Weekly Assignment 4
Parallel Functional Programming

Troels Henriksen and Cosmin Oancea
DIKU, University of Copenhagen

December 2025

Introduction

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. Please fill in the required parts in the handed-out code and submit all the code.

Please do not submit the datasets, not even the ones for k-means `kdd_cup.in.gz` and `kdd_cup.out`, which for convenience are included in the handed-out archive.

Task 1: Apply Reverse AD to a `map - reduce min` Composition

This task should be implemented in file `code-hand-out/task-1/min.fut`.

The task consists of implementing the function `task_mapomin` which should have the same semantics as `vjp2 primal as y_bar`, where the primal is defined as a composition of reduce with min and a map, as shown below:

```
def primal [n] (as: [n]f32) : f32 =
  let bs = map (\a -> let b = a*a - 0.5*a in b) as
  let y  = reduce_comm (f32.min) f32.highest bs
  in  y

def task_mapomin [n] (as : [n]f32) (y_bar : f32) : (f32, [n]f32) =
  -- please implement me so that I have the semantics of
  -- vjp2 primal as y_bar
```

Please note that your code should manually implement reverse AD, i.e., should not use any of the differentiation constructs `vjp2`, `vjp`, `jvp2`, or `jvp`.

For refreshing your memory on how to differentiate a simple `map` and a `reduce min` you should probably consult the slides entitled "Classical API for AD", "Differentiating Map: The Simple Case" and "Special Case: Min/Max" in `L7and8-AD.pdf`

Essentially, you will need to first write the code of the primal trace (remember to lift the min operator), followed by the code of the reverse trace, which first computes the adjoint of the `reduce min` and then the adjoint of the `map`. ***The result should consist of:*** (1) the result of the primal, i.e., original program, (2) tupled with the **adjoint** of 'as'.

Make sure that your solution validates.

```
$ futhark bench --backend=cuda min.fut -e validate
```

before measuring runtime, which can be simply achieved by:

```
$ futhark bench --backend=cuda min.fut
```

Your report should contain:

- a short statement related to whether your code validates or not
- the implementation of the lifted operator associated to `f32.min`, together with a brief rationale, e.g., informally, why is it associative and does the job?
- the implementation of the `task_mapomin` function
- a short explanation about the correspondence between the adjoint and primal code fragments, i.e., which code implement the adjoint of the map and which the adjoint of the `reduce min`
- a table with the runtimes of `mapomin_primal`, `mapomin_vjp2` and `mapomin_manual` on the provided datasets, and the AD overhead of `mapomin_vjp2` and `mapomin_manual` (the AD overhead denotes the slowdown factor of the differentiated code in comparison with the primal/original code).

Task 2: Generic Reverse-AD of Reduce with Invertible Operator

This task refers to implementing the function `vjp2_red_inv` in file `code-hand-out/tasks-2-and-3-inv/gen-vjp2-cominv.fut`.

Said function implements the reverse-mode AD of a reduce whose operator is both commutative and invertible. Before you start, please refresh your memory by reading up (at least) the slides entitled "Special Case: Multiplication" and "Generalization based on Invertible Operators" from `L7and8-AD.pdf`. The latter slide is pretty much what you have to implement, but now all the auxiliary functions (`cfwd`, `cbwd`, etc.) are passed explicitly as function arguments. The intent is that by implementing it you will understand better what is going on.

The result should consist of: (1) the result of the primal, i.e., the original reduce, (2) tupled with the **adjoint** of the input, i.e., the to-be-reduced array `as`.

You should probably first generate all needed datasets—also the ones for the next task—by running:

```
$ make datasets
```

The `test-red.fut` file tests your implementation on two different commutative and invertible operators, which are defined in file `ops.fut`:

- multiplication (`*`), which is as in the slides
- the sum-of-product (`SOP`) operator:

```
def sop (p1: f64, s1: f64) (p2: f64, s2: f64) =
  (p1 + p2 + s1*s2, s1+s2)
```

We would recommend that you first validate your implementation before measuring performance. This can be accomplished with:

```
$ futhark bench --backend=cuda test-red.fut -e validate_sop
$ futhark bench --backend=cuda test-red.fut -e validate_mul
```

The validation essentially calls three different algorithms for differentiating reduce (one of them is your implementation), and tests that their results are equal (up to an epsilon).

Once it validates you may measure performance by running all entry points:

```
$ futhark bench --backend=cuda test-red.fut
```

Alternatively, you are welcome to make special targets in the Makefile, that currently validates both tasks 2 and 3 and, as well, runs all entry points of both tasks. Beside validation, the entry points of `test-red.fut` are:

- `primal_sop/mul`: this is the primal code (target to differentiation),
- `vjp2_sop/mul`: this calls Futhark's built-in reverse AD
- `ppad_sop`: this is the implementation proposed in the paper "Parallelism-Preserving Automatic Differentiation for Second-Order Array Languages" cited in the lecture slides and abbreviated PPAD from now on.
- `vjp2_sop/mul_inv`: this calls your implementation.

Your report should contain:

- a short statement related to whether your code validates or not,
- the whole implementation of the `vjp2_red_inv` function, together with a brief explanation of the rationale, i.e., how is the invertible property used to obtain a more efficient differentiation.
- a table with the runtimes of `primal_sop/mul`, `vjp2_sop/mul`, `ppad_sop`, and `vjp2_sop/mul_inv` on the provided datasets, and the AD overhead of the latter three.

Task 3: Generic Reverse-AD of Reduce-By-Index with Invertible Operator

This task refers to implementing the function `vjp_redbyind_inv` in file `code-hand-out/tasks-2-and-3-inv/gen-vjp2-comminv.fut`.

Said function is supposed to implement the reverse-mode AD of a reduce-by-index whose operator is both commutative and invertible. Before you start, please refresh your memory by reading up (at least) the slides entitled "Differentiating General-Case Reduce-By-Index" and, more importantly, "Differentiating Red-by-Index with Invertible Operator" from `L7and8-AD.pdf`. The latter slide explains what you have to implement.

The testing file associated with reduce-by-index is `test-hist.fut`. Its code structure is similar to the one in the previous task. ***The result should consist of:*** (1) the result of the primal, i.e., the original reduce-by-index, tupled with (2) the **adjoint** of the `dest` array, i.e., the one that is "overwritten" by reduce-by-index, and (3) the *adjoint* of the input-data array `vs`, i.e., the array that holds the to-be-accumulated data. **Remember to make the datasets first**.

Your report should contain: the same as in the previous task but for the entry points related to reduce-by-index. Please note that PPAD does not have an entry point here (because it is not aimed at reduce-by-index).

Task 4: Implement kMeans with Nested Forward & Reverse AD

This task refers to implementing three pieces of missing code in file `code-hand-out/tasks-4-kmeans/kmeans.fut`:

(a) the cost function

(b) the nested reverse in forward mode AD

(c) the update of the cluster centers

Before you start, it might be useful to refresh your memory by reading up the slides referring to subsection "Case Study: k-Means" from `L7and8-AD.pdf`.

We provide a reference input and result dataset named `kdd_cup.in.gz` and `kdd_cup.out`, respectively.

Your report should contain:

- a short statement related to whether your code validates or not,
- the implementation of the three missing pieces of the code and a short explanation for each one.