Using the GPU

For an introductory discussion of *Graphical Processing Units* (GPU) and their use for intensive parallel computation purposes, see GPGPU.

One of Theano's design goals is to specify computations at an abstract level, so that the internal function compiler has a lot of flexibility about how to carry out those computations. One of the ways we take advantage of this flexibility is in carrying out calculations on a graphics card.

There are two ways currently to use a gpu, one of which only supports NVIDIA cards (CUDA backend) and the other, in development, that should support any OpenCL device as well as NVIDIA cards (GpuArray Backend).

CUDA backend

If you have not done so already, you will need to install Nvidia's GPU-programming toolchain (CUDA) and configure Theano to use it. We provide installation instructions for Linux, MacOS and Windows.

Testing Theano with GPU

To see if your GPU is being used, cut and paste the following program into a file and run it.

```
from theano import function, config, shared, sandbox
import theano.tensor as T
import numpy
import time

vlen = 10 * 30 * 768  # 10 x #cores x # threads per core
iters = 1000

rng = numpy.random.RandomState(22)
x = shared(numpy.asarray(rng.rand(vlen), config.floatX))
f = function([], T.exp(x))
print(f.maker.fgraph.toposort())
t0 = time.time()
for i in range(iters):
    r = f()
t1 = time.time()
print("Looping %d times took %f seconds" % (iters, t1 - t0))
print("Result is %s" % (r,))
if numpy.any([isinstance(x.op, T.Elemwise) for x in f.maker.fgraph.toposort()]):
    print('Used the cpu')
else:
    print('Used the gpu')
```

The program just computes the `exp()` of a bunch of random numbers. Note that we use the `shared` function to make sure that the input *x* is stored on the graphics device.

If I run this program (in check1.py) with `device=cpu`, my computer takes a little over 3 seconds, whereas on the GPU it takes just over 0.64 seconds. The GPU will not always produce the exact same floating-point numbers as the CPU. As a benchmark, a loop that calls `numpy.exp(x.get_value())` takes about 46 seconds.

```
$ THEANO_FLAGS=mode=FAST_RUN,device=cpu,floatX=float32 python check1.py
[Elemwise{exp,no_inplace}(<TensorType(float32, vector)>)]
Looping 1000 times took 3.06635117531 seconds
Result is [ 1.23178029  1.61879337  1.52278066 ...,  2.20771813  2.29967761
  1.62323284]
Used the cpu

$ THEANO_FLAGS=mode=FAST_RUN,device=gpu,floatX=float32 python check1.py
Using gpu device 0: GeForce GTX 580
[GpuElemwise{exp,no_inplace}(<CudaNdarrayType(float32, vector)>),
HostFromGpu(GpuElemwise{exp,no_inplace}.0)]
Looping 1000 times took 0.638810873032 seconds
Result is [ 1.23178029  1.61879349  1.52278066 ...,  2.20771813  2.29967761
  1.62323296]
Used the gpu
```

Note that GPU operations in Theano require for now `floatX` to be *float32* (see also below).

Returning a Handle to Device-Allocated Data

The speedup is not greater in the preceding example because the function is returning its result as a NumPy ndarray which has already been copied from the device to the host for your convenience. This is what makes it so easy to swap in `device=gpu`, but if you don't mind less portability, you might gain a bigger speedup by changing the graph to express a computation with a GPU-stored result. The `gpu_from_host` op means "copy the input from the host to the GPU" and it is optimized away after the `T.exp(x)` is replaced by a GPU version of `exp()`.

```
from theano import function, config, shared, sandbox
import theano.sandbox.cuda.basic_ops
import theano.tensor as T
import numpy
import time

vlen = 10 * 30 * 768  # 10 x #cores x # threads per core
iters = 1000

rng = numpy.random.RandomState(22)
x = shared(numpy.asarray(rng.rand(vlen), 'float32'))
f = function([], sandbox.cuda.basic_ops.gpu_from_host(T.exp(x)))
print(f.maker.fgraph.toposort())
t0 = time.time()
for i in range(iters):
    r = f()
t1 = time.time()
print("Looping %d times took %f seconds" % (iters, t1 - t0))
print("Result is %s" % (r,))
print("Numpy result is %s" % (numpy.asarray(r),))
if numpy.any([isinstance(x.op, T.Elemwise) for x in f.maker.fgraph.toposort()]):
    print('Used the cpu')
else:
    print('Used the gpu')
```

The output from this program is

```
$ THEANO_FLAGS=mode=FAST_RUN,device=gpu,floatX=float32 python check2.py
Using gpu device 0: GeForce GTX 580
[GpuElemwise{exp,no_inplace}(<CudaNdarrayType(float32, vector)>)]
Looping 1000 times took 0.34898686409 seconds
Result is <CudaNdarray object at 0x6a7a5f0>
Numpy result is [ 1.23178029  1.61879349  1.52278066 ...,  2.20771813  2.29967761
  1.62323296]
Used the gpu
```

Here we've shaved off about 50% of the run-time by simply not copying the resulting array back to the host. The object returned by each function call is now not a NumPy array but a "CudaNdarray" which can be converted to a NumPy ndarray by the normal NumPy casting mechanism using something like `numpy.asarray()`.

For even more speed you can play with the `borrow` flag. See Borrowing when Constructing Function Objects.

What Can Be Accelerated on the GPU

The performance characteristics will change as we continue to optimize our implementations, and vary from device to device, but to give a rough idea of what to expect right now:

- Only computations with *float32* data-type can be accelerated. Better support for *float64* is expected in upcoming hardware but *float64* computations are still relatively slow (Jan 2010).
- Matrix multiplication, convolution, and large element-wise operations can be accelerated a lot (5-50x) when arguments are large enough to keep 30 processors busy.
- Indexing, dimension-shuffling and constant-time reshaping will be equally fast on GPU as on CPU.
- Summation over rows/columns of tensors can be a little slower on the GPU than on the CPU.
- Copying of large quantities of data to and from a device is relatively slow, and often cancels most of the advantage of one or two accelerated functions on that data. Getting GPU performance largely hinges on making data transfer to the device pay off.

Tips for Improving Performance on GPU

- Consider adding `floatX=float32` to your `.theanorc` file if you plan to do a lot of GPU work.
- Use the Theano flag `allow_gc=False`. See GPU Async capabilities
- Prefer constructors like `matrix`, `vector` and `scalar` to `dmatrix`, `dvector` and `dscalar` because the former will give you *float32* variables when `floatX=float32`.
- Ensure that your output variables have a *float32* dtype and not *float64*. The more *float32* variables are in your graph, the more work the GPU can do for you.
- Minimize tranfers to the GPU device by using `shared` *float32* variables to store frequently-accessed data (see `shared()`). When using the GPU, *float32* tensor `shared` variables are stored on the GPU by default to eliminate transfer time for GPU ops using those variables.
- If you aren't happy with the performance you see, try running your script with `profile=True` flag. This should print some timing information at program termination. Is time being used sensibly? If an op or Apply is taking more time than its share, then if you know something about GPU programming, have a look at how it's implemented in theano.sandbox.cuda. Check the line similar to *Spent Xs(X%) in cpu op, Xs(X%) in gpu op and Xs(X%) in transfer op*. This can tell you if not enough of your graph is on the GPU or if there is too much memory transfer.

- Use nvcc options. nvcc supports those options to speed up some computations: *-ftz=true* to flush denormals values to zeros., *–prec-div=false* and *–prec-sqrt=false* options to speed up division and square root operation by being less precise. You can enable all of them with the *nvcc.flags=–use_fast_math* Theano flag or you can enable them individually as in this example: *nvcc.flags=-ftz=true –prec-div=false*.
- To investigate whether if all the Ops in the computational graph are running on GPU. It is possible to debug or check your code by providing a value to *assert_no_cpu_op* flag, i.e. *warn*, for warning *raise* for raising an error or *pdb* for putting a breakpoint in the computational graph if there is a CPU Op.

GPU Async capabilities

Ever since Theano 0.6 we started to use the asynchronous capability of GPUs. This allows us to be faster but with the possibility that some errors may be raised later than when they should occur. This can cause difficulties when profiling Theano apply nodes. There is a NVIDIA driver feature to help with these issues. If you set the environment variable CUDA_LAUNCH_BLOCKING=1 then all kernel calls will be automatically synchronized. This reduces performance but provides good profiling and appropriately placed error messages.

This feature interacts with Theano garbage collection of intermediate results. To get the most of this feature, you need to disable the gc as it inserts synchronization points in the graph. Set the Theano flag `allow_gc=False` to get even faster speed! This will raise the memory usage.

Changing the Value of Shared Variables

To change the value of a `shared` variable, e.g. to provide new data to processes, use `shared_variable.set_value(new_value)`. For a lot more detail about this, see Understanding Memory Aliasing for Speed and Correctness.

Exercise

Consider again the logistic regression:

```
import numpy
import theano
import theano.tensor as T
rng = numpy.random

N = 400
feats = 784
D = (rng.randn(N, feats).astype(theano.config.floatX),
rng.randint(size=N,low=0, high=2).astype(theano.config.floatX))
training_steps = 10000

# Declare Theano symbolic variables
x = T.matrix("x")
y = T.vector("y")
w = theano.shared(rng.randn(feats).astype(theano.config.floatX), name="w")
b = theano.shared(numpy.asarray(0., dtype=theano.config.floatX), name="b")
x.tag.test_value = D[0]
y.tag.test_value = D[1]

# Construct Theano expression graph
p_1 = 1 / (1 + T.exp(-T.dot(x, w)-b)) # Probability of having a one
prediction = p_1 > 0.5 # The prediction that is done: 0 or 1
xent = -y*T.log(p_1) - (1-y)*T.log(1-p_1) # Cross-entropy
cost = xent.mean() + 0.01*(w**2).sum() # The cost to optimize
gw,gb = T.grad(cost, [w,b])

# Compile expressions to functions
train = theano.function(
            inputs=[x,y],
            outputs=[prediction, xent],
            updates=[(w, w-0.01*gw), (b, b-0.01*gb)],
            name = "train")
predict = theano.function(inputs=[x], outputs=prediction,
            name = "predict")

if any([x.op.__class__.__name__ in ['Gemv', 'CGemv', 'Gemm', 'CGemm'] for x in
        train.maker.fgraph.toposort()]):
    print('Used the cpu')
elif any([x.op.__class__.__name__ in ['GpuGemm', 'GpuGemv'] for x in
          train.maker.fgraph.toposort()]):
    print('Used the gpu')
else:
    print('ERROR, not able to tell if theano used the cpu or the gpu')
    print(train.maker.fgraph.toposort())

for i in range(training_steps):
    pred, err = train(D[0], D[1])

print("target values for D")
print(D[1])

print("prediction on D")
print(predict(D[0]))
```

Modify and execute this example to run on GPU with `floatX=float32` and time it using the command line `time python file.py`. (Of course, you may use some of your answer to the exercise in section Configuration Settings and Compiling Mode.)

Is there an increase in speed from CPU to GPU?

Where does it come from? (Use `profile=True` flag.)

What can be done to further increase the speed of the GPU version? Put your ideas to test.

Note

- Only 32 bit floats are currently supported (development is in progress).
- `Shared` variables with *float32* dtype are by default moved to the GPU memory space.
- There is a limit of one GPU per process.
- Use the Theano flag `device=gpu` to require use of the GPU device.
- Use `device=gpu{0, 1, ...}` to specify which GPU if you have more than one.
- Apply the Theano flag `floatX=float32` (through `theano.config.floatX`) in your code.
- `Cast` inputs before storing them into a `shared` variable.
- Circumvent the automatic cast of *int32* with *float32* to *float64*:
  - Insert manual cast in your code or use *[u]int{8,16}*.
  - Insert manual cast around the mean operator (this involves division by length, which is an *int64*).
  - Notice that a new casting mechanism is being developed.

Solution

GpuArray Backend

If you have not done so already, you will need to install libgpuarray as well as at least one computing toolkit. Instructions for doing so are provided at libgpuarray.

While all types of devices are supported if using OpenCL, for the remainder of this section, whatever compute device you are using will be referred to as GPU.

Warning

While it is fully our intention to support OpenCL, as of May 2014 this support is still in its infancy. A lot of very useful ops still do not support it because they were ported from the old backend with minimal change.

Testing Theano with GPU

To see if your GPU is being used, cut and paste the following program into a file and run it.

```
from theano import function, config, shared, tensor, sandbox
import numpy
import time

vlen = 10 * 30 * 768  # 10 x #cores x # threads per core
iters = 1000

rng = numpy.random.RandomState(22)
x = shared(numpy.asarray(rng.rand(vlen), config.floatX))
f = function([], tensor.exp(x))
print(f.maker.fgraph.toposort())
t0 = time.time()
for i in range(iters):
    r = f()
t1 = time.time()
print("Looping %d times took %f seconds" % (iters, t1 - t0))
print("Result is %s" % (r,))
if numpy.any([isinstance(x.op, tensor.Elemwise) and
              ('Gpu' not in type(x.op).__name__)
              for x in f.maker.fgraph.toposort()]):
    print('Used the cpu')
else:
    print('Used the gpu')
```

The program just compute `exp()` of a bunch of random numbers. Note that we use the `theano.shared()` function to make sure that the input *x* is stored on the GPU.

```
$ THEANO_FLAGS=device=cpu python check1.py
[Elemwise{exp,no_inplace}(<TensorType(float64, vector)>)]
Looping 1000 times took 2.6071999073 seconds
Result is [ 1.23178032  1.61879341  1.52278065 ...,  2.20771815  2.29967753
  1.62323285]
Used the cpu

$ THEANO_FLAGS=device=cuda0 python check1.py
Using device cuda0: GeForce GTX 275
[GpuElemwise{exp,no_inplace}(<GpuArray<float64>>), HostFromGpu(gpuarray)
(GpuElemwise{exp,no_inplace}.0)]
Looping 1000 times took 2.28562092781 seconds
Result is [ 1.23178032  1.61879341  1.52278065 ...,  2.20771815  2.29967753
  1.62323285]
Used the gpu
```

Returning a Handle to Device-Allocated Data

By default functions that execute on the GPU still return a standard numpy ndarray. A transfer operation is inserted just before the results are returned to ensure a consistent interface with CPU code. This allows changing the deivce some code runs on by only replacing the value of the `device` flag without touching the code.

If you don't mind a loss of flexibility, you can ask theano to return the GPU object directly. The following code is modifed to do just that.

```
from theano import function, config, shared, tensor, sandbox
import numpy
import time

vlen = 10 * 30 * 768  # 10 x #cores x # threads per core
iters = 1000

rng = numpy.random.RandomState(22)
x = shared(numpy.asarray(rng.rand(vlen), config.floatX))
f = function([], sandbox.gpuarray.basic_ops.gpu_from_host(tensor.exp(x)))
print(f.maker.fgraph.toposort())
t0 = time.time()
for i in range(iters):
    r = f()
t1 = time.time()
print("Looping %d times took %f seconds" % (iters, t1 - t0))
print("Result is %s" % (numpy.asarray(r),))
if numpy.any([isinstance(x.op, tensor.Elemwise) and
              ('Gpu' not in type(x.op).__name__)
              for x in f.maker.fgraph.toposort()]):
    print('Used the cpu')
else:
    print('Used the gpu')
```

Here the `theano.sandbox.gpuarray.basic.gpu_from_host()` call means "copy input to the GPU". However during the optimization phase, since the result will already be on th gpu, it will be removed. It is used here to tell theano that we want the result on the GPU.

The output is

```
$ THEANO_FLAGS=device=cuda0 python check2.py
Using device cuda0: GeForce GTX 275
[GpuElemwise{exp,no_inplace}(<GpuArray<float64>>)]
Looping 1000 times took 0.455810785294 seconds
Result is [ 1.23178032  1.61879341  1.52278065 ...,  2.20771815  2.29967753
  1.62323285]
Used the gpu
```

While the time per call appears to be much lower than the two previous invocations (and should indeed be lower, since we avoid a transfer) the massive speedup we obtained is in part due to asynchronous nature of execution on GPUs, meaning that the work isn't completed yet, just 'launched'. We'll talk about that later.

The object returned is a GpuArray from pygpu. It mostly acts as a numpy ndarray with some exceptions due to its data being on the GPU. You can copy it to the host and convert it to a regular ndarray by using usual numpy casting such as `numpy.asarray()`.

For even more speed, you can play with the `borrow` flag. See Borrowing when Constructing Function Objects.

What Can be Accelerated on the GPU

The performance characteristics will of course vary from device to device, and also as we refine our implementation.

This backend supports all regular theano data types (float32, float64, int, ...) however GPU support varies and some units can't deal with double (float64) or small (less than 32 bits like int16) data types. You will get an error at compile time or runtime if this is the case.

By default all inputs will get transferred to GPU. You can prevent an input from getting transferred by setting its tag.target attribute to 'cpu'.

Complex support is untested and most likely completely broken.

In general, large operations like matrix multiplication, or element-wise operations with large inputs, will be significatly faster.

GPU Async Capabilities

By default, all operations on the GPU are run asynchronously. This means that they are only scheduled to run and the function returns. This is made somewhat transparently by the underlying libgpuarray.

A forced synchronization point is introduced when doing memory transfers between device and host.

It is possible to force synchronization for a particular GpuArray by calling its `sync()` method. This is useful to get accurate timings when doing benchmarks.

Software for Directly Programming a GPU

Leaving aside Theano which is a meta-programmer, there are:

- **CUDA**: GPU programming API by NVIDIA based on extension to C (CUDA C)
    - Vendor-specific
    - Numeric libraries (BLAS, RNG, FFT) are maturing.
- **OpenCL**: multi-vendor version of CUDA
    - More general, standardized.
    - Fewer libraries, lesser spread.
- **PyCUDA**: Python bindings to CUDA driver interface allow to access Nvidia's CUDA parallel computation API from Python
    - Convenience:

      Makes it easy to do GPU meta-programming from within Python.

      Abstractions to compile low-level CUDA code from Python (`pycuda.driver.SourceModule`).

      GPU memory buffer (`pycuda.gpuarray.GPUArray`).

      Helpful documentation.
    - Completeness: Binding to all of CUDA's driver API.
    - Automatic error checking: All CUDA errors are automatically translated into Python exceptions.
    - Speed: PyCUDA's base layer is written in C++.
    - Good memory management of GPU objects:

      Object cleanup tied to lifetime of objects (RAII, 'Resource Acquisition Is Initialization').

      Makes it much easier to write correct, leak- and crash-free code.

      PyCUDA knows about dependencies (e.g. it won't detach from a context before all memory allocated in it is also freed).

  (This is adapted from PyCUDA's documentation and Andreas Kloeckner's website on PyCUDA.)
- **PyOpenCL**: PyCUDA for OpenCL

Learning to Program with PyCUDA

If you already enjoy a good proficiency with the C programming language, you may easily leverage your knowledge by learning, first, to program a GPU with the CUDA extension to C (CUDA C) and, second, to use PyCUDA to access the CUDA API with a Python wrapper.

The following resources will assist you in this learning process:

- **CUDA API and CUDA C: Introductory**
  - NVIDIA's slides
  - Stein's (NYU) slides
- **CUDA API and CUDA C: Advanced**
  - MIT IAP2009 CUDA (full coverage: lectures, leading Kirk-Hwu textbook, examples, additional resources)
  - Course U. of Illinois (full lectures, Kirk-Hwu textbook)
  - NVIDIA's knowledge base (extensive coverage, levels from introductory to advanced)
  - practical issues (on the relationship between grids, blocks and threads; see also linked and related issues on same page)
  - CUDA optimisation
- **PyCUDA: Introductory**
  - Kloeckner's slides
  - Kloeckner' website
- **PYCUDA: Advanced**
  - PyCUDA documentation website

The following examples give a foretaste of programming a GPU with PyCUDA. Once you feel competent enough, you may try yourself on the corresponding exercises.

Example: PyCUDA

```
# (from PyCUDA's documentation)
import pycuda.autoinit
import pycuda.driver as drv
import numpy

from pycuda.compiler import SourceModule
mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
  const int i = threadIdx.x;
  dest[i] = a[i] * b[i];
}
""")

multiply_them = mod.get_function("multiply_them")

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
        drv.Out(dest), drv.In(a), drv.In(b),
        block=(400,1,1), grid=(1,1))

assert numpy.allclose(dest, a*b)
print(dest)
```

Exercise

Run the preceding example.

Modify and execute to work for a matrix of shape (20, 10).

Example: Theano + PyCUDA

```
import numpy, theano
import theano.misc.pycuda_init
from pycuda.compiler import SourceModule
import theano.sandbox.cuda as cuda

class PyCUDADoubleOp(theano.Op):

    __props__ = ()

    def make_node(self, inp):
        inp = cuda.basic_ops.gpu_contiguous(
           cuda.basic_ops.as_cuda_ndarray_variable(inp))
        assert inp.dtype == "float32"
        return theano.Apply(self, [inp], [inp.type()])

    def make_thunk(self, node, storage_map, _, _2):
        mod = SourceModule("""
    __global__ void my_fct(float * i0, float * o0, int size) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if(i<size){
        o0[i] = i0[i]*2;
    }
  }""")
        pycuda_fct = mod.get_function("my_fct")
        inputs = [storage_map[v] for v in node.inputs]
        outputs = [storage_map[v] for v in node.outputs]

        def thunk():
            z = outputs[0]
            if z[0] is None or z[0].shape != inputs[0][0].shape:
                z[0] = cuda.CudaNdarray.zeros(inputs[0][0].shape)
            grid = (int(numpy.ceil(inputs[0][0].size / 512.)), 1)
            pycuda_fct(inputs[0][0], z[0], numpy.intc(inputs[0][0].size),
                       block=(512, 1, 1), grid=grid)
        return thunk
```

Use this code to test it:

```
>>> x = theano.tensor.fmatrix()
>>> f = theano.function([x], PyCUDADoubleOp()(x))
>>> xv = numpy.ones((4, 5), dtype="float32")
>>> assert numpy.allclose(f(xv), xv*2)
>>> print(numpy.asarray(f(xv)))
```

Exercise

Run the preceding example.

Modify and execute to multiply two matrices: *x* * *y*.

Modify and execute to return two outputs: *x* + *y* and *x* - *y*.

(Notice that Theano's current *elemwise fusion* optimization is only applicable to computations involving a single output. Hence, to gain efficiency over the basic solution that is asked here, the two operations would have to be jointly optimized explicitly in the code.)

Modify and execute to support *stride* (i.e. to avoid constraining the input to be *C-contiguous*).

Note

- See Other Implementations to know how to handle random numbers on the GPU.
- The mode *FAST_COMPILE* disables C code, so also disables the GPU. You can use the Theano flag optimizer='fast_compile' to speed up compilation and keep the GPU.