# Optional Exercises on Stencil Fusion
# Parallel Functional Programming

Troels Henriksen and Cosmin Oancea
DIKU, University of Copenhagen

January 2023

**Introduction**

**There is no deadline and no submission, since these exercises are optional and not checked, i.e., it is up to you if you solve them or not and they will not influence your grade.**

**Task 1: Fusing Stencils: The Blur Example (Halide Lecture)**

The file `blur-fusion.cpp` contains several of the implementations discussed in the Halide lecture. More precisely, the breadth-first scheduling, the fully/totally-fused scheduling and the sliding-window scheduling are already implemented there.

Your task is to fill in the missing code corresponding to the implementation of:

a) tiled-fusion scheduling in function `tiledFused`, and

b) sliding-window within tiles scheduling in function `tiledWindow`.

The code structure and the OpenMP parallelization is already provided to you, together with the validation. Your task is to fill in the missing parts of the code (see comments in the code itself).

After your implementation validates for both cases, please run them on (different) multicore hardware that you have access to and measure the runtime of each implementation on each such hardware (e.g., which of the five schedules is the fastest, and what speedup is achieved in comparison with the breath-first scheduling).

Please note that `futharkhpa01..3fl.unicph.domain` use the same type of hardware, so test it only once for one of them. Similar for `gpu01..4-diku-apl`.

### Task 2: Iterative Stencil Fusion

This task refers to the following simple one-dimensional iterative stencil:

```
for(int q=0; q<ITER; q++) {
    forall(int x=0; x<DIM_X; x++) // parallel
        out[x] = ( inp[x-1]+inp[x]+inp[x+1])/3;
    tmp = inp; inp = out; out = tmp; // switch inp with out
}
```

The breadth-first scheduling is already implemented in file `it1d-stencil.cpp`, function `breadthFirst`.

Your task is to fill in the implementation of the function `tiledFused` in the same file `it1d-stencil.cpp`, which has the semantics of:

1 tiling the loop of index `x` with a tile `T`,

2 stripmining the outer loop of count `ITER` by a `FUSEDEG` factor, and

3 'interchanging' the stripmined slice inside the loop of index `x`, while adding enough redundant computation for the ghost zones to respect the original program behavior.

Assuming for simplicity that `FUSEDEG` evenly divides `ITER`, the structure of the resulted code is given below:

```
for(int qq=0; qq<ITER; qq+=FUSEDEG) {

  forall(int tx=0; tx<DIM_X; tx+=T) { // parallel

    float tile[2][-FUSEDEG .. T+FUSEDEG]; // allocated per thread

    // 1. initialize tile[0][-FUSEDEG   : FUSEDEG+T] from array 'inp'

    for(int32_t q0 = 0; q0 < FUSEDEG; q0++) {
      // 2. compute one iteration of the original stencil using
      //    the 'tile' array, which is allocated per thread:
      //    iter q0=0 computes tile[1][-FUSEDEG+1 : FUSEDEG+T-1] and
      //              reads from tile[0][-FUSEDEG   : FUSEDEG+T  ],
      //
      //    iter q0=1 computes tile[0][-FUSEDEG+2 : FUSEDEG+T-2] and
      //              reads from tile[1][-FUSEDEG+1 : FUSEDEG+T-1]
      //    ... and so on ...
    }

    // 3. copy back the result 'tile' to array 'out'
  }
  tmp = inp; inp = out; out = tmp; // double buffering
}
```

Your task is to implement the missing code, such that the program validates, and to test it on as many different multicore hardware as you have

access to and report the speedup over the breadth-first scheduling for each of them.