Video Motion Detection

Alessandro Puccia
Matricola 547462
Department of Computer Science

July 2022

1 Introduction

Given a video, a simple motion detection algorithm can be split into 3 parts: greyscaling, convolution and detection where the detection happens by counting the number of different pixels of the convoluted frames to the background frame. In particular, convolution happens considering a kernel that can assume different odd sizes starting from 3x3. Using bigger kernels result in a more blurred frame at the cost of an increased computation time.

Before discussing the parallel solutions, I started considering how the algorithm behaves considering a sequential execution. To ease the computation done in the convolution part, all the frames are padded considering the kernel size. Over a full high definition video and a H1 kernel of size 11x11, I got the following results:

- **Reading** a frame takes about 9 milliseconds.
- **Padding** a frame takes about 5 milliseconds, it involves the `copyMakeBorder` function of openCV.
- **Greyscaling** a frame takes about 7 milliseconds, the largest amount is taken by the allocation of a new frame with the function `Mat::zeros`.
- **Convoluting** a frame takes about 203 milliseconds.
- **Detecting** a frame takes about 0.5 milliseconds.

Reading, padding and the creation of a frame are performed by using openCV operations while greyscaling and the convolution are reimplemented as required. Since detecting a frame takes less time than the other phases and since that part of the algorithm has similarities with the convolution part I'll put them together (*conv_det*). Reading, padding and the creation of a new frame in the greyscaling phase are not parallelizable.

2 Parallel architectures

Starting from a $Comp(Seq(read), Seq(pad), Seq(grey), Seq(conv\_det))$ I applied some refactoring rules and considered also `rplshell` solutions. I considered first the following solution (see fig. 1):

$$Pipe(Seq(read), Seq(pad), Seq(grey), Farm(conv\_det))$$

where the *farm* component has the *grey* stage as an emitter and does not have the collector since the workers will simply accumulate values on a shared state variable. Given the previous timings and assuming to employ 29 workers for the farm component we have an ideal service time of about 9 milliseconds. This is the lower bound because of the *read* stage. Drawbacks of this solution is the time needed to access data structures to pass intermediate results to the following stages and the movement of data between caches.

Another solution (see fig. 2) would be to instead consider:

$$Farm(Comp(Seq(pad), Seq(grey), Seq(conv\_det))$$

where the *farm* emitter is a thread that performs the reading of a frame and also here there is no collector for the same motivation of before. Given the previous timings and considering 31 workers, we should get a service time that is about 9 milliseconds as before. This should be the better solution because there is less overhead since there is only communication between the emitter and the farm workers. As we will see the pipe solution performs really well and have results that are comparable with the farm one.

Another possibility would be to consider:

$$Pipe(Seq(read), Seq(pad), Seq(grey), Map(conv\_det))$$

where the scatterer is the *grey* stage. I don't think that this is worth it because we would then need a gatherer thread that performs the detection and, as discussed before, this is negligible than the other parts. In the following, we will consider the first two parallel architectures.

2.1 Pipe

In the standard `C++` thread implementation, I considered the `main` thread as the *reader* (*loader*) stage. Before reading the frames the `main` thread spawns the *pad* stage, the *grey* stage and a given amount of workers. All the stages execute a `while true` loop. The reader will exit upon reading an empty frame that is also propagated to the other stages and workers that, after checking it, will exit the loop. The data structure considered for the communication is a `deque` plus a mutex and a condition variable to ensure thread safety.

When considering instead the FastFlow implementation I used the `ff_Pipe` and `ff_Farm` nodes. In particular, the farm node does not have the collector and has the *grey* stage as an emitter. For the scheduling policies, I considered the default *round-robin* one with unbounded queues size and the *auto-scheduling* with queues' size 1.

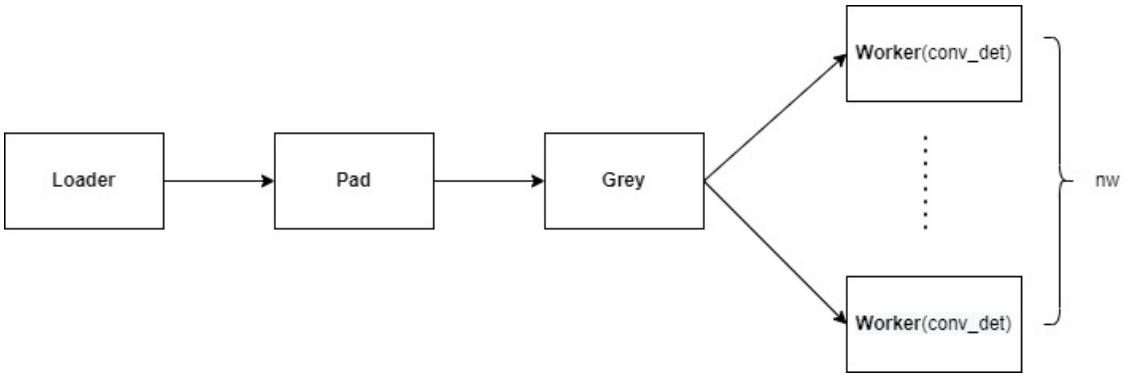

Figure 1: $Pipe(Seq(read), Seq(pad), Seq(grey), Farm(conv\_det))$ architecture where the farm emitter is the *grey* node. Each arrow represents a data structure, but in the `CPP` implementation the farm component will have only one `deque`.

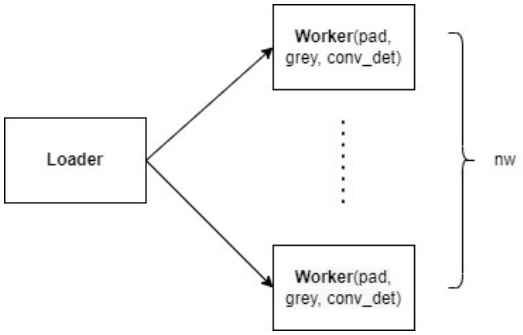

Figure 2: $Farm(Comp(Seq(pad), Seq(grey), Seq(conv\_det)$ architecture where the farm emitter is the *loader* node. Each arrow represents a data structure, but in the `CPP` implementation the farm will have only one `deque`.

2.2 Farm

In the standard `C++` thread implementation, I considered the `main` thread as the frame reader. Before reading the frames the main spawns the given amount of workers that execute a `while true` loop. The reader then starts a `while true` loop from which he will exit upon reading an empty frame. This frame is also propagated to the workers that after checking it will exit the loop. The data structure used for communications is, as before, a thread-safe version of a `deque`.

For the FastFlow implementation I considered the `ff_Farm` where there is no collector and the emitter is a `ff_node_t` that acts as frame loader. I considered the same scheduling policies as before.

3 Results

The target machine for the experiments is equipped with an Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz Ghz with 32 cores. The following experiments are performed assuming a kernel size equal to *11x11*, considering a normal average as defined by the *H1 kernel* and three resolutions of the same video that contains *270* frames (excluding the background): *standard definition* (960x540), *high-definition* (1280x720) and *full high-definition* (1920x1080). The completion times considered are produced out of an average of 5 iterations. I did not load the whole videos in memory but I processed them frame by frame. The plots that are presented in this report are related to the full high-definition video but, for the other resolutions, similar considerations can be made.

Starting from the *completion time* (in figs. 3 and 4), we can see that the standard `C++` threads solution is a lot faster then the FastFlow ones when using less threads than the ones that could be run in parallel in the remote machine. At about 26-28 farm workers in the pipe solution we achieve the smallest completion time for FastFlow while for the complete farm solution the smallest completion time is reached at about 30-32 threads.

Since the number of frames is much bigger than the number of stages both in the complete farm and pipe solutions, we can estimate the *service time* by dividing the completion time by the number of frames. In both solutions, when exploiting all the cores, we have a service time that is about 6-9 milliseconds more than the ideal one. I think that this overhead can be explained by a small part due to the setup of the parallel activities and the larger one due to the passing of intermediate frames, that are dependent on the resolution considered, and that need to be moved between caches.

Considering *speedup* (in figs. 5 and 6), the standard `C++` thread implementation in the farm solution offers a speedup that is bigger than the one achieved with FastFlow. This big difference is not present when considering the pipe solution. FastFlow implementations have similar speedup in both solutions.

Considering *scalability* (in figs. 7 and 8) the FastFlow solutions that employ auto-scheduling are the ones that scale better. The auto-scheduling solution scales better because from my results I noticed that the completion time when using only one worker is bad, almost twice of the other implementations.

Considering *efficiency* (in figs. 9 and 10), in the pipe solution, all three implementations have almost the same efficiency in the segment 26-30 threads while in the farm solution the FastFlow implementations have a bigger difference than the `C++` thread implementation in all the considered worker configurations.

4 Project structure

The project is structured in the following way:

- The `bin` folder will contain the `sequential.out`, `farm.out`, and `pipe.out` executables.

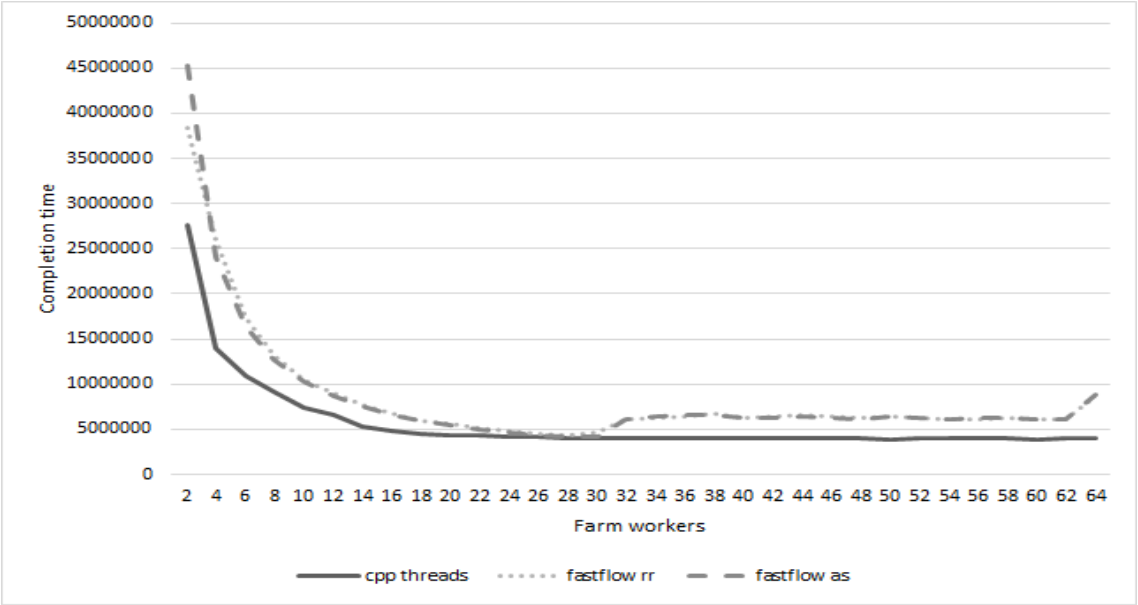

Figure 3: Completion time for the pipe architecture with a full high-definition video and kernel size equal to 11.

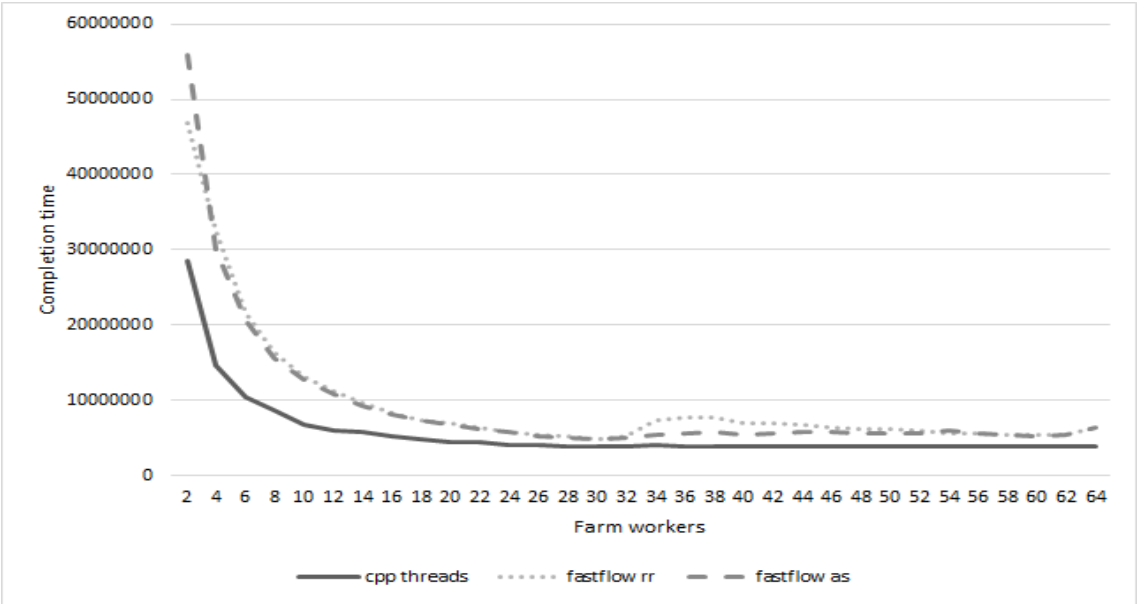

Figure 4: Completion time for the farm architecture with a full high-definition video and kernel size equal to 11.

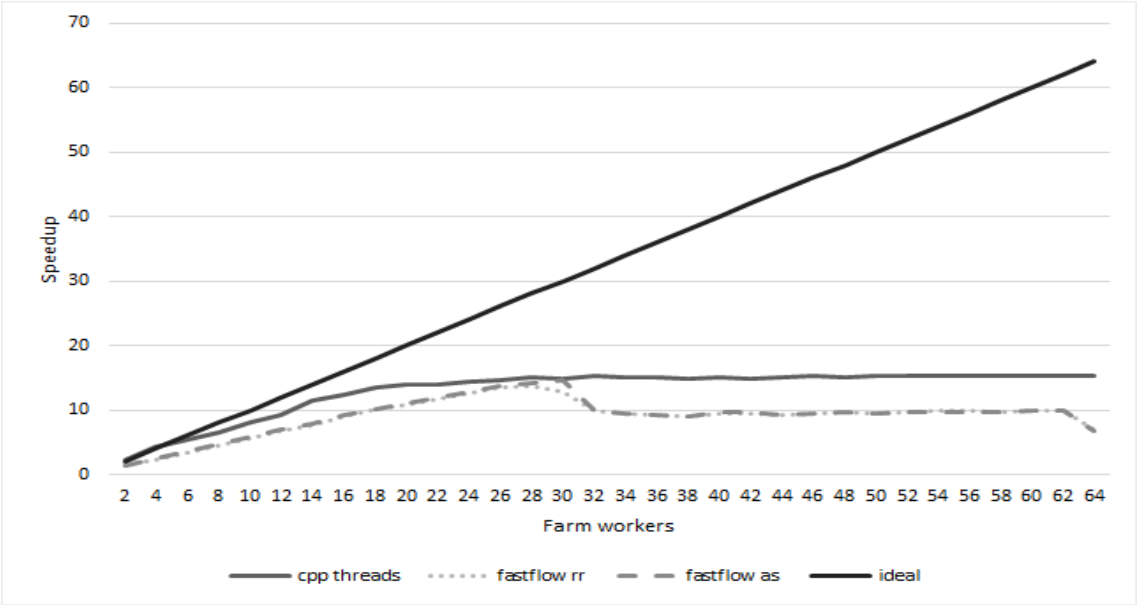

Figure 5: Speedup for the pipe architecture with a full high-definition video and kernel size equal to 11.

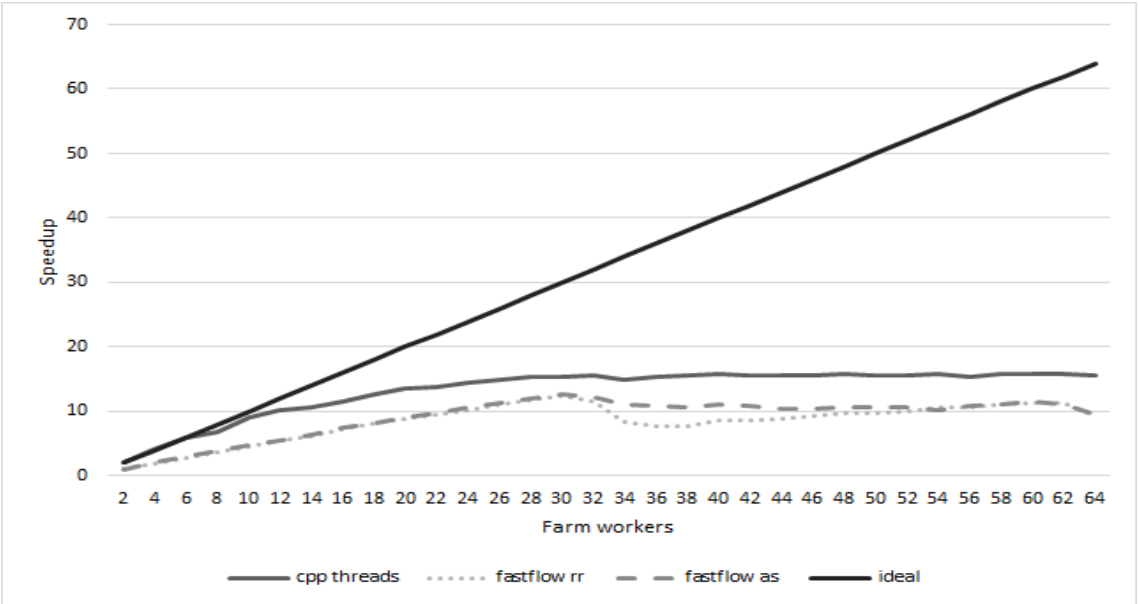

Figure 6: Speedup for the farm architecture with a full high-definition video and kernel size equal to 11.

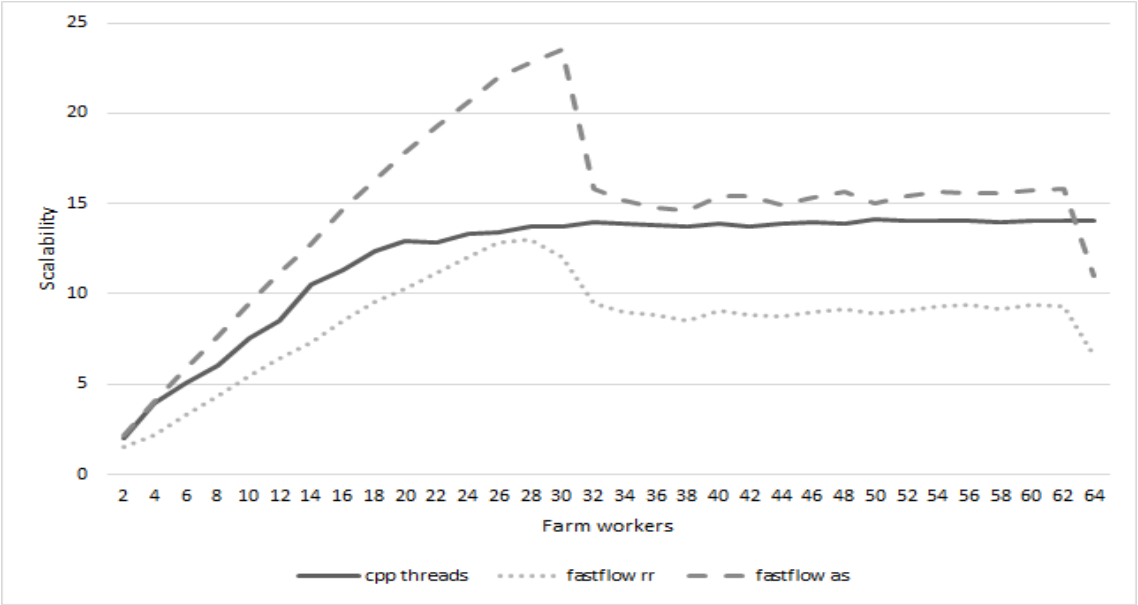

Figure 7: Scalability for the pipe architecture with a full high-definition video and kernel size equal to 11.

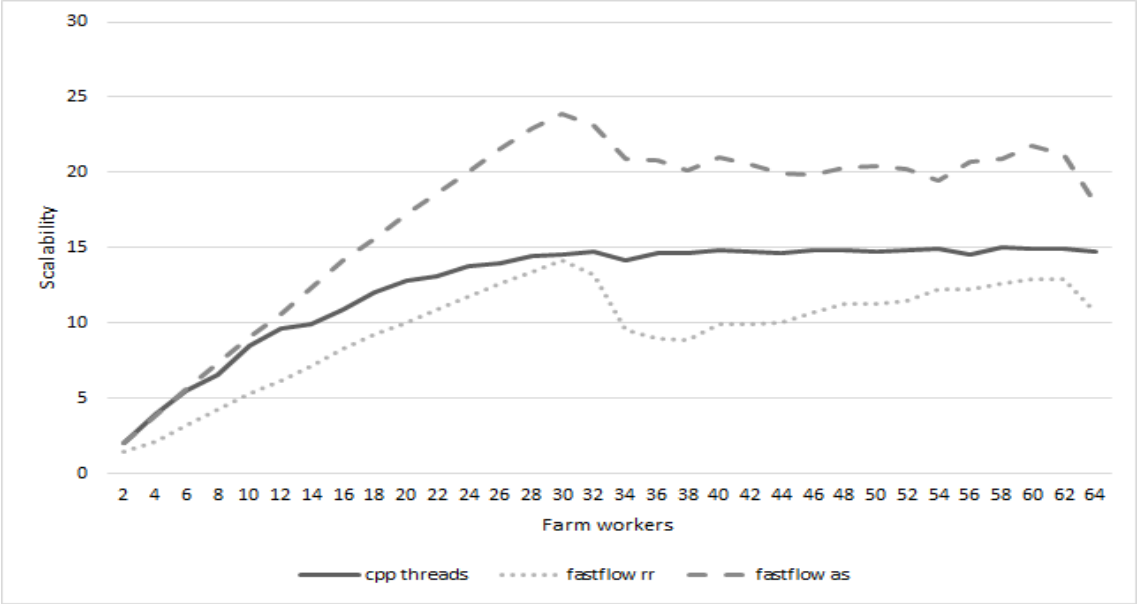

Figure 8: Scalability for the farm architecture with a FHD video and kernel size equal to 11.

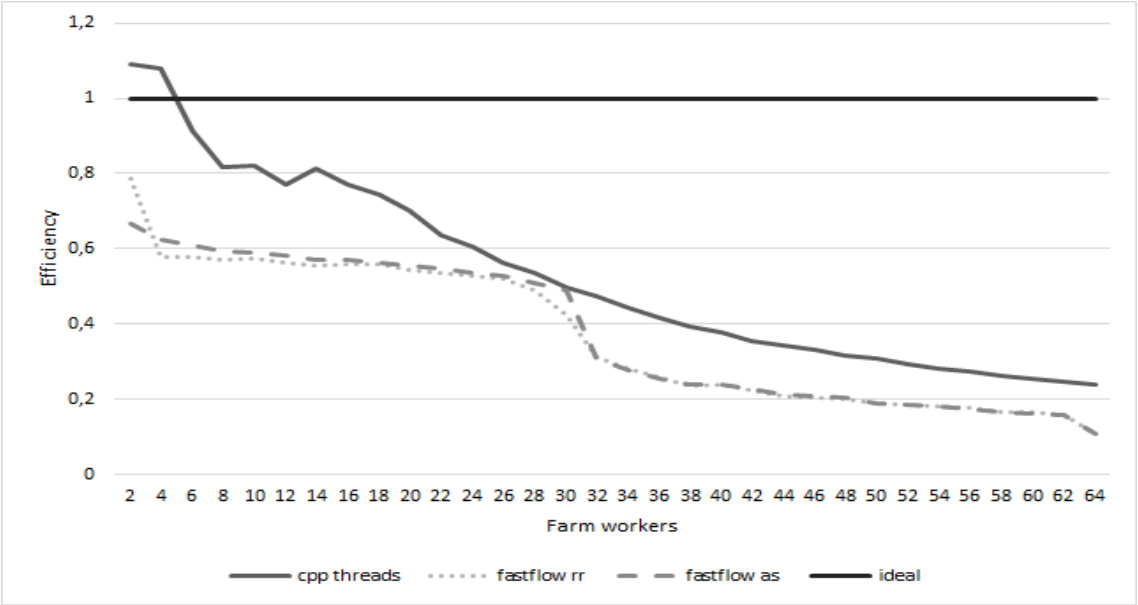

Figure 9: Efficiency for the pipe architecture with a full high-definition video and kernel size equal to 11.

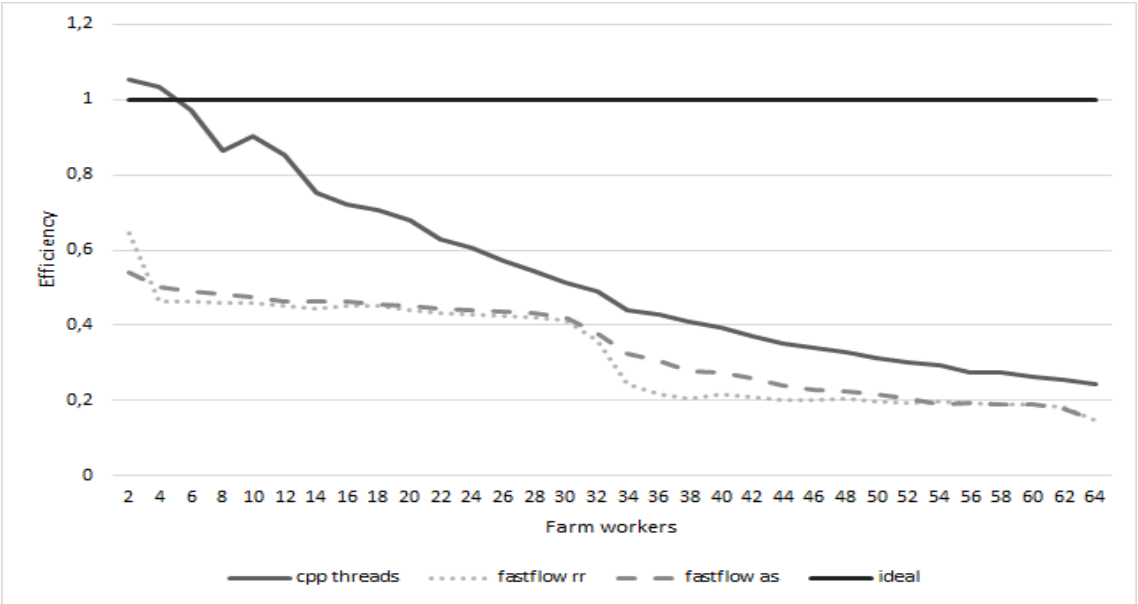

Figure 10: Efficiency for the farm architecture with a full high-definition video and kernel size equal to 11.

- The `build` folder contains all the files that are generated by the `cmake` utility.
- The `data` folder contains the three videos considered.
- The `lib` folder contains the FastFlow repository (*master* branch).
- The `result` folder contains the outputs of the tests executed by the `test.sh` script.
- The `source` folder contains the following `.cpp` files:
  - `ConcurrentDeque`: is a class that implements a thread safe `deque`.
  - `ff_nodes_implementations`: contains all the following subclasses of `ff_node_t` that are used for the FastFlow implementations: `Loader`, `FullWorker`, `Padder`, `Greyscaler`, `ConvolveDetectWorker`.
  - `Utimer`: a slightly changed version of the original `Utimer` class.
  - `VideoMotionDetection`: is a class that provides functions that encapsulate the main phases of the application.
  - `sequential`: contains the sequential solution used in the preliminary analysis.
  - `pipe` and `farm`: both files contain the `C++` threads implementation and the FastFlow round-robin and autoscheduling versions of the pipe and farm solutions.

5 Project instructions

To build the project issue the `make build` command. To setup FastFlow issue the `make ff` command and then compile with `make compile`. The project requires at least `C++ 14` (since the FastFlow implementations use the function `make_unique`) and openCV.

To execute the tests execute the `test.sh` script passing as first argument a resolution between `SD`, `HD` or `FHD` and as second argument a solution between `0` (sequential), `1` (farm) `2` (pipe) or simply issue the different `make test` commands.