AMMI – Introduction to Deep Learning

6.6. Using GPUs

François Fleuret
https://fleuret.org/ammi-2018/
Fri Nov 9 22:38:37 UTC 2018
The size of current state-of-the-art networks makes computation a critical issue, in particular for training and optimizing meta-parameters.
The size of current state-of-the-art networks makes computation a critical issue, in particular for training and optimizing meta-parameters.

Although they were historically developed for mass-market real-time CGI, the highly parallel architecture of GPUs is extremely fitting to signal processing and high dimension linear algebra.

Their use is instrumental in the success of deep-learning.
A standard NVIDIA GTX 1080 has 2,560 single-precision computing cores clocked at 1.6GHz, and delivers a peak performance of $\sim 9$ TFlops.

The precise structure of a GPU memory and how its cores communicate with it is a complicated topic that we will not cover here.
A standard NVIDIA GTX 1080 has 2,560 single-precision computing cores clocked at 1.6GHz, and delivers a peak performance of $\approx 9$ TFlops.

The precise structure of a GPU memory and how its cores communicate with it is a complicated topic that we will not cover here.
A standard NVIDIA GTX 1080 has 2,560 single-precision computing cores clocked at 1.6GHz, and delivers a peak performance of $\approx 9$ TFlops.

The precise structure of a GPU memory and how its cores communicate with it is a complicated topic that we will not cover here.
A standard NVIDIA GTX 1080 has 2,560 single-precision computing cores clocked at 1.6GHz, and delivers a peak performance of $\approx 9$ TFlops.
A standard NVIDIA GTX 1080 has 2,560 single-precision computing cores clocked at 1.6GHz, and delivers a peak performance of $\sim 9$ TFlops.
A standard NVIDIA GTX 1080 has 2,560 single-precision computing cores clocked at 1.6GHz, and delivers a peak performance of $\sim 9$ TFlops.
A standard NVIDIA GTX 1080 has 2,560 single-precision computing cores clocked at 1.6GHz, and delivers a peak performance of $\sim 9$ TFlops.
A standard NVIDIA GTX 1080 has 2,560 single-precision computing cores clocked at 1.6GHz, and delivers a peak performance of \(\sim 9\) TFlops.

The precise structure of a GPU memory and how its cores communicate with it is a complicated topic that we will not cover here.
TABLE 7. COMPARATIVE EXPERIMENT RESULTS (TIME PER MINI-BATCH IN SECOND)

<table>
<thead>
<tr>
<th></th>
<th>Desktop CPU (Threads used)</th>
<th>Server CPU (Threads used)</th>
<th>Single GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCN-S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Caffe</td>
<td>1.329 0.710 0.423</td>
<td>1.279 1.131 0.595</td>
<td>0.382</td>
</tr>
<tr>
<td>CNTK</td>
<td>1.227 0.660 0.435</td>
<td>1.340 0.909 0.654</td>
<td>0.441</td>
</tr>
<tr>
<td>TF</td>
<td>7.062 4.789 2.648</td>
<td>9.571 6.569 3.399</td>
<td>0.630</td>
</tr>
<tr>
<td>MXNet</td>
<td>4.621 2.607 2.162</td>
<td>5.824 3.356 2.395</td>
<td>0.945</td>
</tr>
<tr>
<td>Torch</td>
<td>3.645 2.429 1.424</td>
<td>4.336 2.468 1.543</td>
<td>1.090</td>
</tr>
</tbody>
</table>

| AlexNet-S           |                           |                           |            |
| Caffe               | 1.606 0.999 0.719         | 1.533 1.045 0.797         | 0.850      |
| CNTK                | 3.761 1.974 1.276         | 3.852 2.600 1.567         | 1.168      |
| TF                  | 6.525 2.936 1.749         | 5.741 4.216 2.202         | 0.701      |
| MXNet               | 2.977 2.340 2.250         | 3.518 3.203 2.926         | 2.827      |
| Torch               | 4.645 2.429 1.424         | 4.336 2.468 1.543         | 1.090      |

| RenNet-50           |                           |                           |            |
| MXNet               | 48.000 46.154 44.444      | 57.831 57.143 54.545     | 53.333     |

| FCN-R               |                           |                           |            |
| Caffe               | 2.476 1.499 1.149         | 2.282 1.748 1.403         | 1.211      |
| CNTK                | 1.845 0.970 0.571         | 1.592 0.857 0.501         | 0.325      |
| TF                  | 2.647 1.913 1.157         | 3.410 2.541 1.297         | 0.661      |
| MXNet               | 1.914 1.072 0.719         | 1.609 1.065 0.731         | 0.534      |
| Torch               | 1.670 0.926 0.565         | 1.379 0.915 0.662         | 0.440      |

| AlexNet-R           |                           |                           |            |

| RenNet-56           |                           |                           |            |
| CNTK                | - - -                    | - - -                     | - 0.116    |
| TF                  | - - -                    | - - -                     | - 0.138    |
| MXNet               | 34.409 31.255 30.069      | 44.878 43.775 42.299     | 42.965     |
| Torch               | 5.758 3.222 2.368         | 8.691 4.965 3.040         | 2.560      |

| LSTM                |                           |                           |            |
| Caffe               | - - -                    | - - -                     | - 0.116    |
| CNTK                | 0.186 0.120 0.090         | 0.211 0.139 0.117         | 0.114      |
| TF                  | 4.662 3.385 1.935         | 6.449 4.351 2.238         | 1.183      |
| MXNet               | - - -                    | - - -                     | - 0.133    |

Note: The mini-batch sizes for FCN-S, AlexNet-S, ResNet-50, FCN-R, AlexNet-R, ResNet-56 and LSTM are 64, 16, 16, 1024, 1024, 128 and 128 respectively.

(Shi et al., 2016)
The current standard to program a GPU is through the CUDA ("Compute Unified Device Architecture") model, defined by NVIDIA.
The current standard to program a GPU is through the CUDA ("Compute Unified Device Architecture") model, defined by NVIDIA.

Alternatives are OpenCL, backed by many CPU/GPU manufacturers, and more recently AMD’s HIP ("Heterogeneous-compute Interface for Portability").
The current standard to program a GPU is through the CUDA ("Compute Unified Device Architecture") model, defined by NVIDIA.

Alternatives are OpenCL, backed by many CPU/GPU manufacturers, and more recently AMD’s HIP ("Heterogeneous-compute Interface for Portability").

Google developed its own processor for deep learning dubbed TPU ("Tensor Processing Unit") for in-house use. It is targeted at TensorFlow and offers excellent flops/watt performance.
The current standard to program a GPU is through the CUDA (“Compute Unified Device Architecture”) model, defined by NVIDIA.

Alternatives are OpenCL, backed by many CPU/GPU manufacturers, and more recently AMD’s HIP (“Heterogeneous-compute Interface for Portability”).

Google developed its own processor for deep learning dubbed TPU (“Tensor Processing Unit”) for in-house use. It is targeted at TensorFlow and offers excellent flops/watt performance.

In practice, as of today (27.01.2018), NVIDIA hardware remains the default choice for deep learning, and CUDA is the reference framework in use.
From a practical perspective, libraries interface the framework (e.g. PyTorch) with the “computational backend” (e.g. CPU or GPU)
From a practical perspective, libraries interface the framework (e.g. PyTorch) with the “computational backend” (e.g. CPU or GPU)

- **BLAS** (“Basic Linear Algebra Subprograms”): vector/matrix products, and the cuBLAS implementation for NVIDIA GPUs,
- **LAPACK** (“Linear Algebra Package”): linear system solving, Eigen-decomposition, etc.
- **cuDNN** (“NVIDIA CUDA Deep Neural Network library”) computations specific to deep-learning on NVIDIA GPUs.
Using GPUs in PyTorch
The use of the GPUs in PyTorch is done by creating or copying tensors into their memory.
The use of the GPUs in PyTorch is done by creating or copying tensors into their memory.

As for the type, the device can be specified to the creation operations.

```python
>>> x = torch.zeros(10, 10)
>>> x.device
device(type='cpu')
>>> x = torch.zeros(10, 10, device = torch.device('cuda'))
>>> x.device
device(type='cuda', index=0)
>>> x = torch.zeros(10, 10, device = torch.device('cuda:1'))
>>> x.device
device(type='cuda', index=1)
```
The `torch.Tensor.cuda([id])` method returns a clone on the GPU if the tensor is not already there or returns the tensor itself if it was already there.

The id argument is optional, and a default GPU can be set with the `torch.cuda.device(id)` context manager.

The method `torch.Tensor.cpu()` makes a clone on the CPU if needed.
The `torch.Tensor.cuda([id])` method returns a clone on the GPU if the tensor is not already there or returns the tensor itself if it was already there.

The id argument is optional, and a default GPU can be set with the `torch.cuda.device(id)` context manager.

The method `torch.Tensor.cpu()` makes a clone on the CPU if needed.

⚠️ Moving data between the CPU and the GPU memories is far slower than moving it inside the GPU memory.
```python
>>> m = torch.empty(10, 10).normal_()
>>> m.device
device(type='cpu')
```
```python
>>> m = torch.empty(10, 10).normal_()
>>> m.device
device(type='cpu')
>>> x = torch.empty(10, 100).normal_()
>>> q = m@x
>>> q.device
device(type='cpu')
>>> m = m.cuda()
>>> m.device
device(type='cuda', index=0)
>>> x = x.cuda()
>>> q = m@x  # This is done on GPU (#0)
>>> q.device
device(type='cuda', index=0)
```
Operations maintain the types and devices of the tensors, so you generally do not need to worry about making your code generic regarding these aspects.

To explicitly create new tensors, the best is to use `new_*()` methods.

```python
>>> u = torch.empty(3, 5, dtype = torch.float64).normal_()
>>> v = u.new_zeros(1, 2)
>>> v
tensor([[ 0.,  0.]], dtype=torch.float64)
>>> w = torch.empty(3, 5, dtype = torch.float16,
...    device = torch.device('cuda:1')).fill_(1.0)
>>> w.new_full((2, 3), 1.4)
tensor([[ 1.4004,  1.4004,  1.4004],
        [ 1.4004,  1.4004,  1.4004]], dtype=torch.float16, device='cuda:1')
```
Apart from `copy_()`, operations cannot mix different tensor types or devices:

```python
>>> import torch
>>> x = torch.empty(3, 5).normal_()
>>> y = torch.empty(3, 5).normal_().cuda()
>>> x.copy_(y)
tensor([[ 0.4071,  0.7589, -0.5321,  0.9103, -1.4985],
        [-0.1059,  2.1554, -0.0774, -0.4520,  1.5123],
        [ 0.1322,  0.1002, -0.4071,  1.8927, -0.5800]])
```

```python
>>> x + y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: Expected object of type torch.FloatTensor but found type torch.cuda.FloatTensor for argument #3 'other'
```
Apart from `copy_()`, operations cannot mix different tensor types or devices:

```python
>>> import torch
>>> x = torch.empty(3, 5).normal_
>>> y = torch.empty(3, 5).normal_.cuda()
>>> x.copy_(y)
tensor([[ 0.4071,  0.7589, -0.5321,  0.9103, -1.4985],
        [-0.1059,  2.1554, -0.0774, -0.4520,  1.5123],
        [ 0.1322,  0.1002, -0.4071,  1.8927, -0.5800]])

>>> x + y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: Expected object of type torch.FloatTensor but found type torch.cuda.FloatTensor for argument #3 'other'
```

Similarly if multiple GPUs are available, cross-GPUs operations are not allowed by default, with the exception of `copy_()`.

⚠️
The method `torch.Module.cuda()` moves all the parameters and buffers of the module (and registered sub-modules recursively) to the GPU, and conversely, `torch.Module.cpu()` moves them to the CPU.

⚠️ Although they do not have a “_” in their names, these `Module` operations make changes in-place.
The method `torch.Module.cuda()` moves all the parameters and buffers of the module (and registered sub-modules recursively) to the GPU, and conversely, `torch.Module.cpu()` moves them to the CPU.

⚠️ Although they do not have a “_” in their names, these `Module` operations make changes in-place.

The method `torch.cuda.is_available()` returns a Boolean value indicating if a GPU is available, so a typical snippet of code to use the GPU would be

```python
if torch.cuda.is_available():
    model.cuda()
    criterion.cuda()
    train_input, train_target = train_input.cuda(), train_target.cuda()
    test_input, test_target = test_input.cuda(), test_target.cuda()
```
A very simple way to leverage multiple GPUs is to wrap the model in a `nn.DataParallel`.
A very simple way to leverage multiple GPUs is to wrap the model in a `nn.DataParallel`.

The `forward` of `nn.DataParallel(my_module)` will

1. split the input mini-batch along the first dimension in as many mini-batches as there are GPUs,
2. send them to the forwards of clones of `my_module` located on each GPU,
3. concatenate the results.

And it is (of course!) autograd-compliant.
If we define a simple module to printout the calls to `forward`.

class Dummy(nn.Module):
    def __init__(self, m):
        super(Dummy, self).__init__()
        self.m = m

    def forward(self, x):
        print('Dummy.forward', x.size(), x.device)
        return self.m(x)
x = torch.empty(50, 10).normal_()
model = Dummy(nn.Linear(10, 5))

print('On CPU')
y = model(x)

x = x.cuda()
model.cuda()

print('On GPU w/o nn.DataParallel')
y = model(x)

print('On GPU w/ nn.DataParallel')
parallel_model = nn.DataParallel(model)
y = parallel_model(x)

will print, on a machine with two GPUs:

On CPU
Dummy.forward torch.Size([50, 10]) cpu
On GPU w/o nn.DataParallel
Dummy.forward torch.Size([50, 10]) cuda:0
On GPU w/ nn.DataParallel
Dummy.forward torch.Size([25, 10]) cuda:0
Dummy.forward torch.Size([25, 10]) cuda:1
The end
References