Weekly Assignment 3
Parallel Functional Programming

Troels Henriksen and Cosmin Oancea
DIKU, University of Copenhagen

December 2020

Introduction

The handin deadline is the 10th of December.

The handin is expected to consist of a report in either plain text or PDF file (the latter is recommended unless you know how to perform sensible line wrapping) of 4—5 pages, excluding any figures, along with an archive containing your source code. The report should contain instructions on how to run and benchmark your code.

Task 1: Programming in `ispc`

For this task you will be programming in `ispc`. The code handout contains a `Makefile` and benchmarking infrastructure for several programs implemented in both sequential C and `ispc`. Of these, three `ispc` implementations are blank, and it is your task to finish them.

A program `foo` can be benchmarked with the command `make run_foo`. This will also verify that the result produced by the `ispc` implementation matches the result produced by the C implementation. Feel free to change the input generation if you wish, for example to make the input sets larger or smaller.

Generally, do not expect stellar speedups from these programs. A $\times 2$ speedup over sequential C on DIKUs GPU machines should be considered quite good. By default, the `Makefile` sets `ISPC_TARGET=sse4`, which is a relatively old instruction set. You may get better performance by setting it to `host`, which will tell `ispc` to use the newest instruction set supported on the CPU you are using.

All of the subtasks involve some kind of cross-lane communication. Each subtask contains a list of the builtin functions I found useful in my own implementation, but you do not have to use them, and you are welcome to use any others supported by `ispc`. It is likely that my own solution is not optimal anyway.

Task 2: Prefix sum (`scan.ispc`)

The task here is to implement ordinary *inclusive* prefix sum, which you should be quite familiar with by now.

Recommended builtin functions: `exclusive_scan_add()`, `broadcast()`

Task 3: Removing neighbouring duplicates (`pack.ispc`)

This program compacts an array by removing duplicate neighbouring elements. It has significant similarities to the filter implementation in `filter.ispc`.

Recommended builtin functions: `exclusive_scan_add()`, `reduce_add()`, `extract()`, `rotate()`

Task 4: Run-length encoding (`rle.ispc`)

Run-length encoding is a compression technique by which runs of the same symbol (in our case, 32-bit words) are replaced by a *count* and a *symbol*. For example, the C array

```
{ 1, 1, 1, 0, 1, 1, 1, 2, 2, 2, 2 }
```

is replaced by the array

```
{ 3, 1, 1, 0, 3, 1, 4, 2 }
```

Recommended builtin functions: `all()`

Hint: This task is significantly more tricky than the two others. I advise optimising for the case where each symbol is repeated many times, which can be quickly iterated across in a SIMD fashion, falling back to scalar/single-lane execution when a new symbol is encountered. A rough pseudocode skeleton could be:

```
while at least programCount elements remain to be read:
  read next programCount elements
  if all equal to current element:
    increase count and move to next loop iteration
  else:
    # Use single lane to find the mismatch
    if programIndex == 0:
      ...
```

Task 5: 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 report 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 `gpu01..4` use the same type of hardware, so report it only once for one of them. Your laptop or home desktop has different hardware characteristics, so report for those as well, etc.

Task 6: 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.