Report IRGPU

Nathan Cabasso, Ferdinand Mom

16 mai 2022

Table des matières

1 Goal

The Harris corner detector is a corner detection filter that is commonly used in computer vision algorithms to extract corners and infer features of an image.

Corners are the junction of two edges, where an edge is a sudden change in image brightness. They are the important features in the image and generally termed as interest points which are invariant to translation, rotation and illumination.

Although corners are only a small percentage of the image, they contain the most important features in restoring image information, and they can be used to minimize the amount of processed data in various computer vision areas. They can also be used to compute the optical flow of a video.

1.1 CPU implementation

The Harris corner detector algorithm can be decomposed in several steps :

— **Step 1** : Convert image to grayscale.

— **Step 2** : Compute harris response on the gray scale image :

1. Compute a gaussian derivative (sobel) kernel.
2. Compute the image derivatives I_x and I_y.
3. Compute the structure tensor images I_x^2, I_y^2, and $I_x * I_y$
4. Convolve a gaussian filter with each structure tensor images.
5. Compute the determinant W_{det} and the trace W_{trace} based on previous convolution output.
6. Returns the Harris cornerness response of a given image ($\frac{W_{det}}{W_{trace}+1}$).

— **Step 3** : Perform non-maximal suppression using a morphological opening on Harris corner response.

— **Step 4** : Find top K best keypoints coordinates by sorting the filtered responses.

In order to implement the C++ version, we decided to port the existing python code from previous lab project using the *Twin-it !* dataset. It is worth mentioning that the dataset contained edges that needed to be removed from the Harris response before computing the best keypoints (using morphological erosion). We kept the erosion as a way to sanity check our C++ implementation but removed it when testing on real images. To make our life easier, we decided to abstract away the complexity by creating a *Matrix* class which comes in handy when porting the CPU code to GPU.

```
struct Matrix {
    Matrix();
    Matrix(int height, int width);

    Matrix operator*(const Matrix& rhs) const;
    Matrix operator*(const double& rhs) const;
    Matrix operator+(const Matrix& rhs) const;
    Matrix operator+(const double& rhs) const;
    Matrix operator-(const Matrix& rhs) const;
    Matrix operator/(const Matrix& rhs) const;
    Matrix operator>(const double& rhs) const;
    Matrix operator==(const Matrix& rhs) const;
    double max() const;

    // [...]

    int height, width;
    std::vector<double> data;
};
```

Most of our operations are implemented sequentially (i.e. we loop over the Matrix's values and output a new Matrix as a result). We could have optimized it using multi-threading but decided otherwise, since we only wanted to use our CPU implementation as a baseline for our GPU version.

1.2 GPU implementation

For the GPU implementation, we decided to use **Thrust**, a parallel algorithms library which resembles the C++ Standard Template Library (STL) and embedded it into our *Matrix* class which makes porting CPU code much easier. Here is an example :

```
Matrix compute_harris_response(const Matrix &image)
{
    int size = 3;

    auto img_x = gauss_derivative(image, size, 1);
    auto img_y = gauss_derivative(image, size, 0);

    auto gauss = gauss_filter(size);

    auto W_xx = convolution_2D(img_x * img_x, gauss, size);
    auto W_xy = convolution_2D(img_x * img_y, gauss, size);
    auto W_yy = convolution_2D(img_y * img_y, gauss, size);

    auto W_det = (W_xx * W_yy) - (W_xy * W_xy);
    auto W_trace = W_xx + W_yy;

    return W_det / (W_trace + 1.);
}
```

Listing 1 – CPU version

```
MatrixGPU compute_harris_response_gpu(MatrixGPU &image)
{
    int size = 3;
    auto img_x = sobel_filter_gpu(image, size, 1);
    auto img_y = sobel_filter_gpu(image, size, 0);

    auto gauss = gauss_filter_gpu(size);

    auto I_xx = img_x * img_x;
```

```
    auto I_xy = img_x * img_y;
    auto I_yy = img_y * img_y;

    auto W_xx = convolution_2D_gpu(I_xx, gauss);
    auto W_xy = convolution_2D_gpu(I_xy, gauss);
    auto W_yy = convolution_2D_gpu(I_yy, gauss);

    auto W_det = (W_xx * W_yy) - (W_xy * W_xy);
    auto W_trace = W_xx + W_yy;

    return W_det / (W_trace + 1.);
}
```

Listing 2 – GPU version

Here are the kernels we implemented :

- **grayscale(img)** which takes an 8-bit RGBA image and outputs a floating point grayscale representation.
- **gauss_filter(n)** which generates a Gaussian convolution kernel of size n
- **sobel_filter(n)** which computes the gradient of a gaussian filter of size n.
- **convolution_2D(img, kernel)** which computes a basic 2D convolution of *img* with *kernel*. We implemented 3 variants (naive, tiled, tiled with multiple loads per thread).
- **morph_apply_gpu(img, kernel, mode)** which computes a morphological operation to *img* with *kernel*. The parameter *mode* can be used to switch between erosion and dilatation.
- **morph_dilate_gpu(img, kernel_size)** which performs an optimized but approximated morphological dilatation using square kernel.

The CPU code for computing the top-K best keypoints coordinates differs from the GPU one as we fully relied on 'thrust' library for ease of process.

2 Benchmark

To measure the performance of our algorithm, we decided to use the following tools :

- **Google Benchmark** : straightforward to setup and have many options to benchmark such as counting the number of maximum run of our program given a time constraint.
- **nvprof (Nvidia profiler)** : enables to see the time spent on each kernel call. Useful to identify the bottlenecks.

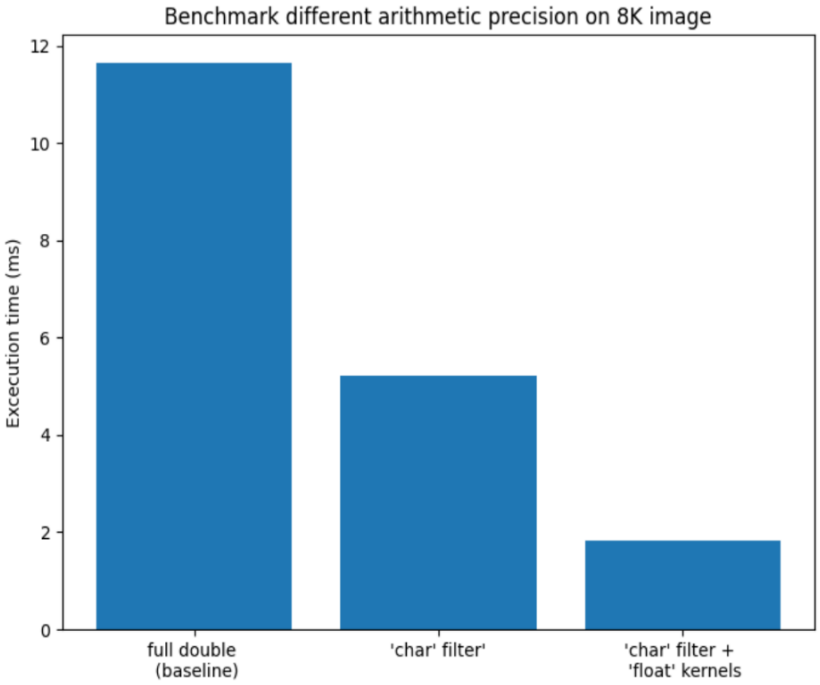
Unfortunately, we were not able to make NVIDIA Nsight Graphics work on our local machines. Therefore, we couldn't visualize the time spending on data transfer between different memory layers.

2.1 Bottleneck optimization

2.1.1 Morphological operation arithmetic precision

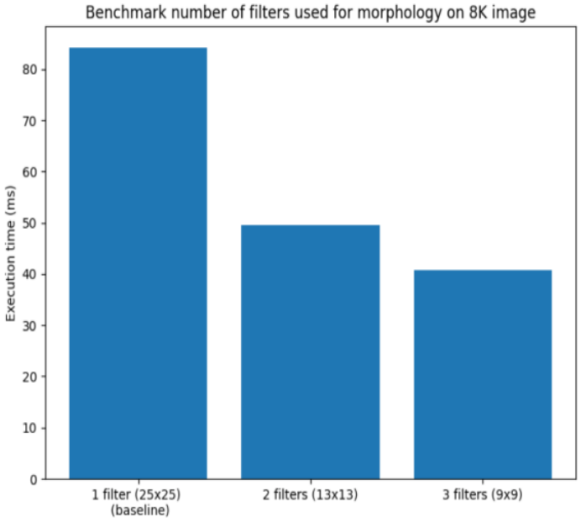
Throughout our benchmarks, we noticed that the main bottleneck was the morphological operations, especially when computing the erosion for images from the *Twin it* ! dataset. Since we used a 50*50 rounded filter, it was taking most of our compute time. First of all, since our morphological filter only consists of zeros and ones, we decided to use a *char* matrix instead of *double*. This allowed us to reduce the execution time of our kernel by half. Since GPUs are optimized for float we went ahead and replaced all our *double* with *float*. This also reduced the execution time by more than half.

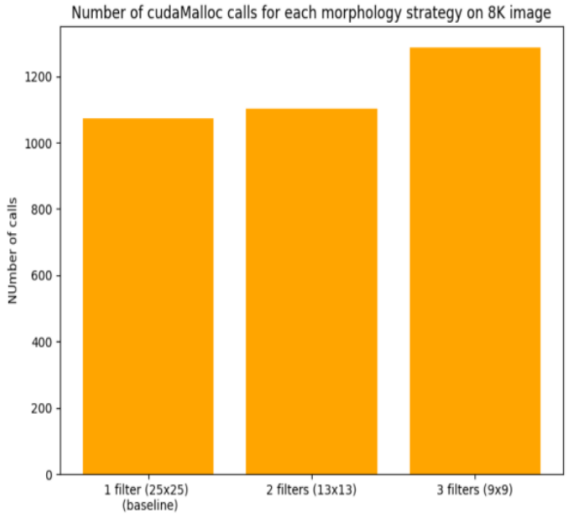
Here are the execution time of our kernels with different arithmetic precision :



2.1.2 Number of filters for morphology

Instead of performing convolution using only one filter of size *25*25*, we decided to approximate this process by splitting into 2 successive convolutions with 2 filters of size *13*13*. We even went further by splitting into 3 successive convolutions with 3 filters of size *9*9*.

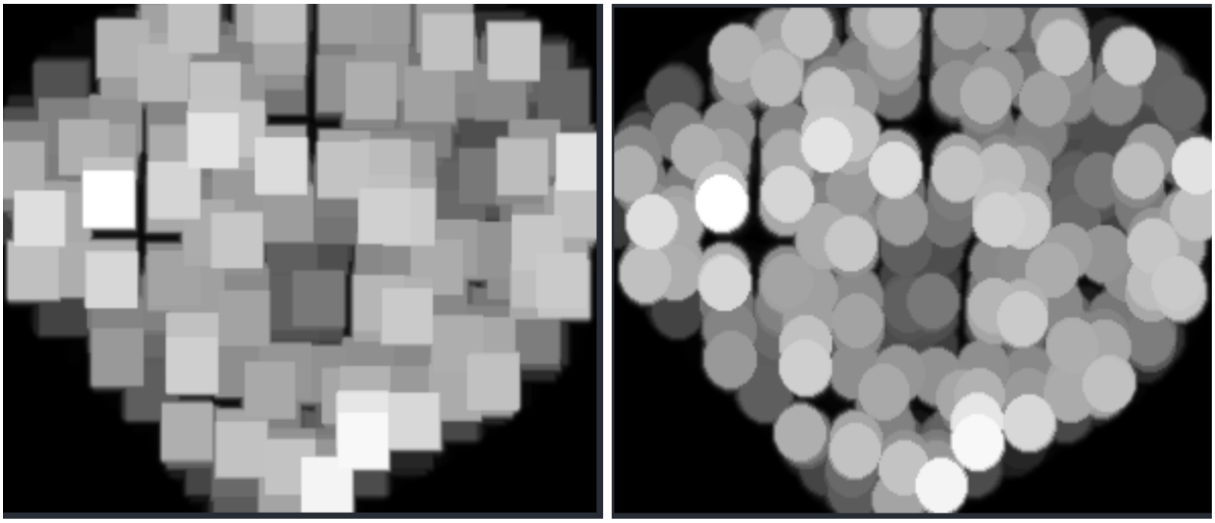




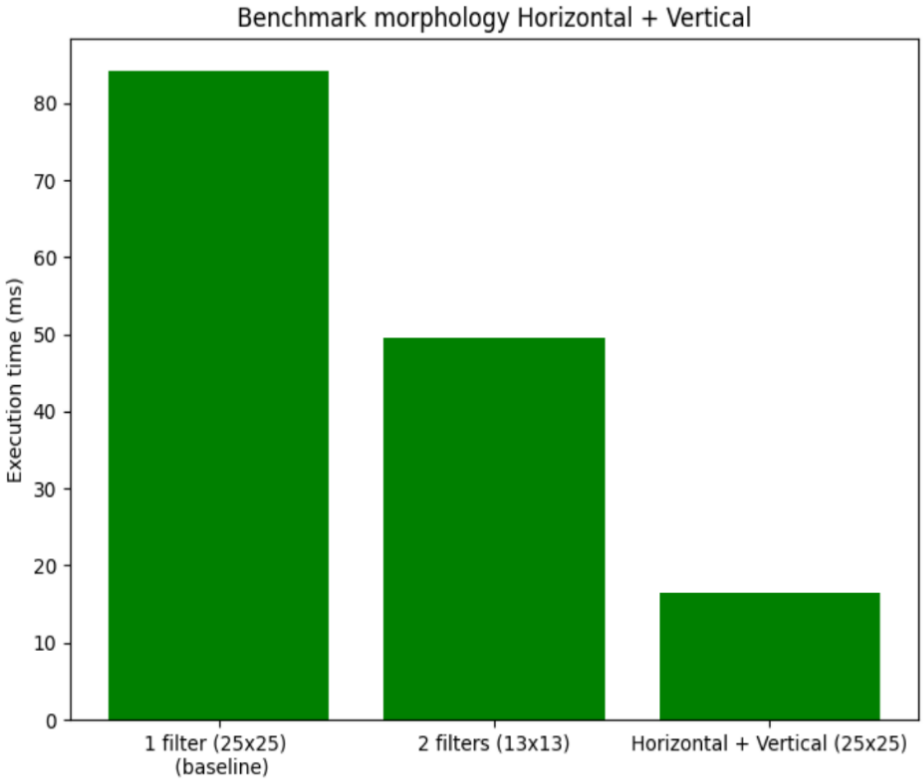
Despite the 3 filters (*9*9*) strategy resulting in faster execution time among the three, it is not worth to pick it because of the increase of *cudaMalloc* calls which results in slower overall execution. That is why the second option 2 filters (*13*13*) looks more appealing.

2.1.3 Horizontal + Vertical filters for morphology

We finally tried a last approach to optimize the morphology operation. Instead of computing the morphology in one go, we decided to split into 2 morphology operations with only horizontal and vertical. This means our kernel is now square instead of round, which is an approximation as we can see below.



Here are the result, the Horizontal + Vertical implementation is faster so we decided to go with it.



2.1.4 Tiled convolution

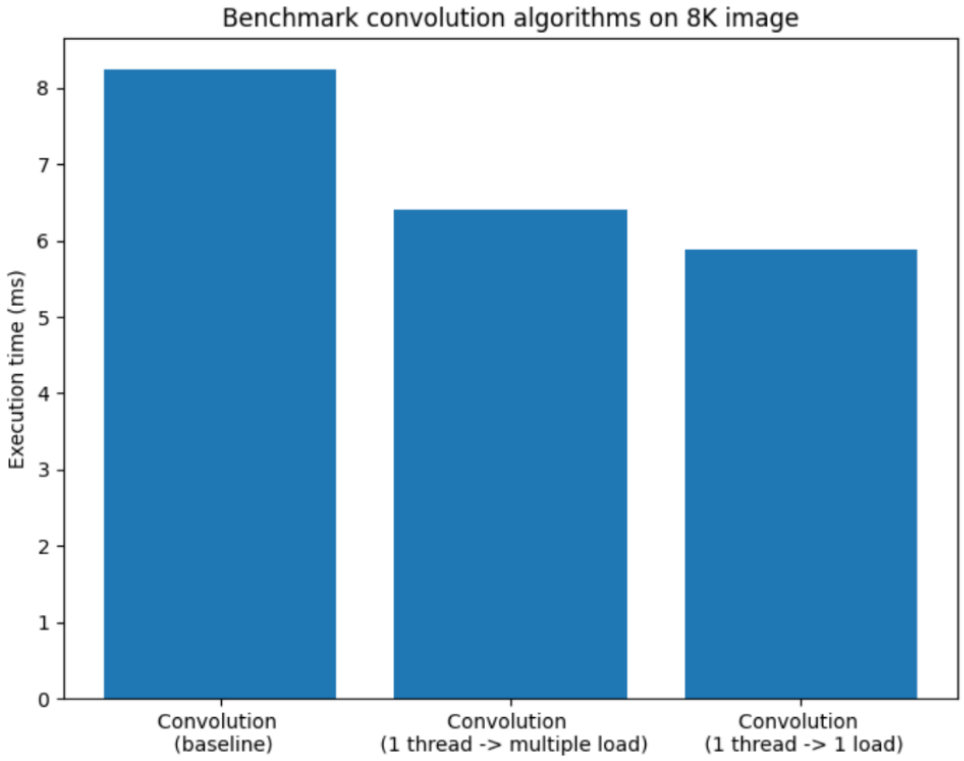
After trying to optimize our morphological operation, we decided to take a look to our convolution, which was taking 12.79% of our execution time.

We decided to try to compute our convolution using tiles which mean dealing with our image as a bunch of patches where convolution is actually performed on instead of the whole image. This allows us to load each patch on our shared memory, allowing faster memory access thus faster computing.

2 versions of it were implemented :

1. 1 thread → 1 load with early return
2. 1 thread → multiple loads

As you can see, the improvements are marginal but noticable.



2.1.5 Early free (fail)

We noticed that during the first run of our pipeline the *cudaMalloc* calls were taking most of the execution time. Then, subsequent runs would be much faster. We theorized that the reason for this was that after the first run, CUDA could reuse buffers that we just freed, since they have the same size.

Cold run (nvprof ./bench)

```
Total iteration = 1
--> 99.50%  560.56ms        35  16.016ms  2.3050us  560.39ms  cudaMalloc
    0.16%  920.29us        13  70.791us  4.6180us  420.46us  cudaDeviceSynchronize
    0.08%  436.55us        56  7.7950us     538ns  130.73us  cudaStreamSynchronize
    0.05%  285.28us        75  3.8030us  2.5100us  29.919us  cudaLaunchKernel
    0.05%  272.36us        41  6.6420us  1.8300us  71.360us  cudaFree
    ...
```

Hot run (nvprof ./bench –benchmark_min_time=5) :

```
Total iterations = 3098
    28.73%  2.34151s     54717  42.793us  1.0890us  456.27us  cudaDeviceSynchronize
    14.24%  1.16072s    235704  4.9240us     554ns  456.39us  cudaStreamSynchronize
    13.60%  1.10827s    315675  3.5100us  2.3460us  350.29us  cudaLaunchKernel
--> 12.54%  1.02178s    147315  6.9360us  1.9210us  115.53ms  cudaMalloc
    12.20%  994.06ms    172569  5.7600us  1.7430us  13.622ms  cudaFree
```

We thought that most of the *cudaMalloc* calls are from the abstraction we put in place for porting the code from CPU to GPU. Indeed, we are not reusing existing buffers which results in many *cudaMalloc* calls. To mitigate that, we tried to free Matrix objects as early as possible, using a C++ hack, so that CUDA could reuse buffers as much as possible inside the same run.

Thus, we attempted to "early-free" objects as follows :

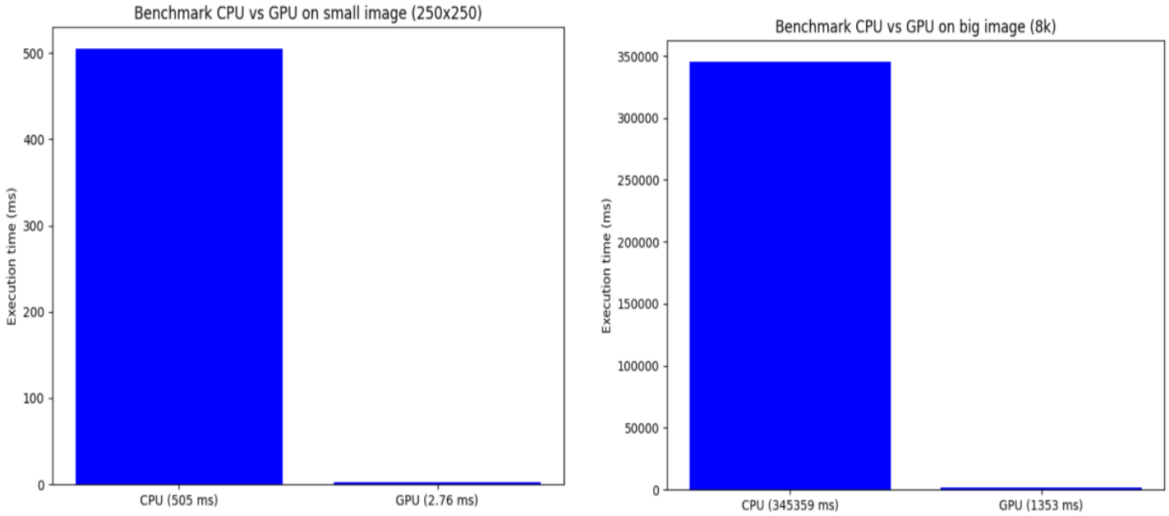
```
// [...]
auto I_xx = img_x * img_x;
{auto _ = std::move(img_x);} // C++ hack
// [...]
}
```

As we can see, our program spend in average, 14.953ms doing *cudaMalloc* which is a negligeable improvement compared to the previous cold run (16.016ms).

```
99.49%  523.37ms        35  14.953ms  2.3620us  523.21ms  cudaMalloc
0.17%  918.63us        13  70.664us  4.3960us  421.54us  cudaDeviceSynchronize
0.08%  420.61us        54  7.7890us     647ns  131.63us  cudaStreamSynchronize
0.06%  320.38us       101  3.1720us     149ns  149.55us  cuDeviceGetAttribute
0.05%  259.89us        73  3.5600us  2.5260us  19.670us  cudaLaunchKernel
0.04%  201.67us        41  4.9180us  1.8180us  59.146us  cudaFree
```

At the end, our intuition was not good so we reverted that change as it made our code less readable. Also, we noticed that the *cudaMalloc* call that was taking most of the total time was the first one, so it may be caused by the initialization of CUDA which make our approach pointless.

2.2 Results summary



2.3 Further improvement

For futher improvements, when computing the top-K best keypoints coordinate, we notice that we can stop the sorting after the K elements.

We also think that because of our extensive use of Thrust's device vectors, we could reduce the number of memory allocations by using more specialized kernels.