# A. Artifact Appendix

## A.1 Abstract

This appendix describes how to reproduce the results described in the paper 'The CoRa Tensor Compiler: Compilation for Ragged Tensors With Minimal Padding'. The experiments in the paper evaluate CoRa on an Nvidia V100 GPU, an 8-core, 16-thread Intel CascadeLake CPU, an 8-core ARM Neoverse N1 CPU and a 64-core ARM Neoverse N1 CPU. Below, we provide instructions to set up and execute CoRa as well as other frameworks used in the evaluation on each of these platforms. As we start with publicly available Docker containers in all cases, one a few GPU-related dependencies (such as cuDNN) as well as CoRa's dependencies (such as the Z3 SMT solver, LLVM and OpenBLAS) need to be installed. We provide instructions for each of these below. In each case, 100 GB of disk space should be more than enough for the evaluation.

## A.2 Artifact check-list (meta-information)

- **Compilation:** We reply on the publicly available compilers `nvcc`, the CUDA compiler to compile CUDA code generated by CoRa, `gcc` to build CoRa and other dependencies and LLVM to facilitate code generation for CPUs.
- **Data set:** We use sequence lengths for 8 different commonly used NLP datasets, all of which are included in the repositories described below.
- **Run-time environment:** The artifact has been tested on Ubuntu 20.04 and with the following versions of different dependencies.
- **Hardware:** We use an Nvidia V100 GPU for the GPU evaluation, and an 8-core, 16-thread Intel CascadeLake CPU, an 8-core ARM Neoverse N1 CPU and a 64-core ARM Neoverse N1 CPU for evaluation CoRa on CPUs.
- **Metrics:** We use execution time as the primary execution metric of evaluation.
- **Output:** We generate CSV files, and also provide Python scripts to generate plots from the raw data.
- **Experiments:** Python/bash scripts are provided to replicate the results.
- **Disk space required:** 100 GB on each backend.
- **Time needed to prepare workflow:** A few hours on each backend.
- **Time needed to complete experiments:** Running all experiments on a backend takes several hours. The experiments on the ARM CPUs are particularly slow, taking 2-3 days to complete.
- **Publicly availability:** Yes, in the form of GitHub repositories (linked later) as well as on Zenodo at the url `https://doi.org/10.5281/zenodo.6326455`.

## A.3 Description

### A.3.1 Software dependencies

**GPU:** Below, we describe the environmrnt we use across the different hardware backends we evaluate CoRa on. On all of the backends, we use Ubuntu 20.04. Some of the frameworks below are already installed as part of the Docker images we start with (described below), while some need to be manually or installed. This is described below in §A.4.
**Dependencies Common across Backends:** CoRa requires the following frameworks on all platforms: Z3 4.8.8, LLVM 9.0.0, cmake $\geq$ 3.5 and g++ $\geq$ 5.0.
**Nvidia GPU:** CUDA 11.1 (V11.1.105), cuDNN 8.2.1, PyTorch 1.9.0+cu111, FasterTransformer (modified on top of FasterTransformer v4.0 (commit `dd4c071`) and provided as part of the `cora_benchmarks` repository). Make sure that nvcc is on the `PATH`.
**ARM CPUs:** OpenBLAS 0.3.10, PyTorch 1.10.0 with ARM Compute Library 21.12, TensorFlow 2.6.0 with ARM Compute Library 21.09
**Intel CPU:** Intel oneAPI MKL v2021.3

### A.3.2 Data sets

All datasets are included in the `cora_benchmarks` repository.

## A.4 Installation

Below, we provide installation instructions for setting up the environment as described above in §A.3.1 and building CoRa on all the hardware backends.

### A.4.1 Nvidia GPU:

**Setting up the environment:**

1. Start from docker container: `nvcr.io/nvidia/tensorflow:20.12-tf1-py3`.
2. Install Z3 4.8.8 available at `https://github.com/Z3Prover/z3/releases`. Download the file `z3-4.8.8-x64-ubuntu-16.04.zip` and move the contents of `lib/` and `include/` in this download to `/usr/lib` and `/usr/include` accordingly.
3. Install cuDNN 8.2.1 (for CUDA 11.x) by using the tar file installation method described on `https://docs.nvidia.com/deeplearning/cudnn/install-guide/index.html\#installlinux-tar`.
4. Install libtinfo-dev and libxml2-dev by running `apt install libtinfo-dev libxml2-dev`.
5. Download pre-built LLVM 9.0.0 for Ubuntu 19.04 from the LLVM download page (`https://releases.llvm.org/download.html`). Let the path to the root of the downloaded LLVM distribution be `LLVM_PATH`.

**Building CoRa:**

1. Clone the CoRa repo as follows:

```
git clone --recurse-submodules -j8
        https://github.com/pratikfegade/cora
```

Make sure you're on the `ragged_tensors` branch Let the root of CoRAbe `CORA_ROOT`.

2. In the file `cora/config.cmake.gpu`, set the variable `USE_LLVM` to `<LLVM_PATH>/bin/llvm-config`.
3. Build CoRAlike so:
```
mkdir build
cp config.cmake.gpu build/config.cmake
cd build
cmake ..
make -j8 tvm
```
4. Finally, the compiled shared library libtvm.so should be present in the build directory
5. Make sure that your `PYTHONPATH` contains `${CORA_ROOT}/python` before moving on to the experiments

**Installing PyTorch:**
Install PyTorch 1.9.0+cu111 like so:
```
pip3 install torch==1.9.0+cu111 \\
    torchvision==0.10.0+cu111 \\
    torchaudio==0.9.0 \\
    -f https://download.pytorch.org/whl/torch_stable.html
```
**Building FasterTransformer:**

1. Clone the `cora_benchmarks` repository available at `https://github.com/pratikfegade/cora_benchmarks`.
2. Build FasterTransformer like so:
```
cd cora_benchmarks/bert_layer/faster_transformer
mkdir build
cd build
cmake -DSM=70 -DCMAKE_BUILD_TYPE=Release ..
make -j8
```

### A.4.2 ARM CPUs

**Setting up the environment for PyTorch and CoRA:**

1. Start from docker container `armswdev/pytorch-arm-neoverse-n1:r21.12-torch-1.10.0-onednn-acl`
2. Download source for Z3 4.8.0 from `https://github.com/Z3Prover/z3/releases`. Build and install Z3 as follows:
```
cd z3-4.8.0.0
mkdir build
cd build
cmake -G "Unix Makefiles" ../
sudo make -j8 install
```
3. Build and install OpenBLAS as follows:
```
git clone https://github.com/xianyi/OpenBLAS
git checkout \\
    ee823b6ed98b2c9f8d72d9cdac195c64a6386a92
make TARGET=NEOVERSEN1 -j8
sudo make PREFIX=/usr/local/ install
```
4. Download pre-built LLVM 9.0.0 for Aarch64 from the LLVM download page (https://releases.llvm.org/download.html). Let the path to the root of the downloaded LLVM distribution be `LLVM_PATH`.
5. Install libtinfo-dev, libtinfo5-dev and libxml2-dev by running `apt install libtinfo-dev libxml2-dev libtinfo-dev`.

**Building CoRA:**
Just as on the GPU backend, clone the CoRA repository. Checkout the `arm` branch. Run the following commands in the root of the repository. Ensure that the variable `USE_LLVM` points to the `<LLVM_PATH>/bin/llvm-config` before executing the commands.
```
mkdir build
cp config.cmake.arm build/config.cmake
cd build
cmake ..
make -j8 tvm
```
Make sure that your `PYTHONPATH` contains `${CORA_ROOT}/python` before moving on to the experiments.
**TensorFlow**: We use the publicly available container `armswdev/tensorflow-arm-neoverse-n1:r21.09-tf-2.6.0-onednn-acl` in order to exexcute TensorFlow on the ARM machines. No additional environment set up needs to be done for this.

### A.4.3 Intel CPU

**Setting up the environment:**

1. Start from docker container `ubuntu:20.04`
2. Install Z3 4.8.8 available at `https://github.com/Z3Prover/z3/releases`. Download the file `z3-4.8.8-x64-ubuntu-16.04.zip` and move the contents of `lib/` and `include/` in this download to `/usr/lib` and `/usr/include` accordingly.
3. Install Intel oneAPI Math Kernel Library from `https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html`. The exact version (v2021.3) we used for our evaluation no longer seems to be available for download. Therefore the performance results for MKL runs may differ.
4. Install libtinfo-dev and libxml2-dev by running `apt install libtinfo-dev libxml2-dev`.
5. Download pre-built LLVM 9.0.0 for Ubuntu 19.04 from the LLVM download page (`https://releases.llvm.org/download.html`). Let the path to the root of the downloaded LLVM distribution be `LLVM_PATH`.
6. Install libtinfo-dev, libtinfo5-dev and libxml2-dev by running `apt install libtinfo-dev libxml2-dev libtinfo-dev`.

**Building CoRA:**
Just as on the GPU backend, clone the CoRA repository. Checkout the `arm` branch. Run the following commands in the root of the repository. Ensure that the variable `USE_LLVM`

points to the <LLVM_PATH>/bin/llvm-config before executing the commands.

```
mkdir build
cp config.cmake.arm build/config.cmake
cd build
cmake ..
make -j8 tvm
```

Make sure that your PYTHONPATH contains ${CORA_ROOT}/python before moving on to the experiments.

### A.5 Experiment workflow

CoRA is a tensor compiler. It takes an input a description of a tensor operator and generates LLVM or CUDA code for the kernel depending on whether the target is a CPU or a GPU. For evalauting individual kernels, such as trmm, vgemm or any of the 9 kernels that make up the transformer layer (illustrated in Figure3 of the paper), merely executing the python script implementing the kernel will compile the kernel, generate the code and execute it. For evalauting the entire transformer layer, or parts of it (such the self-attention or the MHA modules), we separate the steps of code generation and execution. We first generate compiled code for each of the operators in the form of shared libraries, which are then loaded and executed to form the layer to benchmark layer performance. The scripts provided automate all of these steps.

### A.6 General Evaluation Notes and Instructions

We provide instructions to replicate the results in the paper on the three hardware backends below in §A.7, §A.7.6 and §A.7.7. We provide scripts to generate plots from the raw data when applicable, and a draft of the paper (cora_benchmarks/expected_results_plotting_script/cora_paper_draft.pdf) as a way to provide expected results unless otherwise mentioned.

1. All paths below are relative to the root of the cora_benchmarks repository
2. The section, figure and table numbers in the titles refer to the section, table and figure numbers in the provided draft of the paper.
3. The starred measurements are primary performance measurements, whie the rest cvan be thought of as analysis of the primary performance measurements.

The script cora_benchmarks/expected_results_plotting_script/make_plots.py can be used to generate plots from the raw data when applicable. It can be executed as follows:

```
python3 make_plots.py --csv-dir <CSV_DIR> \\
                      --fig-dir <FIG_DIR>
```

where CSV_DIR should to the path to a folder that contains all the csv files generated from all the evalaution across the three backends. The script will generate the plots in FIG_DIR.

### A.7 Nvidia GPU Evaluation

Checkout the gpu branch in the cora_benchmarks repository.

#### A.7.1 Section 7.1: Matrix Multiplication

**Variable-Sized Batched GEMM (vgemm)*:**

1. Compile vbatch_gemm/cbt, vbatch_gemm/gemm_cublas by running make in those directories
2. Run
   ```
   python3 scripts/vbatch_gemm_eval.py \\
           --target cuda --max-batches 50 \\
           --out-dir results
   ```
   Output will be generated in results/vbatch_gemm_results_cuda.csv.

   By default, the above script only runs a subset of the configurations evaluated in the paper as a way to demonstrate that the artifact is functional. In order to run the full evaluation as in the paper, pass the --full-evaluation option to the script.
3. Batch size 512 may need to run separately with CoRAas it sometimes OOMs. This can be like so:
   ```
   python3 ./vbatch_gemm/tvm/vbatch_gemm.py \\
           --target cuda --batch-sizes 512 \\
           --max-batches 10 \\
           --data-file ./vbatch_gemm/data.txt \\
   ```

**Triangular Matrix Multiplication GEMM (trmm)*:**

1. Compile trmm/cublas by running make in those directories
2. Run
   ```
   python3 scripts/trmm_eval.py --target cuda \\
                                --out-dir results
   ```
   By default, the above script only runs a subset of the configurations evaluated in the paper as a way to demonstrate that the artifact is functional. In order to run the full evaluation as in the paper, pass the --full-evaluation option to the script. Output will be generated in results/trmm_results_cuda.csv.

#### A.7.2 Section 7.2: The Transformer Model

**Table 4: Transformer encoder forward execution latencies on Nvidia V100***

1. Ensure that the line #define OP_TIMES 1 (line 3) of the file bert_layer/faster_transformer/fastertransformer/time_map.h is commented out.
2. Build bert_layer/fastertransformer by running
   ```
   cmake -DSM=70 -DCMAKE_BUILD_TYPE=Release ..
   make
   ```
   in the folder bert_layer/fastertransformer/build.
3. Run
   ```
   python3 scripts/bert_layer_eval.py \\
   ```

```
--out-dir results \\
--max-batches 50 \\
--target cuda
```

By default, the above script only runs a subset of the configurations evaluated in the paper as a way to demonstrate that the artifact is functional. In order to run the full evaluation as in the paper, pass the `--full-evaluation` option to the script. Output will be generated in `results/bert_layer_results_cuda.csv`.

This will execute the model and generate execution latencies for a single encoder layer of the transformer model. During execution, CoRAincurs some overheads (referred to as prelude overheads) which, in a multi-layered model would be amortized across execution of multiple layers. In the paper, we report the execution latencies for a layer assuming a 6-layer model. In order to compute these execution latencies, we separately measure the prelude overheads and subtract 5/6th these prelude costs from the execution latencies measured above. The prelude overheads can be measured by running

```
python3 scripts/bert_layer_eval.py \\
       --target cuda --out-dir results \\
       --max-batches 50 --prelude-overhead
```

Output will be generated in `results/bert_layer_prelude_results_cuda.csv`.

**Figures 13 and A.10: Per-operator execution time breakdown for RACE dataset at batch size 128 (Figure 13) and COLA dataset at batch size 32 (Figure A.10 in Section D.8) on the V100 GPU**

1. Ensure that the line `#define OP_TIMES 1` (line 3) of the file `bert_layer/faster_transformer/fastertransformer/time_map.h` is not commented.
2. Build `bert_layer/fastertransformer` by running
   ```
   cmake -DSM=70 -DCMAKE_BUILD_TYPE=Release ..
   make
   ```
   in the folder `bert_layer/fastertransformer/build`.
3. Run
   ```
   python3 scripts/op_times_eval.py \\
          --target cuda --out-dir results \\
          --max-batches 50 --gen-libs
   ```
   Output will be generated in `results/per_op_times_results_cuda.csv`.

Just like above, we separately measure per-operator prelude overheads and subtract the appropritae amount from the per-operator ltencies measured above. These per-operator prelude overheads can be measured by running

```
python3 scripts/op_times_eval.py \\
       --target cuda --out-dir results \\
       --max-batches 50 --gen-libs \\
       --prelude-overhead
```

Output will be generated in `results/per_op_times_prelude_results_cuda.csv`.

**Figure 11: MHA with and without layout change op fusion for the RACE dataset**

Run

```
python3 scripts/pad_fusion_eval.py \\
       --target cuda --out-dir results \\
       --max-batches 50
```

Output will be generated in `results/pad_fusion_results_cuda.csv`.

**Figure A.4: Masked Scaled Dot-Product Attention with and without padding for the attention matrix for the RACE and MNLI datasets (more discussion in Section D.3)**

Run

```
python3 scripts/pad_fusion_eval.py \\
       --target cuda --max-batches 50 \\
       --out-dir results
```

Output will be generated in `results/bert_layer_mmha_results_cuda.csv`.

**Figure A.5: Memory consumption (more discussion in Section D.5)***

Run

```
python3 scripts/bert_layer_eval.py \\
       --target cuda --out-dir results \\
       --max-batches 50 --mem
```

Output will be generated in `results/bert_layer_mem_results_cuda.csv`.

### A.7.3 Sections 7.3 and D.6: Operation Splitting and Horizontal Fusion (AttnV in Section 7.3 and QKt in Section D.6)

Run

```
python3 scripts/bin_packed_eval.py \\
       --target cuda --out-dir results \\
       --max-batches 50
```

Output will be generated in `results/bin_packed_results_cuda.csv`.

### A.7.4 Section 7.4: CoRAOverheads

**Prelude overheads**

Run

```
python3 scripts/prelude_overheads_eval.py \\
       --out-dir results --max-batches 50
```

Output will be generated in `results/prelude_overheads_breakdown_results_cuda.csv`.

**Partial padding overheads (Figure A.8)**

Run

```
python3 intro_study/flops.py \\
        --max-batches 50 \\
        --out-file results/intro_flops.csv
```

**Ragged tensor overheads and load hoisting (Figure A.9)**
Run

```
python3 scripts/ragged_overheads_eval.py \\
        --target cuda --out-dir results \\
        --max-batches 50
```

Output will be generated in `results/ragged_overheads_results_cuda.csv`.

### A.7.5 Sections 7.5 and D.4: Evaluation Against Sparse Tensor Compilers*

1. Compile `taco/taco_bcsr_trmm`, `taco/taco_bcsr_trmul`, `taco/taco_csr_trmm`, `taco/taco_csr_trmul`, `taco/taco_csr_tradd` by running `make` in `taco/`
2. Run

   ```
   python3 scripts/taco_tr_eval.py \\
           --target cuda \\
           --out-dir results
   ```

   By default, the above script only runs a subset of the configurations evaluated in the paper as a way to demonstrate that the artifact is functional. In order to run the full evaluation as in the paper, pass the `--full-evaluation` option to the script. Output will be generated in `results/taco_results_cuda.csv`.

## A.8 Intel CPU Evaluation

Checkout the `intel` branch in the `cora_benchmarks` repository.

**Variable-Sized Batched GEMM (vgemm)*:**

1. Compile `vbatch_gemm/mkl`, by running make in those directories
2. Run

   ```
   python3 scripts/vbatch_gemm_eval.py \\
           --target cpu --max-batches 50 \\
           --out-dir results
   ```

   By default, the above script only runs a subset of the configurations evaluated in the paper as a way to demonstrate that the artifact is functional. In order to run the full evaluation as in the paper, pass the `--full-evaluation` option to the script. Output will be generated in `results/vbatch_gemm_results_cpu.csv`.

## A.9 8- and 64-core ARM CPU Evaluation

On the ARM CPU, we use one Docker image to evaluate PyTorch and CoRA, and a second one to evaluate TensorFlow. Therefore, one needs to separately invoke the evaluation scripts in both the containers and then manually merge the generated CSV files. Checkout the `arm_8` or the `arm_64` branch in the `cora_benchmarks` repository depending on whether the 8-core or the 64-core ARM CPU is used.

**Table 5: MHA evaluation on ARM CPU for PyTorch and CoRA***
In the container where we set up PyTorch and CoRA, run the following command from the root of the `cora_benchmarks` repository:

```
python3 scripts/bert_layer_eval.py \\
        --out-dir results \\
        --max-batches 50 \\
        --target cpu --pt-cora
```

Output will be generated in the file `results/bert_layer_results_cpu.csv`.

**Table 5: MHA evaluation on ARM CPU for TensorFlow**
In the TensorFlow container, run the following command from the root of the `cora_benchmarks` repository:

```
python3 scripts/bert_layer_eval.py \\
        --out-dir results \\
        --max-batches 50 \\
        --target cpu --tf
```

The open `--append` can be used if the results are to be appended to the same CSV file as the one with the results for PyTorch and CoRA. In any case, the output will be generated in/appended to the file `results/bert_layer_results_cpu.csv`.

By default in both the cases above, the script only runs a subset of the configurations evaluated in the paper as a way to demonstrate that the artifact is functional. In order to run the full evaluation as in the paper, pass the `--full-evaluation` option to the script.

We had to re-run these experiments after the reviews and the provided paper draft does not contain the revised results. The revised expected results can be found in the two provided CSV files named `cora_benchmarks/expected_results_plotting_script/arm_mha_8_core_results.csv` and `cora_benchmarks/expected_results_plotting_script/arm_mha_64_core_results.csv`. The files contain the results for the 8- and 64-core ARM CPUs respectively.

## A.10 Experiment customization

1. **GPU Evaluation:** The GPU evalaution has been tested on GPUs with compute capability 70. While the evaluation may run on other GPUs as well, we have not yet tested it thoroughly.
2. **Intel CPU Evaluation:** The Intel CPU evaluation should work for other CPUs beyond the CascadeLake CPUs we have tested it on. However, given that our schedules have been tuned for the 8-core CascadeLake CPU, the performance might not be optimal. Further, when running on other Intel CPUs, the LLVM target triple would need to be changed (in the file `cora_benchmarks/scripts/common.py` on line 10).

3. **ARM CPU Evaluation:** The ARM CPU evaluation should work for other CPUs beyond the Neoverse N1 CPUs we have tested it on. However, given that our schedules have been tuned for the Neoverse N1 CPU, the performance might not be optimal. Further, when running on other Intel CPUs, the LLVM target triple would need to be changed (in the file `cora_benchmarks/run_utils.py` on line 39). The number of threads would also be need to be changed for PyTorch evalaution in the files `cora_benchmarks/bert_layer/pytorch/layer_cpu_micro_batch.py` and `cora_benchmarks/bert_layer/pytorch/layer_cpu.py`.