Sorbonne Université – SESI M2

MU5IN160 – Parallel Programming

Hands-on Session 6 – Multi-threading for Motion Application

Short introduction In this session, we will work on the Motion streaming application. In the last session, we focused on converting the Motion application through an explicit dataflow representation with the AFF3CT DSEL. Today, we will benefit from the code transformations we made before to parallelize the application over the CPU multi-core architecture.

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 November, 5) 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 files 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 update the repository of the Motion project:

```
git pull origin master && git submodule update --init --recursive
```

Then, you have to re-generate the Makefike and to re-compile a sub-part of the application:

```
cd build && cmake .. && make -j4
```

2 Motion Parallelization

In this section, we will focus on reducing the execution time by using different types of parallelism. We will mainly rely on the multi-core architecture (≈ multi-thread programming) of the CPU. To measure the execution time, we will not consider the video decoding time, the visualization time and the logs time.

Motion comes with the `--vid-in-buff` Motion parameter to hide the video decoding time: the frames are decoded and put into a buffer during the video initialization. This behavior is representative of a real embedded application, where there would be no video decoding. Instead, a camera would send frames at a fixed rate. Be careful, buffering frames can take a lot of memory (and time). Thus, we will limit this buffer to 100 frames.

Here is the command line to use for the measurements:

```
./bin/motion2 --vid-in-buff --vid-in-path ./traffic/1080p_day_street_top_view_snow.mp4 \
              --vid-in-stop 100 --flt-s-min 2000 --knn-d 50 --trk-obj-min 5
```

As a consequence, we will not take into account the whole execution time of the Motion application but only the sequence or the pipeline execution time. And, we will focus on increasing its throughput (= number of frames per second or FPS).

Of course for debugging and validation purpose, you will need to use the logs (`--log-path`) and the visualization (`--vid-out-play --vid-out-id`) parameters.

Work to do Run the `motion2` and `motion2-aff3ct` executables and note their throughput in FPS.

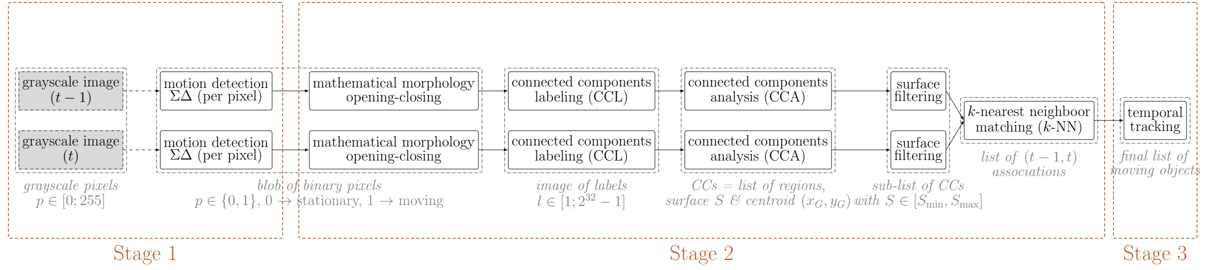


Figure 1: Motion detection and tracking processing graph. In gray and italic: the output of each processing. The graph is split into a 3-stage pipeline.

2.1 Pipeline Strategy

AFF3CT comes with a round-robin *producer/consumer* implementation to support the *pipeline* parallelism (and more specifically the synchronizations between the pipeline stages). This type of parallelism is well adapted to the streaming applications like Motion. It improves the throughput of the system.

We will instantiate the 3-stage pipeline given in Fig. 1. Other stages decomposition is possible but it will not lead to significant improvements (and it will not help for the next sections).

To instantiate a pipeline, we need to delimit its stages (= sub-parts of the graph). In AFF3CT, a stage is defined by 3 vectors of tasks (in the exact following order):

1. A list of the first tasks in the stage,
2. A list of the last tasks in the stage (these tasks are included in the stage),
3. A list of exceptions that represents the tasks that should not be included in the current stage (can be empty).

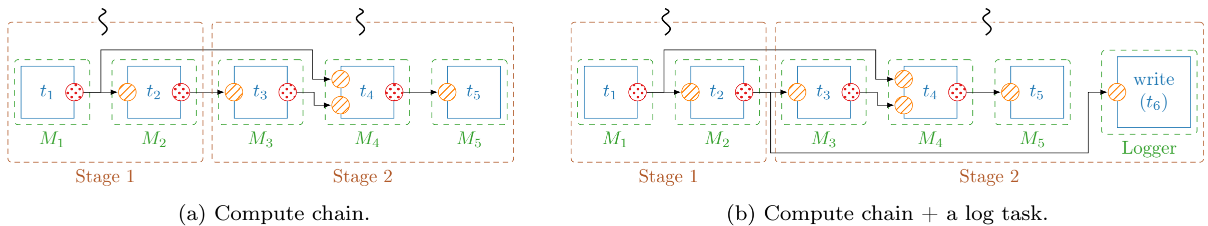
In other words, for each stage, AFF3CT will build the sub-graph from the first and last tasks and it will exclude the exception tasks. When building the sub-graph, if some of the tasks are bound to tasks that are in a previous stage, then AFF3CT will not automatically add these tasks to the current stage. **In this case, you need to explicitly add these tasks into the list of first tasks.**

Additionally, in order to facilitate the debugging process, the pipeline constructor requires the vectors of first and last tasks of the corresponding sequence. This way AFF3CT can compare if the stages match the initial sequence. If it does not match, AFF3CT will automatically return an error and will output two debug graphs (in fact, it will store these two files on the file system):

- `dbg_ref_sequence.dot`: the graph corresponding to the sequence (= without parallelization),
- `dbg_cur_pipeline.dot`: the graph corresponding to the pipeline (= with the stages).

Then, you can visually compare the two graphs and find the mismatch. Common mistakes are:

- A task is missing in the pipeline graph: this is often the case because this task needs to be explicitly declared in the list of first tasks,
- A same task is present in two (or more) different stages: this is generally because the task has been automatically added to a previous stage. In this case, using the exclusion list of tasks can solve the problem.



(a) Compute chain. (b) Compute chain + a log task.

Figure 2: Example of a chain decomposed into two pipeline stages.

```
// 1) creation of the module objects
M1 m1(); M2 m2(); M3 m3(); M4 m4(); M5 m5(); Logger log();

// 2) binding of the tasks
m2["t2::in" ] = m1["t1::out"];
m3["t3::in" ] = m2["t2::out"];
m4["t4::in1"] = m3["t1::out"];
m4["t4::in2"] = m3["t3::out"];
m5["t5::in" ] = m4["t4::out"];
m6["t6::in" ] = m5["t5::out"];
// conditionnal binding of the log task
if (is_log)
  log["write::in"] = m2["t2::out"];

// definition of the pipeline stages in the 'pip_stages' variable
std::vector<std::tuple<std::vector<runtime::Task*>,
                       std::vector<runtime::Task*>,
                       std::vector<runtime::Task*>>> pip_stages =
    { // pipeline stage 1
      std::make_tuple<std::vector<runtime::Task*>,
                      std::vector<runtime::Task*>,
                      std::vector<runtime::Task*>>(
        { &m1("t1"), },                                  // first tasks of stage 1
        { &m2("t2"), },                                  // last tasks of stage 1
        { /* no exclusion in this stage */ }             // tasks excluded from stage 1
      ),
      // pipeline stage 2
      std::make_tuple<std::vector<runtime::Task*>,
                      std::vector<runtime::Task*>,
                      std::vector<runtime::Task*>>(
        { &m3("t3"), &m4("t4"), },                       // first tasks of stage 2
        { /* last tasks will be automatically found */ }, // last tasks of stage 2
        { /* no exclusion in this stage */ } ),          // tasks excluded from stage 2

    /* if you'd had more stages, you could have continued their declaration here */
    };

// if there is the log task
if (is_log) {
    // in stage 1 ('pip_stages[0]'): add the log task at the end of the exclusion list ('get<2>')
    std::get<2>(pip_stages[0]).push_back(&log("write"));
    // in stage 2 ('pip_stages[1]'): add the log task at the end of the first tasks list ('get<0>')
    std::get<0>(pip_stages[1]).push_back(&log("write"));
}

// the first tasks of the sequence, used to check the pipeline stages
std::vector<runtime::Task*> seq_first_tasks = { &m1("t1") };

// declaration (= construction) of a pipeline object 'pip'
runtime::Pipeline pip(seq_first_tasks, pip_stages,
    {     1,          1 },  // number of threads per stage -> one thread per stage
    {         1,        },  // buffer size between stages -> size 1 between stage 1 and 2
    {       false,      },  // active waiting between stage 1 and stage 2 -> no
    { false,      false },  // enable pinnig -> no
    {   {0},        {1} }); // pinning to threads -> ignored because pinning is disabled

// pipeline execution, note that two different functions can be used for the stop condition
pip.exec({ [] (const std::vector<const int*>& statuses) { return false; },     // stage 1 stop
           [] (const std::vector<const int*>& statuses) { return false; }, }); // stage 2 stop
```

Source code 1: Source code corresponding to the 2-stage pipeline described in Fig. 2. Depending on the `is_log` value (`true` or `false`), the code will build either the Fig. 2a pipeline or the Fig. 2b pipeline.

Fig. 2 introduces a simple example with a 2-stage pipeline. The compute chain (= tasks that do not include logs and visualization) is given in Fig. 2a. Fig.2b shows a graph where we added a log task for debugging purpose. The main goal of this graphs is to help you to bootstrap in your work.
The corresponding source code is given in Code 1. It is interesting to note that for the first time we introduce a **conditional binding depending on the `is_log` boolean value** (lines 11-13). Also, the list of the stages is declared in a dedicated `pip_stages` variable (line 18) before to call the pipeline constructor (line 50). It makes possible to define the "compute" graph, and its decomposition in stages, only once. After that, depending on the `is_log` value, the `pip_stages` variable can be modified to add the log task (lines 39-44). Note that the t_4 is added to the first tasks list of stage 2. This is because t_4 depends on t_1 which is declared in the stage 1. Finally, the stop condition of a pipeline can be set as one stop condition per stage (see lines 57-59). Generally, we will only customize the stop condition of the last stage.

Work to do #1 Implement the 3-stage pipeline given in Fig. 1 in the `motion2_aff3ct.cpp` file (it is strongly recommended to save your previous working implementation with the sequence in an other file...). Moreover, your code should be able to enable/disable the logs and the visualization. Fig. 2 and Code 1 are here to help you and you must take inspiration from it. Validate that your new pipeline version is working (see Note #2 in the previous session). Measure and report the achieved FPS in the `motion2_aff3ct.cpp` file.

Note #1 If you implemented a sequence with a `Switcher` (to skip the first execution of $\Sigma\Delta$), there is a known problem: a `select` or a `commute` task cannot be directly at the interface of a synchronization between two pipeline stages. To make this work, you need to add a `Relayer` (`relay` task) after the `select` task. If you did not implemented a sequence with a `Switcher` then you can ignore this note.

The code for displaying pipeline task statistics differs slightly from the code for a sequence. Indeed, you first need to iterate over each stage before to print the statistics.

```
1  const bool ordered = true, display_throughput = false;
2  auto stages = pip.get_stages();
3  for (size_t s = 0; s < stages.size(); s++) {
4      const int n_threads = stages[s]->get_n_threads();
5      std::cout << "#" << std::endl << "# Pipeline stage " << (s + 1) << " ("
6                << n_threads << " thread(s)): " << std::endl;
7      tools::Stats::show(stages[s]->get_tasks_per_types(), ordered, display_throughput);
8  }
```

Source code 2: Display the statistics of the tasks for a pipeline.

Work to do #2 Modify the code that displays the task statistics in `motion2_aff3ct.cpp` (see Code 2).

When using the `--stats` option, you should have an output similar to the Code 3. As you can see, the pipeline automatically added new `push_x` and `pull_x` tasks for you. These tasks are taking care of the communications and synchronizations between the pipeline stages. In other words, they represent a *producer/consumer* implementation. For instance, the `push_1` task of the `Adp_1_to_n_0` module sends/produces output data from stage 1 to stage 2. And, in stage 2, the `pull_n` task of the `Adp_1_to_n_0` module receives/consumes the input data from stage 1. Sometimes `push_x` and `pull_x` tasks take a lot of time in the stage. **It means that they are waiting.**

Work to do #3 From the output statistics on the Jetson TX2, which stage limits the overall throughput of the pipeline? Note your answer in the `motion2_aff3ct.cpp` file.

```
# Pipeline stage 1 (1 thread(s)):
# ----------------------------------||----------------------------||-------------------------------
#        Statistics for the         ||       Basic statistics     ||        Measured latency
#            given task             ||          on the task       ||
# ----------------------------------||----------------------------||-------------------------------
# -------------|--------------------||----------|----------|--------||----------|----------|----------
#       MODULE |               TASK ||    CALLS |     TIME |   PERC ||  AVERAGE |  MINIMUM |  MAXIMUM
#              |                    ||          |      (s) |    (%) ||     (us) |     (us) |     (us)
# -------------|--------------------||----------|----------|--------||----------|----------|----------
# Adp_1_to_n_0 |             push_1 ||      100 |     1.66 |  50.39 || 16592.90 |     0.75 | 18569.21
# Sigma_delta0 |            compute ||      101 |     0.81 |  24.72 ||  8060.59 |  7980.08 |  8568.33
# Sigma_delta1 |            compute ||      100 |     0.79 |  24.08 ||  7928.90 |  7866.54 |  8604.50
#        Video |           generate ||      101 |     0.01 |   0.38 ||   123.89 |     0.00 |   252.00
#      Delayer |            produce ||      101 |     0.01 |   0.31 ||    99.50 |    54.88 |   131.29
#      Delayer |           memorize ||      100 |     0.00 |   0.13 ||    43.54 |    36.96 |   101.83
# -------------|--------------------||----------|----------|--------||----------|----------|----------
#        TOTAL |                  * ||      100 |     3.29 | 100.00 || 32932.16 | 16019.56 | 36316.68
#
# Pipeline stage 2 (1 thread(s)):
# ----------------------------------||----------------------------||-------------------------------
#        Statistics for the         ||       Basic statistics     ||        Measured latency
#            given task             ||          on the task       ||
# ----------------------------------||----------------------------||-------------------------------
# -------------|--------------------||----------|----------|--------||----------|----------|----------
#       MODULE |               TASK ||    CALLS |     TIME |   PERC ||  AVERAGE |  MINIMUM |  MAXIMUM
#              |                    ||          |      (s) |    (%) ||     (us) |     (us) |     (us)
# -------------|--------------------||----------|----------|--------||----------|----------|----------
#      Morpho1 |            compute ||      100 |     1.32 |  39.34 || 13180.10 | 13110.33 | 13680.50
#      Morpho0 |            compute ||      100 |     1.31 |  39.21 || 13134.11 | 13111.88 | 13961.21
#         CCL0 |              apply ||      100 |     0.22 |   6.52 ||  2183.68 |  2074.96 |  3518.04
#         CCL1 |              apply ||      100 |     0.22 |   6.47 ||  2168.97 |  2053.67 |  2415.08
#         CCA1 |            extract ||      100 |     0.13 |   3.96 ||  1325.75 |  1103.25 |  1960.42
#         CCA0 |            extract ||      100 |     0.13 |   3.92 ||  1314.09 |   724.04 |  1920.88
# Adp_1_to_n_0 |             pull_n ||      100 |     0.02 |   0.52 ||   172.73 |     0.04 | 17246.83
#  Ftr_filter1 |             filter ||      100 |     0.00 |   0.02 ||     7.02 |     4.38 |    27.46
#  Ftr_filter0 |             filter ||      100 |     0.00 |   0.02 ||     6.48 |     4.38 |    16.71
#          KNN |              match ||      100 |     0.00 |   0.02 ||     6.25 |     1.54 |    62.88
# Adp_1_to_n_1 |             push_1 ||      100 |     0.00 |   0.00 ||     0.63 |     0.21 |     6.58
# -------------|--------------------||----------|----------|--------||----------|----------|----------
#        TOTAL |                  * ||      100 |     3.35 | 100.00 || 33499.82 | 32188.67 | 54816.59
#
# Pipeline stage 3 (1 thread(s)):
# ----------------------------------||----------------------------||-------------------------------
#        Statistics for the         ||       Basic statistics     ||        Measured latency
#            given task             ||          on the task       ||
# ----------------------------------||----------------------------||-------------------------------
# -------------|--------------------||----------|----------|--------||----------|----------|----------
#       MODULE |               TASK ||    CALLS |     TIME |   PERC ||  AVERAGE |  MINIMUM |  MAXIMUM
#              |                    ||          |      (s) |    (%) ||     (us) |     (us) |     (us)
# -------------|--------------------||----------|----------|--------||----------|----------|----------
# Adp_1_to_n_1 |             pull_n ||      100 |     3.35 |  99.98 || 33472.32 | 32689.25 | 52477.75
#     Tracking |            perform ||      100 |     0.00 |   0.02 ||     6.25 |     2.08 |    44.42
# -------------|--------------------||----------|----------|--------||----------|----------|----------
#        TOTAL |                  * ||      100 |     3.35 | 100.00 || 33478.57 | 32691.34 | 52522.17
```

Source code 3: Task statistics output for the pipeline described in Fig. 1. The automatically added `Adaptor` tasks (`push_x` and `pull_x`) are highlighted. The code ran on the Apple Silicon M1 Pro CPU. For each stage, the tasks are ordered by their execution time.

2.2 Tasks Replication

In Fig. 1, the ΣΔ and the *tracking* tasks cannot be replicated because they have a data dependency over time. As a consequence, the tasks of the stages 1 et 2 cannot be replicated. These dependencies are not explicit in Fig. 1 because they come from inner data:

- In `sigma_delta_compute` function: the `sd_data->M` and `sd_data->V` buffers,
- In `tracking_perform` function: `tracking_data->history` and `tracking_data->tracks` buffers.

However, in theory, stage 2 tasks can all be replicated. For instance, in the Code 1 you can perform the replication of stage 2 by replacing the second "1" line 51. If you replace `{ 1, 1 }` with `{ 1, 4 }`, the pipeline will replicate stage 2 four times and it will run a total of 5 threads (1 for stage 1 and 4 for stage 2).

Work to do #1 Modify the pipeline in the `motion2_aff3ct.cpp` file to run the second stage over two threads (2 times replication). What is happening? Is it normal?

You should not be able to replicate stage 2 because some of the tasks are stateful. To overcome this problem you need to implement the `clone` method (and maybe the `deep_copy` method) on the involved modules.

```
1  Morpho* Morpho::clone() const {
2      auto m = new Morpho(*this);
3      m->deep_copy(*this); // we override this method just after
4      return m;
5  }
6  // in the deep_copy method, 'this' is the newly allocated object while 'm' is the former object
7  void Morpho::deep_copy(const Morpho& m) {
8      // call the 'deep_copy' method of the Module class
9      Module::deep_copy(m);
10     // allocate new morpho inner data
11     this->morpho_data = morpho_alloc_data(m.morpho_data->i0, m.morpho_data->i1,
12                                           m.morpho_data->j0, m.morpho_data->j1);
13     // initialize the previously allocated data
14     morpho_init_data(this->morpho_data);
15 }
```

Source code 4: `clone` and `deep_copy` methods of the `Morpho` module.

Code 4 gives you the implementation of the `clone` and `deep_copy` method for the `Morpho` module.

Work to do #2 Implement the missing `clone` and `deep_copy` methods for stage 2 tasks. Then, run the pipeline with 2 replications in stage 2. Is it faster? Increase the number of threads in stage 2, what do you observe? Note the new FPS in your `motion2_aff3ct.cpp` file.

2.3 Tasks Graph Simplification

Fig. 3 presents an optimized version of the tasks graph shown in Fig. 1. One may have noticed that in Fig. 1 some computations are performed twice! Indeed, the computations on the $t-1$ frame could be memorized from a stream to the next one. In Fig. 3, the list of CCs at t, after the surface filtering, are memorized with a `Delayer`. This way, in the next stream at $t+1$, they can be reused (`produce` task of the `Delayer`) to perform the k-NN algorithm.

The `produce` and `memorize` tasks are stateful and by definition they have a data dependency over time. Thus, they cannot be replicated. Fig. 3 gives you the new stages decomposition.

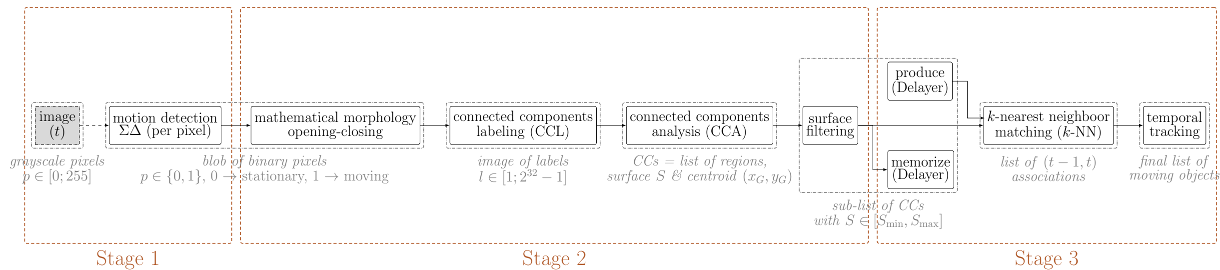


Figure 3: MOTION simplified graph. In gray and italic: the output of each processing. The graph is split into a 3-stage pipeline.

Work to do #1 Implement this simplified tasks graph version in a new `motion_aff3ct.cpp` file. This will produce the `motion-aff3ct` executable. Verify that this new version is working by comparing its logs output with the `motion2` executable logs. Then, report the achieved FPS in the `motion_aff3ct.cpp` file. It is better? How much and why?

Work to do #2 The k-NN task could have been replicated and a 5-stage pipeline could have been implemented. Why this does not sound like a good idea? Comment your answer in the `motion_aff3ct.cpp` file.

2.4 Data Parallelism with OpenMP

Now, the limiting stage should be the stage 1 because of the $\Sigma\Delta$ task. This task cannot be replicated because of its data dependency over time. However, for one given frame, each pixel is computed independently of the others! This means that the loop over the pixels can be split in multiple sub-domains and each sub-domain can be affected to a thread.

Work to do #1 Write a multi-threaded version the code of the $\Sigma\Delta$ algorithm (`sigma_delta_compute` function). For this you will use the OpenMP library.

Now you will have 3 different types of parallelism in MOTION: the pipeline, the replication and the data parallelism inside the $\Sigma\Delta$ task (with OpenMP). Remember that you can control the number of OpenMP threads with the `OMP_NUM_THREADS` environment variable.

Work to do #2 Find a configuration of threads that gives the best throughput. You can change the number of OpenMP threads and the number of replications in stage 2 of the pipeline. Report this configuration and the achieved throughput in the `motion_aff3ct.cpp` file.

2.5 Task Vectorization

An other way to speedup the $\Sigma\Delta$ task is to vectorize it with the SIMD instructions. As each pixel computation is independent, the $\Sigma\Delta$ task is a very good candidate for vectorization.

Work to do #1 Write a vectorized version the $\Sigma\Delta$ algorithm (inside the `sigma_delta_compute` function). For this you will use the MIPP wrapper (do not forget to add the `#include <mipp.h>` header).

Now you can combine the 3 previous types of parallelism with a new one: the vectorization of $\Sigma\Delta$.

Work to do #2 Find a configuration of threads that gives the best throughput (with the $\Sigma\Delta$ vectorized task). Report this configuration and the achieved throughput in the `motion_aff3ct.cpp` file.

2.6 [Bonus] Data Parallelism versus Pipeline+Replication

The most compute intensive tasks of the MOTION application are the $\Sigma\Delta$ and the morphology. We saw before that $\Sigma\Delta$ can easily be speedup over its data parallelism. In fact, this is the same for the morphology algorithms!

Work to do #1 First you will create a new `motion.c` file where you will implement the simplified graph given in Fig. 3. It will produced a new `motion` executable without AFF3CT (no pipeline, no replication). You can start from a copy-paste of the `motion2.c` file.

Work to do #2 Parallelize the morphology tasks with OpenMP. Compare the throughput of the `motion` executable (only data parallelism) with the throughput of the `motion-aff3ct` executable (data, pipeline and replication parallelisms). Which one gives the highest throughput? Why? Report the achieved results in the corresponding source files.

Final note We could also have vectorized the morphology task, it is combination of stencils like the `blur` kernel we worked on in previous sessions... But we will keep this for an other time ;-).