

CHAPTER 6

Computation Graph

A computation graph is a basic theoretical tool that underlies modern deep learning libraries. It is also an important component in Owl. This chapter first gives a bird’s-eye view on the computation graph in Owl and its importance in computing. We then demonstrate how to use it in Owl with some examples. Then we will continue to cover the design and implementation details of the computation graph module and how it is fitted into Owl’s functor stack.

6.1 The Definition of Computation Graph

As a functional programmer, it is basic knowledge that a function takes an input and then produces an output. The input of a function can be the output of another function which then creates dependency. If we view a function as one node in a graph, and its input and output as incoming and outgoing links to other functions, respectively, as the computation continues, these functions are chained together to form a *directed acyclic graph* (DAG). Such a DAG is often referred to as a computation graph.

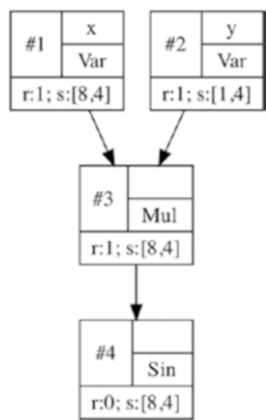


Figure 6-1. Computation graph of a simple function: $\sin(x*y)$

Figure 6-1 shows an example graph for calculating function $\sin(x * y)$.¹ The computation graph contains several pieces of information which are essential for debugging the applications. These information include the node index, operation type, reference counter, shapes of data, etc. For example, in Figure 6-1 the row vector y of shape $[1; 4]$ is broadcast on the matrix x of shape $[8; 4]$ in the `Mul` operation.

Dynamic Graph and Static Graph

The computation graph can be either implicitly constructed or explicitly declared in the code. Implicit graph construction is often achieved by operator overloading and a graph is constructed during the runtime, while explicit declaration may use *domain-specific languages* (DSL) to construct a graph during the compilation phase with a fix structure. The two methods lead to two different kinds of computation graphs: *dynamic graph* and *static graph*; each has its own pros and cons.

A dynamic graph is constructed during the runtime. Due to *operator overloading*, its construction can be naturally blended with a language's native constructs such as `if ... else ...` and `for` loops. This renders greatest flexibility and expressiveness. By using a dynamic computation graph, users are even free to construct a different network for each training sample. On the other hand, a static graph needs to be declared using a specific DSL, which tends to have a steeper learning curve. It is defined only once in training. Because the structure of a graph is already known during the compilation phase, there is a great space for optimization. However, it is sometimes very difficult to use static graphs to express conditions and loops when using with native code together.

The flexibility of a dynamic graph comes at the price of lower performance. Facebook's PyTorch and Google's TensorFlow are the typical examples of dynamic and static graphs, respectively. Many programmers need to make a choice between these two different types. A common practice is "using PyTorch at home and using TensorFlow in the company." In other words, PyTorch is preferred for prototyping, and TensorFlow is an ideal option for production use.²

Owl does something slightly different from these two in order to get the best parts of both worlds. Owl achieves this by converting a dynamic graph into a static one during

¹ This figure is generated with the tool provided by the `CGraph` module in Owl, which we will discuss in detail in this chapter.

² In Sep. 2019, TensorFlow rolled out version 2.0. Starting from this version, TensorFlow uses eager execution by default, which aims to be easier for users to get started with.

the runtime. The motivation is based on an observation: in many cases, a computation graph is continuously reevaluated after its construction. This is especially true for those iterative optimization algorithms. Thus, we only update some inputs of the graph in each iteration.

If we know that the graph structure remains the same in every iteration, rather than reconstructing it all the time, we can convert it into a static graph before the iterative evaluation. This is exactly what Owl does. By doing so, the programmer can enjoy the flexibility offered by the dynamic graph construction with operator overloading and, at the same time, still achieve the best performance from a static graph.

Compared to TensorFlow, the time overhead for the graph conversion and optimization is deferred to the runtime in Owl. Owl uses the *just-in-time compilation* (JIT) that is performed during the execution of a program. Since the graph compilation takes place in runtime, the JIT compiler can utilize dynamic runtime information and enable better optimization. You may worry about the performance and wonder if it is going to slow down your fancy DNN application. The fact is, even for large and complex graphs, this JIT compilation and optimization are often quite fast.

For example, in an LSTM network constructed using Owl,³ there are 15,105 nodes and 21,335 edges. Owl is able to compile the graph within 230ms and then optimize its structure within 210ms. The optimized graph contains only 8224 nodes and 14,444 edges and runs much faster. Note that you only need to do it once before training. For smaller networks, it often just takes several milliseconds. The CGraph module has implemented various graph optimization techniques to achieve this, which will be discussed in Section 6.3.

Technically, JIT is very straightforward to implement in Owl's architecture. Given a deep neural network, Owl first runs both forward pass and backward pass. Because of the computation graph, the calculation becomes symbolic, and we can obtain the complete computation graph to calculate the loss and gradients of a neural network. It then passes this static graph to the optimization engine.

³Code example: Constructing an LSTM network using Owl. URL: https://github.com/owlbarn/owl/blob/master/examples/lazy_lstm.ml

Significance in Computing

Now that you know the basic ideas of a computation graph, you may ask why it matters. Actually, a computation graph plays a core role in any machine learning framework. Both TensorFlow [1] and PyTorch [42], the most popular deep learning libraries, use a computation graph as the central data structure. A computation graph makes many things a lot easier. Here is an incomplete list of its potential benefits:

- Simulating lazy evaluation in a language with eager evaluation
- Incremental computation (a.k.a. self-adjusting computation)
- Reducing computation complexity by optimizing the structure of a graph
- Reducing memory management overhead by preallocating space
- Reducing memory footprint by reusing allocated memory space
- Natural support for parallel and distributed computing
- Natural support for heterogeneous computing
- Natural support for symbolic maths

Some of the benefits are very obvious. Memory usage can certainly be optimized if the graph structure is fixed and the input shapes are known beforehand. One optimization is reusing previously allocated memory, which is especially useful for those applications involving large ndarray calculations. In fact, this optimization can also be performed by a compiler by tracking the reference number of allocated memory, a technique referred to as *linear types* [50]. Some may appear less obvious at first glance. For example, we can decompose a computation graph into multiple independent *subgraphs*, and each can be evaluated in parallel on different cores or even computers. Maintaining the graph structure also improves fault tolerance, by providing natural support for rollback mechanisms.

The computation graph provides a way to abstract the flow of computations; therefore, it is able to bridge the high-level applications and low-level machinery of various hardware devices. This is why it has natural support for heterogeneous computing.

The computation graph has more profound implications on the scalability and security of scientific computing systems. Because the memory allocated for each node is mutable, the algorithmic differentiation becomes more scalable when evaluating large and complex graphs. At the same time, mutable transformation is handled by Owl internally, so programmers can still write safe functional code.

6.2 Applications Inside the Computing System

Before diving into the details of the design of the computation graph module, in this section let's first show some examples of using the CGraph module and how a computation can be transformed into lazy evaluation.

Basic Numerical Operations

Let's start with a simple operation that adds up one ndarray and one scalar. Normally, with the Narray module, what we do is

```
module N = Dense.Ndarray.D

let x = N.ones [|2;2|];;
let y = 2.;;
let g = N.add_scalar x y;;
```

Now, let's make this function into a computation graph which can be lazy evaluated by CGraph.

```
module N = Owl_computation_cpu_engine.Make
  (Owl_algodiff_primal_ops.D)
```

The Make function here is actually a *functor*. For those who are not familiar with the idea of functor, it is a powerful tool in OCaml to build generic code and structure large-scale systems. To put into plain words, a functor is a function that creates modules from modules. As we will explain in Section 6.3, the computation graph is designed as a *functor stack*. Different aspects of the computation graph, such as memory management and graph optimization, are added into the CGraph by creating a new module based

on an existing one, layer by layer. So far, it suffices to know that the functor creates a module `N`, which provides exactly the same ndarray operations, except that all the operations are conducted on *symbols* which represent ndarray instead of real objects allocated in memory.

```
let x = N.var_arr ~shape:[2;2] "x";;
let y = N.var_elt "y";;
let g = N.add_scalar x y;;
```

Next, we define two variables. The first `x` is an ndarray (`arr`), and `y` is a scalar (`elt`). At this point, we only define these two as placeholders with no real data. That is to say, we do not care about what specific ndarray or scalar these two variables are. Then we use the `add_scalar` function to get another lazy evaluated ndarray `g`. That finishes that lazy calculation. So far, we only know that `g` is calculated by adding `x` and `y`, but have no idea what their values are. To get the value of the lazy expression `g`, we need to first assign values to `x` and `y`:

```
let x_val = Dense.Ndarray.D.ones [2;2];;
let y_val = 2.;;
let _ = N.assign_arr x x_val;;
let _ = N.assign_elt y y_val;;
```

Here, `x` is assigned a double-precision ndarray of 1s, and `y` is float number 2. Note the two different assignment methods for ndarray and scalar. Finally, we can evaluate the ndarray `g`:

```
# N.eval_arr [|g|]
- : unit = ()

# N.unpack_arr g
- : Owl_algodiff_primal_ops.D.arr =
C0 C1
R0 3 3
R1 3 3
```

The `eval_arr` evaluates the whole graph but does not return the result. To extract the calculation result, we need to use the `unpack_arr` or `unpack_elt` function. The result is a 2x2 ndarray, the values of which are all 3s, just as expected. So where does the calculation happen? Remember that the `CGraph` module `N` is built based on the double-precision

type `Owl_algodiff_primal_ops` module. As we have explained in Chapter 3, this module is actually an `Ndarray.D` module, with some extra matrix and linear algebra operations. Therefore, in this example, `g` is calculated using C-based `ndarray` calculation. If we switch the base module from `Ndarray` to `Owl_base_ndarray`, the calculation is then performed using native OCaml.

Algorithmic Differentiation with CGraph

In real applications, we often need to deal with CGraphs that are constructed in the algorithmic differentiation process. Here is an example of using the dense `ndarray` module to compute the gradients of a function:

```
include Owl_algodiff_generic.Make
  (Owl_algodiff_primal_ops.D)

let f x y = Maths.((x * sin (x + x) + ((pack_float 1.) * sqrt x)
  / (pack_float 7.)) * (relu y) |> sum')

let x = Dense.Ndarray.D.ones [|2;2|] |> pack_arr
let y = pack_elt 2.
let z = (grad (f x)) y |> unpack_elt
```

Based on the chain rule, the `Algodiff` module automatically constructs a graph that computes the gradient of input function `f`. The result is contained in scalar `z`. However, the graph is constructed internally, but sometimes we need to have access to this graph and apply optimizations. Obviously, it is extremely difficult for the users to manually construct the computation graph that computes the gradient of the function `f`. Note that the `Algodiff` module is also built using functors. Its base module follows the `Ndarray` interface. By changing it from the `Ndarray` to `CGraph` module, we can make `z` to be a computation graph instead of a scalar value, as the following code snippet shows:

```
module G = Owl_computation_cpu_engine.Make
  (Owl_algodiff_primal_ops.D)
include Owl_algodiff_generic.Make (G)

let f x y =
  Maths.((x * sin (x + x) + ((pack_float 1.) *
    sqrt x) / (pack_float 7.)) * (relu y) |> sum')
```

```
let x = G.var_arr ~shape:[|2;2|] "x" |> pack_arr
let y = G.var_elt "y" |> pack_elt
let z = (grad (f x)) y
```

Most of the code stay unchanged. Notice how the CGraph module is treated as an alternative to the Nddarray module in building the AD module, since they follow the same set of interfaces required by the Algodiff module of its base module. They decide if the AD module uses normal or lazy evaluation. By executing this piece of code, the result `z` contains a computation graph constructed by the backward propagation pass in performing algorithmic differentiation.

The next thing we need to do is to assign values to inputs and evaluate `z`. That requires building a graph based on the input and output, as shown by the following code:

```
let inputs = [|
unpack_arr x |> G.arr_to_node;
unpack_elt y |> G.elt_to_node
|]
let outputs = [| unpack_elt z |> G.elt_to_node |]
let g = G.make_graph inputs outputs "graph"
```

To build a graph, we need to specify the input and output *nodes*. It might be a bit confusing, since there are two layers of packing and unpacking: the first from the AD node to the CGraph element and the second from the CGraph element to ndarray or scalar. We need `AD.unpack_arr` and `AD.unpack_elt` to unwrap AD type data (ndarray and scalar) into CGraph ndarray and scalar values. And then, to build the explicit computation graph, we need to use the `G.arr_to_node` and `G.elt_to_node` functions to make them into graph nodes first. Finally, an explicit computation graph can be built with the `make_graph` function.

After constructing the graph `g`, we can then assign real data values to the computation graph. Note that we need to first unpack the Algodiff values to CGraph values before assignment:

```
let x_val = Dense.Nddarray.D.ones [|2;2|];;
let y_val = 2.;;
let _ = G.assign_arr (unpack_arr x) x_val;;
let _ = G.assign_elt (unpack_elt y) y_val;;
```

Finally, we can evaluate the whole graph by simply calling

```
G.eval_graph g;;
```

Since the whole graph is evaluated, the output ndarray `z` is also evaluated. We can first unpack it from the `Algodiff` value into normal `CGraph` ndarray and then get its value by another layer of unpacking:

```
# unpack_elt z |> G.unpack_elt
- : float = 4.20861827873129801
```

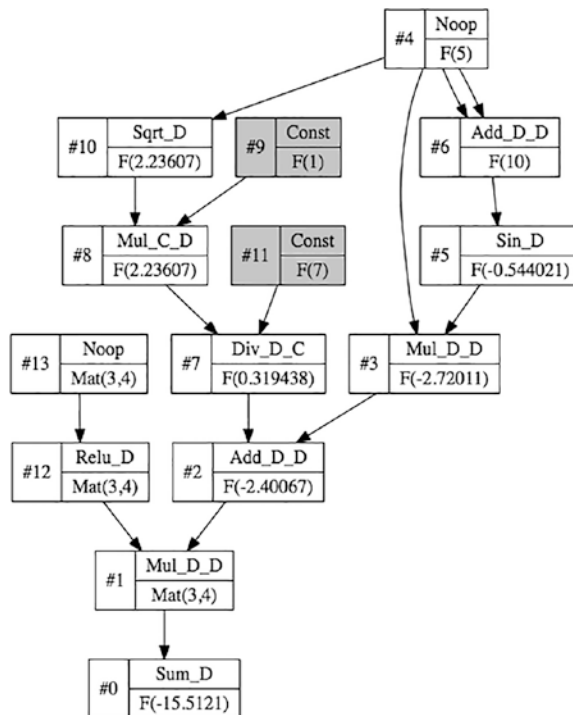


Figure 6-2. *Computation graph of a simple math function*

You might be wondering why bother to build the graph through all these layers of packing and unpacking when we can directly evaluate the value `z`. One main reason is to enable various optimizations on the graph before executing it, as we will explain in the following sections. Another reason is that evaluation is not always the target. For example, we often need to visualize the generated computation graph. The computation graph is very helpful in both debugging and understanding the characteristics of your numerical computations. Owl provides the `graph_to_dot` function to facilitate you in

generating computation graphs. It converts the computation graph into a [dot](#) format string. The dot file can be visualized with tools such as [Graphviz](#). For example, the following code generates a dot file for the graph we have constructed in this example, and this graph is shown in [Figure 6-2](#).

```
let s = G.graph_to_dot g
let _ = Owl_io.write_file "cgraph.dot" s
```

Deep Neural Network

Since the optimization and neural network modules are built on the algorithmic differentiation module, they can also benefit from the power of the computation graph. Suppose we have a network built from CGraph-based neural network `nn`, we can then use the forward and backward functions to get the forward inference and backward propagation computation graphs from the neural network graph module, with the CGraph ndarray variable. Actually, for ease of access, Owl has provided another functor to build the neural network module based on the CGraph module:

```
module CPU_Engine = Owl_computation_cpu_engine.Make
  (Owl_algodiff_primal_ops.S)
module CGCompiler = Owl_neural_compiler.Make (CPU_Engine)

open CGCompiler.Neural
open CGCompiler.Neural.Graph
open CGCompiler.Neural.Algodiff

let make_network input_shape =
  input input_shape
  |> lambda (fun x -> Maths.(x / pack_flt 256.))
  |> conv2d [|5;5;1;32|] [|1;1|] ~act_typ:Activation.Relu
  |> max_pool2d [|2;2|] [|2;2|]
  |> dropout 0.1
  |> fully_connected 1024 ~act_typ:Activation.Relu
  |> linear 10 ~act_typ:Activation.(Softmax 1)
  |> get_network ~name:"mnist"
```

The CGraph-built neural network module does not require any change of code in building the CNN except for the headers. To build a normal neural network, we use the `Neural` module, and now we only need to change that to the `CGCompiler`. `Neural` module. Here, the `owl_neural_compiler` functor compiles a DNN definition and training configuration into a device-dependent static graph. As its output, the `CGCompiler` is a computation graph-powered neural network compiler module. `CGCompiler` also provides training functions. Note that the data requires proper packing around original `ndarray`.

```
let pack x = CGCompiler.Engine.pack_arr x
    |> Algodiff.pack_arr

let train network =
  let x, _, y = Dataset.load_mnist_train_data_arr () in
  let x = pack x in
  let y = pack y in
  CGCompiler.train network x y
```

Similarly, the inference can be done with the `CGCompiler.model` function. To make the existing DNN program into a lazy evaluation version, all we need to do is to update the header and use packing/unpacking properly for the data.

One of the key performance improvements CGraph has on the neural network lies in its ability of graph and memory optimization. To motivate you to understand more about the design and optimization of the CGraph module, here is an example. Let's train a LeNet-like DNN based on the MNIST dataset, using the normal version [mnist_cnn.ml](#) and the CGraph-powered version [lazy_mnist.ml](#).⁴ Similar to the preceding example code, both scripts train the same convolution neural network in 60 iterations. In one of our evaluations on a normal laptop, `mnist_cnn.ml` takes 30s to finish and approximately consumes 4GB memory, while `lazy_mnist.ml` only takes 5s and consumes about 0.75GB. This performance improvement is astounding. If these numbers make you interested in knowing how the magic happens, please keep reading the next section. We will unveil the underlying mechanism of Owl's computation graph.

⁴Both code snippets are available from the source code of Owl.

6.3 Design of Computation Graph Module

Owl implements the computation graph in a very unique and interesting way. Let’s first see several principles we followed in designing and developing this module:

- Nonintrusive, the original functor stack should work as it was
- Transparent to the programmers as much as possible
- Support both eager and lazy evaluations
- Flexible enough for future extension on other devices

The computation graph is implemented in a self-contained stack. We have devised a way to “inject” it into Owl’s original functor stack. If it sounds too abstract, please have a look at the final product in Figure 6-3.

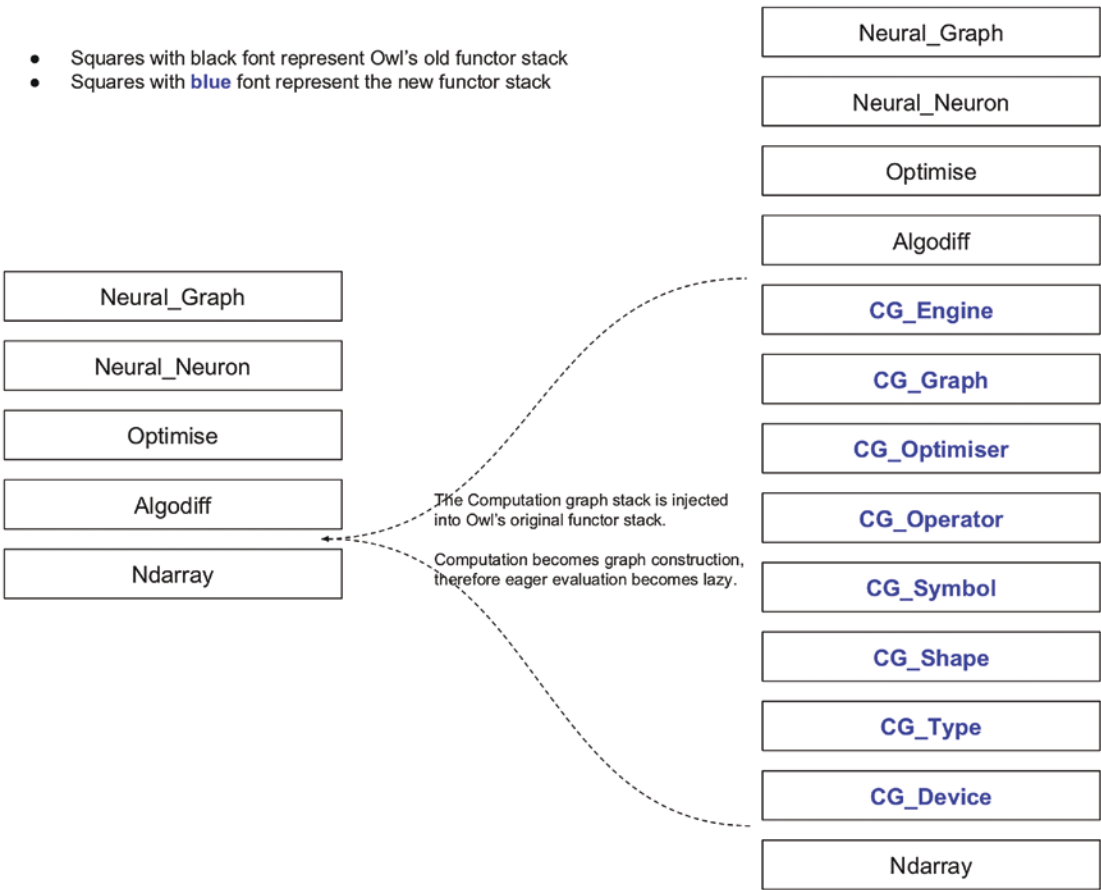


Figure 6-3. Computation graph functor stack in Owl

The left figure shows part of Owl's original functor stack, and the right one shows how the current one looks like after injection. In the very initial design, Nddarray implements a set of fundamental n-dimensional array operations, then Algodiff defines abstract mathematical operations for differentiation, finally the Optimise engine glues low-level maths with high-level deep neural network applications. The whole stack is parameterized by the number type abstraction in Nddarray:

- Nddarray: Provides number type abstraction and implements the fundamental numerical operations
- Algodiff: Implements algorithmic differentiation
- Optimise: Uses the derivative information to build an optimization engine
- Neural_Neuron: Implements various kinds of neuron functions which can be optimized
- Neural_Graph: Connects neurons together to form a network so that we can train a useful model

Based on this architecture, the whole functor stack of the computation graph can be inserted between the Nddarray and Algodiff modules. The design principle is that *the functor stack of a numerical system should be parameterized by both number type and device type*. The number type provides data representation (real or complex, single or double, row-based or column-based layout, etc.) which decides how a math construct should be built and operated. The device type provides hardware representation (CPU, GPU, FPGA, etc.) which decides how the computation should be performed on a specific device.

The following list summarizes the functionality of each functor in the CGraph stack. The order and naming of these functors can give you a rough understanding about how it is designed, as follows:

- Device: Device abstraction contains device-dependent types and functions.
- Type: Type definition of various (mathematical) operations.
- Shape: Provides the *shape inference* function in the graph.
- Symbol: Provides various general functions to manipulate symbols.

- **Operator:** Implements math operators (+, -, *, /, etc.) which decide how the symbols should be connected to form a graph.
- **Optimiser:** Optimizes the structure of a given graph by searching and optimizing various patterns.
- **Graph:** Manipulates computation graphs at a high level, for example, visualization, connecting inputs and outputs.
- **Engine:** Evaluates a computation graph on a specific device.

Simply put, the injected computation graph stack provides an abstraction layer similar to symbolic maths. Without injecting the computation graph, the OCaml returns 2 if you calculate 1+1; now it returns a graph of several nodes. Thus, the original eager evaluation becomes symbolic operation and pure graph manipulation, and the graph can be lazily evaluated.

The shape inference functionality is able to infer the data shape of every node in a graph from its input. This allows Owl to calculate how much memory is required to evaluate the graph and preallocate this space. Owl can further track the reference number of each function node and reuse the allocated memory as much as possible, which reduces both memory footprint and garbage collector (GC) overhead, significantly improving the computation speed.

The engine functor sits on top of the stack. This is where a computation graph finally gets executed. The engine functor contains two submodules, one for initializing the graph and the other for evaluating the graph. We can try the following snippet in an OCaml REPL such as `utop`. Both snippets generate a module for DNN applications; the difference is that the first one uses the old stack, whereas the second one uses the new stack with the computation graph.

```
module M =
  Owl_neural_generic.Flatten (
    Owl_neural_graph.Make (
      Owl_neural_neuron.Make (
        Owl_optimise_generic.Make (
          Owl_algodiff_generic.Make (
            Dense.Ndarray.S))))));;
```

As to the new stack that contains computation graph functors, we can see it is indeed much deeper.

```
module M =
  Owl_neural_generic.Flatten (
    Owl_neural_graph.Make (
      Owl_neural_neuron.Make (
        Owl_optimise_generic.Make (
          Owl_algodiff_generic.Make (
            Owl_computation_engine.Flatten (
              Owl_computation_cpu_engine.Make_Nested (
                Owl_computation_graph.Make (
                  Owl_computation_optimiser.Make (
                    Owl_computation_operator.Make (
                      Owl_computation_symbol.Make (
                        Owl_computation_shape.Make (
                          Owl_computation_type.Make (
                            Owl_computation_cpu_device.Make (
                              Dense.Ndarray.S)))))))))))));;
```

We have introduced the different components of the computation graph module. Next, we will dive deep into the implementation of core functionalities of this module: the construction of a graph, the optimization of the graph structure, the evaluation, the memory optimization in execution, etc.

Computing Device

A computation graph is an abstract construct to express the logic of a function. To calculate the outcome of a function, computation graphs need to be evaluated on a physical device. The device can be anything as long as it has the capability to perform numerical operations, such as the CPU, GPU, etc. To extend Owl on a new device, we only need to create a new device module and define how the basic operations can be performed on this device. Because a majority of the CGraph module is device independent, the device layer becomes very lightweight, which further makes Owl very easy to extend.

The following functor defines a CPU device. The functor's input is the type of data which will be manipulated on the device. In our case, they are either ndarray or scalar values. This makes perfect sense if you are familiar with computer architecture. The data are often stored and processed differently on devices of different architectures. Making a new device is simply creating an abstract record type in OCaml. The other two functions are for packing and unpacking data into the types which a device can process.

```
module Make (A : Narray_Mutable) = struct
  module A = A

  type device =
    { device_type : device_type
    ; initialised : bool
    }

  type value =
    | ArrVal of A.arr
    | EltVal of A.elm

  let make_device () = { device_type = CPU; initialised = false }

  let arr_to_value x = ArrVal x

  let value_to_arr = function
    | ArrVal x -> x
    | _         -> failwith "Owl_computation_device: value_to_arr"

  ...

end
```

For example, OpenCL is a framework for developing cross-platform programs; these programs can execute on heterogeneous platforms consisting of the CPU, GPU, DSP, FPGA, and other processors or hardware accelerators. The following code defines an OpenCL-compatible device. Compared to the CPU device, the most noticeable difference on the OpenCL device is that the values are represented very differently. The data can be stored either on the memory attached to the CPU or the memory attached to the GPU. Quite often, the data has to be transferred between the two disjoint memory systems for performance considerations. Moreover, the computation performed on a

GPU is defined by kernels which are written in C-like DSL. Different computing units communicate through events.

```

module Make (A : Ndarray_Mutable) = struct
  module A = A

  type device =
    { device_type : device_type
    ; initialised : bool
    }

  type cpu_mem = A.arr

  type value =
    { mutable cpu_mem : cpu_mem array
    ; mutable gpu_mem : cl_mem array
    ; mutable kernel : cl_kernel array
    ; mutable events : cl_event array
    }

  let make_device () = { device_type = OpenCL; initialised = false }

  let arr_to_value x =
    let cpu_mem = [| x |] in
    let gpu_mem = [| |] in
    let kernel = [| |] in
    let events = [| |] in
    { cpu_mem; gpu_mem; kernel; events }

  let value_to_arr x =
    if Array.length x.cpu_mem > 0
    then x.cpu_mem.(0)
    else failwith "value_to_arr: not evaluated yet"

  ...
end

```

There are four attributes associated with a value regarding its storage, computation, and communication on an OpenCL device, that is, CPU memory and GPU memory for storage, kernel for computation, and event for communication between computing units.

Types of Operation

The `Owl_computation_type` functor takes a device module as its input, then specifies all the possible operations on the given device. Whenever we want to extend the set of operations, we need to add the corresponding constructor of the new operation to the sum type `op`. The current set of operations covers a wide range of unary and binary numerical functions, such as `Abs`, `Neg`, `Add`, as well as functions for neural networks such as `MaxPool3d`.

```
module Make (Device : Owl_types_computation_device.Sig) = struct
  module Device = Device
  open Device

  type state =
    | Valid
    | Invalid

  type t = attr Owl_graph.node

  and block =
    { size : int
    ; block_id : int
    ; mutable active : t option
    ; mutable memory : value
    ; mutable nodes : t list
    }

  and attr =
    { mutable op : op
    ; mutable freeze : bool
    ; mutable reuse : bool
    ; mutable state : state
```

```

    ; mutable shape : int array option array
    ; mutable value : value array
    ; mutable block : block array option
  }

  and arr = Arr of t
  and elt = Elt of t

  and op =
    | Noop
    | Var
    | Const
    | Abs
    | Neg
    ...

end

```

Here, `attr` is a record type containing the properties of an operation. These properties will be utilized to initialize a graph and optimize its memory usage and evaluation performance. For example, the `reuse` field specifies whether the memory associated with an operation can be shared with other operations. The `block` type stores the memory and all the operations which are sharing this memory.

Shape Inference

The shape of data might change while traveling through different nodes in a computation graph. The shape information is very valuable for debugging and optimization purposes. When all the inputs of a given function are known, the shape of the outcome can be decided; hence, the shape information of a computation graph becomes available. The `Owl_computation_shape` functor is created for automating shape inference. The core function of this functor is `infer_shape` which calls the corresponding shape inference function of an operator using pattern matching.

```

module Make (Type : Owl_computation_type_sig.Sig) = struct
  module Type = Type

  let infer_shape operator args =

```

```

let input_shapes = Array.map (fun a -> (Owl_graph.attr
a).shape) args in
match operator with
| Noop -> _infer_shape_01 input_shapes
| Create shape -> [| Some shape |]
...
| Scalar_Add -> _infer_shape_00 input_shapes
| Scalar_Sub -> _infer_shape_00 input_shapes
...
| Abs -> _infer_shape_01 input_shapes
| Neg -> _infer_shape_01 input_shapes
...
| Add -> _infer_shape_03 input_shapes
| Sub -> _infer_shape_03 input_shapes
...
| Conv1d (padding, stride) -> _infer_shape_11 input_shapes
padding stride
| Conv2d (padding, stride) -> _infer_shape_12 input_shapes
padding stride
| Conv3d (padding, stride) -> _infer_shape_13 input_shapes
padding stride
...
end

```

There are over 30 shape inference functions defined. We can take a closer look at those frequently used ones. For example, scalar operators such as `Scalar_Add` do not require shape information, so its inference function only returns an empty array. The reason for using an array of arrays as the returned type is because an operator might produce multiple ndarrays as outputs.

```
let _infer_shape_00 _input_shapes = [| Some [|] |]
```

The `_infer_shape_01` pattern is defined for a *unary operator* with a single input and a single output such as `Abs`. These operators do not change the shape of the data; the input and output have exactly the same shape. Thus, the inference function simply returns the shape of the input as the shape of the output.

```
let _infer_shape_01 input_shapes =
  match input_shapes.(0).(0) with
  | Some s -> [| Some Array.(copy s) |]
  | None    -> [| None   |]
```

If the inputs have the same shape, *binary operators* like `Add` will produce an output of the same shape. However, if the shapes of inputs are different, broadcasting must be taken into account to correctly calculate the output shape. Luckily, the broadcasting rules can be codified easily from that used in the `Ndarray` module for shape inference purposes.

```
let _infer_shape_03 input_shapes =
  let s0 = input_shapes.(0).(0) in
  let s1 = input_shapes.(1).(0) in
  match s0, s1 with
  | Some s0, Some s1 -> [| Some Owl_utils_infer_shape.(broadcast1 s0 s1) |]
  | _, _             -> [| None   |]
```

We do not cover all the shape inference patterns here. If you are interested, the readers are encouraged to read the source code of `Owl` to learn more about them. When the shape of a graph is known, we can exploit this information to calculate the total memory consumption, discover optimization opportunities, validate the consistency of inputs and outputs, identify potential bugs, etc.

Creating and Linking Nodes

Now we come to a higher level of abstraction: the graph itself. For inputs, outputs, and operators in a computation graph, we need to create their corresponding nodes. Because numerical operations are composable, the output of a function can become the inputs of other functions. We also need to link the nodes according to their input and output dependencies. The `Owl_computation_symbol` and `Owl_computation_operator` functors are developed for this purpose. The input of `Owl_computation_symbol` is a shape module, so the functor can infer the shape automatically while constructing a graph. The most basic functions like `arr_to_node` and `node_to_arr` pack and unpack ndarrays to and from a `CGraph` node.

```

module Make (Shape : Owl_computation_shape_sig.Sig) = struct
  module Shape = Shape

  let node_to_arr x = Arr x

  let arr_to_node = function
    | Arr x -> x

  let node_to_elt x = Elt x

  let elt_to_node = function
    | Elt x -> x

  ...
end

```

The general function for creating a node is `make_node`. The function utilizes the node type defined in the `Owl_graph` module which provides a comprehensive set of functions to manipulate a graph.

```

let make_node ?name ?value ?shape ?freeze ?reuse ?state op =
  let shape =
    match shape with
    | Some s -> s
    | None    -> [| None |]
  in
  let state =
    match state with
    | Some s -> s
    | None    -> Invalid
  in
  let reuse =
    match reuse with
    | Some s -> s
    | None    -> true
  in

```

```

let freeze =
  match freeze with
  | Some s -> s
  | None   -> false
in
let value =
  match value with
  | Some v -> v
  | None   -> [||]
in
let attr = { op; freeze; reuse; state; shape; value; block = None } in
let node = Owl_graph.node ?name attr in
if value <> [||] then make_value_block value.(0) node;
node

```

Because the `Owl_graph` module is designed for general-purpose graph manipulation such as construction, iteration, search, pruning, etc., the node properties are kept minimal and can be extended. The node type has a type parameter as you can see in its definition; therefore, we can attach related attributes of an operation to the node.

```

type 'a node =
{ mutable id : int
; mutable name : string
; mutable prev : 'a node array
; mutable next : 'a node array
; mutable attr : 'a
}

```

Inputs need to connect to an operator to produce an output. If the output becomes the input of another operator, the computations are chained together. `make_node` only creates nodes, while `make_then_connect` does the linking job. The `make_then_connect` function internally calls `make_node` first to create a child node and then connects the outputs of the parent nodes to the inputs of the child node. Special attention is required if there are duplication in parent nodes, for example, in the $y = x + x$ case.

```

let make_then_connect ?shape op parents =
  let shape =
    match shape with
    | Some s -> s
    | None    -> infer_shape op parents
  in
  let child = make_node ~shape op in
  connect_ancestors parents [| child |];
  let uniq_parents = Owl_utils_array.unique parents in
  Array.iter
    (fun parent ->
      if (attr parent).freeze = false
      then connect_descendants [| parent |] [| child |])
    uniq_parents;
  child

```

For simple creation functions of ndarray, such as `empty`, `zeros`, etc., `make_node` is sufficient because these functions do not require any parents to provide inputs except its own shape information.

```

module Make (Symbol : Owl_computation_symbol_sig.Sig) = struct
  module Symbol = Symbol

  let empty shape = make_node ~shape:[| Some shape |] (Empty shape)
                        |> node_to_arr

  let zeros shape = make_node ~shape:[| Some shape |] (Zeros shape)
                        |> node_to_arr

  ...
end

```

For unary operators which do require the output of a parent node, `make_then_connect` is called to connect the parent's output to the operator's input. The outputs of parent nodes are unpacked from the `arr` type, while the outputs of a child are packed back into the `arr` type.

```

let abs x = make_then_connect Abs [| arr_to_node x |] |> node_to_arr
let neg x = make_then_connect Neg [| arr_to_node x |] |> node_to_arr

```

Binary operators work in a similar way. The only difference is that the inputs are from two parents rather than one comparing to unary operators.

```
let add x y = make_then_connect Add [| arr_to_node x; arr_to_node y |]
    |> node_to_arr

let sub x y = make_then_connect Sub [| arr_to_node x; arr_to_node y |]
    |> node_to_arr

let mul x y = make_then_connect Mul [| arr_to_node x; arr_to_node y |]
    |> node_to_arr

let div x y = make_then_connect Div [| arr_to_node x; arr_to_node y |]
    |> node_to_arr
```

With these basic functions, we can construct very complicated computation graphs. Quite often, the underlying computation graph may appear more complicated than the actual function defined in code; neural network applications are good examples.

Optimization of Graph Structure

The Optimiser functor is in charge of graph structure manipulation. It searches for various *structural patterns* in a graph and then performs various optimizations, such as removing unnecessary computations, fusing computation nodes, etc. All the patterns are defined in the Owl_computation_optimiser functor.

Let us first look at the heart of this functor, `_optimise_term` function, as follows. This function traverses backward from leaves to their ancestors recursively, looking for certain patterns to optimize. The patterns which the functor can recognize are coded in various `pattern_*` functions which call each other recursively. The source code is organized so that it is very straightforward to plug in more patterns to extend the optimizer's capability.

```
module Make (Operator : Owl_computation_operator_sig.Sig) = struct
  module Operator = Operator

  let rec _optimise_term x =
    Owl_log.debug "optimise %s ..." (node_to_str x);
    if is_valid x = false
```

```

then (
  (match get_operator x with
  | Noop -> pattern_003 x
  | Empty_shape -> pattern_000 x
  | Zeros_shape -> pattern_000 x
  ...
  | Add -> pattern_001 x
  | Sub -> pattern_000 x
  | Mul -> pattern_019 x
  | Div -> pattern_007 x
  ...
  | Scalar_Add -> pattern_010 x
  | Scalar_Sub -> pattern_010 x m
  ...
  | Dot (_transa, _transb, _alpha, _beta) -> pattern_005 x
  | Fused_Adagrad (_rate, _eps) -> pattern_000 x
  | _ -> failwith "Owl_computation_optimiser:_optimise_term");
  validate x)
end

```

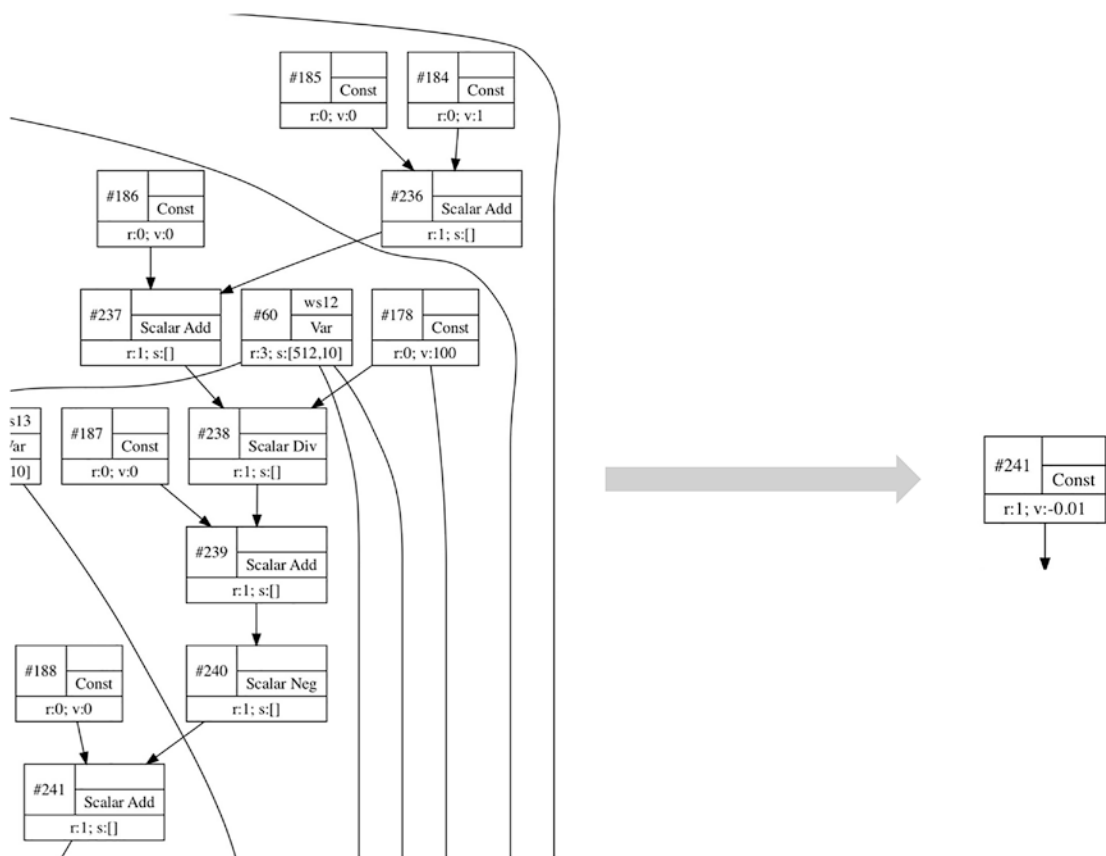


Figure 6-4. Optimization techniques in a computation graph: constant folding

In this part, we will explain three most commonly used graph optimization patterns: constant folding, operations fusing, and removing zeros. *Constant folding* is a very basic pattern to reduce graph size. In a computation graph, it is common to see that a lot of constants are involved. As a result, some subgraphs can be precalculated. Figure 6-4 shows such an example. In this subgraph, the nodes #241 depends on are either constants or operations on constants. Therefore, the value of node #241 is already decided. We can thus fold this subgraph into one single node before evaluating the whole graph.

From the definition of the `_optimise_term` function, we can see the `Scalar_Add` operator triggers the `pattern_010` function. This function first tries to optimize the parent nodes, and then it checks whether both parents are constants. If so, the function evaluates the expression based on the current operator, creates a new constant node for

the result, and removes the current node and its parents. By doing so, all the expressions which can be evaluated during this phase will be folded into a constant, which can save a lot of time during the graph evaluation phase.

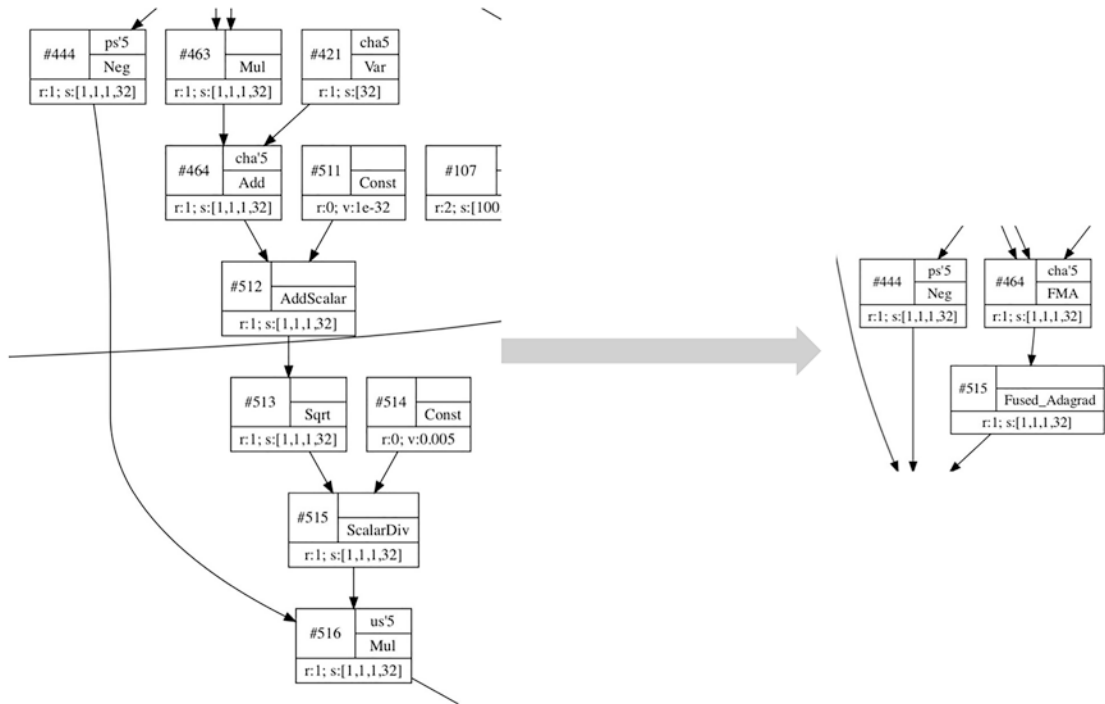


Figure 6-5. Optimization techniques in a computation graph: fusing operations

```
and pattern_010 x =
  let parents = parents x in
  let a = parents.(0) in
  let b = parents.(1) in
  _optimise_term a;
  _optimise_term b;
  match get_operator a, get_operator b with
  | Const, Const ->
    let a_val = node_to_elt a |> elt_to_float in
    let b_val = node_to_elt b |> elt_to_float in
    let c_val = pattern_011 (get_operator x) a_val b_val in
    set_parents x [||];
```

```

set_reuse x false;
set_operator x Const;
freeze x;
set_value x [| float_to_elt c_val |> unpack_elt |> elt_to_value |]
| _ -> ()

```

The next pattern, *fusing operations*, combines multiple operations into one, if applicable. For example, in Figure 6-5, nodes #421, #463, and #464 are fused into one fma node (i.e., *fused-multiply-add operation*). Owl also recognizes complicated patterns, for example, a pattern formed by nodes #511–#515 appears a lot in DNN training that uses the Adagrad (adaptive subgradient) training method. Fusing all these operations into one single operation can improve computing efficiency as well as numerical accuracy. Besides, this optimization also effectively reduces the round trips to the memory, which saves a lot of time when operating on large ndarrays.

In the source code, fusing FMA operation depends on the `pattern_004` function. The function first checks if the current operator is `Add`, then checks if one of the inputs is from the multiplication operator. If both conditions are satisfied, the pattern is identified. The `refnum` is a counter tracking how many times the output of an operator has been referred to by other expressions. If `refnum` is greater than one, we cannot fuse the operator because its output is used by another operator as input.

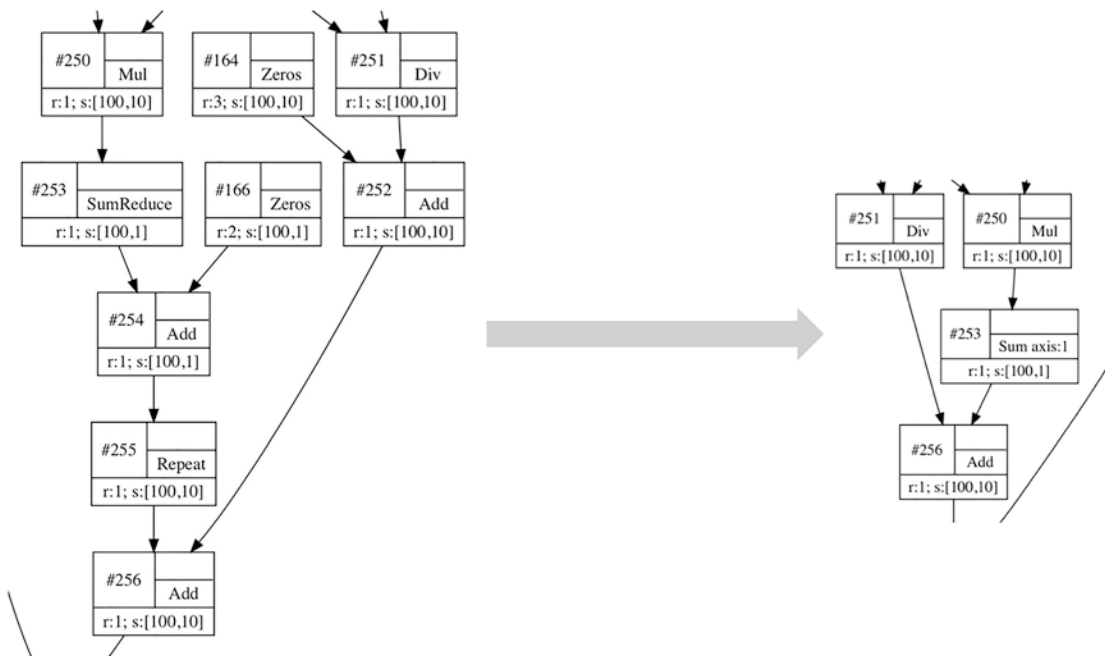


Figure 6-6. Optimization techniques in a computation graph: adding zero

```

and pattern_004 x =
  if get_operator x = Add
  then (
    let x_parents = parents x in
    let a = x_parents.(0) in
    let b = x_parents.(1) in
    if get_operator a = Mul && refnum a = 1
    then (
      let new_parents = Owl_utils_array.(parents a @ [| b |]) in
      set_parents x new_parents;
      replace_child a x;
      set_operator x FMA;
      remove_node a)
    else if get_operator b = Mul && refnum b = 1
    then (
      let new_parents = Owl_utils_array.(parents b @ [| a |]) in
      set_parents x new_parents;

```

```

    replace_child b x;
    set_operator x FMA;
    remove_node b))

```

Next, the *adding zero* pattern is trivial to see in the graph. If one node adds another zeros node, then the zeros node can be safely removed. In the example shown in Figure 6-6, nodes #164 and #166 are removed, and the others are folded. Moreover, node #255 for the repeat operation is also removed because the add operation already supports the broadcasting operation. Removing #255 can save some runtime memory in the evaluation.

The `pattern_002` function detects both $x + 0$ and $0 + x$ patterns. The implementation is intuitive. After an Add operator is identified, the function checks whether one of the inputs is zero. If so, the Zero node is removed, and the current Add operator is replaced with the Noop operator.

```

and pattern_002 x =
  let x_parents = parents x in
  let a = x_parents.(0) in
  let b = x_parents.(1) in
  if get_operator x = Add
  then (
    match get_operator a, get_operator b with
    | Zeros _, _ ->
      set_operator x Noop;
      remove_edge a x;
      _optimise_term x
    | _, Zeros _ ->
      set_operator x Noop;
      remove_edge b x;
      _optimise_term x
    | _, _ -> ())

```

There are also other patterns that focus on specific calculations, such as multiplication, division, repeat, sum-reduce, etc. Please refer to the source code if you are interested in them. To show how effectively the Optimiser works, we again

use the aforementioned LeNet-like CNN that is trained on the MNIST dataset. The original network has 201 nodes and 239 edges; after applying the graph optimization in Optimiser, the whole computation graph consists of only 103 nodes and 140 edges.

Optimizing a graph structure to improve evaluation performance is an advanced topic. But as you can see in the previous step-by-step illustration, advanced functionalities can be decomposed into a set of simple functions identifying specific patterns and optimizing locally using a typical divide-and-conquer approach. The graph optimization in TensorFlow follows a somewhat similar path. The computation graph in TensorFlow is first constructed using the Python frontend, and via a layer of the C API, this graph is converted to a format that the C++ backend can recognize. After that, the graph is optimized using various techniques, including common subexpression elimination, constant folding, removing identity nodes, removing dead nodes, etc. If you look at the source code of TensorFlow, this part of functionalities is taken care of by the common runtime module of its core engine.

Computation Engine

Finally, we have reached the top of the CGraph functor stack: the *computation engine*. Because a computation graph has to be evaluated on the hardware, each type of device must implement its own computing engine. The following code shows the engine for CPU devices. The core function `eval_gen` consists of two steps. The first step is to initialize the graph by calling `_init_terms`. The second step is to evaluate the graph by calling `_eval_terms`.

```
module Make_Nested (Graph : Owl_computation_graph_sig.Sig) = struct
  module Graph = Graph
  module CG_Init = Owl_computation_cpu_init.Make (Graph)
  module CG_Eval = Owl_computation_cpu_eval.Make (Graph)

  let eval_gen nodes =
    CG_Init._init_terms nodes;
    CG_Eval._eval_terms nodes

  let eval_elt xs = Array.map elt_to_node xs |> eval_gen
  let eval_arr xs = Array.map arr_to_node xs |> eval_gen
```

```

let eval_graph graph =
  Graph.invalidate_rvs graph;
  Graph.get_outputs graph |> eval_gen
end

```

For comparison, let us also create a loop at the computing engine for OpenCL devices. The functor structure of the OpenCL computing engine is almost the same except the `eval_gen` function.. The function has a bit more code because the procedure of setting up a computing environment is much more complicated on an *OpenCL-compatible device* than on a CPU device. The procedure consists of many steps including specifying context, accelerator, *command queue*, *kernel programs*, etc. The evaluation outputs also need to be explicitly copied from *GPU memory* to *CPU memory* for further processing.

```

let eval_gen dev_id nodes =
  let ctx = Owl_opengl_context.(get_opengl_ctx default) in
  let dev = Owl_opengl_context.(get_dev default dev_id) in
  let cmdq = Owl_opengl_context.(get_cmdq default dev) in
  let prog = Owl_opengl_context.(get_program default) in
  let param = ctx, cmdq, prog in

  CG_Init.init_nodes nodes param;
  Array.iter
    (fun y ->
      CG_Eval._eval_term y param;
      let y_val = (get_value y).(0) in
      CG_Eval.gpu_to_cpu_copy param y_val |> ignore)
    nodes;
  Owl_opengl_base.CommandQueue.finish cmdq

```

The `_eval_terms` function consists of many `_eval_map_*` functions to perform actual computation. Let us look at a simple one `_eval_map_00` for CPU devices. This function is for operators with a single input and a single output. The `_eval_map_00` simply applies the operator's function `f` to its input `x`, then returns the result.

```

and _eval_map_01 x f =
  _eval_terms (parents x);
  let inputs = Array.map (fun parent ->
    value_to_arr (get_value parent).(0)) (parents x) in
  let out = value_to_arr (get_value x).(0) in
  f ~out inputs

```

On the other hand, the similar function for OpenCL devices is more complicated. Because the computation takes place on an accelerator, we need to set up the command queue for communication and *event queue* for synchronizing computing units. We also need to specify the suitable kernels for computing logic. These kernels are compiled dynamically during the runtime and then copied to the computing units of an accelerator. When the output is finally ready, we must explicitly dispatch the event to notify the dependent.

```

and _eval_map_01 x param =
  Array.iter (fun parent -> _eval_term parent param) (parents x);
  let _, cmdq, _ = param in
  let kernel = (get_value x).(0).kernel.(0) in
  let items = [ node_numel x ] in
  let wait_for = aggregate_events (parents x) |> Array.to_list in
  let event = Owl_openc1_base.Kernel.enqueue_ndrange ~wait_for cmdq kernel
  1 items in
  Device.append_events (get_value x).(0) [| event |]

```

Programming a GPU is very much like programming a computer cluster. The gain of parallel computing comes with inevitable synchronization and communication overhead. Therefore, GPU computing only makes sense when the computation complexity is high enough to dwarf other overheads.

When offloading the computation to a GPU, we should avoid transmitting data back and forth between the host and the device memory, so eager evaluation is not ideal in this context because the performance will be throttled by copying. This is the gap between CPU computing and a language with eager evaluation. The computation graph essentially fills the gap between Owl and GPU computing simply because the laziness can be simulated now.

From an implementation perspective, we only need to write a new engine functor for GPU devices to evaluate a graph; all the others remain the same. Comparing to the CPU engine, the OpenCL engine maintains the memory allocated on both the host and the device for each node, copying only happens whenever it is necessary, and the allocated memory on the device is reused as much as possible.

6.4 Optimizing Memory Usage in Computation Graph

In the previous sections, we have introduced the CGraph stack. Before concluding this chapter, we would like to show optimizations we have made to reduce memory usage in the CGraph module. One principle we have been following during developing Owl is to always get driven by real-world applications. Besides the image recognition example, we have built an image segmentation application, a challenging and interesting use case for Owl. Seeking to push the performance of this application, we manage to further optimize the design of the CGraph module. We will present this deep neural network, Mask R-CNN, in Chapter 5. This section is mainly based on the work done by Pierre Vandenhove on Owl during his internship in the OCaml Labs [49].

The first issue after constructing the network, called *Mask R-CNN*, or MRCNN, in Owl was that its memory usage, in inference mode was huge. The network has over 400 layers. A reasonable input image size for this network is a 1024-pixel-wide square. To avoid reinitializing the network for every picture, it is a good practice to keep its input size fixed and to resize instead all the images to that size. Unfortunately, obtaining detections for one picture with such size required over 11GB of RAM, which was too much for a normal laptop. There is surely a big room for improvement.

We first try to apply the graph structure optimization we have mentioned in the previous section. The number of nodes of the Mask R-CNN network drops from 4095 to 3765, but its effect in memory reduction is limited. To this end, we need to add in the CGraph functor stack another important layer: memory management. Specifically, we need the ability to preallocate a memory space to each node, to decrease the overall memory consumption and reduce the garbage collector overhead. The key is to find the allocated memory block in the graph that is no longer required and assign it to other nodes that are in need.

The memory manipulation functionalities are implemented in the `Engine` functor in the stack. The key data structure is the `block` type we mentioned in Section 6.3. The `block` type is a record which maintains a list of nodes sharing the same memory. The initial strategy to allocate memory to a node u in Owl's computation graph module was simply to reuse the memory of a direct predecessor with the same output shape as u when that is possible. For example, if we add two `ndarrays` of the same shape, the output `ndarray` can reuse the memory block of one of them. This optimization decreases the memory consumption of Mask R-CNN from 11GB to 7GB. This 36% reduction looks quite impressive, but can we do even better?

To describe the process of allocating memory in a computation graph, it is interesting to first look at the *pebble game*, which was introduced in 1973 to explain register allocation [45]. The pebble game is played on a directed acyclic graph. Each node can store at most one pebble. The game begins with no pebble on any node. At each step, the player can do one of the following moves:

1. If a vertex v has no predecessor, the player can place a pebble on v .
2. If all predecessors of a vertex v are pebbled, the player can place a pebble on v or slide a pebble from one of its predecessors to v .
3. The player can remove any pebble from a vertex (and reuse that pebble later).

The goal of the game is to place a pebble at least once on some fixed output vertices of the graph. Figure 6-7 shows an example of an optimal pebbling strategy using the previous computation graph (gray nodes are pebbled), using moves 1 -> 2 -> 3 -> 1 -> 2 -> 2.

We assume that the goal is to pebble node 5.

This game relates to the memory allocation of the computation graph if we see pebbles as memory blocks used to store the output value of a node. We assume that the values of the inputs are known (move 1). We can only compute the value of a vertex if all its predecessors are simultaneously stored in memory (move 2). The *sliding* move means that the memory of a node can be overwritten by its successor during its computation (*inplace reuse*). We can always reuse a memory block from any other node (move 3). Given a graph, the idea is thus to find a strategy to pebble it using the minimum number of pebbles, in other words, using as little memory as possible.

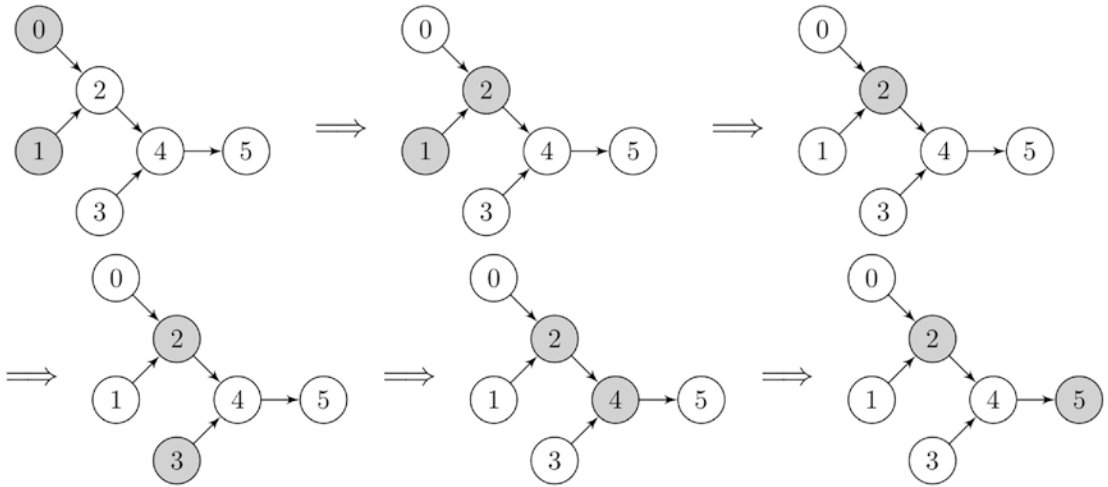


Figure 6-7. Modeling a computation graph memory optimization problem as a pebble game

We also want to avoid pebbling any node twice in order to keep the execution time as low as possible, because that would mean that we compute the same node twice. Given these constraints, finding a strategy using the least amount of pebbles is unfortunately NP complete [45]. Since computation graphs can have a few thousand nodes, we implement a fast heuristic instead of an exact algorithm.

Now we can apply the pebble game process in our memory allocation process. We propose to share memory between nodes that (1) are not necessarily a parent/child pair and (2) that do not have the same output size (by allocating a large block of memory once, without necessarily using all of it all the time). To do this efficiently, we first have to fix an evaluation order (in practice, any topological order). Given this order, we can pinpoint the moment when the memory of a node becomes useless by keeping a counter of how many times it has been used. When it has been used by all its children, we can recycle its memory. Then to allocate memory to a node, we simply check which blocks are available, and we select the one with the closest size (in order not to waste too much memory). If no block is available, we allocate a new one. This can be executed in $O(n \cdot \log(n))$ time, which is negligible compared to the actual cost of evaluating the graph.

Note that some operations cannot overwrite their inputs while they are being computed (the *sliding* move from the pebble game is forbidden) and that some nodes cannot be overwritten for practical purposes, typically constant nodes or neural network weights. When evaluated in the right order, the computation graph needs much smaller blocks of memory than the non-optimized version. As an example, part of an optimized computation graph is shown in Figure 6-8. Each color corresponds to a memory block, and white nodes always need to be kept in memory.

The code in `add_node_to_block` illustrates the steps of introducing a new node. If the memory block of the parent is reusable, the function checks whether the memory is large enough for accommodating the output of the current operator. If so, the node includes the current operator to the list of nodes sharing the same memory block. Moreover, the memory is reshaped according to the shape of the output.

```
let add_node_to_block x block =
  let dst_shp = node_shape x in
  let dst_numel = node_numel x in
  let src_val = value_to_arr (_get_value_block block) in
  let dst_val = arr_to_value
    (A.reshape (A.sub_left src_val 0 dst_numel) dst_shp) in
  block.nodes <- x :: block.nodes;
  _set_block x [| block |];
  (attr x).value <- [| dst_val |]
```

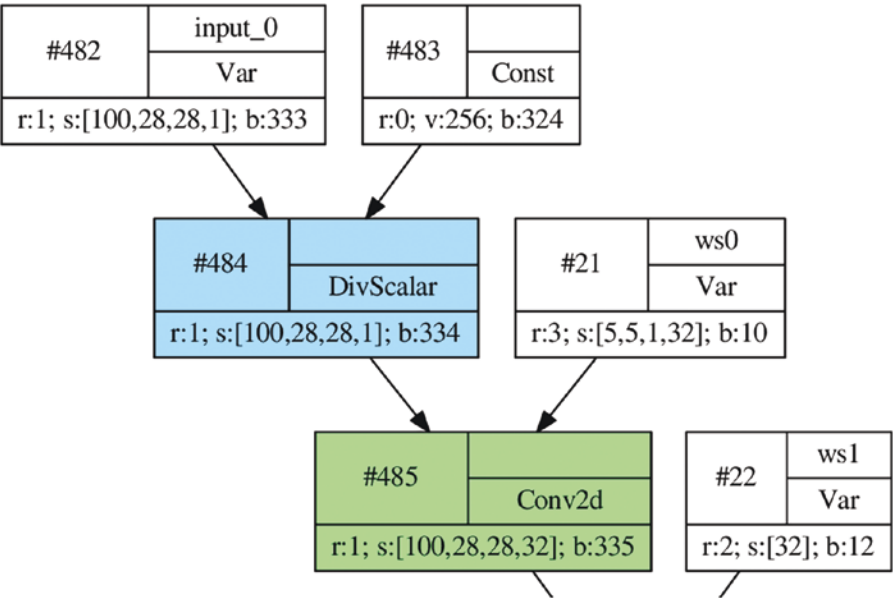


Figure 6-8. Optimized memory allocation

Table 6-1. Evaluation of the Effect of CGraph Memory Optimization Using Different DNN Architectures

Architecture	Time	Time with CG (Building	Memory	Memory with
	Without CG	+ Evaluating) (s)	Without CG	CG (MB)
	(s)		(MB)	
InceptionV3	0.565	0.107 + 0.228 = 0.335	625.76	230.10
ResNet50	0.793	0.140 + 0.609 = 0.749	1309.9	397.07
MNIST (Training)	20.422	0.144 + 10.920 = 11.064	3685.3	895.32
Mask R-CNN	11.538	0.363 + 8.379 = 8.742	6483.4	870.48

Implementing this effectively further reduced the memory consumption of Mask R-CNN from 7GB to 1GB for a 1024x1024 picture. Table 6-1 shows some more statistics illustrating what the computation graph with this new algorithm achieves. These experiments run on a laptop with an Intel i5-6300HQ and 8GB of RAM. In this evaluation, the InceptionV3 and ResNet50 networks are tested with a 299x299 image, and Mask R-CNN is tested with a 768x768 image. The MNIST line refers to the small LeNet-like neural network we used in Section 6.2. The final result is calculated as an

average over 30 evaluations, without reusing precomputed nodes when a computation graph is used. The graph building phase includes graph construction, optimization, and memory initialization.

This evaluation result shows the impact of the CGraph module. On one hand, with graph structure optimization, the execution time of the neural network is significantly reduced. This decrease is especially obvious for inference on large networks such as Mask R-CNN or model training. On the other hand, the memory reduction is also impressive, achieving as large as more than 10x reduction for the inference with the Mask R-CNN network.

6.5 Summary

In this chapter, we introduced the core computation graph module in Owl. We started with a general introduction of the computation graph in numerical computing and why we build that in Owl. Then we used several examples to demonstrate how the computation graph module is used in Owl. This was followed by the internal design of this module, most importantly the CGraph stack and its position in the Owl architecture. The computation graph creates a large optimization space, and in this chapter, we presented two of them in detail. The first is the graph structure optimization, and the second is to optimize the memory allocation in the computation graph. A computation graph is an important research topic, and we believe there is still much potential in this module for performance improvement.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.