Korea Advanced Institute of Science and Technology
CS492 Spring 2021
Project 5

Jinwoo Hwang
jwhwang@casys.kaist.ac.kr

Due: June. 18th 11:59 PM KST

Rules

- Do not copy from others. You will get huge penalty according to the University policy. Sharing of code between students is viewed as cheating.
- Every code was written and will be tested in the provided docker environment. You need to implement your code in the given environment.
- Read carefully not only this document but the comment in the skeleton code.
- This is the version 1.0 document which means not perfect. Any specifications or errors can be changed during the project period with announcement. Stay tuned to Piazza.

Introduction

In this project, you are going to optmize the your GPU version implementation from the last project even further (1) by exploiting the low precision operation supported in the recent GPUs and (2) by reducing the number of memory copy.

Project Structure

This project contains the following files:

- `examples/`: Example files for using tensor cores
- `src/`: Directory containing the source code for the helper functions
- `src/Makefile`: Makefile to run the code
- `sample.jpg, sample_out.jpg`: Sample input, output image file

Quantization

Using lower precision for computation brings two major upsides. First, the required amount of memory is decreased. For example, in the case of half-precision floating-point format (FP16) which uses 16 bits requires half the size of memory compared to 32 bits for single precision (FP32). Secondly, if the hardware supports operation directly on the lower precision, it may shorten the computation time. Execution time can be sensitive to memory or arithmetic bandwidth. Half-precision halves the number of bytes accessed, thus reducing the time spent in memory-limited operations. Luckily, the NVIDIA RTX Titan GPU that we are using in class integrates Turing tensor cores which work directly with INT4/INT8/FP16 operations.

However, there is also an obvious downside. You will lose the expressiveness of the variables and the accumulated error might lead to an inaccurate result. Fortunately, the inference computation tends to be more tolerable to errors compared to the training process where gradients in small sizes play a vital role.

Tensor Core

The tensor cores were introduced first in NVIDIA Volta GPU architectures, then pushed further in the next Turing architecture to support INT4 and INT8 operations directly. Interestingly, the tensor cores are designed to use mixed precisions. For instance, when multiplying two FP16 matrixes, they first accumulate the results in FP32, and later convert it into FP16. According to NVIDIA, such a design was chosen because they found out that the accumulation into single precision is critical to achieving good training results. In cuBLAS, you can control such behavior by changing `computeType` in `cublasGemmEx` and check the consequences. For more information on the tensor cores, I strongly recommend reading the NVIDIA Volta Whitepaper.

Also, you can check out the theoretical throughput of the tensor cores that NVIDIA reported from this link. Note that the Turing architecture has compute capability of 7.5.

Implementation

Overview

Inside the `src/`, you will find a similar project structure to the previous projects. Start by copying your `dnn_cublas.py or dnn_cuda.py` from project 4 as `dnn.py`. The `__init__.py` is modified to hand over a new boolean variable `precision` from the command line argument. Change your `dnn.py` accordingly.

(1) Make `dnn.py` use the tensor cores in the precision specified from the command line argument.
(2) Optimize `dnn.py` by reducing the number of memory copy between host and GPU.

Since we are only interested in the final values and not the intermediate values when performing the forward propagation, we could eliminate the unnecessary memory copy of those values by leaving them inside GPU device memory and only their pointers to the next layer. In this project, you are required to optimize your code to use exactly two memory copies between host memory and GPU device memory, namely, one for submitting the input tensor, and the other for retrieving the final output tensor. Memory copies triggered during the initialization step of the model and the `cudaMemcpy` with `cudaMemcpyDeviceToDevice` which occurs only within a GPU will not be counted. To achieve this optimization, you will need to use both CUDA and cuBLAS. Note that using CPU computation in the middle of layer means you will have to copy the tensors back and forth and also note that the tensor core API is only exposed for CUDA, not cuBLAS.

Lastly, read the content of the `Makefile` carefully and make sure your code works with the following three commands since it will be used when grading your codes.

```
$ make {run_int8|run_fp16|run_fp32}
```

Functionality (35 pt)

- FP32
  - Memory Copy Optimization (5pt)
- INT8
  - Implementation (10pt)
  - Memory Copy Optimization (5pt)
- FP16
  - Implementation (10pt)
  - Memory Copy Optimization (5pt)

Report (25 pt)

Implement time measuring code inside `dnn_*.cu` to measure the time took for type conversion, memory copy, and computation and include them in your report.

The `__init__.py` generates `FP32.npy`, `FP16.npy`, and `INT8.npy` containing the values from the output tensor. Open them by using 'np.load' and measure the normalized root-mean-square error (NRMSE) of the quantized tensors against the FP32 value and include them in the report.

$$NRMSE = \frac{\sqrt{\sum_i (x_i - y_i)^2 / |X|}}{y_{max} - y_{min}}, x_i \in X, y_i \in Y$$

where x_i is an approximation of value y_i.

Write a detailed report that describes your code. Your report must include

- Analysis on the performance breakdown
- Analysis on the tradeoff between the errors and the inference speedup
- Data types of the tensors throughout the forward propagation
  (e.g., INT8 `->` conv0 `->` INT32 `->` bn0 ...)

The running time should be measured in the given KCloud GPU server. The purpose of the report is to check your understanding. Please write the answer clear.

Using the GPU Server

For project 5, you don't have to use the docker image. The GPU server has been set up as the same environment as the docker image. Running your code on the server should not take up the entire GPU memory, but your code might fail if many students are running their code at the same time. In such case, you will have to wait until the GPU is idle. You can check who is using the GPU with `nvidia-smi`, and other tools like `htop` and `who` will be helpful as well.
DO NOT launch python in interpreter mode and import TensorFlow module since it would hold up the GPU resource and prevent your classmates from using it.

Handin

You will need to submit a single tarball containing 4 files(`dnn.py, dnn_fp32.cu, dnn_fp16.cu, dnn_int8.cu`) and a report in PDF format. The name of the file should be `[student ID].tar`. Upload your tar file to KLMS.

For project 5, late submission will only be permitted up to one more day (20 June) due to the end of the semester.

Reference

The APIs in CUDA and cuBLAS tend to change often as their version changes. Note that guides for different versions might contain misleading information. The CUDA & cuBLAS version is `10.0`.

- cuBLAS - https://docs.nvidia.com/cuda/archive/10.0/cublas/index.html
- cuBLAS - https://developer.nvidia.com/sites/default/files/akamai/cuda/files/Misc/mygpu.pdf
- Debugging - https://forums.developer.nvidia.com/t/debugger-for-python-c-c-cuda/67966/2