Appendix: Artifact Description/Artifact Evaluation

ARTIFACT DOI

https://doi.org/10.5281/zenodo.11634155

ARTIFACT IDENTIFICATION

Our paper enables GPU graph (CUDA Graph as an implementation example) execution through OpenMP directives, enhancing heterogeneous programming while maintaining a simple, directive-based programming style. The work is based on LLVM 17 and CUDA graph. Our benchmarks show that the proposed framework performs consistently better than the original OpenMP target tasking model, and is at least as efficient as threading model, or in many cases, outperforming the latter.

This artifact allows users to reproduce our experiments. It is delivered as a Docker image in *tarll* format , users can build a docker container from it.

It contains the customized LLVM source code with the benchmarks' source code presented in the paper. Detailed execution instructions and data handling scripts are provided to facilitate step-by-step reproduction of the study.

REPRODUCIBILITY OF EXPERIMENTS

Once the docker container is instantiated with interactive mode, one simply needs to execute prepared scripts to reproduce the results.

- To load the image from the provided file:
  ```
  $ docker load -input »TheTarBallFile«
  ```
- The loaded image ID is *7772e3880d07*. One can create a tag for the loaded image by doing the following:
  ```
  $ docker tag 7772e3880d07 icpp_arti_157:latest
  ```
- Next, it is needed to instantiate a container from the loaded image by execute the following:
  ```
  $ docker run -it -gpus all icpp_arti_157:latest
  ```

Once connected to the container after its instantiation, one can build the customized llvm compiler (roughly **1 hour** with 10 CPU cores):

```
$ cd $HOME/llvm;
$ ./build.sh
```

The script will utilize all available CPU resource to compile the compiler. If the script is finished without error, the compiler binaries should be in the `PATH` environment variable.

Reproduce the Experiments:

To build all benchmarks (about **10 minutes**), do:

```
$ cd $HOME/apps/TDG2CUDAGraphBenchs
$ ./build_all.sh
```

To run the built benchmarks to obtain the average execution time (Section 4.2 in the paper), do:

```
$ ./run_mean_time.sh
```

To run the built benchmarks to obtain the scalability results (Section 4.3 in the paper), do:

```
$ ./run_scalability.sh
```

Both scripts require the number of executions as a parameter. The results in the paper were obtained with 50 iterations, with 20 CPU cores, it took **2 hours**. Consider to lower number of iterations to shorten the needed time, e.g., to 30 iterations. To customize the number of threads, users can set the OMP_NUM_THREADS environment variable, if not, the script sets it to 20.

Expected results: Average Time (Paper Section 4.2)

```
$ ./read_mean_time.sh
```

The above script reads the execution time recorded by each of the benchmarks and computes mean execution time.

The executions of `read_data.sh` will print a series of execution times. Corresponding to the three versions tested in the paper:

(1) original blocking, referred to as *target tasks* in the paper,
(2) original threading, which corresponds to code parallelized with `teams distribute`,
(3) TDG, which is generated with the proposed framework.

For `TDG`, the results should show it has similar execution time as `original threading` and shorter time when compared to `original blocking`. The speedup over `original blocking` varies with the underlying architecture, according to our experiments, the expected relative speedup of 1.2x to 2x (Figure 7 in the paper).

Expected results: Scalability Assessment (Paper Section 4.3)

For scalability tests, the relative speedup of `TDG` over `original blocking` should be increasing when the number of graph nodes increases. Furthermore, as the HEAT simulator incurs more runtime overhead, our framework should demonstrate better relative speedup (2x to 10x) than for Nbody simulation (1.1x to 2x).

Additionally, this experiments can be realized with thread oversubscription by setting OMP_NUM_THREADS, experimenters should see slight time increment with `TDG` against large overhead incurred by `original blocking`.

ARTIFACT DEPENDENCIES AND REQUIREMENTS

(1) Hardware dependencies: the docker image is built for Intel X86 CPU architecture and Nvidia GPUs
(2) Software dependencies: the image is built on a Ubuntu 20.04 (5.15.0-105-generic) machine. The OS of the image is Ubuntu 22.04. The CUDA version is 11.8, according to Nvidia, the minimum host CUDA driver version to support it is 450.80.02. Other packages are installed in the image.

ARTIFACT INSTALLATION DEPLOYMENT PROCESS

The artifact is delivered as a docker file. Experimenters can instantiate a container from this image and proceed.