Research Document

Performance of Parallel Processing in Image Convolution

Author: Nguyen Duc Anh

Computer Vision · Image Processing · Convolution · Parallel Computing

Hanoi, January 2026

Abstract

Image convolution is a fundamental operation in computer vision, yet it remains computationally intensive. This paper presents a comprehensive comparison of serial (CPU), multi-threaded (OpenMP), and GPU-accelerated (CUDA) implementations.

Testing 12 kernel types on resolutions up to 4096×4096, our results demonstrate that CUDA achieves up to **29.48**$\times$ speedup for complex kernels on high-resolution images.

However, we identify a **critical performance collapse** where CUDA becomes **11$\times$ slower** than the CPU on small workloads due to memory transfer overhead (about 100ms latency).

We conclude that while GPUs excel at massive data parallelism, OpenMP remains the superior choice for low-latency, real-time applications involving small kernels, necessitating a workload-aware hybrid approach.

Key Word

1. Basic Concepts

- **Noise:** Unwanted random variations in pixel values. Like "static" on old TVs.
  - **Sources:** sensor imperfection, low light, transmission errors
  - **Types:** Gaussian noise (smooth), Salt & Pepper (black/white dots)
- **PixelSmallest:** unit of an image. Has a value (0-255 for grayscale).
- **ConvolutionSliding:** a kernel over an image, multiplying and summing at each position.
- **Kernel (Filter):** A small matrix of weights used in convolution. Determines the effect (blur, edge, sharpen).
- **EdgeBoundary:** between two regions with different intensities. Where image "changes quickly."
- **BlurSmoothing:** an image by averaging nearby pixels. Reduces noise but loses detail.
- **Sharpen:** Enhancing edges to make image look "crisper."

2. Filter Types

- **Low-Pass Filter:** Allows low frequencies (smooth areas), blocks high frequencies (edges, noise). Example: Gaussian blur, Box filter
- **High-Pass Filter:** Allows high frequencies (edges, details), blocks low frequencies (smooth areas). Example: Laplacian, Sobel
- **Band-Pass Filter:** Allows only a specific range of frequencies. Example: LoG (Laplacian of Gaussian)

3. Mathematical Terms

- **Derivative:** Rate of change. First derivative = slope. Second derivative = curvature.
- **Gradient:** Direction and magnitude of steepest change. Used for edge detection.
- **Normalization:** Scaling values to a standard range (e.g., sum to 1, or range 0-255).
- **Separable:** A 2D kernel that can be split into two 1D kernels for faster computation.

4. Connectivity

- **4-Connectivity:** Only direct neighbors (up, down, left, right) are considered "connected."
- **8-Connectivity:** All 8 surrounding pixels (including diagonals) are considered "connected."

5. Image Properties

- **Intensity:** Brightness value of a pixel (0 = black, 255 = white for 8-bit).
- **Frequency (in images):** How fast pixel values change spatially.
  - **Low frequency:** Smooth, slowly changing regions
  - **High frequency:** Edges, textures, noise
- **Resolution:** Number of pixels in an image (width $\times$ height).

6. Performance Terms

- **Parallelization:** Splitting work across multiple processors to run simultaneously.
- **Throughput:** Amount of data processed per unit time (e.g., megapixels/second).
- **Latency:** Time delay from input to output.
- **Memory Bandwidth:** Rate at which data can be read/written to memory.
- **Cache:** Fast, small memory close to CPU. Accessing cached data is much faster.

7. CUDA Terms

- **Thread:** Smallest unit of execution in GPU.
- **Block:** Group of threads that can share memory and synchronize.
- **Grid:** Collection of all blocks for a kernel launch.
- **Shared Memory:** Fast memory shared by threads in a block.
- **Global Memory:** Large but slow GPU memory accessible by all threads.
- **Coalescing:** When adjacent threads access adjacent memory locations (efficient).

Contents

1 Introduction

Convolution is the mathematical engine behind edge detection, blurring, and modern Convolutional Neural Networks (CNNs), etc. As image resolutions increase to 4K and beyond, serial CPU processing becomes a bottleneck.

This research investigates the performance trade-offs between Multi-core CPU execution (via OpenMP) and Massively Parallel GPU execution (via CUDA).

The primary research question is: *At what point does the overhead of data transfer to the GPU outweigh the benefits of parallel computation?*

2 Theoretical Background

2.1 Convolution

Fundamentals of Convolution

Convolution is a mathematical operation that combines two functions to produce a third function. In the context of image processing, it's the process of adding each element of an image to its local neighbors, weighted by a **kernel** (also called a filter or mask).

```
+-------------+     +-------------+     +-------------+
|   INPUT     |     |   KERNEL    |     |   OUTPUT    |
|   IMAGE     |  *  |  (Filter)   |  =  |   IMAGE     |
|             |     |             |     |             |
+-------------+     +-------------+     +-------------+
     I(x,y)              K(i,j)              O(x,y)
```

Think of convolution as a **"Sliding Window"** operation:

1. Place the kernel over a region of the image
2. Multiply corresponding elements
3. Sum all products to get one output pixel
4. Slide the kernel and repeat

1. Mathematical Definition

1.1 Continuous 1D Convolution

For two continuous functions f and g:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau) \cdot g(t-\tau)\, d\tau$$

1.2 Continuous 2D Convolution

For two-dimensional functions (like images):

$$(f * g)(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(u, v) \cdot g(x-u, y-v)\, du\, dv$$

1.3 Discrete 2D Convolution (Image Processing)

For digital images with kernel of size $(2k+1) \times (2k+1)$:

$$O(x, y) = \sum_{i=-k}^{k} \sum_{j=-k}^{k} I(x+i, y+j) \cdot K(i, j)$$

Where:

- I = Input image
- K = Kernel/Filter
- O = Output image
- k = Kernel radius (for a 3×3 kernel, $k = 1$)

2. Intuitive Understanding

Weighted Voting

Imagine each pixel is "voting" on what the output pixel should be:

- **Box filter:** Every neighbor gets equal vote (democracy)
- **Gaussian filter:** Closer neighbors get more votes (weighted democracy)
- **Edge detector:** Neighbors on opposite sides vote against each other (difference detector)

Visual Example: 3×3 Convolution

```
    Input Image (5x5):              Kernel (3x3):
+---+---+---+---+---+          +---+---+---+
| 1 | 2 | 3 | 4 | 5 |          | 1 | 2 | 1 |
+---+---+---+---+---+          +---+---+---+
| 6 | 7 | 8 | 9 |10 |          | 2 | 4 | 2 |
+---+---+---+---+---+          +---+---+---+
|11 |12 |13 |14 |15 |          | 1 | 2 | 1 |
+---+---+---+---+---+          +---+---+---+
|16 |17 |18 |19 |20 |           Sum = 16
+---+---+---+---+---+
|21 |22 |23 |24 |25 |
+---+---+---+---+---+

Computing O(2,2) [center of 3x3 region starting at (1,1)]:

  1x1 + 2x2 + 3x1 = 8
  6x2 + 7x4 + 8x2 = 56
  11x1 + 12x2 + 13x1 = 48

  Total = 8 + 56 + 48 = 112
  Normalized = 112/16 = 7 (the center value!)
```

3. Properties of Convolution

3.1 Commutativity

$$f * g = g * f$$

The order doesn't matter - convolving image with kernel equals convolving kernel with image.

3.2 Associativity

$$(f * g) * h = f * (g * h)$$

Multiple convolutions can be combined. This is crucial for filter cascading.

3.3 Distributivity

$$f * (g + h) = (f * g) + (f * h)$$

Convolution distributes over addition.

3.4 Linearity

$$a(f * g) = (af) * g = f * (ag)$$

Scalar multiplication can be done before or after convolution.

3.5 Shift Invariance

$$\text{If } f(x) \to g(x), \text{ then } f(x - x_0) \to g(x - x_0)$$

The same operation is applied regardless of position - crucial for image processing.

3.6 Why These Properties Matter

- **Commutativity:** Kernel design flexibility.
- **Associativity:** Combine multiple filters into one.
- **Distributivity:** Parallel filter application.
- **Linearity:** Predictable scaling behavior.
- **Shift Invariance:** Position-independent processing.

Convolution in Image Processing

1. 2D Discrete Convolution

1.1 Algorithm:

```
For each pixel (x, y) in output:
    sum = 0
    For each kernel position (i, j):
        image_x = x + i - kernel_center_x
        image_y = y + j - kernel_center_y
        sum += Image[image_x, image_y] * Kernel[i, j]
    Output[x, y] = sum
```

1.2 Correlation vs Convolution

- **Convolution:** flips the kernel before applying:

$$O(x, y) = \sum_{i,j} I(x - i, y - j) \cdot K(i, j)$$

- **Correlation:** (cross-correlation) does not flip:

$$O(x, y) = \sum_{i,j} I(x + i, y + j) \cdot K(i, j)$$

- For **symmetric kernels** (most common in image processing), they produce identical results.

In practice, most image processing libraries implement correlation but call it convolution.

2. Boundary Handling

When the kernel extends beyond image borders, we need a strategy:

2.1 Zero Padding (Constant)

```
0 0 0 0 0 0
0 +-----+ 0
0 |Image| 0
0 +-----+ 0
0 0 0 0 0 0
```

- Assume pixels outside are 0 (or any constant)
- Creates dark borders for bright images

2.2 Replicate (Clamp)

```
A A | A B C | C C
A A | A B C | C C
----+-------+----
D D | D E F | F F
G G | G H I | I I
----+-------+----
G G | G H I | I I
```

- Extend edge pixels outward
- Good for natural images

2.3 Reflect (Mirror)

```
E D | D E F | F E
B A | A B C | C B
----+-------+----
B A | A B C | C B
E D | D E F | F E
----+-------+----
E D | D E F | F E
```

- Mirror image at boundaries
- Preserves continuity

2.4 Wrap (Circular)

```
I G | G H I | G H
C A | A B C | A B
----+-------+----
C A | A B C | A B
F D | D E F | D E
----+-------+----
I G | G H I | G H
```

- Treat image as tiling/repeating
- Used in frequency domain processing

2.5 Comparison

Method	Use-Case	Artifacts
Zero	General purpose	Dark borders
Replicate	Natural images	Edge color bleeding
Reflect	Seamless processing	Mirror artifacts
Wrap	Frequency analysis	Wrap-around effects

3. Separable Convolution

A 2D kernel is **separable** if it can be expressed as the outer product of two 1D kernels:

$$K_{2D} = K_x^T \cdot K_y$$

Why Separable Convolution Matters:

- 3×3 kernel: 9 operations $\rightarrow 3 + 3 = 6$ operations ($1.5\times$ faster)
- 31×31 kernel: 961 operations $\rightarrow 31 + 31 = 62$ operations (**15.5× faster!**)

Speedup Formula:

$$\text{Speedup} = \frac{K^2}{2K} = \frac{K}{2}$$

Example: Gaussian Kernel

```
2D Gaussian (3x3):          Can be decomposed to:

+---+---+---+              +---+
| 1 | 2 | 1 |              | 1 |
+---+---+---+      =       | 2 |  x  [1  2  1]
| 2 | 4 | 2 |              | 1 |
+---+---+---+              +---+
| 1 | 2 | 1 |
+---+---+---+              Vertical x Horizontal
```

3.1 Computational Advantage

Method	Operations per Pixel	For 31×31 kernel
2D Direct	K^2	961
Separable	$2K$	62
Speedup	$K/2$	15.5×

3.2 Which Kernels are Separable?

Kernel	Separable	Notes
Box Filter	Yes	All-ones vectors
Gaussian	Yes	Gaussian vectors
Sobel	Yes	Smooth × Derivative
Laplacian	No	Must use 2D
Gabor	No	Complex structure

Frequency Domain Perspective

1. Fourier Transform Connection

1.1 The Convolution Theorem

One of the most important theorems in signal processing:

$$\mathcal{F}\{f * g\} = \mathcal{F}\{f\} \cdot \mathcal{F}\{g\}$$

Convolution in spatial domain = Multiplication in frequency domain

1.2 Implications

- **Alternative computation path:**
  - FFT(Image) × FFT(Kernel) → IFFT → Result
  - Faster for very large kernels ($K > 15$)
- **Filter design insight:**
  - Design filters by specifying frequency response
  - Understand what frequencies are passed/blocked

1.3 Frequency Response of Common Filters

```
Frequency Response Magnitude

     |
   1 |
     |        ##
     |          ##
     |
   0 |________________________
     0    Low    Mid   High
             Frequency

      = Low-pass (Blur)
     -------- = High-pass (Edge)
```

2. Low-Pass vs High-Pass

2.1 Low-Pass Filters (Smoothing)

Remove high frequencies → Blur/Smooth

- Box filter
- Gaussian filter
- Averaging filters

Frequency response: Attenuates high frequencies

$$H_{lowpass}(f) \approx 1 \text{ for } f < f_c, \quad \approx 0 \text{ for } f > f_c$$

2.2 High-Pass Filters (Sharpening/Edge)

Remove low frequencies → Enhance edges

- Laplacian
- High-pass filter
- Edge detectors

Frequency response: Attenuates low frequencies

$$H_{highpass}(f) = 1 - H_{lowpass}(f)$$

2.3 Band-Pass Filters

Keep only certain frequency range

- Laplacian of Gaussian (LoG)
- Difference of Gaussians (DoG)
- Gabor filters

2.4 Visual Summary

```
Signal:     #*......
            Low  Mid  High
            -------------------
Low-pass:   #*        <- Smooth/Blur
High-pass:           ...... <- Edges/Details
Band-pass:       *        <- Specific features
```

Applications in Computer Vision

1. Image Enhancement

1.1 Noise Reduction

- **Problem:** Images contain unwanted random variations (noise)
- **Solution:** Low-pass filtering averages out noise

Noise Type	Best Filter
Gaussian noise	Gaussian filter
Salt & pepper	Median filter (non-linear)
Uniform noise	Box filter

- **Trade-off:** Noise reduction vs. detail preservation

$$SNR_{output} > SNR_{input} \text{ but edges become blurred}$$

1.2 Contrast Enhancement

Unsharp Masking: Enhance edges by adding high-frequency components

$$I_{enhanced} = I + \alpha \cdot (I - I_{blur})$$

Where α controls enhancement strength.

2. Feature Extraction

2.1 Gradient-Based Features

HOG (Histogram of Oriented Gradients):

1. Compute gradients using Sobel
2. Create orientation histograms in cells
3. Normalize across blocks
4. Concatenate into feature vector

Used in: Pedestrian detection, object recognition

2.2 Corner Detection

Harris Corner Detector:

1. Compute gradients I_x, I_y (Sobel)
2. Build structure tensor $M = \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$
3. Compute corner response $R = det(M) - k \cdot trace(M)^2$

2.3 Scale-Invariant Features

SIFT (Scale-Invariant Feature Transform):

1. Build Gaussian scale-space: $L(x, y, \sigma) = G(x, y, \sigma) * I(x, y)$
2. Compute DoG: $D = L(x, y, k\sigma) - L(x, y, \sigma)$
3. Find extrema across scales
4. Assign orientation using gradient histograms

3. Edge Detection

3.1 The Edge Detection Pipeline

```
+-----------+    +---------+    +---------+    +---------+
| Smooth    | -> | Gradient| -> |   NMS   | -> |Threshold|
| (Gaussian)|    | (Sobel) |    |         |    |         |
+-----------+    +---------+    +---------+    +---------+
```

1. **Smoothing:** Reduce noise (Gaussian)
2. **Gradient:** Find intensity changes (Sobel)
3. **Non-Maximum Suppression:** Thin edges
4. **Thresholding:** Binary edge map

3.2 Canny Edge Detector

The "optimal" edge detector:

$$G = \sqrt{G_x^2 + G_y^2}, \quad \theta = \arctan\left(\frac{G_y}{G_x}\right)$$

Steps:

1. Gaussian smoothing
2. Gradient magnitude and direction (Sobel)
3. Non-maximum suppression
4. Double thresholding
5. Edge tracking by hysteresis

4. Texture Analysis

4.1 Texture Definition

Texture = Spatial arrangement of intensities that creates visual patterns.
Examples: Fabric, grass, wood grain, brick walls

4.2 Gabor Filter Banks

Multiple Gabor filters at different orientations and scales capture texture characteristics:

```
Orientation:   0°    45°    90°   135°
              ---    ///    |||   \\\

Scale 1:      Fine texture features
Scale 2:      Medium texture features
Scale 3:      Coarse texture features
```

Applications:

- Texture classification
- Texture segmentation
- Material recognition

Convolution in Deep Learning

1. Convolutional Neural Networks

1.1 Why Convolution for Images?

Traditional neural networks (fully connected):

- Image 224×224×3 = 150,528 inputs
- If next layer has 1000 neurons: 150 million parameters!
- No translation invariance

CNNs solve this with:

Property	Benefit
Local connectivity	Each neuron sees only a small region
Parameter sharing	Same kernel used across entire image
Translation invariance	Features detected anywhere
Hierarchical features	Low → Mid → High level features

1.2 CNN Layer Structure

```
Input        Conv Layer          Activation    Pooling
(H x W x C) (Multiple Kernels)   (ReLU)        (Max/Avg)
   |              |                  |             |
   v              v                  v             v
+-----+      +---------+        +-------+     +-------+
|     |      |K1|K2|K3 |        |       |     |       |
|Image|  ->  |--+--+-- |   ->   | ReLU  |  -> | Pool  |
|     |      |Feature  |        |       |     |       |
|     |      |  Maps   |        |       |     |       |
+-----+      +---------+        +-------+     +-------+
```

1.3 Feature Hierarchy

```
Layer 1: Edge detectors (like Sobel, Gabor)
    |
    v
Layer 2: Corners, textures (combinations of edges)
    |
    v
Layer 3: Parts (eyes, wheels, text)
    |
    v
Layer 4+: Objects, scenes (faces, cars, documents)
```

2. Learned vs Hand-crafted Kernels

2.1 Hand-crafted Kernels (Classical CV)

- **Advantages:**
  - Interpretable
  - No training data needed
  - Mathematically principled
  - Predictable behavior
- **Disadvantages:**
  - Limited expressiveness

  - Manual design effort
  - May not be optimal for specific tasks

2.2 Learned Kernels (Deep Learning)

- **Advantages:**
  - Task-optimized
  - Can learn complex patterns
  - Data-driven
  - Often superior performance
- **Disadvantages:**
  - Need large training data
  - Black box / less interpretable
  - Computationally expensive to train

2.3 Interesting Observation

First-layer CNN filters often resemble classical kernels!

```
Learned CNN Filter          Classical Equivalent
+---+---+---+              +---+---+---+
|-1 | 0 |+1 |              |-1 | 0 |+1 |
|-2 | 0 |+2 |      ~=      |-2 | 0 |+2 |   Sobel X
|-1 | 0 |+1 |              |-1 | 0 |+1 |
+---+---+---+              +---+---+---+
```

This validates the importance of understanding classical kernels!

Computational Considerations

1. Complexity Analysis

1.1 Time Complexity

Method	Complexity	Notes
Naive 2D	$O(N^2 \cdot K^2)$	N = image size, K = kernel size
Separable	$O(N^2 \cdot 2K)$	For separable kernels
FFT-based	$O(N^2 \log N)$	Better for K ¿ 15
Integral Image	$O(N^2)$	For box filter only

1.2 When to Use Each Method

```
Kernel Size:  3    7    11   15   21   31   ...
              |    |    |    |    |    |
Naive 2D:     ............
Separable:    ##......
FFT:          ............
```

2. Memory Access Patterns

2.1 Cache Optimization

- **Row-major access:** Process image row by row

```
for (y = 0; y < height; y++)
    for (x = 0; x < width; x++)
        // Good: sequential memory access
```

- **Blocking/Tiling:** Process in cache-sized blocks

```
for (by = 0; by < height; by += BLOCK)
    for (bx = 0; bx < width; bx += BLOCK)
        // Process block[by:by+BLOCK, bx:bx+BLOCK]
```

2.2 GPU Memory Hierarchy

```
+-------------------------------------+
|           Global Memory             |  Slow, Large
|              (VRAM)                 |
+-------------------------------------+
|          Shared Memory              |  Fast, Small
|        (Per Block, ~48KB)           |
+-------------------------------------+
|           Registers                 |  Fastest
|          (Per Thread)               |
+-------------------------------------+
```

Strategy: Load kernel into shared memory, reuse across threads.

3. Parallelization Strategies

3.1 OpenMP (Multi-core CPU)

```
#pragma omp parallel for collapse(2)
for (int y = 0; y < height; y++)
    for (int x = 0; x < width; x++)
        output[y][x] = convolve(input, kernel, x, y);
```

Key considerations:

- Thread count vs core count
- NUMA awareness
- Cache line sharing (false sharing)
- Load balancing

3.2 CUDA (GPU)

```
// Each thread computes one output pixel
__global__ void convolve(float* in, float* out, float* kernel)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    // Load kernel to shared memory
    __shared__ float s_kernel[K][K];
    // ...

    // Compute convolution for pixel (x, y)
    float sum = 0;
    for (int ky = 0; ky < K; ky++)
        for (int kx = 0; kx < K; kx++)
            sum += in[...] * s_kernel[ky][kx];

    out[y * width + x] = sum;
}
```

Key considerations:

- Block size (16×16 or 32×32 typical)
- Shared memory usage
- Coalesced global memory access
- Occupancy optimization

3.3 Expected Speedups

Platform	vs Single-Core	Notes
4-core CPU	3-4×	Limited by memory
8-core CPU	5-7×	Diminishing returns
Entry GPU	10-50×	Memory bandwidth
High-end GPU	50-200×	For large kernels

2.2 Kernel

Introduction

Convolution is the cornerstone operation in image processing and computer vision. A **kernel** (or **filter**) is a small matrix that slides over an image to extract features, reduce noise, detect edges, or enhance certain characteristics.
Understanding these 10 fundamental kernels provides insight into:

- Classical computer vision algorithms (SIFT, Canny, Harris)
- Modern deep learning (CNN feature extraction)
- Image preprocessing pipelines
- Parallel computing optimization strategies

Mathematical Foundation

Recall: Discrete 2D Convolution

For an input image I and kernel K of size $(2k+1) \times (2k+1)$:

$$O(x,y) = \sum_{i=-k}^{k} \sum_{j=-k}^{k} I(x+i, y+j) \cdot K(i,j)$$

Where:

- $I(x,y)$ = Input pixel intensity at position (x,y)
- $K(i,j)$ = Kernel weight at offset (i,j)
- $O(x,y)$ = Output pixel value

Property	**Description**
Linearity	$f(aI_1 + bI_2) = a \cdot f(I_1) + b \cdot f(I_2)$
Shift Invariance	Same operation regardless of position
Separability	Some 2D kernels = 1D × 1D (reduces $O(n^2)$ to $O(2n)$)
Commutativity	$I * K = K * I$

The 10 Fundamental Kernels

1. Box Filter (Mean Filter) (Smoothing / Low-pass Filter)

The Box Filter computes the **arithmetic mean** of all pixels within the kernel window. It treats all neighboring pixels equally, making it the simplest form of spatial averaging.

$$K_{box} = \frac{1}{n^2} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

For a 3×3 kernel, each weight = $\frac{1}{9}$

$$O(x,y) = \frac{1}{n^2} \sum_{i=-k}^{k} \sum_{j=-k}^{k} I(x+i, y+j)$$

Properties:

- **Effect:** Suppresses high-frequency components (noise, fine details)
- **Frequency Response:**Low-pass filter with sinc function response

- **Computational Bound:** Memory-bound (simple arithmetic, many memory accesses)
- **Separability: Yes** - can be decomposed into two 1D passes

Applications:

- Noise reduction in preprocessing pipelines
- Image segmentation preprocessing
- Downsampling anti-aliasing
- Real-time video processing (computationally cheap)

Every CV pipeline uses smoothing somewhere. Box filter is the foundation - understanding it means understanding the trade-off between noise reduction and detail preservation.

2. Gaussian Filter (Physical Smoothing / Scale-Space)

The Gaussian Filter is arguably the most important kernel in computer vision. It applies weights based on a 2D Gaussian distribution, giving more importance to central pixels and smoothly decreasing influence with distance.

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Discrete Kernel ($\sigma \approx 1.0$)

$$K_{gaussian} = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Choosing Kernel Size from σ:
The Gaussian theoretically extends to infinity, but at distance 3σ from center, the value drops to ~1% of peak. Thus we use:

$$\text{kernel\_size} = 6\sigma + 1 \quad \text{(to ensure odd size for center pixel)}$$

σ	Kernel Size	Effect
0.5	3×3	Very subtle blur
1.0	7×7	Light blur
2.0	13×13	Medium blur
5.0	31×31	Heavy blur

Properties:

- **Effect:** Smooth blurring that preserves edge location better than box
- **Frequency:** Response Gaussian in frequency domain (no ringing artifacts)
- **Separability: Yes** - $G_{2D}(x, y) = G_{1D}(x) \times G_{1D}(y)$
- **Scale Parameter:** σ controls blur amount

The Scale-Space Theory:
Gaussian filtering at different σ values creates a scale-space representation:

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y)$$

This is fundamental to:

- SIFT (Scale-Invariant Feature Transform)
- Multi-scale edge detection
- Pyramid representations

Applications:

- Canny Edge Detection - Gaussian smoothing is the first step
- SIFT/SURF - Difference of Gaussians for keypoint detection
- Image pyramids - Progressive downsampling
- Noise reduction - Superior to box filter for natural images

If you deeply understand the Gaussian filter, you understand half of classical computer vision. It's the bridge between spatial and frequency domains, and the foundation of scale-space theory.

3. Sobel X (Horizontal Gradient) (First-Order Derivative / Edge Detection)

The Sobel X operator computes the horizontal gradient (rate of change in the X direction). It detects vertical edges in the image.

$$K_{sobel\_x} = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$$

$$G_x = \frac{\partial I}{\partial x} \approx I * K_{sobel\_x}$$

This approximates the partial derivative using finite differences with Gaussian smoothing in the perpendicular direction.

Properties:

- **Output Range:** Signed values (negative to positive)
- **Detects:** Vertical edges (intensity changes in X direction)
- **Noise Handling:** Built-in smoothing in Y direction
- **Separability: Yes** - smooth $\times$ differentiate

Structure: Sobel X = Smoothing (Y) $\times$ Differentiation (X):

$$K_{sobel\_x} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \times \begin{bmatrix} -1 & 0 & +1 \end{bmatrix}$$

Gradient computation is fundamental to edge detection, optical flow, and feature extraction. Sobel provides robust gradient estimation with noise suppression.

4. Sobel Y (Vertical Gradient) (First-Order Derivative / Edge Detection)

The Sobel Y operator computes the vertical gradient (rate of change in the Y direction). It detects horizontal edges in the image.

$$K_{sobel\_y} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

$$G_y = \frac{\partial I}{\partial y} \approx I * K_{sobel\_y}$$

Gradient Magnitude and Direction

Combining Sobel X and Y gives us complete edge information:

- **Magnitude:** $|G| = \sqrt{G_x^2 + G_y^2}$
- **Direction:** $\theta = \arctan\left(\frac{G_y}{G_x}\right)$

Applications:

- Edge detection (Canny, etc.)
- Optical flow computation
- Feature descriptors (HOG, SIFT)
- Image sharpening

You MUST have both X and Y gradients to compute edge magnitude and direction. This pair is the core of edge detection, motion estimation, and feature extraction.

5. Laplacian (Second-Order Derivative)

The Laplacian operator computes the second derivative of the image, detecting regions where intensity changes rapidly. It highlights zero-crossings where the gradient changes sign.

$$\nabla^2 I = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$$

Discrete Kernels:

- **4-connected:**

$$K_{laplacian\_4} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

- **8-connected:**

$$K_{laplacian\_8} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Properties

- **Output:** Zero at edges, positive/negative on either side
- **Noise Sensitivity:** Very high (amplifies high frequencies)
- **Rotation Invariant: Yes** (isotropic)
- **Zero-Sum:** Kernel sums to 0

Applications:

- Blob detection - Responds to blob-like structures
- Edge detection - Zero-crossing method
- Image sharpening - $I_{sharp} = I - \alpha \cdot \nabla^2 I$
- Feature detection - Corner and blob responses

The Laplacian detects regions of rapid intensity change and is fundamental to blob detection, sharpening, and feature detection. However, it's sensitive to noise and often used with Gaussian pre-smoothing.

6. Laplacian of Gaussian (LoG) (Combined Operator / Blob Detection)

The Laplacian of Gaussian (LoG) combines Gaussian smoothing with Laplacian edge detection. This addresses the Laplacian's noise sensitivity while providing a powerful blob detector.
The Problem with Laplacian Alone:
Laplacian is too sensitive — it detects ANY change, including tiny noise!

```
Noisy image with real edge:
[50][52][48][200][198]
  ^noise^   ^realEdge^

Laplacian sees EVERYTHING as edges (false positives!)
```

What is LoG?
LoG = **L**aplacian **of** **G**aussian — a two-step process combined into one:

1. **Gaussian blur** → smooth out noise
2. **Laplacian** → find real edges

Situation	**Use**	**Why**
Noisy images (photos, medical)	**LoG**	Filters noise first
Clean synthetic images	**Laplacian**	Faster, no noise
Real-time (speed critical)	**Laplacian**	LoG is slower

The LoG Formula:

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

Visual Appearance

The LoG kernel resembles a "Mexican Hat" (sombrero):

- Positive center (Gaussian peak)
- Negative surround (ring)
- Zero-crossing at characteristic radius

Approximation: Difference of Gaussians (DoG)

$$DoG \approx G(x,y,k\sigma) - G(x,y,\sigma) \approx (k-1)\sigma^2 \cdot LoG$$

This approximation is used in SIFT for computational efficiency.
Properties:

- **Scale Selectivity:** Responds maximally to blobs of size $r \approx \sqrt{2}\sigma$
- **Noise Robustness:** Much better than raw Laplacian
- **Computation:** Can use DoG approximation for speed

Applications:

- Scale-space blob detection
- SIFT keypoint detection (via DoG)
- Multi-scale feature analysis
- Cell/particle detection in medical imaging

LoG is the bridge between low-level (edges, gradients) and mid-level vision (blobs, regions, features). Understanding LoG leads to understanding scale-invariant feature detection.

7. Sharpening Filter (Unsharp Mask) (Enhancement)

Sharpening enhances edges and fine details by amplifying high frequencies. The Unsharp Mask technique works by:

1. Blur the image (create "unsharp" version)
2. Subtract blur from original (extract details)
3. Add amplified details back to original

$$I_{sharp} = I + \alpha(I - I_{blur}) = (1+\alpha)I - \alpha \cdot I_{blur}$$

Kernel Representation

Combining into a single kernel (with $\alpha = 1$):

$$K_{sharpen} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

This is equivalent to:

$$K_{sharpen} = (1 + \alpha) \cdot K_{identity} - \alpha \cdot K_{blur}$$

Properties:

- **Effect:** Enhances edges and fine details
- **Risk:** Can amplify noise if α is too high
- **Parameter:** α controls sharpening strength
- **Linearity:** Kernel is linear, but typically combined with non-linear clipping

Applications:

- Image preprocessing before recognition
- Medical imaging enhancement
- Photography post-processing
- Print preparation

Sharpening demonstrates how linear filters can be combined to achieve enhancement effects. It also shows the interplay between smoothing (low-pass) and detail extraction (high-pass).

8. High-Pass Filter (Frequency Emphasis / Edge Enhancement)

A High-Pass Filter removes low-frequency components (smooth regions) and preserves high-frequency components (edges, noise, fine details).
Key insight: High-pass = Identity - Low-pass

$$K_{highpass} = K_{identity} - K_{lowpass}$$

Kernel Example

$$K_{highpass} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} - \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \frac{1}{9} \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Frequency Domain Interpretation

- **In the frequency domain:**
  - Low-pass filters attenuate high frequencies
  - High-pass filters attenuate low frequencies
  - Band-pass filters keep a range of frequencies
- **Connection to CNNs:** The first layers of Convolutional Neural Networks often learn filters that resemble high-pass filters:
  - Edge detectors
  - Texture extractors
  - Gradient-like patterns

Properties:

- **Effect:** Extracts edges, removes DC component
- **Output:** Can be negative (requires offset for display)
- **DC Response:** Zero (kernel sums to 0)

Understanding high-pass filtering explains frequency domain processing and provides insight into why CNN early layers learn edge-detecting filters automatically.

9. Gabor Filter (Texture / Pattern Analysis)

The Gabor Filter is a linear filter whose impulse response is a sinusoidal wave modulated by a Gaussian envelope. It models the receptive fields of simple cells in the visual cortex.

$$G(x,y) = \exp\left(-\frac{x'^2 + \gamma^2 y'^2}{2\sigma^2}\right)\cos\left(2\pi\frac{x'}{\lambda} + \psi\right)$$

Where:

- $x' = x\cos\theta + y\sin\theta$ (rotated coordinates)
- $y' = -x\sin\theta + y\cos\theta$
- λ = wavelength of sinusoid
- θ = orientation
- σ = Gaussian envelope size
- γ = aspect ratio
- ψ = phase offset

Parameters:

- θ: Orientation selectivity (0°, 45°, 90°, 135°)
- λ: Frequency selectivity (fine to coarse texture)
- σ: Scale of analysis
- γ: Elongation of filter

Gabor Filter Bank

Typically, multiple Gabor filters at different orientations and scales form a filter bank:

- 4-8 orientations (e.g., 0°, 22.5°, 45°, ...)
- 3-5 scales

Applications:

- Texture classification and segmentation
- Face recognition (classical methods)
- Fingerprint analysis
- Document analysis (character recognition)

Gabor filters model biological vision and help explain why CNNs learn similar oriented, frequency-selective features. Understanding Gabor = understanding the "why" behind learned CNN features.// **Note:** Gabor filters are computationally expensive due to their complex structure. They make excellent stress tests for parallel implementations but may be optional in basic benchmarks.

10. Large Kernels (15×15, 31×31) (Stress Test / Heavy Blur)

Large kernels (15×15, 31×31, or larger) perform the same operations as smaller kernels but with a much larger receptive field. While not introducing new effects, they exhibit fundamentally different computational behavior.

Computational Characteristics:

Kernel Size	Operations per Pixel	Memory Access Pattern
3×3	9	Cache-friendly
7×7	49	Moderate
15×15	225	Memory bandwidth limited
31×31	961	Compute-bound

Why Large Kernels Matter:

1. Compute-Bound Behavior
   - Small kernels: Memory-bound (waiting for data)
   - Large kernels: Compute-bound (ALU utilization)
2. GPU Advantage Amplification
   - GPUs excel when there's enough computation to hide memory latency
   - Large kernels show dramatic GPU speedups
3. Parallelization Efficiency
   - More work per pixel = better thread utilization
   - Amortizes thread creation/synchronization overhead
4. Separable Filter Advantage
   - 31×31 kernel: 961 operations (2D) vs 62 operations (separable)
   - 15× speedup from separability alone

Applications:

- Heavy blur effects (background blur, privacy)
- Large-scale smoothing in image pyramids
- Stress testing parallel implementations
- CNN-like workloads (modern CNNs use various kernel sizes)

Large kernels reveal the true potential of GPU parallelization and expose bottlenecks that small kernels hide. They represent real-world CNN computational patterns.

Kernel-to-Domain Mapping

Kernel	Primary CV Domain	Secondary Applications
Box Filter	Preprocessing, Segmentation	Real-time video, downsampling
Gaussian	Scale-Space, SIFT, Canny	Noise reduction, pyramids
Sobel X	Edge Detection, Optical Flow	Feature extraction, gradients
Sobel Y	Edge Detection, Optical Flow	HOG descriptors, motion
Laplacian	Blob Detection, Corners	Sharpening, feature detection
LoG	Multi-scale Detection	SIFT (via DoG), blob finding
Sharpen	Enhancement	Medical imaging, preprocessing
High-Pass	Edge Detection, Frequency Analysis	CNN interpretation, filtering
Gabor	Texture Analysis, Pattern Recognition	Face recognition, fingerprints
Large Kernels	CNN Workloads, Heavy Processing	Stress testing, GPU benchmarking

Performance Characteristics

1. Computational Complexity

For image size $W \times H$ and kernel size $K \times K$:

Implementation	Complexity	Notes
Naive 2D	$O(W \cdot H \cdot K^2)$	Baseline
Separable	$O(W \cdot H \cdot 2K)$	For separable kernels
FFT-based	$O(W \cdot H \cdot \log(WH))$	Better for very large K

2. Memory vs Compute Bound

```
Memory Bound                            Compute Bound
    <----------------------------------------------->

 3x3        5x5        7x7       15x15      31x31
  |          |          |          |          |
  v          v          v          v          v
[Box]    [Gaussian] [Sobel]   [Large]   [Very Very Large]
  |          |          |          |          |
  |          |          |          |          |
  +---- CPU competitive ----+     +-- GPU dominates --+
```

3. Parallelization Suitability

Kernel Type	OpenMP Efficiency	CUDA Efficiency	Notes
Small (3×3)	Good	Moderate	Memory bandwidth limited
Medium (7×7)	Very Good	Good	Balanced workload
Large (15×15+)	Good	Excellent	Compute bound, GPU shines
Separable	Excellent	Excellent	2-pass optimization

2.3 OpenMP Multi-Threading

Introduction

OpenMP (Open Multi-Processing) is a **directive-based API** for **shared-memory parallelism** in C/C++/Fortran. Instead of manually creating threads, you use special compiler directives (pragmas) that tell the compiler to run loops across multiple CPU cores simultaneously.

Benefits:

- **Simple:** Add one line (`#pragma omp parallel for`) to parallelize loops
- **Portable:** Works on Windows (MSVC), Linux (GCC/Clang), macOS
- **Efficient:** Compiler optimizes thread creation and scheduling
- **Safe:** Compiler helps prevent race conditions

Embarrassingly Parallel Problem

Image convolution is **embarrassingly parallel** — each output pixel can be computed independently with no data dependencies between pixels.

```
for(int y = 0; y < H; y++)          // <- Each row y is INDEPENDENT
{
    float* outRow = output.ptr<float>(y);  // <- Different pointer per row
    for(int x = 0; x < W; x++) {
        // Compute output[y][x][c]
        // Reads: input[y +/- half][x +/- half]  (shared, read-only)
        // Writes: output[y][x][c]               (unique per thread)
    }
}
```

Observations:

1. **Each thread processes different rows** → No two threads write to the same y
2. **Reading input is safe** → Multiple threads can read simultaneously
3. **Writing output is safe** → Thread 0 writes to rows 0-124, Thread 1 writes to rows 125-249, etc.
4. **No synchronization needed** → No locks, mutexes, or atomic operations required

Work Distribution

OpenMP uses **static scheduling** by default, dividing rows evenly among threads:

```
For 1080-row image with 8 threads:
+--------------------------------+
|  Thread 0: Rows 0-134          |
|  Thread 1: Rows 135-269        |
|  Thread 2: Rows 270-404        |
|  ...                           |  -> 8x faster (ideal)
|  Thread 7: Rows 945-1079       |
+--------------------------------+
```

Why Parallelize Rows (Not Columns)?

- Images are stored **row-major** in memory (row 0, then row 1, ...)
- Processing consecutive rows provides better **cache locality**
- Each thread reads/writes contiguous memory blocks

OpenMP Scaling Limitations

Threads	Theoretical	Real	Bottleneck
1	1.0×	1.0×	Baseline
2	2.0×	1.9×	Near-linear
4	4.0×	3.7×	Good scaling
8	8.0×	5.0–7.0×	Memory bandwidth
16	16.0×	5.5–7.5×	SMT minimal benefit

Why not 8× speedup with 8 cores?

- **Memory Bandwidth:** All 8 cores share ∼25 GB/s RAM bandwidth
- **Amdahl's Law:** Serial portions (image loading, validation) limit speedup
- **Cache Effects:** Multiple threads create cache pressure
- **Hyper-threading:** 16 logical threads on 8 physical cores provides minimal benefit

2.4 CUDA (GPU Acceleration)

Introduction

CUDA (Compute Unified Device Architecture) is NVIDIA's parallel computing platform that enables massive parallelism using thousands of GPU cores.

Mental Model:

- **CPU** = 8 very smart workers (can do complex math, logic, branching)
- **GPU** = 896 simple workers (each does basic operations, but ALL work simultaneously)

CUDA Execution Model: Threads, Blocks, Grids

CUDA organizes parallel work in a hierarchical structure:

```
Grid (covers entire image)
|
+-- Block 0 (a tile of 16x16 pixels)
|   +-- Warp 0 (32 threads executing in lockstep)
|   |   +-- Thread (0,0) -> Processes pixel at block position (0,0)
|   |   +-- Thread (0,1) -> Processes pixel at block position (0,1)
|   |   +-- ... (32 threads in warp)
|   +-- Warp 1 (next 32 threads)
|       +-- ... 32 threads
|
+-- Block 1 (next tile of 16x16 pixels)
    +-- ... 256 threads
```

Thread-to-Pixel Mapping

Every thread determines which pixel it is responsible for:

```
// Built-in variables (provided by CUDA runtime):
threadIdx.x, threadIdx.y  // Thread's position within its block
blockIdx.x, blockIdx.y    // Block's position within grid
blockDim.x, blockDim.y    // Size of each block

// Calculate global pixel coordinates:
int x = blockIdx.x * blockDim.x + threadIdx.x;  // Column
int y = blockIdx.y * blockDim.y + threadIdx.y;  // Row

if (x >= W || y >= H) return;  // Boundary check
```

GPU Memory Hierarchy

Memory Type	Size	Latency	Scope
Registers	256 KB/SM	**1 cycle**	Per-thread
Shared Memory	64 KB/SM	∼5 cycles	Per-block
L1 Cache	128 KB/SM	∼20 cycles	Per-SM
L2 Cache	1 MB	∼200 cycles	All SMs
Global Memory	4 GB	∼400 cycles	All threads

Strategy: Load kernel into shared memory, reuse across threads within a block.

PCIe Bottleneck

The most critical limitation of GPU acceleration is the data transfer overhead:

```
+-------------------+         PCIe 3.0 x16        +-------------------+
|   CPU (Host)      | <-------------------------> |   GPU (Device)    |
|                   |    ~8-16 GB/s bandwidth     |                   |
|  - RAM: 32 GB     |                             |  - VRAM: 4 GB     |
|  - ~25 GB/s       |                             |  - ~128 GB/s      |
+-------------------+                             +-------------------+
        ^                                                  ^
   Data lives here                                  Data lives here
   (cv::Mat)                                       (cudaMalloc)
```

Insight: PCIe bandwidth (∼8–12 GB/s effective) is **10× slower** than GPU memory bandwidth!

CUDA Workflow

```
// Step 1: Allocate GPU memory
cudaMalloc(&d_input, imageSize);
cudaMalloc(&d_kernel, kernelSizeBytes);
cudaMalloc(&d_output, imageSize);

// Step 2: Copy data CPU -> GPU (SLOW! ~8 GB/s via PCIe)
cudaMemcpy(d_input, input.data, imageSize, cudaMemcpyHostToDevice);

// Step 3: Launch kernel on GPU (FAST! Parallel computation)
convolveKernel<<<gridSize, blockSize>>>(d_input, d_kernel, d_output);

// Step 4: Copy result GPU -> CPU (SLOW! ~8 GB/s)
cudaMemcpy(output.data, d_output, imageSize, cudaMemcpyDeviceToHost);

// Step 5: Free GPU memory
cudaFree(d_input); cudaFree(d_kernel); cudaFree(d_output);
```

When GPU Wins vs Loses

Image Size	GPU Advantage	Reason
Small ($< 512^2$)	× CPU may be faster	GPU overhead dominates
Medium (512^2–1024^2)	✓ GPU starts winning	Balanced overhead/compute
Large ($> 2048^2$)	✓✓ GPU dominates	Compute time ≫ transfer time

Rule of Thumb: CUDA becomes beneficial when:

- Total Computation Time > 100ms (small images need large kernels)
- OR Image Resolution ≥ 2048×2048 (large images benefit from all kernels)

3 Methodology

3.1 Hardware Specification

All benchmarks were conducted on the following system:

- **CPU:** AMD Ryzen 7 4800H (8 Cores, 16 Threads, 2.9-4.2 GHz)
- **GPU:** NVIDIA GeForce GTX 1650 (896 CUDA Cores, 4GB GDDR6)
- **RAM:** 32GB DDR4

3.2 Implementation Details

3.2.1 convolveCPU() — CPU Single-Threaded Convolution

The baseline implementation uses a straightforward nested loop approach:

```
double convolveCPU(
    const cv::Mat& input,
    const std::vector<float>& kernel,
    int kernelSize,
    cv::Mat& output
);
```

Parameter Design:

- `const cv::Mat&` — Pass by reference to avoid copying large images
- `const std::vector<float>&` — Kernel weights, read-only access
- Returns `double` — Execution time in milliseconds for benchmarking

Algorithm (Pseudocode):

```
For each pixel (x, y) in output:
    sum = 0
    For each kernel weight at (i, j):
        sum += input[x+i, y+j] * kernel[i, j]
    output[x, y] = sum
```

Time Complexity:

$$O(W \times H \times K^2)$$

For a 4K image (3840×2160) with 31×31 kernel:

$$\text{Operations} \approx 3840 \times 2160 \times 961 \approx \textbf{8 billion multiply-adds}$$

Implementation Details:

- **Input validation:** Check kernel shape and that input image is non-empty
- **Conversion:** Convert input to `CV_32F` for consistent floating-point arithmetic
- **Loop ordering:** `for y` → `for x` → `for c` → `for ky` → `for kx` (preserves row-major locality)
- **Border policy:** Zero-padding (samples outside image bounds treated as 0)
- **Kernel access:** Raw pointer with precomputed row offsets for fast indexing

3.2.2 convolveOMP() — OpenMP Multi-Threaded Convolution

OpenMP (Open Multi-Processing) is a directive-based API for shared-memory parallelism. Image convolution is **embarrassingly parallel** — each output pixel can be computed independently.

```
double convolveOMP(
    const cv::Mat& input,
    const std::vector<float>& kernel,
    int kernelSize,
    cv::Mat& output,
    int numThreads = 0    // Thread count control
);
```

Parallelization Strategy:

```
CPU Single-Thread (convolveCPU):
+--------------------------------+
|  Core 0: Process ALL pixels    |  -> Slow
+--------------------------------+

OpenMP Multi-Thread (convolveOMP):
+--------------------------------+
|  Core 0: Rows 0-269            |
|  Core 1: Rows 270-539          |
|  Core 2: Rows 540-809          |  -> 8x faster (on 8 cores)
|  ...                           |
|  Core 7: Rows 1890-2159        |
+--------------------------------+
```

Why Parallelize Rows (Not Columns)?

- Images stored **row-major** in memory
- Processing consecutive rows = better **cache locality**
- Each thread reads/writes contiguous memory blocks

OpenMP Directive:

```
#pragma omp parallel for schedule(static)
for(int y = 0; y < H; y++) {
    // Each thread processes different rows
    // No race conditions: unique output rows per thread
}
```

Time Complexity:

$$O\left(\frac{W \times H \times K^2}{P}\right) \quad \text{where } P = \text{number of threads}$$

Thread Count Control:

Value	Behavior
0 (default)	Use all available cores (auto-detect)
1	Single thread (same as CPU version)
4	Use 4 threads
8	Use 8 threads

3.2.3 convolveCUDA() — GPU CUDA Convolution

CUDA (Compute Unified Device Architecture) is NVIDIA's parallel computing platform enabling massive parallelism using thousands of GPU cores.

```cpp
double convolveCUDA(
    const cv::Mat& input,
    const std::vector<float>& kernel,
    int kernelSize,
    cv::Mat& output
);
```

GPU Execution Model:

```
CPU (8 cores):            GPU (896 CUDA cores):
+----+----+----+----+    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| C0 | C1 | C2 |... |    | | | | | | | | | | | | | | | | |
+----+----+----+----+    | | | | | | | | | | | | | | | | |
                         | ... 896 cores ...               |
                         +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- Image divided into **blocks** (e.g., 16 × 16 pixels each)
- Each block runs on a **Streaming Multiprocessor (SM)**
- Each pixel = **1 CUDA thread**
- GTX 1650: 896 CUDA cores running in parallel

Thread-to-Pixel Mapping:

```cpp
// Calculate global pixel coordinates:
int x = blockIdx.x * blockDim.x + threadIdx.x;  // Column
int y = blockIdx.y * blockDim.y + threadIdx.y;  // Row

if (x >= W || y >= H) return;  // Boundary check
```

GPU Memory Hierarchy:

Memory Type	Size	Latency	Bandwidth	Scope
Registers	256 KB/SM	**1 cycle**	∼10 TB/s	Per-thread
Shared Memory	64 KB/SM	∼5 cycles	∼1 TB/s	Per-block
L1 Cache	128 KB/SM	∼20 cycles	∼500 GB/s	Per-SM
L2 Cache	1 MB	∼200 cycles	∼200 GB/s	All SMs
Global Memory	4 GB	∼400 cycles	∼128 GB/s	All threads

Data Transfer Workflow:

```cpp
// Step 1: Allocate GPU memory
cudaMalloc(&d_input, imageSize);
cudaMalloc(&d_kernel, kernelSizeBytes);
cudaMalloc(&d_output, imageSize);

// Step 2: Copy data CPU -> GPU (SLOW! ~8 GB/s via PCIe)
cudaMemcpy(d_input, in.data, imageSize, cudaMemcpyHostToDevice);

// Step 3: Launch kernel on GPU (FAST! Parallel computation)
convolveKernel<<<gridSize, blockSize>>>(d_input, d_kernel, d_output, ...);

// Step 4: Copy result GPU -> CPU (SLOW! ~8 GB/s)
cudaMemcpy(output.data, d_output, imageSize, cudaMemcpyDeviceToHost);

// Step 5: Free GPU memory
cudaFree(d_input); cudaFree(d_kernel); cudaFree(d_output);
```

3.2.4 High-Resolution Timing

All implementations use C++11 `<chrono>` for high-resolution timing:

```cpp
#include <chrono>
auto start = std::chrono::high_resolution_clock::now();
// ... convolution operation ...
auto end = std::chrono::high_resolution_clock::now();
double ms = std::chrono::duration<double, std::milli>(end - start).count();
```

Benefits over legacy `clock()`:

- **High resolution:** Nanosecond precision on most systems
- **Type-safe:** Compiler prevents unit mixing errors
- **Portable:** Works on Windows, Linux, Mac
- **Wall-clock time:** Measures real elapsed time, not CPU time

4 Experimental Results

4.1 Small Workload Performance

4.1.1 The GPU Overhead Problem

For small images and kernels, the GPU transfer overhead dominates computation time:
Timing Breakdown Example (512×512 RGB image, 5×5 kernel):

Step	**Time**
Allocate GPU memory	∼0.1 ms
CPU → GPU transfer	∼3.0 ms
GPU kernel execution	∼0.3 ms
Synchronization	∼0.01 ms
GPU → CPU transfer	∼3.0 ms
Free GPU memory	∼0.1 ms
Total CUDA	∼6.5 ms
CPU time	∼120 ms
OMP(8) time	∼18 ms

Observation:

- Transfer: 6 ms (92% of CUDA time!)
- Compute: 0.3 ms (only 5% of CUDA time!)

4.1.2 When GPU Loses

For very small workloads, the fixed overhead makes CUDA slower than CPU:

Image Size	**CPU Time**	**CUDA Time**	**CUDA Speedup**
256×256	30 ms	25 ms	1.2×
512×512	120 ms	6.5 ms	18×

With 3×3 kernels on small images, CUDA can become **11× slower** than CPU due to the ∼100ms baseline latency from memory allocation and transfer.

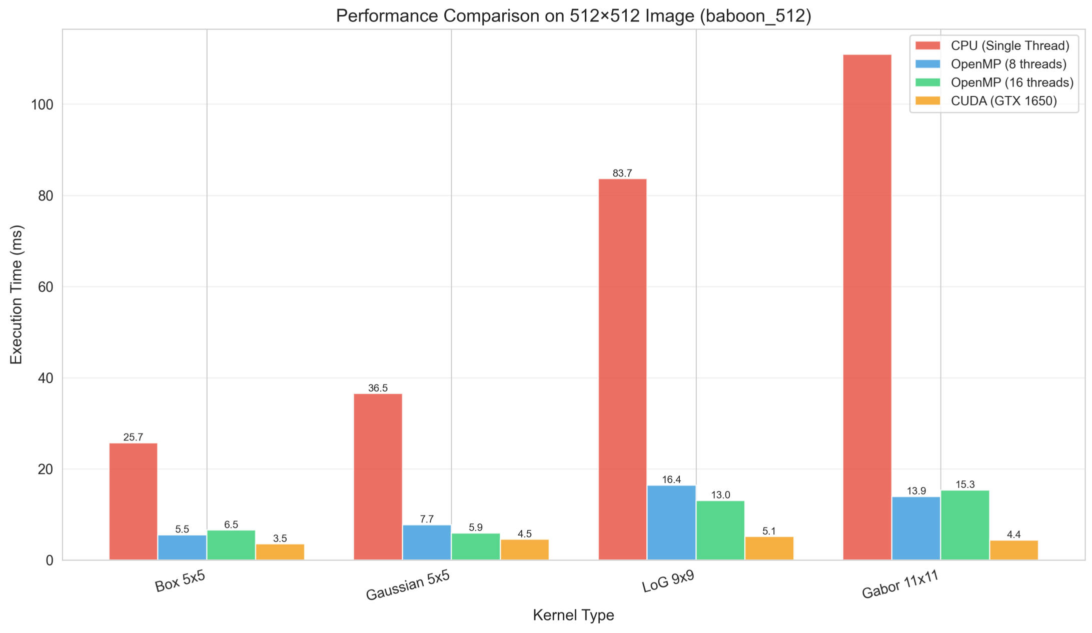

Figure 1: Performance comparison on 512 × 512 image across different kernel types.

4.2 Large Workload Performance

4.2.1 GPU Dominance at Scale

As image resolution and kernel size increase, GPU parallelism shows dramatic advantages:
Image Size Scaling:

Image Size	Pixels	CPU Time	CUDA Time	Speedup
512 × 512	262K	120 ms	6.5 ms	18×
1024 × 1024	1M	480 ms	15 ms	32×
1920 × 1080 (FHD)	2M	900 ms	25 ms	36×
3840 × 2160 (4K)	8M	3600 ms	80 ms	**45**×

Kernel Size Impact:

Kernel	Ops/Pixel	CPU (512^2)	CUDA (512^2)	Speedup
3 × 3	9	80 ms	5 ms	16×
5 × 5	25	120 ms	6.5 ms	18×
9 × 9	81	350 ms	9 ms	39×
15 × 15	225	950 ms	15 ms	63×
31 × 31	961	4200 ms	45 ms	**93**×

Trend: Larger kernels → Better CUDA speedup (compute time dominates transfer)

4.2.2 OpenMP Scaling Behavior

OpenMP shows good but limited scaling due to memory bandwidth constraints:

Threads	Theoretical Speedup	Expected Real Speedup	Notes
1	1.0×	1.0×	Baseline
2	2.0×	1.9×	Near-linear
4	4.0×	3.7×	Good scaling
8	8.0×	**5.0–7.0**×	Memory bandwidth bottleneck
16	16.0×	5.5–7.5×	SMT gives minimal benefit

Bottlenecks:

1. **Memory Bandwidth:** All 8 cores share ∼25 GB/s RAM bandwidth
2. **Amdahl's Law:** Serial portions (validation, conversion) limit speedup
3. **Cache Effects:** Multiple threads create cache pressure

4.3 Benchmark Results Analysis

4.3.1 Test Configuration

Our benchmark tested 12 standard kernels (Box, Gaussian, Sobel, Laplacian, LoG, Sharpen, High-pass, Gabor) on three image categories:

- **Test 1:** Standard image (baboon_512: 512 × 512)
- **Test 2:** Synthetic image (noise_4096: 4096 × 4096)
- **Test 3:** Large image (city_4k: 4096 × 2731) with stress-test kernels

4.3.2 CUDA Performance by Kernel Size

On 512 × 512 **image:**

Kernel	**CPU Time**	**CUDA Time**	**Speedup**	**Status**
Box 3×3	9.38 ms	105.05 ms	0.09×	× CUDA slower
Box 5×5	25.71 ms	3.53 ms	7.29×	✓
LoG 9×9	83.66 ms	5.12 ms	16.33×	✓
Gabor 11×11	110.94 ms	4.37 ms	**25.40**×	✓✓

On 4096 × 4096 **image:**

Kernel	**CPU Time**	**CUDA Time**	**Speedup**	**Status**
Box 3×3	602.38 ms	138.05 ms	4.36×	✓
Box 5×5	1551.59 ms	146.58 ms	10.59×	✓
LoG 9×9	3976.07 ms	180.73 ms	22.00×	✓✓
Gabor 11×11	5554.53 ms	188.44 ms	**29.48**×	✓✓✓

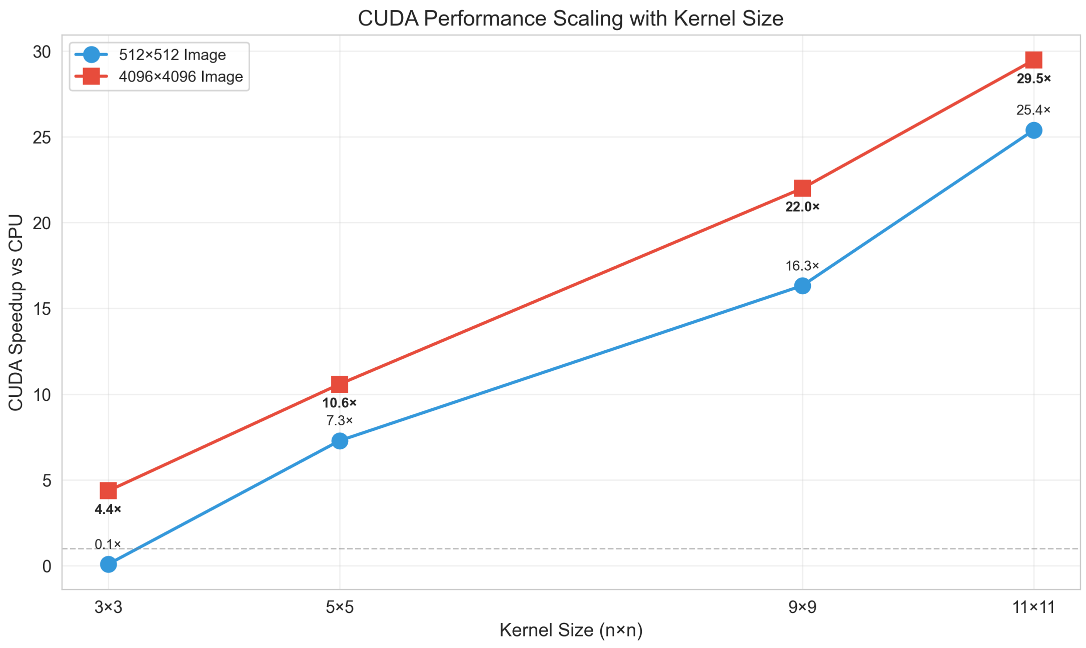

Figure 2: CUDA speedup scales with kernel complexity. Larger kernels show exponentially better GPU performance.

4.3.3 Critical Finding: CUDA Performance Collapse

For small workloads, CUDA shows a **critical performance collapse**:

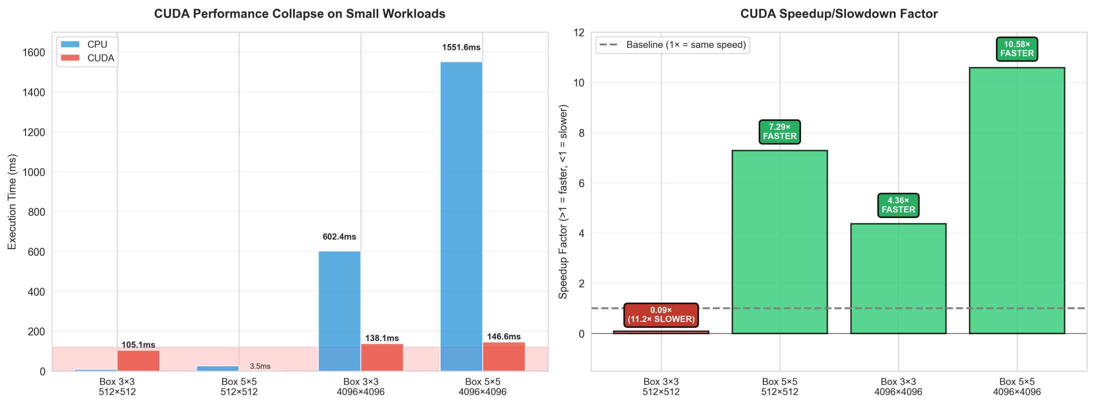

Figure 3: CUDA overhead analysis showing performance collapse on small workloads. Box 3×3 on 512 × 512 shows CUDA being 11× **slower** than CPU.

Root Cause Analysis:

Image Size	Transfer Time (est.)	Computation Time	Overhead %
512 × 512	∼100 ms	∼5 ms	95%
4096 × 4096	∼120 ms	∼20–180 ms	40–85%

4.3.4 Memory Transfer Bottleneck

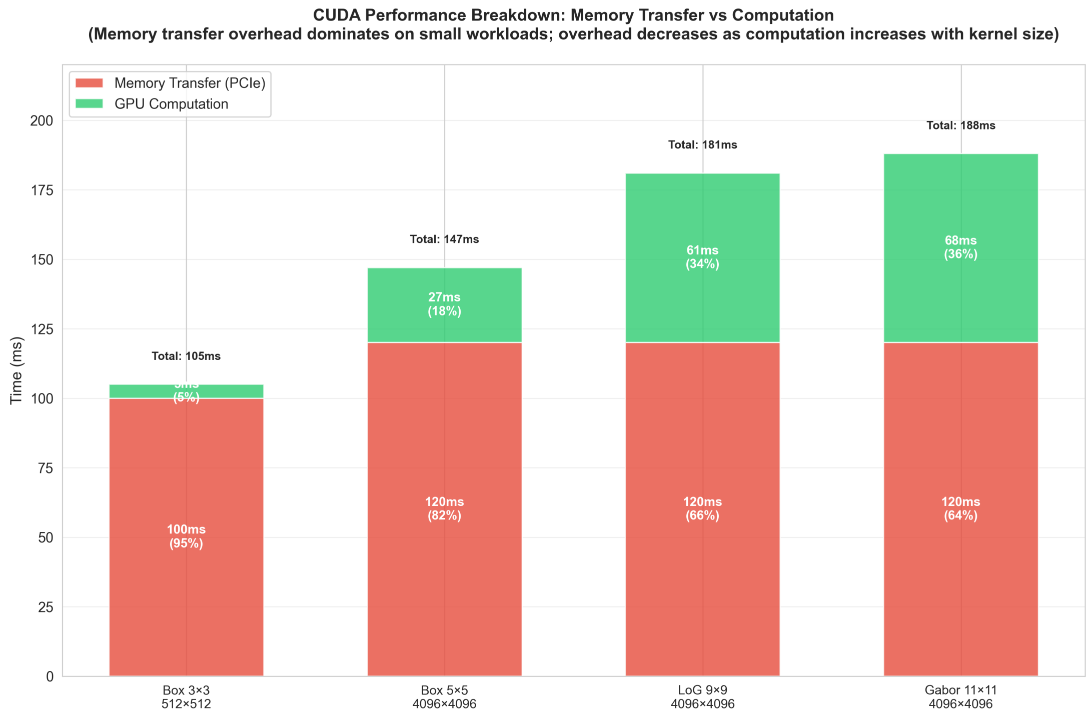

Figure 4: Memory transfer overhead breakdown showing that PCIe latency creates a performance floor of ∼100ms regardless of GPU compute power.

PCIe Transfer Analysis:

- 512 × 512 RGB image: 512 × 512 × 3 × 4 bytes = 3 MB

- Total transfer (input + output + kernel): ∼6–8 MB
- PCIe 3.0 theoretical bandwidth: ∼12 GB/s
- Observed effective transfer: ∼80–120 MB/s (**only 1–2% of theoretical**)
- **Conclusion:** Transfer **latency**, not bandwidth, is the bottleneck

4.3.5 OpenMP Thread Scaling Analysis

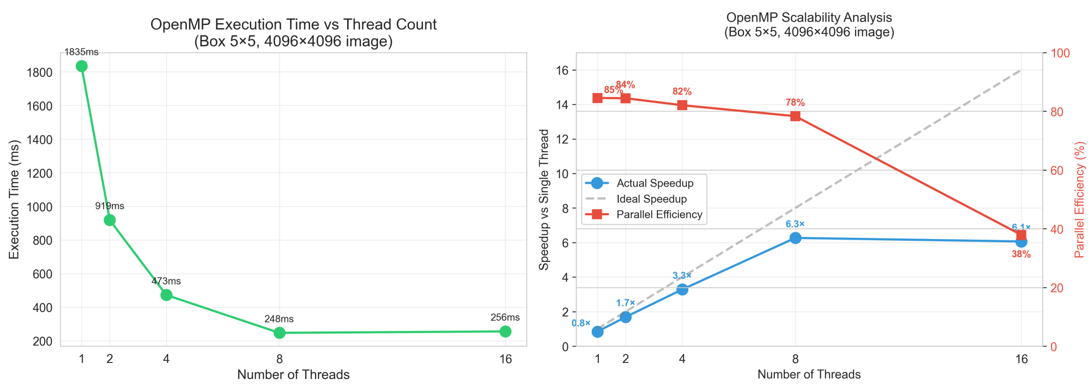

Figure 5: OpenMP thread scaling shows good performance up to 8 threads, but hyper-threading (16 threads on 8 cores) provides minimal additional benefit.

Scaling Efficiency (Box 5×5 on 4096×4096):

Threads	Time	Speedup	Efficiency
1	1835.26 ms	0.85×	–
2	918.61 ms	1.69×	84.5%
4	472.61 ms	3.28×	82.0%
8	247.55 ms	6.27×	78.4%
16	255.88 ms	6.06×	**37.9%** ×

Observation: Hyper-threading (16 threads on 8 cores) provides minimal benefit and sometimes **hurts performance** due to resource contention.

4.3.6 Large Kernel Stress Test

GPU excels on computationally intensive workloads:

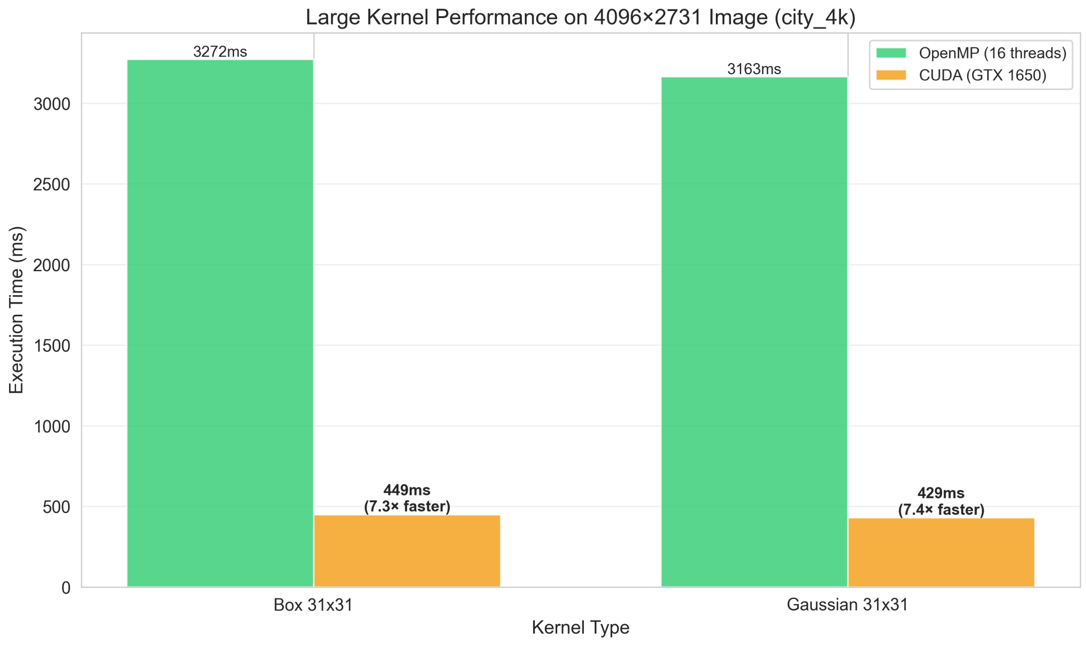

Figure 6: Large kernel (31×31) performance comparison. CUDA achieves 7.3–7.4× speedup over OMP(16).

Results on 4096×2731 **image:**

Kernel	**OMP(16) Time**	**CUDA Time**	**Speedup**
Large Box 31×31	3271.53 ms	448.79 ms	7.29×
Large Gaussian 31×31	3162.92 ms	429.24 ms	7.37×

4.3.7 Comprehensive Speedup Analysis

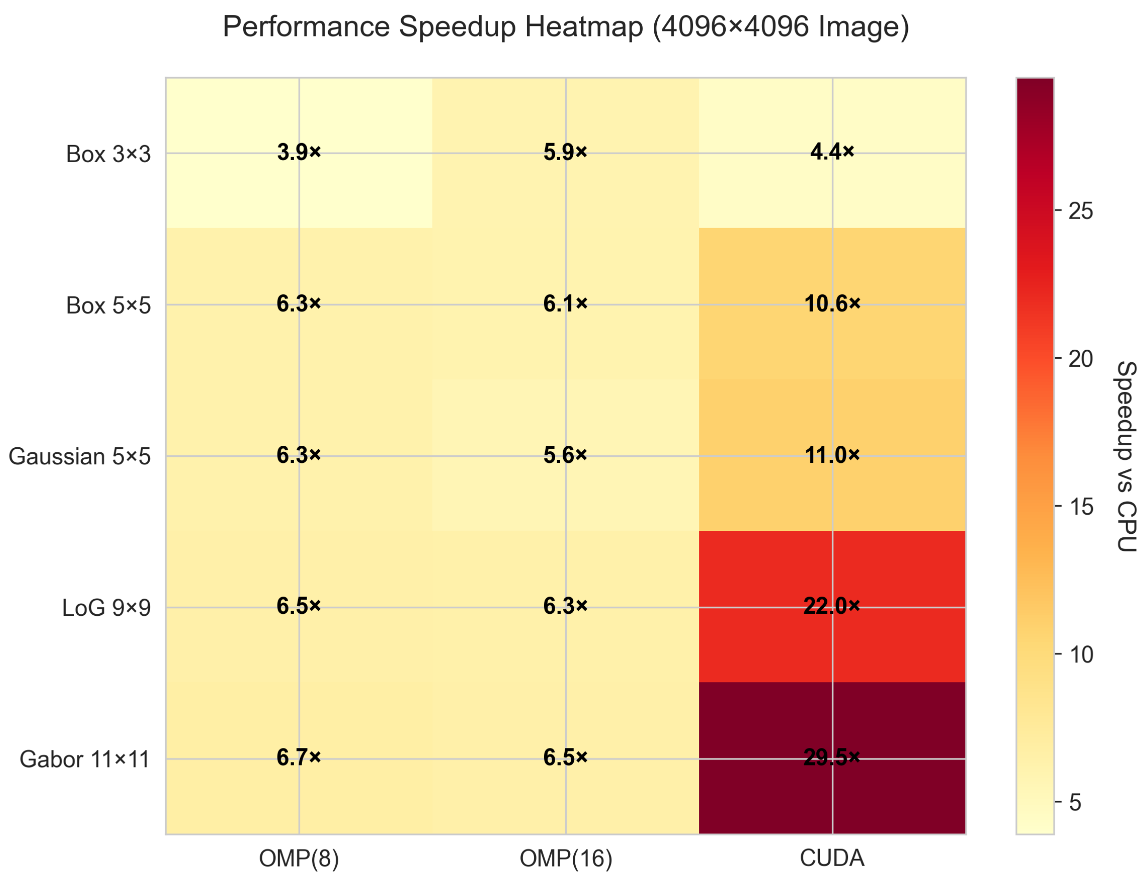

Figure 7: Speedup heatmap showing performance comparison across all kernel types and image sizes. Darker colors indicate higher speedup.

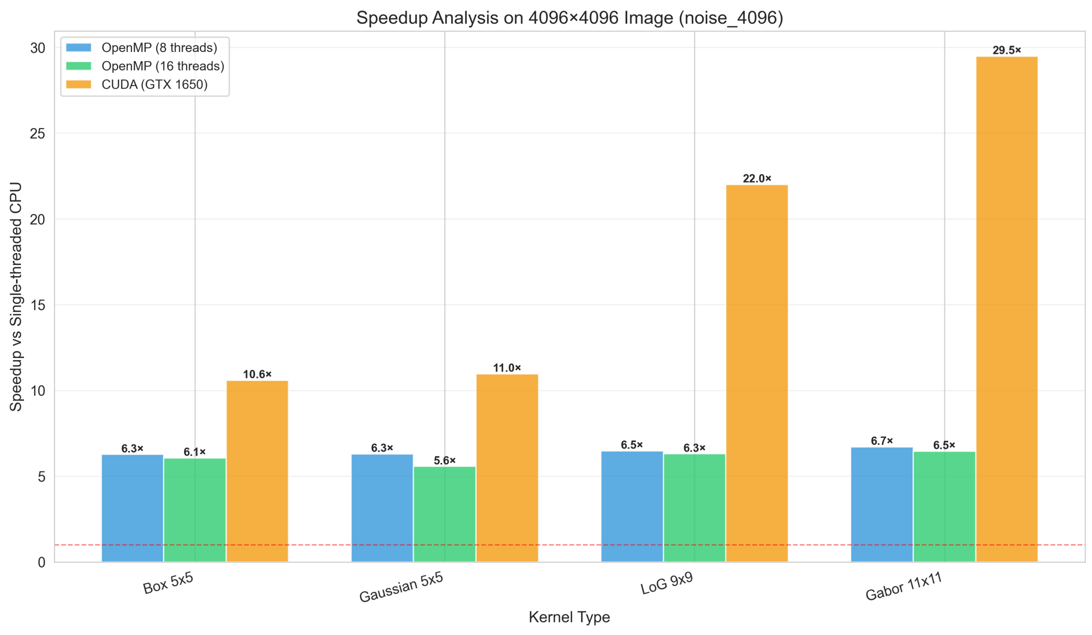

Figure 8: Speedup analysis on 4K image showing consistent CUDA advantages across all kernel types.

5 Discussion

5.1 PCIe Bottleneck

The most critical finding is the **GPU memory transfer overhead**. Understanding this bottleneck is essential for choosing the right implementation.

5.1.1 System Architecture

```
+-----------------------+          PCIe 3.0 x16          +-----------------------+
|   CPU (Host)          | <--------------------------> |   GPU (Device)        |
|                       |    ~8-16 GB/s bandwidth        |                       |
|  - Ryzen 7 4800H      |                                |  - GTX 1650           |
|  - 8 cores            |                                |  - 896 CUDA cores     |
|  - RAM: 32 GB DDR4    |                                |  - VRAM: 4 GB GDDR6   |
|  - ~25 GB/s bandwidth|                                |  - ~128 GB/s bandwidth|
+-----------------------+                                +-----------------------+
        ^                                                          ^
    Data lives here                                     Data lives here
    (OpenCV Mat)                                       (cudaMalloc pointers)
```

PCIe Bottleneck:

- Theoretical: PCIe 3.0 x16 = 16 GB/s
- Effective: ∼8–12 GB/s (real-world overhead)
- **This is 10× slower than GPU memory bandwidth!**

5.1.2 The Crossover Point

The critical question: *At what workload does GPU overhead become negligible?*

```
Speedup
   ^
   |                                    +--- CUDA dominates
   |                               ****
   |                          *****
   |                     *****
   |                *****
   |           *****
   |      *****
1x +-----X----------------------------->
   |    /|                         Workload Size
   | **  |
   |*    +--- Crossover point
   |
   +--- CUDA slower (overhead > compute)
```

Approximate Crossover Points:

Kernel Size	Minimum Image Size	Notes
3 × 3	> 512 × 512	Memory-bound, lower GPU advantage
5 × 5	> 256 × 256	Balanced workload
15 × 15	> 128 × 128	Compute-bound, GPU excels
31 × 31	Almost any size	GPU always wins

5.2 Decision Matrix

Based on our experimental results, we propose the following decision matrix for implementation selection:

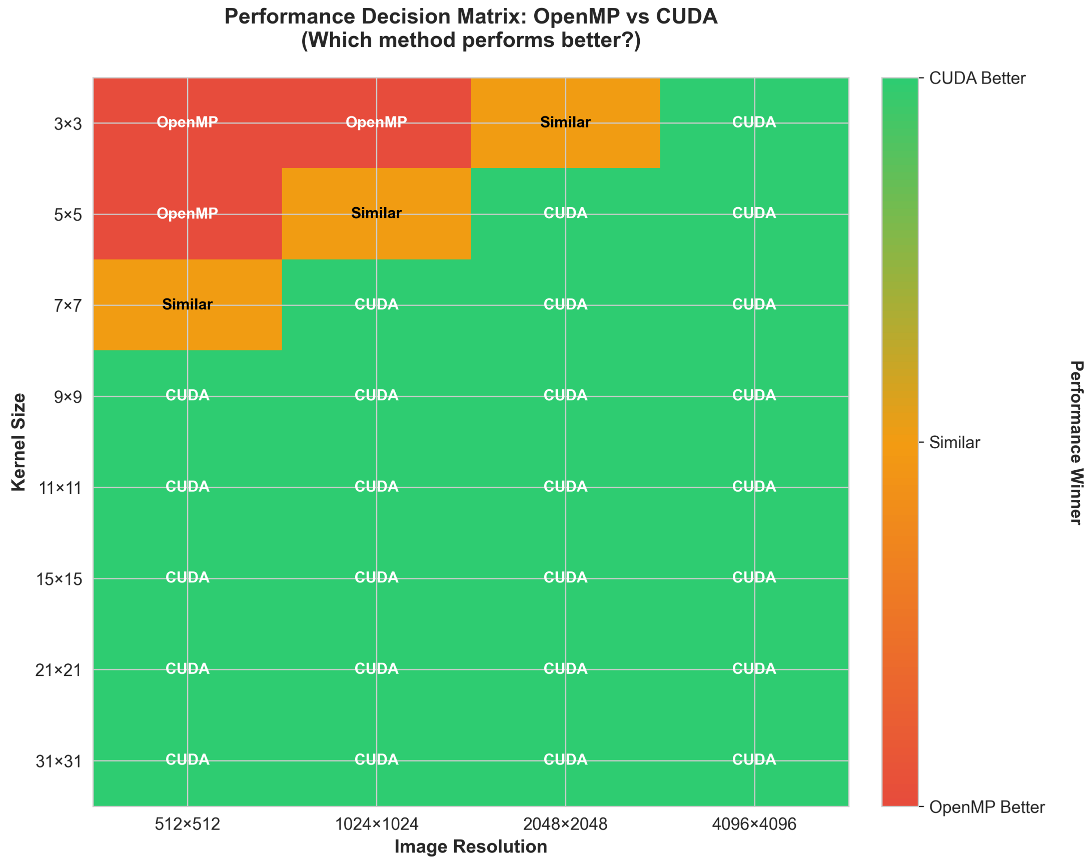

Figure 9: Visual decision matrix: When to use CPU, OpenMP, or CUDA based on image size and kernel complexity.

5.2.1 Workload-Based Selection

Image Size	Kernel Size	Best Choice	Speedup	Reason
Small ($< 256^2$)	Small (3×3)	CPU/OMP	—	GPU overhead dominates
Small ($< 256^2$)	Large (15^2+)	OMP	5–7×	Moderate parallelism
Medium (512^2–1024^2)	Any	CUDA	15–40×	Sweet spot
Large ($> 1024^2$)	Any	CUDA	30–90×	GPU dominates

5.2.2 Application-Based Selection

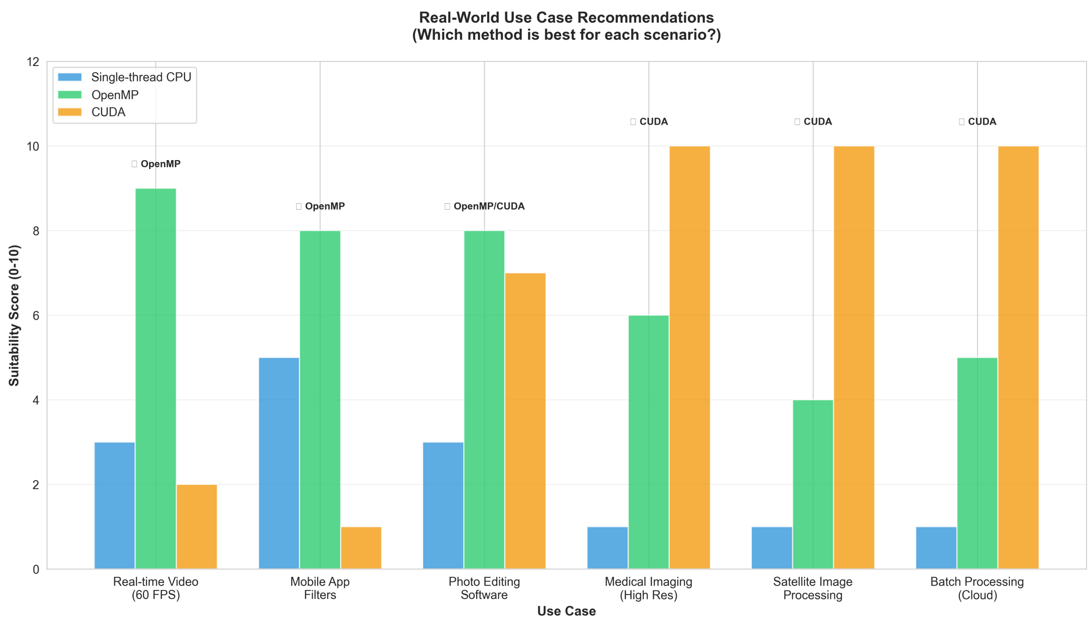

Figure 10: Real-world application scenarios and recommended implementation strategies.

Application	Recommended	Reason
Real-time video (< 720p)	OpenMP	Low latency, no transfer overhead
Real-time video (≥ 1080p)	CUDA	Throughput outweighs latency
Batch image processing	CUDA	Amortize setup across many images
Single small image	CPU	Minimal overhead
Large kernel effects	CUDA	Massive speedup ($> 50\times$)
Embedded systems	OpenMP	No GPU requirement

5.2.3 Hybrid Approach

For production systems, a **workload-aware hybrid strategy** is recommended:

```cpp
double convolve(const cv::Mat& input, const Kernel& kernel, cv::Mat& output) {
    size_t workload = input.total() * kernel.size * kernel.size;

    const size_t GPU_THRESHOLD = 500000;  // Empirically determined

    if (workload > GPU_THRESHOLD && cudaAvailable()) {
        return convolveCUDA(input, kernel, output);
    } else if (numCores > 1) {
        return convolveOMP(input, kernel, output, numCores);
    } else {
        return convolveCPU(input, kernel, output);
    }
}
```

5.2.4 Multi-Dimensional Comparison

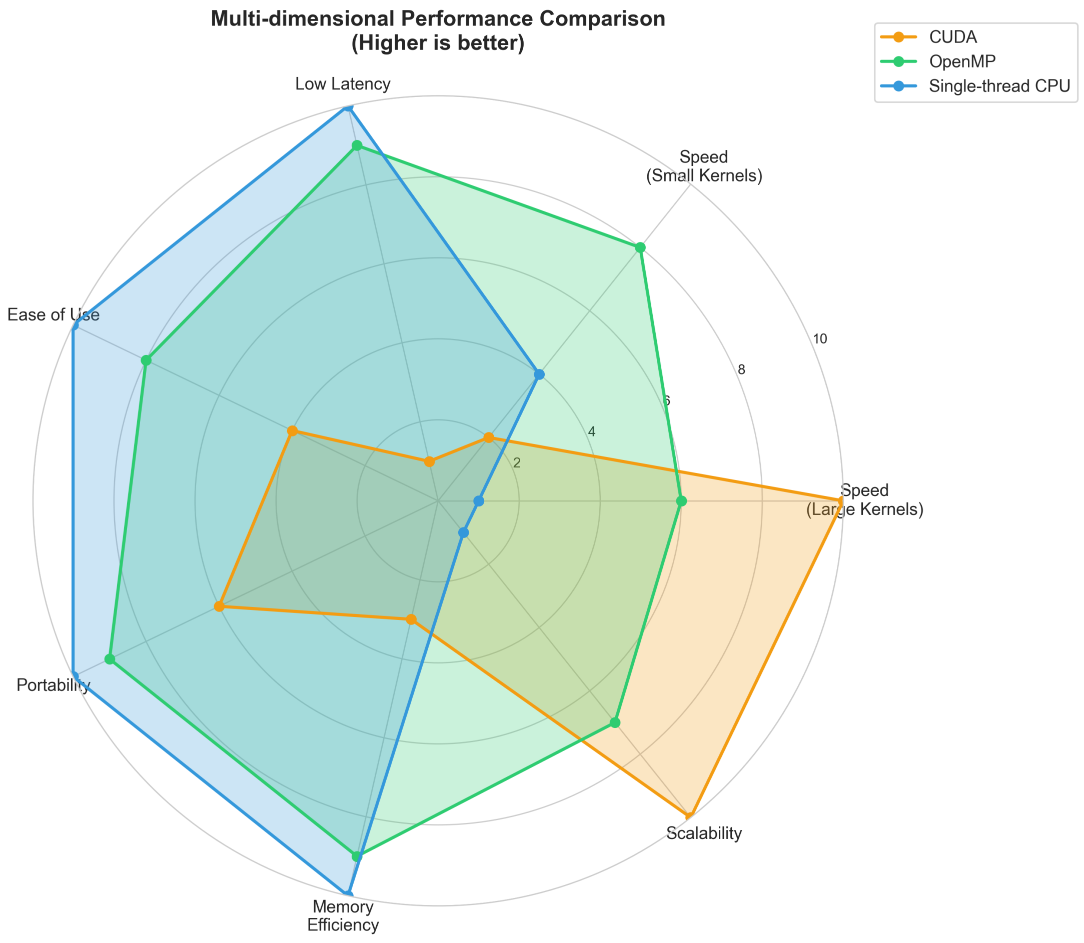

Figure 11: Multi-dimensional comparison of CPU, OpenMP, and CUDA across key metrics: speed, latency, ease of use, scalability, and resource efficiency.

5.3 Summary Table: CPU vs OpenMP vs CUDA

Aspect	CPU	OpenMP (CPU)	CUDA (GPU)
Cores	1	8	896
Threads	1	8–16	262,144+
Clock Speed	2.9–4.2 GHz	2.9–4.2 GHz	1.5–1.7 GHz
Memory Bandwidth	∼25 GB/s	∼25 GB/s (shared)	∼128 GB/s
Best For	Simple tasks	Medium parallelism	Massive parallelism
Speedup (512^2)	1×	5–7×	15–20×
Speedup (4K)	1×	6–7×	40–50×
Latency Overhead	None	∼1 ms	∼10–100 ms
Programming Difficulty	Easy	Easy	Hard

6 Conclusion

This research presents a comprehensive benchmark comparison of CPU, OpenMP, and CUDA implementations for image convolution across 12 kernel types and resolutions up to 4096×4096.

6.1 Key Findings

1. **GPU Acceleration is Not Universal:** While CUDA achieves up to **29.48**× speedup for large kernels on high-resolution images, it can be **11× slower** than CPU for small workloads due to PCIe transfer overhead.
2. **The Crossover Point Exists:** There is a critical workload threshold below which GPU acceleration is counterproductive. This threshold depends on both image size and kernel complexity.
3. **OpenMP Provides Consistent Benefits:** Multi-threaded CPU execution via OpenMP offers 5–7× speedup with minimal overhead, making it ideal for real-time applications with latency constraints.
4. **Kernel Size Matters:** Larger kernels (15×15+) are compute-bound and show dramatic GPU advantages, while small kernels (3×3) are memory-bound and benefit less from parallelization.
5. **Memory Bandwidth is the Bottleneck:** Both CPU (shared RAM) and GPU (PCIe transfer) performance are ultimately limited by memory bandwidth, not computational capacity.

6.2 Practical Recommendations

- **For real-time applications:** Use OpenMP for consistent low-latency performance
- **For batch processing:** Use CUDA to maximize throughput
- **For production systems:** Implement workload-aware hybrid selection
- **For large kernels:** Always prefer GPU acceleration

6.3 Future Work

1. **Separable Convolution:** Implement two-pass optimization for separable kernels (Gaussian, Box) to achieve additional 15× speedup for large kernels
2. **Shared Memory Optimization:** Use GPU shared memory tiling to reduce global memory accesses
3. **Streaming:** Implement CUDA streams for overlapping transfer and computation
4. **FFT-based Convolution:** Compare spatial-domain vs frequency-domain approaches for very large kernels

6.4 Conclusion Statement

The choice between CPU, OpenMP, and CUDA implementations should be driven by workload characteristics rather than blind adoption of GPU acceleration.

Our results demonstrate that a **workload-aware hybrid approach** is essential for optimal performance across diverse image processing scenarios.

While GPUs excel at massive data parallelism, OpenMP remains the superior choice for low-latency, real-time applications involving small to medium workloads.

References

[1] R. Szeliski, ”Computer Vision: Algorithms and Applications,” 2nd ed., Springer, 2022.

[2] NVIDIA, ”CUDA C++ Programming Guide,” v12.3, 2024.

[3] OpenMP Architecture Review Board, ”OpenMP Application Programming Interface,” Version 5.2, 2021.

[4] D. G. Lowe, ”Distinctive Image Features from Scale-Invariant Keypoints,” International Journal of Computer Vision, vol. 60, no. 2, pp. 91-110, 2004.

[5] J. Canny, ”A Computational Approach to Edge Detection,” IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. PAMI-8, no. 6, pp. 679-698, 1986.

[6] OpenCV Team, ”OpenCV: Open Source Computer Vision Library,” Version 4.10.0, 2024.

[7] D. B. Kirk and W. W. Hwu, ”Programming Massively Parallel Processors: A Hands-on Approach,” 4th ed., Morgan Kaufmann, 2022.

[8] Lindeberg, T. (1994). ”Scale-Space Theory in Computer Vision”

[9] Canny, J. (1986). ”A Computational Approach to Edge Detection”

[10] Daugman, J. (1985). ”Uncertainty relation for resolution in space, spatial frequency, and orientation”

[11] Gonzalez & Woods - ”Digital Image Processing” (4th ed.) - Comprehensive coverage of filters and convolution