Università degli Studi di Camerino

Scuola di Scienze e Tecnologie

Corso di Parallel and Distributed Computing

CUDA-Optimized 2D Convolution: A Study in GPU Parallel Computing

Students
Kiril Buga, Yannick Schmid

Supervisers
Andrea Polini

Ahmed Umair

A.A. 2025/2026

Indice

1. Introduction

1.1 Motivation

Modern image processing and computer vision workloads involve applying the same mathematical operation to millions of pixels. A 4K image contains over 8 million pixels, and even a simple 3×3 filter requires 9 multiply-accumulate operations per pixel, totaling over 75 million floating-point operations for a single pass. On a conventional CPU, these operations execute sequentially, making large-scale image processing slow and impractical for real-time applications.

Graphics Processing Units (GPUs) offer a fundamentally different computing model. Where a CPU optimizes for single-thread latency with a handful of powerful cores, a GPU provides thousands of lightweight cores optimized for throughput. 2D convolution is an ideal candidate for GPU acceleration because every output pixel can be computed independently, making it an embarrassingly parallel workload. This project explores how to systematically exploit this parallelism using NVIDIA's CUDA platform, progressing from a naive implementation to increasingly optimized versions that leverage the GPU's memory hierarchy and execution model.

1.2 Objectives

The objectives of this project are:

- Implement a correct sequential CPU baseline for 2D convolution with zero-padding boundary handling.
- Develop three progressively optimized CUDA kernels: a naive global memory version (V1), a constant memory version (V1_const), and a shared memory tiled version (V2).
- Benchmark performance across three dimensions: image size, filter kernel size, and thread block configuration.
- Verify the correctness of all GPU implementations against the CPU reference using quantitative error metrics.

1.3 Report Organization

Chapter 2 introduces the parallel computing concepts that underpin GPU programming, covering architecture, memory hierarchy, and optimization techniques. Chapter 3

provides a concise overview of the image processing operations used as the computational workload. Chapter 4 describes the project's implementation and benchmarking methodology in detail. Chapter 5 presents the experimental results, and Chapter 6 concludes with findings and future work.

2. Background: Parallel Computing

2.1 Why Parallelism?

The traditional CPU architecture is built around a small number of powerful cores (typically 4–16), each equipped with deep pipelines, branch predictors, and large multi-level caches (Hennessy e Patterson, 2017). This design minimizes the latency of individual operations, making CPUs excellent for sequential, control-flow-heavy workloads. However, when the same operation must be applied to millions of independent data elements, the CPU's per-core performance advantage becomes irrelevant—what matters is throughput.

A GPU inverts this trade-off. Instead of a few complex cores, a GPU contains hundreds or thousands of simpler cores grouped into processing units called Streaming Multiprocessors (SMs) (Lindholm et al., 2008). Each core lacks the sophisticated control logic of a CPU core, but the sheer number of cores enables massive data parallelism. The GPU model assumes that if one thread stalls (e.g., waiting for a memory access), another thread is ready to execute immediately, keeping the hardware busy. This is known as latency hiding through occupancy (Hwu et al., 2023).

Amdahl's Law provides the theoretical framework for understanding parallelization gains (Amdahl, 1967). If a fraction p of a program is parallelizable and the remaining fraction $(1 - p)$ is inherently sequential, the maximum speedup with N processors is:

$$\text{Speedup} = \frac{1}{(1-p) + \frac{p}{N}} \tag{2.1}$$

For 2D convolution, the parallel fraction is extremely high: every output pixel is computed independently from its local neighborhood in the input image. The only sequential components are memory allocation, data transfer between host (CPU) and device (GPU), and result collection. This makes convolution an ideal workload for GPU acceleration, with theoretical speedups approaching the ratio of GPU-to-CPU computational throughput.

2.2 GPU Architecture Fundamentals

2.2.1 Streaming Multiprocessors and CUDA Cores

An NVIDIA GPU is organized as an array of Streaming Multiprocessors (SMs) (Lindholm et al., 2008; NVIDIA Corporation, 2024). Each SM is an independent processing unit containing multiple CUDA cores (also called shader processors), a register file,

shared memory, and scheduling logic. This project benchmarks on two GPUs: the NVIDIA GeForce GTX 1050 Ti (Pascal, compute capability 6.1) with 6 SMs and 768 CUDA cores, and the NVIDIA Tesla T4 (Turing, compute capability 7.5) with 40 SMs and 2560 CUDA cores.

When a CUDA kernel is launched, the GPU scheduler distributes thread blocks across the available SMs. Each thread block runs entirely on a single SM and cannot migrate to another. An SM can host multiple thread blocks concurrently, limited by the SM's resources (registers, shared memory, and maximum thread count). This block-to-SM mapping is the fundamental unit of work distribution on the GPU (Nickolls et al., 2008).

2.2.2 Thread Hierarchy: Threads, Warps, Blocks, and Grids

CUDA organizes parallel execution in a four-level hierarchy (NVIDIA Corporation, 2024; Nickolls et al., 2008):

- **Thread:** The smallest unit of execution. Each thread has a unique ID within its block (`threadIdx.x`, `threadIdx.y`) and runs the same kernel code on different data. In this project, each thread typically computes one output pixel.
- **Warp:** A group of 32 threads that execute in lockstep using Single Instruction, Multiple Threads (SIMT) execution (Lindholm et al., 2008). All threads in a warp execute the same instruction at the same time. If threads in a warp take different branches (warp divergence), both paths are serialized, reducing efficiency. The warp is the fundamental scheduling unit on the GPU.
- **Thread Block:** A programmer-defined grouping of threads (e.g., $16 \times 16 = 256$ threads, or $32 \times 8 = 256$ threads). Threads within a block can cooperate through shared memory and synchronize with barriers. Blocks are assigned to SMs and execute independently of other blocks.
- **Grid:** The collection of all thread blocks needed to process the entire input. For a 1024×1024 image with 16×16 thread blocks, the grid is $64 \times 64 = 4{,}096$ blocks, each containing 256 threads, for a total of over one million threads.

In the project's code, the grid and block dimensions are computed as:

```
1 dim3 block(block_x, block_y);
2 dim3 grid((width + block_x - 1) / block_x,
3        (height + block_y - 1) / block_y);
```

Codice 2.1: Grid and block dimension computation

This ceiling-division formula ensures the grid covers the entire image, even when dimensions are not evenly divisible by the block size. Threads that map to positions outside the image boundaries are deactivated with a bounds check.

2.2.3 Memory Hierarchy

The GPU memory hierarchy is central to understanding performance optimization (Hwu et al., 2023; NVIDIA Corporation, 2024). Each level trades capacity for speed:

- **Registers:** The fastest storage, private to each thread. Variables like loop counters and accumulators reside in registers. Access latency is approximately 1 clock cycle (Jia et al., 2018). Both GPUs used in this project provide 65,536 32-bit registers per SM.

- **Shared Memory:** A fast, programmer-managed memory space shared by all threads in a block. It acts as a software-controlled cache. Access latency is roughly 5–10 cycles, making it approximately 100× faster than global memory (Hwu et al., 2023; Jia et al., 2018). The GTX 1050 Ti provides 48 KB of shared memory per SM, while the T4 provides up to 64 KB. This is the key resource exploited by the V2 tiled convolution kernel.

- **Constant Memory:** A 64 KB read-only memory space, cached on each SM. When all threads in a warp read the same address, the value is broadcast to all threads in a single transaction (NVIDIA Corporation, 2024). This makes it ideal for small, read-only data accessed uniformly, such as convolution filter coefficients. This is the key resource exploited by the V1_const kernel and also used by V2.

- **Global Memory:** The largest memory space, accessible by all threads across all SMs. It has the highest latency (400–600 clock cycles). The GTX 1050 Ti has 4 GB at 112 GB/s peak bandwidth, while the T4 has 16 GB at 320 GB/s (Jia et al., 2018). The input and output image arrays reside in global memory. Efficient use of global memory requires coalesced access patterns (see Section 2.3.1).

2.3 Key Optimization Techniques

2.3.1 Memory Coalescing

When threads in a warp access contiguous addresses in global memory, the hardware combines these individual requests into a single wide memory transaction (typically 128 bytes). This is called memory coalescing and is critical for achieving high bandwidth utilization (NVIDIA Corporation, 2024; Ryoo et al., 2008). Conversely, scattered or strided access patterns result in multiple separate transactions, wasting bandwidth.

In 2D image processing, images are stored in row-major order: `pixel = image[y * width + x]`. When the block's x-dimension equals the warp size (32), threads in the same warp process adjacent pixels in the same row, resulting in perfectly coalesced reads. This is why the benchmarks test a 32×8 block configuration, which has the same total thread count (256) as 16×16 but aligns each row of threads with a full warp.

2.3.2 Tiling and Shared Memory

The naive convolution kernel (V1) has a fundamental inefficiency: each thread reads its pixel's entire neighborhood from global memory independently. For a 3×3 kernel, neighboring threads share 6 of their 9 input pixels, but each thread fetches all 9 from global memory. For larger kernels, this redundancy grows rapidly.

Shared memory tiling solves this problem (Hwu et al., 2023; Micikevicius, 2009). The idea is to cooperatively load a tile of input data, including a boundary region called the halo, into shared memory once. Then all threads in the block compute their

output pixels by reading from the fast shared memory instead of slow global memory (Podlozhnyuk, 2007).

The tile dimensions include the halo:

$$\begin{aligned} \texttt{SHARED\_W} &= \texttt{TILE\_W} + 2 \times \texttt{HALF\_K} \\ \texttt{SHARED\_H} &= \texttt{TILE\_H} + 2 \times \texttt{HALF\_K} \end{aligned} \tag{2.2}$$

where $\texttt{TILE\_W} \times \texttt{TILE\_H}$ is the output region (matching the block dimensions) and $\texttt{HALF\_K} = \lfloor \texttt{kernel\_size}/2 \rfloor$ is the halo width on each side. For a 16×16 block with a 3×3 kernel, the shared memory tile is $18 \times 18 = 324$ floats. Each of the 256 threads in the block reads 9 values from shared memory during convolution, yielding a data reuse factor of $(256 \times 9)/324 \approx 7.1\times$. This means each value loaded from global memory is used approximately 7 times, dramatically reducing bandwidth pressure.

The loading phase may require each thread to load more than one element, since the shared memory tile (e.g., $18 \times 18 = 324$ elements) is larger than the number of threads in the block (e.g., 256). The code handles this with a nested loop:

```
const int num_loads_x = (SHARED_W + BLOCK_X - 1) / BLOCK_X;
const int num_loads_y = (SHARED_H + BLOCK_Y - 1) / BLOCK_Y;

for (int ly = 0; ly < num_loads_y; ++ly) {
   for (int lx = 0; lx < num_loads_x; ++lx) {
      // Each thread loads tile[shared_y][shared_x]
      // from global memory
   }
}
```

Codice 2.2: Cooperative tile loading with multiple loads per thread

Pixels outside the image boundary are loaded as zero, implementing zero-padding implicitly.

2.3.3 Constant Memory for Filter Weights

The convolution filter kernel is a small array (9 to 49 floats for 3×3 through 7×7 kernels), is read-only during execution, and is accessed with the same index pattern by all threads. These properties make it a textbook use case for constant memory.

In the project, the filter is stored in a `__constant__` array:

```
__constant__ float c_kernel[MAX_KERNEL_ELEMENTS];
```

Codice 2.3: Constant memory declaration for filter weights

and copied from host memory using `cudaMemcpyToSymbol()` (NVIDIA Corporation, 2024). During convolution, all threads in a warp access `c_kernel[ky * kernel_size + kx]` with the same indices at each step, triggering a single cached read that is broadcast to all 32 threads. This eliminates 31 redundant global memory reads per warp per kernel element. The V1_const kernel exploits this optimization as an intermediate step between the fully naive V1 and the shared memory tiled V2.

2.3.4 Template Specialization and Loop Unrolling

The V2 shared memory kernel is implemented as a C++ template parameterized on `KERNEL_SIZE`:

```
template <int KERNEL_SIZE>
__global__ void conv2d_kernel_v2_shared(...)
```

Codice 2.4: Template declaration for V2 kernel

This allows the compiler to treat halo sizes and convolution loop bounds as compile-time constants. The shared memory is allocated dynamically using `extern __shared__ float tile[]`, with the tile dimensions (including halo) computed at runtime and passed as kernel launch parameters. The convolution loops are fully unrolled with `#pragma unroll`:

```
#pragma unroll
for (int ky = 0; ky < KERNEL_SIZE; ++ky) {
   #pragma unroll
   for (int kx = 0; kx < KERNEL_SIZE; ++kx) {
      acc += tile[(threadIdx.y + ky) * shared_w
                + (threadIdx.x + kx)]
          * c_kernel[ky * KERNEL_SIZE + kx];
   }
}
```

Codice 2.5: Unrolled convolution loop in V2 kernel

Loop unrolling eliminates branch instructions and loop overhead, replacing them with a straight-line sequence of multiply-accumulate operations (NVIDIA Corporation, 2024; Volkov, 2010). For a 3×3 kernel, this produces 9 inline operations with no loop control overhead. The project instantiates 3 template variants (one per supported kernel size: 3, 5, 7), and a runtime dispatch selects the correct one based on the requested kernel size.

2.3.5 Synchronization Barriers

When threads cooperatively load data into shared memory, all threads must finish loading before any thread begins reading. Without synchronization, a thread could read a shared memory location that has not yet been written by another thread, leading to incorrect results.

CUDA provides the `__syncthreads()` intrinsic, which acts as a block-level barrier: execution halts until every thread in the block has reached the barrier (NVIDIA Corporation, 2024). In the V2 kernel, `__syncthreads()` is placed between the loading phase and the computation phase:

```
// Phase 1: All threads cooperatively load tile
// into shared memory
tile[shared_y * shared_w + shared_x] =
   input[global_y * width + global_x];

__syncthreads(); // Barrier: wait for all loads

// Phase 2: Each thread computes convolution
// from shared memory
acc += tile[sy * shared_w + sx]
    * c_kernel[ky * KERNEL_SIZE + kx];
```

Codice 2.6: Synchronization barrier between load and compute phases

This barrier is block-scoped—it does not synchronize across different blocks. Cross-block synchronization requires separate kernel launches or atomic operations.

3. Background: Image Processing

3.1 2D Convolution

2D convolution is a mathematical operation that combines an input image I with a small matrix called a kernel (or filter) K to produce an output image O (Gonzalez e Woods, 2018; Szeliski, 2022). For each pixel in the output, the kernel is centered on the corresponding input pixel, element-wise products are computed between the kernel and the overlapping image region, and the results are summed:

$$O(x, y) = \sum_{k_x, k_y} I(x + k_x, y + k_y) \cdot K(k_x, k_y) \tag{3.1}$$

The kernel slides across every pixel position in the image. Different kernel values produce different effects: smoothing, edge detection, sharpening, and more. All images in this project are single-channel (grayscale), stored as 1D arrays of `float` values in the range $[0, 1]$ using row-major order (`pixel = image[y * width + x]`).

3.2 Zero-Padding Boundary Handling

When the kernel overlaps the edge of the image, some kernel positions fall outside the image boundary. Zero-padding treats these out-of-bounds pixels as having a value of zero (Gonzalez e Woods, 2018; Szeliski, 2022). In the implementation, this is handled with bounds checking:

```
1 if (ix < 0 || ix >= width) continue; // skip: contributes 0
2 if (iy < 0 || iy >= height) continue;
```

Codice 3.1: Zero-padding boundary handling

This is the simplest boundary handling strategy and preserves the output image dimensions (the output is the same size as the input).

3.3 Common Filters

The project implements several standard convolution filters (Gonzalez e Woods, 2018), summarized in Table 3.1.

As an example, the 3×3 Gaussian kernel weights the center pixel most heavily and gradually decreases influence outward:

Filter	**Sizes**	**Purpose**
Gaussian	3×3, 5×5, 7×7	Smoothing / noise reduction (Gonzalez e Woods, 2018)
Sobel X / Y	3×3	Edge detection (gradient) (Sobel e Feldman, 1968; Gonzalez e Woods, 2018)
Laplacian	3×3	Edge detection (2nd-order) (Marr e Hildreth, 1980; Gonzalez e Woods, 2018)
Box Blur	3×3, 5×5	Uniform smoothing (equal weights)
Sharpen	3×3	Edge enhancement
Emboss	3×3	Relief / emboss visual effect
Identity	any odd	Pass-through (correctness testing)

Tabella 3.1: Convolution filters implemented in the project

$$K_{\text{Gaussian}} = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \tag{3.2}$$

The Sobel X kernel, originally proposed by Sobel e Feldman (1968) and widely adopted in image processing (Gonzalez e Woods, 2018), detects vertical edges by computing horizontal intensity differences:

$$K_{\text{Sobel X}} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{3.3}$$

3.4 Why Convolution is a Good Parallel Workload

2D convolution is well suited for GPU parallelization for three reasons (Hwu et al., 2023; Podlozhnyuk, 2007):

1. **Data independence:** Each output pixel depends only on a small local neighborhood in the input image. There are no data dependencies between output pixels, making the operation embarrassingly parallel.

2. **Regular access patterns:** The kernel slides across the image in a predictable pattern, producing regular, structured memory accesses that are amenable to coalescing and tiling.

3. **Scalable arithmetic intensity:** The number of operations per pixel scales as $O(K^2)$ with kernel size K. Larger kernels increase the computation-to-memory ratio, shifting the workload from memory-bound toward compute-bound, which is favorable for GPU execution.

4. Project Methodology and Implementation

4.1 Project Structure and Build System

The project is organized into modular source files with clear separation of concerns:

```
src/
  main.cpp            Benchmark harness, correctness tests, timing
  cpu_convolution.cpp / .h Sequential CPU reference implementation
  cuda_convolution.cu / .cuh GPU kernels (V1, V1_const, V2)
  filters.cpp / .h    Filter coefficient definitions
  image_utils.h       Synthetic image generators and error metrics
Makefile              Build configuration
```

Codice 4.1: Project directory structure

The Makefile supports two build modes:

- **Full CUDA build** (`make all`): compiles both CPU and GPU code using `g++` (C++20) for host code and `nvcc` (C++17) for CUDA device code. The target GPU architecture is configurable via `CUDA_ARCH` (default: 75 for Turing).
- **CPU-only build** (`make cpu_only`): compiles only the CPU code with the `-DCPU_ONLY` preprocessor flag, which conditionally excludes all CUDA code. This allows the project to build and run correctness tests on machines without an NVIDIA GPU.

4.2 CPU Baseline

The CPU implementation serves as the correctness reference against which all GPU versions are validated. It uses a straightforward four-level nested loop: for each output pixel (x, y), iterate over all kernel positions (k_x, k_y), accumulate the product of the input pixel and the corresponding kernel weight, and write the result to the output:

```
for (int y = 0; y < height; ++y) {
   for (int x = 0; x < width; ++x) {
      float acc = 0.0f;
      for (int ky = -half_k; ky <= half_k; ++ky) {
         for (int kx = -half_k; kx <= half_k; ++kx) {
            // bounds check + accumulate
            acc += input[iy * width + ix]
                * kernel[kernel_idx];
         }
      }
      output[y * width + x] = acc;
   }
```

```
13 }
```

Codice 4.2: CPU baseline convolution implementation

The implementation supports arbitrary odd-sized kernels (3×3, 5×5, 7×7) and validates input dimensions at runtime. Out-of-bounds kernel positions are skipped (zero-padding).

4.3 GPU V1: Naive Global Memory Kernel

The first GPU implementation maps one thread to each output pixel. Each thread computes the full convolution independently by reading all required input values directly from global memory:

```
__global__ void conv2d_kernel_v1(
    const float* __restrict__ input,
    float* __restrict__ output,
    int width, int height,
    const float* __restrict__ kernel,
    int kernel_size
) {
    const int x = blockIdx.x * blockDim.x + threadIdx.x;
    const int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x >= width || y >= height) return;

    float acc = 0.0f;
    for (int ky = -half_k; ky <= half_k; ++ky)
        for (int kx = -half_k; kx <= half_k; ++kx)
            acc += input[(y+ky)*width + (x+kx)]
                * kernel[...];

    output[y * width + x] = acc;
}
```

Codice 4.3: V1 naive global memory kernel

The `__restrict__` qualifier tells the compiler that the `input`, `output`, and `kernel` pointers do not alias each other, enabling more aggressive optimizations. Threads that map outside the image bounds exit early.

V1 serves as the baseline for measuring the impact of each subsequent optimization. Its primary limitation is excessive global memory traffic: for a 3×3 kernel, each thread performs 9 global memory reads for both input pixels and filter weights. Adjacent threads read heavily overlapping neighborhoods, but this overlap is not exploited.

4.4 GPU V1_const: Constant Memory Kernel

V1_const is structurally identical to V1—one thread per pixel, no shared memory—but reads filter weights from `__constant__` memory instead of global memory. The filter is copied to the device's constant memory before kernel launch using `cudaMemcpyToSymbol()`.

Because all threads in a warp access the same filter weight at each step (the kernel index depends only on the loop iteration, not the thread ID), constant memory's broadcast mechanism serves all 32 threads from a single cached read. This eliminates redundant global memory traffic for the filter weights while keeping the implementation simple.

V1_const isolates the performance contribution of constant memory caching, providing a data point between the fully naive V1 and the shared memory tiled V2.

4.5 GPU V2: Shared Memory Tiled Kernel

V2 addresses V1's redundant memory accesses through shared memory tiling (described in Section 2.3.2). The algorithm proceeds in two phases:

Phase 1—Cooperative Loading: All threads in the block collaboratively load the tile (including halo) from global memory into shared memory. Each thread may load multiple elements to cover the shared memory region, which is larger than the block.

Phase 2—Computation: After synchronizing with `__syncthreads()`, each thread computes its output pixel by reading only from shared memory (for image data) and constant memory (for filter coefficients). The convolution loops are annotated with `#pragma unroll` for compile-time unrolling.

The kernel is templated on `KERNEL_SIZE`, making convolution loop bounds and halo sizes compile-time constants. Shared memory is allocated dynamically (`extern __shared__`), with tile dimensions computed at runtime from the block configuration. The wrapper function dispatches to the correct template instantiation (kernel sizes 3, 5, or 7) and falls back to V1 for unsupported sizes.

4.6 Benchmarking Methodology

Performance is measured differently for CPU and GPU to account for their distinct execution models:

- **CPU timing:** Uses `std::chrono::high_resolution_clock` to measure wall-clock time. Each benchmark is averaged over 3 iterations.
- **GPU timing:** Uses CUDA events (`cudaEventRecord` / `cudaEventElapsedTime`), which measure time on the GPU's internal clock. This avoids including CPU-side overhead in the measurement. Each benchmark includes 1 warmup iteration (to prime caches and initialize the GPU driver), followed by 10 timed iterations. Results are averaged.

The `CudaTimer` class encapsulates GPU timing:

```
class CudaTimer {
   void start() { cudaEventRecord(start_); }
   void stop() { cudaEventRecord(stop_);
                 cudaEventSynchronize(stop_); }
   float elapsed_ms() const {
      float ms;
      cudaEventElapsedTime(&ms, start_, stop_);
      return ms;
   }
};
```

Codice 4.4: CudaTimer class for GPU timing

All benchmarks use synthetic random noise images generated with a fixed seed (42) for reproducibility. Speedup is computed as CPU_time/GPU_time.

The benchmarks sweep three dimensions:

1. **Image size:** 256×256 to 4096×4096 (Gaussian 3×3, 16×16 blocks)
2. **Kernel size:** 3×3, 5×5, 7×7 (1024×1024 image, 16×16 blocks)
3. **Block configuration:** 8×8, 16×16, 32×8, 32×16, 32×32 (2048×2048 image, Gaussian 5×5)

4.7 Correctness Verification

Correctness is established by comparing GPU outputs against the CPU reference output using the maximum absolute error metric:

```
float max_abs_error(const vector<float>& a,
                    const vector<float>& b) {
   float max_err = 0.0f;
   for (size_t i = 0; i < a.size(); ++i)
      max_err = max(max_err, abs(a[i] - b[i]));
   return max_err;
}
```

Codice 4.5: Maximum absolute error computation

A 64×64 checkerboard test image is convolved with each filter on both CPU and GPU. The test passes if the maximum absolute error is below 10^{-5} for all three GPU kernels (V1, V1_const, V2).

Seven filter configurations are tested: Gaussian $3 \times 3/5 \times 5/7 \times 7$, Sobel X 3×3, Laplacian 3×3, and Box Blur $3 \times 3/5 \times 5$.

5. Results

5.1 Hardware and Software Environment

Benchmarks were run on two distinct GPU platforms to evaluate how architectural differences affect kernel performance.

Property	**Value**
GPU	NVIDIA GeForce GTX 1050 Ti
GPU Architecture	Pascal (`sm_61`)
CUDA Cores	768
GPU Memory	4 GB GDDR5
Memory Bandwidth	112 GB/s
Shared Memory per SM	48 KB
Streaming Multiprocessors	6

Tabella 5.1: Environment A: NVIDIA GeForce GTX 1050 Ti (Pascal)

Property	**Value**
GPU	NVIDIA Tesla T4
GPU Architecture	Turing (`sm_75`)
CUDA Cores	2560
GPU Memory	16 GB GDDR6
Memory Bandwidth	320 GB/s
Shared Memory per SM	64 KB
Streaming Multiprocessors	40

Tabella 5.2: Environment B: NVIDIA Tesla T4 (Turing, Google Colab)

Property	**Value**
CUDA Toolkit Version	12.2
Operating System	Ubuntu 22.04 LTS
Compiler (Host)	`gcc` 11.4.0 (C++20)
Compiler (Device)	`nvcc` 12.2 (C++17)

Tabella 5.3: Common software stack

The T4 has approximately 3.3× more CUDA cores, 2.9× higher memory bandwidth, and 6.7× more SMs than the GTX 1050 Ti, making it representative of a datacenter-class accelerator compared to a consumer desktop GPU.

5.2 Correctness Results

All GPU kernels produce bit-identical output to the CPU baseline (max absolute error = 0.00), as shown in Table 5.4.

Filter	**Size**	**V1**	**V1_const**	**V2**	**Status**
Gaussian	3×3	PASS	PASS	PASS	PASS
Gaussian	5×5	PASS	PASS	PASS	PASS
Gaussian	7×7	PASS	PASS	PASS	PASS
Sobel X	3×3	PASS	PASS	PASS	PASS
Laplacian	3×3	PASS	PASS	PASS	PASS
Box Blur	3×3	PASS	PASS	PASS	PASS
Box Blur	5×5	PASS	PASS	PASS	PASS

Tabella 5.4: Correctness verification results across all GPU kernels and filter configurations

All seven filter configurations pass correctness verification across all three GPU kernel versions, confirming that the constant memory and shared memory optimizations do not introduce numerical errors.

5.3 Image Size Sweep (Gaussian 3×3, 16×16 Blocks)

Tables 5.5 and 5.6 show how performance scales with image size on each platform.

Image Size	**CPU (ms)**	**V1 (ms)**	**V1c (ms)**	**V2 (ms)**	**Speedup V1**	**Speedup V2**
256×256	1.00	0.017	0.020	0.030	59×	34×
512×512	3.99	0.073	0.072	0.103	54×	39×
1024×1024	16.11	0.287	0.270	0.385	56×	42×
2048×2048	65.99	1.132	1.047	1.519	58×	43×
4096×4096	265.23	4.500	4.176	5.819	59×	46×

Tabella 5.5: Image size sweep — GTX 1050 Ti (Gaussian 3×3, 16×16 blocks)

Image Size	**CPU (ms)**	**V1 (ms)**	**V1c (ms)**	**V2 (ms)**	**Speedup V1**	**Speedup V2**
256×256	0.90	0.014	0.017	0.020	64×	44×
512×512	3.58	0.033	0.033	0.044	110×	81×
1024×1024	14.66	0.102	0.100	0.123	143×	120×
2048×2048	60.01	0.407	0.383	0.470	148×	128×
4096×4096	257.72	1.853	1.707	2.158	139×	119×

Tabella 5.6: Image size sweep — Tesla T4 (Gaussian 3×3, 16×16 blocks)

5.4 Kernel Size Sweep (1024×1024 Image, 16×16 Blocks)

Tables 5.7 and 5.8 show the impact of convolution kernel size on performance.

Kernel Size	**CPU (ms)**	**V1 (ms)**	**V1c (ms)**	**V2 (ms)**	**V2/V1**
3×3	16.29	0.274	0.260	0.356	0.77×
5×5	35.73	0.600	0.556	0.402	1.49×
7×7	67.68	0.968	0.862	0.510	1.90×

Tabella 5.7: Kernel size sweep — GTX 1050 Ti (1024 × 1024 image, 16 × 16 blocks)

Kernel Size	**CPU (ms)**	**V1 (ms)**	**V1c (ms)**	**V2 (ms)**	**V2/V1**
3×3	16.97	0.122	0.116	0.147	0.83×
5×5	34.99	0.249	0.216	0.222	1.12×
7×7	66.66	0.463	0.338	0.350	1.32×

Tabella 5.8: Kernel size sweep — Tesla T4 (1024 × 1024 image, 16 × 16 blocks)

5.5 Block Size Sweep (2048×2048, Gaussian 5×5)

Tables 5.9 and 5.10 show performance across different thread block configurations.

Block Config	**Threads/Block**	**V1 (ms)**	**V1c (ms)**	**V2 (ms)**	**V2/V1**
8×8	64	2.555	2.292	1.558	1.64×
16×16	256	2.407	2.215	1.588	1.52×
32×8	256	2.305	2.047	1.418	1.63×
32×16	512	2.265	2.063	1.870	1.21×
32×32	1024	2.303	2.100	2.043	1.13×

Tabella 5.9: Block size sweep — GTX 1050 Ti (2048 × 2048 image, Gaussian 5 × 5)

Block Config	Threads/Block	V1 (ms)	V1c (ms)	V2 (ms)	V2/V1
8×8	64	1.545	1.277	0.866	1.78×
16×16	256	0.980	0.849	0.846	1.16×
32×8	256	0.855	0.821	0.688	1.24×
32×16	512	0.903	0.861	0.789	1.14×
32×32	1024	0.973	0.886	0.898	1.08×

Tabella 5.10: Block size sweep — Tesla T4 (2048×2048 image, Gaussian 5×5)

5.6 Discussion of Results

5.6.1 Cross-Platform Comparison

The T4 provides dramatically higher absolute performance than the GTX 1050 Ti across all benchmarks, as expected from its 3.3× CUDA core count, 2.9× memory bandwidth, and 6.7× SM count advantages. At 4096×4096 with Gaussian 3×3, V1 runs in 1.853 ms on the T4 versus 4.500 ms on the GTX 1050 Ti—a 2.4× raw speedup. The T4's advantage is even more pronounced in CPU-relative speedup: V1 achieves 139× on the T4 versus 59× on the GTX 1050 Ti at 4096×4096. This difference is partly due to the T4's Colab host CPU being slightly faster at the CPU baseline (257.72 ms vs 265.23 ms), but is dominated by the T4's superior GPU throughput.

Interestingly, the relative benefit of V2 over V1 is smaller on the T4 than on the GTX 1050 Ti. For 7×7 kernels, V2/V1 is 1.90× on the GTX 1050 Ti but only 1.32× on the T4. The T4's larger L1/L2 caches and higher memory bandwidth reduce the penalty of V1's redundant global memory reads, diminishing the relative advantage of shared memory tiling. This suggests that as GPU architectures improve their cache hierarchies, explicit shared memory management becomes less critical—though still beneficial.

5.6.2 Image Size Scaling

On the GTX 1050 Ti, GPU kernels achieve 34–59× speedup over the CPU baseline, with speedup growing as image size increases. At 256×256, the GPU is underutilized and kernel launch overhead is proportionally significant. By 4096×4096 (16M pixels), the GPU's parallelism is fully exploited and V1 achieves 59× speedup.

On the T4, the same trend holds but at much higher magnitudes: speedups range from 64× at 256×256 to 148× at 2048×2048 for V1. The T4's 40 SMs (versus 6 on the GTX 1050 Ti) can accommodate far more concurrent thread blocks, enabling it to saturate parallelism at larger image sizes more effectively. The slight speedup drop at 4096×4096 on the T4 (139× versus 148× at 2048×2048) suggests that memory bandwidth becomes the bottleneck at the largest sizes.

On both GPUs, V2 is consistently slower than V1 for this small 3×3 kernel because the shared memory tiling overhead exceeds the data reuse benefit when only 9 neighbor reads are needed per pixel.

5.6.3 V1_const Performance

V1_const consistently outperforms V1 on both platforms, demonstrating that placing filter weights in constant memory is a robust, architecture-independent optimization. On the GTX 1050 Ti, the improvement ranges from 5–12%. On the T4, V1_const provides a similar 2–8% improvement for image size and block sweeps, but a more pronounced benefit for larger kernel sizes: at 7×7, V1_const is 27% faster than V1 on the T4 (0.338 ms vs 0.463 ms) compared to 11% on the GTX 1050 Ti. The T4's larger constant cache may contribute to this increased benefit at larger kernel sizes.

5.6.4 Kernel Size Impact

The V2/V1 ratio reveals a critical crossover point on both platforms, though the crossover behavior differs:

- **GTX 1050 Ti:** V2 is 23% slower than V1 at 3×3 (0.77×), pulls ahead at 5×5 (1.49×), and reaches 1.90× at 7×7.
- **Tesla T4:** V2 is 17% slower than V1 at 3×3 (0.83×), barely ahead at 5×5 (1.12×), and reaches 1.32× at 7×7.

On both GPUs, shared memory tiling becomes increasingly valuable as kernel size grows. However, V2's relative advantage is consistently smaller on the T4 because its higher memory bandwidth and improved cache hierarchy reduce V1's memory bottleneck. The crossover point (where V2 first outperforms V1) occurs between 3×3 and 5×5 on both platforms, confirming that shared memory tiling is most impactful for kernels with 25 or more neighbor reads per pixel.

5.6.5 Block Configuration

On both GPUs, the 32×8 configuration achieves the best V2 performance: 1.418 ms on the GTX 1050 Ti and 0.688 ms on the T4. With 32 threads along the x-axis, each row of a thread block forms exactly one warp, ensuring perfectly coalesced memory access during the tile loading phase. The small tile height (8 rows) keeps shared memory usage low ($(32 + 4) \times (8 + 4) = 432$ floats for 5×5), allowing more concurrent blocks per SM.

At the other extreme, 32×32 blocks (1024 threads) require a large shared memory tile ($(32 + 4) \times (32 + 4) = 1296$ floats), which limits concurrent blocks per SM. V2/V1 drops to 1.13× on the GTX 1050 Ti and 1.08× on the T4 at this configuration.

The T4 shows greater sensitivity to block configuration for V1: performance ranges from 0.855 ms (32×8) to 1.545 ms (8×8), a 1.8× spread. On the GTX 1050 Ti, V1's spread is only 1.13× (2.305 ms to 2.555 ms). This suggests that the T4's larger SM count amplifies the occupancy penalty of suboptimal block sizes.

V1_const outperforms V1 at every block configuration on both platforms, confirming that constant memory caching is an orthogonal optimization that benefits regardless of thread block geometry or GPU architecture.

6. Conclusion

6.1 Summary of Findings

This project implemented and benchmarked three progressively optimized CUDA kernels for 2D image convolution on two GPUs: an NVIDIA GeForce GTX 1050 Ti (Pascal, 768 cores, 6 SMs) and an NVIDIA Tesla T4 (Turing, 2560 cores, 40 SMs):

- All three GPU kernels (V1, V1_const, V2) produce bit-identical output to the CPU baseline across all tested filter configurations on both platforms, confirming correctness.
- The naive GPU kernel (V1) achieves 54–59× speedup over the CPU on the GTX 1050 Ti and up to 148× on the T4 for Gaussian 3×3 convolution, demonstrating how GPU parallelism scales with hardware resources.
- Constant memory caching (V1_const) provides a consistent 5–12% improvement over V1 on the GTX 1050 Ti and up to 27% on the T4 for larger kernels, making it a low-effort, high-value optimization across architectures.
- Shared memory tiling (V2) provides substantial benefit for larger kernels on both GPUs: 1.90× faster than V1 at 7×7 on the GTX 1050 Ti and 1.32× on the T4. The smaller relative gain on the T4 indicates that improved cache hierarchies in newer architectures partially compensate for V1's redundant memory reads, though V2 remains beneficial.
- The 32×8 block configuration is optimal for V2 on both platforms, combining warp-aligned memory coalescing with low shared memory footprint to maximize SM occupancy.

6.2 Lessons Learned

- **Memory hierarchy matters more than raw parallelism.** The V1 kernel already uses thousands of threads, but its performance is limited by global memory bandwidth. V2's shared memory tiling provides substantial improvement for larger kernels by exploiting data locality, without launching more threads.
- **Small optimizations compound.** V1_const demonstrates that even a simple change—moving filter weights to constant memory—yields a consistent 5–12% speedup. In production workloads where convolution is applied repeatedly, these gains are significant.

- **Shared memory tiling has a crossover point.** Tiling introduces overhead (cooperative loading, synchronization barriers, shared memory allocation). For small kernels like 3×3, this overhead exceeds the data reuse benefit on both GPUs. The optimization becomes worthwhile starting at 5×5 kernels, and its advantage grows with kernel size. However, the relative benefit is architecture-dependent: V2's advantage over V1 is larger on the GTX 1050 Ti ($1.90\times$ at 7×7) than on the T4 ($1.32\times$), because newer architectures' improved caches partially mitigate V1's redundant memory accesses.
- **Block shape matters as much as block size.** A 32×8 block and a 16×16 block have the same number of threads (256), but the 32-wide block aligns with warp boundaries for coalesced memory access, resulting in measurably better performance.
- **Template metaprogramming enables zero-cost abstraction on GPUs.** By making kernel size a template parameter, the compiler can fully unroll convolution loops and compute halo sizes at compile time, producing optimal code for each configuration.

6.3 Future Work

- **Kernel fusion:** Implement a fused Gaussian+Sobel edge detection kernel with shared memory tiling to demonstrate the combined benefit of reducing kernel launches and optimizing memory access.
- **Multi-GPU support:** Distribute convolution across multiple GPUs using domain decomposition with halo overlap and CUDA streams for concurrent execution.
- **Separable filter decomposition:** Gaussian filters are separable, meaning a 2D $K \times K$ convolution can be decomposed into two 1D passes (one horizontal, one vertical) with $O(2K)$ operations instead of $O(K^2)$. This could significantly accelerate 5×5 and 7×7 Gaussian filters.
- **Half-precision arithmetic:** Explore FP16 computation on architectures that support it (Volta and newer) to potentially double throughput for applications that tolerate reduced precision.
- **Profiling with NVIDIA Nsight Compute:** Use hardware performance counters to measure achieved memory bandwidth, occupancy, warp efficiency, and shared memory bank conflicts, providing deeper insight into kernel bottlenecks.
- **Benchmark on newer architectures:** Test on Ampere (RTX 3000), Ada (RTX 4000), or Hopper GPUs to observe the impact of larger shared memory, higher bandwidth, and architectural improvements.

Elenco dei codici

Elenco delle tabelle

Bibliografia

Amdahl, Gene M. (1967). «Validity of the single processor approach to achieving large scale computing capabilities». In: *AFIPS Conference Proceedings*. Vol. 30, pp. 483–485. DOI: 10.1145/1465482.1465560.

Gonzalez, Rafael C. e Richard E. Woods (2018). *Digital Image Processing*. 4th. Pearson.

Hennessy, John L. e David A. Patterson (2017). *Computer Architecture: A Quantitative Approach*. 6th. Morgan Kaufmann.

Hwu, Wen-mei, David Kirk e Izzat El Hajj (2023). *Programming Massively Parallel Processors: A Hands-on Approach*. 4th. Morgan Kaufmann. ISBN: 978-0-323-91231-0.

Jia, Zhe et al. (2018). «Dissecting the NVIDIA Volta GPU architecture via microbenchmarking». In: *arXiv preprint arXiv:1804.06826*. URL: https://arxiv.org/abs/1804.06826.

Lindholm, Erik et al. (2008). «NVIDIA Tesla: A unified graphics and computing architecture». In: *IEEE Micro* 28.2, pp. 39–55. DOI: 10.1109/MM.2008.31.

Marr, David e Ellen Hildreth (1980). «Theory of edge detection». In: *Proceedings of the Royal Society of London, Series B* 207.1167, pp. 187–217. DOI: 10.1098/rspb.1980.0020.

Micikevicius, Paulius (2009). «3D finite difference computation on GPUs using CUDA». In: *Proceedings of the 2nd Workshop on General Purpose Processing on Graphics Processing Units*. ACM, pp. 79–84. DOI: 10.1145/1513895.1513905.

Nickolls, John et al. (2008). «Scalable parallel programming with CUDA». In: *ACM Queue* 6.2, pp. 40–53. DOI: 10.1145/1365490.1365500.

NVIDIA Corporation (2024). *CUDA C++ Programming Guide*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

Podlozhnyuk, Victor (2007). *Image convolution with CUDA*. Rapp. tecn. NVIDIA Corporation. URL: https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_64_website/projects/convolutionSeparable/doc/convolutionSeparable.pdf.

Ryoo, Shane et al. (2008). «Optimization principles and application performance evaluation of a multithreaded GPU using CUDA». In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, pp. 73–82. DOI: 10.1145/1345206.1345220.

Sobel, Irwin e Gary Feldman (1968). *A 3x3 isotropic gradient operator for image processing*. Presented at the Stanford Artificial Intelligence Project.

Szeliski, Richard (2022). *Computer Vision: Algorithms and Applications*. 2nd. Springer. DOI: 10.1007/978-3-030-34372-9.

Volkov, Vasily (2010). «Better performance at lower occupancy». In: *Proceedings of the GPU Technology Conference (GTC 2010)*. NVIDIA. URL: https://www.nvidia.com/content/gtc-2010/pdfs/2238_gtc2010.pdf.