RaNNC

Release 0.4.2.post13

DIRECT, National Institute of Information and Communications Te

Jan 21, 2021

CONTENTS:

RaNNC is a deep learning framework for automatic model/data parallelism. RaNNC decomposes a computational graph of a PyTorch model and distributes the subgraphs onto multiple compute nodes. You can train a billion-scale parameter models using RaNNC.

> Automatic Graph Partitioning for Very Large-scale Deep Learning, Masahiro Tanaka, Kenjiro Taura, Toshihiro Hanawa and Kentaro Torisawa, In the Proceedings of 35th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2021), Portland, Oregon USA, May, 2021. (to appear)

CHAPTER

ONE

INSTALLATION

1.1 Prerequisites

RaNNC works only with CUDA devices (CPU only/TPU environments are not supported). RaNNC requires following libraries and tools at runtime.

- *CUDA*: A CUDA runtime must be available at the runtime environment. Currently RaNNC is tested with CUDA 10.2.
- *NCCL*: NCCL (Version >= 2.7.3 is required) must be available at the runtime environment. RaNNC uses NCCL both for allreduce and P2P communications.
- *MPI*: A program using RaNNC must be launched with MPI. MPI libraries must also be available at runtime. RaNNC is tested with OpenMPI v4.0.5.
- *libstd++*: `libstd++` must support `GLIBCXX_3.4.21` to use the distributed `pip` packages (The packages are built with gcc 5.4.0).

1.2 Installation

This version of RaNNC requires PyTorch v1.7.1. `pip` packages for `linux_x86_64` are available from the following links.

- `For Python 3.7`
- `For Python 3.8`

You can create a new conda environment and install RaNNC by the following commands.

```
conda create -n rannc python=3.8
conda activate rannc
conda install pytorch==1.7.1 cudatoolkit=10.2 -c pytorch
pip install pyrannc-0.5-cp38-cp38m-linux_x86_64.whl
```

CHAPTER

TWO

TUTORIAL

RaNNC distributes PyTorch models onto multiple nodes or processes using MPI. Follow the steps below to learn the basic usage of RaNNC.

2.1 Steps to use RaNNC

2.1.1 0. Set up environment

Ensure required tools and libraries (CUDA, NCCL, OpenMPI, etc.) are available. The libraries must be included in `LD_LIBRARY_PATH` at runtime.

2.1.2 1. Import RaNNC

Insert `import` in your script.

```
import pyrannc
```

2.1.3 2. Wrap your model

Wrap your model by `pyrannc.RaNNCModule` with your optimizer. You can use the wrapped model in almost same manner as the original model (See below). Note that the original model must be on a CUDA device.

```
model = Net()
model.to(torch.device("cuda"))
opt = optim.SGD(model.parameters(), lr=0.01)
model = pyrannc.RaNNCModule(model, optimizer)
```

If you don't use an optimizer, pass only the model.

```
model = pyrannc.RaNNCModule(model)
```

2.1.4 3. Run forward/backward passes

A `RaNNCModule` can run forward/backward passes as with a `torch.nn.Module`.

```
x = torch.randn(64, 3, requires_grad=True).to(torch.device("cuda"))
out = model(x)
out.backward(torch.randn_like(out))
```

Inputs to `RaNNCModule` must be CUDA tensors. RaNNCModule has several more limitations about a wrapped model and inputs/outputs. See *Limitations* for details. The optimizer can update model parameters just by calling `step()`.

The program below (`examples/tutorial_usage.py`) shows the above usage with a very simple model.

```
import torch
import torch.nn as nn
import torch.optim as optim

import pyrannc


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(3, 2, bias=False)
        self.fc2 = nn.Linear(2, 3, bias=False)

    def forward(self, x):
        x = self.fc1(x)
        x = self.fc2(x)
        return x


model = Net()
model.to(torch.device("cuda"))
opt = optim.SGD(model.parameters(), lr=0.01)
model = pyrannc.RaNNCModule(model, opt)

x = torch.randn(64, 3, requires_grad=True).to(torch.device("cuda"))
out = model(x)

target = torch.randn_like(out)
out.backward(target)

opt.step()
```

2.1.5 4. Launch

A program using RaNNC requires to be launched by `mpirun`. You can launch the above example script by:

```
mpirun -np 2 python tutorial_usage.py
```

`-np` indicates the number of ranks (processes). RaNNC allocates one CUDA device for each rank. In the above example, there must be two available CUDA devices.

2.2 How RaNNC works

2.2.1 Automatic parallelism

RaNNC analyzes a given model and determines the best combination of different parallelisms. RaNNC can combine the three following parallelisms.

- Data parallelism

- Model parallelism
- Pipeline parallelism

See IPDPS 2021 paper for the details of the automatic parallelism.

2.2.2 Data distribution

Each process launched by MPI is expected to load different (mini-)batches. RaNNC automatically gathers the batches from all ranks and compute them as one batch. `torch.utils.data.distributed.DistributedSampler` will be useful for this purpose.

CHAPTER

THREE

LIMITATIONS

Although a `RaNNCModel` is designed to work like `torch.nn.Module`, it has the following limitations.

3.1 Control constructs are ignored

RaNNC uses a computation graph produced by PyTorch's tracing function. As explained in the document, the tracing function does not record control constructs including conditional branches and loops.

3.2 Arguments and return values

Arguments and outputs of a `RaNNCModel` must satisfy the following conditions.

- Arguments must be (mini-)batches tensors, whose first dimension corresponds to samples in a mini-batch.
- Keyword arguments are not allowed.
- Outputs must be (mini-)batches tensors, or a loss value (scalar tensor).

3.3 Tensor data types

Currently RaNNC does not support *TF32 (TensorFloat-32)*.

CHAPTER

FOUR

FAQS

4.1 Does RaNNC work with Apex AMP?

Yes. Convert your model with `amp.initialize()` and pass the resulting model to `RaNNCModule` with `use_amp_master_params=True`.

4.2 How to save/load a RaNNC module

Use `state_dict()` of the RaNNC module. The returned *state_dict* can be saved and loaded as with PyTorch.

Please make sure `state_dict()` must be called from all ranks. Otherwise, the call of `state_dict()` is blocked because RaNNC gathers parameters across all ranks.

4.3 How to use gradient accumulation

As default, RaNNC implicitly performs allreduce (sum) of gradients on all ranks after a backward pass. To prevent the allreduce, you can use `pyrannc.delay_grad_allreduce(False)`.

After a specified number of forward/backward steps, you can explicitly perform allreduce with `allreduce_grads` of your `RaNNCModule`.

4.4 My model takes too long until partitioning is determined

By setting `save_deployment=true`, RaNNC outputs the deployment state to a file `deployment_file` after partitioning is determined. You can load the deployment file by setting `load_deployment=true`. This greatly save your time if you run a program using RaNNC with similar settings, e.g. with different learning rate. (See also *Configurations*)

When you are unsure that partitioning process keeps going or already failed, you can change the log level of the partitioning module. Changing log levels of `MLPartitioner` and `DPStaging` will show you the progress of partitioning process. (See also *Logging*)

4.5 Custom cpp functions does not work with RaNNC

4.6 How to use a model that takes kwargs

4.7 Does RaNNC work with torch.distributed package?

CHAPTER

FIVE

API REFERENCES

CHAPTER

SIX

LOGGING

RaNNC uses spdlog and spdlog_setup for logging. You can configure the logging by a configuration file placed at `~/.pyrannc/logging.toml`.

Since RaNNC has loggers associated with internal modules, you can set a log level for each module. The below shows an example of the loggign configuration file.

```
global_pattern = "[%Y-%m-%d %T.%f] [%L] <%n>: %v"

# Sinks
[[sink]]
name = "console_st"
type = "stdout_sink_st"

[[sink]]
name = "stderr_st"
type = "color_stdout_sink_st"

# Loggers
[[logger]]
name = "root"
sinks = ["console_st"]
level = "info"

[[logger]]
name = "RaNNCModule"
sinks = ["stderr_st"]
level = "info"

[[logger]]
name = "RaNNCProcess"
sinks = ["stderr_st"]
level = "info"

[[logger]]
name = "GraphLauncher"
sinks = ["stderr_st"]
level = "info"

[[logger]]
name = "GraphValueStorage"
sinks = ["stderr_st"]
level = "info"

[[logger]]
```

(continues on next page)

(continued from previous page)

```
name = "GraphUtil"
sinks = ["stderr_st"]
level = "info"

[[logger]]
name = "Decomposer"
sinks = ["stderr_st"]
level = "info"

[[logger]]
name = "Decomposition"
sinks = ["stderr_st"]
level = "info"

[[logger]]
name = "GraphProfiler"
sinks = ["stderr_st"]
level = "info"

[[logger]]
name = "ParamStorage"
sinks = ["stderr_st"]
level = "info"

[[logger]]
name = "GraphConnector"
sinks = ["stderr_st"]
level = "info"

[[logger]]
name = "TorchDriver"
sinks = ["stderr_st"]
level = "info"

[[logger]]
name = "AllReduceRunner"
sinks = ["stderr_st"]
level = "info"

[[logger]]
name = "MLPartitioner"
sinks = ["stderr_st"]
level = "info"

[[logger]]
name = "DPStaging"
sinks = ["stderr_st"]
level = "info"
```

CHAPTER

SEVEN

BUILDING FROM SOURCE

7.1 Compiler version

You must use GCC v5.4 or newer. We tested RaNNC with GCC v5.4 and v7.1. Note that, however, RaNNC must be built complying ABI of PyTorch.

RaNNC is built with *Pre-cxx11 ABI* (`_GLIBCXX_USE_CXX11_ABI=0`) as default because PyTorch installed via conda is built with *Pre-cxx11 ABI*. You can change the ABI setting in `CMakeLists.txt`. PyTorch provides you with a function below to know how the binary is compiled.

7.2 Build and Install

You need to set some environment variables before building RaNNC to help cmake find dependent libraries.

Table 1: Variables for building configurations

Variable	
CUDA_HOME	Path to a CUDA runtime directory.
MPI_DIR	Path to an MPI installation directory.
BOOST_DIR	Path to a Boost libraries directory.
CUDNN_ROOT_DIR	Path to a cuDNN libraries directory.
LD_LIBRARY_PATH	Must contain the path to NCCL lib directory.

The building process refers to PyTorch installed with conda. Therefore, install PyTorch with your python and run `setup.py`. The following script shows configurations to install RaNNC from the source.

```
#!/usr/bin/env bash

 # Activate conda
 source [CONDA_PATH]/etc/profile.d/conda.sh
 conda activate rannc

 # Set dependencies
 export CUDA_HOME="$(dirname $(which nvcc))/../"
 export MPI_DIR="$(dirname $(which ompi_info))/../"
 export BOOST_DIR=[BOOST_DIR_PATH]
 export CUDNN_ROOT_DIR=[YOUR_CUDNN_DIR_PATH]

 python setup.py build -g install
```

Makefiles under `docker/` show the complete process to build and install RaNNC. They are used to build pip packages.

CHAPTER

EIGHT

CONFIGURATIONS

RaNNC's runtime configurations can be set in the following two ways:

- *Config file*: RaNNC automatically loads a configuration file at `~/.pyrannc/rannc_conf.toml`. Names of configuration items must be in lower case. The path to the configuration file can be set by an environment variable `RANNC_CONF_DIR`.
- *Environment variables*: You can overwrite configuration by setting environment variables. Names of variables follows `RANNC_<CONF_ITEM_NAME>` in upper case. For example, you can set *mem_margin* in the following table by a variable `RANNC_MEM_MARGIN`.

Table 1: Configurations

Name	Default	
mem_margin	0.1	Memory margin for model partitioning.
save_deployment	true	Save deployment of a partitioned model if set to true.
load_deployment	false	Load deployment of a partitioned model if set to true.
deployment_file	`/tmp/rannc_deployment.bin`	Path of deployment file to save/load.
min_pipeline	1	Minimum number of microbatches for pipeline parallelism
max_pipeline	32	Maxmum number of microbatches for pipeline parallelism
opt_param_factor	2	Factor to estimate memory usage by an optimizer. For example, Set this item to 2 for Adam because the optimizer uses two internal data *v* and *s*, whose sizes are equivalent to parameter tensors.
trace_events	false	Trace internal events if set to true. When true, the event tracing significantly degrades performance.
event_trace_file	`/tmp/rannc_event_trace.json`	Path to an event trace file.

The following is an example of the configuration file.

```
profiling=false
dump_graph=false
dump_graph_prefix="graph_dump"
partition_num=2
replica_num=1
pipeline_num=1
validate_comm=false
display_comm_value=false
profiling_iter=1
consolidate_grads=false
```

(continues on next page)

(continued from previous page)

```
checkpointing=true
checkpointing_no_last=false
auto_parallel=false
p2p_comm=true
opt_param_factor=2
min_partition_num=5
max_partition_num=30
mem_margin=0.1
do_uncoarsening=true
min_pipeline=1
max_pipeline=4
save_deployment=true
load_deployment=false
save_graph_profile=false
load_graph_profile=false
trace_events=false
verify_recomp=false
coarsen_by_time=false
skip_grad_scaling=false
```