Parallel Programming – TP5: Dataflow for Motion Application

Short introduction   In this session, we will work on a streaming application that detects and tracks moving objects from a video sequence. Contrary to the previous sessions, we will not use EASYPAP this time. The later is not adapted for streaming applications. A working streaming application will be given to you and you will have to use AFF3CT to implement the MOTION application through an explicit dataflow representation.

Very important, about the submission of your work   At the end of this session you will have to upload the following files on Moodle: 1) a zip of the `src` folder and 2) a zip of the `include` folder. After that you will have 2 weeks (until October, 29) to complete your work and update your first submission. You have to work in group of two people but each of you will have to upload the file on Moodle. Finally, please write your name plus the name of your pair at the top of all these files.

1   Appetizer

First you need to clone the repository of the MOTION project:

```
git clone --recursive https://gitlab.lip6.fr/cassagnea/motion-se.git
```

The MOTION project uses CMake in order to generate a Makefile: follow the README instructions to compile the code.
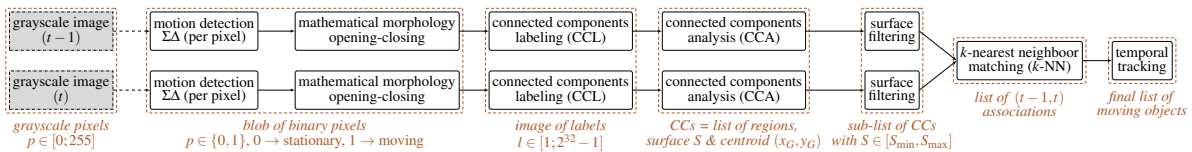

Figure 1: MOTION detection and tracking processing graph.

Fig. 1 presents the different algorithms used to detect moving objects and to track them over time. To make it work, two strong assumptions are made: 1) the camera is fixed, 2) the light intensity is constant over time. First, an image is read from a camera (or a video sequence) and then it is converted in a grayscale image. Then, the $\Sigma\Delta$ algorithm is triggered. This algorithm is able to detect if a pixel is moving over time. It returns a binary image, if a pixel value is 0, then it means that it is not moving. Else, if a pixel value is 1, then it means that it is moving. After that, morphology algorithms are applied[1]. This is a pre-processing to regroup moving pixels together and eliminate isolated pixels. Then, from a binary image, a connected components labeling algorithm is performed. The later, gives the same label to a group of pixel that are connected to each other. CCL returns an image of labels where $l = 0$ means no object and $l > 0$ means a moving object. From this image of label, some components are extracted (CCA): for each object the center of mass (x_G, y_G), the bounding box $([x_{min}, x_{max}, y_{min}, y_{max}])$ and the surface S are extracted. Depending on their surface, the objects are filtered ($S_{min} < S < S_{max}$).

From two images at $t-1$ and t, a matching algorithm determines which objects are the same in the two different images (mainly according to their distance). At the end, the identified object are tracked to have a constant identifier over time.

This graph of tasks is then repeated until the video sequence is over. It is not mandatory to understand perfectly each algorithm. The purpose of this session is to work on a streaming application, representative of a real application, and to perform optimizations at the tasks graph level.

In this graph, two tasks cannot be replicated. The per pixel motion algorithm requires its previous output to compute the current binary image. It detects intensity variations over time. It is almost the same for the tracking algorithm that maintains a list of tracks that are updated according to the last frame.

---

[1]Mathematical morphology: `https://en.wikipedia.org/wiki/Mathematical_morphology`

1.1 Run MOTION

To run the code you will need some input videos. You can download a videos collection on Moodle (see the "Artifacts" section) from this web link: `http://www.potionmagic.eu/~adrien/data/traffic.zip`. First, unzip the `traffic.zip` and from the `build` directory run the code with the following command:

```
./bin/motion2 --vid-in-path ./traffic/1080p_day_street_top_view_snow.mp4 \
              --flt-s-min 2000 --knn-d 50 --trk-obj-min 5 --vid-out-play --vid-out-id
```

You should see a window with a top view of a highway and some moving cars (see Fig. 2) and you should see green bounding boxes around the cars.

Figure 2: MOTION screenshot (with `--vid-out-play --vid-out-id` parameters).

1.2 Architecture of the Project

MOTION is mainly a C-style project but it is compiled in C++ to use AFF3CT. The sources are located in the `src` folder, and there are 3 sub-folders:

- `common`: contains implementations of the processing tasks,
- `main`: contains source files that correspond to a final binary executable,
- `wrapper`: contains C++ files to wrap the C-style processing functions into AFF3CT modules and tasks.

The headers are located in the `include` folder. Inside there are two sub-folders: `c/motion` for the C-style headers and `cpp/motion` for the C++ headers.

2 From a Standard Description to a Dataflow Description

2.1 `motion2` Executables

We will convert the `motion2` main into a dataflow description (= AFF3CT modules and tasks). The `motion2` is located here `src/main/motion2.c`. This implementation is very close to the tasks graph presented in Fig. 1.

Work to do #1 Understand the code, run the `motion2` executable and play with the parameters (`-h` shows and describes the available parameters).

To help you in the task, we created an other main based on `motion2.c` and we converted some C functions into AFF3CT modules for you. See the `motion2_aff3ct.cpp` file.

Work to do #2 Understand the code, run the `motion2-aff3ct` executable and play with the parameters (`-h` shows and describes the available parameters). You can understand the code of `motion2_aff3ct.cpp` by comparing it with the C-style `motion2.c` code.

Work to do #3 Create new AFF3CT modules and tasks, each time you will create new `.cpp` and `.hpp` files in the `wrapper` folders. You will only declare input and output sockets (no forward sockets at this time):

1. `Sigma_delta`: with a `compute` task that will call the `sigma_delta_compute` function,

2. `Morpho`: with a `compute` task that will call the `morpho_compute_opening3` and `morpho_compute_closing3` functions,

3. `CCL`: with an `apply` task that will call the `CCL_LSL_apply` function,

4. `Features_CCA`: with an `extract` task that will call the `features_extract` function,

5. `Features_filter`: with an `filter` task that will call the `features_filter_surface` and `features_shrink_basic` functions,

6. `KNN`: with a `match` task that will call the `kNN_match` function,

7. `Tracking`: with a `perform` task that will call the `tracking_perform` function.

You will add the AFF3CT modules and tasks incrementally in the `motion2_aff3ct.cpp` file and you will test if their integration is working (you can compare the logs with a diff, see the Note #1 below). Have a look on how we did this for the other AFF3CT tasks that are given to you. You will follow the same philosophy: 1) bind the sockets to the buffers allocated in the main file and 2) call the `.exec()` method explicitly.

Note #1 `motion2` is our golden model. To compare the results of `motion2` and `motion2-aff3ct` you can generates the logs of `motion2` executable first:

```
./bin/motion2 --vid-in-path ./traffic/1080p_day_street_top_view_snow.mp4 \
  --vid-in-stop 20 --flt-s-min 2000 --knn-d 50 --trk-obj-min 5 --log-path logs_refs
```

Secondly, you can generate the logs of the `motion2-aff3ct` executable:

```
./bin/motion2-aff3ct --vid-in-path ./traffic/1080p_day_street_top_view_snow.mp4 \
 --vid-in-stop 20 --flt-s-min 2000 --knn-d 50 --trk-obj-min 5 --log-path logs_aff3ct
```

Finally you need to compare the logs together:

```
diff logs_refs logs_aff3ct
```

If the later command returns nothing, it means that `motion2` and `motion2-aff3ct` are equivalent (in term of features). This is good!

Work to do #4 At this point you should only have AFF3CT tasks that call their `.exec()` method explicitly. Now, the objective is to get rid of this. You will bind the tasks together and create a sequence. You should remove all the data allocations from the main file, AFF3CT will do it for you.

Note #2 In the `motion2` executable, some tasks are not executed in the first stream. To simplify, in the `motion2-aff3ct` executable, you will always execute all the tasks. This will change the results a little bit.

Note #3 You will need to use a Delayer module to keep the $t - 1$ image in memory. If not the image will we overwritten when calling the `video("generate")` task.

Note #4   To help you in the debugging, you can print the inner sequence graph with the `Sequence.export_dot` method.

Work to do #5   Before the sequence execution, you will enable the statistics of the task (`Sequence.get_modules` method). And after the sequence execution you will print them at the end (`tools::Stats::show` function). What do you see? Is it different than from the `motion2` executable? Explain.

Work to do #6   When you think it's necessary, create new tasks, postfixed with `_fwd`, that use forward socket instead of the combination of input and output sockets. Do it incrementally to validate that the application is still working. Can you see an improvement in the statistics of the tasks?

To be continue...