Video motion detect

Giuseppe Lombardi
Master Degree in Computer Science.
g.lombardi11@studenti.unipi.it.
SPM, Academic Year: 2021/2022

September 4, 2022

Contents

1 Sequential pattern

The pattern of sequential architecture consists of four independent and consecutive steps: `read`, `RGBtoGrey`, `smoothing`, `detection` and `collect`. Figure 1 shows the sequential pattern. Each frame of the video source is read, transformed from RGB to greyscale, smoothed and finally compared with the background (the first video frame), producing a `detected` flag. Finally, the program collects the results by increasing the variable `totalDiff` that expresses the number of frames with motion detected. As illustrated in figure 1, to perform the `RGBtoGrey` and `smoothing` operations, it is necessary to create two auxiliary variables (`grey` and `smoothed`) in which to save the greyscale and smoothed frame respectively. The program initializes these two variables via heap allocation at the beginning of the frame stream (by command `new Mat`) and uses them for each frame of the video source.

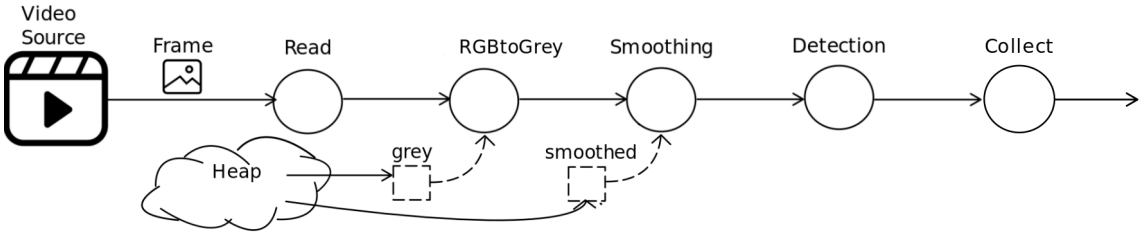

Figure 1: Standard sequential pattern.

1.1 Sequential optimization

First of all, the aim is to optimise the sequential pattern and minimise the sequential execution time T_{seq}. The most time-consuming operation is `smoothing`. At the same time, the fastest function is detection. Figure 2 shows the average time of each operation performed on the total number of frames processed. Sequential pattern optimisation aims to decrease the sequential computational cost and enable the auto-vectorization [1] of code loops.

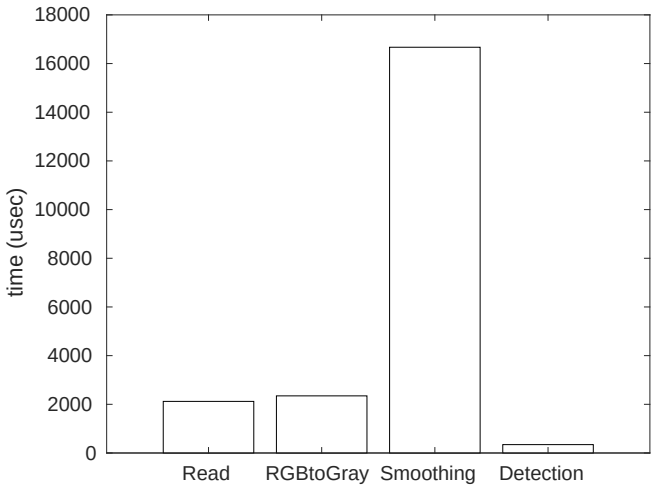

Figure 2: Latency of each operation.

In the design phase of the sequential architecture, I considered several alternatives to optimise computation time. Adding padding to the auxiliary frame grey decreases the time cost of `smoothing` while slightly increasing the temporal cost of `RGBtoGrey`. Indeed, padding allows the use of exact neighbourhood definition

for each pixel of the greyscale frame. Not using padding means considering specific neighbourhoods for pixels at the frame's edges., i.e. checking the position of each pixel. This control flow does not allow the vectorisation of smoothing loop.

Furthermore, in the final implementation (Figure 4), the sequence of functions {`RGBtoGrey`, `smoothing`, `detection`} was replaced with a single function `composition` to simplify the pattern, reduce the computational time and facilitate the construction of the parallel architecture. The `composition` function executes the first for loop, where it computes the grey frame storing it in a bigger frame with zero-padded at the edges, and the second in which, at the same time, it calculates the smoothed pixel and compares it with the background pixel at the same position.

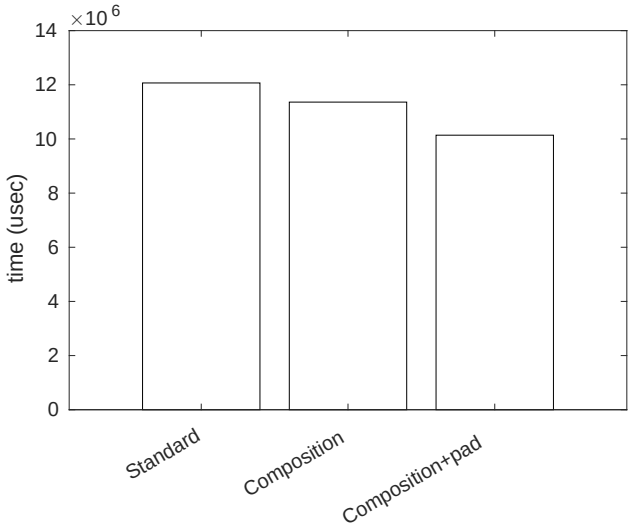

Figure 3: Sequential pattern optimization. The main step choosen for the optimized sequential pattern is the `composition` with padding .

A further reason for using the composition function is that, except for the backgound frame, the construction of greyscale and smoothed frames is not necessary to achieve the result required by the task. In fact, as Figure 4 shows, the composition function uses the auxiliary padded frame aux allocated in the stack memory to store the greyscale pixels temporarily. Then the function smooths each internal pixel (excluding zeros at the edges of the frame), compares it with the pixel at the same background frame position and collects the detection flag.

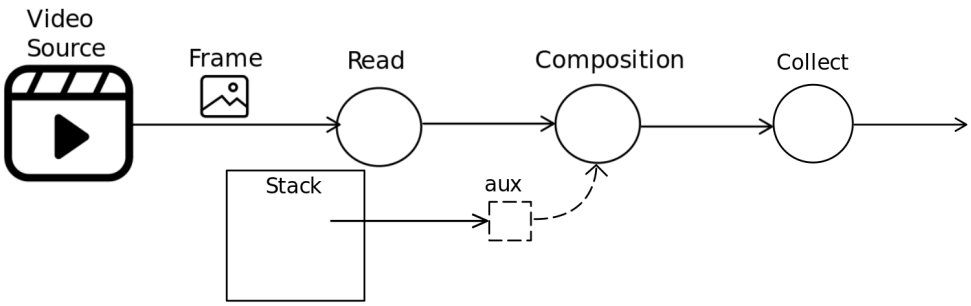

Figure 4: Optimized sequential patterns.

2 Parallel architecture

The parallelisation of the *video motion detect* was done via *stream parallelism*. Given a video source of m frames and background f_0, the parallel architecture receives a stream of images $f_1, \ldots, f_{m-1}$ available at distinct times $t_1 < \cdots < t_m$ where $\Delta_i = t_{i+1} - t_i$ is the execution time for reading frame f_{i+1}, i.e., $T_{\texttt{Read}(f_{i+1})}$. Stream parallel computations is modelled creating a task for each one of the frame to detect. The aim of parallel architecture is to reduce the *completion time* of the whole stream of tasks.

The starting parallel pattern is a *pipeline* of the three stages `read`, `composition`, `collect`. The aim is to reduce the service time of the pipeline and then reduce the service time of stage `composition` (the most time-consuming stage). A farm can reduce the service time of stage `composition`. Each farm worker performs the function `composition` at a given frame. Similarly, a thread pool with a producer and several consumers that process the image and increment the external variable `totalDiff`, is helpful.

The detected flags are collected by incrementing an integer variable, using the *sum* operation. This operation is distributive and associative, so the final result does not depend on the particular order in which the images are processed and the results collected. These properties allow the use of a farm or a thread pool.

2.1 Native Parallel

The native parallel architecture exploits a *thread pool*. The architecture has a single *producer* (main thread) that handles the stream of frames and a thread pool with `nw` *consumers* that execute the work (Figure 5). The producer reads each frame, creates the respective task and submits it in a centralised, concurrent queue. When free, each consumer fetches a task from the queue and executes it, incrementing the shared and non-thread-local variable `totalDiff`. The consumer threads are created once before the parallel computation starts and destroyed after the parallel computation terminates, following the RAII programming technique.

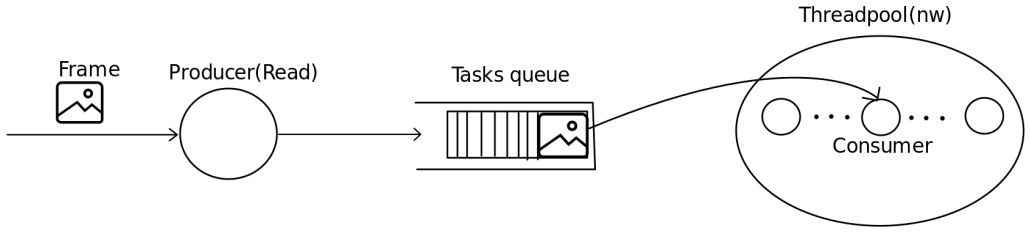

Figure 5: Native thread pool.

The task queue is managed via a global *mutex* and *conditional variable*. A *unique lock* protects operations on the *shared queue*. Specifically, when the pattern submits a task, it notifies one of the waiting threads. Within the main loop, each thread (if active) takes the task from the queue and executes it. A boolean variable `stop` manages the termination of the thread pool, initially set to false. If the stack is empty or the stop variable is false, the thread waits for a possible notify one from a submit.

The variable `totalDiff`, shared between threads, is declared as an `atomic<int>`. The task T_i is obtained by a *bind* of the `work` function and the frame f_i. The `work` function performs the following actions:

- it detects the movement of the frame;
- it decrements the atomic variable; `toCompute` (representing the number of the remaining tasks to be executed);
- If `toCompute` equals zero, it stops the thread pool, setting the stop variable to true and notifying all threads.

A possible alternative is using *packaged tasks* that allow us to receive a future so that we know when tasks end. The advantage of this alternative is that the producer can submit new tasks for execution at any time. The disadvantage of this architecture is the need to use a vector of packaged tasks equal in size to the number of frames to be processed, thus requiring high memory usage.

2.2 FastFlow

The parallel pattern of FastFlow is a *farm* of the emitter-worker-collector type (Figure 6). The emitter executes the read function, creates the respective task and sends it to the worker farm. Each worker executes the task assigned to it by the emitter. Finally, the collector collects the results produced by the workers in the `totalDiff` variable. Each task is obtained as an object of a wrapper class of the related frame. The task `T` has the attributes `T.frame` and `T.detected`, in which the worker will save the result of the composition function. The wrapper class facilitates the flow of tasks within the pattern. By executing a generic task, the worker updates the detected attribute of the object without having to modify shared variables.

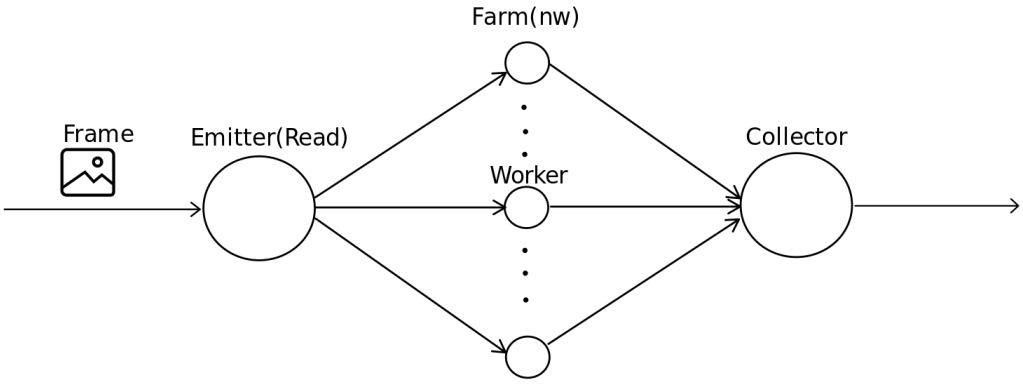

Figure 6: FastFlow farm.

The alternative patterns considered are the following:

- a pipeline made up of emitter, farm and collector; in this case, we keep the farm's default emitter and collector.
- master/worker pattern, with the implementation of an emitter/collector, through the `wrap_around()` function.

The first alternative was discarded because it has two additional nodes and requires many specific threads. The second, on the other hand, was discarded as the latency of the emitter is not low, and it is preferable to use a specific node.

3 Experiments

All times reported in the figures correspond to the average of ten trials.

Figure 7 shows the sequential time and the *parallel time* as the number of workers of the native and FastFlow architecture varies.

The parallel times of the two architectures with a single worker are slightly higher than the sequential one. As the number of workers increases, the parallel time decreases. Initially, the two curves are steep, but as the number of workers increases, the curves flatten out. This trend is mainly due to the `overhead` times, shown in figure 8. We have the best parallel times for native architecture with ten workers and up. The best times for the FastFlow architecture are obtained for a narrower range of workers, from 16 to 30.

It can be seen from Figure 7 that the parallel time of the FastFlow architecture peaks with 31 and 32 workers. This is because some threads are in concurrency, as the architecture must reserve one thread for the emitter and one for the collector.

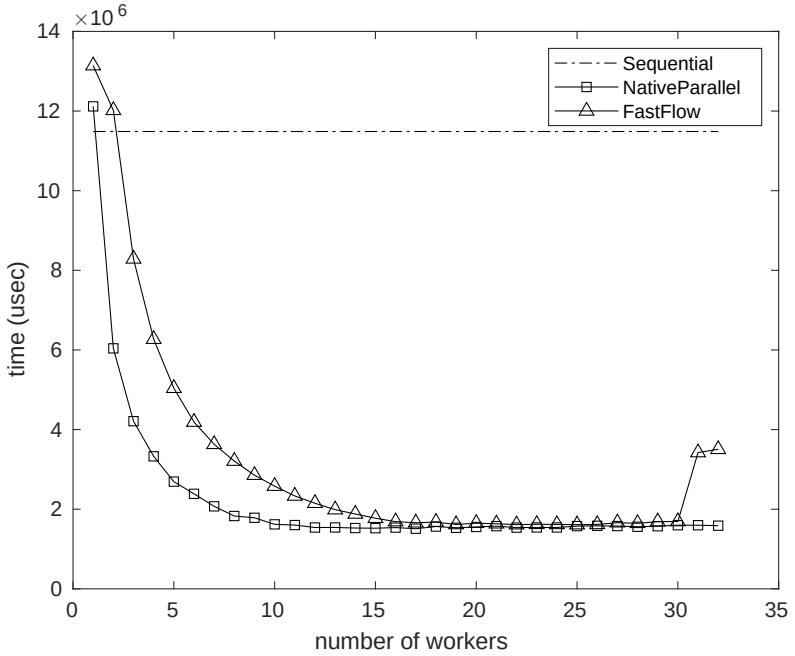

Figure 7: Parallel time.

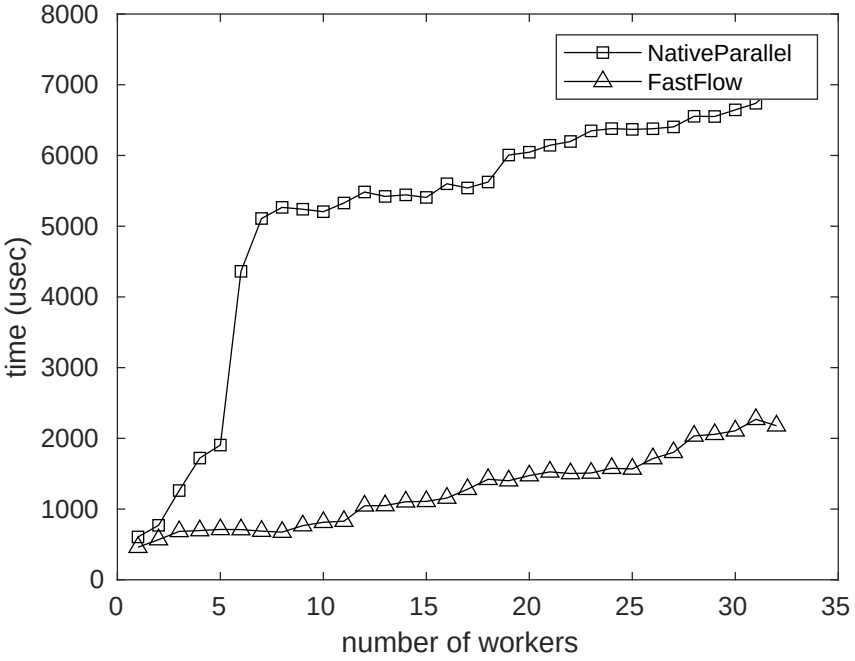

Figure 8: Overhead time

The parallel time of the native architecture initially decreases more rapidly. This is more visible in Figure 9, which shows the *scalability* of the respective architectures. The native architecture initially scales better than FastFlow but flattens out sooner.

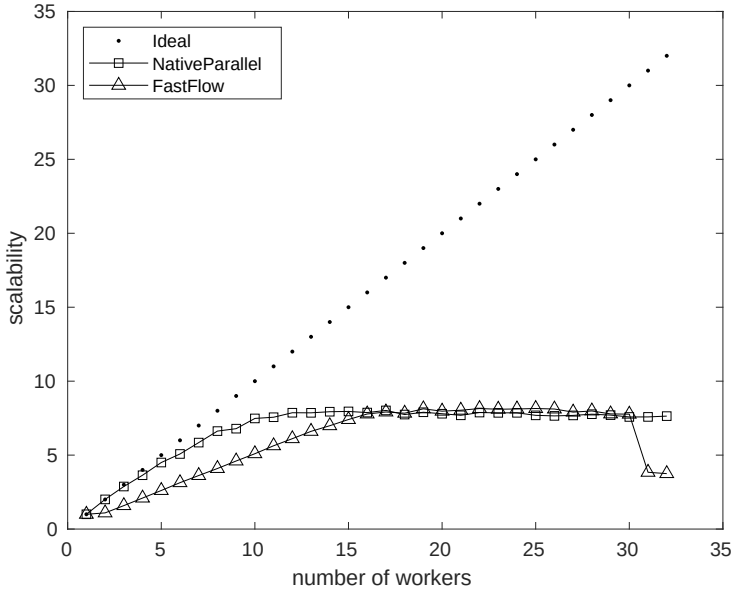

Figure 9: Scalability.

Figure 10 shows the *efficiency* of the two architectures as the number of workers varies. For a small number of workers or workers equal to 31 or 32, the efficiency of the native architecture is better than the FastFlow architecture. On the other hand, when the number of workers is suitable for the task and the machine used, the efficiency of the two architectures is very similar.

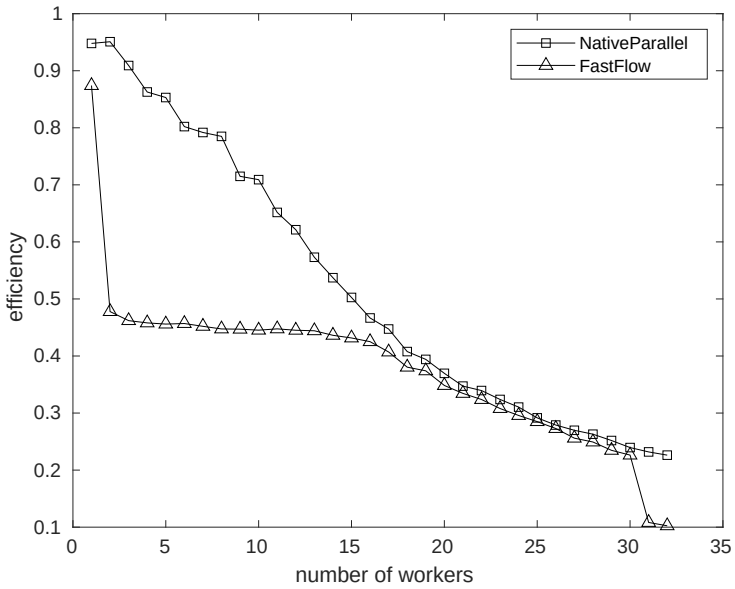

Figure 10: Efficiency.

4 Code instructions

In the main folder, the script `compiles.sh` re-compiles and executes the `main.cpp` file. The source code, included in the `main.cpp`, is located in the `./src` subdirectory and is subdivided as follows:

- `VideoDetection.cpp` contains the data structures and operations to load the video, background, transform frames and compare them with the background to detect motion.
- `Sequential.cpp` contains the sequential pattern code.
- `NativeParallel.cpp` contains the ThreadPool class and the code that executes the native parallel pattern.
- `FastFlow.cpp` contains FastFlow's operational node data structures and the code that executes FastFlow's farm pattern.

To re-compile the code execute the command:

```
./compile.sh
```

To run the code execute the command:

```
./main version k out [nw]
```

with the parameters:

- `version` indicates the version of the pattern to be executed [`0` sequential, `1` native parallel, `2` fastflow];
- `k` indicates the percentage in the range [0,1] of frame pixels required to detect movement;
- `out` indicates the type of output [`0` number of frames with motion detected, `1` elapsed time, `2` other statistics];

- `nw` (optional) indicates the number of workers for parallel versions. By default, the program uses many workers for parallel versions equal to the hardware concurrency.

In particular, by setting the `out` parameter equal to `2`, in the case of sequential, the program prints for each operation (`RGBtoGrey`, `RGBtoGrey_pad`, `smoothing`, `smoothing_pad`,`composition`,`composition_pad`) the average of the times over the total number of frames. In contrast, in the case of parallel versions, by setting `out` equal to `2`, the programme prints the overhead time.

In addition, in the main folder, the files `time_par.sh` and `overhead.sh` prints out respectively the execution and the overhead times of the chosen parallel pattern as the number of workers in the set {1,...,32} changes. Both scripts compute the times as an average over `trials` run, where trials is an editable variable within the script.

References

[1] *Auto-vectorization in GCC*. URL: https://gcc.gnu.org/projects/tree-ssa/vectorization.html (visited on 08/31/2022).