Parallel Computing for Many Core Processors (CS 677)

Project Report

Adrien Huet

May 12, 2023

Introduction

Object detection is a fundamental task in computer vision that involves locating objects within an image or video and classifying them into predefined categories. It has become a critical area of research due to its numerous practical applications in fields such as autonomous driving, surveillance, robotics, and healthcare, among others. Object detection allows machines to understand and interpret visual data, providing them with the ability to recognize and react to objects in their surroundings. Although a vast diversity of object detectors were designed over the past two decades, most rely on common computer-vision techniques such as filtering and connected components labeling.

Throughout this semester, the CS677 course focused on teaching the basics of GPU programming through the CUDA API. As an end-of-semester project, we were tasked to implement a solution of our choice in accordance with Professor Mordohai specifications. I found Object Detectors to be an interesting topic for GPU-acceleration. Being a matter of video processing, I expected the use of a GPU-accelerated algorithm to be much more efficient approach compared to a CPU-based implementation.

This report details my proceedings and findings for the implementation of a simple Moving Object Detector (referenced as MOD from now on) runnable on a GPU.

1 The Moving Object Detector

The goal of a MOD is to detect a object moving through a background in a given video input. Such a MOD could be described as the following operations:

1. Reserve the first frame $g0$ of the video as a background reference for further frames.
2. Convert both the background $g0$ and the frame to process $g1$ into grayscale.
3. Smooth the two images $g0$ and $g1$ (e.g. with a Gaussian filter).
4. Compute the difference $d = |g0 - g1|$ between the two images.
5. Threshold the image d.
6. Perform morphological closing/opening to remove non-meaningful artifacts.
7. Find and label the connected components.
8. Compute and output the bounding boxes.
9. (*Render new video with boxes drawn onto each frame*)

These operations will constitute our base algorithm for testing and benchmarking.

2 Suitability for GPU acceleration

It could be argued that all of the above steps are executable on a GPU: grayscaling and difference (steps 2 and 4) both rely on pixel-wise operations, hence the use of a GPU to perform those in parallel is obvious. We can expect blurring, morphological operations and labeling (steps 3, 6 and 7) to be the most work-intensive among these steps, as they rely on pixel locality for their computation. Steps 3 and 6 are still relevant in the context of parallelism, as we want to apply the same operation on every pixel's locality. For step 7, parallel algorithms have been implemented by the scientific communitiy, and most see use in modern applications. Those three steps should be the main focus of this project. Steps 1 and 7 will rely on CPU computation. Their impact on performance is however negligible, and we can expect this MOD to run almost completely on the GPU.

3 GPU Implementation

Steps of the algorithms ported to the GPU

1. Reserve the first frame $g0$ of the video as a background reference for further frames. ✓
2. Convert both the background $g0$ and the frame to process $g1$ into grayscale. ✓
3. Smooth the two images $g0$ and $g1$ (e.g. with a Gaussian filter). ✓
4. Compute the difference $d = |g0 - g1|$ between the two images. ✓
5. Threshold the image d. ✓
6. Perform morphological closing/opening to remove non-meaningful artifacts. ✓
7. Find and label the connected components. ✓
8. Compute and output the bounding boxes. **x**
9. (*Render new video with boxes drawn onto each frame*) **x**

Almost all of the steps above have found their CUDA kernel equivalent in the implementation. Figure 1 shows a detailed output of each step executed on the GPU for a given image and background.

Input: Step 1 was only a matter of memory management, specifically two memory copies from host arrays to device arrays (1 for background, 1 for current frame).

Simple Operations: Steps 2, 4 and 5 can be seen as simple pixel-wise/reduction operations, which did not prove to be difficult. They did not see any modification during the optimization process, as all rely on a single read/write per thread, where each thread assumes the computation of one output element. Figure 2 shows the kernels associated to each step:

Convolutions: Steps 3 (Gaussian blur) and 6 (morphological operations) can be considered in the same manner as 2D convolutions, and thus offered lots of optimization possibilities, whilst presenting their own challenges. Gaussian blur uses a Gaussian matrix as the convolution kernel. Morphological dilation and erosion have been chosen to use a circle kernel. Both use 15x15 kernels, and shared the same changes for optimization, detailed in the optimization section. Each kernel is computed on the host, and moved to global memory in the first baseline version. Figure 3 and 4 shows the respective CUDA kernels for V1.0:

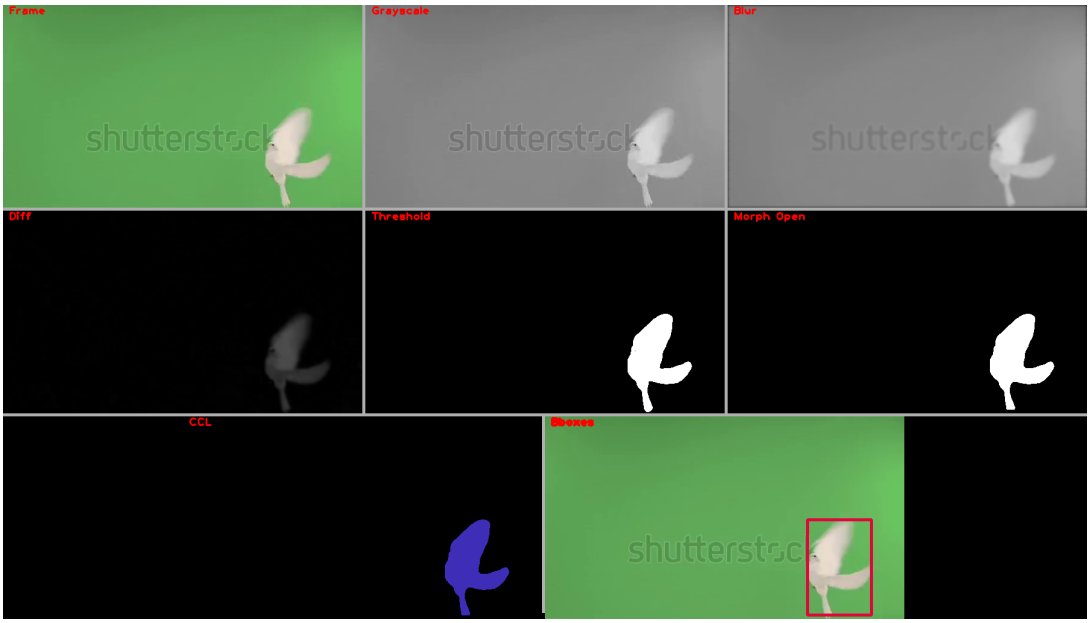


Figure 1: Visualization of the MOD

```cpp
__global__ void grayscaleGPU(const uchar3 *src, uchar *dst, int height,
                             int width)
{
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    if (col >= width || row >= height)
    {
        return;
    }

    uchar3 inputPixel = src[row * width + col];
    uchar grayValue = static_cast<uchar>(
        0.299f * inputPixel.x + 0.587f * inputPixel.y + 0.114f * inputPixel.z);
    dst[row * width + col] = grayValue;
}
```

(a) grayscaleGPU

```cpp
__global__ void thresholdGPU(const uchar *src, uchar *dst, int height,
                             int width, uchar threshold, uchar maxval)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

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

    if (src[y * width + x] < threshold)
        dst[y * width + x] = 0;
    else
        dst[y * width + x] = maxval;
}
```

(b) threshGPU

```cpp
__global__ void diffGPU(const uchar *src1, const uchar *src2, uchar *dst,
                        int height, int width)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

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

    dst[y * width + x] = abs(src1[y * width + x] - src2[y * width + x]);
}
```

(c) diffGPU

Figure 2: Simple operations kernels

```cpp
__global__ void blurGPU(const uchar *src, uchar *dst, int height, int width,
                        float *kernel, size_t ksize)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

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

    int radius = ksize / 2;
    float sum = 0;

    for (int i = -radius; i <= radius; i++)
    {
        for (int j = -radius; j <= radius; j++)
        {
            // Skip when out of bounds
            if (y + i >= 0 && y + i < height && x + j >= 0 && x + j < width)
            {
                sum += kernel[(i + radius) * ksize + (j + radius)]
                    * src[(y + i) * width + (x + j)];
            }
        }
    }

    dst[y * width + x] = static_cast<uchar>(round(sum));
}
```

Figure 3: blugGPU

```cpp
__global__ void dilateGPU(const uchar *src, uchar *dst, int height, int width,
                          uchar *circleKernel, size_t ksize)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

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

    int radius = ksize / 2;
    uchar value = 0;
    for (int i = -radius; i <= radius; i++)
    {
        for (int j = -radius; j <= radius; j++)
        {
            uchar kernelValue =
                circleKernel[(i + radius) * ksize + (j + radius)];
            // We can skip 0 elements
            if (!kernelValue)
                continue;

            // 0 if we are out of bounds
            uchar pixel = 0;
            // In-bounds, so take the value
            if (y + i >= 0 && y + i < height && x + j >= 0 && x + j < width)
                pixel = src[(y + i) * width + (x + j)];

            value |= pixel & kernelValue;
        }
    }
    __syncthreads();
    dst[y * width + x] = value == 1 ? 255 : 0;
}
```

(a) dilateGPU

```cpp
__global__ void erodeGPU(const uchar *src, uchar *dst, int height, int width,
                         uchar *circleKernel, size_t ksize)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x >= width || y >= height)
    {
        return;
    }
    int radius = ksize / 2;
    uchar value = 1;
    for (int i = -radius; i <= radius; i++)
    {
        for (int j = -radius; j <= radius; j++)
        {
            uchar kernelValue =
                circleKernel[(i + radius) * ksize + (j + radius)];
            // We can skip 0 elements
            if (!kernelValue)
                continue;

            // 0 if we are out of bounds
            uchar pixel = 0;
            // In-bounds, so take the value
            if (y + i >= 0 && y + i < height && x + j >= 0 && x + j < width)
                pixel = src[(y + i) * width + (x + j)];

            value &= pixel & kernelValue;
        }
    }
    __syncthreads();
    dst[y * width + x] = value == 1 ? 255 : 0;
}
```

(b) erodeGPU

Figure 4: Morphological CUDA kernels

Connected Components Labelling: Step 7 proved to be the most exotic part of this algorithm to implement on a GPU. Several references in the research literature were consulted to achieve this ([1], [2], [3], [4], [5], [6], [7]). While the CPU baseline used the built-in multi-threaded algorithms from OpenCV, the GPU versions relied on the Union-Find (UF) algorithm, presented in [4] and [5]. Note that in this implementation, 8-way connectivity was used to determine connected components (as opposed to 4-way connectivity), where the neighborhood mask is described with a Rosenfeld mask (see Figure 5).

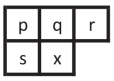
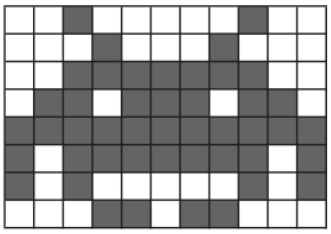


Figure 5: Rosenfeld 2D mask used to find neighboring pixels

The UF algorithm interprets the input image as a forest of trees, with each pixel associated to a node whose label represent the index of the parent node in the image. Its goal is to use an "Union" procedure on trees that are neighbors in the image, effectively merging them at their respective roots, the latter computed using a "Find" procedure (going "up" the three from a given node until reaching the root). Figure 6 shows the two functions on which UF relies, adjusted for GPU computation.



(a) Binary Input

1	2	0	4	5	6	7	8	0	10	11
12	13	14	0	16	17	18	0	20	21	22
23	24	0	0	0	0	0	0	0	32	33
34	0	0	37	0	0	0	41	0	0	44
0	0	0	0	0	0	0	0	0	0	0
0	57	0	0	0	0	0	0	0	65	0
0	68	0	70	71	72	73	74	0	76	0
78	79	80	0	0	83	0	0	86	87	88

(b) Output Initialization

1	1	0	4	4	4	4	4	0	10	10
1	1	14	0	16	17	18	0	20	21	22
23	24	0	0	0	0	0	0	0	32	33
34	0	0	37	0	0	0	41	0	0	44
0	0	0	0	0	0	0	0	0	0	0
0	57	0	0	0	0	0	0	0	65	0
0	68	0	70	71	72	73	74	0	76	0
78	79	80	0	0	83	0	0	86	87	88

(c) Provisional Result

1	1	0	1	1	1	1	1	0	1	1
1	1	1	0	1	1	1	0	1	1	1
1	1	0	0	0	0	0	0	0	1	1
1	0	0	37	0	0	0	41	0	0	1
0	0	0	0	0	0	0	0	0	0	0
0	57	0	0	0	0	0	0	0	57	0
0	57	0	57	57	57	57	57	0	57	0
57	57	57	0	0	57	0	0	57	57	57

(d) Output Labels

(a) GPU steps

```
__device__ int findGPU(int *L, int index)
{
    int label = L[index];
    while (label - 1 != index)
    {
        index = label - 1;
        label = L[index];
    }
    return index;
}
```

(b) Find procedure findGPU

```
__device__ void mergeGPU(int *L, int a, int b)
{
    bool done = false;
    int old;
    while (!done)
    {
        a = findGPU(L, a);
        b = findGPU(L, b);
        done = (a == b);
        if (a == b)
            done = true;
        else
        {
            if (!done && a > b)
                swap(&a, &b);
            old = atomicMin(&L[b], a + 1);
            done = (old == b + 1);
            b = old - 1;
        }
    }
}
```

(c) Union procedure mergeGPU

Figure 6: The Union-Find algorithm on the GPU

In a first step, each foreground node see its label initialized with the raster index of its associated pixel in the image, while background nodes stay at 0 (output from threshold is a binary input (Fig. 6.a.a). This means that each node now represents the root of a tree (Fig. 6.a.b). The kernel responsible for this step, `initCCL()`, is shown on Figure 7 below:

```
__global__ void initCCL(const uchar *src, int *dst, int height, int width)
{
    // UF algo
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    int idx = y * width + x;

    if (src[idx] != 0)
        dst[idx] = idx + 1;
    else
        dst[idx] = 0;
}
```

Figure 7: CCL initialization step

In a second step, each tree is merged based on the neighborhood of each node. In 8-way connectivity, the neighborhood of a pixel can be obtained using a Rosenfeld mask (Figure 5). Each node sees its value updated to point to the new parent, which, in this case, is the minimum value of neighboring nodes. The GPU makes use of atomic operations to avoid concurrent access on the node labels, as several threads might modify the value of a given node. Figure 6.a.c shows what would be a partial state of the image if computation was stopped during this step. The associated kernel, `mergeCCL`, is shown on Figure 8 below:

```
__global__ void mergeCCL(const uchar *src, int *dst, int height, int width)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    int idx = y * width + x;

    if (src[idx] != 0)
    {
        // Not on top edge
        if (y > 0)
        {
            // top left neighbour
            if (x > 0 && src[(y - 1) * width + (x - 1)] != 0
                && (y - 1) * width + (x - 1) < idx)
                mergeGPU(dst, idx, (y - 1) * width + (x - 1));

            // top neighbour
            if (src[(y - 1) * width + x] != 0 && (y - 1) * width + x < idx)
                mergeGPU(dst, idx, (y - 1) * width + x);

            // top right neighbour
            if (x < width - 1 && src[(y - 1) * width + (x + 1)] != 0
                && (y - 1) * width + (x + 1) < idx)
                mergeGPU(dst, idx, (y - 1) * width + (x + 1));
        }

        // Not on left edge, left neighbour
        if (x > 0 && src[y * width + (x - 1)] != 0 && y * width + (x - 1) < idx)
            mergeGPU(dst, idx, y * width + (x - 1));
    }
}
```

Figure 8: CCL Merge step

In a third and final step, each node sees its value updated with the root of the tree it belongs to, instead of just the parent node. This produces the final output: a symbolic image where each pixel value labels

the component the pixel belongs to (Figure 6.a.d). The associated kernel is `compressCCL`, shown on Figure 9 below:

```
__global__ void compressCCL(const uchar *src, int *dst, int height, int width)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    int idx = y * width + x;

    if (src[idx] != 0)
        dst[idx] = findGPU(dst, idx) + 1;
}
```

Figure 9: CCL Compression step

The resulting `int` array can then be moved back to the host and used as input to compute the bounding boxes. It is worth noting that the maximum number of components the CCL algorithm can delimit in a single image is equal to the maximum value of a label. In this case the upper `int` limit. Figure 10 shows an example where several objects are detected in a given frame.

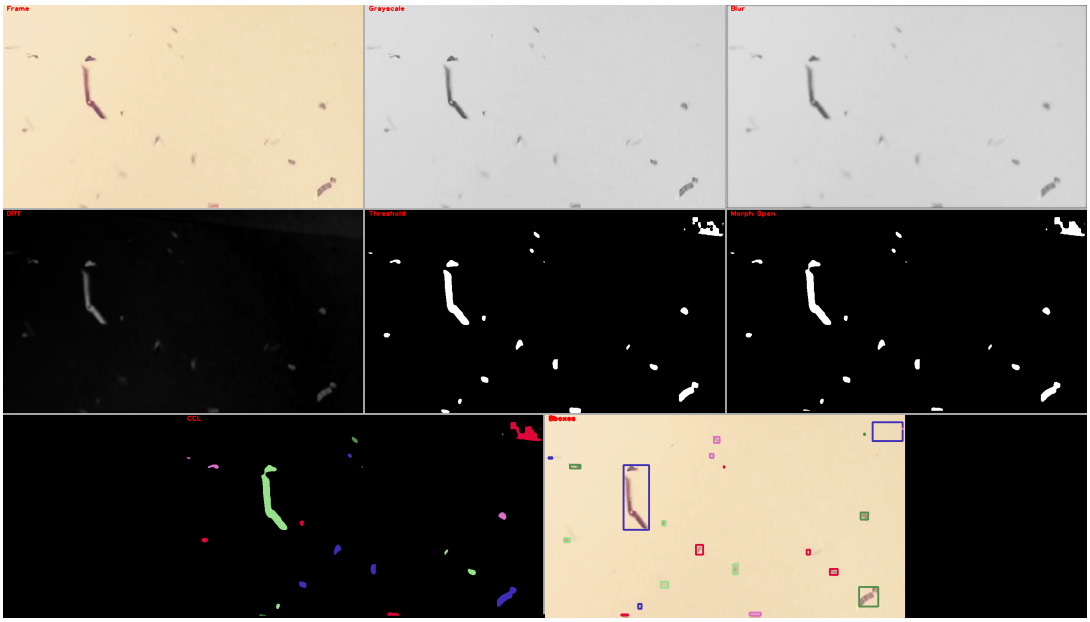


Figure 10: Visualizaition of the MOD on several objects

Bounding Boxes and Output: Step 8 was not implemented on the GPU, and was achieved with a simple fill function that would take the symbollic image output from the previous step and output a OpenCV rectangle for each component. This was mostly due to the fact that bounding boxesare strongly data-dependent, and comuting them on the device would require heavy memory management, although I did not further research the matter. Step 9 is done with the help of the OpenCV API, which also proved useful to manage input/output operations. It is also possible to simply output the bounding boxes parameters without any type of rendering for later use as masks by other programs.

Optimization process

The baseline discussed above constituted a first version of the algorithm (GPU v1.0), and while it showed increased performance over a naive CPU baseline (see Results section), much could be improved.

GPU v1.1

This first improved version focused of reworking the way the application managed memory. The naive GPU v1.0 used to allocate, deallocate and compute all resources for each frame in the video, leaving a non-negligible memory footprint in terms of performance. GPU v1.1 now allocates all device buffers only once, and perform background processing only once. This results in only 2 memory copies for each frame (1 for original input frame, 1 for CCL output), leaving the processed background frame in device memory for the remaining of the process.

GPU v1.2

This second improved version consisted in implementing shared memory tiling for eligible steps in the algorithm, and using constant memory to improve performance. Grayscaling, difference and thresholding do not benefit from those improvement, as they consist in pixel-wise operations. No change was made upon them. Blurring and morphological operations however benefit greatly from the use of faster memory. The tiling strategy seen in class had to be adapted to larger filter sizes, as both of those steps use rather large (15x15) kernels. The load follows a iterative process with increasing stride, where every thread will load several element from the input tile. See Figure 11 below:

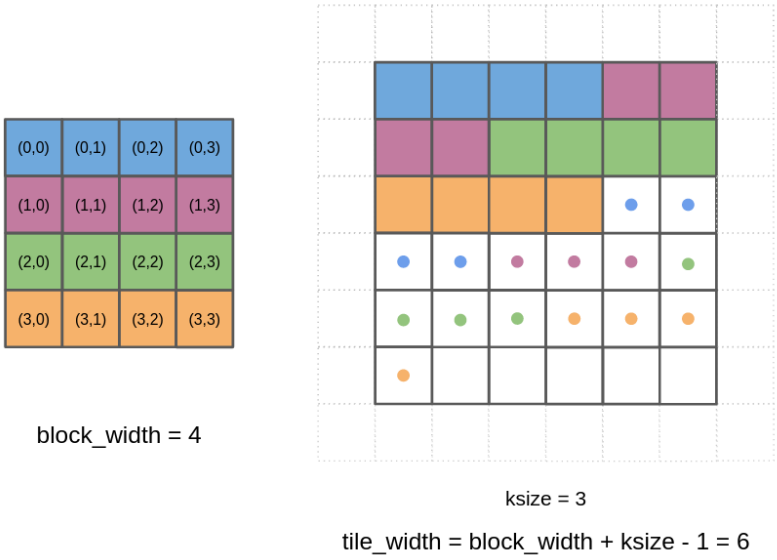


Figure 11: The shared memory load strategy. On the left a block with indexed threads, on the right, the shared memory tile. Full colored blocks represent the initial batch load, colored circles represent the second batch load. This batch loading is repeated until all necessary elements are present in shared memory.

For flexibility, shared memory is dynamically allocated at kernel launch (hence the `extern` keyword). This same memory strategy was applied to morphological operations: only convolution operation changes. Figure 12 shows the `blurTiledConstantGPU` kernel.

GPU v1.3

GPU v1.3 solely consisted on optimizing launch parameters with a grid search. It was found that a block size of 16 was optimal for most inputs (see Results).

```cpp
__global__ void blurTiledConstantGPU2(const uchar *src, uchar *dst, int height,
                                      int width)
{
    extern __shared__ uchar shared_src[]; // Dynamic shared memory allocation
    size_t tile_width = blockDim.x + c_gauss_ksize - 1; // Output tile dim
    size_t block_width = blockDim.x; // Input tile dim

    int o_idx = threadIdx.y * block_width + threadIdx.x; // Tile write index
    int o_row = o_idx / tile_width; // Tile dest row
    int o_col = o_idx % tile_width; // Tile dest col
    int i_row =
        blockIdx.y * block_width + o_row - (c_gauss_ksize / 2); // Source row
    int i_col =
        blockIdx.x * block_width + o_col - (c_gauss_ksize / 2); // Source col
    int i_idx = (i_row * width + i_col); // Source read index

    // First batch loading
    if (i_row >= 0 && i_row < height && i_col >= 0 && i_col < width)
        shared_src[o_row * tile_width + o_col] = src[i_idx];
    else
        shared_src[o_row * tile_width + o_col] = 0;

    // Second -> nth batch loading
    for (int iter = 1;
         iter <= (tile_width * tile_width) / (block_width * block_width);
         iter++)
    {
        // Same operations + iteration offest
        o_idx = threadIdx.y * block_width + threadIdx.x
            + iter * (block_width * block_width);
        o_row = o_idx / tile_width;
        o_col = o_idx % tile_width;
        i_row = blockIdx.y * block_width + o_row - (c_gauss_ksize / 2);
        i_col = blockIdx.x * block_width + o_col - (c_gauss_ksize / 2);
        i_idx = (i_row * width + i_col);
        if (o_row < tile_width && o_col < tile_width)
        {
            if (i_row >= 0 && i_row < height && i_col >= 0 && i_col < width)
                shared_src[o_row * tile_width + o_col] = src[i_idx];
            else
                shared_src[o_row * tile_width + o_col] = 0.0;
        }
    }
    // Loading is done
    __syncthreads();

    float accum = 0;
    int y, x;
    for (y = 0; y < c_gauss_ksize; y++)
        for (x = 0; x < c_gauss_ksize; x++)
            accum +=
                shared_src[(threadIdx.y + y) * tile_width + threadIdx.x + x]
                * c_gaussianKernel[y * c_gauss_ksize + x];
    y = blockIdx.y * block_width + threadIdx.y;
    x = blockIdx.x * block_width + threadIdx.x;
    if (y < height && x < width)
        dst[(y * width + x)] = static_cast<uchar>(round(accum));
}
```

Figure 12: blurTiledConstantGPU kernel, same memory load is used for dilateTiledConstantGPU and erodeTiledConstantGPU, with only the convolution operation unchanged.

Use of resources

Taking into account the latest GPU implementation, we can summarize global/constant memory usage and bandwidth the following way (Table 1):

Scope	Memory allocation	Bandwidth
Launch	• 1x Background image (uchar3) freed • 1x Current image (uchar3) • 1x Processed background buffer (uchar) • 1x Processed frame buffer (uchar) • 1x Morph swap buffer (uchar) • 1x CCL symbolic image (int) Total: 10 * width * height bytes	• 1x Background image HtoD (uchar3) • 1x Blur kernel HtoC (ksize² float) • 1x Morph kernel HtoC (ksize² uchar) Total: 3 * width * height + 225 * ksize² bytes
Frame	None	• 1x Current image HtoD (uchar3) • 1x CCL symbollic image DtoH (int) Total: 7 * width * height bytes/frame

Table 1: Memory usage GPU v1.3

In terms of resources, below are two tables. The first (Table 2) regroups the `nvprof` metrics I considered important to this project. This configuration was run on a random frame in a video of dimensions 596x336 with a `block_width` of 16. The second (Table 3) details register usage for each of the implemented kernels.

Kernel	Occupancy	SM_eff %	Branch_eff %	sh_load_transac/req	sh_store_transac/req	sh_eff %	gld_eff %
grayscaleGPU	0.796097	84.48	100.00	0.000000	0.000000	0.00	20.90
blurTiledConstantGPU	0.722304	98.95	100.00	1.000000	0.539258	24.96	33.58
grayscaleGPU	0.809453	84.67	100.00	0.000000	0.000000	0.00	20.90
blurTiledConstantGPU	0.720450	98.99	100.00	1.000000	0.539258	24.96	33.58
diffGPU	0.786948	81.86	100.00	0.000000	0.000000	0.00	35.90
thresholdGPU	0.805545	85.16	100.00	0.000000	0.000000	0.00	35.90
dilateTiledConstantGPU	0.714877	99.16	100.00	0.644444	0.539258	24.93	33.58
erodeTiledConstantGPU	0.717115	99.12	100.00	0.644444	0.539258	24.93	33.58
initCCL	0.811553	83.86	100.00	0.000000	0.000000	0.00	36.36
mergeCCL	0.614099	63.50	87.95	0.000000	0.000000	0.00	39.90
compressCCL	0.689342	79.00	95.05	0.000000	0.000000	0.00	42.63

Table 2: `nvprof` metrics

Occupancy achieved is on average 70%, and SM efficiency is 85%. We see almost no branch divergence, and shared memory usage improves global memory efficiency in the kernels concerned by these modifications. Register use is however very high for those 3 latter kernels, which could explain their lower occupancy compared the others. For the `mergeCCL` and `compressCCL` kernels, their subpar occupancy could be explained respectively by the use of atomic operations and the non-coalesced global memory accesses.

Kernel	# of registers used
grayscaleGPU	11
blurTiledConstantGPU	37
diffGPU	8
thresholdGPU	7
dilateTiledConstantGPU	37
erodeTiledConstantGPU	37
initCCL	7
mergeCCL	15
compressCCL	8

Table 3: Number of registers used by each kernel. Obtained with the `--ptxas=-v` option during compilation.

4 Results

The different GPU implementations were benchmarked against 2 CPU versions. The first CPU version is a naive, hand-made and single-threaded implementation; the second uses the OpenCV API with all the optimizations it provides (multithreading, tiling, etc.). In order to measure the impact of the different GPU versions, simple kernel-wise benchmarking was performed, with a main focus on processed frame per second (FPS) in a unique video input. This metric was measured in all versions.

All of the following GPU benchmarks were launched with the same input used to profile the kernels (596x336 video input of 124 frames with 1 object). My machine uses the following hardware:

- Processor (quad-core): Intel® Core™ i7-6700HQ CPU @ 2.60GHz
- GPU: NVIDIA Corporation GM107M [GeForce GTX 960M]
- Memory: 8 GiB

See below for the benchmark tables of each implementation.

Input:	• 596x336 • 1 object • 2400ms duration

Step	**Frame avg.**	**Total execution time**	**Execution load (%)**
grayscale	1.03ms	128ms	0.25%
blur	258ms	31.95s	61.5%
• getGaussianMatrix	13.5μs	1679μs	0.003%
diff	265μs	32.8ms	0.1%
threshold	279μs	34.6ms	0.07%
morph	124ms	15.4s	29.6%
• getCircleKernel	1.17μs	145μs	0.0003%
connectedComps	15.1ms	1.88s	3.61%
bboxes	17.1ms	2.12s	4.09%
FPS: 2.392 **Time spent**: 51.9s			

Table 4: Hand-made CPU v1.0 benchmark

Step	Frame avg.	Total execution time	Execution load (%)
grayscale	216μs	26.8ms	9.53%
blur	303μs	37.59ms	13.4%
diff	15.6μs	1.93ms	0.69%
threshold	11.1μs	1.38ms	0.49%
morph	1127.669μs	140ms	49.8%
connectedComps	191μs	23.7ms	8.43%
bboxes	109μs	13.5ms	4.82%
FPS: 450.4 **Increase over CPU v1.0:** $\sim$ 448 FPS **Time spent**: 281ms			

Table 5: OpenCV v1.0 benchmark

From those first two tables (4, 5), we can see that OpenCV uses strong optimization, allowing the object detector to run with real-time performance without the need for a GPU. We will focus on this implementation to compare our GPU implementation performances.

Configuration:	• blockDim: 32x32 • gridDim: 19x11

Step	Frame avg.	Total execution time	Execution load (%)
grayscaleGPU	56μs	6.95ms	0.57%
blurGPU	3.04ms	378ms	30.9%
• getGaussianMatrix	1.89μs	234μs	0.02%
diffGPU	23.2μs	2.87ms	0.24%
thresholdGPU	22.1μs	2.74ms	0.22%
morph	4.24ms	526ms	43%
• dilateGPU	2.11ms	261ms	21.4%
• erodeGPU	2.11ms	262ms	21.4%
• getCircleKernel	1.34μs	166μs	0.014%
connectedComps	518μs	64.3ms	5.26%
• initCCL	29.1μs	3.61ms	0.3%
• mergeCCL	84.5μs	10.5ms	0.86%
• compressCCL	21.1μs	2.62ms	0.21%
bboxes	793μs	98.3ms	8.04%
Mem. Management	1.21ms	150ms	12.3%
FPS: 100.9 **Increase over CPU v1.0:** $\sim$ 99 FPS **Time spent**: 1.24s			

Table 6: GPU v1.0 benchmark

Using a naive GPU implementation (v1.0) already show some great improvements over a naive CPU implementation (Table 6). We can however see that blurring and morphological operations still represent the major part of the computation. Memory allocations, frees and copies are all timed within the category `Mem. Management` of the benchmarks. Being poorly managed in v1.0, these operations take their toll on global performance, and were the subject of the next GPU version.

Configuration:	• blockDim: 32x32 • gridDim: 19x11

Step	**Frame avg.**	**Total execution time**	**Reduction over last (%)**	**Execution load (%)**
grayscaleGPU	34.2μs	4.24ms	39%	0.46%
blurGPU	1.53ms	189ms	50%	20.6%
• getGaussianMatrix	0.03μs	3.74μs	98%	0.0004%
diffGPU	23.4μs	2.91ms	-	0.32%
thresholdGPU	22.6μs	2.81ms	-	0.31%
morph	4.23ms	525ms	-	57%
• dilateGPU	2.11ms	262ms	-	28.4%
• erodeGPU	2.11ms	262ms	-	28.4%
• getCircleKernel	0.009μs	1.15μs	99%	0.00013%
connectedComps	427μs	53ms	-	5.76%
• initCCL	29.3μs	3.63ms	-	0.39%
• mergeCCL	45.6μs	5.65ms	-	0.61%
• compressCCL	17.3μs	2.15ms	-	0.233%
bboxes	628μs	77.8ms	-	8.45%
Mem. Management	490μs	60.8ms	59%	6.6%
FPS: 135.5 **Increase over GPU v1.0:** $\sim$ 35 FPS **Time spent**: 0.921s				

Table 7: GPU v1.1 benchmark

As discussed in the optimization process, this version (v1.1, Table 7) focused on improving the application memory management. It made sure that all resources were allocated, freed and computed (kernel matrices) only once, and that the background frame was not reprocessed for every frame. We can see significant improvements in terms of performance for the pre-processing processing kernels (blur and grayscale) and in overall memory management as well.

Configuration:	• blockDim: 32x32 • gridDim: 19x11

Step	**Frame avg.**	**Total execution time**	**Reduction over last (%)**	**Execution load (%)**
grayscaleGPU	46.1μs	5.71ms	-	0.88%
blurGPU	912μs	113ms	40%	17.4%
• getGaussianMatrix	0.06μs	7.33μs	-	0.001%
diffGPU	28.4μs	3.52ms	-	0.54%
thresholdGPU	25.1μs	3.11ms	-	0.48%
morph	2.28ms	282ms	46%	43.5%
• dilateGPU	1.12ms	139ms	47%	21.4%
• erodeGPU	1.13ms	140ms	14%	21.5%
• getCircleKernel	0.018μs	2.27μs	-	0.0004%
connectedComps	597μs	74ms	-	11.4%
• initCCL	34.7μs	4.31ms	-	0.66%
• mergeCCL	60.8μs	7.53ms	-	1.16%
• compressCCL	19.2μs	2.38ms	-	0.37%
bboxes	457μs	56.7ms	-	8.74%
Mem. Management	711μs	88.2ms	-	13.6%
FPS: 192.8 **Increase over GPU v1.1:** $\sim$ 60 FPS **Time spent**: 0.649s				

Table 8: GPU v1.2 benchmark

This next version (v1.2, Table 8) of the GPU implementation allowed the blurring and morphological kernels to use shared and constant memory. Kernel matrices are copied into constant memory at launch, and strategy for shared memory loading has been discussed earlier. These modifications showed massive improvements for those kernels. However, they still represent most of the execution time ($> 60\%$).

Configuration:	• blockDim: **16x16** • gridDim: 38x21

Step	**Frame avg.**	**Total execution time**	**Reduction over last (%)**	**Execution load (%)**
grayscaleGPU	32.5μs	4.03ms	29%	0.72%
blurGPU	908μs	113ms	-	20.1%
• getGaussianMatrix	0.03μs	4μs	-	0.001%
diffGPU	24.2μs	3ms	15%	0.54%
thresholdGPU	24.5μs	3.04ms	-	0.54%
morph	2.24ms	279ms	-	49.8%
• dilateGPU	1.11ms	138ms	-	24.8%
• erodeGPU	1.12ms	139ms	-	24.8%
• getCircleKernel	0.015μs	1.89μs	-	0.0003%
connectedComps	413μs	51.2ms	30%	9.17%
• initCCL	23.2μs	2.87ms	33%	0.51%
• mergeCCL	46.4μs	5.75ms	23%	1.03%
• compressCCL	20.4μs	2.53ms	-	0.45%
bboxes	363μs	45ms	-	8.05%
Mem. Management	478μs	59.3ms	-	10.6%
FPS: 224 **Increase over GPU v1.2:** $\sim$ 30 FPS **Time spent**: 0.559s				

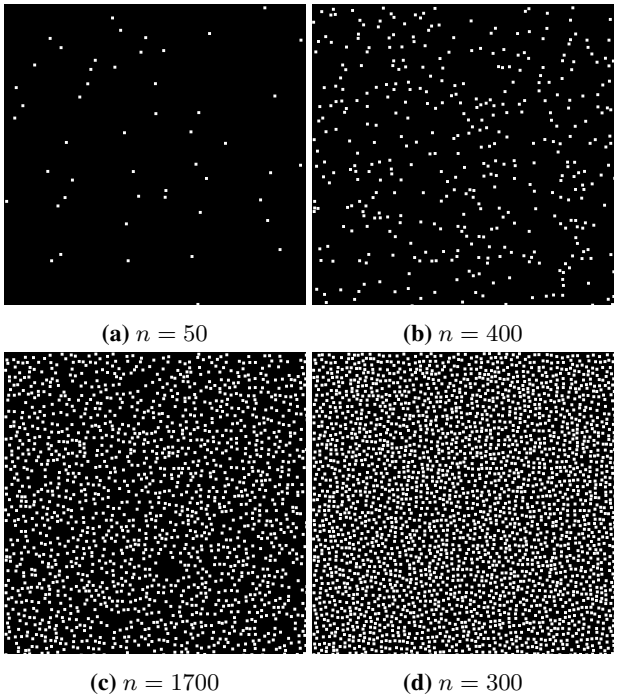
Table 9: GPU v1.3 benchmark

This version(v1.3, Table 9) was only about parameter optimization. After a quick grid search, it has been shown that 16-wide blocks are the most efficient for this computation. Shared memory tile size is computed from block size, so this modification solely had an impact on the grayscale kernel and the CCL algorithm.

Time complexity

Following the project presentation, Professor Mordohai tasked me to study the performance of the CCL algorithm with a variable number of connected components in a image. The CCL algorithm on the GPU is strongly data-dependant, as memory access and atomic operations are performed relative to the geometry of the components, whereas a=the CPU-based implementation shows a complexity increasing with frame size.

For this purpose, a simple random image generator has been created. Figure 13 shows example outputs of size 2048x2048 given different number of componentsto generate.



(a) $n = 50$ **(b)** $n = 400$

(c) $n = 1700$ **(d)** $n = 300$

Figure 13: Some 2048x2048 outputs from the image generator for n components

In order to get precise measurement, both the GPU (v1.3) and the CPU (OpenCV) implementations performed several iteration for each image of N connected components. With 2048x2048 image, the range for N goes from 0 to approximately 3600, after which the image becomes too crowded to fit more components. It is worth noting that this experiment focuses on the number of connected components, whereas some examples in literature use generators with parametric pixel density and granularity.

From a range of 0 to 3600 20x20 components within 2048x2048 images, with a stride of 10 and 10 iterations per size, we obtained Figure 14.

As expected, OpenCV's `connectedComponents()` follows a relatively constant complexity, the image size remaining constant. However, the CCL algorithms performs better when the number of components does not exceed a certain threshold. In this situation, we can expect this threshold to be around 50% pixel density in the input image. However, as discussed above, this experiment uses constant geometry for components, and those results might differ depending on geometrical complexity.

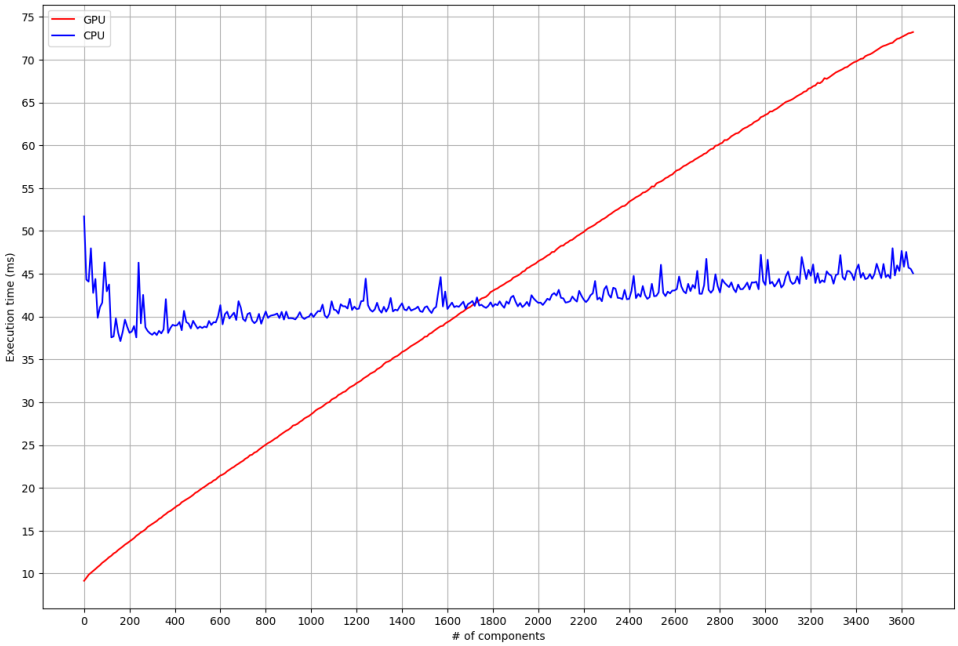


Figure 14: Time complexity (ms) with respect to the number of connected components. In red GPU performance, in blue CPU performance

Conclusion & Insights

Although the last GPU implementation was not able to outperform OpenCV's multithreaded algorithms, they rely on the same mechanics. Both shares similarities in terms of execution load, where convolution operations are showed to be the most expansive. Moreover, further optimizations are possible for the GPU implementation, providing future versions insights:

- Tile prefetching for "shared memory kernels": shared memory usage is relatively low. We could further benefit from it by hiding latency from global memory using a double-buffered tiling strategy as seen during class.
- Loop unrolling: unrolling loops could improve performance on shared memory tiling or convolution operations.
- Different CCL algorithms: the Union-Find algorithm was chosen here for simplicity, but more developed graph algorithms have been designed in the recent years [7].

GPU algorithms are also very hardware-dependent, so further analysis of the different performance metrics shown in Table 2 could lead to improvments for specific devices. Overall, this project proved to be an interesting challenge, and potential for better solutions.

References

[1] Suzuki, Kenji & Horiba, Isao & Sugie, Noboru. (2003). Linear-time Connected-component Labeling Based on Sequential Local Operations. Computer Vision and Image Understanding. 89. 1-23. 10.1016/S1077-3142(02)00030-9.

[2] Wu, Kesheng & Otoo, Ekow & Suzuki, Kenji. (2005). Two Strategies to Speed up Connected Component Labeling Algorithms.

[3] Dillencourt, Michael & Samet, Hanan & Tamminen, Markku. (1992). A General Approach to Connected-Component Labelling for Arbitrary Image Representations.. J. ACM. 39. 253-280. 10.1145/128749.128750.

[4] Oliveira, Victor & Lotufo, Roberto. (2010). A Study on Connected Components Labeling algorithms using GPUs. SIB-GRAPI. 2010.

[5] Allegretti, Stefano & Bolelli, Federico & Grana, Costantino. (2019). Optimized Block-Based Algorithms to Label Connected Components on GPUs. IEEE Transactions on Parallel and Distributed Systems. PP. 1-1. 10.1109/TPDS.2019.2934683.

[6] Allegretti, Stefano & Bolelli, Federico & Cancilla, Michele & Grana, Costantino. (2018). Optimizing GPU-Based Connected Components Labeling Algorithms. 175-180. 10.1109/IPAS.2018.8708900.

[7] Bolelli, Federico & Allegretti, Stefano & Baraldi, Lorenzo & Grana, Costantino. (2019). Spaghetti Labeling: Directed Acyclic Graphs for Block-Based Connected Components Labeling. IEEE Transactions on Image Processing. PP. 1-1. 10.1109/TIP.2019.2946979.