OCANNL optimization framework

Tensor shape inference, concise notation, multidevice runtime

BY ŁUKASZ STAFINIAK*

Abstract

OCANNL is a Deep Learning framework with first-order automatic differentiation (aka. backprop) that implements low level backends, puts emphasis on shape inference and concise notation, supports multiple devices parallelism. Currently, at the core OCANNL provides explicit compilation and synchronization.

1 Powerful tensor shape inference

The most distinctive aspect of OCANNL is its row-based shape inference. A tensor shape in OCANNL is composed of three rows of axes: batch, input and output. In the underlying n-dimensional array implementation of tensors, inputs are innermost, but we use a types-inspired syntax: "batch|input→output" (or "input→output", "batch|output", "output"), where "batch", "input", "output" are axis entries (an axis entry is either comma-separated or an individual character).

The constraints we derive to perform shape inference are subtyping relations (inequalities) between rows. The subtyping relation accounts for *broadcasting*: matching dimension-1 angainst a greater dimension, and allowing for more axes in a row. A dimension specifies a single axis in a shape. The constraints include both dimension variables and row variables. Uniquely, the row variables in OCANNL are embedded: a row specification consists of innermost axes, an optional row variable, and optional outermost axes that come to the left of the row variable in the solution. The constraint solver solves both equations and inequalities between pairs of rows and dimensions.

In OCANNL, a *shape logic* specification serves two purposes: it is an input to shape inference, but, once the shapes are inferred, it is also an input to *projections inference*. Projections contribute to the semantics of an operation by specifying how the tensors should be indexed. Currently, most aspects of a shape specification can be expressed using a syntax resembling the *einsum notation*, but much more general. For unary operations the syntax is RHS⇒LHS, and for binary operations RHS1;RHS2⇒LHS, where RHS,LHS,RHS1,RHS2 are shape specifications. The syntax of a named row variable is `..v..` for name `v`, and an un-named row variable `...` stands for a row variable named `batch`, `input`, or `output` depending on where `...` appears. To give some concrete examples, tensor multiplication (matrix multiplication generalized to multiple axes), would be an operation with operator `*`, accumulator `+`, and shape logic:

```
...|..args..->...; ...|...->..args.. => ...|...->...
```

When no additional constraints are put on them, variables that appear on the RHS only (to the left of `=>`), whether row or dimension, have their corresponding axes reduced (e.g. summed over) by the operation. Reducing (summing out) of the leftmost batch axis would be:

```
s...|...->... => ...|...->...
```

With an additional constraint on `s`, this also represents a slice at a position of the leftmost batch axis. When numbers appear as dimensions in a specification, they provide axis positions. For example, reducing (e.g. summing) all entries of a tensor at 0-indexed position 1 of outermost output axis and 2 of innermost output axis:

```
...|...->1...2 => 0
```

For pragmatic reasons, shape inference is organized into stages and makes a few heuristic assumptions at later stages. OCANNL also supports dimension labels (but not labels for selecting axes).

*. Since April 2024, ahrefs sponsors Łukasz's work on OCANNL.

2 Code: representation, notation, optimization

Computation in OCANNL is organized into a declarative layer of *tensor* expressions, on top of an imperative layer of *tensor node* assignments. Both layers come with PPX extension points: `%op` (operations) for tensor expressions, and `%cd` (code) for assignments. A tensor comprises a value node, a gradient node, *forward* assignments and *backprop* assignments. Tensor operations notation fetaures *parameter punning* (strings become let-bindings of tensors) and inline output dimensions specification. Full example of a Multi Layer Perceptron with 2 hidden layers and Rectified Linear Unit non-linearity `(?/)`, defining tensors `b1`, `w1`, `b2`, `w2`, `b3`, `w3`, and a tensor-returning function `mlp`:

```
let%op mlp x =
  "b3" + ("w3" * ?/("b2" hid_dim + ("w2" * ?/("b1" hid_dim + ("w1" * x)))))
```

Code notation for a Stochastic Gradient Descent update for a single parameter `p`, where `learning_rate` is a tensor, e.g. can undergo rate decay, `(!.)` embeds a float as a tensor:[1]

```
let sgd_one ~learning_rate ~momentum ~weight_decay ~nesterov p =
  [%cd "pg" =: p.grad + (!.weight_decay *. p);
       if Float.(momentum > 0.0) then (
         "b" =: (!.momentum *. b) + pg;
         if nesterov then pg =+ !.momentum *. b else pg =: b);
       p =- learning_rate *. pg]
```

Once the user collects the assignments of a desired routine, they are translated to a C language-like representation, interpreted to decide which tensor nodes should be *virtual*, computations of virtual tensor nodes (e.g. forward for value nodes and backprop for gradient nodes) are inlined, the code is simplified wrt. mathematical identies, and passed to a backend that the user selected.

3 Code execution: backends, devices, synchronization

In OCANNL, a tensor node is an "identity" shared across the "frontend" (OCaml runtime aka. *host*), backends and their devices – the same tensor node can correspond to arrays of numbers in multiple places. A tensor node can be *merged* – pointwise reduced – across instances. Tensor nodes can be:

- *Virtual* – without corresponding arrays and absent from the generically optimized code.
- *Local* – an array is cached for the duration of a computation but not persisted across calls to compiled functions.
- *On-device* – an array is stored on the devices that compute with it and persisted across function calls. It is available for merging across devices (for devices that support merging / P2P), but not for visualization or storing to disk.
- *Hosted* – the array is stored in a globally addressable memory, in addition to on devices where it is computed with (or as part of one of them, if "hosting on device", or only on the host and not on devices, for some backends).

Tensor nodes have each a numeric precision – OCANNL supports mixed precision computing.

A backend may optionally compile a routine in a relocatable way, to save on compilation times, and later link it to particular device contexts. Currently, OCANNL assumes that devices have queues of length 1: scheduling a new task blocks till completion of the old task on the device. CPU backends in OCANNL offer CPU cores as separate devices. OCANNL offers a round-robin scheduler for multi-device data parallel training, with "logarithmic" merging of gradients.

OCANNL is integrated with the package ppx_minidebug, including tracing device execution.

OCANNL was inspired by Andrej Karpathy's micrograd, motivated by projects tinygrad in Python, and very recently Luminal in Rust. We will also study the project llm.c as the "gold standard" for optimized compilation results. Functionality-wise, OCANNL has barely started: as of May 2024, future work include providing GPU parallelization, a neural networks toolbox, a more convenient and powerful scheduling. The design of OCANNL might still significantly evolve.

1. Algorithm from https://github.com/tinygrad/tinygrad