Deep learning

6.6. Using GPUs

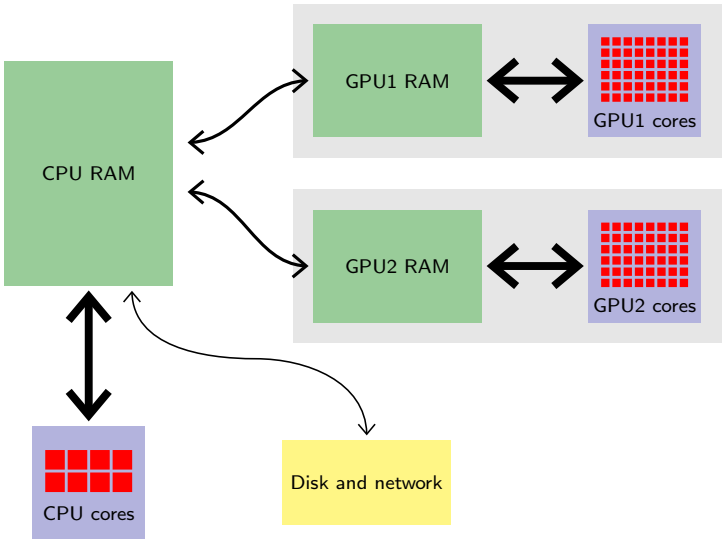
François Fleuret

https://fleuret.org/dlc/

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 (Raina et al., 2009; Ciresan et al., 2010; Krizhevsky et al., 2012; Shi et al., 2016).



A standard NVIDIA GTX 3090 has $10,500$ computing cores clocked at 1.5GHz, and delivers a peak performance of $\simeq 35$ 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.

Notes

The thickness of the arrows illustrate the speed of the communication between components. Thin arrows mean slow access wile thick arrows means fast.
Each core of the CPU is able to run its own program and has access to the CPU memory (CPU RAM) at high speed, and to the disk and network through a slower bus. A CPU may have up to tens of cores (16, 32, sometimes 64).
A GPU comes with its own memory (GPU RAM) and thousands of cores. The GPU cores have a very fast access to the GPU memory, while the access between the CPU and GPU memories is quite slow: moving data from the CPU memory to the GPU memory is inefficient.
We will not cover the organization of the GPU memory (groups of cores, groups of groups, cache, etc.) but it is worth mentioning that properly programming a GPU at a low level is an extremely difficult task that requires strong expertise.

TABLE 7. COMPARATIVE EXPERIMENT RESULTS (TIME PER MINI-BATCH IN SECOND)

		Desktop CPU (Threads used)				Server CPU (Threads used)						Single GPU		
		1	2	4	8	1	2	4	8	16	32	G980	G1080	K80
FCN-S	Caffe	1.324	0.790	**0.578**	15.444	1.355	0.997	0.745	**0.573**	0.608	1.130	0.041	**0.030**	0.071
	CNTK	1.227	0.660	**0.435**	-	1.340	0.909	0.634	0.488	**0.441**	1.000	0.045	**0.033**	0.074
	TF	7.062	4.789	2.648	**1.938**	9.571	6.569	3.399	1.710	0.946	**0.630**	0.060	**0.048**	0.109
	MXNet	4.621	2.607	2.162	**1.831**	5.824	3.356	2.395	2.040	**1.945**	2.670	-	**0.106**	0.216
	Torch	1.329	0.710	**0.423**	-	1.279	1.131	0.595	0.433	**0.382**	1.034	0.040	**0.031**	0.070
AlexNet-S	Caffe	1.606	0.999	**0.719**	-	1.533	1.045	**0.797**	0.850	0.903	1.124	0.034	**0.021**	0.073
	CNTK	3.761	1.974	**1.276**	-	3.852	2.600	1.567	1.347	**1.168**	1.579	0.045	**0.032**	0.091
	TF	6.525	2.936	1.749	**1.535**	5.741	4.216	2.202	1.160	**0.701**	0.962	0.059	**0.042**	0.130
	MXNet	2.977	2.340	2.250	**2.163**	3.518	3.203	2.926	2.828	**2.827**	2.887	0.020	**0.014**	0.042
	Torch	4.645	2.429	**1.424**	-	4.336	2.468	1.543	1.248	**1.090**	1.214	0.033	**0.023**	0.070
RenNet-50	Caffe	11.554	7.671	**5.652**	-	10.643	8.600	6.723	**6.019**	6.654	8.220	-	**0.254**	0.766
	CNTK	-	-	-	-	-	-	-	-	-	-	0.240	**0.168**	0.638
	TF	23.905	16.435	10.206	**7.816**	29.960	21.846	11.512	6.294	**4.130**	4.351	0.327	**0.227**	0.702
	MXNet	48.000	46.154	44.444	**43.243**	57.831	57.143	54.545	54.545	**53.333**	55.172	0.207	**0.136**	0.449
	Torch	13.178	7.500	**4.736**	4.948	12.807	8.391	5.471	4.164	**3.683**	4.422	0.208	**0.144**	0.523
FCN-R	Caffe	2.476	1.499	**1.149**	-	2.282	1.748	1.403	1.211	1.127	**1.127**	0.025	**0.017**	0.055
	CNTK	1.845	0.970	0.661	**0.571**	1.592	0.857	0.501	0.323	**0.252**	0.280	0.025	**0.017**	0.053
	TF	2.647	1.913	1.157	**0.919**	3.410	2.541	1.297	0.661	0.361	**0.325**	0.033	**0.020**	0.063
	MXNet	1.914	1.072	0.719	**0.702**	1.609	1.065	0.731	0.534	0.451	**0.447**	0.029	**0.019**	0.060
	Torch	1.670	0.926	**0.565**	0.611	1.379	0.915	0.662	0.440	0.402	**0.366**	0.025	**0.016**	0.051
AlexNet-R	Caffe	3.558	2.587	**2.157**	2.963	4.270	3.514	3.381	**3.364**	4.139	4.930	0.041	**0.027**	0.137
	CNTK	9.956	7.263	**5.519**	6.015	9.381	6.078	4.984	**4.765**	6.256	6.199	0.045	**0.031**	0.108
	TF	4.535	3.225	1.911	**1.565**	6.124	4.229	2.200	1.396	1.036	**0.971**	**0.227**	0.317	0.385
	MXNet	13.401	12.305	12.278	**11.950**	17.994	17.128	16.764	**16.471**	17.471	17.770	0.060	**0.032**	0.122
	Torch	5.352	3.866	**3.162**	3.259	6.554	5.288	4.365	**3.940**	4.157	4.165	0.069	**0.043**	0.141
RenNet-56	Caffe	6.741	5.451	**4.989**	6.691	7.513	**6.119**	6.232	6.689	7.313	9.302	-	**0.116**	0.378
	CNTK	-	-	-	-	-	-	-	-	-	-	0.206	**0.138**	0.562
	TF	-	-	-	-	-	-	-	-	-	-	0.225	**0.152**	0.523
	MXNet	34.409	31.255	**30.069**	31.388	44.878	43.775	**42.299**	42.965	43.854	44.367	0.105	**0.074**	0.270
	Torch	5.758	3.222	**2.368**	2.475	8.691	4.965	3.040	**2.560**	2.575	2.811	0.150	**0.101**	0.301
LSTM	Caffe	-	-	-	-	-	-	-	-	-	-	-	-	-
	CNTK	0.186	0.120	**0.090**	0.118	0.211	0.139	0.117	0.114	**0.114**	0.198	0.018	**0.017**	0.043
	TF	4.662	3.385	1.935	**1.532**	6.449	4.351	2.238	1.183	0.702	**0.598**	0.133	**0.065**	0.140
	MXNet	-	-	-	-	-	-	-	-	-	-	0.089	**0.079**	0.149
	Torch	6.921	3.831	**2.682**	3.127	7.471	4.641	3.580	**3.260**	5.148	5.851	0.399	**0.324**	0.560

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)

Notes

The table shows the results of a detailed benchmark between desktop CPU, server CPU, and single GPU for several deep learning libraries. The reported values are the time in seconds is takes to process one mini-batch. The take-home message is that GPUs provide a speed up of an order of magnitude or more.

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

Alternatives are OpenCL, backed by several CPU/DSP manufacturers, and more recently AMD's HIP/ROCm.

Google developed its own line of processors for deep learning dubbed TPU ("Tensor Processing Unit") which offer excellent flops/watt performance.

In practice, as of today (29.03.2022), 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)

- 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.

Operations on tensors in a device's memory are done by the said device.

Notes

Multiplying two tensors which are on the CPU (resp. GPU) memory is done by the CPU (resp. GPU).
Basic tensor operators, and virtually all functions require operand to be on the same device. E.g. when a forward pass has been done on the GPU, the output is on the GPU. Computing a loss with the target values would require the targets to also be on the GPU.

As for the type, the device can be specified to the creation operations as a device, or as a string that will implicitly be converted to a device.

```
>>> 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)
>>> x = torch.zeros(10, 10, device = 'cuda:0')
>>> x.device
device(type='cuda', index=0)
```

The `torch.Tensor.to(device)` returns a clone on the specified device **if the tensor is not already there** or returns the tensor itself if it was already there.

The argument `device` can be either a string, or a device.

Alternatives are `torch.Tensor.cuda([gpu_id])` and `torch.Tensor.cpu()`.

Moving data between the CPU and the GPU memories is far slower than moving it inside the GPU memory.

```
>>> u = torch.tensor([1, 2, 3])
>>> u.device
device(type='cpu')
>>> v = u.to('cuda') # copy of u
>>> v
tensor([1, 2, 3], device='cuda:0')
>>> v[0] = 5
>>> u
tensor([1, 2, 3])
>>> w = u.to('cpu') # this is u itself
>>> w
tensor([1, 2, 3])
>>> w[0] = 5
>>> u
tensor([5, 2, 3])
```


Notes

Here, changing `v` does not change `u` because `v` was a copy.
But `w` points to the same data as `u` because the latter was already on the device.

```
>>> m = torch.randn(10, 10)
>>> m.device
device(type='cpu')
>>> x = torch.randn(10, 100)
>>> q = m@x
>>> q.device
device(type='cpu')
>>> m = m.to('cuda')
>>> x = x.to('cuda')
>>> q = m@x # This is done on GPU (#0)
>>> q.device
device(type='cuda', index=0)
```

Since operations maintain the types and devices of the tensors, you generally do not need to worry about making your code generic regarding these aspects.

To explicitly create new tensors you can use a tensor's `new_*()` methods.

```
>>> u = torch.randn(3, 5, dtype = torch.float64)
>>> v = u.new_zeros(1, 2)
>>> v
tensor([[0., 0.]], dtype=torch.float64)
>>> w = torch.empty(3, 5, dtype = torch.float16,
...                 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]], device='cuda:1', dtype=torch.float16)
```


Notes

`new_*` methods allow to create tensors which are of the same type and on the same device as the object which we call them on:

- `u` is of type `float64` on the CPU, and so is `u.new_*`.
- `w` is of type `float16` on the GPU, and so is `w.new_*`.

Apart from `copy_()`, operations cannot mix different tensor types or devices:

```
>>> import torch
>>> x = torch.randn(3, 5)
>>> y = torch.randn(3, 5).to('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_()`.

Another exception to this rule are 0d tensors, which act as scalars and can be combined without device constraint.

Notes

One has to explicitly move the objects to the same device before making an operation.
`x + y` fails because `x` is located on the CPU while `y` is on the GPU.

The method `torch.Module.to(device)` moves all the parameters and buffers of the module (and registered sub-modules recursively) to the specified device.

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

Notes

A tensor can be moved to a device with `x = x.to(...)`. This is the case for minibatches and target values.
A module (network, criterion) can be moved to a device by simply calling `model.to(...)` and `criterion.to(...)`.

The method `torch.cuda.is_available()` returns a Boolean value indicating if a GPU is available, so a typical GPU-friendly code would start with

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

and then have some `device = device` in some places, and/or

```
model.to(device)
criterion.to(device)
train_input, train_target = train_input.to(device), train_target.to(device)
test_input, test_target = test_input.to(device), test_target.to(device)
```

Multiple GPUs with `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 `forward`s 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().__init__()
        self.m = m

    def forward(self, x):
        print('Dummy.forward', x.size(), x.device)
        return self.m(x)
```

```
x = torch.randn(50, 10)
model = Dummy(nn.Linear(10, 5))

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

x = x.to('cuda')
model.to('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
```

Notes

This little example shows that on a machine with two GPUs, the input batch of fifty samples is split in two batches of twenty-five samples.
`nn.DataParallel` allows to leverage multiple GPUs with a minimal change of the code, by simply wrapping the model.

References

D. C. Ciresan, U. Meier, L. M. Gambardella, and J. Schmidhuber. **Deep big simple neural nets excel on handwritten digit recognition**. CoRR, abs/1003.0358, 2010.

A. Krizhevsky, I. Sutskever, and G. Hinton. **Imagenet classification with deep convolutional neural networks**. In Neural Information Processing Systems (NIPS), 2012.

R. Raina, A. Madhavan, and A. Y. Ng. **Large-scale deep unsupervised learning using graphics processors**. In International Conference on Machine Learning (ICML), pages 873–880, 2009.

S. Shi, Q. Wang, P. Xu, and X. Chu. **Benchmarking state-of-the-art deep learning software tools**. CoRR, abs/1608.07249, 2016.