Varuna

Nitika Saran

Jul 16, 2021

CONTENTS:

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

CHAPTER

ONE

LAUNCHING VARUNA

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 from the launching machine, to be set up with necessary code/libraries in code_dir and have gpus_per_node GPUs working. The job is launched with all workers running the user_training_script and args.

This launcher passes a few arguments to the user training script for Varuna. These should be passed during *Varuna* initialisation in the python script: * 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. (see *Launching Varuna*)

For profiling and automatic partitioning, Varuna needs sample inputs. For this, a `get_batch_fn` needs to be passed during initialisation which returns a sample input batch of a given size. This is used to profile the model's computation graph and should return a dictionary of keywords to args, similar to the `step` function.

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
def get_batch_fn(size, device=None):
   batch = dataset[:size]
   if device is not None:
      batch = [t.to(device) for t in batch]
   inputs, mask = batch
   return {'inputs': inputs, '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, get_batch_fn, 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)
```

class varuna.**Varuna**(*model*, *stage_to_rank_map*, *get_batch_fn*, *batch_size*, *chunk_size*, *fp16=False*, *local_rank=- 1*, *device=- 1*, *shared_weights=None*, *from_cache=True*)

Module to implement varuna training. The model must be wrapped in an instance of `Varuna` before training. This should be done before optimizer creation and the `model` passed should be on CPU.

Creating a `Varuna` instance profiles the model briefly using `dummy_inputs` and partitions it according to the distributed rank and launcher arguments. The partitioned model is then moved to the allocated cuda device. The

profiling information is cached and can be re-used on resuming, unless `from_cache` is False. The `Varuna` module performs mixed precision training internally if enabled through the `fp16` arg, no external handling is required.

Parameters

- **model** (*torch.nn.Module*) – The model to initialize for training.
- **stage_to_rank_map** (*dict*) – Placement of pipeline stages in the distribued job, encoded as a string. Passed by `varuna.launcher` to each worker as an argument.
- **get_batch_fn** (*function(size: int, device: torch.device or None)*) – Function to get sample input batches of a given size, as dictionaries. These are used to profile the model structure as `model(**get_batch_fn(k, device='cpu))`.
- **batch_size** (*int*) – Global batch size for the distributed training job.
- **chunk_size** (*int*) – The micro-batch size to be used for pipeline parallelism.
- **fp16** (*bool*) – whether to enable mixed precision training.
- **local_rank** (*int*) – The local rank as passed by `varuna.launcher`. If not given, defaults to the global rank.
- **device** (*int*) – index of the cuda device to use. Recommended to be the same as local_rank, which is the default if not specified.
- **shared_weights** (*list or None*) – A list of tuples, where each each tuple is a pair of weight names (strings), such that the two weights are shared in the model (see weight sharing)
- **from_cache** (*bool*) – Whether to use cached profiling information if available.

Note: Optimizer initiliastion should be done after `Varuna` initialisation, so that the `param_group` s for the optimizer only contain parameters from the partitioned model. This is important both for memory usage and correctness of fp16 training. Once `Varuna` and the optimizer are initialised, *set_optimizer()* should be called to connect the two.

---

set_optimizer(*optimizer*, *loss_scale='dynamic'*, *init_loss_scale=1048576*, *min_loss_scale=1.0*)

Configure optimizer for training. if `fp16` is enabled, this function initializes the mixed precision state in apex.

Parameters

- **optimizer** (*torch.nn.Optimizer*) – the optimizer for training.
- **loss_scale** (*float or "dynamic", optional*) – A floating point number for a static loss scale or the string "dynamic" for dynamic loss scaling.
- **init_loss_scale** (*float, optional*) – Initial loss scale (for dynamic scaling)
- **min_loss_scale** (*float, optional*) – minimum loss scale (for dynamic scaling)

step(*inputs*, *clip_grad_max_norm=None*)

Perform a single training step. Executes forward and backward passes for the global batch. This function must be called by all distributed workers in the training loop. After this function, the optimizer gradients are reduced accross data parallel replicas and overflow is checked for mixed precision training. Returns average loss and a boolean for overflow.

Parameters

- **inputs** (*dict*) – The inputs to the model as a dictionary. These should be coordinated amongst workers - the global batch is sharded across data parallel replicas, so each worker should have `global_batch_size / data_parallel_depth` number of examples. And all pipeline stages of the same data parallel replica should recieve the same inputs.
- **clip_grad_max_norm** (*float or None, optional*) – If given, the L2 gradient norm of the entire model is clipped to this upper bound.

Returns A tuple of the form (average_loss, overflow)

Return type tuple[float, bool]

checkpoint(*global_store*, *step=None*, *tempdir=None*, *shard=False*, *on_demand=False*)

Writes a varuna checkpoint with model parameters, optimizer state etc. Each checkpoint is a directory, written under the given path.

Parameters

- **global_store** (*dict*) – path to a folder accessible by all nodes/ranks in the training job. For example, path to a mounted blob storage. This is where the varuna checkpoint folder is written.
- **step** (*int or None, optional*) – iteration number for checkpoint. If None, it'll be taken from varuna's tracked progress.
- **tempdir** (*str, optional*) – path to a local directory to which to write checkpoints temporarily, and sync with the global store in the background. Lowers checkpoint write time in the critical path.
- **shard** (*bool, optional*) – whether to shard checkpoint writes over data parallel workers as well. Speeds up checkpoint

load_checkpoint(*global_store*, *iteration*, *check_complete=True*)

Loads a varuna checkpoint from a shared directory. Each varuna checkpoint is a directory named as "varuna_ckpt_<iteration>". So the path under which all such checkpoints were written should be specified.

Parameters

- **global_store** (*str*) – path under which varuna checkpoints were written. Should be accessible by all workers.
- **iteration** (*int*) – Which iteration checkpoint to load.
- **check_complete** (*bool, optional*) – Check that the checkpoint is complete before loading it. A checkpoint can be incomplete if the write was interrupted.

evaluate(*inputs*, *batch_size=None*)

Evaluate the model on the given inputs. This must be called on all workers because it uses pipeline & data parallelism. Inputs should be for the respective data parallel replica and have `batch_size/data_parallel_depth` examples, similar to *step()*. Returns loss averaged over all workers.

Parameters

- **inputs** (*dict*) – Model inputs as dictionary. The number of examples for these inputs should be the same as the batch_size defined for training.
- **batch_size** (*int, optional*) – Batch size for evaluation, if not given it's the same as training batch size.

Returns average loss

Return type float

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

INDICES AND TABLES

- genindex
- modindex
- search

INDEX