Interactive Programming for Artificial Intelligence

Deep Learning for Programmers

Dragan Djuric

SAMPLE CHAPTER

DRAFT

An Interactive Tutorial with CUDA, OpenCL, MKL-DNN, Java, and Clojure

https://dragan.rocks aiprobook.com

DRAGAN DJURIC

DEEP LEARNING FOR PROGRAMMERS SAMPLE CHAPTER [DRAFT 0.1.0]

DRAGAN ROCKS

The book is available at
https://aiprobook.com/deep-learning-for-programmers.
Subscribe to the Patreon campaign, at
https://patreon.com/deep_learning,
and get access to all available drafts, and a number of nice perks: from various acknowledgments of your support, to complimentary handcrafted hardcover of the book once it is finished (only selected tiers).

All proceeds go towards funding the author's work on these open-source libraries.

By supporting the book you are also supporting the open-source ecosystem.

PUBLISHED BY DRAGAN ROCKS

HTTP://AIPROBOOK.COM

First printing, 19 May 2019

Sample Chapter

GPU computing with CUDA and OpenCL

This chapter at least demonstrates a network running on the GPU. We generalize the existing code so it can run on any supported GPU device: on an Nvidia GPU with CUDA, and on an AMD GPU with OpenCL. Our code will be so platform-agnostic that we will even be able to mix CUDA and OpenCL[1].

[1] Keep in mind that we do this only for fun, and that it is a bad idea for production code.

The inference layer type

We're starting from the existing layer type, and trimmed down to keep just the batch version. All functions that we have used, `axpy`, `mm`, etc., are polymorphic and general in respect to the device they execute on: CPU, Nvidia GPU, AMD GPU, or Intel GPU. The implementations of activation functions that we have used are general, too.

The dispatch to the right implementation is being done by the type of the vector or matrix structure at hand. The constructor function that we have used is hard-coded for using `double` floating point numbers, to exist in main memory, and use the native CPU backend[2].

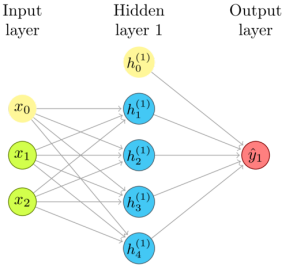

Figure 1: A neural network.

[2] `dge` and `dv` constructors.

```
(defn fully-connected [activ-fn in-dim out-dim]
  (let-release [w (dge out-dim in-dim)
                bias (dv out-dim)]
    (->FullyConnectedInference w bias activ-fn)))
```

Generalize the code

There is only one thing that we have to do to make this code completely general: use general constructors from the `core` namespace, instead of the convenience methods from the `native` namespace. These methods are, in this case, `ge` (general matrix) instead of `dge` (double general native matrix), and `vctr` instead of `dv` (double native vector). The only difference in these methods is that they require an engine-specific factory as their first argument.

We accommodate the `fully-connected` constructor to accept the factory that determines the implementation technology as an input.

```
(defn fully-connected [factory activ-fn in-dim out-dim]
  (let-release [w (ge factory out-dim in-dim)
                bias (vctr factory out-dim)]
    (->FullyConnectedInference w bias activ-fn)))
```

Now, we repeat the example of running the network with `native-double`. That is the same factory that is used by the `dge` and `dv` methods, available in the `native` namespace. We can use `native-float` in its place, to use single-precision floating point computations on the CPU, some of the GPU factories, configure another factory coded by a 3-rd party, or even use the same code provided by Neanderthal, but configured in a different way.

```
(with-release [x (ge native-double 2 2 [0.3 0.9 0.3 0.9])
               ones (vctr native-double 1 1)
               layer-1 (fully-connected native-double tanh! 2 4)
               a-1 (ge native-double 4 2)
               layer-2 (fully-connected native-double sigmoid! 4 1)
               a-2 (ge native-double 1 2)]
  (transfer! [0.3 0.1 0.9 0.0 0.6 2.0 3.7 1.0] (weights layer-1))
  (transfer! [0.7 0.2 1.1 2] (bias layer-1))
  (transfer! [0.75 0.15 0.22 0.33] (weights layer-2))
  (transfer! [0.3] (bias layer-2))
  (transfer (layer-2 (layer-1 x ones a-1) ones a-2)))

=>
#RealGEMatrix[double, mxn:1x2, layout:column, offset:0]
┌       ↓       ↓       ┐
→       0.44    0.44
└                       ┘
```

The display of the result in this example has been modified a little. Instead of doing a `println` as in the previous chapters, we transfer the resulting matrix to main memory. As `println` is typically not used in production code, `transfer` is more general and platform-agnostic way to ensure that the output values survive the `with-release` scope.

Don't forget that, in this example, we have used `with-release` for *all* bindings, even the output `a-2`. I do this because the code should support CPU *and GPU*. In the main memory releasing the data is of great help, but it is optional in a REPL session, since the memory eventually gets released by the JVM[3]. On the GPU, however, JVM can not do anything. The underlying GPU buffer that is not released explicitly, is not released at all until we release the whole context. Therefore, get in the habit of always taking care of the lifecycle and release all vectors, matrices and other structures as soon as possible.[4]

[3] With a few caveats since JVM might not do it as soon as we may hope.

[4] In most cases it is best to do this automatically by `with-release` or `let-release`. In more advanced cases, we can directly call `release`.

During interactive sessions, we would like to see the result in the REPL. But, how, if the data stored in the result that is being returned (`a-2`) is released just the moment before it is about to be printed. Here, the `transfer` method transfers the data from wherever it is (main memory or GPU memory) to the equivalent object in the main memory.

This particular network

We are going to run this code on different devices, so wrapping it into a function might be a good idea. Note that we provide `factory` as the argument, and everything else is general and the same for all platforms!

```
(defn this-particular-network [factory]
  (with-release [x (ge factory 2 2 [0.3 0.9 0.3 0.9])
                 ones (vctr factory 1 1)
                 layer-1 (fully-connected factory tanh! 2 4)
                 a-1 (ge factory 4 2)
                 layer-2 (fully-connected factory sigmoid! 4 1)
                 a-2 (ge factory 1 2)]
    (transfer! [0.3 0.1 0.9 0.0 0.6 2.0 3.7 1.0] (weights layer-1))
    (transfer! [0.7 0.2 1.1 2] (bias layer-1))
    (transfer! [0.75 0.15 0.22 0.33] (weights layer-2))
    (transfer! [0.3] (bias layer-2))
    (transfer (layer-2 (layer-1 x ones a-1) ones a-2))))
```

We can call this function and instruct it to use double-precision floating point computation on the CPU.

```
(this-particular-network native-double)

=>
#RealGEMatrix[double, mxn:1x2, layout:column, offset:0]
┌       ↓       ↓       ┐
→       0.44    0.44
└                       ┘
```

It can use single-precision floating point computation, still on the CPU.[5]

[5] If you are struggling to find the difference in these two printouts, note that the first prints "double", while the second prints "float".

```
(this-particular-network native-float)

=>
#RealGEMatrix[float, mxn:1x2, layout:column, offset:0]
┌       ↓       ↓       ┐
→       0.44    0.44
└                       ┘
```

CUDA on an Nvidia GPU

The same code, without changes, runs on the GPU! The only thing that it needs, is the factory that sets it up with appropriate engines.

For engines based on Nvidia's CUDA[6] platform, we use functions from `uncomplicate.clojurecuda.core`[7] namespace to choose and set up the GPU itself. We may have more than one graphics accelerator in our system, and Neanderthal has to know which one to use. `with-default` is a method that will choose the best device that you have, and set it up automatically. There are more fine grained methods in the ClojureCUDA[8] library if you need more control.

[6] `https://developer.nvidia.com/cuda-toolkit`

[7] We are showing the namespace requires in the code below, but will skip that in the following chapters.

[8] `https://clojurecuda.uncomplicate.org`

```
(require
 '[uncomplicate.clojurecuda.core :as cuda
   :refer [current-context default-stream synchronize!]])
```

Next, we use the `cuda-float` constructor to create a factory whose engines will use single-precision floating point computations in the default context and stream provided by ClojureCUDA. We may need more than one factory for advanced computations.

```
(require '[uncomplicate.neanderthal.cuda :refer [cuda-float]])

(cuda/with-default
  (with-release [cuda-factory (cuda-float (current-context) default-stream)]
    (this-particular-network cuda-factory)))

=>
#RealGEMatrix[float, mxn:1x2, layout:column, offset:0]
┌       ↓       ↓       ┐
→       0.44    0.44
└                       ┘
```

OpenCL on an AMD GPU

In case you have an AMD or Intel GPU, you will not be able to work with CUDA platform. For more universal coverage Neanderthal supports OpenCL[9] which is an open platform equivalent to CUDA, that supports all major hardware vendors: AMD, Intel, and even Nvidia.

Instead of ClojureCUDA, you'll use ClojureCL[10] to set up the execution environment. Other than a few differences in terminology, most of the knowledge of parallel computing on the GPU is transferable between CUDA and OpenCL.

[9] `https://www.khronos.org/opencl/`

[10] `https://clojurecl.uncomplicate.org`

```
(require
 '[uncomplicate.neanderthal.opencl :refer [opencl-float]]
```

```
 '[uncomplicate.clojurecl.core :as opencl
   :refer [*context* *command-queue* finish!]])

(opencl/with-default
  (with-release [opencl-factory (opencl-float *context* *command-queue*)]
    (this-particular-network opencl-factory)))

=>
#RealGEMatrix[float, mxn:1x2, layout:column, offset:0]
┌       ↓       ↓       ┐
→       0.44    0.44
└                       ┘
```

We can even mix CUDA and OpenCL

With Neanderthal, we can even combine code that partly runs on an Nvidia GPU and partly on an AMD GPU. For performance reasons, it is unlikely that we would do this often in "real" code. However, we can do it for fun and learning. This example has been included mainly to demonstrate how flexible Neanderthal and Clojure are. This is something that we would struggle to do in competing platforms!

```
(opencl/with-default
  (cuda/with-default
      (with-release [opencl-factory (opencl-float *context* *command-queue*)
                     cuda-factory (cuda-float (current-context) default-stream)
                     x (ge opencl-factory 2 2 [0.3 0.9 0.3 0.9])
                     ones-opencl (vctr opencl-factory 1 1)
                     layer-1 (fully-connected opencl-factory tanh! 2 4)
                     a-1 (ge opencl-factory 4 2)
                     a-1-cuda (ge cuda-factory 4 2)
                     ones-cuda (vctr cuda-factory 1 1)
                     layer-2 (fully-connected cuda-factory sigmoid! 4 1)
                     a-2 (ge cuda-factory 1 2)]
        (transfer! [0.3 0.1 0.9 0.0 0.6 2.0 3.7 1.0] (weights layer-1))
        (transfer! [0.7 0.2 1.1 2] (bias layer-1))
        (transfer! [0.75 0.15 0.22 0.33] (weights layer-2))
        (transfer! [0.3] (bias layer-2))
        (layer-1 x ones-opencl a-1)
        (transfer! a-1 a-1-cuda)
        (transfer (layer-2 a-1-cuda ones-cuda a-2)))))

=>
#RealGEMatrix[float, mxn:1x2, layout:column, offset:0]
```

```
┌       ↓       ↓       ┐
→       0.44    0.44
└                       ┘
```

Micro benchmark

One aspect of GPU computing is *how to do it at all*. With Neanderthal, ClojureCL and ClojureCUDA it is not that hard. Another question is: *is it worth the trouble*?

Nvidia GTX 1080Ti

We are going to measure the same superficial example that we used in previous chapters. The heavily-optimized native CPU engine backed by Intel's MKL[11] computed one pass in 6 seconds. We hope that Nvidia's GeForce GTX 1080Ti[12], at 11 TFLOPS[13], will be able to do it much faster.

Please note the `(synchronize!)`[14] call. GPU calls are asynchronous[15], and here we are making sure that we block the main thread and wait for the computation to complete before we declare victory.

[11] The fastest CPU thing around. See more at `https://software.intel.com/en-us/mkl`

[12] A Nvidia's flagship gaming GPU from 2017.

[13] Tera (10^12) Floating Point Operations (per second).

[14] This function is part of ClojureCUDA.

[15] Read more about asynchronous nature of GPU programming in another book in this series, Interactive GPU Programming with CUDA.

```
(cuda/with-default
  (with-release [factory (cuda-float (current-context)
                                     default-stream)]
    (with-release [x (ge factory 10000 10000)
                   ones (entry! (vctr factory 10000) 1)
                   layer-1 (fully-connected factory tanh! 10000 5000)
                   a1 (ge factory 5000 10000)
                   layer-2 (fully-connected factory sigmoid! 5000 1000)
                   a2 (ge factory 1000 10000)
                   layer-3 (fully-connected factory sigmoid! 1000 10)
                   a3 (ge factory 10 10000)]
      (time
       (do
         (layer-3 (layer-2 (layer-1 x ones a1) ones a2) ones a3)
         (synchronize!))))))

"Elapsed time: 122.529925 msecs"
```

122 milliseconds. It is well worth the trouble! This is roughly *50 times faster* than the optimized engine on my CPU![16]

[16] Note that this depends on both the example and the hardware.

AMD R9 290X

The computer that this book was written on also hosts an AMD GPU, R9 290X,[17] which has the theoretical peak performance of 5 TFLOPS.

[17] An older AMD's flagship gaming GPU from 2013.

```
(opencl/with-default
  (with-release [factory (opencl-float *context* *command-queue*)]
    (with-release [x (ge factory 10000 10000)
                   ones (entry! (vctr factory 10000) 1)
                   layer-1 (fully-connected factory tanh! 10000 5000)
                   a1 (ge factory 5000 10000)
                   layer-2 (fully-connected factory sigmoid! 5000 1000)
                   a2 (ge factory 1000 10000)
                   layer-3 (fully-connected factory sigmoid! 1000 10)
                   a3 (ge factory 10 10000)]
      ;; The first time a BLAS operation is used in OpenCL
      ;; might incur initialization cost. Warm up the engine.
      (layer-1 x ones a1)
      (finish!)
      (time
       (do
         (layer-3 (layer-2 (layer-1 x ones a1) ones a2) ones a3)
         (finish!))))))

"Elapsed time: 330.683851 msecs"
```

This is roughly 3 times slower than Nvidia. It is still worth the effort, since it is *almost 20 times faster* than the CPU.

Someone may be disappointed by this not being close enough to GTX 1080Ti's speed, since R9 290X should be twice as slow by the specifications (5 TFOPS vs 11 TFLOPS). Instead of Nvidia's proprietary BLAS matrix routines, Neanderthal uses an open-source engine in its OpenCL backend. Although it's not *that* much behind, it can not match Nvidia's hardware optimizations at the same level. When we have an Nvidia's GPU, we can still use ClojureCL but if we need maximum performance, we should use ClojureCUDA.

A laconic performance comparison

Let us sum this up. In this example, we managed to accelerate a top-performance CPU code by *20 times* with an old AMD GPU, and *50 times* with a fairly recent, but not the best, Nvidia GPU, keeping the same code!