Artifact for Euro-Par 2020 paper `Accelerating Nested Data Parallelism: Preserving Regularity`

This is the source code we used to do the experiments of Section 5 (Evaluation). More specifically you can reproduce the four figures of Figure 3 with this code. The repository can be found here it can be cloned with

```
git clone --single-branch -b artifact \
 https://github.com/sakehl/FourierTests.git
```

NOTE You do not have to clone this repository for running everything, it is also included in the docker image. But if you want to inspect the code we use, this repo is more usefull than a docker image.

Getting started guide

Requirements

- A linux machine (only tested with Ubuntu 18.04)
- Nvidia GPU, atleast 11GB of memory is needed to run all experiments. (Less memory is possible, but some experiments will run out of memory)
- Atleast nvidia driver version >= 418.39 must be installed.
- CUDA toolkit is not needed, comes with the docker.

Installation

Install the following

- Docker
- Nvidia's Docker runtime

We need a folder on the host, so you can easily see the produced (intermediate) results. In the folder where you are going to run the docker image and experiments, make a `data` folder. E.g.

```
mkdir ~/euro-par-20-accelerate
cd ~/euro-par-20-accelerate
mkdir data
```

Now do the following

```
sudo docker run --gpus all -it --privileged --mount\
type=bind,source="$(pwd)"/data,target=/root/FourierTests/data \
lvandenhaak/accelerate-euro-par-20
```

And see if everything is working.

Step-by step instruction on how to reproduce the results

NOTE: If you want to shorten the experiments, read the next section first. Otherwise the runs might take up to 3-5 hours. The original experiments were conducted on a GeForce RTX 2080Ti (compute capability 7.0, 68 multiprocessors = 4352 cores at 1.65GHz, 11GB RAM) backed on by 16-core Threadripper 2950X (1.9GHz, 64GB RAM).

1. Start the docker image

   ```
   sudo docker run --gpus all -it --privileged --mount\
    type=bind,source="$(pwd)"/data,target=/root/FourierTests/data \
    lvandenhaak/accelerate-euro-par-20
   ```

2. To reproduce the quicksort results do (inside the docker bash)

   ```
   ./make_quicksort_dat.sh
   ```

   Note this can take up a long time, on our machine about 2 to 3 hours. Mostly busy compiling Accelerate's Irregular version.

3. Inspect the results in the data folder on your host. `quicksort-100.pdf`, `quicksort-1000.pdf` and `quicksort-10000.pdf` should be created and look similar to figure 3 of the paper. The files `quicksort-100.dat` contain the raw data points. The files `result_quicksort_Regular_100_1.csv` contain the output we get from the nvidia profiler (nvprof). They can be compared with the exact paper results, which can be found in ResultsPaper.

4. To reproduce the fourier results do (inside the docker bash)

   ```
   ./make_fourier_dat.sh
   ```

   Note also takes a long time, about 1 to 2 hours on our machine

5. Inspect the results in the data folder on your host. `fourier32x32.pdf`, `quicksort-1000.pdf` and `quicksort-10000.pdf` should be created and look similar to figure 3 of the paper.

Shortening the runs

We've provided some arguments to the above bash scripts, to run shorter versions, although not all figures will be produced in that case. This can be run with

```
./make_quicksort_dat.sh --short
./make_fourier_dat.sh --short
```

Run specific benchmark again

To run a specific benchmark again, you can give some other arguments, e.g.

```
./make_quicksort_dat.sh --regular -m 100 -n 1
./make_fourier_dat.sh --futhark -n 1
./make_fourier_dat.sh --irregular --no-input
```

For quicksort, valid values for m are 100, 1000 and 1000 (the three differunt subgraphs). Valid values for n are 1, 100, 1000, 2000, 5000, 10000. Valid versions (as indicated by the legend of Figure 3) are –regular (Accelerate, Regular) –irregular (Accelerate), –futhark (Futhark).

For fourier, valid values for n are 1, 100, 1000, 5000, 10000, 20000. Valid versions (as indicated by the legend of Figure 3) are –regular (Accelerate, Regular) –irregular (Accelerate), –futhark (Futhark), –cufft (cuFFT) and –normal (Normal).

Add the option `--no-input` to not remake the input again.

Out of memory problems

Unfortunately, processes that use the maximum ammount of memory of the GPU (or nearly so) can stall a long time. You can just cancel a specific benchmark with `[ctrl] + c`. It will give a warning that it isn't included in the .dat file, but it should still process fine.

Structure of code

- First of all, you can see the `Dockerfile` that was used to build the docker image you are using. It uses CUDA 10.1, LLVM 9, installs the code in the repository via stack, and get futhark version 0.13.2 from the futhark site. It also sets some environment variables.
- app/Main.hs contains the benchmarks of the Accelerate versions. Basically, it just runs them 10 times, given the right arguments.
- src/Quicksort.hs contains the code used for quicksort (`quicksort`) in Accelerate. This is the normal non-nested version. In src/QuickSortTest.hs the nested version (works on list of 1d arrays) can be found, which is again called from Main.
- src/FFTAdhoc.hs contains the code used for fourier in Accelerate (`fft2D`). This is the normal non-nested version. In src/FourierTest.hs the nested version (works on list of 2d arrays) can be found, which is again called from Main. The normal (src/FourierTest.hs) version can also be found there. The cuFFT version can be found here, a fork of the accelerate-fft package.
- Futhark/fft-lib.fut contains the Futhark fourier version we are using, the source code of the fft can be found here from the futhark github. Which we coppied here in our repository.
- Futhark/quicksort.fut contains the Futhark quicksort version we are using, it is the same algorithm that Accelerate uses, actually we just ported the Accelerate version to Futhark.
- input_gen.py generates random intergers in lists, and outputs them to the data directory.

- make_fourier_dat.sh is a bash script that does all the fourier experiments.
- Both accelerate and futhark are profiled with nvprof. We use these options `--csv` (output as csv) `-f` (overwrite output files) `-u ms` (measure in ms) `--trace gpu` (only measure gpu activity) `--log-file data/result_fourier_$v\_$n.csv` (log file output) for Accelerate, for Futhark we add `--profile-child-processes` since it spawns child processes which do the actual compitations. Note this profiling is thus only profiling the executing times of the kernels and memory transfer overheads (`--gpu` option), as indicated in the paper, this is not the actual time the program is running, since there is idle time in between kernels. Also note, we don't measure the compile time, which is during runtime for Accelerate, but is not GPU activity.
- Note that we use Futhark with `futhark bench -r 9`, this is actually executes the Futhark program 10 times, the bench option always does one warm-up run, that is profiled.
- You can see that we add `--exclude=$n` to the futhark version, this was the easiest way to select a certain benchmark (see the tags in Futhark/fft-lib.fut)
- We call input_gen.py from the bash script, to generate the input files
- All the profiling is stored in files with names `data/result_fourier_$v\_$n.csv` where `$v` is the version and `$n` is the n value (which is plotted in the x-axis of the figures)
- We call process_csv.py which processes all the outputed csv files, into on `data/fourier32x32.dat`
- We call gnuplot with fourier32x32.gnuplot as argument, which produces the output file `data/fourier32x32.pdf`
- make_quicksort_dat.sh does the same things, but for quicksort. Eventually it produces the files `data/quicksort-100.pdf`, `data/quicksort-1000.pdf` and `data/quicksort-10000.pdf`
- The accelerate compiler versions can be found in the file `stack.yaml`. And specifaclly this Accelerate version. The static analyses from section 3 of the paper can be found here, the part that turns of the analyses can be found here. This is interacted on with the function `setforceIrreg :: IO ()`, which is used in app/Main.hs on line 41.