Politecnico di Torino

GPU Programming course

Efficient GPU implementation of computer vision techniques

Students:
Gobbi Dennis and **Ferraiuolo Pasquale**

Student IDs:
s330438 and **s332262**

Academic Year 2024/2025

Contents

Chapter 1

Computer Vision Techniques

Classical computer vision techniques like Harris corner detection, Canny edge detection and Otsu binarization form the backbone of image analysis, enabling critical tasks such as feature extraction, edge localization, and adaptive thresholding. However, their computational demands—intensive gradient calculations (Harris), multi-stage edge filtering (Canny), and iterative histogram analysis (Otsu)—hinder real-time performance on CPUs. Modern GPUs, thanks to their massively parallel architectures, address these bottlenecks by accelerating pixel/region-level operations. For instance, the convolution, that is a core operation to both Harris and Canny algorithm, or Otsu's histogram generation, benefit greatly from intrinsic GPUs' parallel paradigm. GPU implementations often achieve 10x+ speedups, enabling real-time applications in robotics, autonomous systems, and medical imaging. This paper explores optimized GPU implementations of these algorithms, analyzing architectural synergies, performance gains, and practical trade-offs for resource-efficient vision systems.
The code can be found at https://github.com/GDennis01/GPU_Project_Final

1.1 Pre-processing for Canny Edge and Harris Corner Detection

Before applying Canny edge detection or Harris corner detection, critical pre-processing steps ensure robust feature extraction. First, RGB-to-grayscale conversion simplifies analysis by reducing the image to a single intensity channel, as both algorithms rely on gradient magnitudes. Next, Gaussian blurring mitigates high-frequency noise, which can trigger false edges or spurious corners, by smoothing the image while preserving structural edges. Finally, spatial gradient calculation using the Sobel operator computes horizontal (G_x) and vertical (G_y) intensity derivatives.
For Canny, these gradients determine edge strength and direction for non-maximum suppression, while Harris leverages them to construct the structure tensor, analyzing local intensity variations to identify corner regions.

We will now analyze in depth how these methods work.

RGB to Grayscale. A grayscale pixel value is computed using the weighted mean of the RGB components:

$$\text{Gray} = R \cdot 0.299 + G \cdot 0.587 + B \cdot 0.114 \tag{1.1}$$

The weights (0.299, 0.587, 0.114)[8] are standard values for luminance conversion.

Convolution. Convolution involves sliding a kernel over an image. At each position, the output pixel is the sum of element-wise products between the kernel and the overlapping image region. Formally:

$$(f \star g)[m,n] = \sum_k \sum_l f[k,l] \cdot g[m-k, n-l] \tag{1.2}$$

If the kernel can be represented as a product between a row vector and a column vector, then we can apply the convolution operator twice: once with the row-kernel and once with the column-kernel [5].
The following is an example of a separable kernel:

$$\begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} +1 \\ +2 \\ +1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -1 \end{bmatrix} \tag{1.3}$$

As this operation is crucial to the implementation of our computer vision techniques, we explored different solutions that we'll analyze in the chapter 2.3.2.

Gaussian Blur After converting the image to grayscale, a gaussian blur is applied to smooth out any noises and to enhance the image features. This is done by simply convoluting an image with a gaussian normal kernel, that is, a kernel whose values follow a normal guassian distribution.
The value of the gaussian kernel at position (x, y) is formally[7] defined as:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{1.4}$$

We use $\sigma = 1.5$, balancing noise reduction and feature preservation.

Spatial gradient computation. Once the image has been blurred, we need to compute spatial gradient in both x and y direction. This can be done by simply convoluting the image with specific *sobel* kernels.
These gradients allow us to find edges in an image as they enhance left-right (or top-down) differences.
Gradients are formally[9] computed as follows:

$$G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * A \tag{1.5}$$

$$G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A \tag{1.6}$$

1.2 Pre-processing for Otsu Binarization

Unlike edge or corner detection methods, Otsu's thresholding requires minimal pre-processing, as it operates directly on the statistical distribution of pixel intensities. The primary step involves RGB-to-grayscale conversion to reduce the image to a single-channel intensity representation, enabling histogram-based threshold optimization. While techniques like Gaussian blurring are standard in Canny or Harris workflows to suppress noise, Otsu's method inherently analyzes global intensity variance without relying on spatial gradients (e.g., Sobel operators). This simplicity makes Otsu computationally lightweight, though noise-sensitive scenarios may still benefit from optional smoothing. The algorithm's core focus is maximizing inter-class variance in the grayscale histogram, bypassing gradient calculations or edge-specific filtering.

1.3 Harris Corner Detection

The purpose of the Harris Corner Detector is to find corners in an image. To understand what a corner is, we first have to define what an edge is: an area with abrupt (pixel) intensity changes; a corner instead is an intersection of two edges.
The Harris Corner Detector identifies these corners through a series of computational steps. Before everything, the (gray-scale) image is first convolved with a guassian 2D kernel to reduce the noise. Then, it computes the horizontal (I_x) and vertical (I_y) intensity gradients of the image using derivative filters (e.g., Sobel operators), which highlight edges by capturing directional intensity changes.
Next, it constructs a structure tensor matrix $M = \begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{bmatrix}$ for each pixel, combining the squared gradients (I_x^2,I_y^2) and their product (I_xI_y) over a local window. This matrix reflects the local intensity variation in different directions. To aggregate these values spatially, the gradients are weighted by a Gaussian kernel, emphasizing the central pixel's neighborhood.
The detector then computes the Harris response R for each pixel using the formula $R = \frac{det(M)}{trace(M)}$. This response quantifies the likelihood of a corner: high R values indicate corners (significant intensity changes in multiple directions), negative values correspond to edges (dominant single-direction change), and low absolute values represent flat regions.
To refine the results, non-maximum suppression is applied, retaining only local maxima in R to avoid duplicate detections for the same corner. Finally, a threshold is set on R to select the most prominent corners, filtering out weak responses caused by noise or texture. Typically this threshold is derived from the maximum value in the response

map R scaled down by a factor α.

1.3.1 Shi-Tomasi response

The Shi-Tomasi corner detection method, proposed as an improvement to the Harris detector, modifies the corner response calculation to better distinguish true corners. Instead of using the Harris response formula $R = \frac{det(M)}{trace(M)}$, Shi-Tomasi defines its response R as the minimum of the two eigenvalues (λ_1, λ_2) of the structure tensor M:

$$R = min(\lambda_1, \lambda_2) \tag{1.7}$$

Here λ_1 and λ_2 represent the principal intensity change rates in orthogonal directions. A high value for both eigenvalues indicates a corner, while a high value for only one suggests an edge. By focusing on the smaller eigenvalue, Shi-Tomasi directly measures the "weakest" directional intensity variation, ensuring that only points with significant changes in both directions (true corners) are retained.
Shi-Tomasi is particularly favored in applications like feature tracking (e.g., optical flow) due to its robustness and ability to produce more reliable corners under affine transformations or noise.

1.4 Otsu's method for image binarization

Otsu's method [4] is a widely used technique in image processing for automatic binarization, which converts a grayscale image into a binary image by separating pixels into foreground and background classes.
The method determines an optimal threshold by maximizing the inter-class variance between these two groups,defined as follows:

$$\sigma^2(t) = \omega_0(t) \cdot \omega_1(t) \cdot [\mu_0(t) - \mu_1(t)]^2 \tag{1.8}$$

where

- w_0 and w_1 are the probabilities of class (foreground and background) occurrence for the threshold t
- μ_0 and μ_1 are the mean of the foreground and background class respectively for the threshold t

It assumes the image histogram is bimodal, with peaks corresponding to foreground and background distributions. By exhaustively evaluating all possible thresholds, Otsu's algorithm selects the one that minimizes intra-class variance or equivalently maximizes inter-class separability.
This non-parametric, unsupervised approach is computationally efficient and effective for images with clear contrast between objects and background. However, its performance

may degrade with non-bimodal histograms or overlapping intensity distributions, leading to suboptimal segmentation in such cases. Despite this limitation, Otsu's method remains a foundational tool due to its simplicity and adaptability across diverse applications: as an automatic thresholding algorithm, Otsu's method can also be used to initialize the *high threshold* for the **Canny Edge Detection** algorithm.

1.5 Canny Edge Detection

Canny edge detection is a widely used algorithm in image processing designed to identify edges in an image efficiently while minimizing noise interference. The process involves five key steps.
First, the (gray-scale)image is smoothed using a Gaussian filter to reduce high-frequency noise.
Next, gradient magnitude and direction are computed, typically via the Sobel operator, to highlight regions of rapid intensity change. The magnitude is expressed as the pythagorean theorem of the two gradients I_x and I_y: $|\nabla I| = \sqrt{I_x^2 + I_y^2}$ while direction is computed as the *arctan* of the y-direction gradient over the x-direction one: $\theta = arctan(\frac{I_x}{I_y})$.
Third, a bilinear interpolation non-maximum suppression (NMS) is applied to refine edge localization. Instead of considering only discrete neighboring pixels, the gradient magnitude is interpolated along the gradient direction using bilinear interpolation. This provides a more accurate suppression mechanism by comparing a pixel's gradient magnitude to its interpolated neighbors along the edge direction and suppressing it if it is not a local maximum.
Fourth, double thresholding is applied to distinguishes strong and weak edges: pixels above a high threshold are confirmed as edges, while those between low and high thresholds are considered potential edges. As a threshold, the Otsu's method, explained in the previous chapter, can be used to define the high threshold; the low threshold instead is usually set to an half or a third of the high one.
Finally, edge tracking by hysteresis connects these potential edges by incorporating weak pixels adjacent to strong edges, ensuring coherent edge contours.

1.6 Naive optical flow

We included a simple optical flow example based on Harris corner detection. In this approach, the Harris map is used to identify keypoints, and their displacement between frames is tracked to estimate motion. An example can be visualized trough the Images 1.1, 1.2, 1.3. While naive, this snippet highlights how edge and corner detection can form the building blocks for more advanced techniques in motion analysis and feature tracking.

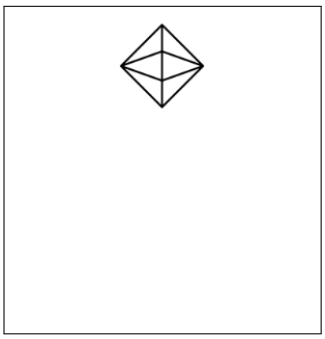

Figure 1.1: First image

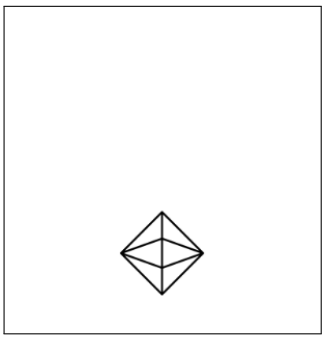

Figure 1.2: Second image

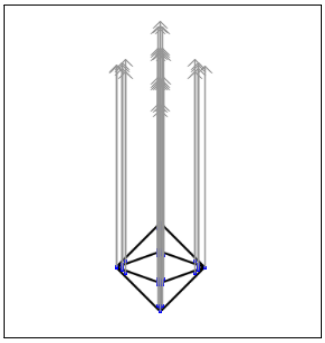

Figure 1.3: Result

Chapter 2

Implementation

2.1 Code structure

The project is structured into several well-defined directories that separate the different aspects of the application. At the top level, directories such as `build`, `include`, and `input` organize build artifacts, header files, and input data respectively.

The heart of the project lies in the `src` directory, which contains the core source code. Here, both CPU and GPU implementations can be found. The entry points for the application are clearly delineated with `main_cpu.cpp` for CPU-based execution and `main.cpp` for the GPU-based execution.

A README file provides the needed informations to compile and run the code (through a Makefile).

2.2 Entry point

The main function serves as the entry point for the application, orchestrating the image or video processing workflow based on the provided command-line arguments. It begins by parsing the arguments to determine the mode of operation (for example, Harris Corner Detection, Canny Edge Detection, Otsu binarization, or Shi-Tomasi) and to extract the input file name. The code also checks if the input is an image or a video (with a ".mp4" extension) and validates additional parameters like threshold values for edge detection when needed.

Once the arguments are parsed and validated, the main function branches into two main execution paths:

- If the input file is an image, the OpenCV library is used to read it and then the CUDA functions are used to apply the below mentioned operations.

- If the input file is a video, the OpenCV library will be used to extract sequentially each frame of the video before applying the processing.

2.3 Core implementations

2.3.1 Rgb to grayscale

The `rgbToGrayKernel` is a very simple kernel that converts an RGB image to a grayscale image. Each thread computes its unique (x, y) position in the image based on its block and thread indices. If the position is within the image boundaries, the corresponding pixel is read from the input array. The kernel then calculates the grayscale value by applying standard luminance weights and writes the result to the output array located in global memory. We are using using `uchar4` for the input vector because it allows the GPU to load and store four related values in a single, efficient memory transaction. The row-major ordering that is being used is particularly important for achieving memory coalescence.

Figure 2.1: On the left a RGB image, on the right the converted image in gray-scale

2.3.2 Convolution

The main challenges we faced when implementing an efficient GPU convolution were:

- Shared memory usage
- Halo pixel loading

Convolution is a key focus of our project so we explored several implementation approaches. Initially, we implemented a naive version without shared memory, then experimented with different shared memory strategy. Since is often required to perform a 3x3 convolution, we initially implemented an efficient configuration that loads the 4 corners: considering that the data is stored contiguously, this will result in coalesced transactions (Image 2.2). In this version, when defining our kernel, we treated it as a

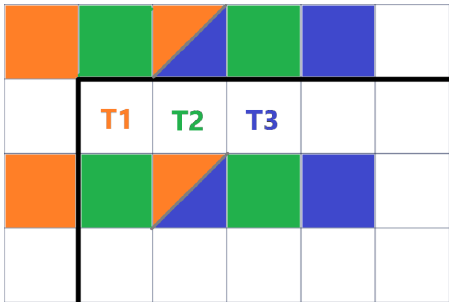

Figure 2.2: Each thread, starting from their index position, loads the 4 edges corresponding to its corners. E.g. Thread 1 with index will load into the shared memory the 4 orange values. Notice that the values outside the black outline are halo points. The example is limited to the first 3 threads for better clarity.

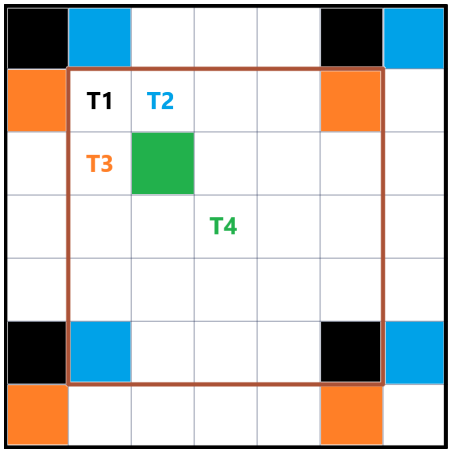

Figure 2.3: Final version of the convolution. Example of shared memory loading of 4 threads. Block size=5x5, kernel size=3x3 (and shared memory 7x7)

`__constant__` in CUDA. It is particularly useful since all the threads need access to the same data, and it avoids redundant memory fetches and allows efficent caching.

The version currently used is an upgraded version of this one: each thread loads from 1 up to 4 separate elements into the shared memory, in a coalesced way with respect to the warps. The strategy can be understood from Image 2.3. Each thread computes its global coordinate and loads corresponding pixel values from global memory into the shared memory array, handling boundary conditions by setting out-of-bound values to zero. After synchronizing the threads to ensure that all data is available, the kernel performs the convolution by iterating over the filter window (with loop unrolling for efficiency), multiplying the shared data by the corresponding filter coefficients and accumulating the results. Finally, the computed sum is written back to the global output array.

The loop of the convolution is this one:

```
#pragma unroll 3 // 3 because we assume the minimum kernel size is 3
for (int i = -filter_radius; i <= filter_radius; ++i)
{
    #pragma unroll 3
    for (int j = -filter_radius; j <= filter_radius; ++j)
    {
        int shX = threadIdx.x + filter_radius + j;
        int shY = threadIdx.y + filter_radius + i;
        sum += data_flat[shY * shared_size + shX] *
               kernel_d[(i + filter_radius) * filter_size + (j + filter_radius)];
    }
}
```

Additionally, we explored a separable convolution implementation inspired by NVIDIA's optimization techniques [1], aiming to leverage computational efficiency (Figure 2.4). However, this approach underperformed for our specific use case due to the small kernel sizes (3×3 and 5×5) involved . The inherent overhead of global memory accesses (required to load input data into shared memory for both the horizontal and vertical convolution) outweighed the theoretical benefits of kernel separation. As a result, the method either introduced latency or showed no measurable improvement over the baseline. While this implementation is no longer active, the code can be found in the repository.

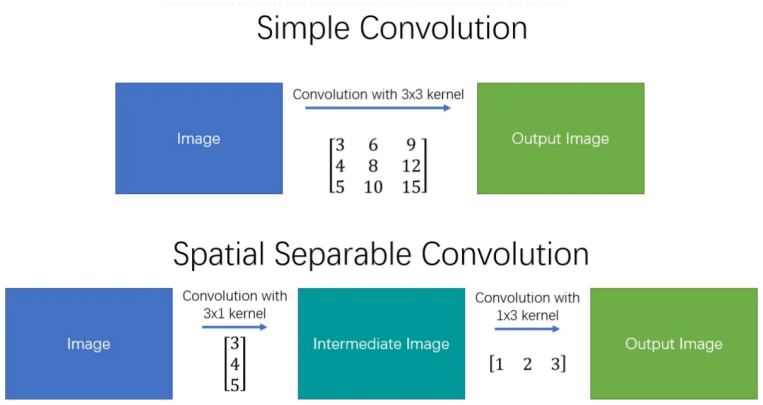

Figure 2.4: Separable convolution. Credits to [6]

2.3.3 Gaussian blur

The gaussian blur, applied to the grayscale image, is performed by applying a convolution based on a pre-computed filter on the host. The filter is obtained by calculating the Gaussian function at each grid point relative to the kernel's center, summing all values, and then normalizing the kernel so that the total sum equals 1.

Figure 2.5: Blurred image obtained from the grayscale image. Kernel size 5x5 with σ 1.5

2.3.4 Spatial gradients

The spatial gradient computation is again performed using the convolutions already defined. It uses some pre-defined kernel that allow to extract the horizontal and vertical features from the image, as showed in 1.5 and 1.6.

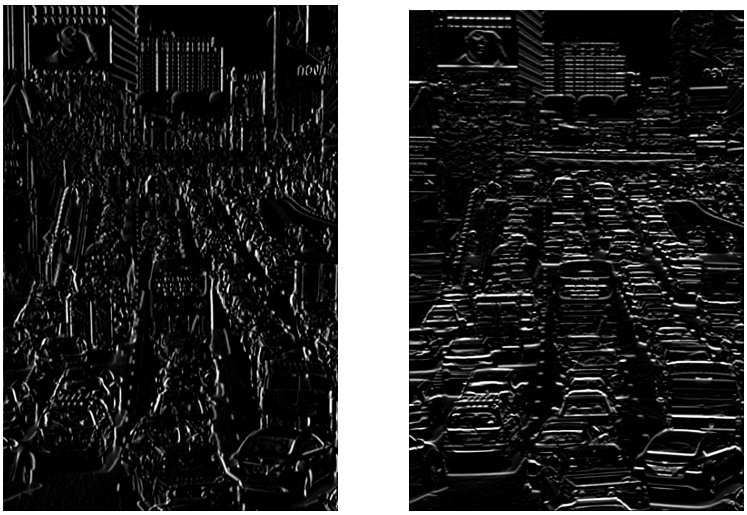

Figure 2.6: Gradients in X(on the left) and Y(on the right) directions obtained from the blurred image

2.3.5 Harris Corner Detection pipeline

The function processes an input image using CUDA streams and memory management to detect corners efficiently. It compute the following operations:

1. Allocates memory on the GPU for gradient matrices (Ix2_d, Iy2_d, IxIy_d, etc.).
2. Computes squared gradients (Ix^2, Iy^2) and the cross-product (IxIy) in parallel using separate CUDA streams.
3. Applies a Gaussian filter to smooth out noise and enhance feature detection.
4. If using Harris detection, computes determinant and trace of the structure tensor. If using shi-tomasi find the minimum eigenvalue.
5. Applies an NMS kernel to suppress weak or redundant corners, keeping only the strongest responses.
6. Uses parallel reduction to find the maximum Harris response value efficiently.
7. Normalizes and colors strong corners based on the computed response values.

Figure 2.7: Traffic image with corners highlighted in red as a result of the Harris pipeline

2.3.6 Shi-tomasi response

The code forms a 2×2 structure tensor using the values A, B, and C. It then computes the trace (the sum of the diagonal elements, A + C) and the determinant (calculated as A * C - B * B). The code select the minimum of lambda1 and lambda2, and stores this value in the `corner_output` array at the index corresponding to the pixel.

```
float trace = A + C;
float determinant = A * C - B * B;

float lambda1 = trace / 2 + sqrtf(trace * trace / 4 - determinant);
float lambda2 = trace / 2 - sqrtf(trace * trace / 4 - determinant);

corners_output[idx] = fminf(lambda1, lambda2);
```

`sqrtf` and `fminf` are used because they are implemented as intrinsic CUDA functions, and additionally, when using compilation flags like `-use_fast_math`, these functions may

be further optimized, sometimes trading a bit of precision for improved speed.

2.3.7 Otsu pipeline

The method computes:

1. The histogram for each pixel in the image
2. The probability of each pixel in the image by dividing its histogram for the total number of pixels
3. The inter-class variance for each threshold (pixel-value) in the range $[0, 255]$
4. The maximum value (as seen below in 2.4.1) of the threshold that maximizes the inter-class variance.
5. The new value of each pixel based on the optimal threshold found.

Since the steps $2 - 4$ are done on the histogram, only **256** threads partake in the computation in these stages.

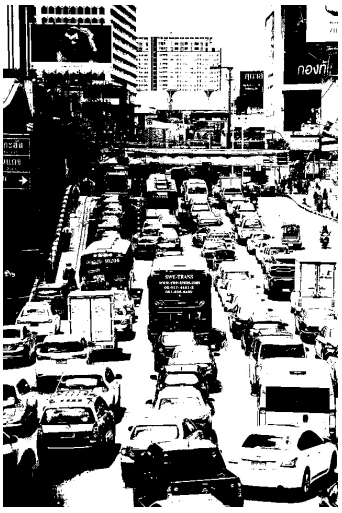

Figure 2.8: Traffic image after being binarized with Otsu thresholding

2.3.8 Canny Edge Detection pipeline

Our Canny Edge Detection method does the following:

1. Combines the spatial gradients to get its magnitude and direction.
2. Based on the gradient's direction of the pixel, applies a Non-Maximum suppression by taking into account the neighbours along the direction.
3. Marks edges (pixels) as either strong,weak or none based on provided thresholds (or an automated generated one through the use of **Otsu's method**).
4. Hysteresis to promote weak edges if connected to other strong edges.

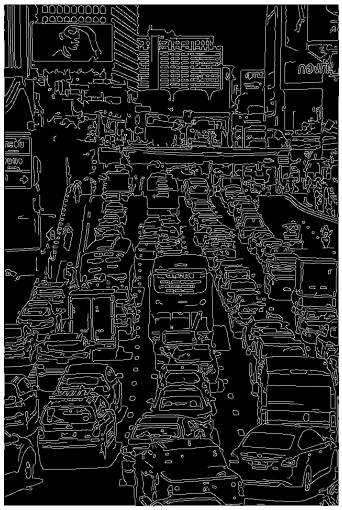

Figure 2.9: Traffic road with detected edges in white

2.4 Additional Implementation

2.4.1 Shuffle Max-Reduction Kernel

shfl CUDA intrinsic

Starting with devices of compute capability 5.0 and later, CUDA provides warp-level shuffle intrinsics like `__shfl_sync()` [2]. These functions enable threads within the same warp (a group of 32 threads executing in lockstep) to directly read registers from neighboring threads without using shared memory. This bypasses slower memory operations, enabling ultra-fast data exchange.
Among the shuffle variants (`__shf_syncl`, `__shfl_up_sync`, `__shfl_down_sync`, `__shfl_xor_sync`), `__shfl_down_sync()` is particularly useful for propagating values **downward** across lanes in a warp. Its signature is [3]:

```
T __shfl_down_sync(unsigned mask, T var, unsigned delta);
```

- `mask`: A 32-bit bitmask specifying which threads in the warp participate. For example, `0xFFFFFFFF` includes all 32 threads.
- `var`: The value from the *current* thread's register to share with others.
- `delta`: The number of lanes to shift `var` downward.

To better understand how these intrinsic functions work, let's take a look at the following example, supposing we have a **warpSize** of **8** instead of the typical **32**:

Original Lane ID	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
var Values	A	B	C	D	E	F	G	H
After **__shfl_down_sync(0xFFFFFFFF, var, 2)**								
Resulting Values	C	D	E	F	G	H	G	H

Since `delta` is **2**, each thread fetch the value of `var` from **2** lanes above, so `thread 0` gets its value from `thread 2`, `thread 1` from `thread 3` and so on. Threads 6 and 7 retain their value since the warp size is 8 and they would have to read from lanes 8 and 9 (which don't exist).

Exploiting shfl to find the maximum value

We implemented a kernel that computes the maximum value inside a matrix, using the reduction paradigm and exploiting the `__shfl_down_sync()` CUDA intrinsics.
The core of the algorithm is the following piece of code that computes the local maximum of a warp of threads:

```
__device__ float warpReduceMax(float val)
{
    for (int offset = warpSize / 2; offset > 0; offset /= 2)
    {
        val = max(val, __shfl_down_sync(0xFFFFFFFF, val, offset));
    }
    return val;
}
```

Thanks to this method, then we compute the **block-level** maximum value by following the classical reduction paradigm. Finally, we compute the actually maximum value of our matrix.
We found out that this version was $\approx 10\%$ faster compared to a shared-memory-based version.

2.4.2 Memory Pinning

Memory pinning (or page-locking) refers to the practice of preventing the operating system from swapping specific regions of host (CPU) memory to disk or relocating them physically. This is critical for optimizing data transfers between the CPU and GPU.
By using functions like `cudaHostRegister()` or `cudaMallocHost()` [10] we were able to speed up CUDA kernel functions by $\approx 30\%$. However the overall execution time didn't benefit greatly due to the additional overhead introduced.

Chapter 3

Comparisons and final results

In this chapter, we evaluate the performance of CUDA kernels under varying parameters and optimization strategies. The results are contextualized using industry-standard metrics such as execution time, throughput (FLOPS) and memory bandwidth utilization.

3.1 Experimental Setup

Benchmarks were performed on two setups: one using an NVIDIA GTX 1650 Ti (Turing Architecture) paired with an Intel i5-10300H, and the other using an NVIDIA GTX 4070 Super paired with an AMD Ryzen 5 7600x. Unless otherwise specified, all benchmark results are based on the GTX 1650 Ti configuration.

3.2 Experiments

3.2.1 Thread Block Configuration Analysis

Optimizing the block size in CUDA applications is critical to achieve peak performance on the GPU. In our experiments, very small block sizes (e.g., 2 threads per block) resulted in extremely poor performance due to low occupancy and the inability to effectively overlap computation with memory operations. As we increased the block size to 8 or 16

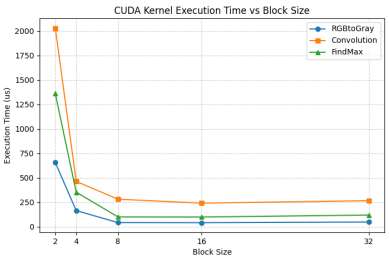

Figure 3.1: Execution time of three different kernels with varying block size.

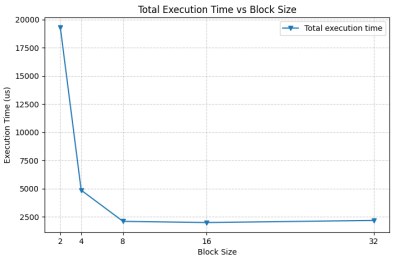

Figure 3.2: Total *kernel* execution time of the Harris Corner Detector algorithm with varying block size.

threads per block (as we can see in Figures 3.1 and 3.2), the GPU could schedule more threads concurrently, leading to better utilization of the available cores and a significant reduction in execution time.
However, further increasing the block size to 32 threads per block did not yield additional improvements; in fact, it sometimes introduced performance penalties likely due to resource contention, such as limited register and shared memory availability per block. These observations underscore the importance of tuning the block size to balance between maximizing parallel execution and minimizing resource bottlenecks, ultimately ensuring that the GPU's computational capabilities are fully leveraged.

3.2.2 Convolution kernel Analysis

We report the results for the convolution function, which is the most used used function. It achieves a good compute/memory balancement, as it can be noted in the Figure 3.3.

The Roofline Analysis in Figure 3.4, illustrates the relationship between Performance (FLOP/s) and Arithmetic Intensity (FLOP/byte). The green dot represents the kernel's performance. It is positioned below the theoretical GPU limit, indicating that the kernel is not fully utilizing the maximum throughput. The kernel appears to be memory-bound, and due to the matrix structure of the data, it would be challenging to further improve memory accesses. This is because the inherent layout and access patterns required for processing the matrix may lead to non-coalesced memory accesses or irregular memory strides.

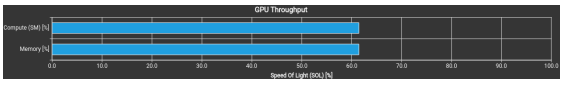

Figure 3.3: GPU troughput

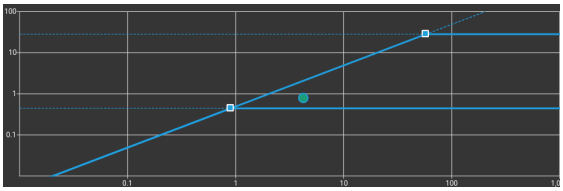

Figure 3.4: Floating points operation roofline. $\frac{\text{Performance[FLOP/S]}}{\text{Arithmetic Intensity[FLOP/byte]}}$

3.2.3 Image size analysis

We evaluated the performance of the Canny Edge Detection algorithm using test images at various resolutions (Figure 3.5). Execution time exhibited a strong positive correlation with image resolution, increasing non-linearly as pixel counts grew.
The algorithm required 10× more processing time at 4K resolution compared to our smallest test image (640×360), demonstrating the significant impact of input dimensions. The complete quantitative results across different input size are detailed in Table 3.1

Figure 3.5: Test image used for the varying-size-benchmark

Image resolution	**Time**
640x360	7ms
960x540	12ms
1280x720	15ms
1920x1080	20ms
2560x1440	37ms
3840x2160	73ms

Table 3.1: Execution time of the Canny algorithm on an image(Fig. 3.5) with varying size

3.2.4 GPU vs CPU performance

The table highlights a significant performance gap between CPUs and GPUs in computer vision tasks. GPUs demonstrate orders-of-magnitude faster (10-20x) execution times compared to CPUs. This demonstrates how essential are GPUs for computer vision tasks, i.e: running Canny Edge detection algorithm on CPU on a video could yield a

maximum of 10 FPS while on a GPU could achieve even a framerate of 120 or higher. Notable is how Otsu binarization performance doesn't differ that much on CPU: this is because it doesn't involve complex operations on the input image, while the other algorithms do.

Figure 3.6: Test image used for CPU vs GPU benchmarking. Size: 960 x 540

	GTX 1650 Ti	**GTX 4070 Super**	**CPU i5-10300H**	**CPU 7600X**
Harris	12 ms	4 ms	145 ms	84 ms
Canny	13 ms	4 ms	170 ms	93 ms
Otsu	7 ms	2 ms	20 ms	8 ms

Table 3.2: Execution time of the Canny algorithm on the Figure 3.6 with different GPUs and CPUs

Bibliography

[1] NVIDIA docs. *Separable Convolution*. https://docs.nvidia.com/vpi/algo_sep_convolution.html.

[2] Justin Luitjens. *Faster Parallel Reductions on Kepler*. https://developer.nvidia.com/blog/faster-parallel-reductions-kepler/.

[3] Nvidia. *Cuda C++ Programming Guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/#warp-shuffle-synopsis.

[4] Nobuyuki Otsu. *A Threshold Selection Method from Gray-Level Histograms*. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4310076. 1979.

[5] Victor Podlozhnyuk. *Separable Convolution*. https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_64_website/projects/convolutionSeparable/doc/convolutionSeparable.pdf.

[6] Chi-Feng Wang. *A Basic Introduction to Separable Convolutions*. https://medium.com/towards-data-science/a-basic-introduction-to-separable-convolutions-b99ec3102728.

[7] Wikipedia. *Gaussian Blur*. https://en.wikipedia.org/wiki/Gaussian_blur#Mathematics.

[8] Wikipedia. *Grayscale*. https://en.wikipedia.org/wiki/Grayscale#Luma_coding_in_video_systems.

[9] Wikipedia. *Sobel Operator*. https://en.wikipedia.org/wiki/Sobel_operator#Formulation.

[10] XiangRongLin. *grayscale-conversion*. https://github.com/XiangRongLin/grayscale-conversion.