



Kernel Image Processing with CUDA

Parallel Computing
(Final-Term Assignment)

Davide Del Bimbo

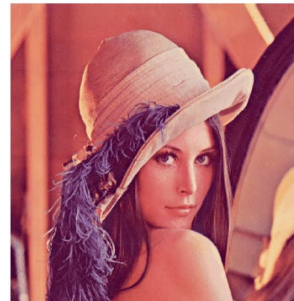
davide.delbimbo@edu.unifi.it

Introduction

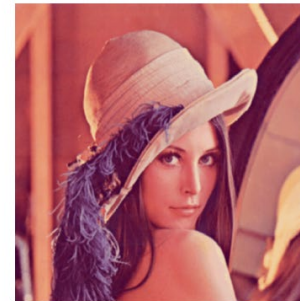
- An image is composed of pixels and each pixel consists of a primary colors combination.
- Kernels allow certain operations to be performed on images (blurring, edge detection, sharpening, etc...).
- A kernel is a small matrix representing a **function** to be applied to an original input image:
 - Each pixel in the processed output image represents a function of nearby pixels in the input image.
- The operation we apply is known as **convolution**:

$$g(x, y) = \omega \otimes f(x, y) = \sum_{i=-a}^a \sum_{j=-b}^b \omega(i, j) \cdot f(x - i, y - j)$$

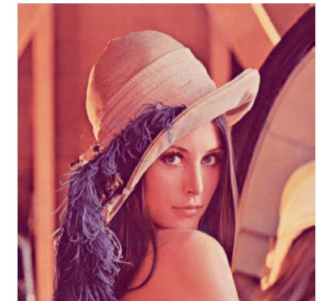
- Different types of filter kernels (depending on kernel coefficients):
 - Box blur.
 - Gaussian blur.
 - Edge detection.
 - Sharpen.
 - Unsharp Mask.
 - ...



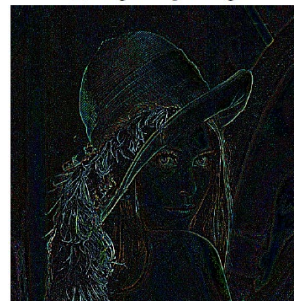
(a) Original input image.



(b) Box Blur effect.



(c) Gaussian Blur effect.



(d) Edge Detection effect.



(e) Sharpen effect.

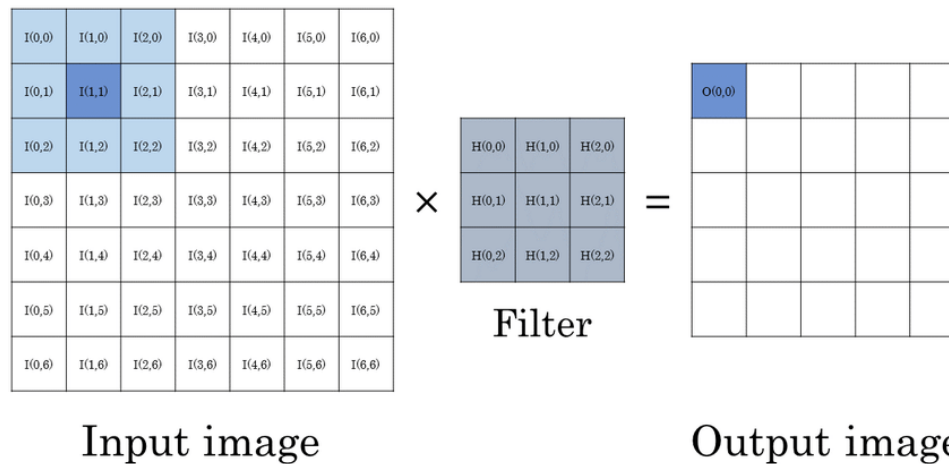


(f) Unsharp Mask effect.

Algorithm

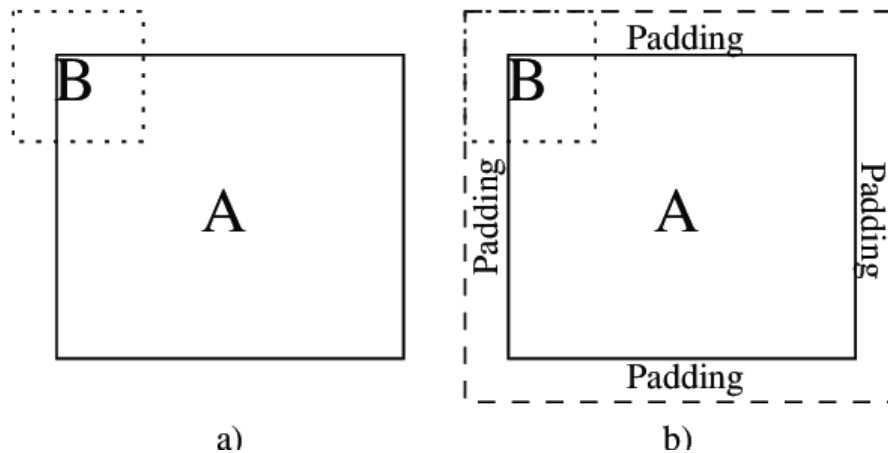
1. Get a *feature map* f and a *kernel* ω .
2. For each pixel in f :
 - A. Set an accumulator to 0.
 - B. Perform the convolution. For each element in ω , if element position corresponds to pixel position, then:

$$\text{accumulator} += \text{pixel} * \text{element}$$
 - C. Set the corresponding output pixel of g to accumulator.



Padding

- Handle boundary cases in which the convolution is performed on pixels located at the edges of the input image.



- Different type of padding:
 - Constant.
 - Replicate.
 - Mirror.
 - ...

| | | | |
|---|-----|-----|-----|
| 0 | 0 | 0 | 0 |
| 0 | 165 | 95 | 215 |
| 0 | 222 | 144 | 199 |
| 0 | 255 | 172 | 83 |

| | | | |
|-----|-----|-----|-----|
| 165 | 165 | 95 | 215 |
| 165 | 165 | 95 | 215 |
| 222 | 222 | 144 | 199 |
| 255 | 255 | 172 | 83 |

| | | | |
|-----|-----|-----|-----|
| 144 | 222 | 144 | 199 |
| 95 | 165 | 95 | 215 |
| 144 | 222 | 144 | 199 |
| 172 | 255 | 172 | 83 |

Implementation

- **Image**: class representing a multi-channel image.
 - `width, height, channels`: image sizes (`int`).
 - `data`: array containing multi-channel image data (`uint8_t`).
 - `is_SoA`: image architecture (`bool`).
- **Kernel**: class representing a two-dimensional kernel.
 - `width, height`: kernel sizes (`int`).
 - `data`: array containing kernel data (`uint8_t`).



```
// Image in multi-channel space.
class Image {
    private:
        int width = 0, height = 0, channels = 0; // Image dimensions.
        float *data = NULL; // Kernel data.
        bool is_SoA = false; // SoA flag.
    public:
        Image(const char* filename, const int channel_force, const bool is_SoA);
        bool load_image(const char* filename, const int channel_force);
        void save_image(const char* filename);
        Image padding(const int padding_width, const int padding_height);
        uint8_t& operator()(const int col, const int row, const int channel) const;
        bool operator==(const Image& other) const;
        void AoS_to_SoA();
}

// Kernel in 2D space.
class Kernel {
    private:
        int width = 0, height = 0; // Kernel dimensions.
        float *data = NULL; // Kernel data.
    public:
        Kernel(const int width, const int height, float *data);
        float& operator()(const int col, const int row) const;
}
```



```
// Image in multi-channel space.
class Image {
    private:
        int width = 0, height = 0, channels = 0; // Image dimensions.
        float *data = NULL; // Kernel data.
        bool is_SoA = false; // SoA flag.
    public:
        Image(const char* filename, const int channel_force, const bool is_SoA);
        bool load_image(const char* filename, const int channel_force);
        void save_image(const char* filename);
        Image padding(const int padding_width, const int padding_height);
        uint8_t& operator()(const int col, const int row, const int channel) const;
        bool operator==(const Image& other) const;
        void AoS_to_SoA();
}

// Kernel in 2D space.
class Kernel {
    private:
        int width = 0, height = 0; // Kernel dimensions.
        float *data = NULL; // Kernel data.
    public:
        Kernel(const int width, const int height, float *data);
        float& operator()(const int col, const int row) const;
}
```

```
// Image in multi-channel space.
```

```
class Image {
```

```
private:
```

```
    int width = 0, height = 0, channels = 0; // Image dimensions.
```

```
    float *data = NULL; // Kernel data.
```

```
    bool is_SoA = false; // SoA flag.
```

```
public:
```

```
    Image(const char* filename, const int channel_force, const bool is_SoA);
```

```
    bool load_image(const char* filename, const int channel_force);
```

```
    void save_image(const char* filename);
```

```
    Image padding(const int padding_width, const int padding_height);
```

```
    uint8_t& operator()(const int col, const int row, const int channel) const;
```

```
uint8_t &Image::operator()(const int col, const int row, const int channel) const {
```

```
    // Get the 1D pixel index.
```

```
    const int pixel_index = is_SoA ?
```

```
        ((channel * width * height) + (row * width) + col) :
```

```
        ((row * width + col) * channels + channel);
```

```
    return data[pixel_index];
```

```
}
```

```
    float *data = NULL; // Kernel data.
```

```
public:
```

```
    Kernel(const int width, const int height, float *data);
```

```
    float& operator()(const int col, const int row) const;
```

```
}
```

```
// Image in multi-channel space.
class Image {
    private:
        int width = 0, height = 0, channels = 0; // Image dimensions.
        float *data = NULL; // Kernel data.
        bool is_SoA = false; // SoA flag.
    public:
        Image(const char* filename, const int channel_force, const bool is_SoA);
        bool load_image(const char* filename, const int channel_force);
        void save_image(const char* filename);
        Image padding(const int padding_width, const int padding_height);
        uint8_t& operator()(const int col, const int row, const int channel) const;
        bool operator==(const Image& other) const;
        void AoS_to_SoA();
}

// Kernel in 2D space.
class Kernel {
    private:
        int width = 0, height = 0; // Kernel dimensions.
        float *data = NULL; // Kernel data.
    public:
        Kernel(const int width, const int height, float *data);
        float& operator()(const int col, const int row) const;
}
```

```
// Image in multi-channel space.
class Image {
private:
    int width = 0, height = 0, channels = 0; // Image dimensions.
    float *data = NULL; // Kernel data.
    bool is_SoA = false; // SoA flag.
public:
    Image(const char* filename, const int channel_force, const bool is_SoA);
    bool load_image(const char* filename, const int channel_force);
    void save_image(const char* filename);
    Image padding(const int padding_width, const int padding_height);
    uint8_t& operator()(const int col, const int row, const int channel) const;
    bool operator==(const Image& other) const;
    void AoS_to_SoA();
}
```

```
float &Kernel::operator()(const int col, const int row) const {
    // Get the 1D kernel index.
    const int kernel_index = (row * width) + col;

    return data[kernel_index];
}
```

```
float& operator()(const int col, const int row) const;
```

```
}
```

Sequential Implementation

- Convolution: method to perform the convolution operation:
 - Get padded image and kernel.
 - Returns convolved image.
- Convolve: method to execute convolution operation several times and calculate the average execution time.

```
Image convolution(const Image& image, const Kernel& kernel, const Image& padded_image) {  
    // Initialize the output image.  
    Image output_image = Image(width, height, channels, image.get_is_SoA()); // Output image.  
  
    // Iterate over the image.  
    for (int channel = 0; channel < channels; channel++) {  
        for (int y = 0; y < height; y++) {  
            for (int x = 0; x < width; x++) {  
                // Output value for the current pixel.  
                float output_value = 0;  
  
                // Iterate over the kernel.  
                for (int ky = 0; ky < kernel_height; ky++) {  
                    for (int kx = 0; kx < kernel_width; kx++) {  
                        // Get the pixel index to be convolved.  
                        const int col = x + kx - floor((float)kernel_width/2) + padding_width;  
                        const int row = y + ky - floor((float)kernel_width/2) + padding_height;  
  
                        // Convolve the pixel.  
                        output_value += padded_image(col, row, channel) * kernel(kx, ky);  
                    }  
                }  
  
                // Set the output value (clamped between 0 and 255).  
                output_image(x, y, channel) = (uint8_t)clamp(0.0f, output_value, 255.0f);  
            }  
        }  
    }  
  
    // Return the convolved image.  
    return output_image;  
}
```

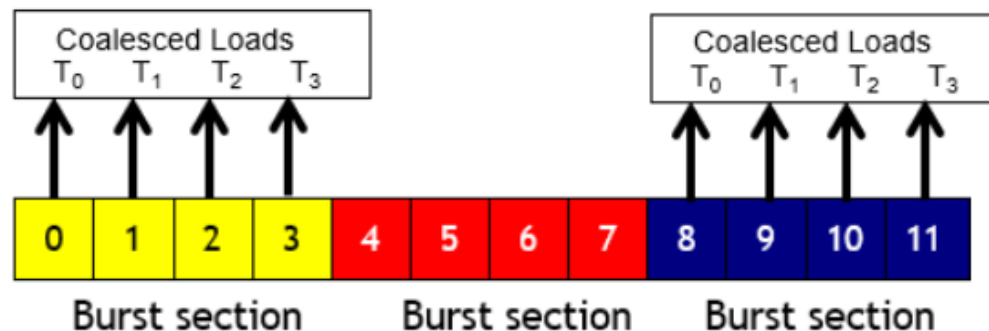
Parallel implementation

- Implemented with CUDA to exploit the power of the GPU.
- Convolution operation represents an embarrassingly parallel problem:
 - Depends exclusively on the pixel values in the original input image and the kernel coefficients.
 - These values are only read and never changed during the convolution operation.
 - The output pixels are written only once and by only one thread.
 - Therefore, each thread can read, perform the convolution operation and write the output to memory without the need of synchronization with other threads.

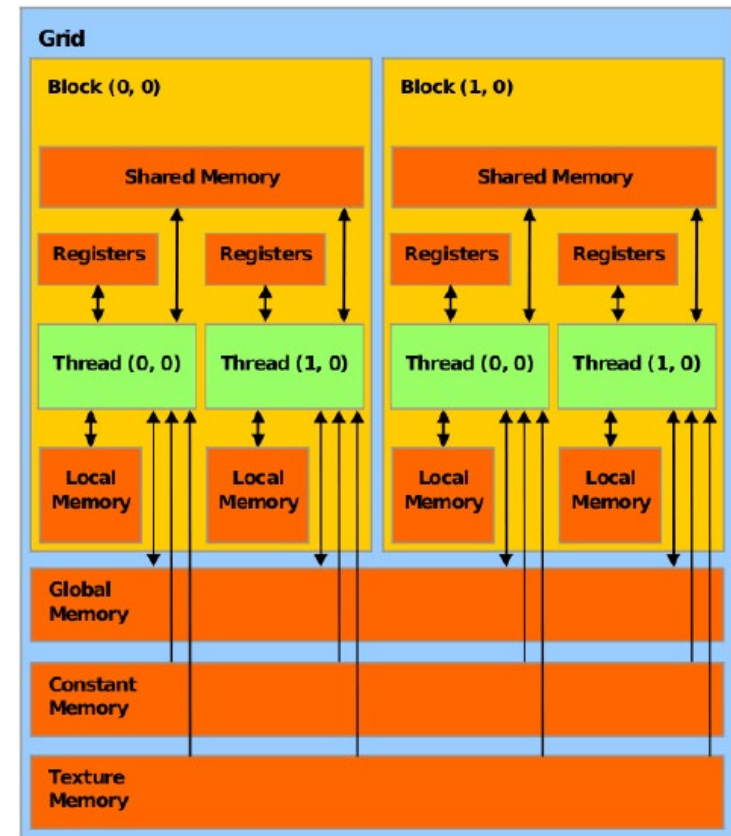




- Can use **SoA** architecture to enable coalesced memory access:
 - SoA architecture turns out to be more cache-friendly, enhancing spatial and temporal locality of data.
 - In fact, during a read of an element, some following values are also loaded into the cache due to **memory burst**.
 - This allow consecutive threads to access consecutive memory addresses, ensuring **coalesced memory access**.



- Different level of device memory:
 - **Global memory:** is the main and slowest memory in the device. Is shared among all blocks in the grid.
 - **Constant memory:** is a small and fast memory that can be used to store variables that will not be changed during processing. Is shared among all blocks in the grid.
 - **Shared memory:** is a small and low-latency memory, shared among all threads in a block.



- Global memory kernel (one thread compute convolution with nearby pixel):

```
__global__ void convolution_kernel_global(uint8_t* d_input, float* d_kernel, uint8_t* d_output,
                                         int width, int height, int channels,
                                         int kernel_width, int kernel_height,
                                         int padding_width, int padding_height, bool is_SoA)
{
    // Calculate the global index in the output image.
    const int x = blockIdx.x * blockDim.x + threadIdx.x; // Column index.
    const int y = blockIdx.y * blockDim.y + threadIdx.y; // Row index.

    // Check if the thread is within the image bounds.
    if(x < width && y < height) {
        for(int channel = 0; channel < channels; channel++) {
            // Output value for the current pixel.
            float output_value = 0.0f;

            // Iterate over the kernel.
            for(int ky = 0; ky < kernel_height; ky++) {
                for(int kx = 0; kx < kernel_width; kx++) {
                    // Get the pixel index to be convolved.
                    const int col = x + kx - floor((float)kernel_width/2) + padding_width;
                    const int row = y + ky - floor((float)kernel_height/2) + padding_height;
                    // Add the convolution value to the output value.
                    output_value += get_pixel_value(d_input, col, row, channel, padded_width,
                                                    padded_height, channels, is_SoA) * get_kernel_value(d_kernel, kx, ky,
                                                    kernel_width, kernel_height);
                }
            }

            // Store the output value in global memory.
            set_pixel_value(d_output, x, y, channel, width, height, channels, is_SoA,
                           (uint8_t)clamp(0.0f, output_value, 255.0f));
        }
    }
}
```

```
__global__ void convolution_kernel_global(uint8_t* d_input, float* d_kernel, uint8_t* d_output,
                                         int width, int height, int channels,
                                         int kernel_width, int kernel_height,
                                         int padding_width, int padding_height, bool is_SoA)
{
    // Calculate the global index in the output image.
    const int x = blockIdx.x * blockDim.x + threadIdx.x; // Column index.
    const int y = blockIdx.y * blockDim.y + threadIdx.y; // Row index.

    // Check if the thread is within the image bounds.
    if(x < width && y < height) {
        for(int channel = 0; channel < channels; channel++) {
            // Output value for the current pixel.
            float output_value = 0.0f;

            // Iterate over the kernel.
            for(int ky = 0; ky < kernel_height; ky++) {
                for(int kx = 0; kx < kernel_width; kx++) {
                    // Get the pixel index to be convolved.
                    const int col = x + kx - floor((float)kernel_width/2) + padding_width;
                    const int row = y + ky - floor((float)kernel_height/2) + padding_height;
                    // Add the convolution value to the output value.
                    output_value += get_pixel_value(d_input, col, row, channel, padded_width,
                                                    padded_height, channels, is_SoA) * get_kernel_value(d_kernel, kx, ky,
                                                    kernel_width, kernel_height);
                }
            }

            // Store the output value in global memory.
            set_pixel_value(d_output, x, y, channel, width, height, channels, is_SoA,
                           (uint8_t)clamp(0.0f, output_value, 255.0f));
        }
    }
}
```

```
__global__ void convolution_kernel_global(uint8_t* d_input, float* d_kernel, uint8_t* d_output,
                                         int width, int height, int channels,
                                         int kernel_width, int kernel_height,
                                         int padding_width, int padding_height, bool is_SoA)
{
    // Calculate the global index in the output image.
    const int x = blockIdx.x * blockDim.x + threadIdx.x; // Column index.
    const int y = blockIdx.y * blockDim.y + threadIdx.y; // Row index.

    // Check if the thread is within the image bounds.
    if(x < width && y < height) {
        for(int channel = 0; channel < channels; channel++) {
            // Get the 1D pixel index.
            const int pixel_index = is_SoA ?
                ((channel * width * height) + (row * width) + col) :
                ((row * width + col) * channels + channel);

            return d_input[pixel_index];
        }
    }

    get_pixel_value(d_input, col, row, channel, padded_width, padded_height, channels, is_SoA)
    * get_kernel_value(d_kernel, kx, ky, kernel_width, kernel_height);
}

// Store the output value in global memory.
set_pixel_value(d_output, x, y, channel, width, height, channels, is_SoA,
                (uint8_t)clamp(0.0f, output_value, 255.0f));
}
```

```
__global__ void convolution_kernel_global(uint8_t* d_input, float* d_kernel, uint8_t* d_output,
                                         int width, int height, int channels,
                                         int kernel_width, int kernel_height,
                                         int padding_width, int padding_height, bool is_SoA)
{
    // Calculate the global index in the output image.
    const int x = blockIdx.x * blockDim.x + threadIdx.x; // Column index.
    const int y = blockIdx.y * blockDim.y + threadIdx.y; // Row index.

    // Check if the thread is within the image bounds.
    if(x < width && y < height) {
        for(int channel = 0; channel < channels; channel++) {
            // Output value for the current pixel.
            float output_value = 0.0f;

            // Iterate over the kernel.
            for(int ky = 0; ky < kernel_height; ky++) {
                for(int kx = 0; kx < kernel_width; kx++) {
                    // Get the pixel index to be convolved.
                    const int col = x + kx - floor((float)kernel_width/2) + padding_width;
                    const int row = y + ky - floor((float)kernel_height/2) + padding_height;
                    // Add the convolution value to the output value.
                    output_value += get_pixel_value(d_input, col, row, channel, padded_width,
                                                    padded_height, channels, is_SoA) * get_kernel_value(d_kernel, kx, ky,
                                                    kernel_width, kernel_height);
                }
            }

            // Store the output value in global memory.
            set_pixel_value(d_output, x, y, channel, width, height, channels, is_SoA,
                           (uint8_t)clamp(0.0f, output_value, 255.0f));
        }
    }
}
```

```
__global__ void convolution_kernel_global(uint8_t* d_input, float* d_kernel, uint8_t* d_output,
                                         int width, int height, int channels,
                                         int kernel_width, int kernel_height,
                                         int padding_width, int padding_height, bool is_SoA)
```

```
{
    // Calculate the global index in the output image.
    const int x = blockIdx.x * blockDim.x + threadIdx.x; // Column index.
    const int y = blockIdx.y * blockDim.y + threadIdx.y; // Row index.
```

```
    // Check if the thread is within the image bounds.
    if(x < width && y < height) {
        for(int channel = 0; channel < channels; channel++) {
            // Output value for the current pixel.
            float output_value = 0.0f;
```

```
            // Iterate over the kernel
```

```
__device__ float& get_kernel_value(float* d_kernel, const int col, const int row, const int width,
const int height) {
```

```
    // Get the 1D kernel index.
    const int kernel_index = (row * width) + col;
```

```
    return d_kernel[kernel_index];
```

```
}
```

```
        * get_kernel_value(d_kernel, kx, ky, kernel_width, kernel_height);
```

```
    }
}
```

```
    // Store the output value in global memory.
    set_pixel_value(d_output, x, y, channel, width, height, channels, is_SoA,
        (uint8_t)clamp(0.0f, output_value, 255.0f));
}
```

- Define a constant memory to store filter (constant variable).

```
__constant__ float c_kernel[MAX_MASK_WIDTH * MAX_MASK_WIDTH];
```

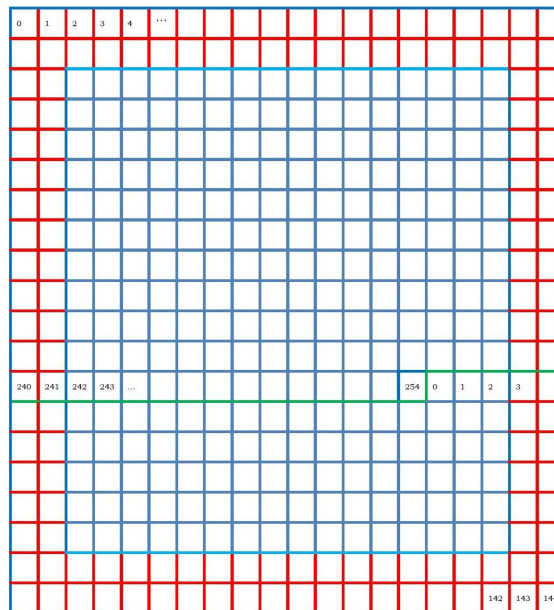
- Similar to global memory kernel (change only the kernel definition for filter parameter and multiplication).

```
__global__ void convolution_kernel_constant(uint8_t* d_input, uint8_t* d_output,  
int width, int height, int channels,  
int kernel_width, int kernel_height,  
int padding_width, int padding_height, bool is_SoA);
```

```
output_value += get_pixel_value(d_input, col, row, channel, padded_width, padded_height,  
channels, is_SoA) * get_kernel_value(c_kernel, kx, ky, kernel_width, kernel_height);
```

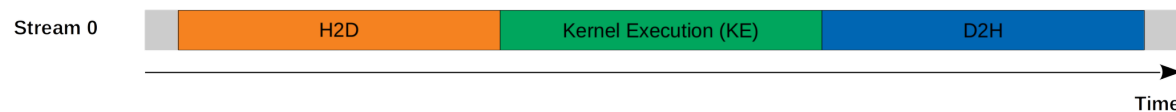


- In addition to using constant memory for fast access to the kernel filter, we also use shared memory among threads in a block to optimize access to shared data.
- Using **tiling** technique to store reusable data from global memory in shared memory (fast access but small dimension).
- A block of TILE_WIDTH^2 threads works on a data tile of size $(\text{TILE_WIDTH} + \text{kernel_width} - 1)^2$, considering boundary elements.

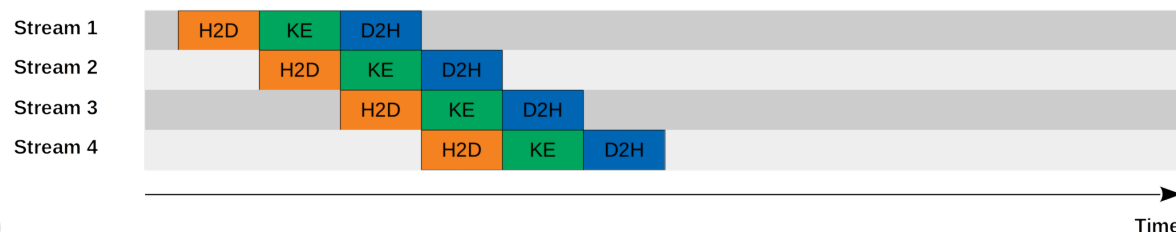


- Primary cost in CUDA implementation is data transfer between host and device.
- Explore asynchronous solution using **pinned memory** and **CUDA streams**:
 - Pinned memory for faster host-device communication. It is memory physically locked in RAM, which the operating system cannot access.
 - Partition kernel workloads with CUDA streams.
 - Transfer data concurrently while a kernel is in execution.
 - Reduces idle time.

Serial Model



Concurrent Model



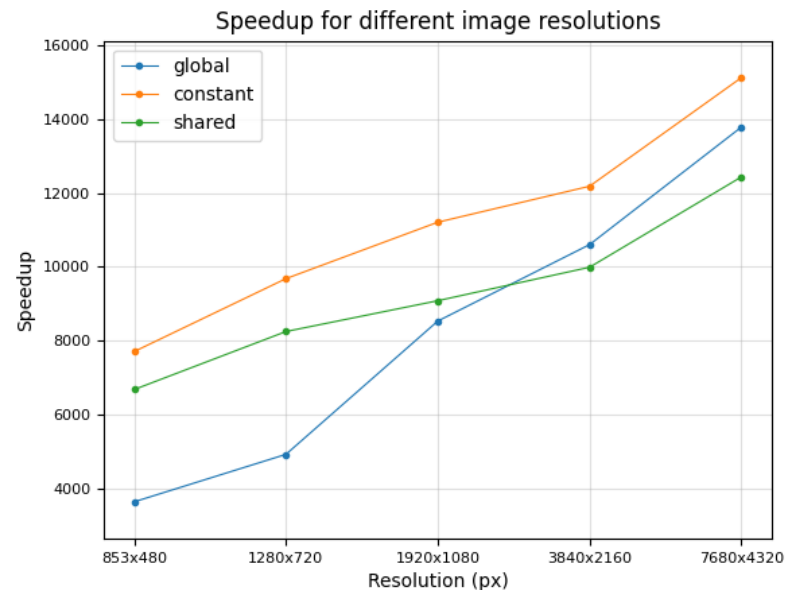
Performance

- Analysis of speedup as the input image resolutions and filter sizes change:
 1. Images 480p, 720p, HD, 4k and 8K, with a Gaussian Blur 3×3 filter.
 2. Filters 3×3 , 5×5 , 7×7 and 9×9 , with HD image.
- Comparison of memory levels.
- Comparison of AoS and SoA architectures.
- Comparison of synchronous and asynchronous loading models.



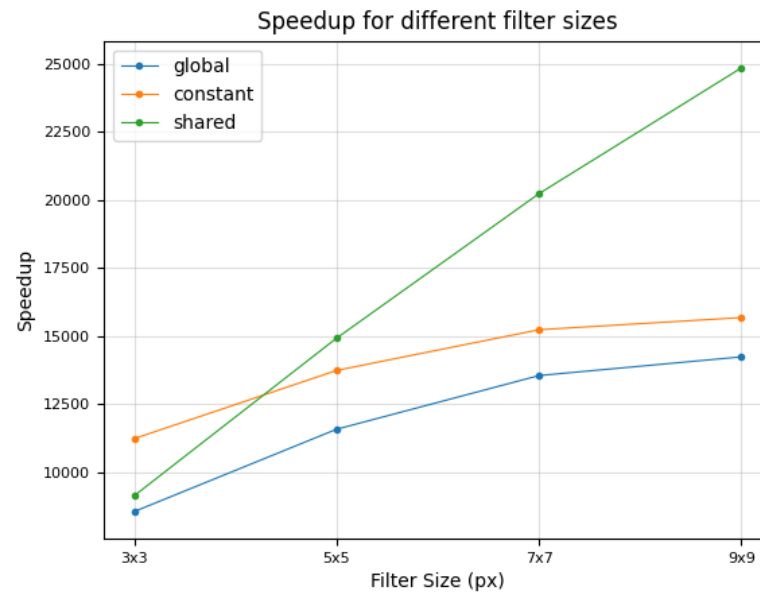
- Speedup as the resolution changes:

| | Global | Constant | Shared |
|-------------|----------|-----------------|---------|
| 480p | 3645.86 | 7709.04 | 6682.36 |
| 720p | 4929.97 | 9681.91 | 8251.72 |
| HD | 8534.28 | 11207.86 | 9084.96 |
| 4K | 10602.63 | 12181.03 | 9990.22 |
| 8K | 13780.7 | 15118.17 | 12433 |

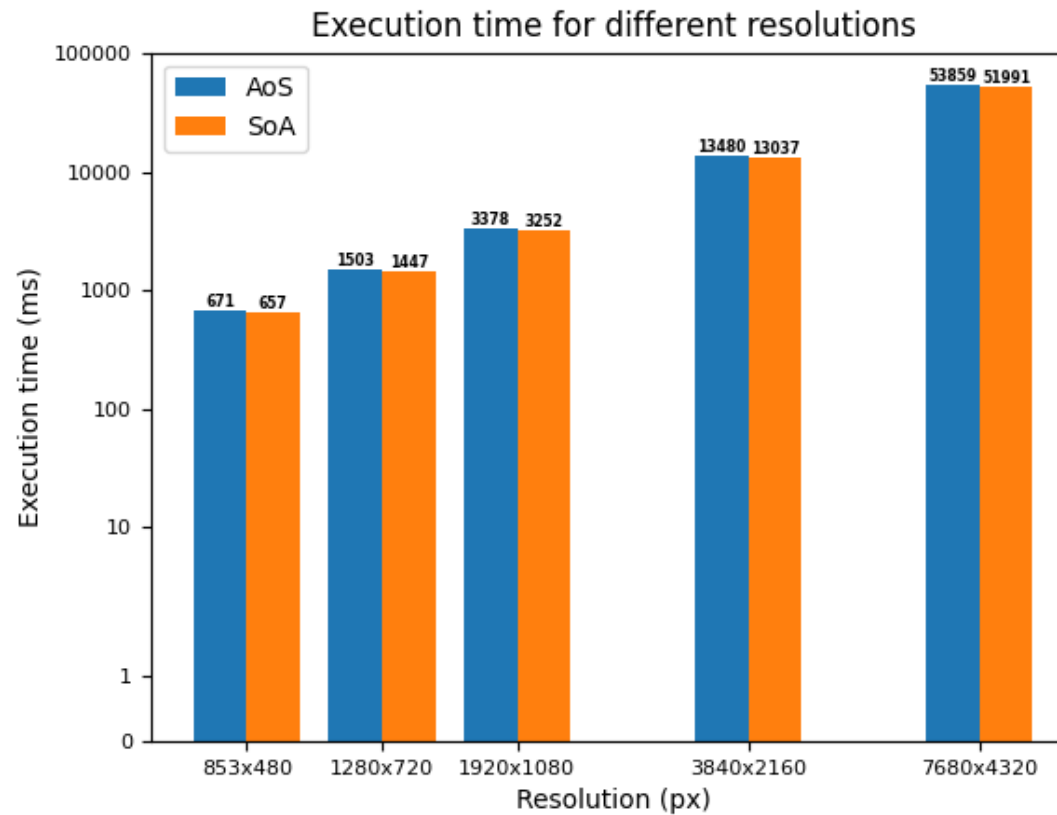


- Speedup as the filter sizes changes:

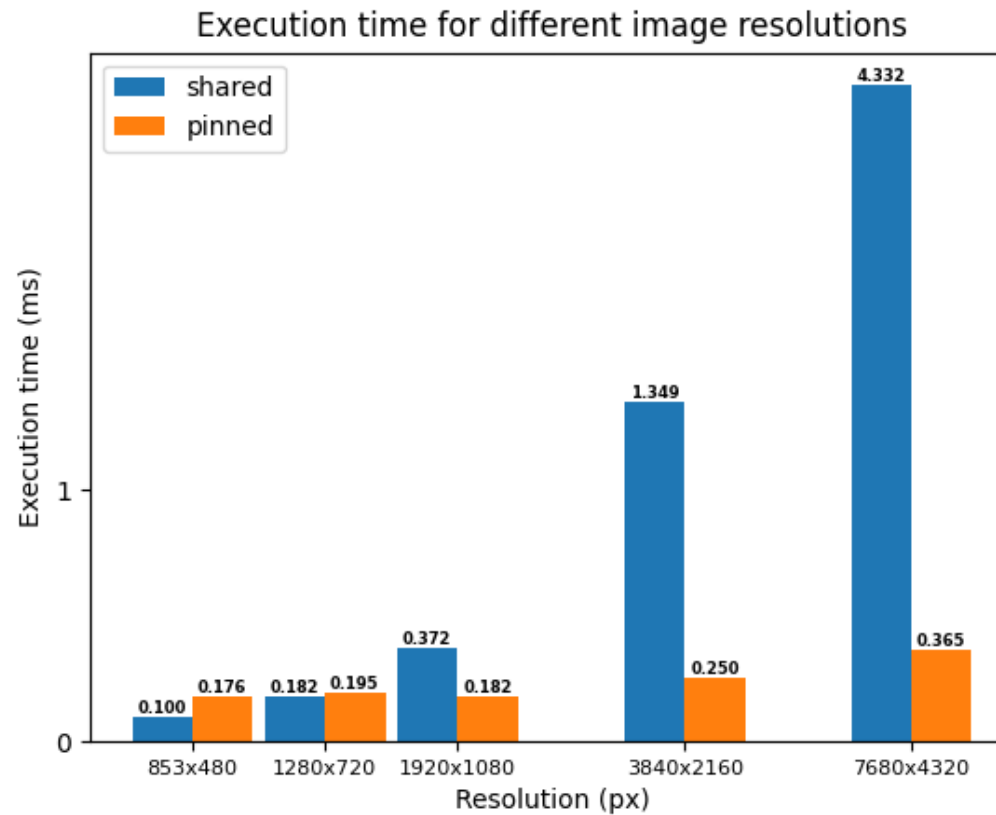
| | Global | Constant | Shared |
|--------------|----------|----------|-----------------|
| 3×3 | 8554.41 | 11232.12 | 9133.13 |
| 5×5 | 11578.48 | 13738.88 | 14922.6 |
| 7×7 | 13550.8 | 15232.74 | 20230.6 |
| 9×9 | 14236.82 | 15679.22 | 24850.18 |



- AoS vs SoA:



Synchronous vs asynchronous loading models:



Conclusions

- Parallel code results in higher performance than sequential code.
- CUDA can exploit the increased number of threads to efficiently perform the convolution operation on large images, with benefits that increase with the number of pixels and kernel size.
- Constant and shared memory can reduce the kernel execution time.
- SoA architecture can reduce the execution time.
- Asynchronous loading greatly reduces computation time.

