Weekly Assignment 2
Data Parallel Programming

Troels Henriksen and Cosmin Oancea
DIKU, University of Copenhagen

November 2024

Introduction

This weekly assignment focuses on list homomorphisms and ISPC programming.

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—6 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.

Subtask 1.1: 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()`

Subtask 1.2: 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()`

Subtask 1.3: 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()`

Hints: 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 2: Tree Operations

This task was inspired by Eric Wastl.

Consider a tree described as a preorder traversal, through a sequence of *steps*. Each step is one of the following:

- `d`x: Go to the next child of the current node, which has an integral value of x.
- `u`: Go to the parent of the current node.

Steps are separated by whitespace. The first step must be `d`x. Examples can be seen on fig. 1. You can also use these to test your implementation. The first step must be `d`x. It is not allowed to add multiple root nodes.

You are given a code handout `trees.fut` with code that can parse a text representation of a traversal into an appropriate Futhark type. Your tasks for this assignment will be to decode the traversal into arrays that describe the depth and parent of each node in the tree, then to implement simple operations on trees. Note that the number of nodes in the tree is the same as the number of `d`x steps. The `u` steps to not contribute to the number of nodes in the tree.

Subtask 2.1: Implement `depths`

Implement the function `depths` that given an array of `step`s produces an array. The length should have an element for each node in the tree. Each element is of the form (d, v), where d is the depth of that node, and v is the value of that node (as given by the `d` step).

Analyse the work and span of your function. Is it work-efficient?

Hints:

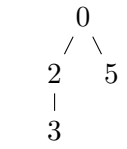
- You will need to implement your own function for computing exclusive scans.
- Remember that `u` steps do not add new nodes to the tree.

Subtask 2.2: Implement `parents`

Implement the function `parents` that given the D array produces the P array, as discussed in class.

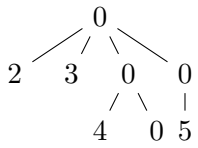
Analyse the work and span of your function. Is it work-efficient?

```
d0
 d2
  d3 u
  u
 d5
```



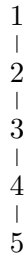
$$D = [0, 1, 2, 1]$$
$$V = [0, 2, 3, 5]$$
$$P = [0, 0, 1, 0]$$

```
d0
 d2 u
 d3 u
 d0
  d4 u
  d0 u
  u
 d0
  d5
```



$$D = [0, 1, 1, 1, 2, 2, 1, 2]$$
$$V = [0, 2, 3, 0, 4, 0, 0, 5]$$
$$P = [0, 0, 0, 0, 3, 3, 0, 6]$$

```
d1
 d2
  d3
   d4
    d5
```



$$D = [0, 1, 2, 3, 4]$$
$$V = [1, 2, 3, 4, 5]$$
$$P = [0, 0, 1, 2, 3]$$

Figure 1: Examples of trees and how they are encoded as steps. The indentation is solely for human readability. Note that we are not required to return to the root node at the end of the traversal.

Subtask 2.3: Implement `subtree_sizes`

A *subtree* rooted at i is the tree with node i as its root. The function `subtree_sizes` computes for each node i in a tree, the sum of all nodes of the subtree rooted at i. For example, for the tree

```
d0 d2 u d3 u d0 d4 u d0 u u d0 d5
```

the result is

$$[14, 2, 3, 4, 4, 0, 5, 5]$$

Implement the function `subtree_sizes`. Analyse the work and span of your function. Is it work-efficient?

Hints:

- A simple implementation is bottom up iteration. If d is the maximum depth of the tree, start by considering nodes at depth d, then $d - 1$, etc.