synkhronos Documentation

Release 1.0.0

Adam Stooke

Sep 26, 2017

Contents

CHAPTER 1

A Multi-GPU Theano Extension for Data Parallelism

Synkhronos is a Python package for accelerating computation of Theano functions under data parallelism with multiple GPUs. The aim of this package is to speed up program execution with minimum changes to user code. Variables and graphs are constructed as usual with Theano or extensions such as Lasagne. Synkhronos replicates the user-constructed functions and GPU-memory variables on all devices. The user calls these functions as in a serial program; parallel execution across all GPUs is automated. Synkhronos supports management of Theano shared variables across devices, either by reading/writing individually or through collective communications, such as all-reduce, broadcast, etc.

Data parallelism requires that functions can equivalently be computed over an entire data set, or alternatively over subsets of the data with the multiple results reduced finally. A common example in machine learning is the computation of a gradient as an expectation value over a minibatch of data (the minibatch can be sub-divided).

Function Batches and Slices

Synkhronos further extends the Theano interface by supporting indexed and sliced function calls. Variables contianing entire data sets can be passed to Synkhronos functions, with an optional input argument to tell which elements (selected by index in 0-th dimension) to use in the function call. This can also be done for implicit inputs.

If the input data set is too large to compute on within the device memory, another optional input argument sets the number of "slices" each worker uses to compute over its assigned data. Results are automatically accumulated over each input slice (each a separate call to the Theano function) within each worker before being reduced once back to the master. This is convenient for running validation or test measurement functions in machine learning, for example.

Under the Hood

When a Synkhronos function is called, it scatters the input data, executes the underlying Theano function simultaneously on all devices, and reduces the outputs back to the master process. Functions may also update Theano shared memory variables (GPU-memory) locally on each device. Collectives on Theano shared variables are provided through the NVIDIA Collective Communications Library (NCCL, via PyGPU), or through CPU-based mechanisms.

Using Multiprocessing, a separate python process is forked for each additional GPU. Explicit function inputs are scattered via OS shared memory. This facilitates greater speedup by minimizing and parallelizing memory copies. Data may be scattered to GPU memories ahead of time for implicit function inputs; this is advantageous for data used repeatedly, device memory permitting.

Barriers guard the execution, both start and finish, of any function or method that requires worker action. This provides a programming framework safe from race conditions and lends the package its name.

This package currently provides for single-node computing only.

Contents:

See the following pages for installation instructions, simple examples, and function reference. See the folder `Synkhronos/demos` for more complete examples including MNIST and ResNet-50. It is suggested to read the code and execute the demos to see printed results.

Installation

To view the source code and / or install:

```
git clone https://github.com/astooke/Synkhronos
cd Synkhronos
pip install .
```

Third party dependencies include `Theano` with its new GPU back-end, `libgpuarray`, `nccl` (v1) for collective GPU communications, `posix_ipc` for allocating shared memory, and `pyzmq` for other CPU-based communications.

PyPI package (possibly) forthcoming.

Currently Python3 compatible only.

The use of `posix_ipc` limits operating system compatibility–Windows is not supported.

Hint: Use Theano flags `device=cpu` and `force_device=True` (see *Importing Lasagne & GpuArray*).

Hint: Compile-lock contention that slows down multi-GPU initialization can be avoided by modifying `Theano\theano\gpuarray\dnn.py`. Where it initializes `version.v = None`, replace with the installed version of cuDNN, for example `version.v = 6020`.

Introductory Example

```
import numpy as np
import theano
import theano.tensor as T
import synkhronos as synk

synk.fork()
x = T.matrix('x')
y = T.vector('y')
z = T.mean(x.dot(y), axis=0)
f_th = theano.function(inputs=[x, y], outputs=z)
```

```
f = synk.function(inputs=[x], bcast_inputs=[y], outputs=z)
synk.distribute()

x_dat = np.random.randn(100, 10).astype('float32')
y_dat = np.random.randn(10).astype('float32')
x_synk = synk.data(x_dat)
y_synk = synk.data(y_dat)
r_th = f_th(x_dat, y_dat)
r = f(x_synk, y_synk)

assert np.allclose(r, r_th)
print("All assertions passed.")
```

Which returns, on a dual-GPU machine:

```
Synkhronos attempting to use 2 of 2 detected GPUs...
Using cuDNN version 6020 on context None
Preallocating 5677/8110 Mb (0.700000) on cuda1
Using cuDNN version 6020 on context None
Preallocating 5679/8113 Mb (0.700000) on cuda0
Mapped name None to device cuda1: GeForce GTX 1080 (0000:01:00.0)
Mapped name None to device cuda0: GeForce GTX 1080 (0000:02:00.0)
Synkhronos: 2 GPUs initialized, master rank: 0
Synkhronos distributing functions...
...distribution complete (0 s).
All assertions passed.
```

The program flow is:

1. Call `synkhronos.fork()`.
2. Build Theano variables and graphs.
3. Build functions through Synkhronos instead of Theano.
4. Call `synkhronos.distribute()`.
5. Manage input data / run program with functions.

In More Detail

Import Theano in CPU-mode, and `fork()` will initialize the master GPU in the main process and additional GPUs in other processes. All Theano variables thereafter are built in the master, as in single-GPU programs. `distribute()` replicates all functions, and their variables, in the additional processes and their GPUs.

A function's `inputs` will be scattered by splitting evenly along the 0-th dimension. In this example, data parallelism applies across the 0-th dimensions of the variable `x`. A function's `bcast_inputs` are broadcast and used wholly in all workers, as the variable `y` in the example.

All explicit inputs to functions must be of type `synkhronos.Data`, rather than numpy arrays. The underlying memory of these objects is in OS shared memory, so all processes have access to it. They present an interface similar to numpy arrays, see `demos/demo_2.py`.

The Synkhronos function is computed simultaneously on all GPUs, including the master. By default, outputs are reduced and averaged, so the comparison to the single-GPU Theano function result passes. Other operations are possible: *sum*, *prod*, *max*, *min*, or `None` for no reduction.

Distribute

After all functions are constructed, calling `distribute()` pickles all functions (and their shared variable data) in the master and unpickles them in all workers. This may take a few moments. Pickling all functions together preserves correspondences among variables used in multiple functions in each worker.

Currently, `distribute()` can only be called once. In the future it could be automated or made possible to call multiple times. Synkhronos data objects can be made before or after distributing, but only after forking.

Theano Shared Variable Example

This example demonstrates management of Theano shared variables in the master and workers. First, the setup, again dual-GPU:

```
synk.fork(2)
s_init = np.ones(2, dtype='float32')
x = T.matrix('x')
s = theano.shared(s_init, name='s')
f = synk.function([x], updates=[(s, T.sum(x * s, axis=0))])
synk.distribute()
x_dat = synk.data(np.array([[1, 1],
                            [2, 2],
                            [3, 3],
                            [4, 4]]).astype('float32'))
print("\ns initial:\n", s.get_value())

f.as_theano(x_dat.data)
print("\ns after Theano call:\n", s.get_value())
```

The resulting output is the correct answer:

```
s initial:
 [ 1.  1.]

s after Theano call:
 [ 10.  10.]
```

Continuing with a reset and a call to the Synkhronos function, we investigate results using `gather()`, one way to collect all shared values into the master:

```
s.set_value(s_init)
f(x_dat)
print("\nlocal s after reset and Synkhronos call:\n", s.get_value())

gathered_s = synk.gather(s, nd_up=1)
print("\ngathered s:\n", gathered_s)

synk.reduce(s, op="sum")
print("\nlocal s after in-place reduce:\n", s.get_value())

gathered_s = synk.gather(s, nd_up=1)
print("\ngathered s after reduce:\n", gathered_s)
```

```
local s after reset and Synkhronos call:
 [ 3.  3.]

gathered s:
```

```
 [[ 3.  3.]
 [ 7.  7.]]

local s after in-place reduce:
 [ 10.  10.]

gathered s after reduce:
 [[ 10.  10.]
 [  7.   7.]]
```

Lastly, to propagate the result to all workers and observe this effect, call the following:

```
synk.broadcast(s, s_init)
f(x_dat)
synk.all_reduce(s, op="sum")
gathered_s = synk.gather(s, nd_up=1)
print("\ngathered s after reset broadcast, Synkhronos call, "
    "and all-reduce:\n", gathered_s)
```

```
gathered s after local reset, broadcast, Synkhronos call, and all-reduce:
 [[ 10.  10.]
 [ 10.  10.]]
```

Notice the use of `broadcast()` to set the same values in all GPUs.

Notes on Collectives

Collectives can be called on any Theano shared variable used in a Synkhronos function. CPU- and GPU-based collectives are available through the same interface. Results of a GPU collective communication may be returned as a new GPU array in the master, but no collective can create a new array (not associated with a Theano shared variable) in a worker.

Synkhronos provides the averaging reduction operation. The reduce operation `avg` is not present in NCCL; Synkhronos uses `sum` and then multiplies by the reciprocal number of GPUs.

Theano Shared Variable Sizes

Beware that the `nccl` collectives assume the same shape variable on each GPU, but it is possible to have different shapes in Synkhronos. In particular, `gather` and `all_gather` may leave off data or add extra data without raising an exception–in this case use CPU-based gather operations. See `demos/demo_3.py` for more about manipulating GPU-variables in workers.

Deep Learning Examples

Two deep learning examples are provided in the `Synkhronos/demos/` directory, one using the MNIST dataset and the other training a ResNet-50 model. Both build models using Lasagne, and this portion of the code remains unchanged. The main change is the construction of the training function.

Update Rules

All first-order update rules (e.g. SGD, Adam) can be computed using multiple devices by the following sequence: 1) workers compute raw gradients on their local data, 2) all-reduce the raw gradient, 3) workers update parameters using

the combined gradient. The tasks of computing the gradient and updating the parameters are split into two Theano (Synkhronos) functions. Lasagne's update rules have been adapted in this fashion in `Synkhronos/extensions/updates.py`. The MNIST and ResNet examples show how to use these updates, and a similar tool for making a single callable training function is provided in `Synkhronos/extensions/train.py`

One speed enhancement for the above scheme is to write the gradients of all layers in a network into one flattened vector. Calling NCCL collectives once on a large variable is faster than making many calls over smaller variables. This flattening pattern is built in to the Synkhronos update rules, which automatically reshape the gradients in the parameter updates.

Data Management

A more subtle difference in the code is the data management. All data must first be written to `synkhronos.Data` objects, to be made available to all worker processes. Thereafter, the entire data objects can be passed to the training function, with the kwarg `batch` used to select the indexes to use. It is possible to pass in a list of (randomized) indexes, and each process will build its own input data from its assigned subset of these indexes. This is not only convenient for shuffling data, but also more efficient. In parallel, each worker process will perform its own memory copy inherent in excerpting a list of indexes from a numpy array. This is the pattern used in `lasagne_mnist/train_mnist_cpu_data.py` and `resnet/train_resnet.py`.

It is also possible to scatter a data set to GPU memories in Theano shared variables. The kwarg `batch_s` selects the indexes of these variables to use in a function call. This can be either one list (slice), to be used in all workers, or a list of lists (slices), one for each. All Theano shared variables to be subject to `batch_s` or slicing must be declared in function creation under the kwarg `sliceable_shareds`. The `lasagne_mnist/train_mnist_gpu_data.py` demo follows this pattern. Scaling and overall speed should improve relative to the CPU-data demo.

Note that the `batch` and `batch_s` kwargs work differently. Applied before scattering, `batch` selects from the whole dataset; `batch_s` applies after scattering and selects from workers' datasets. See `demos/demo_4.py` and `demos/demo_5.py` for examples.

Hint: It is not necessary for the length of a selected data input to be divisible by the number of GPUs. Data will be scattered as evenly as possible. Only when averaging is this not strictly mathematically correct; the results from all workers are averaged with equal weighting.

Other Notes

Scaling / Speedup

In the MNIST demos in particular, experiment with different batch sizes (a command line argument when calling as script) to see the effect on speedup. As the batch size grows, so should GPU utilization. Scaling will be best once the program is GPU-compute bound. (Of course this will affect the learning algorithm–see recent publications on successful large minibatch training.)

Easy Validation and Test

Validation and test functions can be performed conveniently by passing the entire validation or test set to the function call. Use the `num_slices` kwarg to automatically accumulate the results over multiple calls to the underlying Theano function, to avoid out-of-memory errors.

Importing Lasagne & GpuArray

Use the Theano flags `device=cpu,force_device=True` when placing `import lasagne` or `import theano.gpuarray` in file headers.

The reason this is needed is that it is not possible to fork after initializing a CUDA context and then use GPU functions in a sub-processes. Without the `force_device=True` flag, importing `theano.gpuarray` creates a CUDA context. `import theano.gpuarray` is called within `import lasagne`.

API Reference

`synkhronos.`**`fork`**(*n_parallel=None*, *use_gpu=True*, *master_rank=0*, *profile_workers=False*, *max_n_var=1000*, *max_dim=16*)

Forks a Python process for each additional GPU, initializes GPUs. Call before building any Theano GPU-variables or Synkhronos functions.

Parameters

- **`n_parallel`** (*`None, optional`*) – Number of GPUs to use (default uses all)
- **`use_gpu`** (*`bool, optional`*) – Inactive (possibly future CPU-only mode)
- **`master_rank`** (*`int, optional`*) – GPU to use in master process
- **`profile_workers`** (*`bool, optional`*) – If True, records cProfiles of workers (see `synkhronos/worker.py` for details)
- **`max_n_var`** (*`int, optional`*) – Max number of variables in a function call
- **`max_dim`** (*`int, optional`*) – Max number of dimensions of any variable

Returns Number of GPUs using.

Return type int

Raises

- `NotImplementedError` – If `use_gpu==False`
- `RuntimeError` – If already forked.

`synkhronos.`**`close`**()

Close workers and join their processes. Called automatically on exit.

`synkhronos.`**`function`**(*inputs*, *outputs=None*, *bcast_inputs=None*, *updates=None*, *givens=None*, *sliceable_shareds=None*, ***kwargs*)

Replacement for `theano.function()`, with a similar interface. Builds underlying Theano functions, including support for function slicing.

Parameters

- **`inputs`** – as in Theano, to be scattered among workers
- **`outputs`** – as in Theano, with option to specify reduce operation (see notes below)
- **`bcast_inputs`** – as inputs in Theano, to be broadcast to all workers
- **`updates`** – as in Theano, with option to specify reduct operation (see notes below)
- **`givens`** – as in Theano
- **`sliceable_shareds`** – any implicit inputs (Theano shared variables) acting as data-parallel data (i.e. to be subjected to the kwarg `batch_s` and /or to function slicing) must be listed here

- ****kwargs** – passed on to all internal calls to `theano.function()`

Reduce Operations: Outputs: May be specified simply as Theano tensor variables, as in normal Theano, or as two-tuples, as in (var, reduce-op), where reduce-op can be: "avg", "sum", "max", "min", "prod", or None. Default is "avg".

Updates: May be specified as a list of two-tuples, as in normal Theano, or may include triples, as in (var, update, reduce-op). Unlike for outputs, the reduce-op here applies only when using function slicing. Every slice is computed using the original values, and the update is accumulated over the slices. (This may impose some limits on the form of the update expression.) At the end of the function call, all updates are applied only locally, within each worker. This provides clear control to user over when to communicate.

Returns callable object, replacing a theano.Function

Return type sykhronos.function_module.Function

Raises

- `RuntimeError` – If Sykhronos not yet forked, or if already distributed
- `TypeError` – If incorrect format for arguments.
- `ValueError` – If entry in `sliceable_shareds` is not used in function, or for invalid reduce operation requested.

`synkhronos.`**`distribute`**`()`

Replicates all Synkhronos functions and their Theano shared variables in worker processes / GPUs. It must be called after building the last Synkhronos function and before calling any Synkhronos function.

It pickles all underlying Theano functions into one file, which workers unpickle. All Theano shared variable data is included, and correspondences between variables across functions is preserved. The pickle file is automatically deleted by a worker. The default file location is in the directory synkhronos/pkl/, but this can be changed by modifying `PKL_PATH` in synkhronos/util.py.

Raises `RuntimeError` – If not yet forked or if already distributed.

class `synkhronos.function_module.`**`Function`**(*inputs*, *bcast_inputs*, *to_cpu*, *return_list=True*, ***kwargs*)

Class of instances returned by `synkhronos.function()`.

`__call__`(**args*, *output_subset=None*, *batch=None*, *batch_s=None*, *num_slices=1*, ***kwargs*)

Callable as in Theano function.

When called, a Synkhronos function:

1. Assigns input data evenly across all GPUs,
2. Signals to workers to start and which function to call,
3. Calls the underlying Theano function on assigned data subset,
4. Collect results from workers and returns them.

Parameters

- ***args** (`Data`) – Normal data inputs to Theano function
- **output_subset** – as in Theano
- **batch** – indexes to select from scattering input data (see notes)
- **batch_s** – indexes to select from scattered implicit inputs (see notes)

- **num_slices** (*int*) – call the function over this many slices of the selected, scattered data and accumulate results (avoid out-of-memory)
- ****kwargs** (Data) – Normal data inputs to Theano function

Batching: The kwarg `batch` can be of types: (int, slice, list (of ints), numpy array (1-d, int)). It applies *before* scattering, to the whole input data set. If type int, this acts as data[:int].

The kwarg `batch_s` can be of type (slice, list (of ints), numpy array (1-d, int)) or a list of all the same type (one of those three), with one entry for each GPU. It applies *after* scattering, to data already residing on the GPU. If only one of the above types is provided, rather than a list of them, it is used in all GPUs.

In both `batch` and `batch_s`, full slice types are not supported; start and stop fields must be ints, step None.

Slicing: Function slicing by the `num_slices` kwarg applies within each worker, after individual worker data assignment. Results are accumulated within each worker and reduced only once at the end. Likewise, any updates are computed and accumulated using the original variable values, and the updates are applied only once at the end.

Raises `RuntimeError` – If not distributed or if synkhronos closed.

as_theano(**args*, ***kwargs*)
Call the function in the master process only, as normal Theano.

Parameters

- ***args** (data) – Normal inputs to the Theano function
- ****kwargs** (data) – Normal inputs to the Theano function

build_inputs(**args*, *force_cast=False*, *oversize=1.0*, *minibatch=False*, ***kwargs*)
Convenience method which internally calls `synkhronos.data()` for each input variable associated with this function. Provide data inputs as if calling the Theano function.

Parameters

- ***args** – data inputs
- **force_cast** (*bool, optional*) – see `synkhronos.data()`
- **oversize** (*float [1,2], optional*) – see `synkhronos.data()`
- **minibatch** (*bool, optional*) – see `synkhronos.data()`
- ****kwargs** – data inputs

The kwargs `force_cast`, `oversize`, and `minibatch` are passed to all calls to `synkhronos.data()`

Returns data object for function input.

Return type *synkhronos.data_module.Data*

name
As in Theano functions.

output_modes
Returns the reduce operations used to collect function outputs.

update_modes
Returns the reduce operations used to accumulate updates (only when slicing)

synkhronos.**data**(*value=None*, *var=None*, *dtype=None*, *ndim=None*, *shape=None*, *minibatch=False*, *force_cast=False*, *oversize=1*, *name=None*)

Returns a synkhronos.Data object, for data input to functions. Similar to a Theano variable, Data objects have fixed ndim and dtype. It is optional to populate this object with actual data or assign a shape (induces memory allocation) at instantiation.

Parameters

- **value** – Data values to be stored (e.g. numpy array)
- **var** (*Theano variable*) – To infer dtype and ndim
- **dtype** – Can specify dtype (if not implied by var)
- **ndim** – Can specify ndim (if not implied if not implied)
- **shape** – Can specify shape (if not implied by value)
- **minibatch** (*bool, optional*) – Use for minibatch data inputs (compare to full dataset inputs)
- **force_cast** (*bool, optional*) – If True, force value to specified dtype
- **oversize** (*int, [1,2], optional*) – Factor for OS shared memory allocation in excess of given value or shape
- **name** – As in Theano variables

Returns used for data input to functions

Return type synkhronos.Data

Raises TypeError – If incomplete specification of dtype and ndim.

class synkhronos.data_module.**Data**(*ID*, *dtype*, *ndim*, *minibatch=False*, *name=None*)

Type of object required as inputs to functions (instead of numpy arrays). May also be used as inputs to some collectives. Underlying memory is OS shared memory, for multi-process access. Presents a similar interface as a numpy array, with some additions and restrictions. The terms "numpy wrapper" and "underlying numpy array" refer to the same object, which is a view to the underlying memory allocation.

__getitem__(*k*)

Provided to read from underlying numpy data array, as in array[k]. Full numpy indexing is supported.

__len__()

Returns len() of underlying numpy data array.

__setitem__(*k*, *v*)

Provided to write to underlying numpy data array, as in array[k] = v. Full numpy indexing is supported.

alloc_size

Returns the size of the underlying memory allocation (units – number of items). This may be larger than the size of the underlying numpy array, which may occupy only a portion of the allocation (always starting at the same memory address as the allocation).

condition_data(*input_data*, *force_cast=False*)

Test the conditioning done to input data when calling set_value or synkhronos.data()

Parameters

- **input_data** – e.g., numpy array
- **force_cast** (*bool, optional*) – force data type

Returns numpy array of shape and dtype that would be used

Return type TYPE

data
Returns underlying numpy data array. In general, it is not recommended to manipulate this object directly, aside from reading from or writing to it (without changing shape). It may be passed to other python processes and used as shared memory.

dtype
Returns data type.

free_memory()
Removes all references to underlying memory (and numpy wrapper) in master and workers; the only way to shrink the allocation size.

minibatch
Returns whether this data is treated as a minibatch. When using the `batch` kwarg in a function call, all minibatch data inputs will have their selection indexes shifted so that the lowest overall index present in `batch` corresponds to index 0. (It is enforced that all data is long enough to meet all requested indexes.)

name
Returns the name (may be None)

ndim
Returns number of dimensions.

set_length(*length*, *oversize=1*)
Change length of underlying numpy array. Will induce memory reallocation if necessary (for length larger than current memory).

Parameters

- **length** (*int*) – New length of underlying numpy array
- **oversize** (*int, [1,2]*) – Used only if reallocating memory

> **Warning:** Currently, memory reallocation loses old data.

set_shape(*shape*, *oversize=1*)
Change shape of underlying numpy array. Will induce memory reallocation if necessary (for shape larger than current memory).

Parameters

- **shape** (*list, tuple*) – New shape of underlying numpy array
- **oversize** (*int, [1,2]*) – Used only if reallocating memory

> **Warning:** Currently, memory reallocation loses old data.

set_value(*input_data*, *force_cast=False*, *oversize=1*)
(Over)Write data values. Change length, reshape, and/or reallocate shared memory if necessary (applies eagerly in workers).

Parameters

- **input_data** – e.g. numpy array, fed into numpy.asarray()
- **force_cast** (*bool, optional*) – force input data to existing dtype of data object, without error or warning

- **oversize** (*int, [1-2]*) – Factor for oversizing memory allocation relative to input data size

Oversize applies only to underlying shared memory. The numpy array wrapper will have the exact shape of `input_data`.

shape
Returns shape of underlying numpy data array.

size
Returns size of underlying numpy data array.

`synkhronos.`**broadcast**(*shared_vars*, *values=None*, *nccl=True*)
Broadcast master's values (or optionally input values) to all GPUs, resulting in same values for Theano shared variables in all GPUs.

Parameters

- **shared_vars** (*Theano shared variables*) – one or list/tuple
- **values** (*None, optional*) – if included, must provide one for each shared variable input; otherwise existing value in master is used
- **nccl** (*bool, optional*) – If True, use NCCL if available

`synkhronos.`**scatter**(*shared_vars*, *values*, *batch=None*)
Scatter values across all workers. Values are scattered as evenly as possible, by 0-th index. Optional param `batch` can specify a subset of the input data to be scattered.

Parameters

- **shared_vars** (*Theano shared variables*) – one or list/tuple
- **values** (*synk.Data or numpy array*) – one value for each variable
- **batch** (*None, optional*) – Slice or list of indexes, selects a subset of the input data (by 0-th index) to scatter

`synkhronos.`**gather**(*shared_vars*, *nd_up=1*, *nccl=True*)
Gather and return values in Theano shared variables from all GPUs. Does not affect the values present in the variables.

Parameters

- **shared_vars** (*Theano shared variable*) – one, or list/tuple
- **nd_up** (*int, [0,1] optional*) – number of dimensions to add during concatenation of results
- **nccl** (*bool, optional*) – If True, use NCCL if available

Warning: If a Theano shared variable has different shapes on different GPUs (e.g. some have length one longer than others), then using NCCL yields slightly corrupted results. (Extra rows will be added or left off so the same shape is collected from each GPU.) CPU-based gather will always show accurately the actual values on each GPU, regardless of shape mismatch.

Returns gathered values

Return type array (or tuple)

synkhronos.**all_gather**(*shared_vars*, *nccl=True*)

All GPUs gather the values in each variable, and overwrites the variable's data with the gathered data (cannot change ndims).

Parameters

- **shared_vars** (*Theano shared variable*) – one, or list/tuple
- **nccl** (*bool, optional*) – If True, use NCCL if available.

synkhronos.**reduce**(*shared_vars*, *op='avg'*, *in_place=True*, *nccl=True*)

Reduce the values of the shared variables from all GPUs into the master. Worker's values are not affected; by default the master's value is overwritten.

Parameters

- **shared_vars** (*Theano shared variable*) – one, or list/tuple
- **op** (*str, optional*) – reduce operation (avg, sum, min, max, prod)
- **in_place** (*bool, optional*) – If True, overwrite master variable data, otherwise return new array(s)
- **nccl** (*bool, optional*) – If True, use NCCL if available

Returns if `in_place==False`, results of reduction(s)

Return type array (or tuple)

synkhronos.**all_reduce**(*shared_vars*, *op='avg'*, *nccl=True*)

Reduce the values of the shared variables across all GPUs, overwriting the data stored in each GPU with the result.

Parameters

- **shared_vars** (*Theano shared variable*) – one, or list/tuple
- **op** (*str, optional*) – reduce operation (avg, sum, min, max, prod)
- **nccl** (*bool, optional*) – If True, use NCCL if available

synkhronos.**set_value**(*rank*, *shared_vars*, *values*, *batch=None*)

Set the value of Theano shared variable(s) in one GPU.

Parameters

- **rank** (*int*) – Which GPU to write to
- **shared_vars** (*Theano shared variable*) – one, or list/tuple
- **values** (*synk.Data, numpy array*) – one value for each variable
- **batch** (*None, optional*) – slice or list of indexes, selects subset of input values to use

synkhronos.**get_value**(*rank*, *shared_vars*)

Get the value of Theano shared variable(s) from one GPU.

Parameters

- **rank** (*int*) – Which GPU to read from
- **shared_vars** (*Theano shared variable*) – one, or list/tuple

synkhronos.**get_lengths**(*shared_vars*)

Get lengths of Theano shared variable(s) from all GPUs.

Parameters **shared_vars** (*Theano shared variable*) – one, or list/tuple

synkhronos.**get_shapes**(*shared_vars*)
Get shapes of Theano shared variable(s) from all GPUs.

Parameters **shared_vars** (*Theano shared variable*) – one, or list/tuple

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

Symbols

A

B

C

D

F

G

M

N

O

R

S

U