Weekly Assignment 4
Parallel Functional Programming

Troels Henriksen and Cosmin Oancea
DIKU, University of Copenhagen

December 2019

Introduction

The handin deadline is the 19th 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: Polyhedral Transformations:

This task refers to polyhedral analysis, please see `L9-polyhedral.pdf`; the task is also summarized by the last slides in said document.

Please install the `islpy` library by running `pip install -user islpy`.

Your task is to encode in the polyhedral model three code transformations:

- loop interchange (a.k.a., permutation), in file `code-handout/poly-transf/permutation.py`;
- scaling, in file `code-handout/poly-transf/scaling.py`;

- reindexing, in file `code-handout/poly-transf/reindexing.py`.

Please follow the hints and instructions in said files. Your task is to fill in the blanks—in each file—the implementation of:

- the iteration domain,
- the original (sequential) schedule,
- the read/write access relations, and
- most importantly **the transformed schedule**.

Please include in your report:

- your (full) implementation of the (i) iteration domain, (ii) original schedule, (iii) read/write access relations and (iv) the transformed schedule (i.e., only those full lines; do not include the rest of the handed out code);
- a brief explanation of the encoding of the transformed schedule (for the others, the code should be self explanatory);
- for the *permutation (interchange)* and *reindexing* transformation, can you devise a schedule that tests that the transformed loop is parallel? (If so, please report it as well.)