IRGPU: Harris corner detector

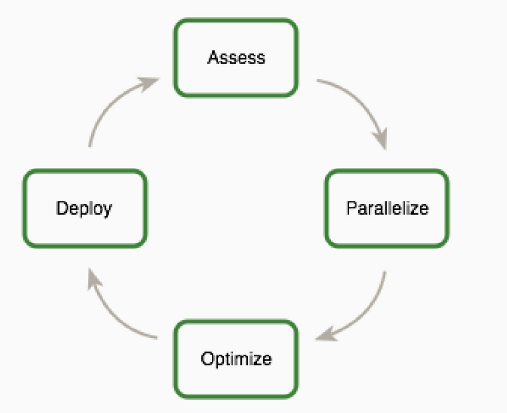
Nathan Cabasso, Ferdinand Mom - IMAGE 2023

I/ Goal

Implement an Harris Corner detector in CPU & GPU.

Pipeline:

- Port the Python code from lab session in C++ to get a working base.
- Port the C++ code to CUDA.
- Check all results are correct.
- Identify what can be optimized, implement, measure, repeat.



A) 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:
    - Compute a gaussian derivative (sobel) kernel.
    - Compute the image derivatives I_x and I_y.
    - Compute the structure tensor images ${I^2}_x$, ${I^2}_y$, and $I_x * I_y$
    - Convolve a gaussian filter with each structure tensor images.
    - Compute the determinant W_{det} and the trace W_{trace} based on previous convolution output.
    - 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.

A) CPU implementation

```
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;
};
```

- Abstraction for easier GPU code porting.
- Sequential implementation (No multi-threading)

B) GPU implementation

CPU

```
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.);
}
```

GPU

```
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.);
}
```

B) GPU implementation

- **Thrust**: a parallel algorithms library which resembles the C++ Standard Template Library (STL).

- **Kernels implemented**:
    - **grayscale(img)**: takes an 8-bit RGBA image and outputs a floating point grayscale representation.
    - **gauss_filter(n)**: generates a Gaussian convolution kernel of size *n*.
    - **sobel filter(n)**: computes the gradient of a gaussian filter of size *n*.
    - **convolution_2D(img, kernel)**: 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)**: 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)**: performs an optimized but approximated morphological dilatation using square kernel.

- 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.

II/ Benchmark

- **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.

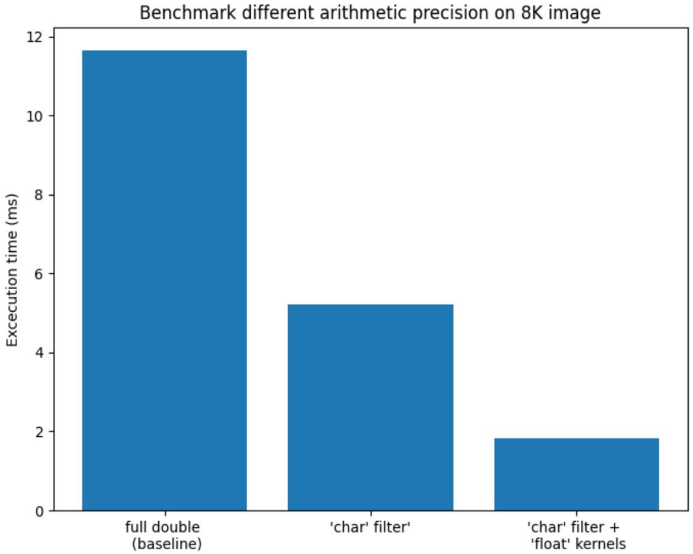
A) Bottleneck optimization

- Morphological operation arithmetic precision
- Numbers of filters for morphology
- Horizontal + vertical filters for morphology
- Tiled convolutions
- Early free (fail)

Morphological operation arithmetic precision

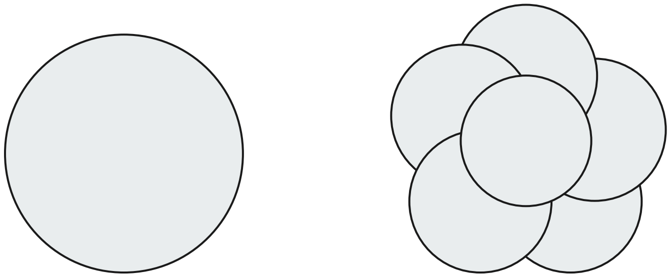
Idea: Since kernel is binary, reduce the size of the kernel by using char instead of double.

Morphological operation arithmetic precision

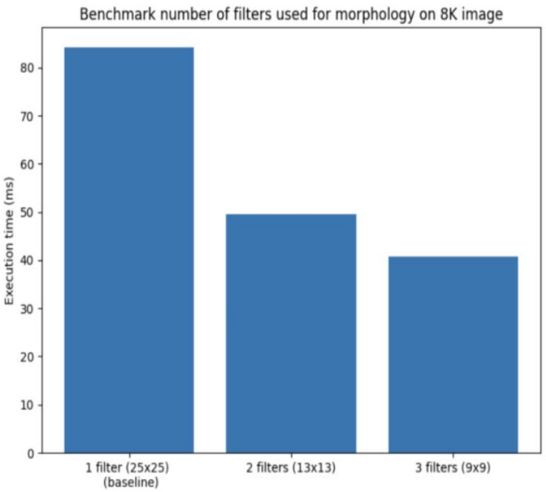


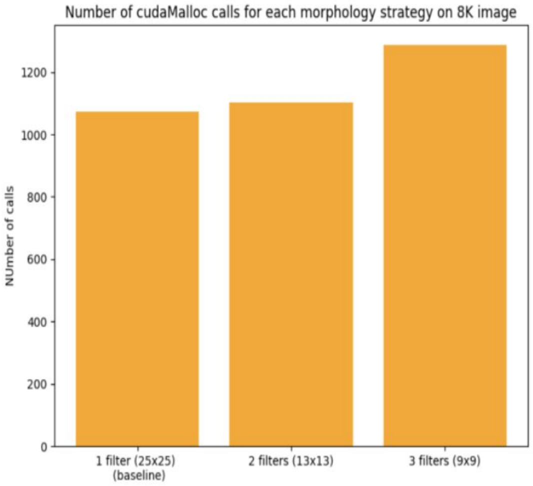
Numbers of filters for morphology

Idea: instead of using a big 25x25 morphological operation, perform two successive 13x13, or even three 9x9.



Numbers of filters for morphology

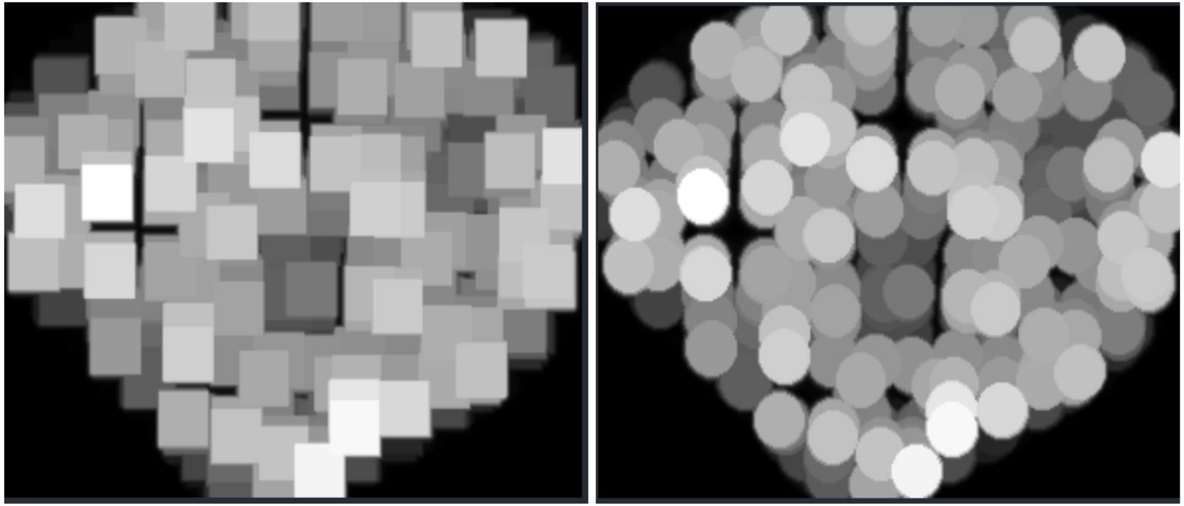




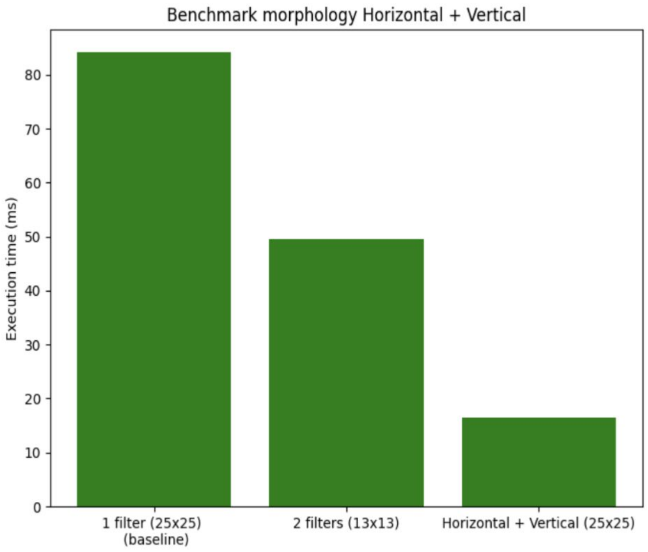
Horizontal + vertical filters for morphology

Idea: compute the morphology on the two axes separately.

Horizontal + vertical filters for morphology



Horizontal + vertical filters for morphology



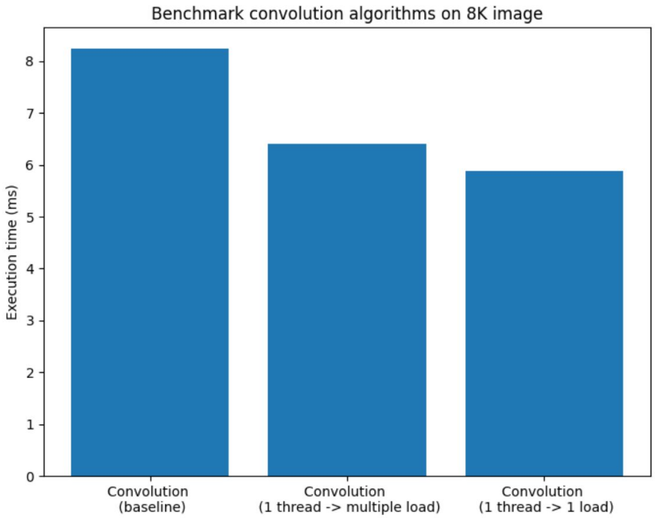
Tiled convolutions

Idea: load tiles in shared memory to allow faster convolution (which was taking 13% of our kernel execution time).

Two solutions:

- One thread <-> one load
- One thread <-> multiple loads

Tiled convolutions



Early free (fail)

<u>Idea:</u> cudaMalloc calls are taking a lot of time at the beginning, then CUDA seems to reallocate buffers much faster.

What if we freed buffers as early as possible to allow reuse inside a single run?

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
    ...
```

Early free (fail)

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

Early free (fail)

Before

Cold run (`nvprof ./bench`):

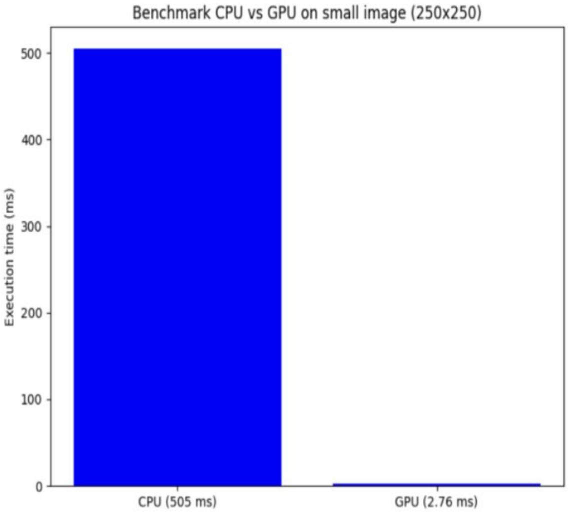
```
Total iteration = 1
--> 99.50%  560.56ms        35  16.016ms  2.3050us  560.39ms  cudaMalloc
    0.16%  920.29us        13  70.791us  4.6180us  420.46us  cudaDeviceSynchronize
    0.08%  436.55us        56  7.7950us     538ns  130.73us  cudaStreamSynchronize
    0.05%  285.28us        75  3.8030us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

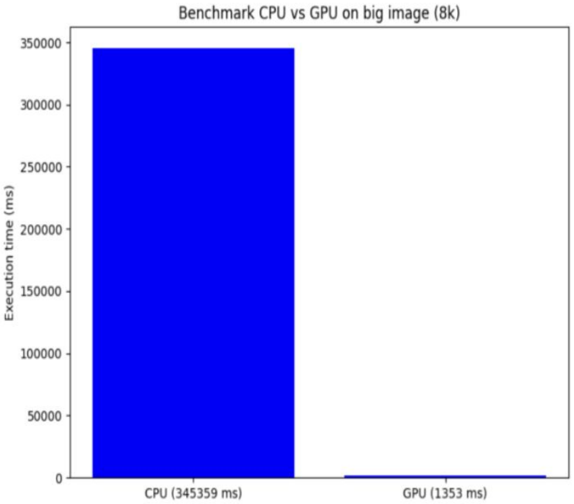
After

Cold run (`nvprof ./bench`):

```
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
```

B) Summary


Benchmark CPU vs GPU on small image (250x250)
Execution time (ms)
500
400
300
200
100
0
CPU (505 ms)
GPU (2.76 ms)


Benchmark CPU vs GPU on big image (8k)
Execution time (ms)
350000
300000
250000
200000
150000
100000
50000
0
CPU (345359 ms)
GPU (1353 ms)