FlexFlow

CMU, Facebook, LANL, MIT, NVIDIA, and Stanford (alphabetical)

Sep 02, 2023

GETTING STARTED

CHAPTER

ONE

OVERVIEW

1.1 News :

- [09/02/2023] Adding AMD GPU support, released Docker images for ROCM 5.3->5.6
- [08/16/2023] Adding Starcoder model support
- [08/14/2023] Released Docker image for different CUDA versions

1.2 Install FlexFlow

1.2.1 Requirements

- OS: Linux
- GPU backend: Hip-ROCm or CUDA
  - CUDA version: 10.2 – 12.0
  - NVIDIA compute capability: 6.0 or higher
- Python: 3.6 or higher
- Package dependencies: see here

1.2.2 Install with pip

You can install FlexFlow using pip:

```
pip install flexflow
```

1.2.3 Try it in Docker

If you run into any issue during the install, or if you would like to use the C++ API without needing to install from source, you can also use our pre-built Docker package for different CUDA versions and the `hip_rocm` backend. To download and run our pre-built Docker container:

```
docker run --gpus all -it --rm --shm-size=8g ghcr.io/flexflow/flexflow-cuda-12.0:latest
```

To download a Docker container for a backend other than CUDA v12.0, you can replace the `cuda-12.0` suffix with any of the following backends: `cuda-11.1`, `cuda-11.2`, `cuda-11.3`, `cuda-11.4`, `cuda-11.5`, `cuda-11.6`, `cuda-11.7`, `cuda-11.8`, and `hip_rocm-5.3`, `hip_rocm-5.4`, `hip_rocm-5.5`, `hip_rocm-5.6`). More info on the Docker images, with instructions to build a new image from source, or run with additional configurations, can be found here.

1.2.4 Build from source

You can install FlexFlow Serve from source code by building the inference branch of FlexFlow. Please follow these instructions.

1.3 Get Started!

To get started, check out the quickstart guides below for the FlexFlow training and serving libraries.

- FlexFlow Train
- FlexFlow Serve

1.4 Contributing

Please let us know if you encounter any bugs or have any suggestions by submitting an issue.

We welcome all contributions to FlexFlow from bug fixes to new features and extensions.

1.5 Citations

FlexFlow Serve:

- Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, Zhihao Jia. SpecInfer: Accelerating Generative Large Language Model Serving with Speculative Inference and Token Tree Verification. In ArXiV, May 2023.

FlexFlow Train:

- Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. Unity: Accelerating DNN Training Through Joint Optimization of Algebraic Transformations and Parallelization. In Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI), July 2022.
- Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond Data and Model Parallelism for Deep Neural Networks. In Proceedings of the 2nd Conference on Machine Learning and Systems (MLSys), April 2019.

- Zhihao Jia, Sina Lin, Charles R. Qi, and Alex Aiken. Exploring Hidden Dimensions in Parallelizing Convolutional Neural Networks. In Proceedings of the International Conference on Machine Learning (ICML), July 2018.

1.6 The Team

FlexFlow is developed and maintained by teams at CMU, Facebook, Los Alamos National Lab, MIT, and Stanford (alphabetically).

1.7 License

FlexFlow uses Apache License 2.0.

CHAPTER

TWO

BUILDING FROM SOURCE

2.1 1. Download the source code

Clone the FlexFlow source code, and the third-party dependencies from GitHub.

```
git clone --recursive https://github.com/flexflow/FlexFlow.git
```

2.2 2. Install system dependencies

FlexFlow has system dependencies on cuda and/or rocm depending on which gpu backend you target. The gpu backend is configured by the cmake variable `FF_GPU_BACKEND`. By default, FlexFlow targets CUDA. `docker/base/Dockerfile` installs system dependencies in a standard ubuntu system.

2.2.1 Targeting CUDA - `FF_GPU_BACKEND=cuda`

If you are targeting CUDA, FlexFlow requires CUDA and CUDNN to be installed. You can follow the standard nvidia installation instructions CUDA and CUDNN.

Disclaimer: CUDA architectures < 60 (Maxwell and older) are no longer supported.

2.2.2 Targeting ROCM - `FF_GPU_BACKEND=hip_rocm`

If you are targeting ROCM, FlexFlow requires a ROCM and HIP installation with a few additional packages. Note that this can be done on a system with or without an AMD GPU. You can follow the standard installation instructions ROCM and HIP. When running `amdgpu-install`, install the use cases hip and rocm. You can avoid installing the kernel drivers (not necessary on systems without an AMD graphics card) with `--no-dkms` I.e. `amdgpu-install --usecase=hip,rocm --no-dkms`. Additionally, install the packages `hip-dev`, `hipblas`, `miopen-hip`, and `rocm-hip-sdk`.

See `./docker/base/Dockerfile` for an example ROCM install.

2.2.3 Targeting CUDA through HIP - `FF_GPU_BACKEND=hip_cuda`

This is not currently supported.

2.3 3. Install the Python dependencies

If you are planning to build the Python interface, you will need to install several additional Python libraries, please check this for details. If you are only looking to use the C++ interface, you can skip to the next section.

We recommend that you create your own ``conda`` environment and then install the Python dependencies, to avoid any version mismatching with your system pre-installed libraries.

The `conda` environment can be created and activated as:

```
conda env create -f conda/environment.yml
conda activate flexflow
```

2.4 4. Configuring the FlexFlow build

You can configure a FlexFlow build by running the `config/config.linux` file in the build folder. If you do not want to build with the default options, you can set your configurations by passing (or exporting) the relevant environment variables. We recommend that you spend some time familiarizing with the available options by scanning the `config/config.linux` file. In particular, the main parameters are:

1. `CUDA_DIR` is used to specify the directory of CUDA. It is only required when CMake can not automatically detect the installation directory of CUDA.
2. `CUDNN_DIR` is used to specify the directory of CUDNN. It is only required when CUDNN is not installed in the CUDA directory.
3. `FF_CUDA_ARCH` is used to set the architecture of targeted GPUs, for example, the value can be 60 if the GPU architecture is Pascal. To build for more than one architecture, pass a list of comma separated values (e.g. `FF_CUDA_ARCH=70,75`). To compile FlexFlow for all GPU architectures that are detected on the machine, pass `FF_CUDA_ARCH=autodetect` (this is the default value, so you can also leave `FF_CUDA_ARCH` unset. If you want to build for all GPU architectures compatible with FlexFlow, pass `FF_CUDA_ARCH=all`. **If your machine does not have any GPU, you have to set FF_CUDA_ARCH to at least one valid architecture code (or ``all``)**, since the compiler won't be able to detect the architecture(s) automatically.
4. `FF_USE_PYTHON` controls whether to build the FlexFlow Python interface.
5. `FF_USE_NCCL` controls whether to build FlexFlow with NCCL support. By default, it is set to ON.
6. `FF_LEGION_NETWORKS` is used to enable distributed run of FlexFlow. If you want to run FlexFlow on multiple nodes, follow instructions in the Multinode tutorial and set the corresponding parameters as follows:

- To build FlexFlow with GASNet, set `FF_LEGION_NETWORKS=gasnet` and `FF_GASNET_CONDUIT` as a specific conduit (e.g. `ibv`, `mpi`, `udp`, `ucx`) in `config/config.linux` when configuring the FlexFlow build. Set `FF_UCX_URL` when you want to customize the URL to download UCX.
- To build FlexFlow with native UCX, set `FF_LEGION_NETWORKS=ucx` in `config/config.linux` when configuring the FlexFlow build. Set `FF_UCX_URL` when you want to customize the URL to download UCX.

1. `FF_BUILD_EXAMPLES` controls whether to build all C++ example programs.
2. `FF_MAX_DIM` is used to set the maximum dimension of tensors, by default it is set to 4.

3. `FF_USE_{NCCL,LEGION,ALL}_PRECOMPILED_LIBRARY`, controls whether to build FlexFlow using a precompiled version of the Legion, NCCL (if `FF_USE_NCCL` is `ON`), or both libraries . By default, `FF_USE_NCCL_PRECOMPILED_LIBRARY` and `FF_USE_LEGION_PRECOMPILED_LIBRARY` are both set to `ON`, allowing you to build FlexFlow faster. If you want to build Legion and NCCL from source, set them to `OFF`.

More options are available in cmake, please run `ccmake` and search for options starting with FF.

2.5 5. Build FlexFlow

You can build FlexFlow in three ways: with CMake, with Make, and with `pip`. We recommend that you use the CMake building system as it will automatically build all C++ dependencies inlcuding NCCL and Legion.

2.5.1 Building FlexFlow with CMake

To build FlexFlow with CMake, go to the FlexFlow home directory, and run

```
mkdir build
cd build
../config/config.linux
make -j N
```

where N is the desired number of threads to use for the build.

2.5.2 Building FlexFlow with pip

To build Flexflow with `pip`, run `pip install .` from the FlexFlow home directory. This command will build FlexFlow, and also install the Python interface as a Python module.

2.5.3 Building FlexFlow with Make

The Makefile we provide is mainly for development purposes, and may not be fully up to date. To use it, run:

```
cd python
make -j N
```

2.6 6. Test FlexFlow

After building FlexFlow, you can test it to ensure that the build completed without issue, and that your system is ready to run FlexFlow.

2.6.1 Set the FF_HOME environment variable before running FlexFlow. To make it permanent, you can add the following line in ~/.bashrc.

```
export FF_HOME=/path/to/FlexFlow
```

2.6.2 Run FlexFlow Python examples

The Python examples are in the examples/python. The native, Keras integration and PyTorch integration examples are listed in `native`, `keras` and `pytorch` respectively.

To run the Python examples, you have two options: you can use the `flexflow_python` interpreter, available in the `build` folder, or you can use the native Python interpreter. If you choose to use the native Python interpreter, you should either install FlexFlow, or, if you prefer to build without installing, export the required environment flags by running the following command (edit the path if your build folder is not named `build`):

```
source ./build/set_python_envs.sh
```

We recommend that you run the `mnist_mlp` test under `native` using the following cmd to check if FlexFlow has been installed correctly:

```
cd "$FF_HOME"
./python/flexflow_python examples/python/native/mnist_mlp.py -ll:py 1 -ll:gpu 1 -
↪ll:fsize <size of gpu buffer> -ll:zsize <size of zero buffer>
```

A script to run all the Python examples is available at `tests/multi_gpu_tests.sh`

2.6.3 Run FlexFlow C++ examples

The C++ examples are in the examples/cpp. For example, the AlexNet can be run as:

```
./alexnet -ll:gpu 1 -ll:fsize <size of gpu buffer> -ll:zsize <size of zero buffer>
```

Size of buffers is in MBs, e.g. for an 8GB gpu `-ll:fsize 8000`

2.7 7. Install FlexFlow

If you built/installed FlexFlow using `pip`, this step is not required. If you built using Make or CMake, install FlexFlow with:

```
cd build
make install
```

CHAPTER

THREE

DOCKER

We provide a ready-to-use Docker container to quickly run FlexFlow with no manual installation required. To use it, follow the steps below.

3.1 Prerequisites

You can build and run the FlexFlow Docker images on any machine, but if you want to train or serve a model, you will need a machine with a NVIDIA or AMD GPU, with drivers installed. You will also need to have Docker and the Nvidia Container Toolkit installed on the host machine. If using an AMD GPU, follow the Deploy ROCm Docker containers instructions.

3.2 Downloading a pre-built package

The fastest way to run FlexFlow is to use one of the pre-built containers, which we update for each commit to the `inference` branch (the `inference` branch is currently ahead of the `master` branch). The available containers are the following, and can be found at this link:

- *flexflow*: the pre-built version of FlexFlow. We currently publish four version targeting AMD GPUs (ROCm versions: 5.3, 5.4, 5.5 and 5.6), and several versions for CUDA GPUs (CUDA versions: 11.1, 11.2, 11.3, 11.4, 11.5, 11.6, 11.7, 11.8 and 12.0). The CUDA images are named `flexflow-<GPU backend>-<GPU software version>`, e.g. flexflow-hip_rocm-5.6 or flexflow-cuda-12.0 or
- *flexflow-environment*: this is the base layer for *flexflow*. The packages are used in CI or for internal use, and contain all the dependencies needed to build/run Flexflow. You may find them useful if you want to build FlexFlow yourself. We also publish four version of *flexflow-environment* for AMD GPUs and, for NVIDIA GPUs, one for each CUDA version in the list above. The naming convention is similar, too. For example, the *flexflow-environment* image for CUDA 12.0 is tagged [flexflow-environment-cuda-12.0](https://github.com/orgs/flexflow/packages/container/package/flexflow-environment-cuda-12.0).

The easiest way to download any of the Docker containers above is to call:

```
./docker/pull.sh <CONTAINER_NAME>
```

where `CONTAINER_NAME` is `flexflow` (or `flexflow-environment`). By default, the script will assume a NVIDIA backend and attempt to detect the CUDA version on your machine, to download the relevant container. If your machine has AMD GPUs, or no GPUs, or if you want to specify the CUDA/ROCM version to download, set the environment variables below:

- `FF_GPU_BACKEND` (supported options: `cuda`, `hip_rocm`) to specify the GPU backend of the Docker container to be downloaded.

- `cuda_version` (supported options: 11.1, 11.2, 11.3, 11.4, 11.5, 11.6, 11.7, 11.8 and 12.0) to specify the CUDA version, when using a `cuda` backend. If `FF_GPU_BACKEND` is set to `hip_rocm`, the `cuda_version` env will be ignored
- `hip_version` (supported options: 5.3, 5.4, 5.5, 5.6) to specify the ROCm version, when using a HIP backend. If `FF_GPU_BACKEND` is set to `cuda`, the `hip_version` env will be ignored.

After downloading a container you can use the `run.sh` script to run it by following the instructions in the section below.

3.3 Building a Docker container from scratch

If you prefer to build one of the Docker containers from scratch, you can do so with the help of the `build.sh` script. You can configure the build via the same environment variables that you'd use to configure a CMake build (refer to the Installation guide and to the `config/config.linux` file). For example, to build for a CUDA backend, you can export `FF_GPU_BACKEND=cuda` (you can also omit this since `cuda` is the default value for `FF_GPU_BACKEND`). When building for the `cuda` backend, you can pick the CUDA version by setting the optional environment variable `cuda_version`, e.g.: `export cuda_version=12.0`. Leaving the `cuda_version` variable blank will let the script autodetect the CUDA version installed on the host machine, and build for that version. Setting the `cuda_version` env will have no effect when building for a GPU backend other than CUDA. Similarly, you can pick the ROCm version by setting `hip_version` when the backend is `FF_GPU_BACKEND=hip_rocm`, whereas the env will be ignored for non-HIP backends.

To build the FlexFlow container, run (the `flexflow` argument of the build script can be omitted):

```
./docker/build.sh flexflow
```

If you only want to build the `flexflow-environment` image (the base layers of the `flexflow` container, used in CI and for other internal purposes), run:

```
./docker/build.sh flexflow-environment
```

3.4 Running a Docker container

After having either built or downloaded a Docker container by following the instructions above, you can run it with the following command (image name argument of the run script can be omitted). Once again, you can set the `FF_GPU_BACKEND`, `cuda_version` and `hip_version` optional environment variables to run the docker image with the desired GPU backend and CUDA/HIP version:

- `FF_GPU_BACKEND` (supported options: `cuda`, `hip_rocm`) to specify the GPU backend of the Docker container to be run.
- `cuda_version` (supported options: 11.1, 11.2, 11.3, 11.4, 11.5, 11.6, 11.7, 11.8 and 12.0) to specify the CUDA version, when using a `cuda` backend. If `FF_GPU_BACKEND` is set to `hip_rocm`, the `cuda_version` env will be ignored
- `hip_version` (supported options: 5.3, 5.4, 5.5, 5.6) to specify the ROCm version, when using a HIP backend. If `FF_GPU_BACKEND` is set to `cuda`, the `hip_version` env will be ignored.

Leaving these variables unset will assume a GPU backend, and instruct the script to autodetect the CUDA version installed on the current machine and run the Docker container with it if available.

```
./docker/run.sh --image_name flexflow
```

If you wish to run the `flexflow-environment` container, run:

```
./docker/run.sh --image_name flexflow-environment
```

N.B.: If you don't have GPUs available on the machine, or you wish to run the docker image without attaching GPUs, you can set the environment variable `ATTACH_GPUS=false` before running the script.

CHAPTER

FOUR

MULTINODE TUTORIAL

4.1 1. Spin up instances

Spin up multiple instances with GPU support. For AWS, we recommend using p3.2xlarge with Deep Learning AMI GPU PyTorch 1.13.1 (Ubuntu 20.04) to simplify the procedure.

Place the instances in a placement group that utilizes the `cluster` strategy to achieve low-latency network performance.

To enable communication between instances, attach the same security group to all instances and add an inbound rule in the security group to allow all incoming traffic from the same security group. An example inbound rule is as follows:

```
Type: Custom TCP
Port range: 1 - 65535
Source: Custom (use the security group ID)
```

You can also use your own GPU cluster, as long as all machines are interconnected with a low-latency network.

4.2 2. Configure and build FlexFlow

Follow steps 1 to 5 in the Build from source guide to download the source code, install system dependencies, install the Python dependencies, configure the FlexFlow build, and build FlexFlow **on each instance at the same path**.

You can skip step 2 (Install system dependencies) if you have spun up instances with Deep Learning AMI, which comes preconfigured with CUDA. Otherwise, you need to install system dependencies on each instance.

For step 4 (Configuring the FlexFlow build), make sure to specify a network using the `FF_LEGION_NETWORKS` parameter. We recommend using `FF_LEGION_NETWORKS=gasnet` and `FF_GASNET_CONDUIT=ucx`. Other configurations are optional.

4.3 3. Configure MPI

MPI is an easy way to launch FlexFlow across all instances simultaneously and set up communication between them.

To use MPI, enable non-interactive `ssh` logins between instances. This can be done by referring to the Open MPI documentation. Here are the detailed steps:

1. Choose one of the nodes as the main instance and create a public/private key pair on the instance. This will be the instance from which you launch MPI commands. Run the following command:

```
ssh-keygen -t ed25519
```

This will create a public key at `~/.ssh/id_ed25519.pub` and a private key at `~/.ssh/id_ed25519`.

1. Append the contents of the **public key** to `~/.ssh/authorized_keys` on all machines (if the file does not exist, create one). Execute the following command on **all instances**:

```
mkdir -p ~/.ssh
echo '<public key>' >> ~/.ssh/authorized_keys
```

Replace `<public key>` with the public key from `~/.ssh/id_ed25519.pub` on the main instance. It should be a single line containing a string like:

```
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIOy5NKYdE8Cwgid59rx6xMqyj9vLaWuXIwy/BSRiK4su instance
```

1. Create a hostfile at `~/hostfile`, with one line for each instance (add more lines if you have more instances):

```
<host1_private_ip> slots=<slot1>
<host2_private_ip> slots=<slot2>
```

`<slot1>` and `<slot2>` refer to the number of slots available for each instance, respectively. Set it to one if you have a GPU on each instance.

1. SSH into each host and make sure you can log into them. It may ask you to verify the public key. Make sure to trust the public key so that it doesn't ask you again.
2. Test MPI by running `mpirun -N 1 --hostfile ~/hostfile hostname`. It should display the hostname of all your nodes. If you encounter any errors like `WARNING: Open MPI accepted a TCP connection from what appears to be another Open MPI process but cannot find a corresponding process entry for that peer.`, add the parameter `--mca btl_tcp_if_include` in the `mpirun` command (refer to this Stack Overflow question).

4.4 4. Test FlexFlow

Follow step 6 in the Build from source guide to set environment variables.

A script to run a Python example on multiple nodes is available at `scripts/mnist_mlp_run.sh`. You can run the script using ``*mpirun*` <https://www.open-mpi.org/doc/current/man1/mpirun.1.php>`_ (if you configured it in step 3) or ``*srun*` <https://slurm.schedmd.com/srun.html>`_.

CHAPTER

FIVE

SERVING OVERVIEW

5.1 What is FlexFlow Serve

The high computational and memory requirements of generative large language models (LLMs) make it challenging to serve them quickly and cheaply. FlexFlow Serve is an open-source compiler and distributed system for **low latency**, **high performance** LLM serving. FlexFlow Serve outperforms existing systems by 1.3-2.0x for single-node, multi-GPU inference and by 1.4-2.4x for multi-node, multi-GPU inference.

5.2 Quickstart

The following example shows how to deploy an LLM using FlexFlow Serve and accelerate its serving using *speculative inference*. First, we import `flexflow.serve` and initialize the FlexFlow Serve runtime. Note that `memory_per_gpu` and `zero_copy_memory_per_node` specify the size of device memory on each GPU (in MB) and zero-copy memory on each node (in MB), respectively. We need to make sure the aggregated GPU memory and zero-copy memory are **both** sufficient to store LLM parameters in non-offloading serving. FlexFlow Serve combines tensor and pipeline model parallelism for LLM serving.

```
import flexflow.serve as ff

ff.init(
        num_gpus=4,
        memory_per_gpu=14000,
        zero_copy_memory_per_node=30000,
        tensor_parallelism_degree=4,
        pipeline_parallelism_degree=1
    )
```

Second, we specify the LLM to serve and the SSM(s) used to accelerate LLM serving. The list of supported LLMs and SSMs is available at *supported models*.

```
# Specify the LLM
llm = ff.LLM("decapoda-research/llama-7b-hf")

# Specify a list of SSMs (just one in this case)
ssms=[]
ssm = ff.SSM("JackFram/llama-68m")
ssms.append(ssm)
```

Next, we declare the generation configuration and compile both the LLM and SSMs. Note that all SSMs should run in the **beam search** mode, and the LLM should run in the **tree verification** mode to verify the speculated tokens from SSMs.

```
# Create the sampling configs
generation_config = ff.GenerationConfig(
    do_sample=False, temperature=0.9, topp=0.8, topk=1
)

# Compile the SSMs for inference and load the weights into memory
for ssm in ssms:
    ssm.compile(generation_config)

# Compile the LLM for inference and load the weights into memory
llm.compile(generation_config, ssms=ssms)
```

Finally, we call `llm.generate` to generate the output, which is organized as a list of `GenerationResult`, which include the output tokens and text.

```
result = llm.generate("Here are some travel tips for Tokyo:\n")
```

5.2.1 Incremental decoding

5.2.2 C++ interface

If you'd like to use the C++ interface (mostly used for development and benchmarking purposes), you should install from source, and follow the instructions below.

5.3 Speculative Inference

A key technique that enables FlexFlow Serve to accelerate LLM serving is speculative inference, which combines various collectively boost-tuned small speculative models (SSMs) to jointly predict the LLM's outputs; the predictions are organized as a token tree, whose nodes each represent a candidate token sequence. The correctness of all candidate token sequences represented by a token tree is verified against the LLM's output in parallel using a novel tree-based parallel decoding mechanism. FlexFlow Serve uses an LLM as a token tree verifier instead of an incremental decoder, which largely reduces the end-to-end inference latency and computational requirement for serving generative LLMs while provably preserving model quality.

5.3.1 Supported LLMs and SSMs

FlexFlow Serve currently supports all HuggingFace models with the following architectures:

- `LlamaForCausalLM` / `LLaMAForCausalLM` (e.g. LLaMA/LLaMA-2, Guanaco, Vicuna, Alpaca, ...)
- `OPTForCausalLM` (models from the OPT family)
- `RWForCausalLM` (models from the Falcon family)
- `GPTBigCodeForCausalLM` (models from the Starcoder family)

Below is a list of models that we have explicitly tested and for which a SSM may be available:

5.3.2 CPU Offloading

FlexFlow Serve also offers offloading-based inference for running large models (e.g., llama-7B) on a single GPU. CPU offloading is a choice to save tensors in CPU memory, and only copy the tensor to GPU when doing calculation. Notice that now we selectively offload the largest weight tensors (weights tensor in Linear, Attention). Besides, since the small model occupies considerably less space, it it does not pose a bottleneck for GPU memory, the offloading will bring more runtime space and computational cost, so we only do the offloading for the large model. [TODO: update instructions] You can run the offloading example by enabling the `-offload` and `-offload-reserve-space-size` flags.

5.3.3 Quantization

FlexFlow Serve supports int4 and int8 quantization. The compressed tensors are stored on the CPU side. Once copied to the GPU, these tensors undergo decompression and conversion back to their original precision. Please find the compressed weight files in our s3 bucket, or use this script from FlexGen project to do the compression manually. [TODO: update instructions for quantization].

5.3.4 Prompt Datasets

We provide five prompt datasets for evaluating FlexFlow Serve: Chatbot instruction prompts, ChatGPT Prompts, WebQA, Alpaca, and PIQA.

5.4 TODOs

FlexFlow Serve is still under active development. We currently focus on the following tasks and strongly welcome all contributions from bug fixes to new features and extensions.

- AMD benchmarking. We are actively working on benchmarking FlexFlow Serve on AMD GPUs and comparing it with the performance on NVIDIA GPUs.
- Chatbot prompt templates and Multi-round conversations
- Support for FastAPI server
- Integration with LangChain for document question answering

5.5 Acknowledgements

This project is initiated by members from CMU, Stanford, and UCSD. We will be continuing developing and supporting FlexFlow Serve. Please cite FlexFlow Serve as:

```
@misc{miao2023specinfer,
      title={SpecInfer: Accelerating Generative Large Language Model Serving with␣
↪Speculative Inference and Token Tree Verification},
      author={Xupeng Miao and Gabriele Oliaro and Zhihao Zhang and Xinhao Cheng and Zeyu␣
↪Wang and Rae Ying Yee Wong and Alan Zhu and Lijie Yang and Xiaoxiang Shi and Chunan␣
↪Shi and Zhuoming Chen and Daiyaan Arfeen and Reyna Abhyankar and Zhihao Jia},
      year={2023},
      eprint={2305.09781},
      archivePrefix={arXiv},
      primaryClass={cs.CL}
}
```

5.6 License

FlexFlow uses Apache License 2.0.

CHAPTER

SIX

TRAINING OVERVIEW

6.1 PyTorch Support

Users can also use FlexFlow Train to optimize the parallelization performance of existing PyTorch models in two steps. First, a PyTorch model can be exported to the FlexFlow model format using `flexflow.torch.fx.torch_to_flexflow`.

```
import torch
import flexflow.torch.fx as fx

model = MyPyTorchModule()
fx.torch_to_flexflow(model, "mymodel.ff")
```

Second, a FlexFlow Train program can directly import a previously saved PyTorch model and autotune the parallelization performance for a given parallel machine.

```
from flexflow.pytorch.model import PyTorchModel

def top_level_task():
  torch_model = PyTorchModel("mymodel.ff")
  output_tensor = torch_model.apply(ffmodel, input_tensor)
  ## Model compilation
  ffmodel.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=[
↪'accuracy'])
  ## Model training
  (x_train, y_train) = cifar10.load_data()
  ffmodel.fit(x_train, y_train, epochs=30)
```

More FlexFlow PyTorch examples: see the pytorch examples folder.

6.2 TensorFlow Keras and ONNX Support

FlexFlow Train prioritizes PyTorch compatibility, but also includes frontends for Tensorflow Keras and ONNX models.

6.3 C++ Interface

For users that prefer to program in C/C++. FlexFlow Train supports a C++ program inference that is equivalent to its Python APIs.

More FlexFlow C++ examples: see the C++ examples folder.

6.4 Command-Line Flags

In addition to setting runtime configurations in a FlexFlow Train Python/C++ program, the FlexFlow Train runtime also accepts command-line arguments for various runtime parameters:

FlexFlow training flags:

- `-e` or *–epochs*: number of total epochs to run (default: 1)
- `-b` or *–batch-size*: global batch size in each iteration (default: 64)
- `-p` or *–print-freq*: print frequency (default: 10)
- `-d` or `--dataset`: path to the training dataset. If not set, synthetic data is used to conduct training.

Legion runtime flags:

- *-ll:gpu*: number of GPU processors to use on each node (default: 0)
- `-ll:fsize`: size of device memory on each GPU (in MB)
- `-ll:zsize`: size of zero-copy memory (pinned DRAM with direct GPU access) on each node (in MB). This is used for prefecthing training images from disk.
- *-ll:cpu*: number of data loading workers (default: 4)
- *-ll:util*: number of utility threads to create per process (default: 1)
- *-ll:bgwork*: number of background worker threads to create per process (default: 1)

Performance auto-tuning flags:

- `--search-budget` or *–budget*: the number of iterations for the MCMC search (default: 0)
- `--search-alpha` or *–alpha*: a hyper-parameter for the search procedure (default: 0.05)
- `--export-strategy` or *–export*: path to export the best discovered strategy (default: None)
- `--import-strategy` or *–import*: path to import a previous saved strategy (default: None)
- `--enable-parameter-parallel`: allow FlexFlow Train to explore parameter parallelism for performance auto-tuning. (By default FlexFlow Train only considers data and model parallelism.)
- `--enable-attribute-parallel`: allow FlexFlow Train to explore attribute parallelism for performance auto-tuning. (By default FlexFlow Train only considers data and model parallelism.) For performance tuning related flags: see performance autotuning.

CHAPTER

SEVEN

TRAINING INTERFACE

7.1 Keras Interface

FlexFlow provides a drop-in replacement for TensorFlow Keras. Running an existing Keras program on the FlexFlow backend only requires a few lines of changes to the program. The detailed instructions are as follows:

7.1.1 1. Replace the Keras header files

Redirect the program to import Keras functions from FlexFlow by using the following import header lines:

```
from flexflow.keras.models import Model, Sequential
from flexflow.keras.layers import Input, Dense, Conv2D, ...
from flexflow.keras.callbacks import Callback, ...
```

7.1.2 2. Modify the main Keras program

FlexFlow requires a Keras program to wrap its model construction in a Python function called `top_level_task()`. This allows FlexFlow to automatically parallelize DNN training across all GPUs on all compute nodes. For example, the following code snippet shows parallelizing AlexNet training in FlexFlow:

```
def top_level_task():
  model = Sequential()
  model.add(Conv2D(filters=64, input_shape=(3,229,229), kernel_size=(11,11), strides=(4,
↪4), padding=(2,2), activation="relu"))
  model.add(MaxPooling2D(pool_size=(3,3), strides=(2,2), padding="valid"))
  model.add(Conv2D(filters=192, kernel_size=(5,5), strides=(1,1), padding=(2,2),␣
↪activation="relu"))
  ## More lines for model construction
  model.add(Activation("softmax"))
  ## Model compilation
  model.compile(optimizer='sgd', loss='sparse_categorical_crossentropy', metrics=[
↪'accuracy'])
  ## Model training
  (x_train, y_train) = cifar10.load_data()
  model.fit(x_train, y_train, epochs=30)

if __name__ == "__main__":
  top_level_task()
```

More FlexFlow Keras examples are available on GitHub.

7.2 PyTorch Interface

Users can use FlexFlow to optimize the parallelization performance of existing PyTorch models in two steps. The PyTorch support requires the PyTorch FX module, so make sure your PyTorch is up to date.

7.2.1 1. Export a PyTorch Model to an external file

A PyTorch model can be exported to the FlexFlow model format and saved into an external file:

```
import torch
import flexflow.torch.fx as fx

# create a PyTorch Model
class MyPyTorchModule(nn.Module):
...

# export the PyTorch Model to an external file
model = MyPyTorchModule()
fx.torch_to_flexflow(model, "filename")
```

7.2.2 2. Import a FlexFlow model from a external file

A FlexFlow program can directly import a previously saved PyTorch model and autotune the parallelization performance for a given parallel machine:

```
from flexflow.torch.model import PyTorchModel

# create input tensors
dims_input = [ffconfig.get_batch_size(), 3, 32, 32]
input_tensor = ffmodel.create_tensor(dims_input, DataType.DT_FLOAT)

# create a flexflow model from the file
torch_model = PyTorchModel("filename")
output_tensor = torch_model.apply(ffmodel, [input_tensor])

# use the Python API to train the model
ffoptimizer = SGDOptimizer(ffmodel, 0.01)
ffmodel.set_sgd_optimizer(ffoptimizer)
ffmodel.compile(
    loss_type=LossType.LOSS_SPARSE_CATEGORICAL_CROSSENTROPY,
    metrics=[
        MetricsType.METRICS_ACCURACY,
        MetricsType.METRICS_SPARSE_CATEGORICAL_CROSSENTROPY,
    ],
)
...
ffmodel.fit(x=dataloader_input, y=dataloader_label, epochs=epochs)
```

More FlexFlow PyTorch examples are available on GitHub.

7.3 ONNX Support

Similar to the PyTorch front-end, FlexFlow also supports training existing ONNX models. Since both ONNX and FlexFlow use Protocol Buffer, make sure they are linked with the Protocol Buffer of the same version.

7.3.1 1. Export a ONNX Model to a external file

A PyTorch model can be exported to the FlexFlow model format and saved into an external file:

```
import onnx
import torch
import torch.nn as nn
from torch.onnx import TrainingMode

# create a PyTorch Model
class MyPyTorchModule(nn.Module):
...

# export the PyTorch model to a ONNX model
model = MyPyTorchModule()
torch.onnx.export(model, (input), "filename", export_params=False, training=TrainingMode.
↪TRAINING)
```

7.3.2 2. Import a FlexFlow model from a external file

A FlexFlow program can directly import a previously saved ONNX model and autotune the parallelization performance for a given parallel machine:

```
from flexflow.torch.model import PyTorchModel

#create input tensors
dims_input = [ffconfig.get_batch_size(), 3, 32, 32]
input_tensor = ffmodel.create_tensor(dims_input, DataType.DT_FLOAT)

# create a flexflow model from the file
onnx_model = ONNXModel("cifar10_cnn.onnx")
output_tensor = onnx_model.apply(ffmodel, {"input.1": input_tensor})

# use the Python API to train the model
ffoptimizer = SGDOptimizer(ffmodel, 0.01)
ffmodel.set_sgd_optimizer(ffoptimizer)
ffmodel.compile(loss_type=LossType.LOSS_SPARSE_CATEGORICAL_CROSSENTROPY,␣
↪metrics=[MetricsType.METRICS_ACCURACY, MetricsType.METRICS_SPARSE_CATEGORICAL_
↪CROSSENTROPY])
...
ffmodel.fit(x=dataloader_input, y=dataloader_label, epochs=epochs)
```

More FlexFlow ONNX examples are available on GitHub.

CHAPTER

EIGHT

TRAINING EXAMPLES

8.1 mT5 Model

We mention a few prerequisites and tips for setting up.

- We assume access to at least one GPU and an installation of Anaconda.
- We assume PyTorch version 1.9.
- Using PyTorch and FlexFlow concurrently requires a CPU version of PyTorch.
  - To install the CPU version of `torch` (and `torchvision`), run:

    ```
    conda install pytorch==1.9.0 torchvision==0.10.0 cpuonly -c pytorch
    ```

  - To install the CPU version of `torch` from source, clone the repository, run `export USE_CUDA=0 USE_CUDNN=0 USE_MKLDNN=1`, run `git submodule sync; git submodule update --init --recursive`, and run `python setup.py develop` (or `python setup.py install`).
- We need an installation of the HuggingFace `transformers` repository.
  - To install `transformers`, run:

    ```
    conda install -c conda-forge transformers
    ```

  - To install `transformers` from source, clone the repository, and run `python setup.py develop` (or `python setup.py install`).
- To run PyTorch-FlexFlow examples, make sure to run `export FF_USE_CFFI=1` to use `cffi` instead of `pybind11`.
- Additional notes:
  - You may need to update `huggingface_hub` with:

    ```
    conda update huggingface_hub
    ```

  - If you encounter `ImportError: Found an incompatible version of torch.`, try updating to a later version of `transformers`.

8.1.1 mT5 in PyTorch

We present an example of training mT5 for the Sinhalese-English translation task from here, reusing some code from here. In this section, we walk through the training script using PyTorch, and in the next section, we walk through the training script using FlexFlow. The corresponding code may be found in `mt5_torch.py` and `mt5_ff.py`, respectively.

To download and uncompress the dataset, run:

```
cd examples/python/pytorch/mt5
wget https://object.pouta.csc.fi/Tatoeba-Challenge/eng-sin.tar
tar -xvf eng-sin.tar
gzip -d data/eng-sin/*.gz
```

This will create a directory `data/` containing a single subdirectory `data/eng-sin/` containing `test.id`, `test.src`, `test.trg`, `train.id`, `train.src`, and `train.trg`.

We extract, prepare, and save the data to `.tsv` by using `DataPreparer.data_to_tsv()` – this creates two new files, `data/train.tsv` and `data/eval.tsv`, and only needs to be done once. Then, we can train using those `.tsv` files. A base implementation for this may be found in `mt5_torch.py`, which saves the `.tsv` files, trains for some number of epochs, and outputs a `.csv` containing the predicted and actual text on the evaluation data.

```
python examples/python/pytorch/mt5/mt5_torch.py
```

Note: Running `mt5_torch.py` requires a GPU-version of PyTorch.

8.1.2 mT5 in FlexFlow

Now, we examine how to write a similar training script using FlexFlow. To begin, FlexFlow dataloaders expect the data to be passed in as `numpy` arrays and to be already preprocessed so that batches may be directly given to the model. In `mt5_ff.py`, `data_to_numpy()` converts the `.tsv` files to `.npy`, and `preprocess_train()` performs the necessary preprocessing.

Note: `data_to_numpy()` takes a while to run.

Next, following the conventional FlexFlow terminology, we define a *top-level task* to train the mT5 model. The key steps are as follows (including some notable code snippets):

- Define `ffconfig = FFConfig()` and `ffmodel = FFModel(ffconfig)` – `ffmodel` is the Python object for the FlexFlow model
- Define the PyTorch mT5 model:

  ```
  model = MT5ForConditionalGeneration.from_pretrained("google/mt5-small")
  ```

- Load the preprocessed training data from the `.npy` files
- Use `ffmodel.create_tensor()` for the `input_ids`, `attention_mask`, and `decoder_input_ids` – these are the input tensors to the model
- Construct a `PyTorchModel()` object wrapping the PyTorch model `model` to enable conversion to FlexFlow:

  ```
  hf_model = PyTorchModel(
      model, is_hf_model=True, batch_size=ffconfig.batch_size,
      seq_length=seq_length,
  )
  ```

  - We pass `is_hf_model=True` since HuggingFace models require a special `symbolic_trace()` distinct from the native PyTorch one.

  - `seq_length` is a tuple `(encoder_seq_length, decoder_seq_length)`.

- Convert the model to FlexFlow:

```
output_tensors = hf_model.to_ff(ffmodel, input_tensors)
```

- Define the optimizer `ffoptimizer`
- Compile the model:

```
ffmodel.compile(
    optimizer=ffoptimizer,
    loss_type=LossType.LOSS_SPARSE_CATEGORICAL_CROSSENTROPY,
    metrics=[
        MetricsType.METRICS_ACCURACY,
        MetricsType.METRICS_SPARSE_CATEGORICAL_CROSSENTROPY,
    ],
)
```

- Create the dataloaders for the `input_ids`, `attention_mask`, `decoder_input_ids`, and `labels`
- Initialize the model layers:

```
ffmodel.init_layers()
```

- Train the model, passing the appropriate dataloaders into `fit()`:

```
ffmodel.fit(
    x=[input_ids_dl, attention_mask_dl, decoder_ids_dl],
    y=labels_dl, batch_size=batch_size, epochs=epochs,
)
```

A base implementation may be found in `mt5_ff.py`.

```
./python/flexflow_python examples/python/pytorch/mt5/mt5_ff.py -ll:py 1 -ll:gpu 1 -
↪ll:fsize 14000 -ll:zsize 4096
```

Note: Running `mt5_ff.py` requires a CPU-version of PyTorch.

CHAPTER

NINE

PYTHON API

This section documents the Python API for FlexFlow Train.

9.1 Models API

Models API in FlexFlow is used to create models .

9.1.1 Model Creation

Model Creation

class flexflow.core.flexflow_cffi.**FFModel**

__init__(*ffconfig*)

Constructor of FFModel.

Parameters **ffconfig** (*FFConfig*) – configurations of FlexFlow and the created model.

Returns FFModel – the model.

Tensor Creation

class flexflow.core.flexflow_cffi.**FFModel**

create_tensor(*dims*, *data_type*, *create_grad=True*)

Instantiate a FlexFlow tensor.

Parameters

- **x** (*list of int*) – a shape tuple/list (integers), including the batch size.
- **data_type** (*DataType*) – the datatype of the created tensor. Options are DT_FLOAT, DT_DOUBLE, DT_INT32, DT_INT64, DT_BOOLEAN.
- **create_grad** (*bool*) – weather the tensor creates a gradients vector. If you don't specify anything, a gradients vector is used.

Returns Tensor – the output tensor.

9.1.2 Model Initialization

Compile

class flexflow.core.flexflow_cffi.**FFModel**

compile(*optimizer=None*, *loss_type=None*, *metrics=None*, *comp_mode=None*)

Configure the model for trainting. FlexFlow uses lazy initialization, so the actual creating of all operations (including creating and partitioning of weight, bias and output tensors) happen during compile.

Parameters

- **optimizer** (*Optimizer*) – optimizer instance.
- **loss_type** (*LossType*) – Enum of LossType. Options are LOSS_CATEGORICAL_CROSSENTROPY, LOSS_SPARSE_CATEGORICAL_CROSSENTROPY, LOSS_MEAN_SQUARED_ERROR_AVG_REDUCE and LOSS_MEAN_SQUARED_ERROR_SUM_REDUCE.
- **metrics** (*MetricsType*) – List of metrics to be evaluated by the model during training and testing. Each of this is a Enum of MetricsType. Options are METRICS_ACCURACY, METRICS_CATEGORICAL_CROSSENTROPY, METRICS_SPARSE_CATEGORICAL_CROSSENTROPY, METRICS_MEAN_SQUARED_ERROR, METRICS_ROOT_MEAN_SQUARED_ERROR, METRICS_MEAN_ABSOLUTE_ERROR
- **comp_mode** (*CompMode*) – Enum of CompMode. Options are COMP_MODE_TRAINING, COMP_MODE_INFERENCE

Returns None – no returns.

Initialization

class flexflow.core.flexflow_cffi.**FFModel**

init_layers()

Initialize layers.

Returns None – no returns.

9.1.3 Model Training and Testing

Fit

class flexflow.core.flexflow_cffi.**FFModel**

fit(*x=None*, *y=None*, *batch_size=None*, *epochs=1*)

Trains the model for a fixed number of epochs (iterations on a dataset).

Parameters

- **x** (*Dataloader*) – Input data. It can be a Dataloader instance or a list of Dataloader instances.
- **y** (*Dataloader*) – Target data (label). It can be a Dataloader instance or a list of Dataloader instances.

- **batch_size** (*int*) – Number of samples per gradient update. It must be identical with `-b` or `--batch-size` from the command line.
- **epochs** (*int*) – Number of epochs to train the model. An epoch is an iteration over the entire `x` and `y` data provided. The default value is 1.

Returns None – no returns.

Evaluate

class flexflow.core.flexflow_cffi.**FFModel**

eval(*x=None*, *y=None*, *batch_size=None*)

Returns the loss value & metrics values for the model in test mode.

Parameters

- **x** (*Dataloader*) – Input data. It can be a Dataloader instance or a list of Dataloader instances.
- **y** (*Dataloader*) – Target data (label). It can be a Dataloader instance or a list of Dataloader instances.
- **batch_size** (*int*) – Number of samples per gradient update. It must be identical with `-b` or `--batch-size` from the command line.
- **epochs** (*int*) – Number of epochs to train the model. An epoch is an iteration over the entire `x` and `y` data provided. The default value is 1.

Returns None – no returns.

Customized Training

class flexflow.core.flexflow_cffi.**FFModel**

backward(*seq_length=None*)

Backward propagation of all layers.

Returns None – no returns.

compute_metrics()

Compute performance metrics.

Returns None – no returns.

forward(*seq_length=None*)

Forward propagation of all layers.

Returns None – no returns.

reset_metrics()

Reset performance metrics.

Returns None – no returns.

update()

Update weights and biases of all layers.

Returns None – no returns.

zero_gradients()

Empty the gradients of all layers.

Returns None – no returns.

9.2 Layers API

Layers are the basic building blocks of neural networks in FlexFlow. The inputs of a layer consists of a tensor or a list of tensors and some state variables, and the outputs of a layer is a tensor or a list of tensors.

9.2.1 Conv2D

class flexflow.core.flexflow_cffi.**FFModel**

conv2d(*input*, *out_channels*, *kernel_h*, *kernel_w*, *stride_h*, *stride_w*, *padding_h*, *padding_w*, *activation=ActiMode.AC_MODE_NONE*, *groups=1*, *use_bias=True*, *shared_op=None*, *kernel_initializer=None*, *bias_initializer=None*, *name=None*)

This layer creates a 2D convolution kernel that is convolved with the layer `input` to produce a tensor of `output`.

The size of input tensor is (N, C_{in}, H, W) and the size of output tensor is $(N, C_{out}, H_{out}, W_{out})$, which can be calculated by:

$$C_{out} = out\_channels$$

$$K_H = kernel\_h$$

$$K_W = kernel\_w$$

$$S_H = stride\_h$$

$$S_W = stride\_w$$

$$P_H = padding\_h$$

$$P_S = padding\_s$$

$$H_{out} = (H - K_H + 2 * P_H)/S_H + 1$$

$$W_{out} = (W - K_W + 2 * P_W)/S_W + 1$$

Parameters

- **input** (*Tensor*) – the input Tensor.
- **out_channels** (*int*) – the dimensionality of the output space (i.e. the number of output filters in the convolution).
- **kernel_h** (*int*) – the height of the 2D convolution window: K_H.
- **kernel_w** (*int*) – the width of the 2D convolution window: K_W.
- **stride_h** (*int*) – the stride of the convolution along the height: S_H.
- **stride_w** (*int*) – the stride of the convolution along the width: S_W.
- **padding_h** (*int*) – the amount of implicit zero-paddings along the height: P_H.

- **padding_w** (*int*) – the amount of implicit zero-paddings along the width: P_W.
- **activation** (*ActiMode*) – Activation function to use. Default is ActiMode.AC_MODE_NONE.
- **groups** (*int*) – the number of groups in this convolution
- **use_bias** (*bool*) – whether the layer uses a bias vector. Default is True.
- **shared_op** (*Op*) – the layer whose parameters are shared with. Default is None.
- **kernel_initializer** (*Initializer*) – Initializer for the kernel weights matrix. If it is set to None, the GlorotUniformInitializer is applied.
- **bias_initializer** (*Initializer*) – Initializer for the bias vector. If it is set to None, the ZeroInitializer is applied.
- **name** (*string*) – the name of the layer. Default is None.

Returns Tensor – the output tensor.

9.2.2 Pool2D

class flexflow.core.flexflow_cffi.**FFModel**

pool2d(*input*, *kernel_h*, *kernel_w*, *stride_h*, *stride_w*, *padding_h*, *padding_w*, *pool_type=PoolType.POOL_MAX*, *activation=ActiMode.AC_MODE_NONE*, *name=None*)

Pooling operation for 2D spatial data.

The size of input tensor is (N, C_{in}, H, W) and the size of output tensor is $(N, C_{out}, H_{out}, W_{out})$, which can be calculated by:

$$C_{out} = out\_channels$$

$$K_H = kernel\_h$$

$$K_W = kernel\_w$$

$$S_H = stride\_h$$

$$S_W = stride\_w$$

$$P_H = padding\_h$$

$$P_S = padding\_s$$

$$H_{out} = (H - K_H + 2 * P_H)/S_H + 1$$

$$W_{out} = (W - K_W + 2 * P_W)/S_W + 1$$

Parameters

- **input** (*Tensor*) – the input Tensor.
- **kernel_h** (*int*) – the height of the 2D pooling window: K_H.
- **kernel_w** (*int*) – the width of the 2D pooling window: K_W.
- **stride_h** (*int*) – the stride of the pooling along the height: S_H.
- **stride_w** (*int*) – the stride of the pooling along the width: S_W.
- **padding_h** (*int*) – the amount of implicit zero-paddings along the height: P_H.

- **padding_w** (*int*) – the amount of implicit zero-paddings along the width: P_W.
- **activation** (*ActiMode*) – Tyoe of pooling function to use. If you don't specify anything, PoolType.POOL_MAX is applied.
- **activation** – Activation function to use. Default is ActiMode.AC_MODE_NONE.
- **name** (*string*) – the name of the layer. Default is None.

Returns Tensor – the output tensor.

9.2.3 Dense

class flexflow.core.flexflow_cffi.**FFModel**

dense(*input*, *out_dim*, *activation=ActiMode.AC_MODE_NONE*, *use_bias=True*, *datatype=DataType.DT_NONE*, *shared_op=None*, *kernel_initializer=None*, *bias_initializer=None*, *kernel_regularizer=None*, *name=None*)

Dense implements the operation: `output = activation(dot(input, kernel) + bias)` where `activation` is the element-wise activation function passed as the activation argument, `kernel` is a weights matrix created by the layer, and `bias` is a bias vector created by the layer (only applicable if `use_bias` is True).

The size of input tensor is (N, C_{in}) and the size of output tensor is (N, C_{out}), where $C_{out} = out\_dim$

Parameters

- **input** (*Tensor*) – the input Tensor.
- **out_dim** (*int*) – dimensionality of the output space.
- **activation** (*ActiMode*) – Activation function to use. Default is ActiMode.AC_MODE_NONE.
- **use_bias** (*bool*) – whether the layer uses a bias vector. Default is True.
- **shared_op** (*Op*) – the layer whose parameters are shared with. Default is None.
- **kernel_initializer** (*Initializer*) – Initializer for the kernel weights matrix. If it is set to None, the GlorotUniformInitializer is applied.
- **bias_initializer** (*Regularizer*) – Initializer for the bias vector. If it is set to None, the ZeroInitializer is applied.
- **kernel_regularizer** – Regularizer for the kernel weights matrix
- **name** (*string*) – the name of the layer. Default is None.

Returns Tensor – the output tensor.

9.2.4 Embedding

class flexflow.core.flexflow_cffi.**FFModel**

embedding(*input*, *num_embeddings*, *embedding_dim*, *aggr*, *dtype=DataType.DT_FLOAT*, *shared_op=None*, *kernel_initializer=None*, *name=None*)

Layer that turns positive integers into dense vectors of fixed size

Parameters

- **input** (*Tensor*) – the input Tensor.

- **num_embeddings** (*int*) – size of the vocabulary, i.e. maximum integer index + 1
- **embedding_dim** (*int*) – dimension of the dense embedding.
- **aggr** (*AggrMode*) – aggregation mode. Options are AGGR_MODE_NONE, AGGR_MODE_SUM and AGGR_MODE_AVG.
- **dtype** (*DataType*) – the tensor data type. Options are DT_BOOLEAN, DT_INT32, DT_INT64, DT_HALF, DT_FLOAT, DT_DOUBLE, DT_INT4, DT_INT8, DT_NONE
- **shared_op** (*Op*) – the layer whose parameters are shared with. Default is None.
- **kernel_initializer** (*Initializer*) – Initializer for the kernel weights matrix. If it is set to None, the GlorotUniformInitializer is applied.
- **name** (*string*) – the name of the layer. Default is None.

Returns Tensor – the output tensor.

9.2.5 Transpose

class flexflow.core.flexflow_cffi.**FFModel**

transpose(*input*, *perm*, *name=None*)

Transposes the `input` tensor. Permutes the dimensions according to perm

Parameters

- **input** (*Tensor*) – the input Tensor.
- **perm** (*List of int*) – A permutation of the dimensions of a.
- **name** (*string*) – the name of the layer. Default is None.

Returns Tensor – the output tensor.

9.2.6 Reverse

class flexflow.core.flexflow_cffi.**FFModel**

reverse(*input*, *axis*, *name=None*)

Layer that reverses specific dimensions of a tensor.

Given a `input` tensor, this operation reverses the dimension `axis`.

Parameters

- **input** (*Tensor*) – the input Tensor.
- **axis** (*int*) – the dimension to reverse.
- **name** (*string*) – the name of the layer. Default is None.

Returns Tensor – the output tensor.

9.2.7 Concatenate

class flexflow.core.flexflow_cffi.**FFModel**

concat(*tensors*, *axis*, *name=None*)

Layer that concatenates a list of inputs.

It takes as input a list of tensors, all of the same shape except for the concatenation axis, and returns a single tensor that is the concatenation of all inputs.

Parameters

- **input** (*List of Tensors*) – the list of input Tensors.
- **axis** (*int*) – the dimension along which to concatenate.
- **name** (*string*) – the name of the layer. Default is None.

Returns Tensor – the output tensor.

9.2.8 Split

class flexflow.core.flexflow_cffi.**FFModel**

split(*input*, *sizes*, *axis*, *name=None*)

Layer that splits a `input` tensor into a list of tensors.

Parameters

- **input** (*Tensor*) – the input Tensor.
- **sizes** (*int or list of int*) – either an int indicating the number of splits along axis or a Python list containing the sizes of each output tensor along axis. If a scalar, then it must evenly divide `input.dims[axis]`; otherwise the sum of sizes along the split axis must match that of the `input`.
- **axis** (*int*) – the dimension along which to split.
- **name** (*string*) – the name of the layer. Default is None.

Returns list of Tensors – the output tensors.

9.2.9 Reshape

class flexflow.core.flexflow_cffi.**FFModel**

reshape(*input*, *shape*, *name=None*)

Layer that reshapes inputs into the given shape.

Given a `input` tensor, this operation returns a output tensor that has the same values as tensor in the same order, except with a new shape given by `shape`.

Parameters

- **input** (*Tensor*) – the input Tensor.
- **shape** (*list of int*) – A list defining the shape of the output tensor.
- **name** (*string*) – the name of the layer. Default is None.

Returns Tensor – the output tensor.

9.2.10 Flat

class flexflow.core.flexflow_cffi.**FFModel**

flat(*input*, *name=None*)

Flattens the input. Does not affect the batch size.

Parameters

- **input** (*Tensor*) – the input Tensor.
- **name** (*string*) – the name of the layer. Default is None.

Returns Tensor – the output tensor.

9.2.11 BatchNorm

class flexflow.core.flexflow_cffi.**FFModel**

batch_norm(*input*, *relu=True*, *name=None*)

Layer that normalizes its inputs.

Batch normalization applies a transformation that maintains the mean output close to 0 and the output standard deviation close to 1.

Parameters

- **input** (*Tensor*) – the list of input Tensors.
- **relu** (*bool*) – whether a ReLU function is applied. Default is True.
- **name** (*string*) – the name of the layer. Default is None.

Returns Tensor – the output tensor.

9.2.12 BatchMatMul

class flexflow.core.flexflow_cffi.**FFModel**

batch_matmul(*A*, *B*, *a_seq_length_dim=None*, *b_seq_length_dim=None*, *name=None*)

Layer that applied batched matrix multiplication onto two input Tensors, `output = x * y`.

Parameters

- **A** (*Tensor*) – the first input Tensor.
- **B** (*Tensor*) – the second input Tensor.
- **a_seq_length_dim** (*int*) – an int when set indicating the a_seq_length_dim dimention of A is a sequence_length dimension
- **b_seq_length_dim** (*int*) – an int when set indicating the b_seq_length_dim dimention of B is a sequence_length dimension
- **name** (*string*) – the name of the layer. Default is None.

Returns Tensor – the output tensor.

9.2.13 Add

class flexflow.core.flexflow_cffi.**FFModel**

add(*x*, *y*, *inplace_a=False*, *name=None*)

Layer that adds two input Tensors, `output = x + y`.

Parameters

- **x** (*Tensor*) – the first input Tensor.
- **y** (*Tensor*) – the second input Tensor.
- **name** (*string*) – the name of the layer. Default is None.

Returns Tensor – the output tensor.

9.2.14 Subtract

class flexflow.core.flexflow_cffi.**FFModel**

subtract(*x*, *y*, *inplace_a=False*, *name=None*)

Layer that subtracts two input Tensors, `output = x * y`.

Parameters

- **x** (*Tensor*) – the first input Tensor.
- **y** (*Tensor*) – the second input Tensor.
- **name** (*string*) – the name of the layer. Default is None.

Returns Tensor – the output tensor.

9.2.15 Multiply

class flexflow.core.flexflow_cffi.**FFModel**

multiply(*x*, *y*, *inplace_a=False*, *name=None*)

Layer that multiplies (element-wise) two input Tensors, `output = x * y`.

Parameters

- **x** (*Tensor*) – the first input Tensor.
- **y** (*Tensor*) – the second input Tensor.
- **name** (*string*) – the name of the layer. Default is None.

Returns Tensor – the output tensor.

9.2.16 Divide

class flexflow.core.flexflow_cffi.**FFModel**

divide(*x*, *y*, *inplace_a=False*, *name=None*)

Layer that divides (element-wise) two input Tensors, `output = x / y`.

Parameters

- **x** (*Tensor*) – the first input Tensor.
- **y** (*Tensor*) – the second input Tensor.
- **name** (*string*) – the name of the layer. Default is None.

Returns Tensor – the output tensor.

9.2.17 Exponential

class flexflow.core.flexflow_cffi.**FFModel**

exp(*x*, *name=None*)

Exponential activation function.

Parameters

- **x** (*Tensor*) – the input Tensor.
- **name** (*string*) – the name of the layer. Default is None.

Returns Tensor – the output tensor.

9.2.18 ReLU

class flexflow.core.flexflow_cffi.**FFModel**

relu(*input*, *inplace=True*, *name=None*)

Rectified Linear Unit activation function.

Parameters

- **input** (*Tensor*) – the input Tensor.
- **name** (*string*) – the name of the layer. Default is None.

Returns Tensor – the output tensor.

9.2.19 ELU

class flexflow.core.flexflow_cffi.**FFModel**

elu(*input*, *inplace=True*, *name=None*)

Exponential Linear Unit. activation function.

Parameters

- **input** (*Tensor*) – the input Tensor.
- **name** (*string*) – the name of the layer. Default is None.

Returns Tensor – the output tensor.

9.2.20 Sigmoid

class flexflow.core.flexflow_cffi.**FFModel**

sigmoid(*input*, *name=None*)

Sigmoid activation function, $sigmoid(x) = 1/(1 + exp(-x))$.

Parameters

- **input** (*Tensor*) – the input Tensor.
- **name** (*string*) – the name of the layer. Default is None.

Returns Tensor – the output tensor.

9.2.21 Tanh

class flexflow.core.flexflow_cffi.**FFModel**

tanh(*input*, *name=None*)

Hyperbolic tangent activation function.

Parameters

- **input** (*Tensor*) – the input Tensor.
- **name** (*string*) – the name of the layer. Default is None.

Returns Tensor – the output tensor.

9.2.22 Softmax

class flexflow.core.flexflow_cffi.**FFModel**

softmax(*input*, *axis=- 1*, *name=None*)

Softmax activation function.

Parameters

- **input** (*Tensor*) – the input Tensor.
- **name** (*string*) – the name of the layer. Default is None.

Returns Tensor – the output tensor.

9.2.23 Dropout

class flexflow.core.flexflow_cffi.**FFModel**

dropout(*input*, *rate*, *seed*, *name=None*)

The Dropout layer randomly sets input units to 0 with a frequency of `rate` at each step during training time, which helps prevent overfitting. Inputs not set to 0 are scaled up by 1/(1 - rate) such that the sum over all inputs is unchanged.

Parameters

- **input** (*Tensor*) – the input Tensor.
- **rate** (*float(0-1)*) – Fraction of the input units to drop.
- **seed** (*int*) – random seed.
- **name** (*string*) – the name of the layer. Default is None.

Returns Tensor – the output tensor.

9.2.24 MultiheadAttention

class flexflow.core.flexflow_cffi.**FFModel**

multihead_attention(*query*, *key*, *value*, *embed_dim*, *num_heads*, *kdim=0*, *vdim=0*, *dropout=0.0*, *bias=True*, *add_bias_kv=False*, *add_zero_attn=False*, *kernel_initializer=None*, *name=None*)

Defines the MultiHead Attention operation as described in Attention Is All You Need which takes in the tensors `query`, `key`, and `value`, and returns the dot-product attention between them:.

Parameters

- **query** (*Tensor*) – the query Tensor.
- **key** (*Tensor*) – the key Tensor.
- **value** (*Tensor*) – the value Tensor.
- **embed_dim** (*int*) – total dimension of the model
- **num_heads** (*int*) – Number of attention heads.
- **kdim** (*int*) – total number of features in key. Default is 0
- **vdim** (*int*) – total number of features in value. Default is 0
- **dropout** (*float(0-1)*) – a Dropout layer on attn_output_weights. Default is 0.0
- **bias** (*bool*) – Whether the dense layers use bias vectors. Default is True.
- **add_bias_kv** (*bool*) – add bias to the key and value sequences at dim=0. Default is False.
- **add_zero_attn** (*bool*) – add a new batch of zeros to the key and value sequences at dim=1. Default is False.
- **kernel_initializer** (*Initializer*) – Initializer for dense layer kernels. If it is set to None, the GlorotUniformInitializer is applied.
- **name** (*string*) – the name of the layer. Default is None.

Returns Tensor – the output tensor.

9.3 Dataloader API

9.3.1 Dataloader Creation

class flexflow.core.flexflow_cffi.**FFModel**

create_data_loader(*batch_tensor*, *full_array*)

Create a SingleDataloader instance.

Parameters

- **batch_tensor** (*Tensor*) – a batch-sized tensor. Usually it is a input tensor of the model.
- **full_array** (*Numpy Array*) – the entire data.

Returns SingleDataloader – returns a dataloader instance.

9.3.2 Use Dataloader for Training

class flexflow.core.flexflow_cffi.**SingleDataLoader**

next_batch(*ffmodel*)

Ask the dataloder to load the next batch to the `batch_tensor`.

Returns None – no returns.

reset()

Reset the current position of the dataloder to 0.

Returns None – no returns.

CHAPTER

TEN

C++ API

The FlexFlow backend is at the core of FlexFlow Train and FlexFlow Serve. It is written entirely in C/C++ and CUDA/HIP. This section documents the API, which is generated by Doxygen and it is available at the following links:

- CUDA version (default version)
- HIP version

The two versions only differ when it comes to the GPU kernels, so the great majority of the entries are identical. If you are unsure which version to use, take a look at the CUDA version.

CHAPTER

ELEVEN

DEVELOPERS GUIDE

11.1 Code Organization

The bulk of the FlexFlow source code is stored in the following folders:

1. `examples`: example DNNs in C++ and Python
2. `include`: the FlexFlow headers
3. `src`: the FlexFlow source code
4. `python`: bindings for the Python interface

The `src` folder is divided into the following subfolders:

1. `loss_functions`: contains the implementation of all the supported loss functions, as well as the backward function to be used during training.
2. `mapper`: contains the implentation of the Legion custom mapper for FlexFlow, `FFMapper`.
3. `metric_functions`: contains the implementation of all the metrics functions, such as accuracy, categorical crossentropy, or mean squared error.
4. `ops`: contains the implementation of all tensor operators.
5. *parallel_ops*: contains the operators used to represent parallelization on the Parallel Computation Graph (PCG) as described in the [Unity paper](https://www.usenix.org/system/files/osdi22-unger.pdf).
6. *recompile*: support for the dynamic recompilation functionality described in [this paper](https://arxiv.org/pdf/2205.01848.pdf)
7. `runtime`: contains the implementation of the high-level FlexFlow runtime
8. `utils`: only contains implementation of the RecordFormatter class.

In many parts of the source code you will see triplets of files with the following three different extensions: `.cc`, `.cpp` and `.cu`. The `.cc` file contains the main, high-level C++ implementation, whereas the `.cpp` and `.cu` file contain, respectively, the HIP and CUDA kernels.

The best way to familiarize with the FlexFlow codebase is to walk through one of the existing examples, then check out the relevant FlexFlow runtime functions that are used in the example. We provide examples in both Python and C++. The Python interface is the most up-to-date, and the one that is intended to be used by users. To learn how to *run* a DNN in FlexFlow, please refer to the scripts in the examples/python folder. The C++ interface is intended mostly for development purposes and may have some rough edges. Nevertheless, the C++ examples are the preferred ones to look at if you want to familiarize with the internals of the FlexFlow implementation.

11.1.1 AlexNet example (C++)

In this section, we will walk through the AlexNet C++ implementation, which can be found in the examples/cpp/AlexNet folder of the repository. You can use this example as a template to write your own C++ DNN model using FlexFlow.

You can start by taking a look at the `alexnet.cc` file, containing the core of the implementation. You will notice the absence of a `main()` function. The FlexFlow C++ interface uses the `main()` function defined in src/runtime/cpp_driver.cc, so you will not need to create a new one when writing a FlexFlow program. Instead, you will use a function called `top_level_task` and with the following signature:

```
void FlexFlow::top_level_task(Task const *task,
                              std::vector<PhysicalRegion> const &regions,
                              Context ctx,
                              Runtime *runtime);
```

Inside the `top_level_task` function, you will want to create a FFModel object, which is usually initialized by passing a FFConfig object to the constructor:

```
FFConfig ffConfig;
FFModel ff(ffConfig);
```

`FFModel` is a very large class, and is the cornerstone of every FlexFlow DNN, providing the methods required to instantiate input tensors, add layers, compile the model, etc…

Tensor creation

The typical first step in a FlexFlow DNN is to define the input tensors. You can do that using the `FFModel.create_tensor` function. In the case of AlexNet:

```
Tensor input;
{
    int const dims[] = {ffConfig.batchSize, 3, 229, 229};
    input = ff.create_tensor<4>(dims, DT_FLOAT);
}
```

In the case of AlexNet, the input tensor has dimension `batch_size x 3 x 229 x 229`, so it is a 4-dimensional tensor. To initialize the tensor, we use the templated `create_tensor` function, which is part of `FFModel`. It may be useful to know that the `create_tensor` function lays out the tensor's dimensions in reverse order. For instance, in the snippet above, printing the `input` tensor (which can be done using the instruction below) will show dimensions: `[229, 229, 3, batch_size]`.

```
input->print("input tensor")
```

There are two versions of the `create_tensor` function: one (used in the last snippet above) uses a template that takes the number of tensor dimensions as its parameter; the second is a wrapper around the first, and takes the number of tensor dimensions as a regular function parameter. Both versions are implemented in `model.cc`, and their signature is identical, except for the number of dimensions parameter. Below, we discuss the implementation of the `create_tensor` wrapper, since it illustrates a common pattern among FlexFlow functions:

```
Tensor FFModel::create_tensor(int numdim,
                              int const dims[],
                              DataType data_type,
                              Layer const *layer,
                              int idx,
```

(continues on next page)

(continued from previous page)

```
                                bool create_grad) {
  switch (numdim) {
#define DIMFUNC(DIM)                                                           \
  case DIM:                                                                    \
    return create_tensor<DIM>(dims, data_type, layer, idx, create_grad);
    LEGION_FOREACH_N(DIMFUNC)
#undef DIMFUNC
    default:
      assert(false && "Unsupported dim!");
  }
}
```

The `LEGION_FOREACH_N(DIMFUNC)` macro is defined in deps/legion/runtime/legion/legion_config.h. The preprocessor replaces the block of code between `#define DIMFUNC(DIM)` and `#undef DIMFUNC` with a `case` statement for each integer between 1 and the `LEGION_MAX_DIM`, controlled by the `Legion_MAX_DIM` Legion CMake variable, which in case of FlexFlow, is set equal to `FF_MAX_DIM` in cmake/legion.cmake. For example, in the default case, where `FF_MAX_DIM` is set to 4, the preprocessor will rewrite the `switch` loop above as follows:

```
switch (numdim) {
    case 1:
        return create_tensor<1>(dims, data_type, layer, idx, create_grad);
    case 2:
        return create_tensor<2>(dims, data_type, layer, idx, create_grad);
    case 3:
        return create_tensor<3>(dims, data_type, layer, idx, create_grad);
    case 4:
        return create_tensor<4>(dims, data_type, layer, idx, create_grad);
    default:
        assert(false && "Unsupported dim!");
}
```

In addition to the two versions of `create_tensor` discussed above, `model.cc` also offers the `create_tensor_legion_ordering` function, which simply creates a tensor without reversing the order of the input dimensions. The explicit template instantiations at the bottom of `model.cc` will ensure that functions such `create_tensor` are only instantiated for number of dimensions that are less or equal to `FF_MAX_DIM`.

Adding layers to a DNN model

Going back to the AlexNet example, after defining the input tensors, we can add each of the DNN's layers by using the corresponding method from `FFModel`. For instance, the first layer is added using:

```
t = ff.conv2d(input, 64, 11, 11, 4, 4, 2, 2, AC_MODE_RELU);
```

The `conv2d` function is defined in src/ops/conv_2d.cc. Just like the other `FFModel` layer functions, it creates a new `Layer` object, populates with all relevant properties, and then enqueues to the list of layers in the `FFModel` class.

Optimizer and training metrics

After adding the DNN layers, the next step before compiling the model for training is to initialize an optimizer and then create a vector with all the metrics that you want to monitor at each training step.

Model compilation

Model compilation consists of the following steps:

1. We initialize an operator for each layer in the model, via the function `create_operators_from_layers()`. Layers work with `Tensor` input/weights/outputs, and are created directly by the user when writing a FlexFlow program. Operators work with `ParallelTensor` objects and they are responsible for running computations by launching kernels on GPUs.
2. Launch the graph optimize task (`GRAPH_OPTIMIZE_TASK_ID`), implemented by`PCG::Graph::graph_optimize_task`, which returns `PCG::GraphOptimalViewSerialized`
    1. call `deserialize_graph_optimal_view(...)` to get `PCG::Graph *best_graph` and `std::unordered_map<PCG::Node, MachineView> optimal_views` from deserialized `PCG::GraphOptimalViewSerialized`
    2. `convert_graph_to_operators()`
    3. print the dot of the best graph obtained
    4. map inputs to parallel tensor and weights to parallel tensor? -> strange for loop to understand better
3. Init performance metrics via the `FFModel::update_metrics_task`
4. Perform inplace optimizations (if enabled)
5. Loop through the operators to do the following (to be understood better):
    1. `parameters.push_back(op->weights[i]);` for each weight in each operator
    2. `op->map_output_tensors(*this);`
    3. `((ParallelOp *)op)->create_input_partition(*this);` if the operator is a parallel operator
6. Check correctness of the operator's input and output tensors' settings
7. Perform fusion optimizations, if enabled
8. Print all operators and their input and output regions
9. Create the tensor for the label
10. Initialize the optimizer
11. In training mode, if NCCL is enabled, initialize all the communicators and other objects

11.2 Continuous Integration

We currently implement CI testing using Github Workflows. Each workflow is defined by its corresponding YAML file in the .github/workflows folder of the repo. We currently have the following workflows:

1. `build.yml`: checks that the build & installation of FlexFlow succeed, using both the CMake and Makefile systems
2. `clang-format-check.yml`: ensures that the source code is properly formatted.

3. `docker-build.yml`: checks that the Docker containers can build and run FlexFlow properly. It also publishes a new version of the FlexFlow containers to the repo's package register for each push to the master branch
4. `gpu-ci.yml`: runs all the tests that require a GPU to run.
5. `gpu-ci-daemon.yml`: an helper workflow that turns on/off the GPU instance used by the test above
6. `multinode-test.yml`: runs the same GPU tests from the `gpu-ci.yml` workflow, but using multiple (simulated) nodes. The test currently simulates two nodes, each with 2 GPUs. To run FlexFlow on multiple nodes, we compile Legion with GASNET enabled, and choose MPI as the GASNET conduit. Compared to the single-node version, this test is much more time-consuming (about 4h instead 40mins at the time of writing), so we only run the test on the FlexFlow `master` branch every other day.
7. `pip-deploy.yml`: builds the `flexflow` pip package and publishes it on `TestPyPI` (if the repository environment variable `DEPLOY_TO_TEST_PYPI` is unset, or set to `false`) or `PyPI` (if `DEPLOY_TO_TEST_PYPI` is set to `true`). When deploying to `PyPI`, a new git tag (with the pip package version) will also be created, and associated with the commit hash that triggered the workflow. The `pip-deploy.yml` can only be launched manually via workflow dispatch. More on the pip packaging in the *section below*.
8. `pip-install.yml`: checks the build & installation of FlexFlow using `pip`
9. `shell-check.yml`: runs shellcheck on all bash scripts in the repo

We also have three placeholder workflows: `build-skip.yml`, `docker-build-skip.yml`, `gpu-ci-skip` and `pip-install-skip.yml`. These always pass and are used only in the case of skipped workflows whose status is required to merge a PR; we implement the "hack" officially recommended by Github (see here).

In the next section, we walk through an example workflow, similar to the ones found in this repo. An important thing to note is that Github workflows do not run unless they are properly linted. If you encounter a formatting/linting error, you can lint your workflow file using `prettier` (installation instructions here):

```
yarn prettier --write <filename.yml>
```

11.2.1 Github Workflow syntax

In this section, we will walk through an example workflow:

```yaml
name: "build"

on:
  pull_request:
    paths:
      - "src/**"
      - ".github/workflows/build.yml"
  push:
    paths:
      - "src/**"
      - ".github/workflows/build.yml"
    branches:
      - "master"
  schedule:
    # Run weekly on Saturday at midnight PT (3am ET / 8am UTC)
    - cron: "0 8 * * 6"
  workflow_dispatch:

concurrency:
```

(continues on next page)

(continued from previous page)

```
  group: build-${{ github.head_ref || github.run_id }}
  cancel-in-progress: true

jobs:
  cmake-build:
    name: Build FlexFlow with CMake
    runs-on: ubuntu-20.04
    steps:
      - name: Checkout Git Repository
        uses: actions/checkout@v3
        with:
          submodules: recursive

      - name: Install CUDA
        uses: Jimver/cuda-toolkit@v0.2.11
        id: cuda-toolkit
        with:
          cuda: "11.8.0"
          # Disable caching of the CUDA binaries, since it does not give us any␣
↪significant performance improvement
          use-github-cache: "false"

      - name: Install FlexFlow Dependencies
        run: .github/workflows/helpers/install_dependencies.sh
```

The first instruction in a workflow file sets the workflow's name. The name is not required to be unique, but it is preferrable to use unique names to avoid conflicts.

Next, the `on:` section allows you to control what events trigger a workflow run. A full list of events that can trigger a workflow run is available here. Each trigger can take options that further filter out the scenarios where the workflow runs. In the example above, we have the following triggers:

1. A `pull_request` trigger, triggering a workflow run when a PR is opened, and for each new commit to a branch associated with an open PR. The `paths` option allows you to choose which files in the repository need to be modified to make the workflow run. For instance, in the example, the `pull_request` trigger is only activated for PRs where either `.github/workflows/build.yml` or a file in the `src` folder is modified.
2. A `push` trigger, triggering a run for each push, no matter if there is an open PR or not. Here, in addition to the `paths` option, we have a `branches` option, restricting the trigger to activate only for commits to the `master` branch, but not for commits to other branches.
3. A `schedule` trigger, triggering the workflow at specific times. The syntax for chron workflows is explained here.
4. A `workflow_dispatch` trigger, enabling authorized users to manually run the workflow.

There are many additional options that are not discussed here. For example, there is a `paths-ignore` option that allows you to run the workflow in any case except if a file at the specified paths is modified.

Next, the `concurrency` section allows you to control how many copies of the same workflow can run in parallel. This is useful, for example, when one pushes a new commit to a branch before the workflows for the previous commits have finished running. Since the old commit is now obsolete, there is no need to wait until the old workflow has finished running before running again on the newer commit. In the example above, for example, we use the `concurrency` section to cancel any queued or in-progress workflow when a newer one is triggered.

Finally, we define the jobs that will run when the workflow is triggered. Each job is specified by adding an indented entry to the `jobs:` section, and will run in parallel in a isolated container. Multiple jobs in the same workflow do not directly share files. The `runs-on` option allows you to control what type of runner to use for the job. In the example,

we use `runs-on: ubuntu-20.04` to run the job on a VM with Ubuntu 20.04. You can also set up the workflow to run on a self-hosted machine by using the option `runs-on: self-hosted` and following the instructions at this link to connect the self hosted machine to the repository.

Each step in a job will be executed sequentially, and if it fails, the remaining steps will be cancelled and the job will be marked as `failed`. Each step is specified by either reusing a Github action or running a shell command (or a script file). For instance, in the example above, the first step uses the Github Action `actions/checkout@v3` to check out the repository, the second step uses the `Jimver/cuda-toolkit@v0.2.11` action to install CUDA, whereas the third step runs a bash script stored in the repo at the path `.github/workflows/helpers/install_dependencies.sh`.

11.3 Pip packages

This section illustrates how we support the automatic deployment of FlexFlow to the `PyPI` and `Test PyPI` repositories. Publishing FlexFlow on `PyPI` makes it possible for users to install FlexFlow on their machine by simply running:

```
pip install flexflow
```

To install from `Test PyPI`, on the other hand, one can use:

```
pip install flexflow --extra-index-url https://test.pypi.org/simple/
```

The installation process currently takes approximately the same time as installing from source by running the command `pip install .` from `FF_HOME` after having cloned the repo. However, installing directly from PyPI allows the user to automatically install the Python dependencies, and removes the step of having to manually clone the repo with all its submodules.

Below, we discuss some important properties of PyPI.

11.3.1 Packaging

When building a `pip` package from a repository, we can decide what files from the repository will be included in the package, and which ones will be left out. To do that, we write a MANIFEST.in file, according to the syntax from the official instructions. In particular, we manually include the submodules (which would otherwise be left out by default), we remove the `.git` folders, which are not needed to build FlexFlow, as well as the `triton` folder, whose contents are not currently in use. Finally, we request that the version.txt file, whose role is described in the section below, is included in the package distribution. Because this file is generated at build time, it would be left out by default if we didn't manually include it.

11.3.2 Source VS Wheel distribution

PyPI allows you to upload a source distribution, together with one (or more) binary distributions of your package. A `pip` package's pre-compiled binary is called a Wheel (formerly, Egg). The advantage of publishing Wheel distributions instead of just the source code is that the installation of the package will be much faster for the user, who will just need to download the binary, and extract its files in the proper locations (all of this is handled automatically when running `pip install <package name>`). If only the source code is available, on the other hand, `pip install <package name>` will first need to compile the package, and then install it.

`PyPI` allows you to upload multiple Wheels to support different Python versions (the Wheel compatible with version of Python installed on the user's machine is downloaded automatically when the user runs `pip install <package name>`), but unfortunately does not yet support uploading a Wheel for each CUDA version, and automatically downloading the relevant one depending on the user's machine configuration. Instead, one needs to upload a Wheel with a distinct name for each CUDA version, and the user will need to specify the name manually at dowload time. For this

reason, to keep things simple, we only publish the source distribution at this moment, and plan to upload Wheels that are specific to each Python version and CUDA version at a later time.

11.3.3 Versioning

PyPI imposes some strict versioning requirements. Among other things, versions need to follow a specific format, and once a given version of a package is published, it can never be replaced. In addition, even if the publisher deletes a version, nobody can never upload a source distribution / Wheel with that same version number again. Finally, when multiple versions of the same package are published, the one with the highest version number (not the one that was uploaded last) will be installed by default.

When publishing a package on PyPI, the version attached to the upload is determined by the `setup.py` script. You can check which version string will be used by running `python setup.py --version`.

The simplest way to version a `pip`package is to hard-code the version number in the `setup.py` script, and committing a change to the repository every time the `pip` package is to be updated. This approach, however, is incompatible with having a script or workflow to automatically update the `pip` package.

If we intend to deploy the latest code to PyPI automatically, we need a way to automatically assign a properly-formatted version string to the code we want to upload. Further, we need to ensure that the assigned version is (1) different from any version (of the same package) already published on PyPI and (2) larger than any previous version. Finally, a trickier requirement is that: (3) at any point in time, the `setup.py` script included in a given version of our package should output a version string that exactly matches the version string recorded in the metadata attached to the package's version at publication time. More about this below.

We follow a simple approach to automatically version the latest code: use the publication's date to generate the version string. For example, on Aug 12, 2023, we can use version string 23.08.12. Assuming that we publish at most one version per day, and that we always publish from the same timezone, we will be able to meet requirements (1) and (2). An additional enhancement to be able to support the update of the package more than once per day (which may be needed in development phase, or if a mistake is made), instead of using the day of the month (12 for August 12, 2023) for the sub-sub-version, we use an index that starts at 0 every month, and is incremented by +1 every time we upload a new version of the package within the same calendar month. So if on Aug 12, 2023 we are updating the package for the first time in the month, we will use version string 23.08.0; if later the same day (or any time before Sept 1, 2023) we wish to upload a new version, we will use string 23.08.1, and so forth.

Having illustrated the general versioning policy, we will need to implement it carefully in `setup.py` to ensure that we meet requirement (3). You can take a look at the `compute_version()` function to see how this is done in practice. The key realization is that we cannot simply compute today's date (using any of the Python libraries that let us do that) and transform it into a string, nor simply get from PyPI the latest available version of our package, and, if it was published on the same calendar month, increment the sub-subversion by +1 to generate the version string of the new upload. We can best illustrate why we cannot do that with an example:

- Today, Aug 12, 2023, we wish to upload a new version to PyPI. As we said above, the version string is computed by `setup.py`. A naive way to do so in `setup.py` would be to compute the date using `date.today()`, and transform the year and month into digit form to generate the version (23) and sub-version (08) parts of the version string. We could then check on PyPI what was the latest published version of the package as of today, and if we found that it was, say, 23.08.05, we would use 5+1=6 as the sub-sub-version for the new upload (so the final version string would be 23.08.06).
- Over the next few days, we upload 3 more versions
- A week later, on Aug 18, 2023, a user trying to install our package, runs `pip install <package name>`. To determine which version it should install, the `pip install` script downloads the most recent X versions of `<package name>` on the user's machine, and, for each version, re-computes the version string by running `python setup.py --version`. When the script attempts to recompute the version string on the package 23.08.06 (which we uploaded on Aug 12, 2023), it will reconstruct the version string by replaying the same instructions that were run on Aug. 12, and obtain a different version string, as follows. Using the current date,

the user will obtain: version=23, sub-version=08, which match the metadata. Checking the latest version of the package available on PyPI, the script finds version 23.08.09 (there were three more submissions since Aug 12). This will translate to sub-sub-version=9+1=10. Noticing that the version included in the Aug 12 package's metadata (23.08.06) does not match the recomputed version (23.08.10), the script will generate unexpected and undesired behavior.

To prevent accidentally breaking requirement (3) as illustrated in the scenario from the example above, we employ a simple hack: when computing our package's version string for the first time by running `setup.py`, we save the string to a file, `python/flexflow/version.txt`, which is added to the `.gitignore` and as such, never committed to the repo. As long as the `version.txt` exists, any subsequent run of `setup.py` will simply read the file, and output the same version string, no matter on which day and/or how many new versions of the package have been uploaded to PyPI since then. When packaging our code to upload it on PyPI, we ensure to delete the `version.txt` file, compute the version string, and then include the `version.txt` in the source distribution that we upload to `PyPI`. In this way, when the user attempts to install the package, `pip install` will download the most recent available versions, run `setup.py` from each distribution, and for each distribution, `setup.py` will always output the correct version string, because it will just read the string recorded in that distribution's `version.txt`.

11.3.4 Test PyPI

Given all the complexities and restrictions of PyPI, Test PyPI was created as a "copy" of PyPI to be used for testing and for being able to make mistakes without affecting the user, or forever losing the opportunity to use a given package name and/or version. We take advantage of Test PyPI as follows. If we intend to deploy to PyPI, we can first deploy to Test PyPI, check the results, fix any issue, and only later deploy to PyPI. All our `pip` related scripts in the repo have been designed to support both Test PyPI and PyPI. In order to let `setup.py` know that it should package a distribution for Test PyPI, one can simply export the following environment variable:

```
export DEPLOY_TO_TEST_PYPI=true
```

Conversely, to upload to PyPI, one can either leave `DEPLOY_TO_TEST_PYPI` unset, or export

```
export DEPLOY_TO_TEST_PYPI=false
```

WARNING!!! More likely than not, the latest version of the `flexflow` package on Test PyPI and PyPI will be out of sync. This is to be expected, because one may need to upload a few drafts on Test PyPI to detect and correct some bugs, before publishing the definitive version on PyPI. Having different latest versions on the two repositories should not cause any issue. However, after uploading to Test PyPI and before uploading to PyPI (or viceversa), **it is EXTREMELY IMPORTANT** to delete the `python/flexflow/version.txt` file.

An easy way to avoid forgetting this, is to only deploy on Test PyPI/PyPI using the `pip-deploy.yml`, which is designed to only upload to one of the two repositories at a given time.

11.3.5 Build vs install dependencies

FlexFlow requires some other Python packages in order to run. In addition, even building FlexFlow requires some packages, and you cannot run `setup.py` without those build requirements. There is a way for us to specify these *install* and *build* requirements in such a way that `pip` will detect if they are missing, and install them. We record the build requirements in the `pyproject.toml` file, whereas we specify the installation requirements by passing a list with each package's name to the `install_requires` key of the `setup()` function in `setup.py`. The installation requirements are automatically read from the `requirements.txt` file.

11.4 Contributing to FlexFlow

We want to make contributing to this project as easy and transparent as possible.

11.4.1 Formatting

We use `clang-format` to format our C++ code. If you make changes to the code and the Clang format CI test is failing, you can lint your code by running: `./scripts/format.sh` from the main folder of this repo.

11.4.2 Documenting the code

We follow the Python Docstring conventions for documenting the Python code. We document the C++ code using comments in any of the conventioned supported by Doxygen see here.

11.4.3 Pull Requests

We actively welcome your pull requests.

1. Fork the repo and create your branch from `master`.
2. If you've added code that should be tested, add tests.
3. If you've changed APIs, update the documentation.
4. Ensure the test suite passes.
5. Make sure your code lints.

11.4.4 Issues

We use GitHub issues to track public bugs. Please ensure your description is clear and has sufficient instructions to be able to reproduce the issue.

11.4.5 License

By contributing to FlexFlow, you agree that your contributions will be licensed under the LICENSE file in the root directory of this source tree.