Reference: https://github.com/ai-dynamo/dynamo/blob/main/container/Dockerfile.vllm

Base docker image:

nvcr.io/nvidia/cuda-dl-base:25.03-cuda12.8-devel-ubuntu24.04

Note: all the python-related installs are in uv virtual environments

Table of contents:

NIXL installation

Step 0: check NVLink speed

Install tmux in your Docker container

```
apt update && apt install -y tmux
```

In one terminal

```
CUDA_VISIBLE_DEVICES=0 UCX_TLS=cuda_ipc,cuda_copy,tcp ucx_perftest
-t tag_bw -m cuda -s 10000000 -n 10 -p 9999 -c 0
```

In another terminal

```
CUDA_VISIBLE_DEVICES=1 UCX_TLS=cuda_ipc,cuda_copy,tcp ucx_perftest
`hostname` -t tag_bw -m cuda -s 100000000 -n 10 -p 9999 -c 1
```

Step 1: install build essentials

```
apt-get update -y && \
```

```
    apt-get -y install \
    ninja-build \
    pybind11-dev \
    python3.12-dev \
    cmake
```

Step 2: install uv

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

You may need to log out and log in again after uv installation

Step 3: export env vars in ~/.bashrc

```
export
LD_LIBRARY_PATH=/usr/local/cuda/compat/lib.real:$LD_LIBRARY_PATH
export NIXL_PLUGIN_DIR=/usr/local/nixl/lib/x86_64-linux-gnu/plugins
```

Step 4: build nixl wheel under uv venv

Clone the repo and create venv

```
cd ~
git clone https://github.com/ai-dynamo/nixl
cd nixl
UV_PYTHON=python3.12 uv venv
source .venv/bin/activate
```

In folder nixl/, build the nixl plugins

```
uv pip install meson
cd ~/nixl/

rm -rf build && \
    mkdir build && \
    uv run meson setup build/ --prefix=/usr/local/nixl && \
    cd build && \
    ninja && \
    ninja install

echo "/usr/local/nixl/lib/x86_64-linux-gnu" >
/etc/ld.so.conf.d/nixl.conf
```

```
echo "/usr/local/nixl/lib/x86_64-linux-gnu/plugins" >>
/etc/ld.so.conf.d/nixl.conf
ldconfig
```

Build nixl wheel

```
cd ~/nixl/
uv build --wheel --out-dir /tmp/dist
```

There should be a wheel named nixl-0.1.1-cp312-cp312-linux_x86_64.whl in /tmp/dist

Step 5: Use the built wheel

In a virtual environment, run

```
uv pip install /tmp/dist/nixl-0.1.1-cp312-cp312-linux_x86_64.whl
```

Step 6: Verify the installation with a benchmarking script

Install dependencies

```
uv pip install torch pyzmq
```

Save the following script to benchmark.py

```
import torch
import time
import argparse
import zmq


from nixl._api import nixl_agent

zmq_socket: zmq.Socket = None


def parse_args():
    parser = argparse.ArgumentParser(description="Benchmark Nixl")
    parser.add_argument("--device", type=str, default="cuda",
help="Device to run the benchmark on")
    parser.add_argument("--num-blocks", type=int, default=100,
help="Number of blocks to create")
```

```
    parser.add_argument("--num-layers", type=int, default=32,
help="Number of layers in each block")
    parser.add_argument("--block-size", type=int, default=256,
help="Size of each block")
    parser.add_argument("--hidden-dim", type=int, default=1024,
help="Hidden dimension of each block")
    parser.add_argument("--dtype", type=str, default="bfloat16",
help="Data type of the blocks")

    parser.add_argument("--role", type=str, required=True,
help="Role of the agent ('creator' or 'peer')")
    parser.add_argument("--operation", type=str, required=True,
help="Operation to perform ('READ' or 'WRITE')")

    parser.add_argument("--host", type=str, default="localhost",
help="Host for ZMQ socket")
    parser.add_argument("--port", type=int, default=5555, help="Port
for ZMQ socket")
    return parser.parse_args()

def init_zmq_socket(host, port, role):
    """
    Initialize the ZMQ socket for communication.
    """
    global zmq_socket
    context = zmq.Context()
    zmq_socket = context.socket(zmq.PAIR)
    if role == "peer":
        zmq_socket.bind(f"tcp://{host}:{port}")
    else:
        zmq_socket.connect(f"tcp://{host}:{port}")
        # Ensure the socket is ready to receive messages
        zmq_socket.setsockopt(zmq.LINGER, 0)
        #zmq_socket.setsockopt(zmq.RCVTIMEO, 1000)

    print("Finished initializing ZMQ socket for role:", role)

def create_dataset(role,
                   device,
                   num_blocks = 100,
                   num_layers = 32,
```

```
                    block_size = 256,
                    hidden_dim = 1024,
                    dtype = torch.bfloat16):
    """
    Create a dataset of random tensors.
    """
    block_shape = (num_layers, 2, block_size, hidden_dim)
    dataset = []
    value = 0 if role == "peer" else 1
    for _ in range(num_blocks):
        block = torch.full(block_shape, value, device=device,
dtype=dtype)
        dataset.append(block)
    return dataset


def create_nixl_agents(role: str, tensors: list[torch.Tensor]):
    """
    Create Nixl agents based on the role.
    """
    agent = nixl_agent(role)
    register_descs = agent.register_memory(tensors)

    local_meta = agent.get_agent_metadata()

    if role == "creator":
        zmq_socket.send(local_meta)
        remote_meta = zmq_socket.recv()
        peer_name =
agent.add_remote_agent(remote_meta).decode("utf-8")
        print("Peer name:", peer_name)
        assert peer_name == "peer", "Peer name mismatch for
role=creator"
    elif role == "peer":
        remote_meta = zmq_socket.recv()
        peer_name =
agent.add_remote_agent(remote_meta).decode("utf-8")
        print("Peer name:", peer_name)
        zmq_socket.send(local_meta)
        assert peer_name == "creator", "Peer name mismatch for
role=peer"
```

```
    return agent, peer_name, register_descs

def initialize_xfer_metadata(
        role: str,
        operation: str,
        agent: nixl_agent,
        peer_name: str,
        register_descs,
    ):
    """
    Initialize transfer metadata.
    """
    local_xfer_descs = register_descs.trim()
    remote_xfer_descs = None
    transfer_handle = None

    if role == "peer":
        # Wait until there is a message from the creator
        msg = zmq_socket.recv().decode("utf-8")
        if msg == "START":
            print("Peer received START message")
        else:
            print("Peer received unexpected message:", msg)
            zmq_socket.close()
            exit(0)

        # send the xfer descs to the peer
        print("Peer sending xfer descs to creator")

zmq_socket.send(agent.get_serialized_descs(local_xfer_descs))

    elif role == "creator":
        zmq_socket.send("START".encode("utf-8"))
        print("Creator sent START message to peer")

        # Wait until there is a message from the peer
        msg = zmq_socket.recv()
        remote_xfer_descs = agent.deserialize_descs(msg)

        print("Creator received xfer descs from peer")
```

```
        transfer_handle = agent.initialize_xfer(
                operation,
                local_xfer_descs,
                remote_xfer_descs,
                peer_name,
                "FINISHED")

    return transfer_handle

def start_transfer(
        role: str,
        agent: nixl_agent,
        transfer_handle,
        peer_name,
    ):
    print("Starting transfer!")
    if role == "creator":
        state = agent.transfer(transfer_handle)
        if state == "ERR":
            print("Error in transfer")
        while True:
            state = agent.check_xfer_state(transfer_handle)
            if state == "DONE":
                print("Transfer finished in creator")
                break
            elif state == "ERR":
                print("Error in transfer")
                break
    else:
        while not agent.check_remote_xfer_done(peer_name,
b"FINISHED"):
            continue
        print("Transfer finished in peer")



def cleanup_transfer(
        agent: nixl_agent,
        transfer_handle,
        register_descs,
```

```
    ):
    # Cleanup the transfer handle and registered descriptors
    if transfer_handle is not None:
        agent.release_xfer_handle(transfer_handle)
    agent.deregister_memory(register_descs)

def cleanup_agent(
        agent: nixl_agent,
    ):
    # Cleanup the agent
    agent.remove_remote_agent(peer_name)

if __name__ == "__main__":
    args = parse_args()
    device = torch.device(args.device)

    # Initialize ZMQ socket
    init_zmq_socket(args.host, args.port, args.role)

    # Create dataset
    dataset = create_dataset(args.role, device,
                             num_blocks=args.num_blocks,
                             num_layers=args.num_layers,
                             block_size=args.block_size,
                             hidden_dim=args.hidden_dim,
                             dtype=getattr(torch, args.dtype))

    # Create Nixl agents
    agent, peer_name, register_descs = create_nixl_agents(args.role,
dataset)

    # Initialize transfer metadata
    start = time.perf_counter()
    transfer_handle = initialize_xfer_metadata(
        args.role,
        args.operation,
        agent,
        peer_name,
        register_descs
    )
    end = time.perf_counter()
```

```
    print(f"\033[32mTime to initialize transfer metadata: {end -
start:.2f} seconds\033[0m")

    # Start transfer
    start = time.perf_counter()
    start_transfer(
        args.role,
        agent,
        transfer_handle,
        peer_name,
    )
    end = time.perf_counter()
    print(f"\033[32mTime to start transfer: {end - start:.2f}
seconds\033[0m")
    print(f"\033[32mTransfer speed: {dataset[0].numel() *
dataset[0].element_size() * len(dataset) / (end - start) / 1e9:.2f}
GB/s\033[0m")

    # Check the result
    if args.role == "peer":
        for i, block in enumerate(dataset):
            assert torch.abs(torch.mean(block) - 1) < 1e-8, f"Block
{i} not equal to 1"
        print("Passed correctness check!")

    # Clean up transfer
    start = time.perf_counter()
    cleanup_transfer(
        agent,
        transfer_handle,
        register_descs,
    )
    end = time.perf_counter()
    print(f"\033[32mTime to cleanup transfer: {end - start:.2f}
seconds\033[0m")

    # Clean up agent
    cleanup_agent(agent)

    # Close ZMQ socket
    zmq_socket.close()
```

Run the script with following commands:

```
# In terminal 1:
UCX_TLS=cuda_ipc,cuda_copy,tcp python3 benchmark.py --role creator
--operation WRITE --device cuda:0
# In terminal 2:
UCX_TLS=cuda_ipc,cuda_copy,tcp python3 benchmark.py --role peer
--operation WRITE --device cuda:1
```

Should see something like the following screenshot in terminal 2

Install LMCache and vLLM

Note: Make sure you are doing this in venv

Step 1: checkout vLLM code with LMCache connector and install

```
git clone https://github.com/vllm-project/vllm
cd vllm
git remote add aposta https://github.com/ApostaC/vllm
git fetch aposta local-dev/lmcache-v1-connector-pr
git checkout local-dev/lmcache-v1-connector-pr
VLLM_USE_PRECOMPILED=1 uv pip install --editable . -v
```

Step 2: install LMCache

```
git clone https://github.com/LMCache/LMCache
cd LMCache/
uv pip install -e . -v
```

Run DP with LMCache + vLLM

In vLLM's root directory:

Terminal 1: start the prefiller instance

```
cd examples/other/LMCache && bash disagg-example.sh prefiller
```

Terminal 2: start the decoder instance

```
cd examples/other/LMCache && bash disagg-example.sh decoder
```

Terminal 3: run the proxy

```
cd examples/other/LMCache && bash disagg-example.sh proxy
```

Terminal 4: start the benchmark

```
uv pip install pandas datasets
cd benchmarks/
python3 benchmark_serving.py --port 9000 --seed $(date +%s) \
        --model meta-llama/Llama-3.1-8B-Instruct \
        --dataset-name random --random-input-len 7500
--random-output-len 200 \
        --num-prompts 200 --burstiness 100 --request-rate 3.6
```