Weekly Assignment 4
Parallel Functional Programming

Troels Henriksen and Cosmin Oancea
DIKU, University of Copenhagen

December 2019

Introduction

The handin deadline is the 20th of December.

The handin is expected to consist of a report in PDF format 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.

This text is incomplete—in a few days we will update it with a second task about automatic differentiation. We will make an announcement after doing so.

Task 1: Polyhedral Transformations

This task refers to polyhedral analysis, please see `L7-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.)

Task 2: Automatic Differentiation of Scan with Multiplication

Apply the rules for the adjoint code-generation of scan; see "Differentiating Scan" subsection in lecture 8 (`L8-reverse-ad.pdf`).

Your task is to implement function `scanAdjoint` in Futhark file `scan-rev-ad.fut` in folder `code-handout/scan-rev-ad`.

Run the program in the following way:

```
$ futhark dataset --f64-bounds=0.5:1.6043 -g [1000000]f64 -b | ./scan-rev-ad
-r 5 -t /dev/stderr
```

or on sizes less than 1000000.

If the execution returns true, then your program is likely correct, i.e., it validates, otherwise it is incorrect.

Write in your report:

- your full implementation (should be about 10 code lines or less);
- whether it validates or not.

Task 3: Optimizing the Reverse AD code of Matrix Multiplication (Pen and Paper)

This task is about applying reverse-mode automatic differentiation and optimizing as much as possible the resulted computation. In terms of maths, it is known that matrix multiplication is a bilinear function, and as such differentiating it should result into two matrix-multiplication computations.

Assume matrices $A, B \in \mathcal{M}^{N \times N}$. Their multiplication is written as:

```
let C =
  map(\A_row ->
        map(\B_col ->
              let ps = map (*) A_row B_col
              let c  = reduce (+) 0.0 ps
            ) (transpose B)
      ) A
...
```

We have said in class that differentiating a `map` will result into a generalized reduction construct, that we will negligently represent here as an imperative-like `forall` loop. **The next page shows the unoptimized adjoint code.** This has been obtained in the following way:

- the two outer-maps are perfectly nested, hence morally equivalent to one flat map. It follows that we do not apply the redundant-computation argument to the second one.
- the dot-product, represented by the map-reduce innermost computation is subject to redundant execution, so we "inline" the adjusted code there.
- For the adjoint code:
  - we initialize the $\overline{ps}$ array with zeros;
  - we translate the `reduce (+)` by the specialized rule for sum, resulting in `map (+`$\overline{r}$`)`;
  - we translate the innermost `map` by a generalized reduction, in which we insert the accumulation statements (`+=`).

```
let C = ...
...
-- Reverse-AD reaches the point of matrix multiplication.
-- By this point we have already computed C̄ ∈ M^{N×N}
-- and also some partial contribution Ā, B̄ ∈ M^{N×N}
-- The unoptimized adjoint code of matrix multiplication is given below:
  forall i = 0 .. N-1 do
    forall j = 0 .. N-1 do
      -- redundant computation of the original code
      let ps = map (*) A[i,:] B[:,j]
      let c  = reduce (+) 0.0 ps

      -- Adjoint Code
      let r̄ = C̄[i,j]
      let p̄s = replicate N 0.0 -- initialize p̄s

      -- translate ''let c  = reduce (+) 0.0 ps''
      let p̄s = map (+r̄) p̄s

      -- translate ''let ps = map (*) A[i,:] B[:,j]''
      --           by means of generalized reduction:
      forall k = 0 .. N-1
        -- original statement was ps[i] = A[i,k] * B[k,j]
        -- hence its adjoints are:
        let Ā[i,k] += B[k,j] * p̄s[k]
        let B̄[k,j] += A[i,k] * p̄s[k]
```

Your task is to apply various rewrite rules–as in transformations that a compiler can possibly do—in order to simplify this code to one that resembles two matrix-matrix multiplications. Write in your report:

- each code transformation (rewrite-rule) that you have applied and the resulted code. Hints:
  - apply dead code elimination
  - how can one simplify `map (+x) (replicate n a)`?)
  - inline $\overline{r} = \overline{C}[i,j]$
  - apply loop distribution and interchange, see the last couple of slides of Lecture 8 (reverse AD).
- Once you have fully optimized the code, it should be possible to rewrite it entirely using a purely-functional notation, as two matrix multiplications (see first figure). Also show this "functional" code!