Varuna

Release

Jul 17, 2021

CONTENTS:

Detailed documentation for Varuna functions and classes can be found and navigated here.

CHAPTER

ONE

LAUNCHING VARUNA

Varuna distributed training is run as a set of processes for each GPU on each machine. It uses PyTorch's distributed framework and must be used with the gloo backend. This distributed process is triggered from a single machine with a list of reachable machines (IPs) as *machine_list* and *gpus_per_node* GPUs on each node. This triggering machine is usually the 'manager' (as explained in *Morphing*). Morphing is enabled by default, to disable it use the *no_morphing* flag. Training with varuna can be run with the *run_varuna* module as follows:

```
python -m varuna.run_varuna --machine_list <file_with_ips> --gpus_per_node <num_gpus_
↪per_node>
--batch-size <total_effective_batch_size> --nstages <number_of_pipeline_stages>
--chunk_size <micro_batch_size_for_pipeline>
--code_dir <working_dir_for_training> user_training_script.py <...user args...>
```

This expects all machines in the *machine_list* to be reachable and to be set up with necessary code/libraries in *code_dir*. The user's code should also be modified to add *CutPoint's and use the 'Varuna* training class. The job is launched with all workers (*gpus_per_node x <num-servers>* in total) running the *user_training_script* with user args and arguments passed by varuna's launcher. Any environment variables that the user wishes to pass to each worker may be specified in an *env_file* passed to the launcher.

These arguments passed by the launcher to the user training script for Varuna must be parsed by user's training script and passed during *Varuna* initialisation:

- rank: process rank in overall distributed job
- local_rank: process rank in the local node
- stage_to_rank_map: varuna config info about stage placement
- chunk_size: micro batch size for Varuna pipeline
- batch-size: per process batch size

The arguments for number of pipeline stages *nstages* and micro-batch size *chunk_size* can be omitted if the user wishes Varuna to determine the most optimal configuration for these. This requires the user to run profiling before training and pass the location of stored profiles to the launcher. (see *Profiling for Varuna*)

CHAPTER

TWO

CUTPOINTS

Varuna slices a DNN model into sequential stages for pipeline parallelism. For this, the model should be annotated with varuna `CutPoint` instances between different operations/ parts of model computation.

A `CutPoint` in varuna is an abstraction to mark a potential point of partitioning in your model. It is implemented as a `torch.nn.Module` instance, which is called on the activations at the potential boundary point. For each `CutPoint`, Varuna can either ignore it or activate it as a partition boundary. CutPoints can be marked anywhere in the model as follows:

```
from varuna import CutPoint

class SampleModel(nn.Module):
def __init__(...):
    ....
    self.cutpoints = [CutPoint() for i in range(num_cutpoints)]
    ....

def forward(input...):
    input = self.some_operation(input)
    input = self.cutpoints[0](input)      # marked as a potential stage boundary
    input = self.some_other_operation(input)
    ....
    for i in range(sub_modules):
    x = sub_module_i(input, ...)
    x = self.cutpoints[i+1](x)         # each cutpoint instance should be used only␣
↪once in a model
    ....
```

Based on the number of desired pipeline stages, Varuna chooses a subset of the given cutpoints and activates them as actual boundaries between stages. For example, if the user marks *n* cutpoints in total, and wants 4 parallel pipeline stages, 3 cutpoints will be activated as partitions between the 4 stages and the rest *n-3* are treated as they don't exist. With this partitioning, each worker in the distributed job runs a sub-section of the model code between two activated `CutPoint` instances, or between one activated `CutPoint` and the beginning/end of the model.

For an activated `CutPoint`, the input to the cutpoint is an intermediate activation in the model that needs to be passed between sequential stages.

Note: The input to any `CutPoint` in the model's execution should be a single `torch.Tensor` of shape *(b, d2, d3, . . .)* where *b* is the number of examples in the input to the model. This is important because Varuna uses micro-batches to parallelize computation and relies on this format for communication between pipeline stages.

Operations separated by CutPoints should preferably have no shared modules/parameters. For weight sharing between different parts of the module, you should register separate `nn.Parameter` instances (even for the same tensor) and pass the pair of parameter names as `shared_weights` to the `Varuna` object.

For example, in language models like BERT and GPT2, the weights for word embedding computation at the beginning of the model are also utilised at the end of the model for prediction logits. So, if this weight is wrapped in two separate `torch.nn.Parameter` instances, they will have two corresponding "parameter names" (string values) in the model (see `named_parameters()` for `torch.nn.Parameter`). These can be passed as a pair of names for each shared weight to `Varuna` as follows:

```
# list of 2-tuples with parameter names
shared_weights = [("language_model.embedding.word_embeddings.weight","lm_head_weight
↪")]
model = Varuna( model, args.stage_to_rank_map, dry_run_input, global_batch_size,
                    args.chunk_size, args.fp16,
                    local_rank=args.local_rank,
                    device=args.local_rank,
                    shared_weights=shared_weights)  # passed to varuna init
```

CHAPTER

THREE

THE VARUNA CLASS

The torch.nn.Module object for your DNN model should be wrapped in a `Varuna` instance for training. This class extends torch.nn.Module and handles distributed pipeline & data parallelism, mixed precision and shared parameter weights internally.

Wrapping in `Varuna` partitions the model into pipeline stages across the distributed job. For this, it uses stage allocation information that is passed by `varuna.launcher` to all worker processes. The launcher uses a string argument `stage_to_rank_map` which must be parsed and used for `Varuna` initialisation.

Sample inputs (with any batch size) also need to be passed for this automatic partitioning. These inputs are used to profile the model's computation graph and should be passed a a dictionary of keywords to args.

The model passed to `Varuna` should be on CPU. Once the profiling and partitioning are done, the model is moved to the assigned GPU. So the user need not do `model.cuda()` anywhere.

Optimizer creation should be after wrapping in `Varuna`, since it requires model parameters as input. The optimizer needs to be registered with Varuna using a setter.

Example:

```
model = MyModel()              # full model on CPU
dry_run_input = {              # sample inputs for MyModel
   'inputs': inputs,           # keyword: arg
   'mask': mask,
   'extra_norm': True
}
# parameter sharing across the model, marked as pairs of param_names
shared_weights = [("language_model.embedding.word_embeddings.weight","lm_head_weight
↪")]
model = Varuna( model, args.stage_to_rank_map, dry_run_input, global_batch_size,
                   args.chunk_size, args.fp16, local_rank=args.local_rank,
                    device=args.local_rank, shared_weights=shared_weights)

# now model is a subset of the original model, moved to the GPU on each process

optimizer = get_optimizer(model)
model.set_optimizer(optimizer)

.. autoclass:: Varuna

     .. automethod:: set_optimizer
     .. automethod:: step
     .. automethod:: checkpoint
     .. automethod:: load_checkpoint
     .. automethod:: evaluate
```

CHAPTER

FOUR

PROFILING FOR VARUNA

The Varuna `Profiler` provides an easy interface for users to profile the compute and communication operations of a model. This processes the model cutpoints in parallel and captures the time and memory consumption for each cutpoint. This profile can then be used to calculate various parameters for Varuna - ideal pipeline and data-parallel dimensions for a given number of GPUs and suitable microbatch sizes for different configs.

class varuna.**Profiler**(*model*, *get_batch*, *device=-1*, *gpus_per_node=None*, *fp16=False*, *out_folder='profiles'*, *pstages_to_profile=None*, *from_cache=True*, *add_to_existing=False*)

Module for varuna profiling. Similar to `Varuna` class, the model must be wrapped in an instance of `Profiler` before optimizer creation and the `model` passed should be on CPU.

Varuna profiling runs in a distributed process and the `Profiler` should be used by each worker. Each worker profiles compute for the different ``CutPoint``s in the model while simultaneously measuring communication links between workers. The profiler should be used in three steps:

```
def get_batch(size):
    # function to get sample batches of given size for profiling
    return batch
profiler = Profiler(model, get_batch_fn, fp16=args.fp16, device = args.local_rank,
                    from_cache=True, out_folder=args.save)
profile = profiler.profile_all(microbatch_sizes_to_profile)
```

Parameters

- **`model`** (*`torch.nn.Module`*) – The model to profile.
- **`get_batch_fn`** (*`function(size, device='cpu')`*) – Function to get batch of a given size, used for different sizes by the profiler
- **`device`** (*`int`*) – index of the cuda device to use. Recommended to be the same as local_rank, which is the default if not specified.
- **`fp16`** (*`bool`*) – whether to enable mixed precision training.
- **`from_cache`** (*`bool`*) – Whether to use cached information anout model structure, if available.
- **`out_folder`** (*`string or PathLike object`*) – Path to folder for saving compute and communication profiles
- **`pstages_to_profile`** – List of indices of cutpoints to profile, by default this contains all cutpoints

:type list or None :param add_to_existing: Whether to continue profiling by adding to cutpoint profiles already saved in out_folder :type add_To_existing: bool

set_optimizer(*optimizer*, *amp_opt_level='O2'*, *loss_scale='dynamic'*, *init_loss_scale=1048576*, *min_loss_scale=None*)

profile_all(*microbatch_sizes*)

CHAPTER

FIVE

MORPHING

Varuna enables distributed training on a changing set of resources, as the list of machines available may grow or shrink. This is done by "morphing" - reconfiguring the training job to process the total effective batch size over the new resources. Varuna performs morphing by checkpointing and restarting efficiently, which requires that the training job has access to a long-living 'manager' machine and a global storage for all workers.

The manager launches the *run_varuna* command, detects changes in the available resource set, slow GPUs or transient errors in the job, and cooridinates checkpoint/restarts. If desirable, the manager can be notified of an upcoming preemption (loss of a machine) through the function *notify_manager_preempt*. For example in Azure, a 'preempt' signal is issued with preemption time.

To enable morphing, the user must make some modifications to their script:

- **An additional *resume_step* argument is passed to each worker for restarts. (So that there** are no race conditions while checking this step from the global storage)
- **A simple signal handler for *SIGUSR1* in the workers to call varuna's *on_demand_checkpoint* (*The Varuna class* class)** and exit. The checkpointing may fail if workers are lost during the call.
- **(recommended) With morphing, *Varuna* checkpointing should be enabled with background copying and sharding flags fo** faster checkpointing. The checkpoint frequency should be high to avoid loss of compute on checkpoint/restarts (in case on demand checkpoints fail).

These changes are illustrated in the megatron example.

The key idea behind morphing is to re-distribute the total *batch_size* specified by the user accross pipeline parallel stages and data parallel replicas. To do this efficiently, it is recommended to use auto-configuration of the dimensions of pipeline and data parallelism as well as the micro-batch size. *AutoConfig* by varuna is enabled if these arguments (*nstages* and *chunk_size*) are not specified while launching *run_varuna*. This estimates the best varuna configuration at each point and requires the user to run profiling before training and specify the location of stored profiles to the launcher. (see *Profiling for Varuna*)

5.1 Slow GPU detection

With low-priority VMs, a user might see faulty "straggler" GPUs that have significantly longer compute times than the others. These are detected by varuna when morphing is enabled by the manager. The IPs with the slow GPUs are written to a file "slow_machines.out". The user may listen on this file to remove machines with faulty GPUs.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

INDEX