CHAPTER 8

Compiler Backends

For a numerical library, it is always beneficial and challenging to extend to multiple execution backends. We have seen how we support accelerators such as the GPU by utilizing a symbolic representation and computation graph standard such as ONNX. In this chapter, we introduce how Owl can be used on more edge-oriented backends, including JavaScript and unikernel.

8.1 Base Library

Before we start, we need to understand how Owl enables compiling to multiple backends by providing different implementations. Owl, as well as many of its external libraries, is actually divided into two parts: a *base* library and a *core* library. The base library is implemented with pure OCaml. For some backends such as JavaScript, we can only use the functions implemented in OCaml.

You may wonder how much we will be limited by the base library. Fortunately, the most advanced modules in Owl are often implemented in pure OCaml, and they live in the base, which includes the modules we have introduced in the previous chapters: algorithmic differentiation, optimization, even neural networks, and many others. Figure 8-1 shows the structure of the core functor stack in Owl.

As we have introduced in Chapter 2, the `Ndarray` module is the core building block in Owl. The base library aims to implement all the necessary functions as the core library `Ndarray` module. The stack is implemented in such a way that the user can switch between these two different implementations without the modules of higher layer. In the Owl functor stack, `Ndarray` is used to support the computation graph module to provide lazy evaluation functionality. Here, we use the `Owl_base_algodiff_primal_ops` module, which is simply a wrapper around the base `Ndarray` module. It also includes a small number of matrix and linear algebra functions. By providing this wrapper instead of

L. Wang and J. Zhao, *Architecture of Advanced Numerical Analysis Systems*,
https://doi.org/10.1007/978-1-4842-8853-5_8

using the Ndarray module directly, we can avoid mixing all the functions in the Ndarray module and make it a large Goliath.

Next, the algorithmic differentiation can build up its computation module based on normal ndarray or its lazy version. For example, you can have an AD that relies on the normal single-precision base ndarray module:

```
module AD = Owl_algodiff_generic.Make
  (Owl_base_algodiff_primal_ops.S)
```

Or it can be built on a double-precision lazy evaluated core Ndarray module:

```
module CPU_Engine = Owl_computation_cpu_engine.Make
  (Owl_algodiff_primal_ops.D)
module AD = Owl_algodiff_generic.Make (CPU_Engine)
```

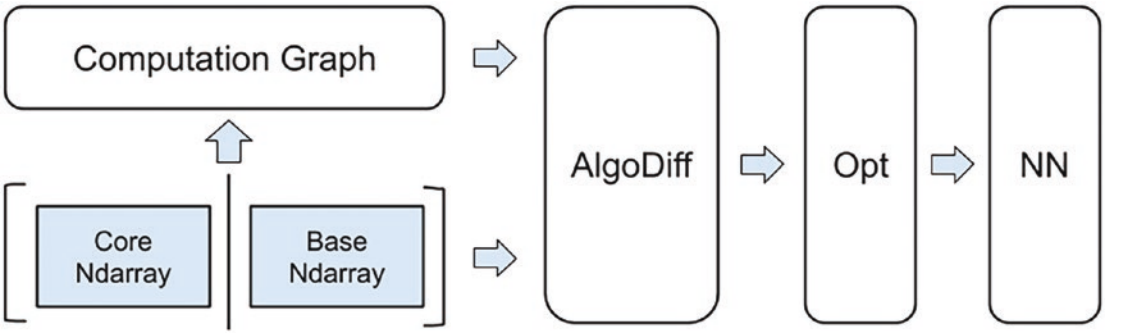

Figure 8-1. *Core functor stack in Owl*

Going up even further on the stack, we have the more advanced optimization and neural network modules. They are both based on the AD module. For example, the following code shows how we can build a neural graph module by layers of functors from the base Ndarray:

```
module G = Owl_neural_graph.Make
            (Owl_neural_neuron.Make
              (Owl_optimise_generic.Make
                (Owl_algodiff_generic.Make
                  (Owl_base_algodiff_primal_ops.S))))
```

Normally, the users do not have to care about how these modules are constructed layer by layer, but understanding the functor stack and typing is nevertheless beneficial, especially when you are creating new module that relies on the base ndarray module.

These examples show that once we have built an application with the core `Ndarray` module, we can then seamlessly switch it to base `Ndarray` without changing anything else. That means that all the code and examples we have seen so far can be used directly on different backends that require pure implementation.

The base library is still an ongoing work, and there is still a lot to do. Though the `Ndarray` module is a large part in the base library, there are other modules that also need to be reimplemented in OCaml, such as the linear algebra module. We need to add more functions such as SVD factorization. Even for the `Ndarray` itself, we still have not totally covered all the functions yet.

Our strategy is to add the base Ndarray functions gradually. We put most of the signature files in the base library, and the core library signature file includes its corresponding signature file from the base library, plus functions that are currently unique to the core library. The target is to total coverage so that the core and base libraries provide exactly the same functions.

As can be expected, the pure OCaml implementation normally performs worse than the C code implemented version. For example, for the convolution operation, without the help of optimized routines from OpenBLAS, etc., we can only provide the naive implementation that includes multiple for-loops. Its performance is orders of magnitude slower than the core library version. Currently, our priority is to implement the functions themselves instead of caring about function optimization, nor do we intend to outperform C code with pure OCaml implementation.

8.2 Backend: JavaScript

At first glance, JavaScript has very little to do with high-performance scientific computing. One important reason we aim to include that in Owl is that the web browser is arguably the most widely deployed technology on various edge devices, for example, mobile phones, tablets, laptops, etc. More and more functionalities are being pushed from data centers to edge for reduced latency, better privacy, and security. And JavaScript applications running in a browser are getting more complicated and powerful. Moreover, JavaScript interpreters are being increasingly optimized, and even relatively complicated computational tasks can run with reasonable performance.

This chapter uses two simple examples to demonstrate how to compile Owl applications into JavaScript code so that you can deploy the analytical code into browsers, using both native OCaml code and Facebook Reason. It additionally requires the use of `dune`, a build system designed for OCaml/Reason projects. As you will see, this will make the compilation to JavaScript effortless.

Native OCaml

We rely on the tool `js_of_ocaml` to convert native OCaml code into JavaScript. Js_of_ocaml is a compiler from OCaml bytecode programs to JavaScript. The process can thus be divided into two phases: first, compile the OCaml source code into bytecode executables, and then apply the `js_of_ocaml` command to it. It supports the core `Bigarray` module among most of the OCaml standard libraries. However, since the `Sys` module is not fully supported, we are careful to not use functions from this module in the base library.

We have described how algorithmic differentiation plays a core role in the ecosystem of Owl, so now we use an example of AD to demonstrate how we convert a numerical program into JavaScript code and then get executed. The example is about optimizing the mathematical function `sin`. The first step is writing down our application in OCaml as follows, then save it into a file `demo.ml`.

```
module AlgodiffD = Owl_algodiff_generic.Make
    (Owl_base_algodiff_primal_ops.D)
open AlgodiffD

let rec desc ?(eta=F 0.01) ?(eps=1e-6) f x =
    let g = (diff f) x in
    if (unpack_flt g) < eps then x
    else desc ~eta ~eps f Maths.(x - eta * g)

let _ =
    let f = Maths.sin in
    let y = desc f (F 0.1) in
    Owl_log.info "argmin f(x) = %g" (unpack_flt y)
```

The code is very simple: the `desc` defines a gradient descent algorithm, and then we use `desc` to calculate the minimum value of the `Maths.sin` function. In the end, we print out the result using the `Owl_log` module's `info` function. Note that we pass in the base `Ndarray` module to the AD functor to create a corresponding AD module.

In the second step, we need to create a `dune` file as follows. This file will instruct how the OCaml code will be first compiled into bytecode and then converted into JavaScript by calling `js_of_ocaml`.

```
(executable
  (name demo)
  (modes byte js)
  (libraries owl-base))
```

With these two files in the same folder, we can then run the following command in the terminal:

```
dune build demo.bc && js_of_ocaml _build/default/demo.bc
```

Or even better, since `js_of_ocaml` is natively supported by `dune`, we can simply execute:

```
dune build
```

The command builds the application and generates a `demo.bc.js` in the `_build/default/` folder. Finally, we can run the JavaScript using `Node.js` (or loading into a browser using an appropriate HTML page).

```
node _build/default/demo.bc.js
```

As a result, we should be able to see the output result showing a value that minimizes the `sin` function and should be similar to

```
argmin f(x) = -1.5708
```

Even though we present a simple example here, the base library can be used to produce more complex and interactive browser applications.

Facebook Reason

Facebook Reason leverages OCaml as a backend to provide type-safe JavaScript. It is gaining its momentum and becoming a popular choice of developing web applications. It actually uses another tool, BuckleScript, to convert the Reason/OCaml code to JavaScript. Since Reason is basically a syntax layer built on top of OCaml, it is very straightforward to use Owl in Reason to develop advanced numerical applications.

In this example, we use reason code to manipulate multidimensional arrays, the core data structure in Owl. First, we save the following code into a reason file called `demo.re`. Note the suffix is *.re* now. It includes several basic math and Ndarray operations in Owl.

```
open! Owl_base;

/* calculate math functions */
let x = Owl_base_maths.sin(5.);
Owl_log.info("Result is %f", x);

/* create random ndarray then print */
let y = Owl_base_dense_ndarray.D.uniform([|3,4,5|]);
Owl_base_dense_ndarray.D.set(y,[|1,1,1|],1.);
Owl_base_dense_ndarray.D.print(y);

/* take a slice */
let z = Owl_base_dense_ndarray.D.get_slice([[],[],[0,3]],y);
Owl_base_dense_ndarray.D.print(z);
```

The preceding code is simple. It creates a random ndarray, takes a slice, and then prints them out. The Owl library can be seamlessly used in Reason. Next, instead of using Reason's own translation of this frontend syntax with `bucklescript`, we still turn to `js_of_ocaml` for help. Let's look at the dune file, which turns out to be the same as that in the previous example:

```
(executable
 (name demo)
 (modes js)
 (libraries owl-base))
```

As in the previous example, you can then compile and run the code with the following commands:

```
dune build
node _build/default/demo.bc.js
```

As you can see, except that the code is written in different languages, the rest of the steps are identical in both example thanks to `js_of_ocaml` and `dune`.

8.3 Backend: MirageOS

Besides JavaScript, another choice of backend we aim to support is the MirageOS. It is an approach to build *unikernels*. A unikernel is a specialized, single address space machine image constructed with library operating systems. Unlike a normal virtual machine, it only contains a minimal set of libraries required for one application. It can run directly on a hypervisor or hardware without relying on operating systems such as Linux and Windows. The unikernel is thus concise and secure, and extremely efficient for distribution and execution on either cloud or edge devices.

MirageOS is one solution to building unikernels. It utilizes the high-level language OCaml and a runtime to provide an API for operating system functionalities. In using MirageOS, the users can think of the Xen hypervisor as a stable hardware platform, without worrying about the hardware details such as devices. Furthermore, since the Xen hypervisor is widely used in platforms such as Amazon EC2 and Rackspace Cloud, MirageOS-built unikernel can be readily deployed on these platforms. Besides, benefiting from its efficiency and security, MirageOS also aims to form a core piece of the Nymote/MISO tool stack to power the Internet of Things.

Example: Gradient Descent

Since MirageOS is based around the OCaml language, we can safely integrate the Owl library with it. To demonstrate how we use MirageOS as a backend, we again use the previous algorithmic differentiation–based optimization example. Before we start, please make sure to follow the installation instruction of the MirageOS. Let's look at the code:

```
module A = Owl_algodiff_generic.Make
  (Owl_algodiff_primal_ops.S)
open A

let rec desc ?(eta=F 0.01) ?(eps=1e-6) f x =
  let g = (diff f) x in
  if (unpack_flt (Maths.abs g)) < eps then x
  else desc ~eta ~eps f Maths.(x - eta * g)

let main () =
  let f x = Maths.(pow x (F 3.) - (F 2.) *
    pow x (F 2.) + (F 2.)) in
  let init = Stats.uniform_rvs ~a:0. ~b:10. in
  let y = desc f (F init) in
  Owl_log.info "argmin f(x) = %g" (unpack_flt y)
```

This part of the code is mostly the same as before. By applying the `diff` function of the algorithmic differentiation module, we use the gradient descent method to find the value that minimizes the function $x^3 - 2x^2 + 2$. Then we need to add something different:

```
module GD = struct
  let start = main (); Lwt.return_unit
end
```

Here, the `start` is an entry point to the unikernel. It performs the normal OCaml function `main` and then returns an `Lwt` thread that will be evaluated to `unit`. Lwt is a concurrent programming library in OCaml. It provides the "promise" data type that can be determined in the future.

All the preceding code is written to a file called `gd_owl.ml`. To build a unikernel, next we need to define its configuration. In the same directory, we create a file called `configure.ml`:

```
open Mirage

let main =
  foreign
    ~packages:[package "owl"]
    "Gd_owl.GD" job
```

```
let () =
  register "gd_owl" [main]
```

It's not complex. First, we need to open the `Mirage` module. Then we declare a value `main` (or you can name it any other name). It calls the `foreign` function to specify the configuration. First, in the `package` parameter, we declare that this unikernel requires the Owl library. The next string parameter "Gd_owl.GD" specifies the name of the implementation file and in that file the module `GD` that contains the `start` entry point. The third parameter `job` declares the type of devices required by a unikernel, such as network interfaces, network stacks, file systems, etc. Since here we only do the calculation, there is no extra device required, so the third parameter is a `job`. Finally, we register the unikernel entry file `gd_owl` with the `main` configuration value.

That's all it takes for coding. Now we can take a look at the compiling part. MirageOS itself supports multiple backends. The crucial choice therefore is to decide which one to use at the beginning by using `mirage configure`. In the directory that holds the previous two files, you run `mirage configure -t unix`, and it configures to build the unikernel into a Unix ELF binary that can be directly executed. Or you can use `mirage configure -t xen`, and then the resulting unikernel will use the hypervisor backend like Xen or KVM. Either way, the unikernel runs as a virtual machine after starting up. In this example, we choose to use Unix as backends. So we run

```
mirage configure -t unix
```

This command generates a `Makefile` based on the configuration information. It includes all the building rules. Next, to make sure all the dependencies are installed, we need to run

```
make depend
```

Finally, we can build the unikernels by simply running

```
make
```

and it calls the `mirage build` command. As a result, now the current directory contains the `_build/gd_owl.native` executable, which is the unikernel we want. Executing it yields a similar result as before:

```
argmin f(x) = 1.33333
```

Example: Neural Network

As a more complex example, we have also built a simple neural network to perform the MNIST handwritten digit recognition task with MirageOS:

```
module N  = Owl_base_algodiff_primal_ops.S
module NN = Owl_neural_generic.Make (N)
open NN
open NN.Graph
open NN.Algodiff

let make_network input_shape =
  input input_shape
  |> lambda (fun x -> Maths.(x / F 256.))
  |> fully_connected 25 ~act_typ:Activation.Relu
  |> linear 10 ~act_typ:Activation.(Softmax 1)
  |> get_network
```

This neural network has two hidden layers, has a small weight size (146KB), and works well in testing (92% accuracy). We can write the weight into a text file. This file is named `simple_mnist.ml`, and similar to the previous example, we can add a unikernel entry point function in the file:

```
module Main = struct
  let start = infer (); Lwt.return_unit
end
```

Here, the `infer` function creates a neural network, loads the weight, and then performs inference on an input image. We also need a configuration file. Again, it's mostly the same:

```
open Mirage

let main =
  foreign
    ~packages:[package "owl-base"]
   "Simple_mnist.Main" job

let () =
  register "Simple_mnist" [main]
```

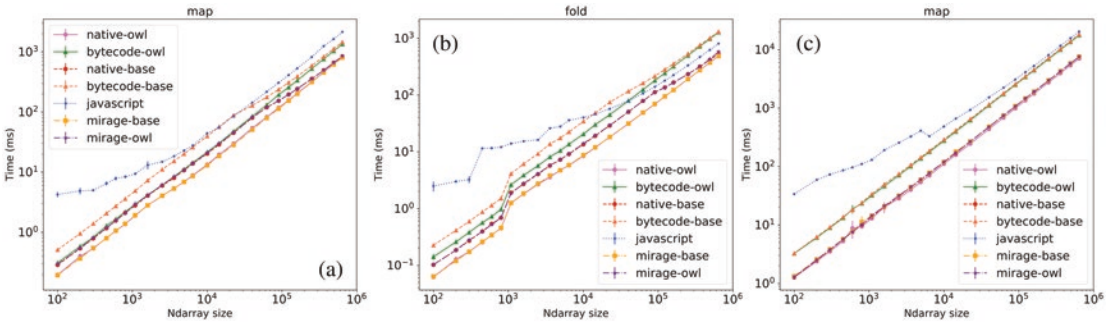

Figure 8-2. *Performance of map and fold operations on ndarray on a laptop and Raspberry Pi*

Once compiled to MirageOS unikernel with Unix backends, the generated binary is 10MB. You can also try compiling this application to JavaScript.

By these examples, we show that the Owl library can be readily deployed into unikernels via MirageOS. The numerical functionality can then greatly enhance the express ability of possible OCaml-MirageOS applications. Of course, here we cannot cover all the important topics about MirageOS; please refer to the documentation of MirageOS and Xen Hypervisor for more information.

8.4 Evaluation

In this section, we mainly compare the performance of different backends. Specifically, we observe three representative groups of operations: (1) `map` and `fold` operations on ndarray; (2) using gradient descent, a common numerical computing subroutine, to get *argmin* of a certain function; (3) conducting inference on complex DNNs, including SqueezeNet and a VGG-like convolution network. The evaluations are conducted on a ThinkPad T460S laptop with an Ubuntu 16.04 operating system. It has an Intel Core i5-6200U CPU and 12GB RAM.

The OCaml compiler can produce two kinds of executables: bytecode and native. Native executables are compiled for specific architectures and are generally faster, while bytecode executables have the advantage of being portable.

For JavaScript, we use the `js_of_ocaml` approach as described in the previous sections. Note that for convenience we refer to the pure implementation of OCaml and the mix implementation of OCaml and C as `base-lib` and `owl-lib` separately, but they are in fact all included in the Owl library. For Mirage compilation, we use both libraries.

Figure 8-2 shows the performance of map and fold operations on ndarray. We use simple functions such as plus and multiplication on 1-d (size <1, 000) and 2-d arrays. The `log-log` relationship between the total size of ndarray and the time each operation takes keeps linear. For both operations, `owl-lib` is faster than `base-lib`, and native executables outperform bytecode ones. The performance of Mirage executives is close to that of native code. Generally, JavaScript runs the slowest, but note how the performance gap between JavaScript and the others converges when the ndarray size grows. For the fold operation, JavaScript even runs faster than bytecode when size is sufficiently large.

Note that for the fold operation, there is an obvious increase in time used at around input size of 10^3 for fold operations, while there is no such change for the map operation. That is because I change the input from one-dimensional ndarray to two-dimensional starting that size. This change does not affect the map operation, since it treats an input of any dimension as a one-dimensional vector. On the other hand, the fold operation considers the factor of dimension, and thus its performance is affected by this change.

In Figure 8-3, we want to investigate if the preceding observations still hold in more complex numerical computation. We choose to use a gradient descent algorithm to find the value that locally minimizes a function. We choose the initial value randomly between $[0, 10]$. For both $sin(x)$ and $x^3 - 2x^2 + 2$, we can see that JavaScript runs the slowest, but this time the `base-lib` slightly outperforms `owl-lib`.

We further compare the performance of DNN, which requires large amount of computation. We compare SqueezeNet and a VGG-like convolution network. They have different sizes of weight and network structure complexities.

Table 8-1 shows that though the performance difference between `owl-lib` and `base-lib` is not obvious, the former is much better. So is the difference between native and bytecode for `base-lib`. JavaScript is still the slowest. The core computation required for DNN inference is the convolution operation. Its implementation efficiency is the key to these differences. Currently, we are working on improving its implementation in `base-lib`.

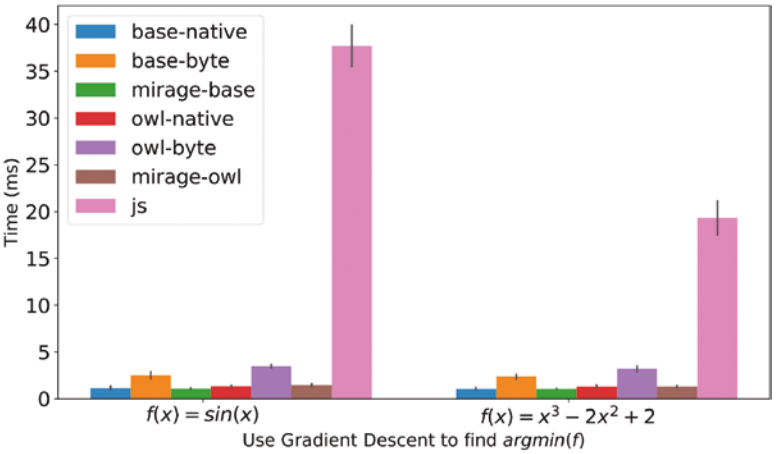

Figure 8-3. *Performance of gradient descent on function f*

Table 8-1. *Inference Speed of Deep Neural Networks*

Time (ms)	VGG	SqueezeNet
owl-native	7.96 (± 0.93)	196.26(± 1.12)
owl-byte	9.87 (± 0.74)	218.99(± 9.05)
base-native	792.56(± 19.95)	14470.97 (± 368.03)
base-byte	2783.33(± 76.08)	50294.93 (± 1315.28)
mirage-owl	8.09(± 0.08)	190.26(± 0.89)
mirage-base	743.18 (± 13.29)	13478.53 (± 13.29)
JavaScript	4325.50(± 447.22)	65545.75 (± 629.10)

We have also conducted the same evaluation experiments on Raspberry Pi 3 Model B. Figure 8-2c shows the performance of the fold operation on ndarray. Besides the fact that all backends run about one order of magnitude slower than that on the laptop, previous observations still hold. This figure also implies that, on resource-limited devices such as Raspberry Pi, the key difference is between native code and bytecode, instead of `owl-lib` and `base-lib` for this operation.

Finally, we also briefly compare the size of executables generated by different backends. We take the SqueezeNet, for example, and the results are shown in Table 8-2. It can be seen that `owl-lib` executives have larger size compared to `base-lib` ones, and JavaScript code has the smallest file size. There does not exist a dominant method of deployment for all these backends. It is thus imperative to choose a suitable backend according to the deployment environment.

Table 8-2. *Size of Executables Generated by Backends*

Size (KB)	Native	Bytecode	Mirage	JavaScript
Base	2437	4298	4602	739
Native	14,875	13,102	16,987	-

8.5 Summary

The base library in Owl was separated from the core module mainly to accommodate multiple possible execution backends. This chapter introduced how the base module works. Then we showed two possible backends: the JavaScript and the unikernel virtual machine. Both backends are helpful to extend the application of Owl to more devices. Finally, we used several examples to demonstrate how these backends are used and their performances.