Kernel Image Processing
Parallel Computing 2025-2026 — Final Assignment

Mattia Manneschi
mattia.manneschi@edu.unifi.it

Abstract

This assignment demonstrates how to implement kernel-based image processing using GPU parallelization with CUDA. The project implements various convolution filters (Gaussian blur, sharpen, edge detection, emboss) and compares the performance of sequential CPU execution against three CUDA implementations with different memory optimization strategies.

Future Distribution Permission

1. Introduction

This project implements an image processing system based on convolution operations, leveraging the parallel computing power offered by NVIDIA GPUs through the CUDA platform. The main objective is to demonstrate the benefits of GPU parallelization compared to sequential CPU processing for computationally intensive operations such as image filtering.

Convolution is a fundamental operation in digital image processing, used to apply filters such as blur, sharpen, edge detection, and emboss effects. Each pixel in the output image is calculated as a weighted linear combination of neighboring pixels in the input image, where the weights are defined by a matrix called kernel. The dataset used during the experiments is the Kodak dataset [3].

1.1. Project Objectives

- Implement a sequential CPU version as baseline for performance comparison.
- Develop three CUDA implementations with different levels of memory optimization.
- Conduct a complete performance analysis with various image and kernel sizes.
- Achieve significant speedups compared to the sequential CPU implementation.
- Provide visual examples of filters applied to real images.

1.2. Hardware Used

Tests were executed on an NVIDIA GeForce GTX 1080 GPU with Pascal architecture (SM 6.1), 2560 CUDA cores, 8 GB GDDR5X memory, 320 GB/s bandwidth, 1607 MHz base clock, and Compute Capability 6.1.

Specification	Value
Architecture	Pascal (SM 6.1)
CUDA Cores	2560
Memory	8 GB GDDR5X
Bandwidth	320 GB/s
Base Clock	1607 MHz
Compute Capability	6.1

Table 1. GPU hardware specifications

2. Theoretical Background

2.1. Convolution Operation

2D convolution is mathematically defined as:

$$g(x,y) = \sum_i \sum_j f(x+i, y+j) \cdot \omega(i,j) \quad (1)$$

where g is the filtered image, f is the original image, and ω is the convolution kernel. For each pixel, the operation requires $K \times K$ multiplications and sums, where K is the kernel dimension.

2.2. Implemented Convolution Kernels

The project implements multiple kernels based on Wikipedia's Kernel Image Processing documentation [1]: Gaussian Blur (3×3, 5×5, 7×7), Box Blur (3×3, 5×5, 7×7), Sharpen (3×3), Sobel X/Y (3×3), Prewitt X/Y (3×3), Laplacian (3×3), and Emboss (3×3).

Kernel	Size
Gaussian Blur	3×3, 5×5, 7×7
Box Blur	3×3, 5×5, 7×7
Sharpen	3×3
Sobel X/Y	3×3
Prewitt X/Y	3×3
Laplacian	3×3
Emboss	3×3

Table 2. Implemented convolution kernels

2.3. CUDA Architecture

CUDA (Compute Unified Device Architecture) is a parallel computing platform developed by NVIDIA. The CUDA architecture organizes threads in a three-level hierarchy: Grid (set of all blocks executing a kernel), Block (group of threads that can cooperate via shared memory), and Thread (minimum execution unit, each thread processes one or more pixels). The GTX 1080 can handle up to 1024 threads per block and up to 2048 active threads per Streaming Multiprocessor.

3. Implementation

3.1. Project Structure

The project is organized in modules: `main.cpp` (entry point and CLI parsing), `cpu_convolution.cpp` (sequential CPU implementation), `gpu_convolution.cu` (3 CUDA versions), `kernels.cpp` (convolution kernel definitions), `image_io.cpp` (image I/O using stb library), `benchmark.cpp` (automated benchmarking system), and `advanced_tests.cpp` (advanced performance tests).

3.2. CPU Implementation (Baseline)

The sequential implementation was developed to run entirely on the CPU and serves as the primary performance baseline for the project. The convolution process is managed through four nested iterative loops that scan the spatial coordinates of the image and, for each pixel, the entire kernel window. From an architectural standpoint, this version is heavily penalized by the access latency of the system RAM, as every single pixel requires multiple redundant reads to fetch the neighbor values necessary for the calculation. The lack of parallelism makes this solution highly inefficient as the image resolution or the filter size increases, yet it establishes the necessary reference point to quantify the speedup achieved by the subsequent accelerated versions.

```
for (int y = 0; y < height; y++) {
  for (int x = 0; x < width; x++) {
    for (int c = 0; c < channels; c++) {
      float sum = 0.0f;
      for (int ky = -half; ky <= half; ky++) {
        for (int kx = -half; kx <= half; kx++) {
          int px = clamp_coord(x + kx, 0,
              width - 1);
          int py = clamp_coord(y + ky, 0,
              height - 1);
          int img_idx = (py * width + px)
              * channels + c;
          int k_idx = (ky + half) * kernel_size
              + (kx + half);
          sum += static_cast<float>(
              input[img_idx]) * kernel[k_idx];
        }
      }
      int out_idx = (y * width + x)
          * channels + c;
      output[out_idx] = static_cast<uint8_t>(
          std::max(0.0f,
            std::min(255.0f, sum)));
    }
  }
}
```

3.3. CUDA Implementations

Three CUDA versions with increasing optimization levels were developed.

3.3.1 Global Memory

The first parallel version leverages the CUDA architecture to distribute the workload across thousands of threads, mapping each individual pixel of the image to a specific thread within a two-dimensional grid. In this configuration, both the image data and the filter coefficients reside in the

GPU's global memory. Although a drastic reduction in computation time is achieved compared to the CPU, the system remains heavily memory-bound. Since adjacent threads require the same pixels to complete the convolution, the global memory is stressed by redundant reads, highlighting the need for a more sophisticated management of the memory hierarchy to eliminate data transfer bottlenecks.

```
__global__ void convolve_kernel_global(
    const uint8_t* __restrict__ input,
    uint8_t* __restrict__ output,
    int width, int height, int channels,
    const float* __restrict__ kernel,
    int kernel_size)
{
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;
  if (x >= width || y >= height) return;
  int half = kernel_size / 2;
  for (int c = 0; c < channels; c++) {
    float sum = 0.0f;
    for (int ky = -half; ky <= half; ky++) {
      for (int kx = -half; kx <= half; kx++) {
        int px = min(max(x + kx, 0), width - 1);
        int py = min(max(y + ky, 0), height - 1);
        int img_idx = (py * width + px)
            * channels + c;
        int k_idx = (ky + half) * kernel_size
            + (kx + half);
        sum += static_cast<float>(input[img_idx])
            * kernel[k_idx];
      }
    }
    int out_idx = (y * width + x) * channels + c;
    output[out_idx] = static_cast<uint8_t>(
        fminf(fmaxf(sum, 0.0f), 255.0f));
  }
}
```

3.3.2 Constant Memory

To improve the efficiency of fetching filter coefficients, a version was implemented that allocates the convolution kernel in the GPU's constant memory. This specialized memory is equipped with a dedicated cache on each Streaming Multiprocessor (SM) and is optimized for read-only operations common to all threads within a warp. Since every thread reads the same coefficient simultaneously during the convolution phase, the GPU can perform a broadcast of the data, serving all threads in a single clock cycle with near-zero latency. This optimization significantly reduces traffic on the global memory, allowing the memory controller to dedicate more bandwidth to loading the image pixels.

```
#define MAX_KERNEL_SIZE 7
#define MAX_KERNEL_ELEMENTS \
    (MAX_KERNEL_SIZE * MAX_KERNEL_SIZE)

__constant__ float d_kernel[MAX_KERNEL_ELEMENTS];

__global__ void convolve_kernel_constant(
    const uint8_t* __restrict__ input,
    uint8_t* __restrict__ output,
    int width, int height, int channels,
    int kernel_size)
{
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;
  if (x >= width || y >= height) return;
  int half = kernel_size / 2;
  for (int c = 0; c < channels; c++) {
    float sum = 0.0f;
    for (int ky = -half; ky <= half; ky++) {
      for (int kx = -half; kx <= half; kx++) {
        int px = min(max(x + kx, 0), width - 1);
        int py = min(max(y + ky, 0), height - 1);
        int img_idx = (py * width + px)
            * channels + c;
        int k_idx = (ky + half) * kernel_size
            + (kx + half);
        sum += static_cast<float>(input[img_idx])
            * d_kernel[k_idx];
      }
    }
    int out_idx = (y * width + x) * channels + c;
    output[out_idx] = static_cast<uint8_t>(
        fminf(fmaxf(sum, 0.0f), 255.0f));
  }
}
```

3.3.3 Shared Memory

The most advanced implementation is based on the use of Shared Memory to minimize global memory accesses. The image is divided into blocks or "tiles," which are cooperatively loaded by the threads of a block into the on-chip SRAM, which offers speeds comparable to those of registers. To correctly handle the boundaries of each tile, a "Halo loading" strategy was implemented, where boundary threads load an extra rim of pixels equal to the kernel radius. Through the use of the `__syncthreads()` synchronization barrier, the system ensures that all data is available before the computation begins. Once the shared memory is populated, each pixel is read from global memory only once (or nearly so), bringing the kernel's efficiency close to the theoretical limits of the hardware.

```
template<int BLOCK_SIZE, int KERNEL_RADIUS>
__global__ void convolve_kernel_shared(
    const uint8_t* __restrict__ input,
    uint8_t* __restrict__ output,
    int width, int height, int channels,
    int kernel_size)
{
  const int TILE_SIZE = BLOCK_SIZE
      + 2 * KERNEL_RADIUS;
```

```
 __shared__ float tile[TILE_SIZE][TILE_SIZE];
 int tx = threadIdx.x;
 int ty = threadIdx.y;
 int x = blockIdx.x * BLOCK_SIZE + tx;
 int y = blockIdx.y * BLOCK_SIZE + ty;
 int half = kernel_size / 2;

 for (int c = 0; c < channels; c++) {
  int shared_x = tx + KERNEL_RADIUS;
  int shared_y = ty + KERNEL_RADIUS;

  // Load central tile
  if (x < width && y < height) {
   tile[shared_y][shared_x] =
     static_cast<float>(
      input[(y * width + x) * channels + c]);
  } else {
   tile[shared_y][shared_x] = 0.0f;
  }

  // Load left halo
  if (tx < KERNEL_RADIUS) {
   int load_x = max(0,
     (int)(blockIdx.x * BLOCK_SIZE)
     - KERNEL_RADIUS + tx);
   int load_y = min(max(y, 0), height - 1);
   tile[shared_y][tx] = static_cast<float>(
     input[(load_y * width + load_x)
      * channels + c]);
  }

  // Load right halo
  if (tx >= BLOCK_SIZE - KERNEL_RADIUS) {
   int load_x = min(
     (int)(blockIdx.x * BLOCK_SIZE
      + BLOCK_SIZE + tx
      - (BLOCK_SIZE - KERNEL_RADIUS)),
     width - 1);
   int load_y = min(max(y, 0), height - 1);
   tile[shared_y][BLOCK_SIZE + KERNEL_RADIUS
     + tx - (BLOCK_SIZE - KERNEL_RADIUS)] =
     static_cast<float>(
      input[(load_y * width + load_x)
       * channels + c]);
  }

  // Load top halo
  if (ty < KERNEL_RADIUS) {
   int load_x = min(max(x, 0), width - 1);
   int load_y = max(0,
     (int)(blockIdx.y * BLOCK_SIZE)
     - KERNEL_RADIUS + ty);
   tile[ty][shared_x] = static_cast<float>(
     input[(load_y * width + load_x)
      * channels + c]);
  }

  // Load bottom halo
  if (ty >= BLOCK_SIZE - KERNEL_RADIUS) {
   int load_x = min(max(x, 0), width - 1);
   int load_y = min(
     (int)(blockIdx.y * BLOCK_SIZE
      + BLOCK_SIZE + ty
      - (BLOCK_SIZE - KERNEL_RADIUS)),
     height - 1);
   tile[BLOCK_SIZE + KERNEL_RADIUS + ty
     - (BLOCK_SIZE - KERNEL_RADIUS)][shared_x]=
     static_cast<float>(
      input[(load_y * width + load_x)
       * channels + c]);
  }

  __syncthreads();

  // Compute convolution
  if (x < width && y < height) {
   float sum = 0.0f;
   #pragma unroll
   for (int ky = -half; ky <= half; ky++) {
    #pragma unroll
    for (int kx = -half; kx <= half; kx++) {
     int k_idx = (ky + half) * kernel_size
       + (kx + half);
     sum += tile[ty + KERNEL_RADIUS + ky]
          [tx + KERNEL_RADIUS + kx]
       * d_kernel[k_idx];
    }
   }
   output[(y * width + x) * channels + c] =
     static_cast<uint8_t>(
      fminf(fmaxf(sum, 0.0f), 255.0f));
  }
  __syncthreads();
 }
}
```

3.4. Border Handling

For pixels at image borders, where the kernel extends beyond boundaries, the clamping strategy is used: pixels outside the image assume the value of the nearest border pixel. This choice avoids visual artifacts and preserves the original image dimensions.

4. Experimental Results

4.1. Test Methodology

Benchmarks were executed using the Kodak dataset, a standard set of 24 high-quality photographic images. For each configuration, 10 iterations were performed, excluding the first (warm-up) and calculating mean and standard deviation of the remaining ones.

Tested configurations include: image sizes 256×256, 512×512, 1024×1024, 2048×2048, 4096×4096; kernel sizes 3×3, 5×5, 7×7; CUDA block sizes 8×8, 16×16, 32×32; kernel types Gaussian, Box, Sobel, Laplacian, Sharpen, Emboss.

4.2. Implementation Comparison

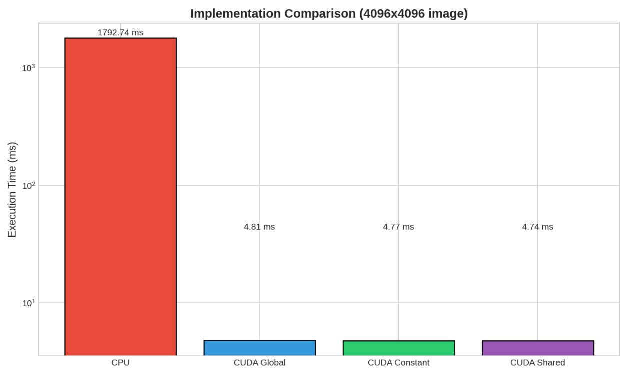

Figure 1. CPU vs CUDA execution time comparison (4096×4096)

The CPU implementation requires about 1793 ms, while all CUDA versions complete the operation in less than 5 ms, achieving an improvement of over 350 times.

4.3. Speedup vs Image Size

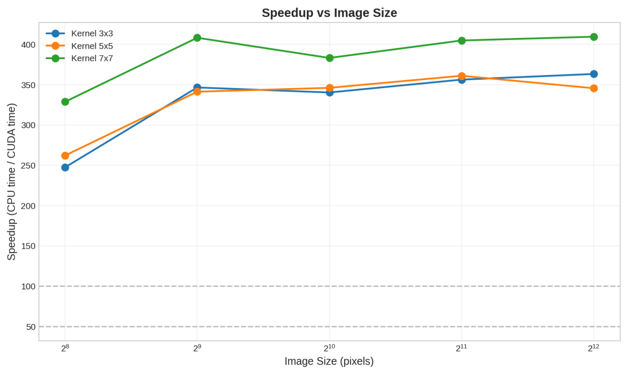

Figure 2. Speedup as a function of image size

Speedup increases with image size, stabilizing around 350-370x for large images. This behavior is expected: larger images allow better exploitation of the GPU's massive parallelism, amortizing data transfer overhead.

4.4. Speedup vs Kernel Size

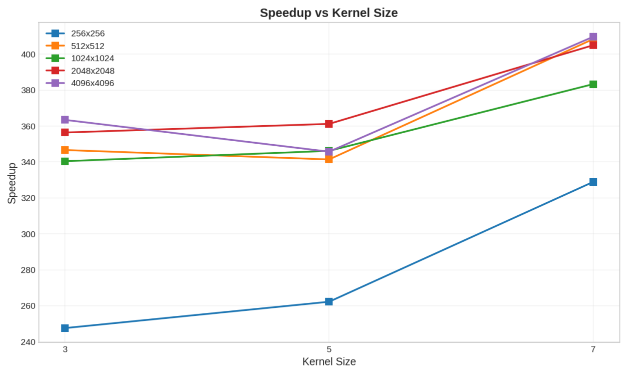

Figure 3. Speedup as a function of kernel size

Speedup increases significantly with larger kernels (up to 410x for 7×7 kernels). This is because larger kernels increase computational work per pixel ($O(K^2)$), favoring the parallel approach over the sequential one.

4.5. Speedup Heatmap

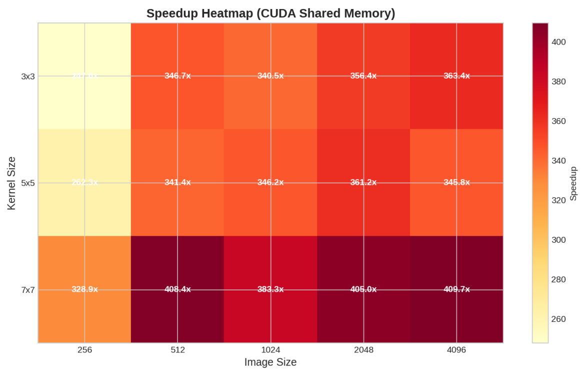

Figure 4. Speedup heatmap (CUDA Shared Memory)

The heatmap shows how speedup varies as a function of both image and kernel size. The highest values (over 400x) are obtained with large images and 7×7 kernels.

4.6. Throughput

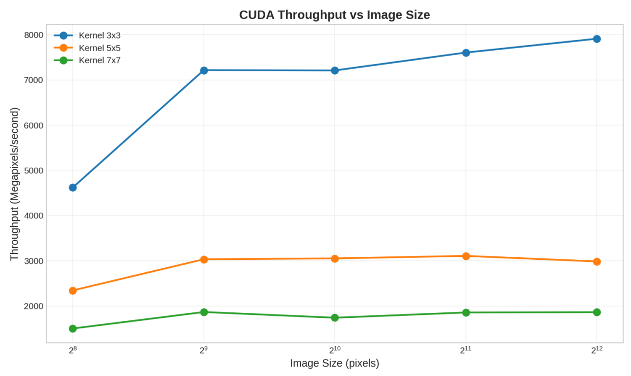

Figure 5. Throughput in Megapixel/second

CUDA throughput reaches approximately 8000 Megapixel/second for 3×3 kernels, enabling real-time 4K image processing (about 3 ms per frame). With larger kernels, throughput decreases but remains in the order of gigapixels per second.

4.7. Block Size Analysis

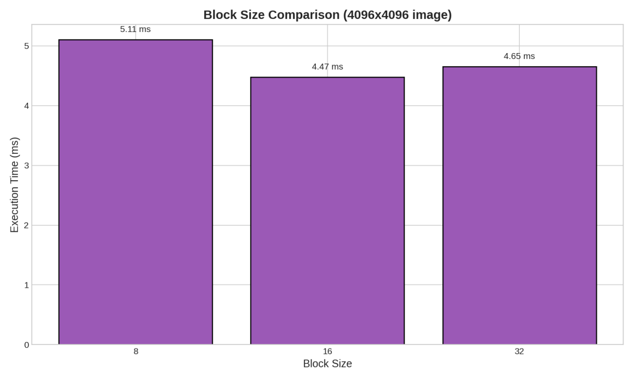

Figure 6. Execution time comparison by block size

Block size analysis shows that 16×16 configurations offer the best performance on the GTX 1080. Blocks too small (8×8) don't fully exploit occupancy, while blocks too large (32×32) can limit the number of active blocks per SM.

4.8. Performance Summary

Metric	Value
Maximum Speedup	447x (Sobel 3×3, 4096×4096)
Average Speedup	345x
Maximum Throughput	8000 MP/s
4K Processing Time (Gaussian 5×5)	3.11 ms
Optimal Block Size	16×16

Table 3. Performance summary

5. Visual Filter Examples

The following images show the effects of various filters applied to a Kodak dataset image. All filters were processed with the CUDA Shared Memory implementation.

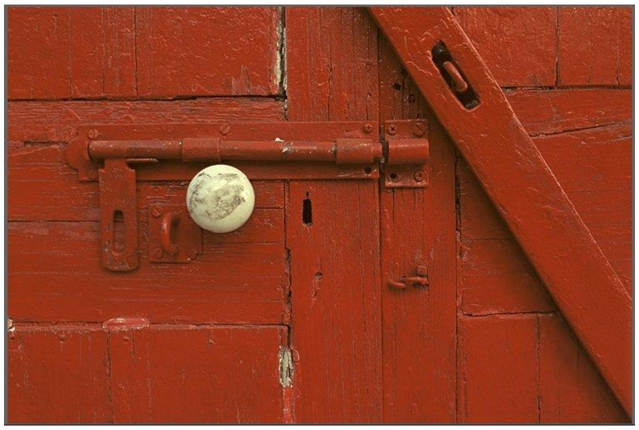

Figure 7. Original image

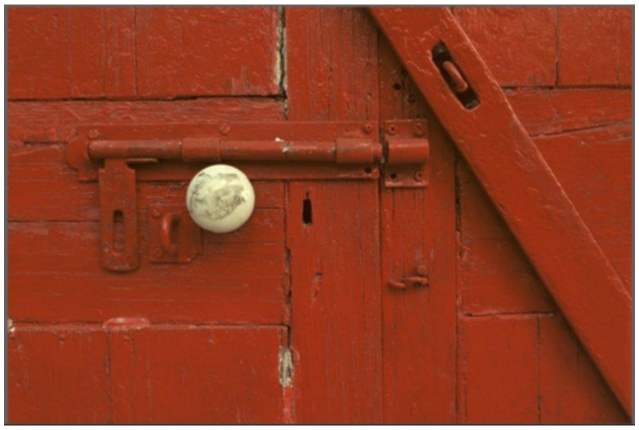

Figure 8. Gaussian Blur 5×5

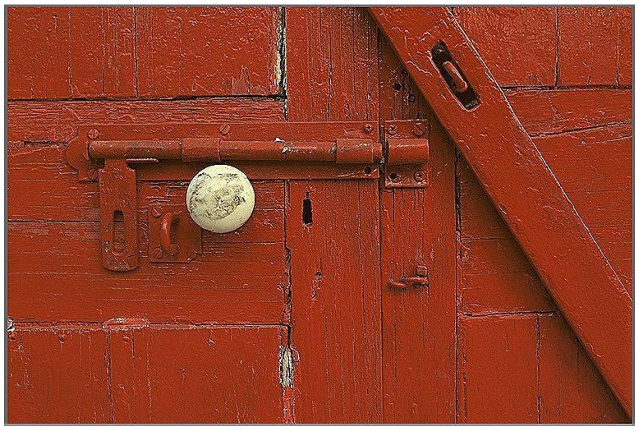

Figure 9. Sharpen filter

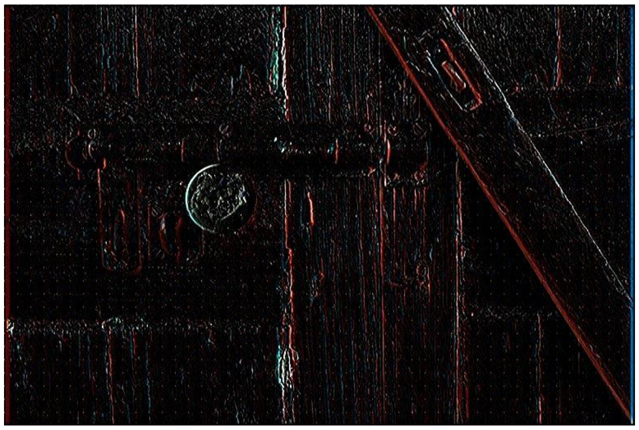

Figure 10. Sobel X (vertical edges)

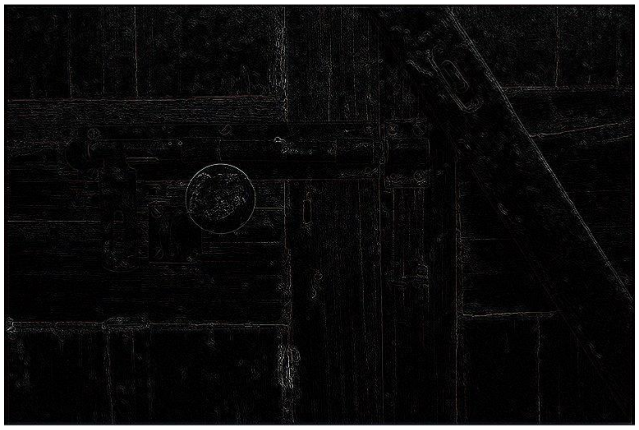

Figure 11. Laplacian filter

6. Conclusions

The project demonstrated the effectiveness of GPU parallelization for image convolution operations. The main results can be summarized as follows.

6.1. Results Analysis

The three CUDA implementations show very similar performance, with differences below 1%. This suggests that for this type of workload, memory bandwidth is not the main bottleneck, and shared/constant memory optimizations have limited impact. However, the speedup compared to CPU is dramatic (>300x), confirming GPU suitability for this type of processing.

Speedup increases with image and kernel size, reaching optimal values for images $\geq$1024$\times$1024 and kernels $\geq$5$\times$5. For small images (256$\times$256), kernel launch overhead and data transfer reduce parallelization benefits.

6.2. Future Developments

- Implementation of separable convolution for Gaussian kernels (reduction from $O(K^2)$ to $O(2K)$)
- Use of CUDA Streams for computation and data transfer overlap
- Support for batch processing of multiple images
- Implementation of more complex kernels (bilateral filter, non-local means)
- Porting to other platforms (OpenCL, Metal, Vulkan Compute)

6.3. Final Considerations

The project confirms that the GPU is the ideal tool for parallel image processing. With a speedup of over 300 times, operations that would require seconds on CPU can be completed in milliseconds, paving the way for real-time applications such as video processing, computer vision, and machine learning.

References

[1] Wikipedia - Kernel (image processing). `https://en.wikipedia.org/wiki/Kernel_(image_processing)`

[2] NVIDIA CUDA C++ Programming Guide. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`

[3] Kodak Lossless True Color Image Suite. `http://r0k.us/graphics/kodak/`

[4] stb_image library. `https://github.com/nothings/stb`

[5] Project Repository. `https://github.com/MattiaManneschi/Kernel-Image-Processing`