# LBP Parallelization with CUDA

Matteo Tinacci
matteo.tinacci@edu.unifi.it

## Abstract

*In this project, a high-performance implementation of the image processing algorithm Local Binary Patterns (LBP) was developed using the NVIDIA CUDA architecture. Starting from a sequential CPU version, several parallelization strategies were implemented: Naive (Global Memory), Shared Memory, Texture/Read-Only Cache optimizations, and CUDA Streams. The performance analysis demonstrates a maximum* ***computational speedup*** *of* ***284x*** *compared to the sequential CPU version and* ***65x*** *compared to an OpenMP implementation (8 threads). Contrary to initial expectations, the standard* ***Global Memory*** *approach achieved the lowest kernel execution time (1.58 ms), proving that modern hardware caches effectively handle spatial locality without manual intervention. Finally, the implementation of an asynchronous pipeline using* ***CUDA Streams*** *and Pinned Memory effectively masked data transfer latency, saturating the PCIe bus and achieving a system throughput of* ***10.6 GB/s****.*

## 1. Introduction

*Local Binary Patterns* (LBP)[1] is a texture descriptor operator widely used in the fields of Computer Vision and Pattern Recognition. Thanks to its computational efficiency and robustness to monotonic grayscale variations, LBP has established itself as a fundamental tool in numerous applications, including face recognition, biomedical texture analysis, and automated surveillance.

However, with the exponential increase in image resolution (4K, 8K) and the need to analyze video streams in real time, the sequential execution of the algorithm on traditional CPUs quickly becomes a bottleneck. Although the algorithm is aritmetically simple, requiring only comparisons and bitwise operations, it requires iterating over millions of pixels for each frame. This characteristic shifts the performance bottleneck from computation to memory bandwidth (*memory-bound*), making the serial approach inadequate for massive datasets or *time-critical* systems. Nevertheless, the nature of the algorithm, where the computation of each pixel is independent of its neighbors, makes it an ideal candidate for massive parallelization on GPU architectures through the CUDA programming model.

The objective of this report is to analyze and optimize the performance of the LBP algorithm by exploiting the hardware capabilities of modern NVIDIA GPUs. The work is structured through several incremental steps:

- The implementation of a **CPU baseline**, both sequential and parallelized with OpenMP, to establish a reliable performance reference ( 450 ms vs  102 ms).
- The development of CUDA kernels evaluating different memory strategies, contrasting the overhead of manual **Shared Memory** management against the efficiency of the **Read-Only Cache** (via the texture pipeline and `__ldg` intrinsic).
- The analysis of block geometry (*block size tuning*) to identify configurations that maximize occupancy and ensure perfect *memory coalescing* (finding the optimal warp-aligned dimensions).
- The implementation of **CUDA Streams** to maximize system throughput (up to 10.6 GB/s) by overlapping computation with memory transfers, mitigating the PCIe bus latency in batch processing scenarios.

## 2. Algorithm and Architecture

### 2.1. Local Binary Patterns (LBP)

Local Binary Patterns is a simple and efficient texture descriptor that labels the pixels of an image by thresholding the neighborhood of each pixel and interpreting the result as a binary number.

In its classical variant ($P = 8, R = 1$), the algorithm operates on a **single-channel grayscale image** using a sliding window of size $3 \times 3$. For each central pixel $g_c$ at coordinates $(x, y)$, the 8 neighboring pixels $g_p$ are considered. The value of each neighbor is compared with that of the center:

- If the neighbor's intensity is greater than or equal to the center's intensity, the bit is set to 1.
- Otherwise, the bit is set to 0.

The resulting bits are concatenated in a specific order (e.g., clockwise starting from the top-left) to form an 8-bit integer (ranging from 0 to 255), which represents the LBP code of that pixel.

Mathematically, the LBP code for a pixel at coordinates $(x, y)$ is defined as:

$$LBP_{P,R}(x,y) = \sum_{p=0}^{P-1} s(g_p - g_c)\, 2^p \tag{1}$$

where $g_c$ is the intensity value of the central pixel, $g_p$ is the intensity of the $p$-th neighbor, and $s(z)$ is the threshold (step) function defined as:

$$s(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \tag{2}$$

**Computational Characteristics:** From an optimization perspective, LBP is characterized by **low arithmetic intensity**. For every output pixel computed, the kernel must fetch 9 distinct values (the center plus 8 neighbors) but performs only basic integer operations (subtractions, comparisons, shifts). This implies that the performance is strictly limited by the rate at which data can be delivered from memory to the compute units (*memory-bound*), rather than by the speed of the floating-point units.

Figure 1. Input image.

### 2.2. CUDA Execution Model

The NVIDIA CUDA architecture adopts a massively parallel execution model known as SIMT (Single Instruction, Multiple Threads). Execution is hierarchically organized as follows:

- **Grid:** The complete set of threads launched to execute a kernel. In our case, the Grid maps the entire image resolution (Width $\times$ Height).
- **Block:** The Grid is divided into blocks of threads. Threads within the same block can cooperate through fast *Shared Memory* and synchronize with each other.

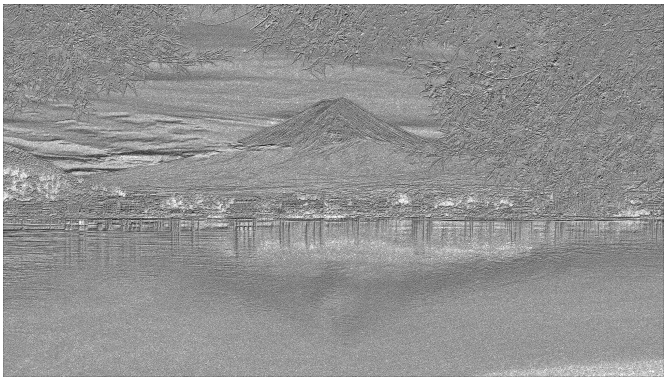

Figure 2. Output image processed with LBP.

- **Thread:** The individual execution unit. In the LBP implementation, we adopt a one-to-one mapping where each thread is responsible for computing a single pixel of the output image.

**Warps and Memory Coalescing:** At the hardware level, threads are grouped and executed in units of 32 called *Warps*. The LBP algorithm is particularly well suited to this architecture because the core computation exhibits a highly regular control flow, minimizing instruction divergence.

However, the most critical aspect for performance is the memory access pattern. Since threads in a Warp execute in lock-step, when consecutive threads access consecutive memory addresses, the GPU hardware can coalesce these requests into a single memory transaction. This makes the choice of block dimensions crucial: a block width that is a multiple of the warp size (32) ensures that memory reads are aligned, maximizing the effective bandwidth.

## 3. Implementation Details

This section analyzes the different parallelization strategies adopted, starting from a sequential baseline and progressing to advanced GPU implementations that exploit the memory hierarchy and asynchronous concurrency.

### 3.1. CPU Baseline (Sequential and OpenMP)

The sequential version serves as the reference for performance evaluation. The algorithm iterates over each internal pixel of the image using two nested loops, computing the LBP code through comparisons with the 8 neighbors and bitwise operations.

Multi-core acceleration is achieved using OpenMP, as shown in Listing 1. The `#pragma omp parallel for` directive distributes the computation across the available threads. The use of the `collapse(2)` clause is crucial: it linearizes the loops over rows and columns into a single iteration space, improving workload balancing (*load*

```cpp
void lbp_process_omp(const unsigned char* input,
                     unsigned char* output,
                     int width, int height) {

    #pragma omp parallel for collapse(2) schedule(
        ↪ static)
    for (int y = 1; y < height - 1; y++) {
        for (int x = 1; x < width - 1; x++) {
            unsigned char center = input[y * width +
                ↪ x];
            unsigned char code = 0;

            code |= (input[(y-1)*width + (x-1)] >=
                ↪ center) << 7;
            code |= (input[(y-1)*width + (x)  ] >=
                ↪ center) << 6;
            code |= (input[(y-1)*width + (x+1)] >=
                ↪ center) << 5;
            code |= (input[(y)  *width + (x+1)] >=
                ↪ center) << 4;
            code |= (input[(y+1)*width + (x+1)] >=
                ↪ center) << 3;
            code |= (input[(y+1)*width + (x)  ] >=
                ↪ center) << 2;
            code |= (input[(y+1)*width + (x-1)] >=
                ↪ center) << 1;
            code |= (input[(y)  *width + (x-1)] >=
                ↪ center) << 0;

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

Listing 1. CPU Parallelization with OpenMP

```cpp
__global__ void lbp_kernel(const unsigned char* in,
                           unsigned char* out,
                           int w, int h) {
    // 1. Thread-to-pixel mapping (x, y)
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    // 2. Boundary check (ignore borders)
    if (x < 1 || y < 1 || x >= w - 1 || y >= h - 1)
        ↪ return;

    // 3. LBP computation
    int idx = y * w + x;
    unsigned char center = in[idx];
    unsigned char code = 0;

    // Comparison with the 8 neighbors
    code |= (in[(y-1)*w + (x-1)] >= center) << 7;
    code |= (in[(y-1)*w + (x)  ] >= center) << 6;
    code |= (in[(y-1)*w + (x+1)] >= center) << 5;
    code |= (in[(y)  *w + (x+1)] >= center) << 4;
    code |= (in[(y+1)*w + (x+1)] >= center) << 3;
    code |= (in[(y+1)*w + (x)  ] >= center) << 2;
    code |= (in[(y+1)*w + (x-1)] >= center) << 1;
    code |= (in[(y)  *w + (x-1)] >= center) << 0;

    out[idx] = code;
}
```

Listing 2. CUDA LBP Kernel (Core Logic)

*balancing*) and ensuring that all cores are utilized regardless of the image aspect ratio.

### 3.2. GPU Parallelization (Naive)

The first CUDA implementation, referred to as *Naive*, adopts a *fine-grained* parallelization strategy: each thread is responsible for computing a single pixel of the image (Listing 2).

All data are stored in **Global Memory**. Each thread computes its global coordinates $(x, y)$ by combining the block indices with the local thread indices. Since the LBP operator requires a $3 \times 3$ neighborhood, a boundary check is implemented to disable threads mapped to the outer image border, thus preventing invalid memory accesses. Despite its simplicity, this kernel benefits from the hardware's ability to coalesce memory accesses when threads in a warp read consecutive addresses (e.g., the central pixel load).

### 3.3. Control Flow and Warp Divergence

Minimizing warp divergence is critical for maximizing GPU throughput. In our LBP implementation, we distinguish between *algorithmic divergence* (which we eliminated) and *structural divergence* (which we minimized):

- **Branchless LBP Logic (Algorithmic):** The core LBP algorithm performs 8 comparisons per pixel ($pixel_{neighbor} \geq pixel_{center}$). Implementing this with standard `if-else` statements would cause massive divergence within every warp. Instead, we utilized **branchless boolean arithmetic**:

  ```
  code |= (neighbor >= center) << bit_pos;
  ```

  Modern NVCC compilers translate these logical comparisons into *predicated instructions* (e.g., `ISETP`) or direct bitwise operations, effectively removing control flow divergence from the computation body entirely.

- **Boundary Handling (Structural):** A boundary check (`if (x<1 || ...)`) is strictly necessary to prevent out-of-bounds memory access. While this instruction creates a branch, it induces divergence **only in the warps mapped to the image perimeter**. Due to the spatial locality of thread mapping, the vast majority of warps fall entirely within the inner region of the image ("active region"). These warps execute the "else" path in perfect lock-step. The performance penalty is therefore statistically negligible, as it scales with the image perimeter ($2(W + H)$) rather than the area ($W \times H$).

### 3.4. Memory Optimizations

Direct access to Global Memory in image processing algorithms often leads to redundant memory accesses: neighboring pixels are read multiple times by adjacent threads. To optimize the data access pattern, two distinct strategies were implemented by exploiting the on-chip memory hierarchy of the GPU.

### 3.4.1 Shared Memory (Tiling with Halo)

This implementation aims to reduce global memory access latency by exploiting *Shared Memory* as a programmable cache (user-managed cache). As shown in Listing 3, the main challenge is addressed through the *Tiling with Halo* technique. Since the LBP operator has a radius of 1 pixel (a $3 \times 3$ neighborhood), threads located at the boundaries of a computation block need to access pixels that spatially belong to adjacent blocks.

The adopted strategy is structured into four phases:

1. **Allocation with Padding:** A buffer of size $(BW + 2) \times (BH + 2)$ is allocated in `__shared__` memory. This extra space hosts the surrounding frame (*Halo* or *Apron*) required by the stencil.

2. **Cooperative Loading:** All threads load their corresponding central pixel from global memory into shared memory. Subsequently, only threads located on the block boundaries perform additional reads to load the Halo pixels, handling corner cases and image borders.

3. **Synchronization:** Invoking the `__syncthreads()` barrier is essential to guarantee memory consistency: no thread is allowed to start the computation until the entire tile (central data + halo) has been fully populated.

4. **In-Cache Computation:** The LBP algorithm is executed by reading exclusively from Shared Memory, which provides latencies comparable to L1 cache.

*Implementation Note:* While Shared Memory is typically faster than Global Memory, the explicit management introduces overhead instructions for address calculation and synchronization. For arithmetic-light kernels like LBP, this overhead must be weighed against the benefits of caching.

```
__global__ void lbp_kernel_shared(const unsigned char* in,
                                  unsigned char* out,
                                  int w, int h) {
    // Shared Memory with padding for the Halo (+2)
    // Allows neighbor reads without leaving the cache
    __shared__ unsigned char s_img[MAX_BD + 2][MAX_BD + 2];

    int tx = threadIdx.x;
    int ty = threadIdx.y;
    // ... global coordinate computation ...

    // 1. Central pixel loading (offset +1)
    // Each thread loads its pixel into shared memory
    // placing it inside the padded frame
    s_img[ty + 1][tx + 1] = in[y_clamped * w + x_clamped];

    // 2. Halo loading (cooperative)
    // Boundary threads load the required neighboring pixels
    if (ty == 0) { // Example: top border
       s_img[0][tx + 1] = in[y_top_halo * w + x_clamped];
    }
    // ... (analogous logic for bottom/left/right borders and corners) ...

    // 3. Synchronization (crucial)
    // Ensures the entire Halo is loaded before computation
    __syncthreads();

    // 4. LBP computation using ONLY Shared Memory
    // Much faster access compared to Global Memory
    if (valid_pixel) {
        unsigned char center = s_img[ty + 1][tx + 1];
        // ... bitwise operations on s_img ...
        out[idx] = code;
    }
}
```

Listing 3. Synthesis of Shared Memory Kernel (Tiling) strategy

### 3.4.2 Optimization via Read-Only Cache (LDG)

To exploit the spatial locality inherent in the LBP stencil pattern while avoiding the setup complexity of the Texture Object API, the implementation utilizes the **Read-Only Data Cache** path via the `__ldg()` intrinsic.

Although the experimental hardware (NVIDIA GTX 1050, Pascal architecture) features a unified L1/Texture cache physical design, utilizing the read-only path offers distinct micro-architectural advantages over standard global loads:

1. **Read-Only Cache Path:** The `__ldg()` intrinsic routes loads through the specific read-only cache path. This signals to the hardware that the data remains constant during kernel execution. This segregation is particularly beneficial when concurrent operations (like memory copies in other streams) are competing for the Load/Store units.

2. **Simplified Instruction Stream:** Unlike Texture Objects, which require host-side resource binding and specialized state management, `__ldg()` compiles to a direct memory load instruction (`LDG.E`). This eliminates the API initialization overhead while still granting access to the high-bandwidth texture cache pipeline.

3. **Handling Unaligned Accesses:** The Read-Only cache pipeline is optimized to handle unaligned memory access patterns more efficiently than the standard Load/-Store unit. In stencil operations like LBP, where neighbor pixels frequently straddle cache line boundaries, this path minimizes the transaction replay penalties often associated with unaligned reads.

```
__global__ void lbp_kernel_fast(const unsigned char* __restrict__ in,
                                unsigned char* __restrict__ out,
                                int w, int h) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    // Boundary check (Software Managed)
    if (x < 1 || y < 1 || x >= w - 1 || y >= h - 1) return;

    int idx = y * w + x;

    // __ldg() leverages the Read-Only path of the L1/Texture Cache
    unsigned char center = __ldg(&in[idx]);
    unsigned char code = 0;

    code |= (__ldg(&in[(y-1)*w + (x-1)]) >= center) << 7;
    code |= (__ldg(&in[(y-1)*w + (x)  ]) >= center) << 6;
    code |= (__ldg(&in[(y-1)*w + (x+1)]) >= center) << 5;
    code |= (__ldg(&in[(y)  *w + (x+1)]) >= center) << 4;
    code |= (__ldg(&in[(y+1)*w + (x+1)]) >= center) << 3;
    code |= (__ldg(&in[(y+1)*w + (x)  ]) >= center) << 2;
    code |= (__ldg(&in[(y+1)*w + (x-1)]) >= center) << 1;
    code |= (__ldg(&in[(y)  *w + (x-1)]) >= center) << 0;

    out[idx] = code;
}
```

Listing 4. Usage of Read-Only Cache (__ldg)

### 3.5. Pipeline Optimization (CUDA Streams)

In large-scale processing scenarios (e.g., real-time video processing), the main bottleneck shifts from the raw computational power of the GPU to the bandwidth of the PCIe bus. To mitigate this latency, we implemented an asynchronous **Batch Processing** pipeline using **CUDA Streams** (Listing 5).

Instead of processing a single image sequentially, the application manages a queue of $N$ independent images. By assigning each image to a distinct CUDA Stream, we achieve execution overlap: while the **Compute Units** calculate the LBP for image $i$, the **Copy Engines** simultaneously transfer image $i+1$ (Host-to-Device) and download the result of image $i-1$ (Device-to-Host).

For this overlap to be effective, it was necessary to use **Pinned Memory** (Page-Locked Memory) on the host side. This prevents the OS from paging memory to disk, allowing the DMA controller to access physical RAM directly without CPU intervention.

**Configuration and Metrics:** To evaluate the scalability of the pipeline, we tested batch sizes ranging from 1 to 100 images. In this configuration, each stream processes a **complete independent frame**. The **Effective Application**

```
// Launch loop over a Batch of N images
for (int i = 0; i < numImages; i++) {

    // 1. Host -> Device (Async)
    // Transfer the full i-th image to the GPU
    cudaMemcpyAsync(d_in[i], h_pinned_in[i], imgSize ,
                    cudaMemcpyHostToDevice, streams[i]);

    // 2. Kernel Launch (Async)
    // Enqueue the task for image 'i' into stream 'i'
    lbp_kernel_fast<<<gridSize, blockSize, 0, streams[i]>>>(
                    d_in[i], d_out[i], width, height );

    // 3. Device -> Host (Async)
    // The copy starts only after kernel 'i' completes
    cudaMemcpyAsync(h_pinned_out[i], d_out[i], imgSize,
                    cudaMemcpyDeviceToHost, streams[i]);
}

// Final synchronization (wait for the whole batch)
cudaDeviceSynchronize();
```

Listing 5. Asynchronous Batch Pipeline with CUDA Streams

**Bandwidth** reported in the results represents the total bidirectional traffic over the batch execution time ($T_{total}$):

$$\text{Bandwidth} = \frac{2 \times (\text{NumImages} \times \text{ImageSize})}{T_{total}}$$

The factor of 2 accounts for both the upload (H2D) and download (D2H) phases, which occur physically over the PCIe lanes.

## 4. Experimental Setup

To evaluate the performance of the proposed implementations, an experimental analysis was conducted on a heterogeneous CPU–GPU platform.

### 4.1. Hardware and Software Configuration

The tests were carried out on the following hardware configuration:

- **CPU:** Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz.
- **GPU:** NVIDIA GeForce GTX 1050 with 4 GB of VRAM.
- **System RAM:** 16 GB.

From a software perspective, the development environment included:

- **Operating System:** Ubuntu 22.04.5 LTS.

- **Compiler:** NVCC version 13.0 with Driver Version: 580.95.05.
- **Libraries:** OpenMP for host-side parallelization.

### 4.2. Dataset and Memory Layout

All performance tests were conducted using a high-resolution input image with the following characteristics:

- **Dimensions:** $4928 \times 2772$ pixels ($\approx 13.6$ Megapixels). This resolution is significantly higher than standard 4K ($3840 \times 2160$), providing a workload capable of fully saturating the GPU resources.
- **Format:** The source image is a **24-bit RGB JPEG**. However, since standard LBP is an intensity-based operator, the image is converted to **8-bit Grayscale** during the host loading phase. Consequently, the GPU processes a single channel luminance map.
- **Memory Payload:** The data is stored in Global Memory as a linear contiguous array (Row-Major order), with a total size of approx. 13.6 MB ($4928 \times 2772 \times 1$ byte).

### 4.3. Benchmarking Methodology

To ensure a rigorous evaluation and statistical stability, the following methodology was applied:

1. **Metrics Definition:**
   - **Kernel Execution Time ($T_{kernel}$):** Measures strictly the GPU computation time using CUDA Events. This metric is used to calculate the *Computational Speedup* of the Naive, Shared, and **Read-Only Cache (Texture)** implementations, isolating algorithmic efficiency from data transfer overheads.
   - **End-to-End Application Time ($T_{total}$):** Measures the total time including Host-to-Device transfer, kernel execution, and Device-to-Host transfer. This metric is used exclusively for the *CUDA Streams* analysis to evaluate the real-world system throughput (GB/s).
2. **Statistical Stability:** Each test configuration was repeated **5 times**, and the arithmetic mean of the execution times was reported. The standard deviation was monitored to ensuring no significant anomalies occurred due to OS jitter.
3. **Warm-up:** A GPU *warm-up* phase (dummy kernel launch) was performed prior to measurements to exclude CUDA context initialization overhead (which can take hundreds of milliseconds) from the results.
4. **Workload Simulation:** Throughput tests (Streams) were conducted on batches of images (up to 100 frames) to simulate a realistic video-processing pipeline where latency hiding becomes critical.

## 5. Results Analysis

### 5.1. Correctness Verification

Before benchmarking the performance, a rigorous validation phase was conducted to ensure that the parallelization strategies did not alter the algorithmic logic.

The GPU implementations (Naive, Shared Memory, and Read-Only Cache) were validated against a sequential CPU "Golden Reference" implementation. Since the LBP operator relies on a $3 \times 3$ neighborhood, the boundary pixels (1-pixel frame) require specific handling.

In our optimized CUDA kernels, we adopted the following strategy:

1. **Initialization:** The output buffer is initialized to zero using `cudaMemset` before kernel execution.
2. **Execution:** The kernels utilize an early-exit guard (`if (x < 1 || y < 1 ...)`) to skip boundary pixels, minimizing branching overhead within the warp.
3. **Validation:** The correctness check is performed exclusively on the **Region of Interest (ROI)**, defined as the image domain excluding the 1-pixel border ($x \in [1, W-2], y \in [1, H-2]$).

**Outcome:** The automated verification suite confirmed a **bit-exact match** (0 mismatches) between the CPU reference and the GPU outputs for all implemented kernels.

### 5.2. Performance Comparison: CPU vs GPU

Table 1 and Figure 3 highlight the drastic reduction in execution times. The sequential CPU takes approximately **450.28 ms**. Using OpenMP (8 threads) reduces this to **102.74 ms** ( 4.4x speedup). The GPU (using the best **Naive** configuration) completes the computation in only **1.58 ms**, achieving a **Computational Speedup of 284x** compared to the sequential CPU. This confirms that the massive parallelism of the GPU is perfectly suited for the pixel-independent nature of LBP.

| Implementation | Configuration | Time (ms) | Speedup |
|---|---|---|---|
| CPU Seq | 1 Thread | 450.28 ms | 1.00x |
| CPU OpenMP | 8 Threads | 102.74 ms | 4.38x |
| **GPU Naive** | **32x8** | **1.58 ms** | **284.11x** |

Table 1. Execution time and computational speedup (Baseline: CPU Seq).

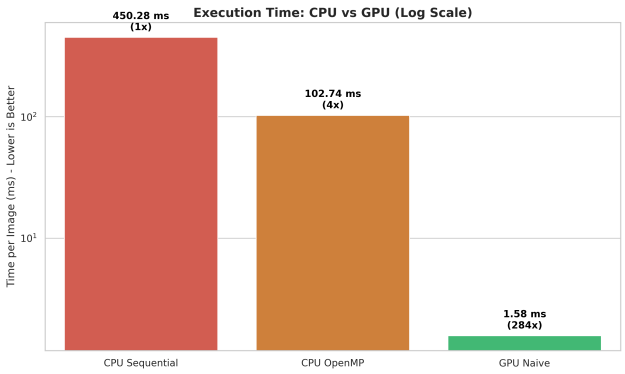


Figure 3. Computation time comparison on a logarithmic scale.

### 5.3. Memory Strategies Analysis

Contrary to initial expectations, the **Naive** implementation (Global Memory) proved to be the most efficient strategy, slightly outperforming the explicit Texture/Read-Only Cache optimization.

Table 2 summarizes the performance of the best configuration for each strategy.

- **Naive (Winner):** The compiler and the Pascal hardware architecture efficiently handle standard global loads. Since the access pattern is perfectly coalesced and spatially local, the L2 and L1 caches are automatically utilized without the need for the `__ldg` intrinsic.
- **Texture (LDG):** While extremely fast (1.59 ms), it introduces a negligible overhead compared to the Naive version. However, it is worth noting that the Texture cache is much more robust against uncoalesced access patterns.
- **Shared Memory:** This strategy is significantly slower (2.93 ms). The overhead of "Tiling" (cooperative loading, boundary handling, and `__syncthreads`) is not justified for an algorithm with such low arithmetic intensity. The GPU spends more time managing the cache manually than performing the actual LBP calculation.

Figure 4 visualizes the **Effective Memory Bandwidth** achieved by the kernels. The Naive and Texture implementations saturate the memory throughput at approximately **17.2 GB/s**, while the Shared Memory implementation drops to **9.3 GB/s** due to the stalled execution cycles during synchronization.

### 5.4. Block Tuning (Naive Strategy)

Focusing on the winning strategy (Naive), we analyzed the impact of block dimensions on performance (Table 3 and Figure 5).

| Strategy | Best Config | Time (ms) | BW (GB/s) |
|---|---|---|---|
| **Naive** | **32 × 8** | **1.585 ms** | **17.24** |
| Texture | 32 × 4 | 1.595 ms | 17.13 |
| Shared | 32 × 8 | 2.930 ms | 9.32 |

Table 2. Comparison of best configurations for each memory strategy.

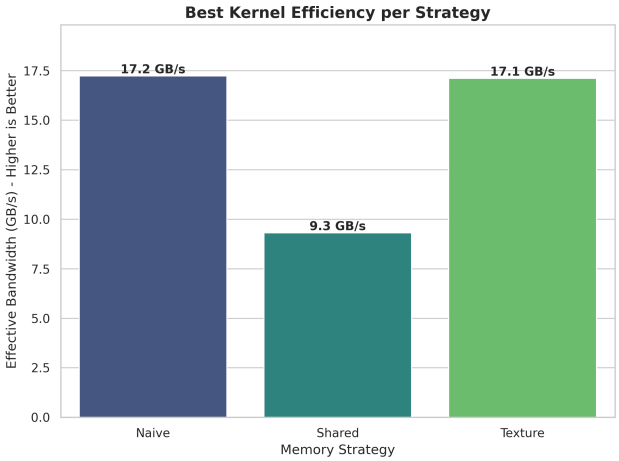


Figure 4. Effective Bandwidth comparison. Naive and Texture strategies nearly maximize the available memory throughput for this access pattern.

- **Optimal Configuration (32×8):** This configuration yielded the best result (**1.585 ms**). A block width of 32 ensures that one Warp handles exactly one row of the block, guaranteeing 100% coalesced memory accesses. The height of 8 provides a total of 256 threads per block, which is a sweet spot for GPU occupancy.
- **Narrow Blocks (32×4):** Performance is extremely close to the optimal case (1.587 ms), confirming that warp-alignment (width 32) is the most critical factor.
- **Tall Blocks (4×32) - The Coalescing Penalty:** This configuration highlights the severity of uncoalesced memory access. With a width of 4, a single Warp is split across 8 image rows. In the Naive implementation, this results in an execution time of **7.90 ms**, which is **5x slower** than the optimal case. The strided access pattern forces the memory controller to perform multiple transactions per request, wasting significant bandwidth.

### 5.5. Scalability and Throughput (CUDA Streams)

Finally, we evaluate the End-to-End system throughput. Figure 6 compares the scalability of two distinct pipeline implementations using CUDA Streams: the **Naive Pipeline** and the **Optimized (LDG) Pipeline**.

| Config ($W \times H$) | Time (ms) | BW (GB/s) |
|---|---|---|
| **32 × 8** | **1.585** | **17.24** |
| 32 × 4 | 1.587 | 17.22 |
| 32 × 16 | 1.676 | 16.30 |
| 16 × 16 | 2.266 | 12.05 |
| 4 × 32 | 7.908 | 3.45 |

Table 3. Impact of Block geometry on Naive Kernel performance. The $4 \times 32$ configuration demonstrates the severe penalty of uncoalesced memory access.

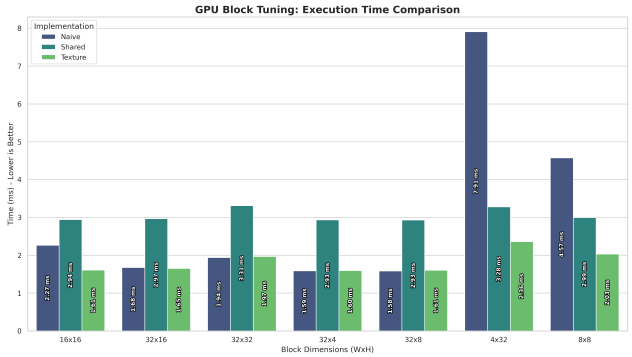


Figure 5. Impact of block geometry on execution time (Naive Kernel).

The results highlight a significant divergence in scalability:

- **Naive Pipeline:** Saturates at approximately **7.5 GB/s**. Despite having a very fast kernel, the system throughput is bottlenecked by memory management overhead on the host side.
- **Optimized Pipeline:** Reaches a saturation point of **10.5 GB/s**.

**Analysis of the Discrepancy:**
It is interesting to note that while the Naive *kernel* was computationally faster (1.58 ms), the Naive *pipeline* is significantly slower. This is attributed to the memory allocation strategy. The Optimized pipeline utilizes **Pinned (Page-Locked) Memory**, which allows the GPU's Copy Engine to perform Direct Memory Access (DMA) concurrently with kernel execution. The Naive implementation likely relies on standard Pageable Memory, which forces the CPU to participate in the transfer, preventing full asynchronous overlap and capping the theoretical bandwidth.

## 6. Conclusions

This work presented and analyzed a high-performance parallel implementation of the Local Binary Pattern operator on NVIDIA GPU architecture. Experimental results confirm that the SIMT approach is extremely effective for pixel-wise image processing algorithms.

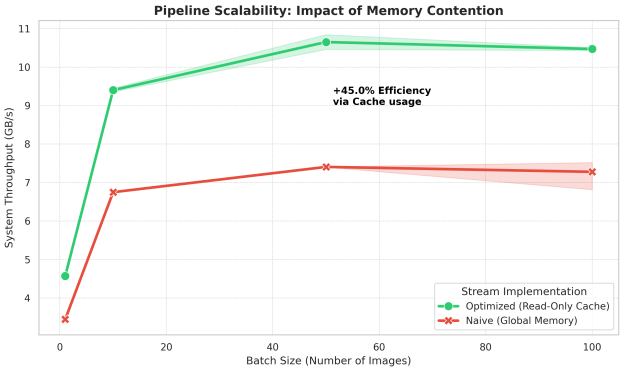


Figure 6. System Throughput comparison. The Optimized pipeline (using Pinned Memory) saturates the PCIe bus at 10.5 GB/s, while the Naive version is limited to 7.5 GB/s.

Isolating the computation phase, the optimized CUDA implementation achieved a **Computational Speedup of 284x** over the sequential CPU version, processing a high-resolution $4928 \times 2772$ frame in just **1.58 ms**. However, in a real-world scenario, this theoretical gain is inherently limited by the PCIe bus transfer latency.

The architectural analysis highlighted key optimizations:

- **Memory Management:** Contrary to classic textbook examples, **standard Global Memory (Naive)** achieved the best kernel performance (1.58 ms), slightly outperforming the Texture path (1.59 ms). This proves that on modern architectures (Pascal), hardware-managed L2/L1 caches handle spatial locality so efficiently that manual management (Shared Memory, 2.93 ms) becomes an unnecessary overhead for small stencils.
- **Coalesced Access:** Block geometry tuning is critical. While wide blocks (e.g., $32 \times 8$) maximize bandwidth, ignoring warp alignment (e.g., $4 \times 32$) results in a **5x performance penalty** (7.90 ms) due to uncoalesced memory transactions.

Finally, to bridge the gap between kernel speed and bus latency, **CUDA Streams** proved essential. It is crucial to note that while the Naive kernel is fast, the system throughput depends on memory allocation: the use of **Pinned Memory** in the optimized pipeline allowed for full asynchronous overlap, saturating the PCIe bandwidth at **10.6 GB/s**, compared to only 7.5 GB/s for the standard pageable memory approach.

## References

[1] Matti Pietikäinen and Guoying Zhao. Local Binary Patterns. *Scholarpedia*, 10(6):9775, 2015. revision #185458.