Video Motion Detector

LEONARDO DA POZZO
Università di Pisa

A.A. 2021/2022

1 Local Machine Details

Whenever the text contains reference to a "local machine", it refers to a Thinkpad T480 with an Intel(R) Core(TM) i5-8350U quad core CPU @ 1.70GHz with two-way hyperthreading and 8GB of installed RAM. See Figure 1 for details.

2 Compiling and Running

To compile the code just step into the main directory and run *make* (you might see warnings coming from the OpenCV library headers). For usage instructions, run the executable with no parameters. Note that the "samples" folder contains a few videos to perform tests on, if necessary.

3 Roadmap

We will start by analyzing the problem and considering a sequential implementation. Once such implementation is in place, we will move onto the measurement of execution times of the various steps in order to understand which parts of the algorithm would benefit the most from parallelization. Based on our measurements, we shall consider various possible approaches, weighing their pros and cons, and describe two implementations: one using C++ threads and one using the FastFlow parallel programming framework.

4 Sequential Implementation

In this section we will consider the sequential implementation of the algorithm and look at some performance figures.

4.1 Frame Processing Steps

Each frame goes through three processing steps:

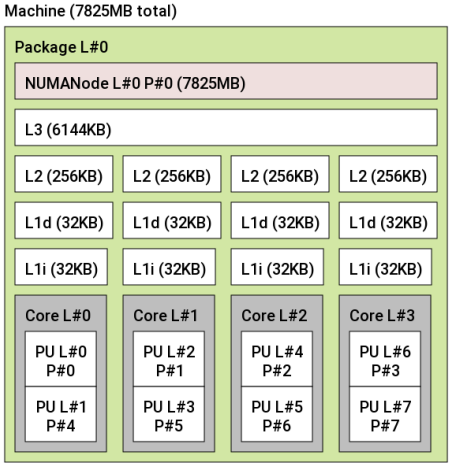

Figure 1: Results of the lstopo command on the local machine.

1. Turning to grayscale: in order to turn a frame to grayscale we create a new matrix of values computed as the average of the R, G, B levels of the corresponding pixel in the original image.

2. Blurring: blurring is done through a process called convolution. This consists in taking a small matrix, called kernel, and filling it with values, called weights. The kernel is then "slid" over the image we want to process, as if we were keeping the center of the kernel over the current pixel being processed. The new value of said pixel is computed by taking the (weighted) average of the surrounding pixels' values multiplied by the corresponding weight on the kernel.

3. Comparing: each frame is compared against the background by checking if more than a certain amount of pixels differ between the two frames. In order to account for noise, we consider two pixels to be different only if their value differs by more than a certain amount.

4.2 Pseudocode

$movementFrames \leftarrow 0$
while $!video.empty$ **do**
    $frame << video$
    $frame.\text{TOGRAYSCALE}()$
    $frame.\text{SMOOTH}()$
    if $frame.\text{COMPARE}(background)$ **then**
        $movementFrames \leftarrow movementFrames + 1$
    end if
end while

	Gray	Blur	Check	Total
1	10.196	37.429	0.943	15695.46
2	10.292	37.444	0.944	15698.01
3	10.363	37.453	0.943	15723.044
4	10.398	37.518	0.944	15770.039
5	10.487	37.637	0.943	15840.37
6	10.196	37.429	0.943	15695.46
7	10.292	37.444	0.944	15698.01
8	10.363	37.453	0.943	15723.044
9	10.398	37.518	0.944	15770.039
10	10.487	37.637	0.943	15840.37
Avg	10.3472	37.4962	0.9434	15745.3846

Figure 2: Average results of test runs of the sequential algorithm performed on the local machine. Times are measured in milliseconds.

4.3 Sequential Performance Evaluation

To start, we consider the results of 10 tests in which we ran the sequential algorithm on a 1920x1080 30 fps 10-seconds long video. Figure 2 shows the average time spent on each of the three processing phases for a single frame, along with the total time spent processing the whole video.

Ranking the algorithm's phases by execution time we get:

1. blurring, 37ms;
2. turning to grayscale, 10ms;
3. comparing against background, 1ms.

As we can also from Figure 3, the most computationally intensive operation is blurring the frame, followed by the application of the grayscale filter. Comparing the current frame and the background requires a negligible amount of time.

4.4 Potential Parallel Approaches

Given what we have seen in the previous subsection, we can consider two main approaches:

- pipe(map(toGrayScale), map(blur), seq(check)): this approach consists in composing the three stages of the algorithm into a pipe and spreading the computation of the two most intensive stages among a number of workers. The introduction of map allows us to reduce the latency of each phase, thus increasing the throughput of the pipe. Care should be taken in the distribution of workers among the two maps in order for the two to reach similar latencies, as the one with the largest latency will act as a bottleneck for the pipe.
- farm(comp(seq(toGrayScale), seq(blur), seq(check)): while the previous approach split the computation of each frame among threads, here we

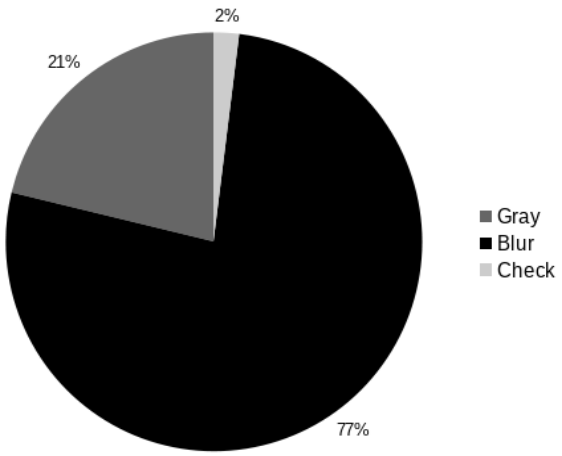

Figure 3: Average percentage of time spent in each computation phase for a single frame.

have each frame being completely processed by a single thread. This is the approach chosen for this project.

5 Motivation Behind the Chosen Approach

In the end, the choice fell on the implementation of a single farm, with frames being completely processed in a single go by a thread. The motivations behind this choice, beside the obvious ease of implementation, are several:

- We don't care about frame ordering: one of the appeals of the pipeline is its ability to parallelize computations over a stream while maintaining the order of its input elements. In our case, this in unimportant, as we just need to process all frames in any order as fast as possible and output a single result (the number of frames in which movement was detected).
- Throughput is not our main concern: the first approach also grants increased throughput, as each frame is computed faster. Similarly to frame ordering, this would be critical if we had to stream the resulting video back to the user, but this is not the case. What we want to minimize is the overall completion time.
- Caching: frames are stored as 2D arrays, and as such their pixels occupy contiguous areas of memory. By avoiding having multiple threads working on the same frame (something which happens in the first approach) we prevent false sharing, which could arise when two threads are working on different pixels which reside in the same cache line.

- Task granularity: blurring a 1920x1080 frame sequentially took, on average, 37 ms, which is a bit on the short side to consider parallelizing it with a map.

6 Parallel Implementation

Here we take into consideration two parallel implementations: one using the C++ built-in thread support and one using the FastFlow parallel programming framework.

6.1 C++ Threads

The first implementation we describe is based on C++ threads. The algorithm starts with the main thread reading frames from the input stream: the first one is turned to grayscale and blurred in order to be used as backgroud, while the rest are packaged in $std :: packaged\_task$ objects along with a function, $processFrame(frame)$, which encloses the frame-processing logic and returns a boolean whose value depends on whether movement was detected or not. Right after creation, packaged tasks are passed to a thread pool through the appropriate $ThreadPool :: insertTask(task)$ method, which inserts the provided task in the pool's queue and wakes up a worker. Once all frames have been packaged and delivered to the thread pool, the main thread tries to get the results through the $std :: future$ objects associated with the tasks, counting the number of frames in which movement was detected.

In the threadpool, auto scheduling is used: threads autonomously fetch a new task from the queue as soon as they have nothing to do. This ensure a fair distribution of the workload, but an eye should be kept on contention problems in the case of large amounts of threads, as the thread pool has a single queue that is locked every time a new task is added to it or a worker extracts a task from it. Arguably, much of this contention could be alleviated by using chunk scheduling, in which each worker has its own queue, with the main thread spreading the frames equally among the workers, accessing each queue exactly one time. This way there would be no necessity to lock the queues, but the worker threads would have to wait until the main thread has read, packaged and delivered the frame block to their queue before they can start working, without mentioning the fact that some threads may finish their assigned load sooner than the others, forcing them to wait.

7 FastFlow

The second implementation exploits the FastFlow parallel programming framework. The basic idea behind the skeleton is similar: we distribute frames in a farm in order to process them in parallel. Some notes on this approach:

- The farm has a dedicated emitter thread which also acts as collector, along with an amount of worker threads equal to the requested parallelism degree. This ends up forming a two-stage pipeline with feedback, allowing us to use one less thread with respect to a farm equipped with a collector. This was done due to the trivial nature of the collector's job in this particular project.
- Like in the first implementation, an auto-scheduling approach is used, this time handled directly by the FF library.

The roles of the two stages can be described as follows:

1. Emitter: this stage starts by opening the video stream and processing the background frame. Once the background is ready, the emitter proceeds to read frames from the stream and to package them in a struct along with a constant reference to the background. Each struct is sent to the workers through the $ff :: ff\_send\_out(task)$ method. Once all tasks are sent, the emitter starts processing the results sent back by the workers.
2. Worker: once a worker receives one of the aforementioned structs, it processes the frame and compares it to the background, passing back to the emitter stage a boolean which is true only if movement was detected.

8 Performance Figures

In the following pages we will look at some performance figures for both approaches. Namely, we will take a look at the execution time and the speedup both on the local and on the remote machine. Figure 4 and Figure 5 show the results of tests executed on the local machines, in which we can see that the two implementations are comparable. Similarly, in Figure 6 and Figure 7 we notice that a something similar happens on the remote machine, with a few outliers which we can probably chalk up to the fact that the remote machine is shared and may have varying loads at different times. These tests were performed on the same 1920x1080 30fps 10 seconds long video mentioned previously.

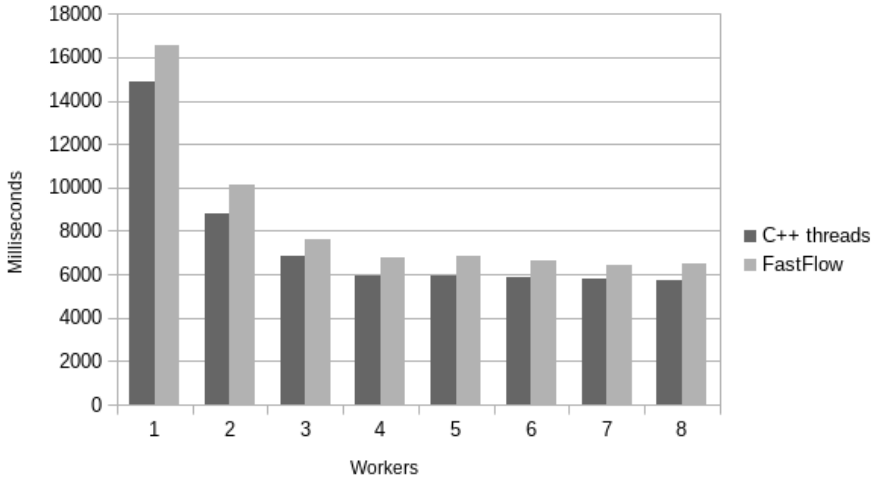

Figure 4: Average execution time on the local machine.

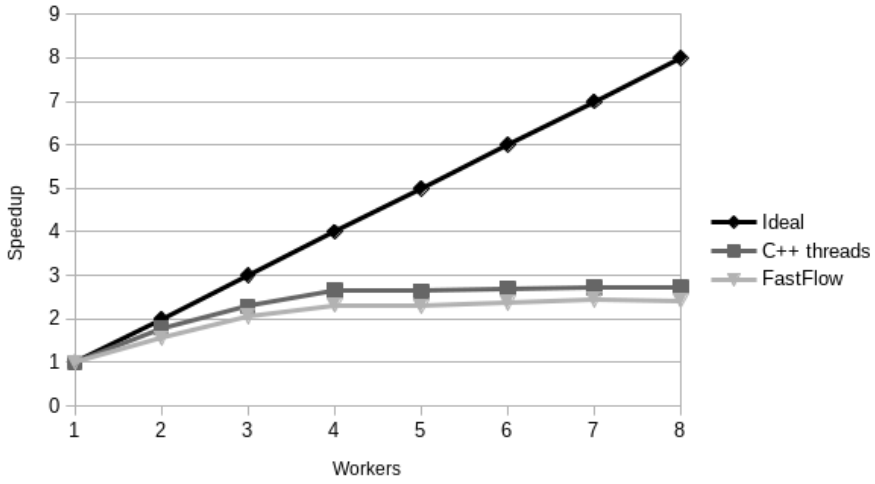

Figure 5: Average speedup on the local machine.

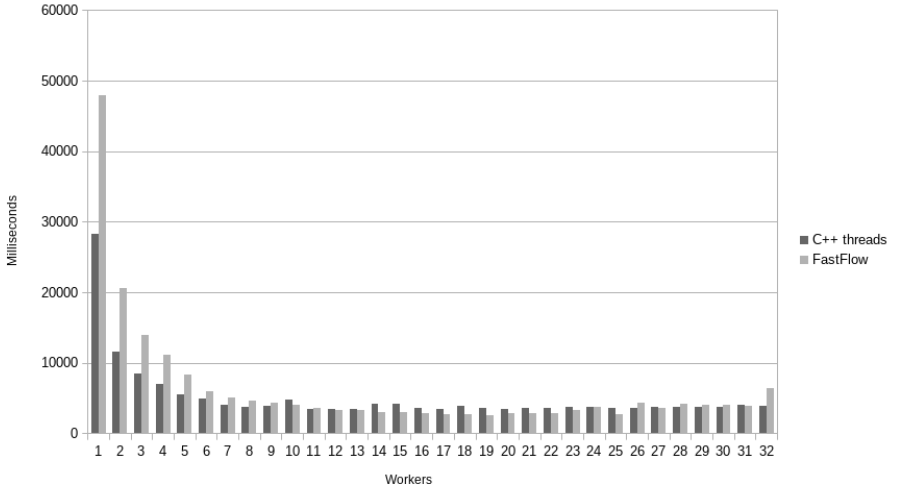

Figure 6: Average execution time on the remote machine.

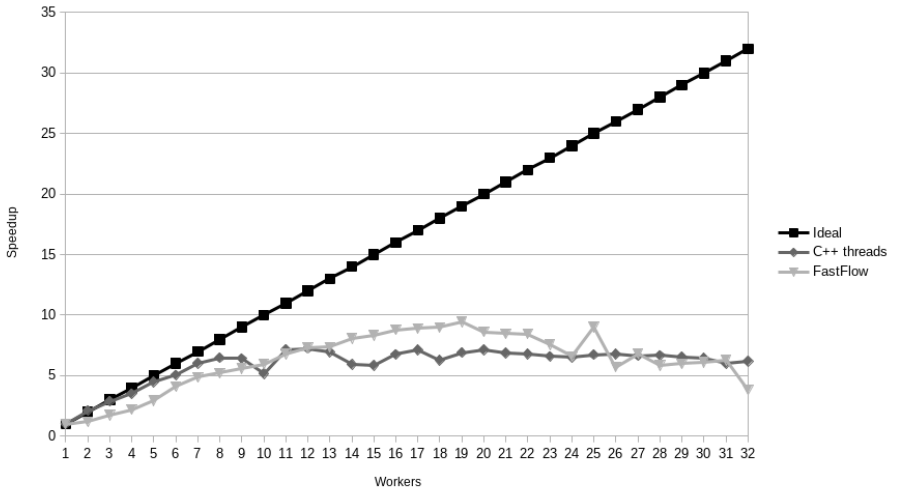

Figure 7: Average speedup on the remote machine.

9 Problems and Potential Improvements

As we saw in the previous section, the program fails to fully exploit the available resources in both implementations, reaching a peak speedup of about 2.8 on the local machine and 12.0 on the remote machine. There is certainly room for improvement. Below we mention some of the most glaring problems:

- (C++ Threads) Thread pool queue contention: having many threads accessing the same queue can cause slowdowns due to constant locking-unlocking and contention. A solution could be a change in the scheduling policy or an increase in the granularity of the tasks in order to reduce the number of queue accesses. Such increase could be obtained by passing bundles of frames to threads, instead of individual frames.

- (Both) Both implementations seem to struggle with exploiting the four additional logical cores offered by the local machine's hyperthreading capability, perhaps due to caching issues, as two logical cores on the same physical core share all cache levels.

- (Both) If the video has less frames than the number of available cores, resources will end up being wasted, as some of them will have no work to do. This would especially be a problem in case of very high resolution videos, as the cores processing the few frames would have to work with a whole lot of pixels all by themselves, while other cores sit idle. A solution to this could be a combination of the two approaches mentioned in Section 4.4: we introduce both a farm of workers, each processing an individual frame, and a map over each frame. Depending on the video resolution we could distribute the resources among these two: more workers in the farm for long, low resolution videos and more workers in the map for short, high resolution videos. Alternatively, we could just keep the two implementations separated and decide to use one over the other based on an analysis of the video's metadata.