Baechi PyTorch System Design (Not peer reviewed)

Chirag Shetty
University of Illinois at Urbana-Champaign
cshetty2@illinois.edu

ACM Reference Format:
Chirag Shetty. 2022. Baechi PyTorch System Design (Not peer reviewed). In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Note: This document has not been reviewed and may contain incorrect information.

1 Baechi Pytorch (B-P)

PyTorch is one of the mainstay frameworks used in ML programming today. It has particularly seen wide adoption in research communities and among citizen data scientists. This is primarily due to PyTorch 's emphasis on usability, ease of programming, and experimentation. These PyTorch users stand to benefit greatly from Baechi's fast placement generation which favors experimentation. Thus we build Baechi over PyTorch, enabling a single-GPU code to scale to a multi-GPU Model Parallel one with only a few lines of code change.

At a high-level, the design is similar to the Tensorflow implementation. The input model is profiled to obtain a representative graph, which is then passed to the placer to get a device placement for the nodes according to the algorithm chosen (m-SCT/m-ETF/m-Topo). However, actualizing the generated placement is challenging given the low support PyTorch offers for cross-device communication and synchronization. It is discussed in greater detail in Section 1.2.

We automate this process of realizing an arbitrary device placement for any given graph in PyTorch. To the best of our knowledge, it is the first system to do so. We do it with the following objectives:

(1) Have PyTorch faithfully follow the placement and execution plan generated by Baechi 's algorithm. The short step times promised by Baechi is contingent on actually executing the computations and communications in the same order as the Execution Simulator (Section **??**)
(2) Minimal and non-intrusive: satisfying (1) should not require the user to make significant code changes or model rewriting. B-P outputs a distributed version of the input model which can be used as a drop-in replacement in the existing training scripts (Fig 4)
(3) Retain PyTorch's programming style - It is a key feature and common practice to include python statements within PyTorch models. These include input modification functions, print, control loops or model specific functions like mask generation routines in Transformer models. We do not require the developer to think or program any differently in order to use B-P.

This automation when combined with Baechi 's placement algorithm offers a powerful tool to build and scale large models without much developer effort. Section 1.1 introduces basic features of PyTorch. Section 1.2 highlights the challenges of implementing model parallelism (MP) using PyTorch. In Section 1.3 we present the design of B-P. Admittedly, we have had to make tactical sacrifices and more work is needed to support some use-cases. These limitations are discussed in Section 1.4.

1.1 Pytorch : Background

To do effective multi-GPU mode splitting, it is necessary to understand the internal workings of PyTorch. Now we give a brief exposition on PyTorch features [1] relevant to our design.

1.1.1 Layers: Typically, Models are built by composing `layers`. Each layer contains its parameters (e.g., weights of a linear layer) or other layers used to compose that layer. Further, each layer has a `forward()` method, which defines how the layer modifies the input. `forward()` of a larger layer is a chain of `forward()` of its constituent layers. But it also often includes usual python statements like scaling the inputs, printing intermediate outputs, control loops, plotting, etc. in `forward()`. Commonly used basic layers, like Linear, Convolutional, LSTM, etc. are inbuilt in the library. Naturally such layers form the placement unit, i.e., the nodes of the graph in B-P.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1.1.2 Control and data flow separation:

There is a strong separation between the control and data flow of a Pytorch program. The control flow is handled like any normal Python program, wherein the host CPU eagerly executes the instructions, resolves branches etc. The result is creation of tensors and sequence of operations on them i.e the data flow. This sequence then runs in parallel with the control flow, on the CPU, GPUs or both.

When run on a GPU, the host CPU code submits the operators as CUDA kernels to the GPU's stream - a FIFO queue. The kernels are then invoked on the GPU asynchronously, unless if there is a need to synchronize the host CPU thread with the GPU stream. Usually, the host CPU thread runs far ahead of the GPU kernel execution (as illustrated in Fig8-2.b). And this is key to Pytorch's good performance in spite of having a dynamic runtime. The abstraction of layers, allows for easy modification of the data flow without affecting the control flow. We utilize this property in our design as explained in Section 1.3.2.

1.1.3 Model Splitting:

A tensor or a layer can be moved from its current device to a device `d` using the `.to(d)` primitive. Placing a layer on a device is equivalent to moving all the parameters of that layer to the device. If the user places a layer of the model on *gpu0*, then they are responsible for ensuring that the input tensor to it is also on *gpu0*. This needs to be programmed into the `forward()` of the main model. Otherwise, the program exits with an error. Implementing efficient model parallelism in PyTorch is challenging because the user must manage all the requisite tensor transfers across the devices. This is in contrast to Tensorflow (TF), where it is enough to assign layers to devices. TF runtime handles the data movement across devices transparently, without any developer intervention. We build such a capability over PyTorch as described in Section 1.3.3.

1.2 Why is Model Parallelism hard in PyTorch?

PyTorch optimizes operation assuming a single-device execution. When using multiple devices to run a model, the users are expected to introduce appropriate data-transfers and synchronizations. Thus, multi-device training often requires careful reasoning and heavy modifications of the model code. This has led to projects like PyTorch-Distributed [2] which automates and simplifies data parallel training. B-P does the same for model parallelism (MP). To demonstrate the engineering effort that B-P saves, we show the challenges in implementing efficient MP in PyTorch, without a tool like B-P.

We will do so by walking through a toy example of how a developer might go about splitting a very simple model across only 2 devices. Refer to Figure 1. The model, shown in Fig. 1(1), has three layers, each of which multiplies the input tensor by a constant - which is the layer's parameter. Fig. 1(1) also shows the single-device code for the model. The code includes a python statement that scales the input by 2 (line 4).

1.2.1 Challenge 1: Poor support for efficient cross-device data transfers:

Now suppose that the developer wants to exploit the inherent parallelism in the model by splitting the two branches across *gpu0* and *gpu1*. Fig. 1(2a) shows the modification one must make to the code. Layers must be placed on appropriate devices (lines 1-3) and intermediate results must be moved accordingly. The scaled input must be moved to *gpu1* (line 8) and a result of `m3` must be moved back to *gpu0* (line 10). This highlights the effort involved in going from one GPU to many.

Secondly, this modification is actually worse than a single-device execution. In Fig. 1(2a), operators for line 4-7 are put on *gpu0*'s stream, then `x` is moved from *gpu0* to *gpu1* on line 8 and line 9 is submitted to *gpu1*. We wanted `m3` to execute on *gpu1* in parallel with `m1` , `m2` on *gpu0*. The actual execution however is as shown in Fig. 1(2b). To ensure correctness, `.to()` conservatively blocks the streams of both sending and receiving device until all the kernels submitted to them so far are completed. This head-of-line blocking is very often unnecessary. Kernel submitted to *gpu1* in line 8 waits until kernels of line 6,7 are completed on *gpu0*, although it need not wait. Thus this straightforward code change will result in a very inefficient implementation.

1.2.2 Challenge 2: code modifications are not obvious and require global reasoning with no systematic rules:

An observant developer, aware of the above problem, would have written `x.to(1)` earlier in the code. Fig. 1(3a) shows the code and Fig. 1(3b) the execution. While this does exploit the parallelism, manually finding such appropriate data transfer points is hard. Further, no general rule can be formulated for the same, since it is dependent on actual execution times of the kernels. For instance, we can choose to transfer out the outputs of a layer as soon as they are ready. But this may stall the sender device unnecessarily if the receiving device's stream has a lot of large kernels pending.

Fortunately, using multiple streams within a device offers a solution. This enable us to execute computation and communication independently. It is quite common to use streams for efficiency by overlapping computation and communication [2][3]. However, as we just demonstrated, use of streams is necessary for efficient model parallelism, even if the communication took no time. But use of multiple streams,

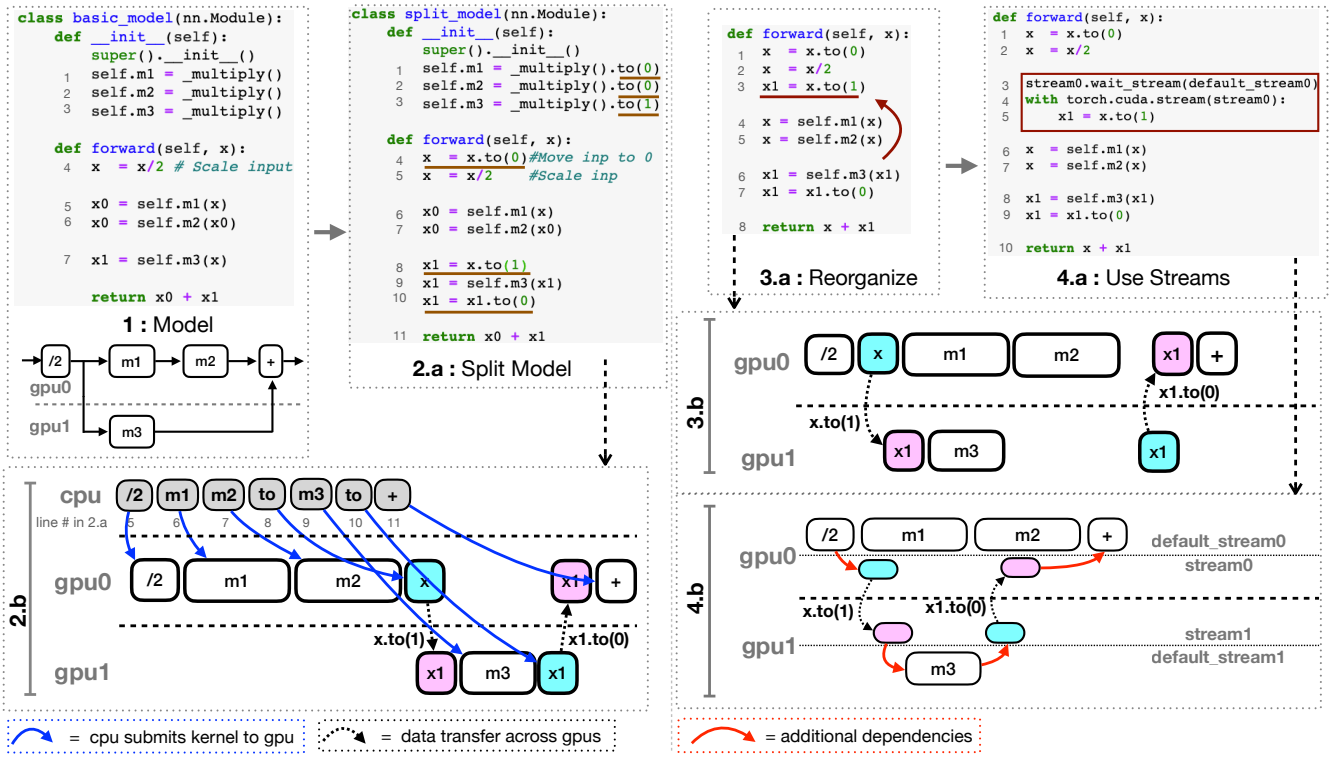


Figure 1: Challenges of Model Parallelism in Pytorch

if not done correctly, can introduce hard-to-debug bugs as we show now.

1.2.3 Challenge 3: Use of streams requires synchronizations and can cause subtle errors:

Fig. 1(4a) shows the modified code. For brevity, a stream is only defined for *gpu0* to *gpu1* transfer of `x` (lines 3-5). Fig. 1(4b) shows the execution, where streams have been defined on both devices. Correct implementation requires introducing several new dependencies (red solid arrows) at each transfer. For instance in Fig. 1(4a), without making `stream0` wait as on line 3, `(x)` will be transferred to *gpu1* on `stream0` even as *gpu0* is modifying `(x)` according to kernels from line 1-2. The final output will be wrong, even though the program runs smoothly. On using multiple streams, the developer must add such synchronizations while also not causing unnecessary blocking. We discuss in detail in Section 1.3.3. This execution (Fig. 1(4b)) exploits parallelism and has the optimal execution time. The output, however, can still be wrong due to bad interactions with PyTorch's memory allocator. Because different streams run independently, dangling references may be created as follows:

PyTorch assumes a single-device single-stream execution and maintains one memory pool per stream. CPU may submit an operation on a tensor to a GPU stream. Once done, it may issue instruction to free-up the tensor from the GPU memory and then reuse the memory for a new tensor. Asynchronous CPU-GPU execution means a tensor may be freed by the CPU, while it is still being used in some operation on the GPU. In a single stream program it is fine, because GPU stream maintains the order in which CPU submits operations to it. The memory deallocation will be after all uses of that tensor in that stream. But in a multi-device-multi-stream setting, freeing tensors in one stream while still being used in other streams can cause errors. And it is very likely to happen given PyTorch 's efficient garbage collector which frees a tensor immediately after its reference count becomes zero. In Fig. 1(4a), the tensor `x` maybe freed and reused on *gpu0*, while `x1=x.to(1)` is still in progress in the separate stream. When the host code executes lines 6, reference to old value of `x` reaches 0. The instruction to free the associated memory on *gpu0* is immediately added to *gpu0*'s default stream. When the default stream reaches this point, the deallocation of `x` may precede or overlap with `x1=x.to(1)` still going on in `stream0`. Thus `x1` may not be the same tensor as `x`. This program will run without errors , but the final output will not be correct. In reality, the effects are even more stealthy when in-place operations like ReLU are used or when models

are run in inference mode (PyTorch automatically frees the intermediate outputs).

Model parallelism even with this simple toy example PyTorch graph is not straightforward. Applying model parallelism to large graphs composed of 100's layers requires not only enormous engineering efforts, but is also prone to errors resulting in incorrect outputs. Thus large scale model parallelism in PyTorch quickly becomes unviable and is not even an option in scaling models to a multi-device setting. But by automating this process, we make it a competent choice for large scale training. We thoroughly scrutinize and resolve the side-effects of multi-stream programming with PyTorch so that the developer does not have to.

1.3 Design

B-P consists of three main phases: profiling, placement generation, and placement execution. The input model is profiled and converted to a NetworkX graph with layers as nodes. The NetworkX graph is then passed to Baechi's Placer. The Placer outputs device placement results of each layer and a topological order in which the layers must be executed on those devices. This process of graph generation, profiling and generating placement is similar to the Tensorflow implementation, except that we build our own profiler, which is described in Section 1.3.1.

The model and the Placer outputs are then passed to the Assigner, which actually moves the layers to the allotted devices and modifies the model to enable distributed execution. This is explained in Section 1.3.2. The returned distributed model can then be used simply as a drop-in replacement of a single-gpu model code in existing training scripts. The Assigner's multi-stream communication protocol (described in Section 1.3.3) addresses the challenges highlighted in the previous section. Finally, in Section 1.3.4 we show the model modifications required for B-P to work.

Note that as long as the forward functions are executed in the right order and synchronized correctly, PyTorch's autograd ensure that the backprop correctly executes in the reverse of this sequence. It automatically manages input-output dependencies, synchronizations, stream affinity etc in the reversed order. This allows us to focus exclusively on the forward graph, like we did in the TF implementation (Section **??**)

1.3.1 Profiling: In Tensorflow , models are given as static graphs[1] and we used the standard Tensorflow profiling tool to measure time and memory requirements of each node (Section **??**). In PyTorch the graph is constructed at runtime by tracking the operations on tensors. And PyTorch's default profiler provides information at the granularity of GPU kernels executed rather than layers [4]. While one could use some correspondences between kernels and layers, it is simpler to build a profiler from scratch, which measures time using hooks as in [3] and memory as described below.

The B-P profiler performs following three main functions:

(1) **Decompose the model into its constituent layers:** By default, the profiler resolves the model to its atomic layers, not composed of any other layers. Usually they are inbuilt modules like Linear, Conv2d etc, optimally implemented in C++ to run as a single kernel. However it is often beneficial to stop at some higher granularity. For instance in Inception models, it is common to define the (Conv2d)→(Batch Norm)→(Relu) combination as a module, since it appears repeatedly. Users can provide a list of names of such modules they defined in their model and the profiler will treat them as a single node. This serves the function of co-location constraints (thoughPyTorch does not necessitate it) and of reducing the number of nodes (Section **??**).
(2) **Construct the graph:** Profiler then runs forward and backward operations on the model. PyTorch creates the computation graph during the backward run. This however also includes operation not corresponding to any layer (e.g., scaling an output by 2 or autograd specific operations like `SelectBackward` etc). Such nodes are appropriately pruned or shunted to obtain the dependency graph between layers of the model.
(3) **Run-time and memory requirements:** Finally the profiler measures and annotates the layers with the average forward run-times over multiple runs (ignoring the first few). Memory footprint is also estimated as described below

Memory requirement estimation: In PyTorch, GPU Memory required to hold all the tensors is reserved once during the first training step and reused in subsequent steps. We categorize each layer's memory requirements as follows:

(a) Persistent memory: Parameters of the layer
(b) Output Memory: Intermediate output of the layer
(c) Parameter gradients (= Persistent memory)
(d) Upstream gradient (= Output Memory)
(e) Memory temporarily used in computing the output/gradient

In inference (forward only runs), outputs of intermediate layers are immediately consumed. In training, memory is allocated to store output of each layer during the first forward run. During first back propagation these intermediate outputs are combined with incoming gradient to compute gradients of parameters of the layer. The output is then immediately released and memory is acquired to store the gradients and held until the end. Subsequent steps follow the same pattern. Peak memory consumption is observed at the

[1]Note: TF version 2 supports eager execution

end of forward runs from second step onwards, when all of a, b, c are occupied. Accordingly, for each layer, we have the following requirements:

Memory	Inference	Training
Permanent	(a)	(a) + (c)+(b)
Temporary	(b)+(e)	(e)+(d)

Some exceptions to this rule are appropriately handled. For example, for operations like the in-place ReLU [2], (b) is set to 0 since no new output tensor is generated (a and c are anyways 0 since ReLU has no associated parameters)

Cross-device communication times is calibrated like in Baechi TF and edge weights are added to the graph **??**. This graph is then input into the Baechi Solver along with the number of GPUs and the memory available in each of them . The solver returns the graph where each layer is annotated with the GPU allotted to it and the topological order in which the layer must be executed.

1.3.2 Assigner: The assigner is responsible for automating model parallelism according to the generated placement. Firstly, the assigner moves layer parameters to the respective assigned devices. Further, it modifies the model to optimally run in this distributed setting. The two pre-requisites to achieve this are 1. reordering the layer executions to match the topological order given by Baechi's placement (call it Baechi-topo-order) and 2. an efficient cross-device multi-stream communication mechanism, which is discussed in the next section.

Need for Reordering layers

In a forward run, the host thread eagerly executes the statements in the model. It consists of `forward()` calls of the constituent layers and normal Python statements. `forward()` calls are made in the order in which they appear in the model's code. The `forward()` s submit kernels to the corresponding layers' GPUs. A GPU then asynchronously executes the kernels in the order in which they were added to the GPU's stream. Execution will thus follow the topological order in which the code is written. The layer kernels may not be executed according to the topological order Baechi followed in generating the placement. Thus we need to reorder the layer executions.

A straightforward approach is to use the computation graph we generated in the profiler and execute the layers in the required order ourselves rather than using the given model. This is done in past works such as [2], [3]. This overhaul however will require heavy code rewriting. Because any non- `forward` () python statement (eg: prints, loops etc) in the original model will not be executed, since the computation graph will be unaware of them. Hence the developer must be instructed to write the model in a specific way. Alternatively, one could parse the model code and add such python statements as nodes in the graph. Building such a general parser would be a significant undertaking.[3].

Instead we achieve low-touch reordering by having the host code spawn a thread to handle each layer's `forward()` as it encounters them and move ahead. Meanwhile, a wrapper around a layer's `forward()` will make it wait until the thread of its parent in the Baechi-topo-order submits the kernel and only then submit its own kernel. This way, the host CPU still executes all of the statements and in the same order as the given code, without worrying about the required topological order. But the layer kernels will be queued in the required order on the devices. We can achieve reordering by only building this wrapper around the `forward()` functions of the layers. Thus developers need not make any changes to the model code to execute according to Baechi's placement. In following section when we refer to `forward()` , we imply such a wrapped function.

Reordering Mechanism

Naively, all layers' `forward()` s can be spawned as threads. But as the number of nodes in the graph grows, large number of threads will be simultaneously launched. Overheads of context switching as they resolve the order among themselves will cause delays in kernel submissions and the GPUs will be idle[4]. Our objective thus is to minimize the number of threads that are spawned during reordering.

Then the key question when the host thread reaches a layer in the code is, whether the layer must be made to wait as a thread or if its `forward()` can be called from the host thread itself. Intuitively, if the model's code is written in the exact same order as Baechi-topo-order, then no thread needs to be spawned. On the other hand, a layer waiting in the host thread for its topo parent layer which is further down in the code will result in a deadlock. To illustrate, Fig. 2.a shows a simple model graph to be run on a single GPU. The given model's code orders the nodes from A through F and Baechi-topo-order is given in blue. Host submits layers A and B, but C can not be called from the host because C will wait for its topo parent E while the host, stalled by C, never reaches E.

So a simple criterion is: to spawn layer as a thread if its topo-parent layer's kernel has not been submitted yet. We will refer to this as the **global-topo based criterion**. Fig. 2.b shows the execution trace. At C, the host sees that C's topo-parent E has not been submitted yet. So C is spawned as a thread.

[2]ReLU can also be out-of-place

[3]TorchScript [5] may be of some help. But, currently, it has low support for multi-GPU codes

[4]One could use multiprocessing instead. But, besides the difficulties in using multiprocessing with torch [6], it will likely choke too when the graph grows very large.

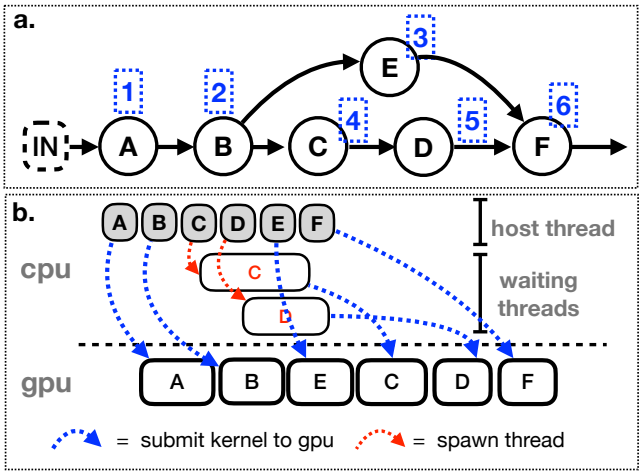
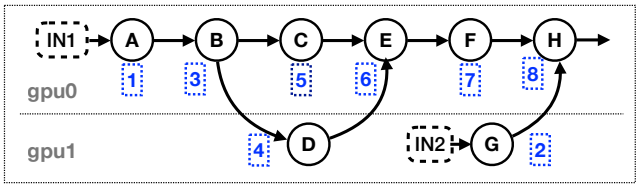


Figure 2: Reordering illustrative example & execution

Similarly, at D, its parent C has been seen by the host but C is still waiting to submit. Hence, D is also spawned as a thread. E can be called in the host because its topo-parent B has already been submitted. As soon as E is submitted, the waiting threads C and then subsequently D can submit. To decide on F, the host checks if F's topo-parent D has submitted. But this result depends on the relative timings of the host checking and D's thread submitting. Irrespective of this, note that it is safe for F to be called in the host itself. F waits until D submits and then submit its kernel. This does not cause a deadlock because the host has reached all of F's precedents (A to E) and they have either been submitted or will eventually do so. Thus a better criterion, which does not depend on relative thread execution times, is as follows: *a layer can be called from the host if* ***all*** *of its topo-precedents have already been reached.* Else it must be made to wait in a thread. This criterion has the added benefit that the host thread need not know if the threads it had spawned have submitted. Keeping a record of what layers it has reached so far suffices. Also note that any such reordering mechanism will increase the net execution time of the host thread. But the GPU kernel executions running in parallel to the host take longer and mask this increase as long as it is not excessive.

While the above criterion is better, it can still cause long waits and GPU idle time. Fig. 3 shows an example graph split across two devices. Here, layer G with a small Baechi-topo-order appears later in the code. As a result, our criterion will make all layers B to F wait as threads. This happens in Transformers for instance, where a typical code calls the encoder and then the decoder. Baechi can generate a placement for it where the initial few nodes within the decoder have lower topo-order so that they can be executed in parallel with the encoder.



Order in the Code	A	B	C	D	E	F	G	H
Required Topo Order	1	3	5	4	6	7	2	8
Topo order on gpu0	1	2	3		4	5		6
Topo order on gpu1				2			1	
Global Topo based	H	T	T	T	T	T	H	H
GPU Topo based →	H	H	H	T	T	T	H	H

H = Host / T = Thread

Figure 3: GPU-wise topo-order based reordering.

The key observation is that only the topological ordering within a device matters. In Fig. 3, it is fine if C's kernel is submitted before node D inspite of having a higher Baechi-topo-order, since they are submitted to two different devices. Thus layers' `forward()` s need to only wait for their GPU-wise topo-parents to submit their kernels (refer to Fig. 3's table). We change the `forward()` wrapper accordingly. The criterion must be changed to: *a layer can be called from the host if all of its* ***GPU-wise*** *topo-precedents have already been reached.* But this has two problems:

- Problem 1: Making `forward()` s wait for the GPU-wise topo-parents only may result in a non-topological ordering and violate graph dependencies. E.g., in Fig. 3, node E's kernel may be submitted on *gpu0* even as E's graph parent D is waiting in *gpu1* for its GPU-wise topo-parent G to be submitted.
- Problem 2: In the new criterion, checking if the host has reached only the GPU-wise topo-precedents can cause a deadlock. At F, its GPU-wise precedents A to E have been reached by the host. Based on that if F's `forward()` is called on the host itself, it will stall the host waiting for its GPU-wise topo-parent E to submit. E never submits because it is waiting for its graph parent D (assuming Problem 1 has been fixed). But D itself is waiting for G, which will never be reached if F stalls the host.

To address these we devise the check `isNotWaiting(layer)`. It is True if, when checked, the `layer` is not required to wait for any other layer and can submit its kernel to its device.

`isNotWaiting(layer)` = True if:

(1) All GPU-wise topo-precedents of `layer` have been reached by the host and

(2) `isNotWaiting(gpuwise_topo_parent_layer)` = True and

(3) `isNotWaiting(graph_parent_layer)` = True for all parents of the `layer` in the graph

(3) and (2) take care of the Problems 1 and 2 respectively. Also, we still need to check (1), because a GPU-wise topo parent of the layer may have `isNotWaiting()` = True i.e it is ready to be submitted, but in the code, it might come after the current layer. Thus we need the topo parent to be reached and not waiting.

The final criterion hence is: a `layer` can be called from the host if `isNotWaiting(layer)` = True. We refer to this as the **GPU-topo based criterion**. As a result only 3 layers (D,E,F) need to be spawned as threads in Fig. 3. `isNotWaiting()` is a recursive check. So it is a concern that it may take long as graph depth increases since `isNotWaiting(layer)` traces back the entire lineage of the `layer`. But once a layers's `isNotWaiting()` turns true, it remains true. We can simply cache the `isNotWaiting()` status of the layers and look them up in checks at subsequent layers. Also note, this criterion retains the benefit of not requiring the host thread to know if past threads have submitted to decide on the current layer.

While one can always create artificial graph structures and device placements on which this protocol will lead to an explosion of number of threads, for practical graphs and Baechi-generated placements for them, this works well. For instance, in a Transformer model with 35 nodes placed across 2 devices using the global-topo based criterion launches 13 nodes as threads, while GPU-topo based criterion launches 0 threads. It is an approximate solution that achieves faithful and automatic execution of the generated placement plan with the minimal engineering overhead.

1.3.3 Communication:

We must now automatically insert streams for communication and synchronization with compute streams on the same device and with communication streams across devices. B-P must handle any arbitrary combination of compute and communication events. Our approach is similar to Rendezvous nodes inTensorflow [7]. But we must do so by only wrapping the `forward()` functions of layers in the Assigner so as to not require model code changes.

The host thread, executing just like on a single device, will pass the output of one layer as input to the next. The child layer however might be on a different device. The parent layer must thus push out its output to the devices of its child layers, and child layers must pull the copy of the passed input on their devices before starting the compute kernels. Also, we must make the compute streams wait for incoming streams, and the outgoing streams for the compute stream.

The communication protocol thus is as follows (refer to Algorithm 1): for every layer we create two streams for every device any of its child layers reside in - a `tx` stream on the layer's GPU and a `rx` stream on child layer's GPU. Only one such stream pair is created for a child device. Each device has one compute stream. When a layer's `forward()` function is launched it waits for its GPU-wise topological parent and all the parent nodes in the graph to be launched(lines 1-3). Then it starts submitting kernels to its GPU. Firstly, the compute stream is made to wait for all `rx` streams to its GPU from all the parent layers (line 4). Once done we know that a copy of the inputs resides on the current GPU. These copies are passed to the actual forward function of the layer (lines 7-8). All the `tx` streams going out of this layer are made to wait for the compute stream to finish the computation (lines 9-11). As soon as the output is ready, it is asynchronously sent to all the child devices through the `tx` streams (lines 12-15). Finally the reference to output is passed back to host code (line16). When GPU actually executes the kernels, the compute stream can move to the next layer assigned to it while `tx` streams are transferring out the output. Thus computation and communication overlap. In some cases, a reference to the output must be maintained until all `tx` streams complete, in order to avoid subtle errors described in Section 1.2.3. Also, the output is sent to a child device only once even if multiple children reside on that device. Note that the `tx` - `rx` streams serve as synchronization points in addition to overlapping communication and computation.

Algorithm 1: The wrapper around `forward()`

```
1  wait for submission of GPU-wise topo parent;
2  for each parent in graph do
3      wait for submission of parent;
4      (compute_stream) wait for (rx_stream from
         parent);
5  end
6  On compute_stream:
7      input_copies = local copies of input;
8      output = forward (input_copies);
9  for each child in graph do
10     (tx_stream to child) wait for (compute_stream);
11 end
12 for each child in graph do
13     On tx_stream to child:
14         send output to child;
15 end
16 return output
```

This protocol makes the actual runtime communication similar to what is assumed by the graph solver. One exception however is that reception at a GPU is always serialized. So, *gpu1→gpu0* and *gpu2→gpu0* simultaneous transfers are serialized, even though they are on two different streams. However, this rarely happens, and the associated effect on step time is small. Similarly, overlapping outgoing tx streams from a device may increase communication times, but the impact is small.

In summary, the protocol sends out the tensors as soon as they are available and synchronizes the independent streams involved.

1.3.4 Code Modifications required: The design goal of B-P is offering seamless model parallelism with minimal code changes. All of the functionalities we discussed so far did not require the developer to make any code modifications. This was mainly possible because we integrated them as wrappers around the `forward()` s. However, this means some operations like concatenate and add, which take multiple inputs, must be defined as PyTorch layers. Only then can the Assigner ensure that tensors being concatenated or added are on the same device. In most cases, this is a few lines of code change as shown in Fig. 4(1). For instance, in Inceptionv3, concatenate is used 7 times and add is used only once.

Besides this, only three lines are required to convert the single device Pytorch model into a multi-device distributed model. It is shown in Fig. 4(2).

1.4 Limitations

We now list some limitations of the current version of B-P:

(1) The profiler does not ensure code coverage. Currently, random dummy data is used to profile and conditional paths, if any, may not be included in the graph. Also, the placer is not designed to handle conditional branches in the computation graph.
(2) Reordering of layers by the Assigner is akin to a compiler optimization albeit at a much higher level. However we violate the as-if rule [8] and side-effects of the optimization is observable if probed. Because the layers are not actually executed in the code order, the output of a layer that has not launched yet will be `None` (by our design). This can sometimes make debugging inconvenient but is unlikely to result in stealthy bugs. In any case, we allow the user to switch to non-reordered but still distributed model by setting a flag. This way the user can debug the model without having to manually split the model or worry about OOMs. Once done, with a single flag switch, the model can be left to train in optimal time with the reordering enabled

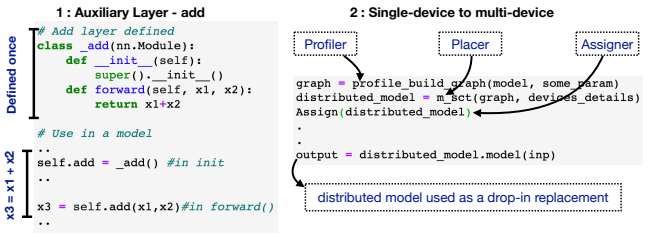


Figure 4: Final Code

(3) Weight-sharing is common is Transformers, which is two different layers sharing the same weights. We do not support this yet.
(4) Forward hooks in PyTorch allow executing a piece of code just before executing a layer. These cannot be used with reordering.

References

[1] author. 2019. title. In *1book*. ACM, Article 26, 17 pages.
[2] author. 2019. title. In *1book*. ACM, Article 26, 17 pages.
[3] author. 2019. title. In *1book*. ACM, Article 26, 17 pages.
[4] author. 2019. title. In *1book*. ACM, Article 26, 17 pages.
[5] author. 2019. title. In *1book*. ACM, Article 26, 17 pages.
[6] author. 2019. title. In *1book*. ACM, Article 26, 17 pages.
[7] author. 2019. title. In *1book*. ACM, Article 26, 17 pages.
[8] author. 2019. title. In *1book*. ACM, Article 26, 17 pages.