
GPUParallel

Release 0.0.4

Vladimir Ivashkin

May 10, 2021

CONTENTS

1	Features	3
2	User Guide	5
2.1	Installation of GPUParallel	5
2.2	Quickstart	5
2.3	API Reference	7
	Python Module Index	9
	Index	11

Release v0.0.4. (*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]
```


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.

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:

```
1 import torch
2 from gpuparallel import GPUParallel, delayed
3
4 def perform(idx, gpu_id, **kwargs):
5     tensor = torch.Tensor([idx]).to(gpu_id)
6     return (tensor * tensor).item()
7
8 result = GPUParallel(n_gpu=2)(delayed(perform)(idx) for idx in range(5))
9 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:

```
1 from gpuparallel import GPUParallel, delayed
2
3 def init(gpu_id=None, **kwargs):
4     global model
5     model = load_model().to(gpu_id)
6
7 def perform(img, gpu_id=None, **kwargs):
8     global model
9     return model(img.to(gpu_id))
10
11 gp = GPUParallel(n_gpu=16, n_workers_per_gpu=2, init_fn=init)
12 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:

```
1 gp = GPUParallel(n_gpu=16, n_workers_per_gpu=2, init_fn=init)
2 overall_results = []
3 for folder_images in folders:
4     folder_results = gp(delayed(perform)(img) for img in folder_images)
5     overall_results.extend(folder_results)
6 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:

```
1 from gpuparallel import GPUParallel, delayed, log_to_stderr, log
2
3 log_to_stderr()
4
5 def perform(idx, worker_id=None, gpu_id=None):
6     hi = f'Hello world #{idx} from worker #{worker_id} with GPU#{gpu_id}!'
7     log.info(hi)
8
9 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(*n_gpu=1, n_workers_per_gpu=1, init_fn: Optional[Callable] = None, progressbar=True, ignore_errors=True*)

Bases: object

__init__(*n_gpu=1, n_workers_per_gpu=1, init_fn: Optional[Callable] = None, progressbar=True, ignore_errors=True*)

Parameters

- **n_gpu** – Number of GPUs to use. The library doesn't check if GPUs really available, it is simply provide consistent `worker_id` and `gpu_id` to both `init_fn` and task functions. `n_gpu = 0` turns on synced debug mode.
- **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 `gpu_id` (or `**kwargs`). Helpful to init all common stuff (e.g. neural networks) here.
- **progressbar** – Allow to use tqdm progressbar.
- **ignore_errors** – Either ignore errors inside tasks or raise them.

__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*) → List

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 `gpu_id` (or `**kwargs`).

Returns List of results

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'*)

Shortcut allowing to display logs from workers.

Parameters **log_level** – Set the logging level of this logger.

PYTHON MODULE INDEX

g

gpuparallel, [7](#)

Symbols

`__call__()` (*gpuparallel.GPUParallel method*), 7
`__del__()` (*gpuparallel.GPUParallel method*), 7
`__init__()` (*gpuparallel.GPUParallel method*), 7

D

`delayed()` (*in module gpuparallel*), 7

G

`gpuparallel`
 module, 7
`GPUParallel` (*class in gpuparallel*), 7

L

`log_to_stderr()` (*in module gpuparallel*), 7

M

`module`
 gpuparallel, 7