Artifact README for "GALA: A High Performance Graph Neural Network Acceleration LAnguage and Compiler"

1 INTRODUCTION

This document provides instructions to evaluate the artifact for GALA. GALA is a compiler and DSL specifically targeting Graph Neural Networks (GNNs). We use publicly available datasets from DGL [2] and OGBN [1] for our evaluations. We also use publicly available, state-of-the-art GNN systems as baselines to compare against GALA.

Storage Requirements: The artifact is approximately 75 MB and uncompresses to 80 MB. The necessary datasets for GALA's main evaluation require approximately 19 GB of storage, while evaluating all the baselines and ablations requires an additional 200 GB of storage.

Hardware Requirements: GALA's evaluations will require an NVIDIA GPU. Specifically, the paper is evaluated using an A100 (80GB) and an H100 (94GB). While we expect to see similar observations in other hardware, difficulties in execution, such as additional out-of-memory errors, may arise.

2 GETTING STARTED GUIDE

After downloading our artifact, please follow the setup instructions for GALA (Section) and then proceed to the data download instructions (Section) to download the necessary datasets for evaluation. Please note that we used the latest versions of our evaluation baselines, which require multiple environments to be set up for execution.

2.1 Evaluation environments

Please follow the following instructions to setup GALA and our baselines.

2.1.1 Manual Setup. We require the following dependencies,

- Cuda Toolkit (GALA uses 12.4, while our other baselines require 11.8)
- LibTorch
- Conda (obtainable via Miniforge: see Section 2.1.2 for setup)
- g++ >= 11.4 and CMake >= 3.25

2.1.2 Installing Miniforge (Conda) on Linux. To set up Conda on a Linux machine, we recommend using Miniforge—a lightweight Conda installer maintained by the conda-forge community. This is particularly useful for clean environments and better compatibility with PyTorch and DGL packages. Use the following commands to download and install Miniforge:

```
# Download the latest Miniforge installer
wget "https://github.com/conda-forge/miniforge/releases/latest/download/
    Miniforge3-$(uname)-$(uname -m).sh"

# Run the installer
bash Miniforge3-$(uname)-$(uname -m).sh
```

After installation, restart your shell or activate Conda using:

```
source ~/miniforge3/bin/activate
```

Author's address:

You can now create and activate the Conda environments using the provided `requirements.txt` files for each baseline. Below are the commands to install dependencies for each baseline using their respective environment files.

GALA and DGL

```
# assuming the Cuda Toolkit v12.4 is being used
conda create -n gala python=3.11
conda activate gala
pip install torch==2.4.1 --index-url https://download.pytorch.org/whl/cu124
conda install -c dglteam/label/th24_cu124 dgl
pip install ogb
conda install packaging
conda install conda-forge::bison
conda install conda-forge::seaborn
conda install anaconda::pandas
conda install anaconda::scipy
cd scripts/Environments
mkdir libtorch
cd libtorch
wget https://download.pytorch.org/libtorch/cu126/libtorch-cxx11-abi-shared-with-
     deps-2.7.1%2Bcu126.zip
unzip libtorch-cxx11-abi-shared-with-deps-2.7.1+cu126.zip
cd ../../..
mkdir build
cd build
cmake ..
make -j5
cd ../codegen
mkdir build
cd build
cmake -DCMAKE_PREFIX_PATH="$PWD/../../scripts/Environments/libtorch/libtorch" ..
```

SeaStar

Note that we built SeaStar using the Cuda Toolkit v11.8.

```
cd scripts/Environments/SeaStar
conda env create --name seastar-gala-ae --file=.yml
tar -xvzf seastar-source.tar.gz
cd seastar
cd dgl-hack && mkdir build && cd build && cmake .. && cd .. && ./compile.sh
cd ..
cd Seastar-Documentation
cd Seastar/python
python setup.py install
```

WiseGraph

```
cd scripts/Environments/WiseGraph
cd a100 # or h100
bash wisegraph_a100.sh  # or wisegraph_h100.sh
# For A100: use wisegraph_a100.sh (requires CUDA Toolkit 11.8)
# For H100: use wisegraph_h100.sh (requires CUDA Toolkit 11.8)
# Ensure the respective CUDA Toolkit version is preinstalled locally
```

SparseTIR

```
cd scripts/Environments/SparseTIR
git clone --recursive https://github.com/uwsampl/SparseTIR.git
# For H100
cp h100_sparseTIR_env.yml SparseTIR/requirements.yml
# Else
cp a100_sparseTIR_env.yml SparseTIR/requirements.yml
cd SparseTIR
conda deactivate
conda env create --file requirements.yml --name stir-gala-ae
conda activate stir
conda install -c conda-forge gcc=12.3.0 gxx=12.3.0 llvm=15.0.7 clang=15.0.7
    llvmdev=15.0.7
export CC=$CONDA_PREFIX/bin/gcc
export CXX=$CONDA_PREFIX/bin/g++
export LLVM_CONFIG=$CONDA_PREFIX/bin/llvm-config
# llvm version issue
cd src/target/llvm
sed -i 's/getAlignment()/getAlign().value()/g' codegen_amdgpu.cc codegen_nvptx.
    cc codegen_llvm.cc
cd ../../..
export LIBRARY_PATH="$CONDA_PREFIX/lib:$CONDA_PREFIX/lib64:$LIBRARY_PATH"
export LD_LIBRARY_PATH="$CONDA_PREFIX/lib:$CONDA_PREFIX/lib64:$LD_LIBRARY_PATH"
mkdir -p build && cd build
cp ../../sparseTIRconfig.cmake config.cmake
# Below Linen Only for H100
ln -s $CONDA_PREFIX/lib/librhash.so.1 $CONDA_PREFIX/lib/librhash.so.0
cmake ..     -DCMAKE_C_COMPILER=$CONDA_PREFIX/bin/gcc     -DCMAKE_CXX_COMPILER=
    $CONDA_PREFIX/bin/g++     -DLLVM_CONFIG=$CONDA_PREFIX/bin/llvm-config     -
    DUSE_CUDA=ON     -DCUDA_TOOLKIT_ROOT_DIR=$CONDA_PREFIX     -
    DCUDA_CUDA_LIBRARY=$CONDA_PREFIX/lib/stubs/libcuda.so   -DCMAKE_LIBRARY_PATH
    =$CONDA_PREFIX/lib/stubs/
make -j$(nproc)
# install python binding
cd ../python
python setup.py install
# gcc/12.3.0 used for installation, gcc/9.5.0 used for running examples
conda install -c conda-forge gcc=9.5.0 gxx=9.5.0
# quick sanity check
cd ../examples/spmm
python bench_spmm.py
```

If any issues arise, please see the SparseTIR documentation for setup instructions and create a Conda environment $stir - gala - ae$.

2.2 Datasets

We use a standard set of benchmark graph datasets sourced from the DGL and OGB repositories. These datasets are downloaded and preprocessed into binary formats to enable efficient loading and compatibility with different evaluation frameworks. To fetch and preprocess all datasets, run the following script within the DGL-specific Conda environment:

```
conda activate gala
cd scripts/Data
python get_all_datasets.py
source ../scripts/Environments/WiseGraph/h100/.venv/cxgnn/bin/activate
python get_all_datasets.py --wisegraph  # for wisegraph
```

The script `get_all_datasets.py`, located in `scripts/Data/`, automatically downloads multiple popular graph datasets (e.g., Cora, Pubmed, Reddit, CoraFull), applies transformations such as self-loop addition, etc. These files are later used by various frameworks such as GALA and WiseGraph, each of which requires inputs in a specific binary format. The preprocessing code and associated utilities are maintained in the `scripts/Data/` directory. The directory also includes a `README.md` file with basic setup and usage instructions. Users can refer to this for further guidance on running the data scripts.

2.2.1 Framework-Specific Preprocessing. Although all our experiments use the same benchmark datasets, originally sourced from the DGL and OGB repositories, we preprocess and store them in different formats tailored to each framework's input requirements. For example, **WiseGraph** requires graph inputs to be explicitly stored in compressed sparse row (CSR) format with auxiliary metadata. Specifically, for each dataset, we convert the DGL graph into CSR representation by extracting the edge list and saving the pointer (‘indptr‘) and index (‘indices‘) arrays as ‘.dat‘ binary files. These files are stored under a processed/ subdirectory specific to each dataset. Despite differences in format, the underlying graph data remain the same as those provided by the original DGL and OGB datasets. The preprocessing only transforms the representation, not the content, ensuring consistency across all evaluated frameworks.

3 STEP-BY-STEP INSTRUCTIONS

The following sections describe how to reproduce our results. Please note the following points,

- Most scripts are run multiple times. First to get the necessary data (e.g. runtimes) and then again to generate the Figure/Table seen in the paper. These results are either printed directly or written to the Results directory as PDFs (Figures 16, 17, 18, 19, and 20).
- Other than Figures 16 and 17, the remaining evaluations were performed on an NVIDIA H100.
- All the presented code snippets assume you are in the initial directory of the artifact (and have a cd scripts/.. line that may be unnecessary to execute all the time).

3.1 Figures 16-17 - Main evaluation against baselines

These figures present the main evaluations of our paper. Figure 16 shows the speedups for inference, while Figure 17 shows the speedups for training. Note that to fully produce (both parts (a) and (b)) for both figures in the paper, results from both an NVIDIA A100 machine and an NVIDIA H100 machine are necessary. Below we list the commands necessary to run GALA, and each baseline, and finally produce the figure once all the necessary data has been obtained.

As each hardware has specific scheduling choices, a choice is exposed in the script. (via –hw with choices being either h100 or a100, h100 is the default and would generate a code that can still run on any NVIDIA-GPU, but may not be optimal).

GALA

```
conda activate gala
cd scripts/Evaluations
python Figures-16-17.py --job gala # gets inference results
python Figures-16-17.py --job gala --train # gets training results
```

The following is an example for running specifically on the A100, (this –hw parameter can be set for other baselines as well)

```
conda activate gala
cd scripts/Evaluations
python Figures-16-17.py --job gala --hw a100 # gets inference results
python Figures-16-17.py --job gala --hw a100 --train # gets training results
```

DGL

```
conda activate gala
cd scripts/Evaluations
python Figures-16-17.py --job dgl # gets inference results
```

SeaStar

```
# Requires Cuda toolkit v11.8 if built using it
conda activate seastar-gala-ae
cd scripts/Evaluations
python Figures-16-17.py --job sea # gets inference results
```

WiseGraph

```
source ../Environments/WiseGraph/h100/.venv/cxgnn2/bin/activate
cd scripts/Evaluations
python WiseGraph.py --job F16n17 --h100 # or --a100
```

SparseTIR

```
conda activate stir-gala-ae
cd scripts/Evaluations
python Figures-16-17.py --job stir # gets inference results
```

Once all the data has been collected from the baselines, the figure can be created.

```
conda activate gala
cd scripts/Evaluations
python Figures-16-17.py --job fig # figure 16 for the current hardware
```

Claim: GALA achieves significant overall speedups over the evaluated baselines through the optimizations it performs.

3.2 Figure 18 - Scalability with different layers and hidden dimensions

We show GALA's scalability through presenting speedups against WiseGraph (a state-of-the-art GNN system released in 2024), for different numbers of layers and hidden dimensions for the Reddit graph dataset, and the GCN GNN model.

```
cd scripts/Evaluations
conda activate gala
python Figure-18.py # get the runtimes for gala
source ../Environments/WiseGraph/h100/.venv/cxgnn2/bin/activate
python WiseGraph.py --job F18 --h100  # runtimes for WiseGraph
conda activate gala
python Figure-18.py --job stats # prints the speedups
```

Claim: GALA maintains its speedups for GNNs even when scaling via multiple layers, and larger hidden dimensions.

3.3 Table 5 - Scalability with different graph sizes

We show GALA's scalability through presenting speedups against WiseGraph (state-of-the-art GNN system released in 2024), and DGL for different graph sizes. These different sizes are obtained by node sampling the OGBN-papers100M dataset for the total node percentages of 1%, 2%, 5%, 10%, and 20% for a GCN model with 2 layers and a hidden dimension of 32.

```
conda activate gala
cd scripts/Data
python get_all_datasets_scale.py # download and process datasets for gala
cd ../Evaluations
python Table-5.py --job dgl # get the runtimes for dgl
python Table-5.py # get the runtimes for gala
source ../Environments/WiseGraph/h100/.venv/cxgnn2/bin/activate
python WiseGraph.py --job T5 --h100
conda activate gala
python Table-5.py --job stats # prints the speedups
```

Claim: GALA maintains its speedups for varying graph dataset sizes.

3.4 Figure 19 - Memory usage vs performance

We show the memory usage and runtime of GALA, DGL, and WiseGraph for a two-layer GCN model with a hidden dimension size of 32.

```
conda activate dgl
cd scripts/Evaluations
conda activate gala
python Figure-19.py # get the runtimes for gala and DGL
source ../Environments/WiseGraph/h100/.venv/cxgnn2/bin/activate
python WiseGraph.py --job F19 --h100
conda activate gala
```

```
python Figure-19.py --job stats # prints the speedups
```

Claim: Different choices in GALA can lead to either better performance at the cost of memory used, or low memory consumption while still being competitive compared to other baseline systems in terms of performance.

3.5 Table 6 - Different methods of sampling

We demonstrate the various sampling methods available in GALA and their respective implications for performance and accuracy.

```
cd scripts/Evaluations
conda activate gala
python Table-6.py # get the runtimes for gala
python Table-6.py --job stats # prints the speedups
```

Claim: Different sampling methods can lead to different outcomes. No sampling is the slowest, but can have the highest accuracy (Reddit, on the other hand, OGBN-Products shows better results with sampling). Data sampling is the fastest, while dynamic kernel sampling has the best accuracy among the sampling methods of GALA. Note that these observations would have a slight variation due to the randomness of values.

3.6 Figure 20 - Input-aware compilation

We show that the input-aware compilation option of GALA achieves similar speedups to hand-optimized schedules.

```
conda activate gala
cd scripts/Evaluations
python Figure-20.py # get the runtimes
python Figure-20.py --job stat # creates the figure
```

Claim: GALA's input-aware compilation's speedups are within 10% of hand-optimized schedules.

REFERENCES

[1] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687*, 2020.

[2] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander Smola, and Zheng Zhang. Deep graph library: Towards efficient and scalable deep learning on graphs, 2019.