(&d_kernel,kernel_size*kernel_size*sizeof(float));
```

```
cudaMemcpy(d_img, img, Nx*Ny*sizeof(float),cudaMemcpyHostToDevice);

//Copy kernel coefficient to device memory

cudaMemcpy(d_kernel,kernel, kernel_size*kernel_size*sizeof(float),cudaMemcpyHostToDevice);
```

# Kernel Launch Configuration

```
//Launch GPU kernel

Nblocks = Ny - (kernel_size-1);
Nthreads = Nx - (kernel_size-1);
conv_img_gpu<<<Nblocks, Nthreads, kernel_size*kernel_size*sizeof(float)>>>
                (d_img, d_kernel, d_imgf, Nx, Ny, kernel_size);
```

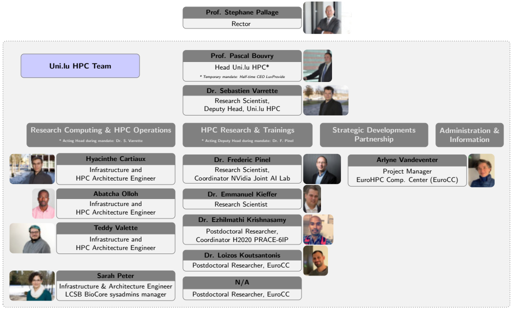
```
//Copy the result from GPU convolution back to the host memory

cudaMemcpy(imgf, d_imgf, Nx*Ny*sizeof(float), cudaMemcpyDeviceToHost);
```

Thank you for your attention...

# Questions?

## *High Performance Computing @ Uni.lu*





University of Luxembourg, Belval Campus
Maison du Nombre, 4th floor
2, avenue de l'Université
L-4365 Esch-sur-Alzette
*mail:* hpc@uni.lu




https://hpc.uni.lu