Video Motion Detection
Final project for SPM

Alex Colucci

13th June 2022

1 Introduction

The following report explores a possible way to parallelize the solution to the problem of counting the number of frames containing motion in a given video stream S. A simple but effective algorithm is to apply a grayscale and blur effect to the frames. Then the first frame is taken as the background and for each of the subsequent frames, calculate the difference between the frame and the background in terms of pixels. If the number exceeds a certain threshold, it means there is motion. At this point, the algorithm needed to achieve the goal can be seen as a composition of the following functions:

- r reads a frame f belonging to a stream S with a resolution of height h and width w;
- g applies grayscale on a given frame f by reducing from 3 channels (RGB) to a single one;
- b applies blur on a given frame f of 1 channel;
- m computes the difference between the background frames and the analyzed frame and checks whether there is motion or not.

Since the domain concerns videos and no assumptions can be made about their length, the safest choice is to use a streaming parallel approach. Moreover, the computation on one frame f is independent of the others (except the background) and this does not imply any requirement on the ordering of the frames. In other words, it is an embarrassingly parallel problem.

1.1 Methodology

Let us briefly describe the methodology that will be used to conduct the experiment.

Based on the above assumptions, the proposed goal is to find the skeleton with the best service time T_s and analyze the speedup as the parameters change. In this case, the videos and the blur algorithms.

- For the videos, two different videos will be used
  - house720.mp4, with a 720p resolution and 84 frames in total;
  - door1080.mov with a resolution of 1080p and, 925 frames in total.
- For blur instead, different algorithms will be tested:

- the standard OpenCV's blur algorithm to apply a box blur, $H1$ of size 3×3;
- a naive convolution algorithm to apply the different kernels $H1$, $H2$, $H3$, $H4$, $H1\_7$ (the last one is a 7×7 matrix of ones);
- an optimized algorithm for applying just the box blur, namely the one defined by the $H1$ matrix.

The kernels tested to blur the image are:

$$H1 = \frac{1}{9}\begin{bmatrix} 111 \\ 111 \\ 111 \end{bmatrix} \quad H2 = \frac{1}{10}\begin{bmatrix} 111 \\ 121 \\ 111 \end{bmatrix} \quad H3 = \frac{1}{16}\begin{bmatrix} 121 \\ 242 \\ 121 \end{bmatrix} \quad H4 = \frac{1}{8}\begin{bmatrix} 111 \\ 101 \\ 111 \end{bmatrix}$$

2 Proposed skeleton

Once the functions have been identified, the next step is to find the best skeleton from a theoretical point of view before implementing with code. Given the functions r, g, b, m, it can be seen that r is I/O bounded and cannot be parallelized. At this point one can reason about the sequential composition $Comp(g, b, m)$, from which by applying the rewriting rules one can obtain different skeletons.

$$N.F. = Farm(Comp(g, b, m)) \tag{1}$$

The most effective model is the normal form Eq. (1), in facts it is shown to guarantee the best service time for a parallel streaming skeleton.

$$Pipe(Seq(g), Farm(b), Seq(m)) \tag{2}$$

One alternative that can be considered is to use a pipeline with a farm Eq. (2). That in theory should not exceed normal form because communication between stages has a big impact in terms of overhead. Because a given frame f that is executed in stage 1, must be passed into stage 2 and so on. Not only that, but each stage is a different thread, so on top of that, locality of reference would not be fully exploited. In reality, however, the communication overhead can be masked, at least as long as the bottleneck is not the I/O operation. So it would be interesting to see in practice how this model performs.

$$Farm(Comp(g, Map(b), m)) \tag{3}$$

Another skeleton could be a two-tier skeleton composed by a farm with a nested map Eq. (3). But it would not be worth it in terms of the overhead of splitting and joining the frame, because it would lead to not taking advantage of the locality of reference.

Based on the above observations, the normal form Eq. (1) and the pipeline of farms Eq. (2) will be taken as a reference from here on.

2.1 Performance model

Considering the normal form Eq. (1), the service time is given by:

$$T_s = \max\{T_e, \frac{t_g + t_b + t_m}{n}, T_c\} \tag{4}$$

where T_e and T_c are the emitter and collector times, respectively. The former computes the function r, while the latter in this case is only an accumulation variable. The other times are the time for the Farm functions with n workers, respectively.

Estimate At this point we moved on to just implementing the functions to estimate their execution time and service time for a single frame. The results of the functions r, g and m can be seen in Table 1, while the results of the function b are exploded into the different implementations and kernels used, shown in Table 2. Two observations can be made:

- the largest costs are r and b. But this was expected because r is an I/O operation and b has a higher complexity than g and m, which are linear over the number of pixels;
- the execution times on videos of different resolution are proportional to the number of pixels

Video resolution	Read frame r	Deque	Grey g	Motion m
720p	2945	1	2109	404
1080p	7247	1	5676	985

Table 1: Execution time in μs for the functions r, g, m and deque.

Video resolution	OpenCV	Box blur	H1 - H2 - H3 - H4	H1_7
720p	1780	7158	24559	61234
1080p	3916	16640	55775	138558

Table 2: Execution time in μs for the different blur algorithms b and convolution matrixes.

Considering the service time Eq. (4) for the normal form, then with the estimated values, for a 1080 video it becomes:

$$T_s = \max\{7247, \frac{5676 + 55775 + 985}{n}, T_c\} = \max\{7247, \frac{62436}{n}, T_c\} \tag{5}$$

In other words, this means, that under the hypothesis that for each thread the ideal service time is reached and the T_c is negligible, then with $n \geq 9$, the bottleneck starts to become the function r, hence T_e will be the maximum.

Similar reasoning, exploiting the same formula can be done for the pipeline of farms Eq. (2).

3 Implementation

This section explains the implementations details and defines the documentation necessary to understand how to replicate the experiments conducted.

3.1 Details

Several parallel solutions have been proposed, namely C++ Threads, FastFlow and OpenMP. For each of them, there is an implementation that allows motion to be detected on a video stream and an implementation that instead first reads the entire video and only then performs the computation. The motivation behind the last case, is to understand how much impact the r function has in some cases.

C++ Threads As for the solution with C++ threads, it is structured in the following way. It's a fork-join model that to handle synchronization uses a safe queue and an accumulator of atomic type to maintain the number of frames with motion. Threads are created, which will go and read frames from the queue and terminate only when an empty frame arrives. On the other hand, the emitter reads frames from the video and places them in the queue. When it reaches the end of the video, it will insert a number of empty frames, as many as the number of threads, to signal the end of the stream (EOS). Two approaches are proposed, one where the pinning technique is not used and one where it is used.

FastFlow The implementation of FastFlow, on the other hand, involves the high level parallel patterns and several normal forms Eq. (1) have been tried:

- farm with emitter, workers and the collector to accumulate;
- farm as above, but without the collector. Each worker will accumulate the number in a local counter and at the end they will increase the global counter that is an atomic variable;
- farm with emitter, workers and the collector to accumulate, but with the on demand scheduling.

Since one of the advantages of fast flow is the speed of stacking different skeletons, the farm pipeline Eq. (2) was also tried in this case

OpenMP The OpenMP was a later implementation, born out of the need to have a third comparison. Here we simply use a single block to create the tasks, and then a task block to perform motion detection with a synchronization point to increase the accumulator, which is an atomic block.

3.2 Documentation

Here a description of how the project is structured, how to compile, execute and see the results.

3.2.1 Download

Firstly, in order to download the project, execute the following commands:

```
$ git clone https://github.com/reuseman/video-motion-detection
$ cd video-motion-detection/
$ ./include/ff/mapping_string.sh
```

3.2.2 Project structure

Here there is a brief introduction on the most important things that can be found in the project folder.

- `docs/ : the LaTeX source for this report;`
- `include/ : the header files;`
  - `ff/ : fastflow framework;`
  - `processing/ : motion detection implementation;`
  - `benchmark.hpp : a tool to benchmark functions;`
  - `shared_queue.hpp : multiple producer/consumer queue;`
  - `...: other files`
- `src/ : the main programs:`
  - `main.cpp : to detect motion and benchmark;`
  - `meter.cpp : to benchamrk the single functions;`
- `videos/ : the videos used for the experiments;`
- `views/ : Python Notebook to visualize results;`
- `...: other files`

3.2.3 Compile

By running CMake you can define at compile time what will be the blur algorithm or kernel used in the executable. To do this we need to set the `CMAKE_BUILD_TYPE` flag to one of the following values: `OPEN_CV_BLUR`, `H1`, `H2`, `H3`, `H4`, `H1_7`, `BOX_BLUR`. Respectively those described in Section 1.1. By default the last one will be used.

```
$ cd video-motion-detection/
$ ./include/ff/mapping_string.sh
$ mkdir build && cd build/
$ cmake -DCMAKE_BUILD_TYPE=BOX_BLUR ..
$ make
```

3.2.4 Execute

At this point, you will have two executables `meter` and `motion-detection`. In order to see the full documentation, use `./motion-detection -h`.

The first one, `meter`, allows you to estimate in microseconds the execution time of individual functions r, g, b, m given an input video.

```
$ ./meter -s ../videos/house720.mp4
```

While `main.cpp` which represents the actual program has two facets.

- to perform motion detection on a video;
- to perform a full benchmark of the motion detection a video with the different parallel implementations with threads from $1, 2, 4, 8, \ldots, 32$.

```
$ ./motion-detection -s ../videos/house720.mp4
$ ./motion-detection -s ../videos/house720.mp4 -w 16 -m 1
$ ./motion-detection -s ../videos/house720.mp4 -b
↪  benchmark-house-box_blur-5iter -i 5
```

Let us explain the above examples. The first command allows the algorithm to run sequentially on the video. The second one, on the other hand, allows executing the algorithm in a parallel with a number of workers equal to 16 and with FastFlow. The third on the other hand allows you to start the benchmark function, which will run C++ threads, FastFlow and OpenMP. Each of them with a number of workers ranging from $1, 2, 4, 8, \ldots, 32$ and will be repeated a number of times equal to the iterations i. The result will be written to a `results.csv` file with the name of the run called `benchmark-house-box_blur-5iter`.

3.2.5 Visualize results

Finally, the results of the csv file can be viewed in the Jupyter Notebook by executing the following commands.

```
$ cd video-motion-detection/
$ python3 -m venv .venv
$ source .venv/bin/activate
$ pip install -r view/requirements.txt
$ jupyter notebook
```

4 Results

This section shows the results of the implementation described in section Section 3.1 and are obtained on the 1080 video with the convolution algorithm using the $H1$ matrix described in Section 1.1. The machine used for the testing was provided with Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz with 32 cores.

Let us first analyze the threads, then the different options tried with FastFlow, and finally a general comparison.

Regarding threads, two options were tried, namely with and without pinning on the 32 cores. In such a way as to be able to avoid the overhead of the context switch. As can be seen in Fig. 1, pinning does not go to improve speedup at all, in fact when the number of cores is low the situation gets worse. Here the explanation could be that setting affinity leads to making worse choices than the kernel would have made, but for a number of threads approaching the number of cores in the machine there begin to be slight and sporadic improvements.

Turning instead to the implementation with FastFlow, the behaviors show subtle differences in Fig. 2. Analyzing `ff` and `ff_acc`, we see how the elimination of the collector in `ff_acc` allows for one free thread to use and thus greater speedup when the machine cores are saturated. It therefore makes sense to avoid using a thread just to accumulate the number of frames with motion. Considering instead `ff` and `ff_on_demand`, we see how the last one has only slightly better performance, but this was an expected behavior, due to the fact that motion detection computation presents for each frame a fairly homogeneous complexity, plus this scheduling makes sense when the emitter does not become the bottleneck, but this is not our case. Finally, between the normal form `ff` and the pipe with the farm `ff_pipe`, it can be seen that the latter has slightly better behavior initially but then gets worse. The normal form is theoretically better, but it must be said that the steady state alternative has communication costs that are masked, at least until the bottleneck becomes the emitter. From that point on, the normal form begins to dominate.

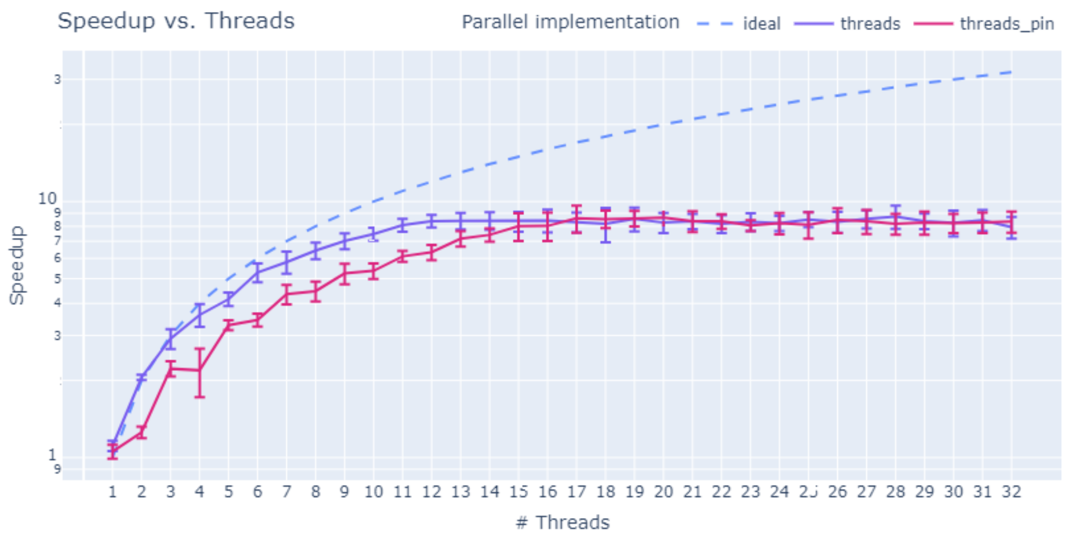

Figure 1: Speedup of Threads vs Pinned threads with the convolution algorithm with H1 over a 1080p video.

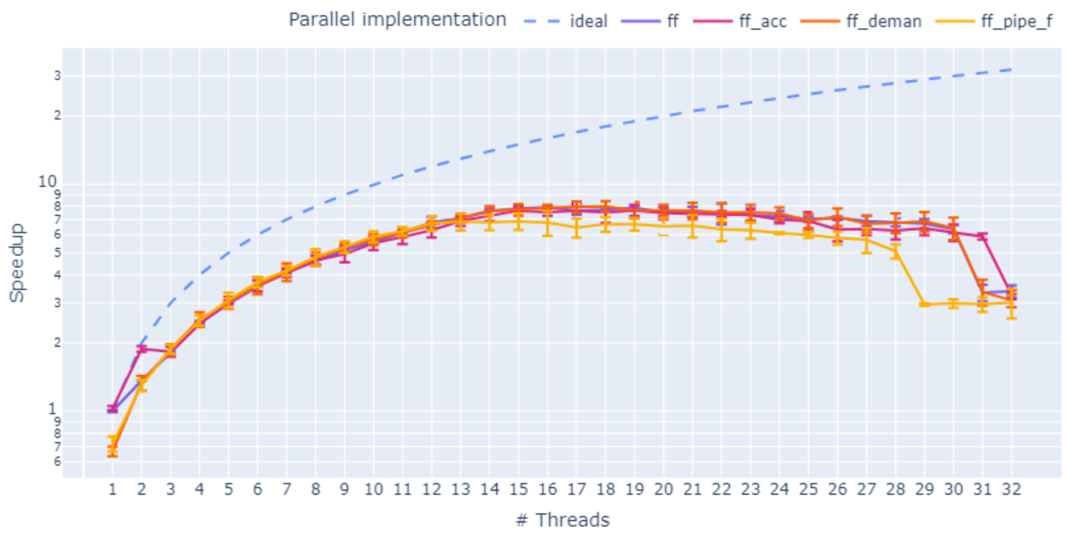

Figure 2: Speedup of FastFlow with the convolution algorithm with H1 over a 1080p video.

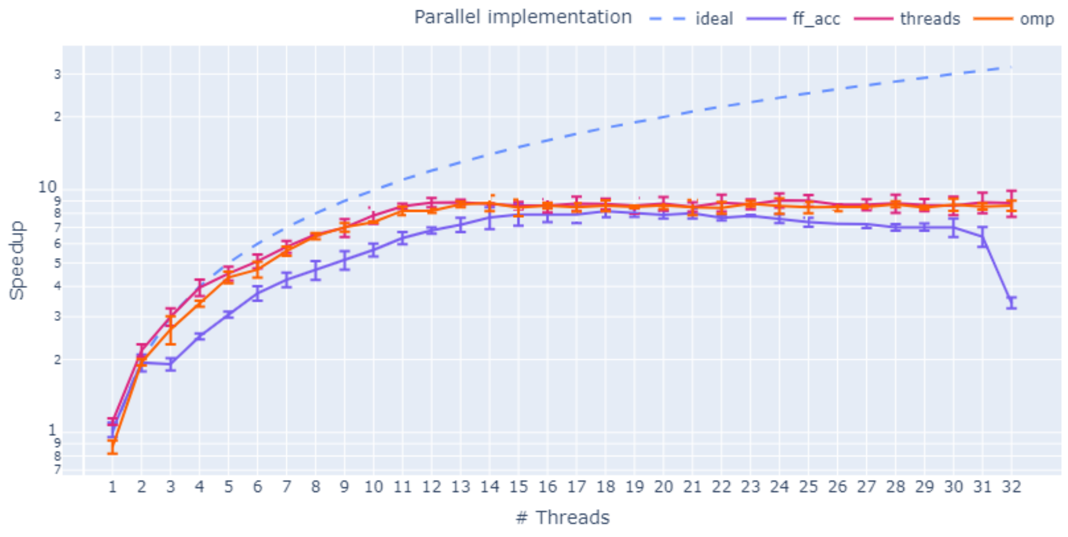

Figure 3: Speedup of Threads, FastFlow and OMP, with the convolution algorithm with H1 over a 1080p video.

To conclude, one can look at the speedup in Fig. 3 that reports the threads, `ff_acc` and OMP. The implementation with threads remains almost always the best, but the gap between the first two is not as marked. Fast Flow has a slightly lower speedup with a low number of threads, but as the number of threads increases it recovers. There is a slight downward spike at the end that could be due to a context switch on the emitter that has become a bottleneck in that case. Moreover, by looking at the efficiency Fig. 4 you can see that has dropped a lot, so throwing more threads at the problem, wouldn't help much.

As for the other kernels, the models have extremely similar behavior. For more details, you may refer to the attached code.

Finally, in figure Fig. 5, there is the case where the entire video is first loaded into memory. Obviously, the speedup improves for all implementations, but this was also expected due to the fact that in Section 2.1 we showed that the bottleneck is the I/O operation of the stream.

If we look at what was the estimated service time in the proposed performance model Eq. (5), we can see that the theoretical lower bound was $7247\mu s$. At this point, taking into consideration the best service times in obtained Table 3, we can see that we came very close. Obviously, there is some overhead that was not taken into account in the estimate, such as the time for concurrent access to the shared queue and the time to set up the whole parallel computation.

	Threads	Fast Flow	OMP
Minimum Ts	7292.59	8092.39	7494.48
Reached with #threads	24	18	14

Table 3: Minimum Service time T_s achieved by the models with the numbers of threads needed.

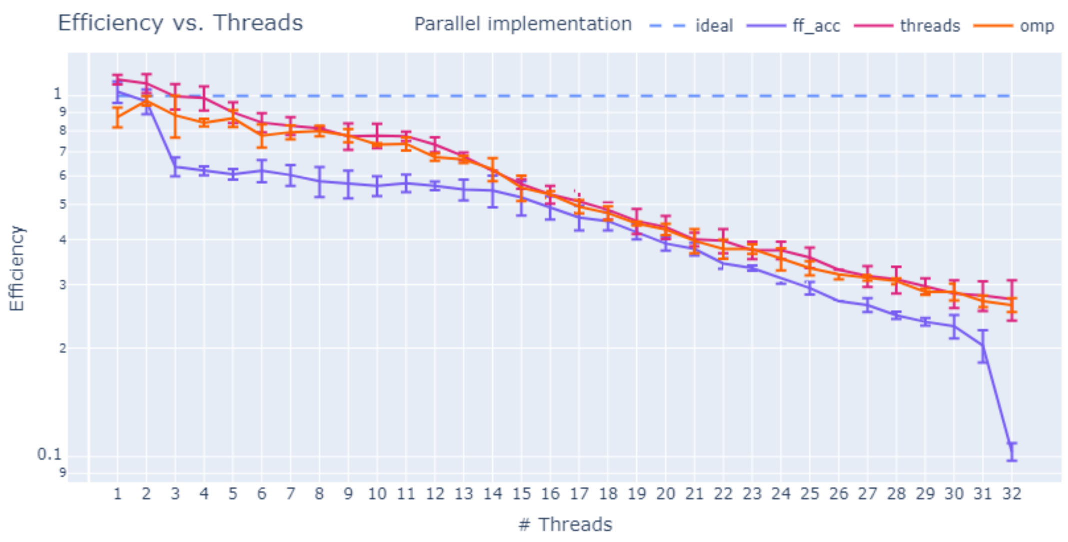

Figure 4: Efficiency of Threads, FastFlow and OMP, with the convolution algorithm with H1 over a 1080p video.

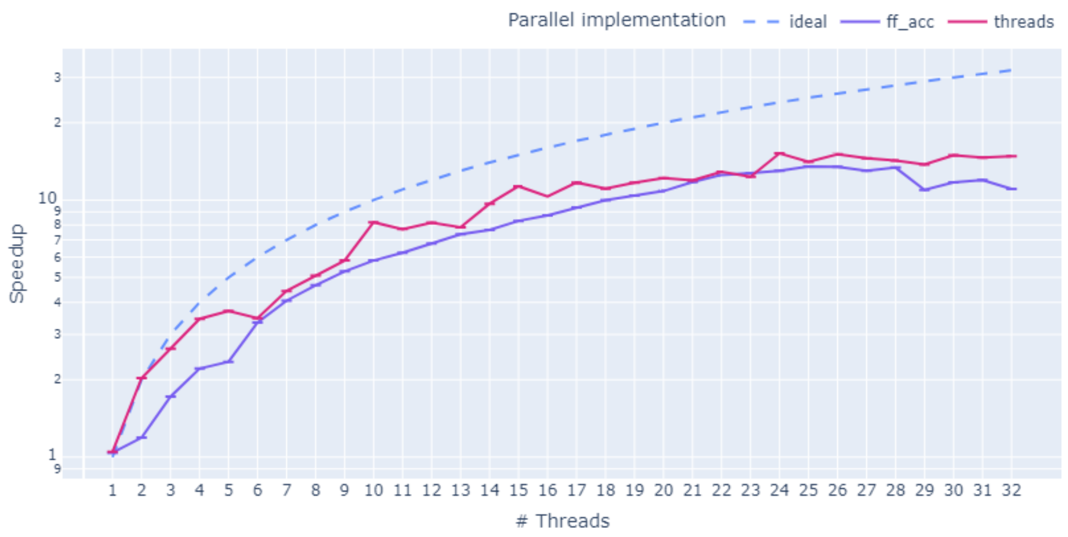

Figure 5: Speedup of Threads and FastFlow, with the convolution algorithm with H1 over a 1080p video that is preloaded in memory.