Real-Time Vehicle Detection Using GPU Acceleration

Presented By: Md. Shihab Uddin
Instructor: Dr. Michael Wise

Project Overview

- Object detection using background subtraction and adaptive threshold technique.
- Implementation of MOG2 and Gaussian-based adaptive thresholding to differentiate the foreground and background
- Performance comparison of host and device on these above algorithms
- Write down the CUDA kernel and implement it using NVIDIA Libraries.

Background Subtraction (MOG2)

- **Definition**:
  - MOG2 (Mixture of Gaussians 2) is a **background subtraction algorithm** that is used to detect moving objects in video sequences.
  - It models the background using an **adaptive mixture of Gaussian distributions** for each pixel.
- **Key Features:**
  - Dynamically adjusts the number of Gaussian distributions per pixel.
  - Robust to changes in lighting and scene dynamics.
  - Can detect shadows.

Hand-Calculated Example

Scenario:

- Pixel intensity values over 5 frames: I_1 = 100, I_2 = 105, I_3 = 110, I_4= 200, and I_5=100.
- K = 3 Gaussian

Step-by-step calculation

1. Frame 1:
   a. I_1 =100 match to Gaussains
   b. Updated Gaussian 1 : $\mu_1 = 100,\ \sigma_1^2 = 90.1, w_1 = \ 0.3367$
2. Frame 2:
   a. I_2 =105 match to Gaussains
   b. Updated Gaussian 1 : $\mu_3 = 80,\ \sigma_3^2 = 100, w_3 = 0.3289$
3. Frame 3:
   a. I_3 =110 does not match to Gaussains
   b. Add new Gaussian 1 : $\mu_1 = 90.249,\ \sigma_1^2 = 99.98, w_1 = 0.3421$

Hand-Calculated Example

4. Frame 4:
   a. I_4 =200 does not match to Gaussains
   b. Again add new Gaussian 1 : $\mu_2 = 90, \sigma_2^2 = 100, w_2 = 0.3289$ $(from\ fram\ 2)$
5. Frame 5:
   c. I_5 =100 match to Gaussains

Final Classification: Frames I_1, I_2, I_5 are background and I_3, I_4 are foreground.

Key Insights

- MOG2 dynamically adjusts the number of Gaussians, making it robust to scene changes.
- MOG2 can distinguish between foreground objects and shadows.
- Computationally efficient for real-time applications.
- Sensitive to noise in low-quality videos.
- Requires tuning of parameters (e.g., learning rate, number of Gaussians).

Adaptive Thresholding (Mean-Based)

- Definition:
  - Adaptive thresholding is a technique used to separate foreground and background pixels by computing local thresholds for each pixel based on its neighborhood.
  - Unlike global thresholding, it handles uneven illumination and varying lighting conditions.

$$T(x, y) = \mu(x, y) - C$$

Where,

$$\mu(x, y) = Mean\ Intensity\ of\ local\ neighborhood$$

$$C = Constant\ offset$$

Hand-Calculated Example

Consider a 3×3 neighborhood of a pixel:

$$\begin{bmatrix} 50 & 55 & 60 \\ 65 & 70 & 75 \\ 80 & 85 & 90 \end{bmatrix}$$

Compute threshold for center pixel I(2,2) = 70

Step 1 : Compute Mean intensity

$$\mu(x,y) = \frac{50+55+60+65+70+75+80+85+90}{9} = 70$$

Continue

Step 2: Compute the threshold (C=10)

$$T(x,y) = \mu(x,y) - C = 70 - 60 = 60$$

Step 3: Classify Pixel

$$I(2,2) = 70 > T(x,y) = 60 \Rightarrow Foreground$$

Key Insights

- Handles uneven illumination effectively.
- Robust to local variations in lighting.
- Sensitive to noise in the local neighborhood.
- Requires tuning of the neighborhood size and constant C

Checkpoint 2

CUDA Implementation

- For first-pass CUDA implementation - ***Mean Based Adaptive Thresholding*** .
- Input Image: 2D grayscale image
- Sliding Window: Image is divided into 3x3 small region

CUDA Kernel Explanation

- Parallel Pixel Processing
  - Each GPU thread handles one pixel
  - Early exit for threads outside image bounds (if (x >= width || y >= height) return)
- Local Neighborhood Calculation
  - Defines a ***blockSize x blockSize*** area around each pixel
  - Uses nested loops to sum intensities in the neighborhood ***(sum += input[yj*width+xi])***
  - Handles image edges with boundary checks **(xi >= 0 && xi < width)**

- Adaptive Threshold Formula
  - Computes mean intensity: **mean = sum / count (count = valid pixels in neighborhood)**
  - Adjusts threshold with constant C: ***threshold = mean - C***
  - Outputs 255 (foreground) if pixel > threshold, else 0 (background)

```
#define BLOCK_SIZE 16
#define RADIUS 1 // For a 3x3 neighborhood

// Adaptive threshold kernel
__global__ void adaptiveThresholdKernel(const unsigned char* input, unsigned char* output, int width, int height, int blockSize, int C) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x >= width || y >= height) return;

    int pad = blockSize / 2;
    int sum = 0;
    int count = 0;

    // Loop through the blockSize area to calculate the local neighborhood sum
    for (int i = -pad; i <= pad; ++i) {
        for (int j = -pad; j <= pad; ++j) {
            int xi = x + i;
            int yj = y + j;

            if (xi >= 0 && xi < width && yj >= 0 && yj < height) {
                sum += input[yj * width + xi];
                count++;
            }
        }
    }

    int mean = sum / count;
    int threshold = mean - C;

    output[y * width + x] = (input[y * width + x] > threshold) ? 255 : 0;
}
```

Figure: Mean Based Adaptive Threshold

Memory allocation and data transfer

- *cudaMalloc()* is used to allocate device memory
- *cudaMemcpy()* transfer the image from host to device
- *cudaEventRecord()* record the execution time

```
// Allocate device memory
unsigned char *d_input, *d_output;
cudaMalloc((void**)&d_input, image_width * image_height * sizeof(unsigned char));
cudaMalloc((void**)&d_output, image_width * image_height * sizeof(unsigned char));

// Copy input image to device
cudaMemcpy(d_input, h_input, image_width * image_height * sizeof(unsigned char), cudaMemcpyHostToDevice);

// Define block and grid sizes
dim3 blockSize(BLOCK_SIZE, BLOCK_SIZE);
dim3 gridSize((image_width + blockSize.x - 1) / blockSize.x, (image_height + blockSize.y - 1) / blockSize.y);

// Threshold constant (C) and block size (to adjust the neighborhood area)
int C = 2;
// Create CUDA events for timing
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

// Start timing
cudaEventRecord(start);

// Launch the adaptive threshold kernel
adaptiveThresholdKernel<<<gridSize, blockSize>>>(d_input, d_output, image_width, image_height, BLOCK_SIZE, C);
```

Host-Device Comparison:

Execution time comparison between the host and cuda implementation.

- Use three sample images to test both host and cuda implementation.

Execution Time	**Host Side Implementation**	**CUDA Implementation**
Image 1	1.1099 s	9.35581 ms
Image 2	3.8448 s	10.0962 ms
Image 3	1.3507 s	1.94576 ms

Host side and CUDA Implementation

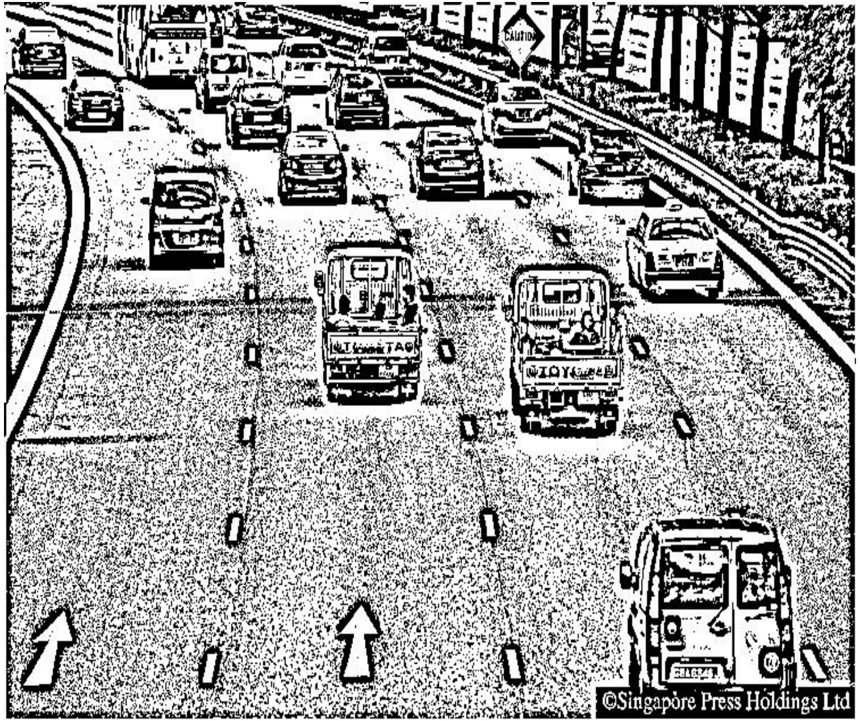

Figure1: Host-side Implementation Image 1

Figure 2: CUDA Implementation Image 1

Host side and CUDA Implementation

Figure 1: Input Image 1

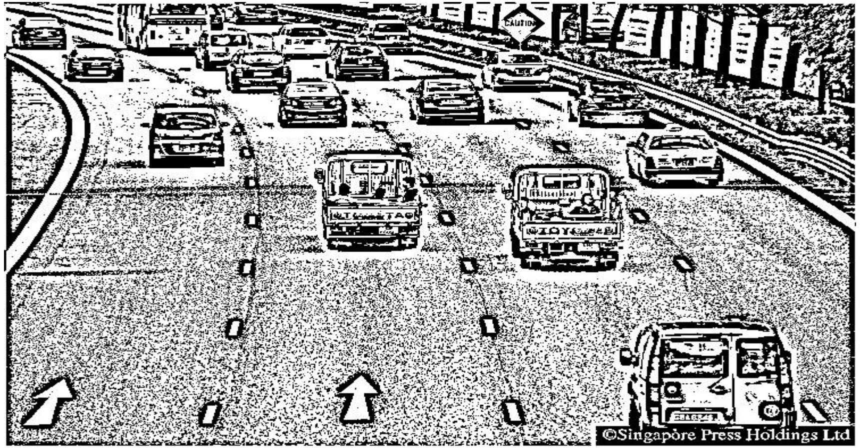

Figure 2: Host-side Implementation Image 1

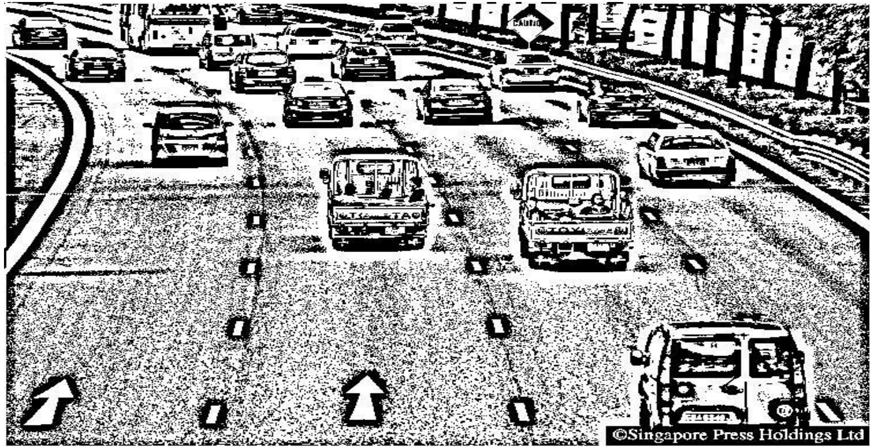

Figure 3: CUDA Implementation Image 1

Host side and CUDA Implementation

Figure4: Input Image 2

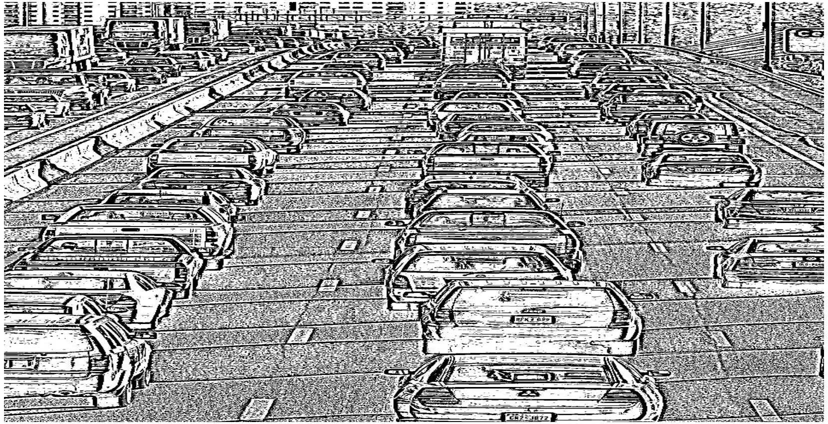

Figure 5: Host-side Implementation Image 2

Figure 6: CUDA Implementation Image 2

Host side and CUDA Implementation

Figure 7: Input Image 3

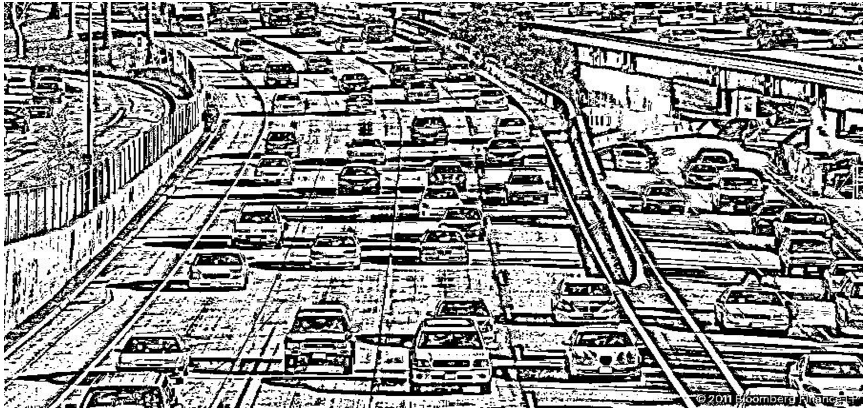

Figure 8: Host-side Implementation Image 3

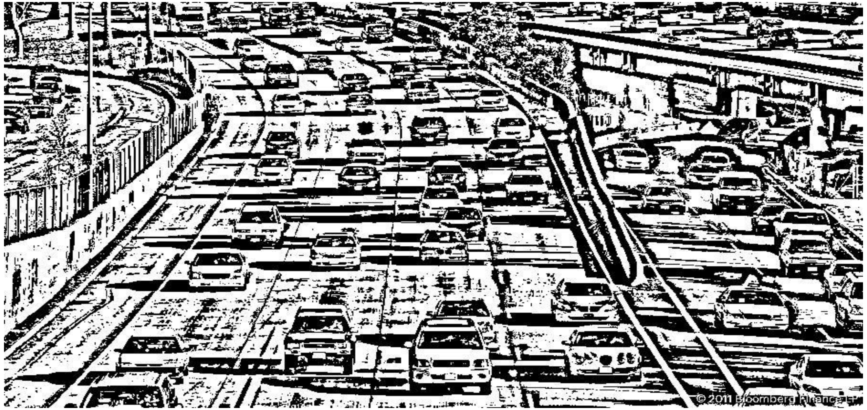

Figure 9: CUDA Implementation Image 3

Discrepancies

- Though both implementations are identical, CUDA implementations produce more darkness in the shadow than the host-side implementations. Reasone could be
    - CUDA kernel uses integer arithmetic operations, while host-side Numpy uses floating-point arithmetic operations. CUDA rounds decimal values (e.g., 127.8 → 127), while Python preserves them, which might result in slightly lower thresholds in CUDA.
    - CUDA perform aggressive thresholding, but python creates mirrored padding (less aggressive)
    - Minor variation in floating-point rounding because of parallel execution in CUDA

Image Loading into CUDA

- Convert JPG into RAW using python
- Load RAW image into cuda implementation
- Save RAW image as output
- Convert RAW image to JPG using python

Github Repository: Link