GPUParallel

Release 0.2.2

Vladimir Ivashkin

Jul 06, 2023

CONTENTS

Release v0.2.2. (*Installation*)

Joblib-like interface for parallel GPU computations (e.g. data preprocessing):

```
import torch
from gpuparallel import GPUParallel, delayed

def perform(idx, gpu_id, **kwargs):
    tensor = torch.Tensor([idx]).to(gpu_id)
    return (tensor * tensor).item()

result = GPUParallel(n_gpu=2)(delayed(perform)(idx) for idx in range(5))
print(sorted(result))  # [0.0, 1.0, 4.0, 9.0, 16.0]
```

CHAPTER

ONE

FEATURES

- *Initialize networks on worker init*
- *Reuse initialized workers*
- *Simple logging from workers*
- Sync mode for tasks debug (use `n_gpu = 0`)
- Progressbar with tqdm: `progressbar=True`
- Optional ignoring task errors: `ignore_errors=True`

See *Quickstart* and *API Reference* for details.

CHAPTER

TWO

USER GUIDE

2.1 Installation of GPUParallel

To install GPUParallel, simply run this simple command in your terminal of choice:

```
$ python -m pip install gpuparallel
```

If you want to use unstable version, you can install it from sources. Clone the repo and install:

```
$ git clone git://github.com/vlivashkin/gpuparallel.git
$ cd gpuparallel
$ python3 -m pip install .
```

Or, as a shortcut:

```
$ python3 -m pip install git+git://github.com/vlivashkin/gpuparallel.git
```

2.2 Quickstart

2.2.1 Basic usage

Calc squares of numbers:

```
import torch
from gpuparallel import GPUParallel, delayed

def perform(idx, gpu_id, **kwargs):
    tensor = torch.Tensor([idx]).to(gpu_id)
    return (tensor * tensor).item()

result = GPUParallel(n_gpu=2)(delayed(perform)(idx) for idx in range(5))
print(sorted(result))  # [0.0, 1.0, 4.0, 9.0, 16.0]
```

2.2.2 Initialize networks on worker init

Function `init_fn` is called on init of every worker. All common resources (e.g. networks) can be initialized here. In this example we create 32 workers on 16 GPUs, init `model` when workers are starting and then reuse workers for several batches of tasks:

```
from gpuparallel import GPUParallel, delayed

def init(gpu_id=None, **kwargs):
    global model
    model = load_model().to(gpu_id)

def perform(img, gpu_id=None, **kwargs):
    global model
    return model(img.to(gpu_id))

gp = GPUParallel(n_gpu=16, n_workers_per_gpu=2, init_fn=init)
results = gp(delayed(perform)(img) for img in fnames)
```

2.2.3 Reuse initialized workers

Once workers are initialized, they keep live until `GPUParallel` object exist. You can perform several queues of tasks without reinitializing worker resources:

```
gp = GPUParallel(n_gpu=16, n_workers_per_gpu=2, init_fn=init)
overall_results = []
for folder_images in folders:
    folder_results = gp(delayed(perform)(img) for img in folder_images)
    overall_results.extend(folder_results)
del gp  # this will close process pool to free memory
```

2.2.4 Simple logging from workers

Use `log_to_stderr()` call to init logging, and `log.info(message)` to log info from workers:

```
from gpuparallel import GPUParallel, delayed, log_to_stderr, log

log_to_stderr()

def perform(idx, worker_id=None, gpu_id=None):
    hi = f'Hello world #{idx} from worker #{worker_id} with GPU#{gpu_id}!'
    log.info(hi)

GPUParallel(n_gpu=2)(delayed(perform)(idx) for idx in range(2))
```

It will return:

```
[INFO/Worker-1(GPU1)]:Hello world #1 from worker #1 with GPU#1!
[INFO/Worker-0(GPU0)]:Hello world #0 from worker #0 with GPU#0!
```

2.3 API Reference

class gpuparallel.**GPUParallel**(*device_ids: Optional[List[str]] = None*, *n_gpu: Optional[Union[int, str]] = None*, *n_workers_per_gpu=1*, *init_fn: Optional[Callable] = None*, *preserve_order=True*, *progressbar=True*, *pbar_description=None*, *ignore_errors=False*, *debug=False*)

Bases: `object`

__init__(*device_ids: Optional[List[str]] = None*, *n_gpu: Optional[Union[int, str]] = None*, *n_workers_per_gpu=1*, *init_fn: Optional[Callable] = None*, *preserve_order=True*, *progressbar=True*, *pbar_description=None*, *ignore_errors=False*, *debug=False*)

Parallel execution of functions passed to `__call__`.

Parameters

- **device_ids** – List of gpu ids to use, e.g. `['cuda:3', 'cuda:4']`. The library doesn't check if GPUs really available, it simply provides consistent `worker_id` and `device_id` to both `init_fn` and task functions.
- **n_gpu** – Number of GPUs to use, shortcut for `device_ids=[f'cuda:{i}' for i in range(n_gpu)]`. Both parameters `n_gpu` and `device_ids` can't be filled. If neither of them filled, single `cuda:0` will be chosen.
- **n_workers_per_gpu** – Number of workers on every GPU.
- **init_fn** – Function which will be called during worker init. Function must have parameters `worker_id` and `device_id` (or `**kwargs`). Helpful to init all common stuff (e.g. neural networks) here.
- **preserve_order** – Return values with the same order as input.
- **progressbar** – Allow to use tqdm progressbar.
- **ignore_errors** – Either ignore errors inside tasks or raise them.
- **debug** – When this parameter is True, parameters n_gpu and device_ids are ignored. Class creates only one worker ([device_id='cuda:0']) and run it in the same process (for better debugging).

__del__()

Created pool will be freed only during this destructor. This allows to use `__call__` multiple times with the same initialized workers.

__call__(*tasks: Iterable[Callable]*) → Generator

Function which submits tasks for pool and collects the results of computations.

Parameters

tasks – List or generator with callable functions to be executed. Functions must have parameters `worker_id` and `device_id` (or `**kwargs`).

Returns

List of results or generator

class gpuparallel.**BatchGPUParallel**(*task_fn: Callable*, *batch_size*, *flat_result=False*, **args*, ***kwargs*)

Bases: *GPUParallel*

__init__(*task_fn: Callable*, *batch_size*, *flat_result=False*, **args*, ***kwargs*)

Parallel execution of `task_fn` with parameters given to `__call__`. Tasks are batched: every arg and kwarg turns into list.

Parameters

- **task_fn** – Task to be executed
- **batch_size** – Batch size
- **flat_result** – Unbatch results. Works only for single tensor output.

__call__(**args*, ***kwargs*) → Generator

All input parameters should have equal first axis to be batched. First arg/kwarg is used to determine size of the dataset. Inputs with other shape (or not Sequence typed) will be copied to every worker without batching. :return: Batched result

gpuparallel.**delayed**(*func*)

Decorator used to capture the arguments of a function. Analogue of joblib's delayed.

Parameters

func – Function to be captured.

gpuparallel.**log_to_stderr**(*log_level='INFO'*, *force=False*)

Shortcut allowing to display logs from workers.

Parameters

- **log_level** – Set the logging level of this logger.
- **force** – Add handler even there are other handlers already.

PYTHON MODULE INDEX

INDEX

Symbols

B

D

G

L

M