Synkhronos: a Multi-GPU Theano Extension for Data Parallelism

Adam Stooke
University of California, Berkeley
adam.stooke@berkeley.edu

Pieter Abbeel
University of California, Berkeley
pabbeel@cs.berkeley.edu

Abstract

We present Synkhronos, an extension to Theano for multi-GPU computations leveraging data parallelism. Our framework automates execution and synchronization across devices, allowing users to write serial programs without risk of race conditions. NCCL is used for high-bandwidth inter-GPU communication. Further enhancements to the Theano's interface include input slicing (with aggregation) and input indexing, which perform common data-parallel computation patterns efficiently. One example use case is synchronous SGD, recently shown to scale well for some deep learning problems. When training ResNet-50, we achieve a near-linear speedup of 7.5x on an NVIDIA DGX-1 (8 GPUs), relative to Theano-only code running a single GPU in isolation. Yet Synkhronos remains general to any data-parallel computation programmable in Theano. By parallelizing individual Theano functions, our framework uniquely addresses a niche between manual multi-device programming and prescribed multi-GPU training routines.

1 Introduction

Theano [1] is the classic auto-differentiation Python package. It translates user's computation expressions into optimized code for fast CPU or GPU execution, and deep learning is among its most common uses. Yet it lacks built-in support for multi-device parallel programming, a clear means to faster computing. While significant speedups in multi-core CPU execution are possible,[1] we focus on use of GPUs.

1.1 Other Frameworks

More recent auto-differentiation frameworks do offer multi-GPU programming in Python, under data or model parallelism. For example, in Tensorflow [2], perhaps the most similar to Theano, users can program "multi-tower" computations[2] with more or less arbitrary device placements for computations. Other successful packages, such as PyTorch [3], Chainer [4], and MXNet [5] now also include tools to automatically use multiple GPUs in training neural networks. Specifically, these tools leverage the fact that SGD and its variants exhibit data parallelism, a simple and powerful motif for scaling [6, 7]. Data parallelism simply requires that a computation can correctly be performed by reducing (or gathering) the results of independent calls over arbitrary data subdivisions, and it has wide applicability to computations amenable to GPU acceleration [8].

[1] In our experience with some BLAS routines (called by Theano), calling separate single-threaded routines on each core in a data-parallel fashion can out-perform the multi-threaded routine using all cores on the full dataset.

[2] https://www.tensorflow.org/tutorials/using_gpu

1.2 Our Framework

Although it shared many elements with those examples, our framework implements a unique level of abstraction, addressing the gap between manual programming of individual devices and pre-fixed multi-GPU training routines. Synkhronos[3] automatically coordinates multiple devices to operate as if one, but in a way fully general to any data-parallel computation programmable in Theano. Usage primarily entails a few line-for-line code exchanges from Theano, and the user's programming responsibilities remain entirely within the original (serial) program.

At the same time, our framework is designed with scaling performance in mind. The multi-processed implementation ensures concurrent operation[4] for high numbers of devices, without requiring MPI. Inclusion of a Numpy-like shared memory interface and use of the NVIDIA Collective Communication Library (NCCL) combine to enable nimble management of both GPU and CPU memories. Lastly, computations are organized to minimize GPU-to-CPU transfers, wherever possible.

1.3 Paper Organization

In this paper we describe our framework from a systems perspective, as follows. First, relevant background on Theano's interface is provided, leading to responsibilities for Synkhronos. Then, we describe the techniques used for employing multiple devices, including matters of setup, synchronization, and memory management. We also introduce additional interface features useful for common data parallel compute patterns, including automated input slicing to avoid out-of-memory errors and input indexing for efficient memory sharing and shuffling. Finally, we investigate performance results in a representative case study before concluding with our future outlook.

2 Background: Design Requirements

To formulate the functionalities required of Synkhronos, we first review relevant features of the Theano interface it must interact with: functions and shared variables.

Theano Components The three main Theano components of interest are: functions, shared variables, and updates. The user constructs symbolic computation expressions in Theano, and then, using its "function" method, builds functions for computing them. Theano compiles optimized code (CPU or GPU), which the user calls later on data inputs. Example functions include forward passes or gradients through neural networks. A Theano shared variable has a data array associated with it which persists across functions and can exist on the GPU. One common use is weights of a neural network. Aside from returning outputs, funcions can modify, or "update" shared variables. A common use is a function which computes a gradient and updates network parameters in place.

Synkhronos Requirements In sum, Synkhronos must prepare Theano functions for each GPU and make the relevant shared variables available on each device. Then it must provide means to 1) run functions on all GPUs simultaneously and reduce the results and 2) modify each device's shared variables, including translating local updates into global ones.

3 Synkhronos Program Flow

Given what functionality Synkhronos must perform relative to Theano, we now describe our design decisions by way of a program overview. Data parallelism and synchronicity are two defining characteristics simplifying the design. Overall functionality is summarized in Figure 1, and example code appears in Appendix A. Discussion on data management is deferred to the following section.

3.1 Computation Setup

Fork Synkhronos automatically forks a separate Python process for each additional GPU. A barrier across all processes guards the start and finish of user calls, preventing race conditions. The user's process acts as master *and* as a parallel worker; it remains available for single-GPU Theano use.

[3] `https://github.com/astooke/Synkhronos`

[4] i.e., in the presence of Python's Global Interpreter Lock

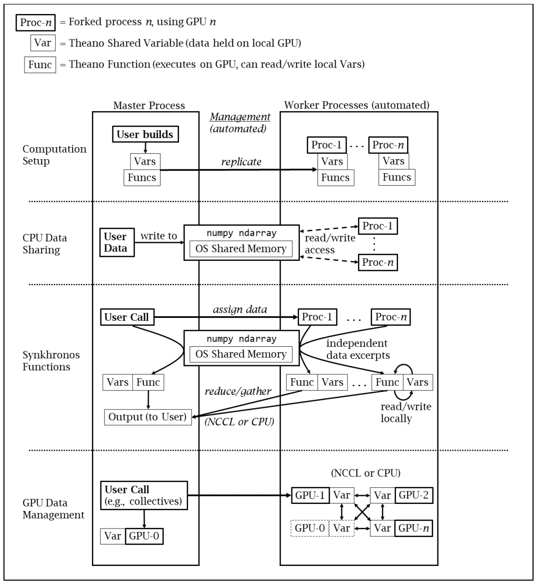


Figure 1: Synkhronos overview: automated worker management for synchronous computations and communications.

Build Functions After forking, the user builds expressions as usual with Theano (or other extensions, e.g. Lasagne [9] for neural networks), and shared variables are thereby allocated on the master GPU. Synkhronos' "function" method is used in place of Theano's for building multi-GPU functions; the main difference to the user is the ability to specify a reduce/gather operation to use for each output (Theano functions are built internally). Thereafter, all functions are distributed to the worker processes using Theano's function serialization, via pickling. When unpickling, all involved shared variables are automatically replicated on every GPU.

3.2 Running Computations

Function Calling Following distribution, the user calls Synkhronos functions just as Theano functions, but now all GPUs participate. A function call induces the sequence: 1) data inputs are scattered equally (as possible) across workers, 2) each device calls the same Theano function on its assigned data, and 3) results are collected back to the master process and returned to the user. For program clarity, function updates to shared variables apply only locally within each GPU.

Shared Variable Management Synkhronos provides several MPI-like collectives, such as *broadcast* and *all-reduce*, which can use NCCL (via the Pygpu package) for high-bandwidth inter-GPU communication (including NVLink). CPU-based collectives are also included, along with the means to get and set values on any individual GPU.

3.3 An Example: Synchronous SGD

Using SGD as an example, the computation sequence could be as follows: 1) a first Synkhronos function takes in data and computes the gradient, storing it in a shared variable (local on each GPU), 2) this variable is all-reduced using Synkhronos, and finally 3) a second Synkhronos function applies the update rule in each GPU using the combined gradient values. Such an adaptation of all update rules included in Lasagne is provided, such as momentum [10], RMSProp [11], and Adam [12], among others. These store the gradients of all variables into one array for faster inter-GPU communication.

4 Synkhronos Data Management

Synkhronos must transfer users' data inputs, typically in the form of Numpy arrays, from the user process to the worker processes. For fast performance, memory copies (within the CPU) and memory transfers (CPU-GPU) should be kept to a minimum. Synkhronos includes special data objects and communication collectives for this purpose.

4.1 Synkhronos Data Objects (for Function Inputs)

Input data is communicated to worker processes using operating system shared memory,[5] exposed to the user in a special data object. Every process has equal read-write access, so workers can concurrently feed their Theano functions by excerpting their assigned data in parallel. Synkhronos wraps each shared memory allocation in the Numpy array interface, allowing full array indexing for user reading and writing. Writing an entire data set (or a large chunk) into such an array obviates the need for any future memory copies, with the use of input indexing (see Section 5.2) if needed, e.g. for minibatches. The lowest tensor dimension is taken to represent independent data points for scattering inputs.[6] Inputs designated for broadcast are simply used as is.

The size of the underlying memory allocation can be made greater than the size of the outward facing Numpy array, preventing reallocation when growing or shrinking an array later. Special methods are provided for reshaping these arrays and freeing their memory. Conveniently, the Numpy array sub-object can be passed to other user Python processes to be read/written as shared memory. In sum, the aim is to present an interface which allows the user to optimize memory performance given the multi-process context.

4.2 Scattering to GPUs (for Shared Variables)

Programs with re-used inputs are often more efficient when input data is stored on the GPU, and the increased aggregate memory of multiple GPUs makes this feasible in a greater number of cases.[7] Synkhronos helps with its “scatter” collective, which evenly divides input arrays into shared variable storage across GPUs. It uses the same scattering scheme as for explicit inputs, i.e. by first tensor dimension. This is a convenient one-line replacement for setting multiple device memories manually (also possible), and it can make use of Synkhronos data objects and input indexing.

5 Synkhronos Extensions to Theano Function Interface

Synkhronos includes two extensions to the Theano interface for calling functions. These support common data-parallel compute patterns, simplifying user code and improving program efficiency.

5.1 Automated Input Slicing & Aggregation

The first extension is automated input data slicing, which can be used to avoid out-of-memory errors during computations too large to do in one call to a device. When slicing is used, workers compute their results by aggregating over multiple calls to the underlying Theano function, each using a subset of the worker’s assigned data. Slice results are aggregated in-place on the GPU. Worker results are reduced once back to the master process. If the function includes updates, all slices are computed using the original values, with updates accumulated and applied only once at the end. Automated slicing also applies to implicit inputs, i.e. data stored in advance on the GPU in shared variables (the memory requirement during computation is often much larger than the stored data). The user provides the number of slices to use as an optional input at each function call.

5.2 (Parallel) Input Indexing

The other extension is input indexing, whereby the user can specify which data elements to use during a function call. Designated indexes can make a slice or can be a list of (random) indexes. This is efficient for shuffling, which necessitates a memory copy, as each work can excerpt its own

[5] It is similar to a Multiprocessing RawArray (see section 17.2.2.6.1 at `https://docs.python.org/3.5/library/multiprocessing.html`, except allocatable before or *after* forking.

[6] Under C-style memory layout this provides contiguous memory assignments.

[7] Our original motivation was a big-batch, hessian-free learning algorithm, which achieved super-linear speedup by fitting each batch across multiple GPUs.

share of the indexes in parallel from the shared memory; no excess memory copies occur. Similar functionality is provided for inputs stored on-GPU in shared variables, where the same or different indexes can be applied to each device. Automated input slicing applies internally in each worker, after input indexing has determined assignments.

6 Scaling Performance

We explored an example case study to answer whether Synkhronos can achieve linear scaling on a modern supervised learning problem. We timed the training of a ResNet-50 [13] model using SGD on an NVIDIA DGX-1 (8x Tesla P100 GPUs, NVLink). Several gigabytes of synthetic images were generated as random single-precision arrays of dimension 3x224x224, and each timing run consisted of the same number of epochs over the entire data set, lasting at least several minutes.

The base case was a Theano-only program running on a single GPU with the rest of the machine idle, using a batch size of 64 images. We ran separate base timings for data shuffling or not. Results from several multi-GPU configurations are shown in Figure 2, all obtained using Synkhronos. No input slicing was used, but input indexing was, and all input data was passed as explicit inputs to the function call (i.e., not stored in advance on the GPU). The network update scheme was as described in Section 3.3.

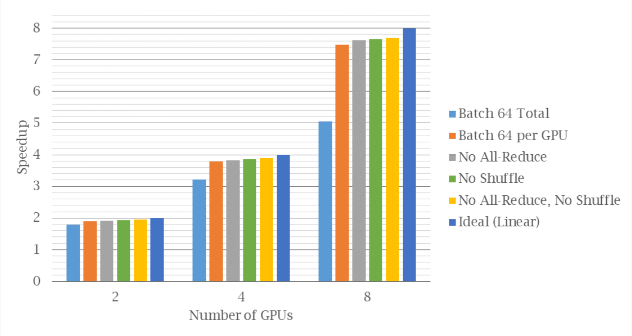


Figure 2: Speedups of ResNet-50 training relative to single-GPU, Theano-only program, with ablations.

6.1 End-to-End Training Timing

Batch - 64 The first test explored the scaling while keeping the algorithm perfectly unchanged. The total batch size remained fixed at 64, so the amount of data processed by each GPU in each minibatch scaled down with the number of GPUs. This yielded sub-linear scaling above 2 GPUs, as GPU utilization decreased, with a speedup only slightly greater than 5x when using all 8 GPUs.

Batch - 64 × #GPU The remainder of the tests used an algorithm modified by scaling up the minibatch size so that each GPU processes 64 images at every call, following the technique successfully applied in [14] to train to convergence on ImageNet in under one hour. This improved the scaling of the fully-communicating algorithm up to 7.5x with shuffling and 7.6x without, as compared against their own respective base cases–nearly linear, as seen in Figure 3. The base Theano code processed 1.75 minibatches per second, giving training speeds for 1-GPU Theano and 8-GPU Synkhronos of roughly 110 and 830 images per second, respectively.

As shown in Figure 2, turning off the gradient all-reduction improved scaling only slightly, to 7.6x with shuffling and nearly 7.7x without. This confirmed a small cost of communication compared to computation in this scenario (Synkhronos permits any intermediate frequency of communication).

Another view is to measure using the 2-GPU case as the baseline, i.e. as 2x speed, which highlights scaling alone by de-emphasizing any fixed overhead in Synkhronos. Under this comparison, both the shuffle and non-shuffle algorithms achieved >7.8x speeds using 8 GPUs. This indicates high likelihood for good scaling up to greater numbers of GPUs.

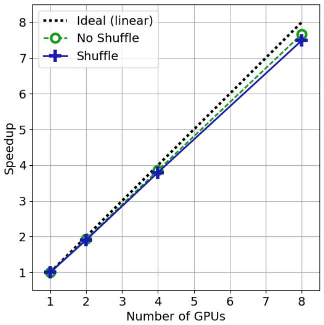


Figure 3: Speedup (near-linear) of ResNet-50 training relative to single-GPU, Theano-only program; with batch scaling.

6.2 Detailed Profiling

Lastly, we examined the main use case in more detail in an attempt to understand scaling imperfections. These tests included full communication, data shuffling, and batch scaling, and they used CUDA launch blocking for CPU-GPU synchronization for proper profiling. Table 1 contains example timings of the training loop's main elements, comparing Theano-only code against 8-GPU Synkhronos.

Starting from the bottom of the table, the all-reduce provided through NCCL added less than one percent in overhead. The exact timing of shuffling and computing naturally varied from call to call, leading to a slight slowdown from the straggler effect, in this case roughly 2%. Shuffling scaled well, to 7.9x, leaving some uncertainty as to the source of diminished scaling in the end-to-end tests. Further profiling gathered through Theano indicated a 1.8% time overhead for CPU-to-GPU data transfer within the Theano function. This increased to 3.6% in the 8 GPU case, as expected on the given hardware, accounting for much of the loss to 7.7x in the Theano function calls. Overall, we measured near-linear scaling of 7.5x, matching the sum of these components' contributions.

Table 1: Detailed profiling, ResNet-50 SGD training (CUDA launch blocking enabled)

	Time (s)		
Item	Theano	Synk-8	Scaling
Total	395.1	52.6	7.5
Theano Function	382.5	49.5	7.7
Shuffle	12.6	1.6	7.9
Straggler Effect	–	1.0	–
All-Reduce Gradient	–	0.46	–

7 Conclusion

We have presented Synkhronos, an extension to Theano for computing with multiple devices under data parallelism. After detailing the framework and functionality, we demonstrated near-linear speedup on a relevant deep learning example, training ResNet-50 with 8 GPUs on a DGX-1. The design emphasizes easy migration from single- to multi-device programming by the user while maintaining full generality to any data parallel computation. It includes flexible tools for efficient memory management, yet currently remains limited to single-node computing. Lastly, because it is written entirely in Python, the package is more widely accessible to modification by interested users. We refer to the code repository[8] for further examples. We hope that this package will accelerate the work of researchers and developers who use it, and that it may contribute helpful concepts for multi-device interfaces for other performance computing frameworks going forward.

[8] https://github.com/astooke/Synkhronos

Acknowledgments

We thank Frédéric Bastien, Pascal Lamblin, and Arnaud Bergeron for discussions and pointers throughout development, and the entire Theano development team for their base of work. Thanks again to Frédéric and to Carlos Florensa, Peter (Xi) Chen, and Anushree Saxena for helpful comments on the manuscript. The DGX-1 used for this research was donated by the NVIDIA Corporation. Adam Stooke gratefully acknowledges the support the Fannie and John Hertz Foundation.

References

[1] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016. URL `http://arxiv.org/abs/1605.02688`.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

[3] Pytorch Development Team. Pytorch. URL `https://github.com/pytorch/pytorch`.

[4] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*, 2015. URL `http://learningsys.org/papers/LearningSys_2015_paper_33.pdf`.

[5] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015. URL `http://arxiv.org/abs/1512.01274`.

[6] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29(12): 1170–1183, December 1986. ISSN 0001-0782. doi: 10.1145/7902.7903. URL `http://doi.acm.org/10.1145/7902.7903`.

[7] Guy E Blelloch. *Vector models for data-parallel computing*, volume 356. MIT press Cambridge, 1990.

[8] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008. ISSN 1542-7730. doi: 10.1145/1365490.1365500. URL `http://doi.acm.org/10.1145/1365490.1365500`.

[9] Sander Dieleman, Jan Schlüter, Colin Raffel, Eben Olson, Søren Kaae Sønderby, Daniel Nouri, et al. Lasagne: First release., August 2015. URL `http://dx.doi.org/10.5281/zenodo.27878`.

[10] Yurii Nesterov. A method of solving a convex programming problem with convergence rate O(1/sqr(k)). *Soviet Mathematics Doklady*, 27:372–376, 1983. URL `http://www.core.ucl.ac.be/~{}nesterov/Research/Papers/DAN83.pdf`.

[11] T. Tieleman and G. Hinton. RMSprop Gradient Optimization. URL `http://www.cs.toronto.edu/~{}tijmen/csc321/slides/lecture_slides_lec6.pdf`.

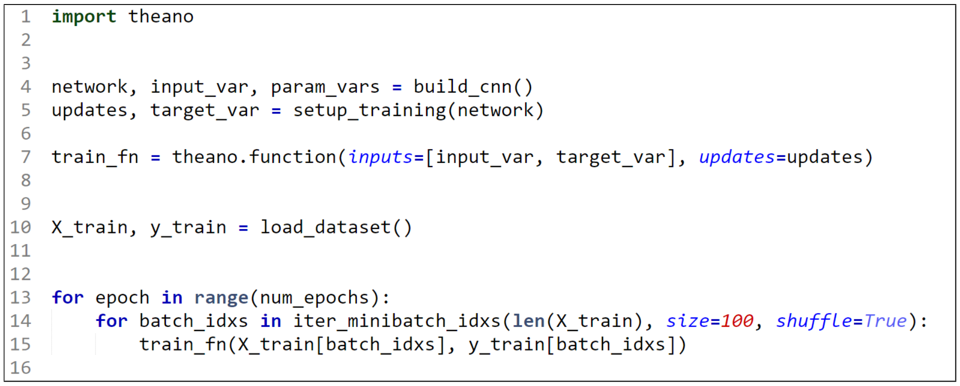
[12] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. URL `http://arxiv.org/abs/1412.6980`.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL `http://arxiv.org/abs/1512.03385`.

[14] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour, 2017.

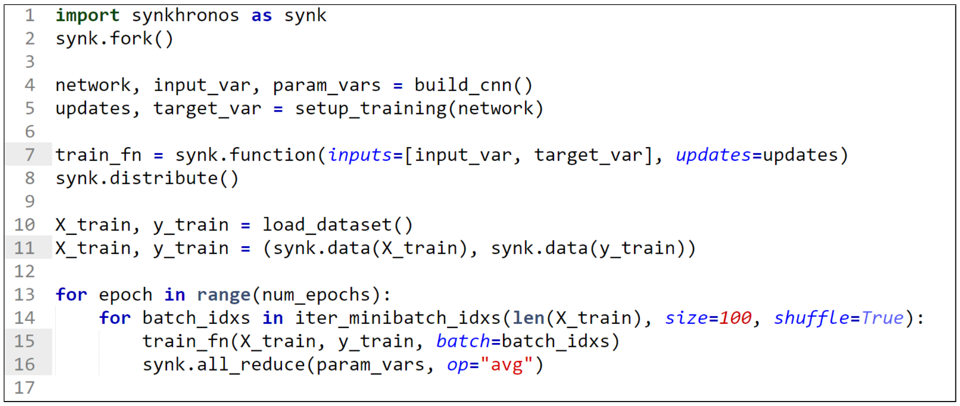
A Code Example

In the figures below we present a simple code example for running stochastic gradient descent in Theano, which is revised into Synkhronos. In the Theano example, the functions `build_cnn()` and `setup_training()` will contain other Theano code (or possibly Lasagne), which is left untouched in the Synkhronos program. The `fork()` command automatically uses all GPUs if not otherwise specified, and the `distribute()` command replicates all functions and Theano shared variables on the worker GPUs. Remaining line numbers where Synkhronos is active are highlighted. The `data()` command is used here to write the training data to operating system shared memory (once). The main change in the training loop is the call to all-reduce the network parameters, which in the case of simple SGD preserves the algorithm. Further examples and demonstrations can be found at the code repository.



```python
import theano


network, input_var, param_vars = build_cnn()
updates, target_var = setup_training(network)

train_fn = theano.function(inputs=[input_var, target_var], updates=updates)


X_train, y_train = load_dataset()


for epoch in range(num_epochs):
    for batch_idxs in iter_minibatch_idxs(len(X_train), size=100, shuffle=True):
        train_fn(X_train[batch_idxs], y_train[batch_idxs])

```

Figure 4: Theano program for single-GPU SGD.



```python
import synkhronos as synk
synk.fork()

network, input_var, param_vars = build_cnn()
updates, target_var = setup_training(network)

train_fn = synk.function(inputs=[input_var, target_var], updates=updates)
synk.distribute()

X_train, y_train = load_dataset()
X_train, y_train = (synk.data(X_train), synk.data(y_train))

for epoch in range(num_epochs):
    for batch_idxs in iter_minibatch_idxs(len(X_train), size=100, shuffle=True):
        train_fn(X_train, y_train, batch=batch_idxs)
        synk.all_reduce(param_vars, op="avg")

```

Figure 5: Synkhronos program for multi-GPU SGD.