HiSVP GFPQ Library User Guide

Issue	**01**
Date	**2019-05-20**

HiSilicon (Shanghai) Technologies Co., Ltd.

Address:	New R&D Center, 49 Wuhe Road, Bantian, Longgang District, Shenzhen 518129 P. R. China
Website:	http://www.hisilicon.com/en/
Email:	support@hisilicon.com

About This Document

Purpose

This document describes how to use the grouped floating-point quantization (GFPQ) library and integrate it to Caffe for efficient network fine-tuning.

Related Version

The following table lists the product versions related to this document.

Product Name	Version
Hi3559A	V100
Hi3559C	V100
Hi3519A	V100
Hi3516D	V300
Hi3516C	V500
Hi3559	V200
Hi3531D	V200

Intended Audience

This document is intended for:

- Technical support engineers
- Software development engineers
- Algorithm development engineer

Symbol Conventions

The symbols that may be found in this document are defined as follows.

Symbol	Description
⚠ DANGER	Indicates an imminently hazardous situation which, if not avoided, will result in death or serious injury.
⚠WARNING	Indicates a potentially hazardous situation which, if not avoided, could result in death or serious injury.
⚠CAUTION	Indicates a potentially hazardous situation which, if not avoided, may result in minor or moderate injury.
NOTICE	Indicates a potentially hazardous situation which, if not avoided, could result in equipment damage, data loss, performance deterioration, or unanticipated results. **NOTICE** is used to address practices not related to personal injury.
📖 NOTE	Calls attention to important information, best practices and tips. **NOTE** is used to address information not related to personal injury, equipment damage, and environment deterioration.

Change History

Changes between document issues are cumulative. The latest document issue contains all changes made in previous issues.

Date	Issue	Change Description
2019-05-20	01	This is the first official release.

Contents

About This Document....................i

1 Introduction....................1

1.1 Overview....................1

1.2 Application Scenario....................1

2 GFPQ Library....................3

2.1 Overview....................3

2.2 Environment Dependency....................3

2.3 File Description....................3

2.4 Interface Description....................4

2.5 Usage of the GFPQ Library (Linux)....................5

2.5.1 Checking the CUDA Version....................5

2.5.2 C++ Sample....................5

2.5.3 Python Sample....................6

2.6 Usage of the GFPQ Library (Windows)....................6

3 Integrating the GFPQ Library in Caffe....................8

3.1 Data Quantization....................8

3.1.1 Downloading and Compiling Caffe....................8

3.1.2 Adding the Quantization Layer QuantLayer....................9

3.1.3 Linking the GFPQ Library....................9

3.2 Weight Quantification....................10

3.2.1 Weight Quantification Process....................10

3.2.2 Quantization with Weight Update in Fine-tuning....................11

3.2.3 Weight Quantization in Layer-Folding....................12

3.3 Fine-Tuning....................13

3.3.1 Building the Fine-Tuning Environment....................13

3.3.2 Fine-Tuning Methods and Precautions....................13

4 FAQs....................15

4.1 GFPQ Library Error....................15

4.2 GFPQ Library Calling Error During Fine-Tuning....................15

Figures

Figure 1-1 Positions of data quantization and dequantization in the NNIE and Caffe..2
Figure 2-1 Checking the GPU version..5
Figure 2-2 Sample running result... 6
Figure 2-3 sample_on_python.py running result...6
Figure 2-4 sample_on_python_gpu.py running result..6
Figure 2-5 sample.exe running result.. 7
Figure 3-1 Position of the quantization layer...8
Figure 3-2 Modifying Makefile.config..9
Figure 3-3 Modifying Makefile — 1..9
Figure 3-4 Modifying Makefile — 2.. 10
Figure 3-5 Weight quantification process.. 11
Figure 3-6 Fine-tuning process.. 12
Figure 3-7 Folding the BatchNorm and Scale layers...13
Figure 4-1 GFPQ library error... 15

1 Introduction

1.1 Overview

Caffe performs floating-point operations, while the neural network inference engine (NNIE) performs int8/int16 operations. Therefore, precision loss is inevitable. It is designed to control the precision loss within 1%. The precision loss of most networks is less than 0.5%, which can fully meet the application requirements.

The quantization algorithm cannot ensure that the precision loss in all cases is within the expected range. The quantization precision is affected by multiple factors. In some cases (for example, a high dispersion of the parameters to be quantized), the quantization precision loss may exceed the designed target.

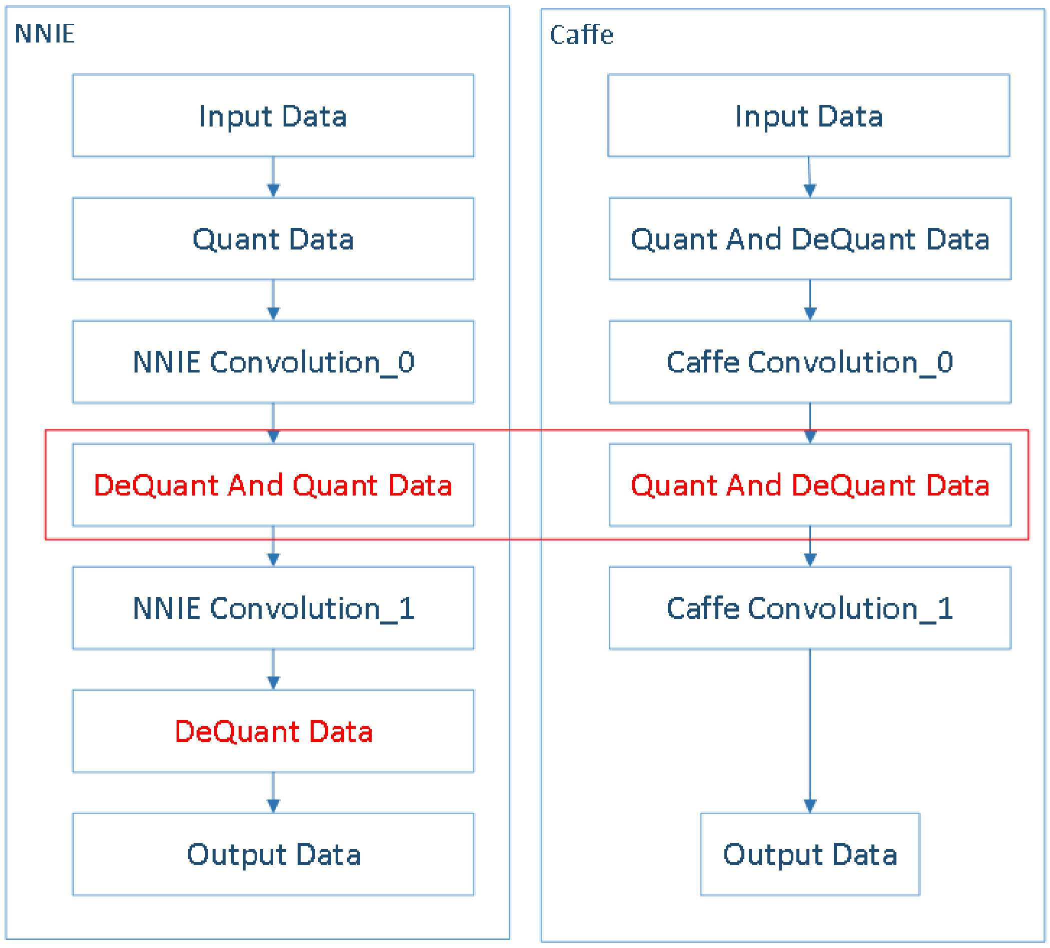
When the quantization precision loss is unacceptable, the network needs to be fine-tuned with quantization and dequantization to reduce the quantization error.

1.2 Application Scenario

The layers that use 8-/16-bit operations of the NNIE include convolutional, deconvolution, DepthwiseConv, pooling, and InnerProduct layers. They are referred as NNIE layers. The data and weight are quantized to the 8-/16-bit format before they participate in the operations.

Therefore, before network fine-tuning, the data and weights of the NNIE layers need to be quantized and dequantized, to reduce the quantization error and improve the precision.

Figure 1-1 Positions of data quantization and dequantization in the NNIE and Caffe



- In the NNIE, DeQuant And Quant Data dequantizes the output from the upper layer Convolution_0, and then quantizes the lower layer Convolution_1.
- In Caffe, Quant And DeQuant Data quantizes and de-quantizes the output from the upper layer Convolution_0.

2 GFPQ Library

2.1 Overview

The grouped floating-point quantization (GFPQ) library provides interfaces for quantizing and dequantizing the input data before output, for example, quantizing the input floating-point data to int8/int16 data, and then dequantizing the int8/int16 data to floating-point data before output. The data difference before and after quantization and dequantization is the quantization error.

2.2 Environment Dependency

- Linux compilation environment: Ubuntu 14.04, GCC 4.8.5
- Windows compilation environment: Windows 10, VS2015
- CUDA 8.0 and later

NOTE

CUDA is required by the GFPQ library of the GPU version, not the CPU version.

2.3 File Description

After **HiSVP_PC_Vx.x.x.x/tools/gfpq_lib/GFPQLibs_cuda8.tgz** is decompressed, the directory structure is as follows:

```
GFPQLibs
├── include
|   └── gfpq.hpp
├── lib
|   ├── gfpq.dll
|   ├── gfpq_gpu.dll
|   ├── gfpq_gpu.lib
|   ├── gfpq.lib
|   ├─── libgfpq.a
|   ├─── libgfpq_gpu.a
|   ├─── libgfpq_gpu.so
|   └─── libgfpq.so
├─── Release_note.txt
└─── sample
├─── CMakeLists.txt
├─── get_gpu_version.sh
├─── run_linux.sh
├─── run_windows.bat
├─── sample.cpp
├─── sample_on_python_gpu.py
└─── sample_on_python.py
```

- **gfpq.hpp** is the interface header file of the GFPQ library.
- **gfpq.dll** is the dynamic library for Windows.
- **gfpq.lib** is the static library for Windows.
- **libgfpq.a** is the static library for Linux.
- **libgfpq.so** is the soft link of the dynamic library for Linux, pointing to **libgfpq_gpu.so.x.x.x**.
- **_gpu** is the GPU version flag.
- **Release_note.txt** is the release description of the GFPQ library.
- The **sample** directory contains GFPQ library samples.

2.4 Interface Description

- HI_GFPQ_QuantAndDeQuant(): It is the quantization and dequantization interface for the CPU version. It supports 8-/16-bit quantization for floating-point and double-precision floating-point data. The input data is the memory allocated by the CPU.
- HI_GFPQ_QuantAndDeQuant_GPU(): It is the quantization and dequantization interface for the GPU version. The input data is the memory allocated by the GPU.
- HI_GFPQ_QuantAndDeQuant_PY(): It is the Python or C interface for the CPU version. It only supports floating-point data.
- HI_GFPQ_QuantAndDeQuant_GPU_PY(): It is the Python or C interface for the GPU version. It only supports floating-point data.
- HI_GFPQ_GetInfo(): It is used to obtain the version number and compilation time of the GFPQ library.
- GFPQ_PARAM_ST: It indicates the quantization mode and quantization coefficients.
  - GFPQ_MODE_INIT: It indicates that no quantization coefficient is configured for quantization. The interface returns the quantized coefficients generated from the input data.

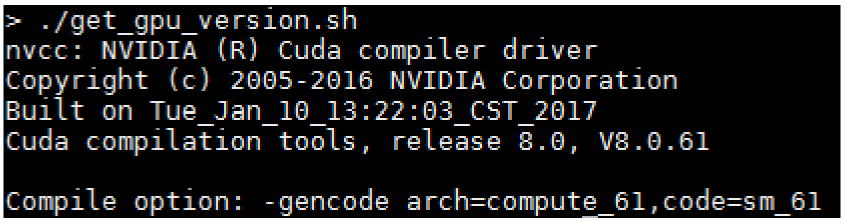
- GFPQ_MODE_UPDATE: It indicates that a quantization coefficient is configured for quantization. The quantization coefficient generated by the interface is compared with the configured quantization coefficient, and the quantization coefficient with a larger coverage is used and returned.
- GFPQ_MODE_APPLY_ONLY: It indicates that no quantization coefficient is generated, and the configured quantization coefficients are used for quantization and dequantization.

2.5 Usage of the GFPQ Library (Linux)

2.5.1 Checking the CUDA Version

Go to the **sample** directory and run the **get_gpu_version.sh** command. **Figure 2-1** indicates that the server supports CUDA and the GPU architecture is sm_61.

Figure 2-1 Checking the GPU version



```
> ./get_gpu_version.sh
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2016 NVIDIA Corporation
Built on Tue_Jan_10_13:22:03_CST_2017
Cuda compilation tools, release 8.0, V8.0.61

Compile option: -gencode arch=compute_61,code=sm_61
```

The GFPQ library supports the following GPU architectures: sm_30, sm_35, sm_37, sm_50, sm_52, sm_60, and sm_61.

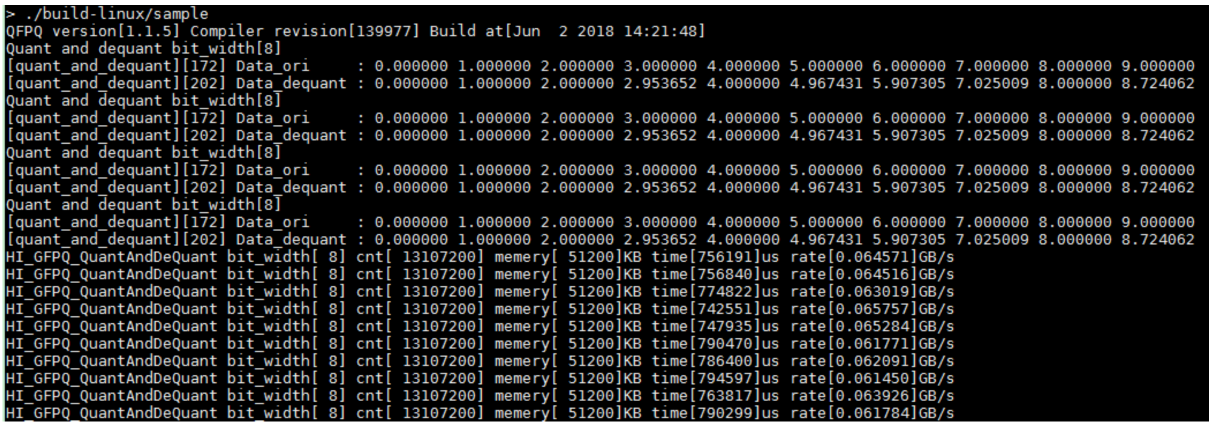
2.5.2 C++ Sample

Decompress the GFPQ library, go to the **sample** directory, and run **run_linux.sh** to generate the **build-linux/sample** file. If the CUDA driver is installed, the **sample_gpu** file is also generated.

Run the sample and check the GFPQ library. As shown in **Figure 2-2**, the version number and compilation time of the GFPQ library, and the quantization result and performance of the sample data are displayed.

For details about the code, see **sample.cpp**.

Figure 2-2 Sample running result



```
> ./build-linux/sample
QFPQ version[1.1.5] Compiler revision[139977] Build at[Jun  2 2018 14:21:48]
Quant and dequant bit_width[8]
[quant_and_dequant][172] Data_ori     : 0.000000 1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.000000 9.000000
[quant_and_dequant][202] Data_dequant : 0.000000 1.000000 2.000000 2.953652 4.000000 4.967431 5.907305 7.025009 8.000000 8.724062
Quant and dequant bit_width[8]
[quant_and_dequant][172] Data_ori     : 0.000000 1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.000000 9.000000
[quant_and_dequant][202] Data_dequant : 0.000000 1.000000 2.000000 2.953652 4.000000 4.967431 5.907305 7.025009 8.000000 8.724062
Quant and dequant bit_width[8]
[quant_and_dequant][172] Data_ori     : 0.000000 1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.000000 9.000000
[quant_and_dequant][202] Data_dequant : 0.000000 1.000000 2.000000 2.953652 4.000000 4.967431 5.907305 7.025009 8.000000 8.724062
Quant and dequant bit_width[8]
[quant_and_dequant][172] Data_ori     : 0.000000 1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.000000 9.000000
[quant_and_dequant][202] Data_dequant : 0.000000 1.000000 2.000000 2.953652 4.000000 4.967431 5.907305 7.025009 8.000000 8.724062
HI_GFPQ_QuantAndDeQuant bit_width[ 8] cnt[ 13107200] memery[ 51200]KB time[756191]us rate[0.064571]GB/s
HI_GFPQ_QuantAndDeQuant bit_width[ 8] cnt[ 13107200] memery[ 51200]KB time[756840]us rate[0.064516]GB/s
HI_GFPQ_QuantAndDeQuant bit_width[ 8] cnt[ 13107200] memery[ 51200]KB time[774822]us rate[0.063019]GB/s
HI_GFPQ_QuantAndDeQuant bit_width[ 8] cnt[ 13107200] memery[ 51200]KB time[742551]us rate[0.065757]GB/s
HI_GFPQ_QuantAndDeQuant bit_width[ 8] cnt[ 13107200] memery[ 51200]KB time[747935]us rate[0.065284]GB/s
HI_GFPQ_QuantAndDeQuant bit_width[ 8] cnt[ 13107200] memery[ 51200]KB time[790470]us rate[0.061771]GB/s
HI_GFPQ_QuantAndDeQuant bit_width[ 8] cnt[ 13107200] memery[ 51200]KB time[786400]us rate[0.062091]GB/s
HI_GFPQ_QuantAndDeQuant bit_width[ 8] cnt[ 13107200] memery[ 51200]KB time[794597]us rate[0.061450]GB/s
HI_GFPQ_QuantAndDeQuant bit_width[ 8] cnt[ 13107200] memery[ 51200]KB time[763817]us rate[0.063926]GB/s
HI_GFPQ_QuantAndDeQuant bit_width[ 8] cnt[ 13107200] memery[ 51200]KB time[790299]us rate[0.061784]GB/s
```

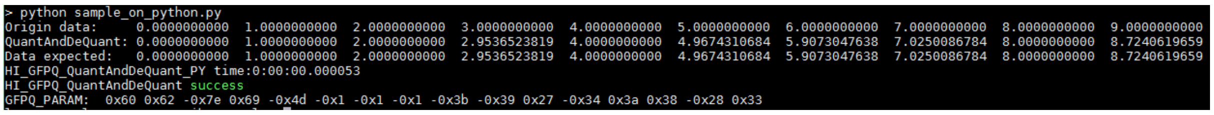
NOTICE

- The Linux version must be decompressed on the Linux OS because the GFPQ library contains a soft link. If the soft link is decompressed on Windows, the soft link file format may be damaged. In this case, the GFPQ library is unavailable.
- **run_linux.sh** of the sample depends on CMake 2.6 or later.

2.5.3 Python Sample

Run **sample_on_python.py**, as shown in **Figure 2-3**.

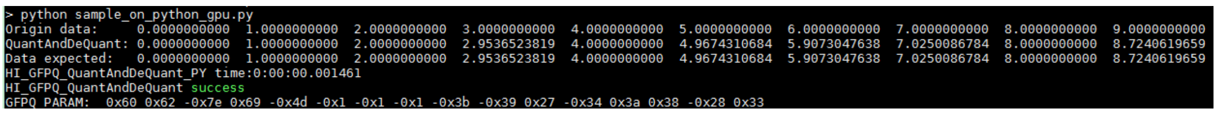
Figure 2-3 sample_on_python.py running result



```
> python sample_on_python.py
Origin data:     0.0000000000  1.0000000000  2.0000000000  3.0000000000  4.0000000000  5.0000000000  6.0000000000  7.0000000000  8.0000000000  9.0000000000
QuantAndDeQuant: 0.0000000000  1.0000000000  2.0000000000  2.9536523819  4.0000000000  4.9674310684  5.9073047638  7.0250086784  8.0000000000  8.7240619659
Data expected:   0.0000000000  1.0000000000  2.0000000000  2.9536523819  4.0000000000  4.9674310684  5.9073047638  7.0250086784  8.0000000000  8.7240619659
HI_GFPQ_QuantAndDeQuant_PY time:0:00:00.000053
HI_GFPQ_QuantAndDeQuant success
GFPQ_PARAM:  0x60 0x62 -0x7e 0x69 -0x4d -0x1 -0x1 -0x1 -0x3b -0x39 0x27 -0x34 0x3a 0x38 -0x28 0x33
```

Run **sample_on_python_gpu.py**, as shown in **Figure 2-4**.

Figure 2-4 sample_on_python_gpu.py running result

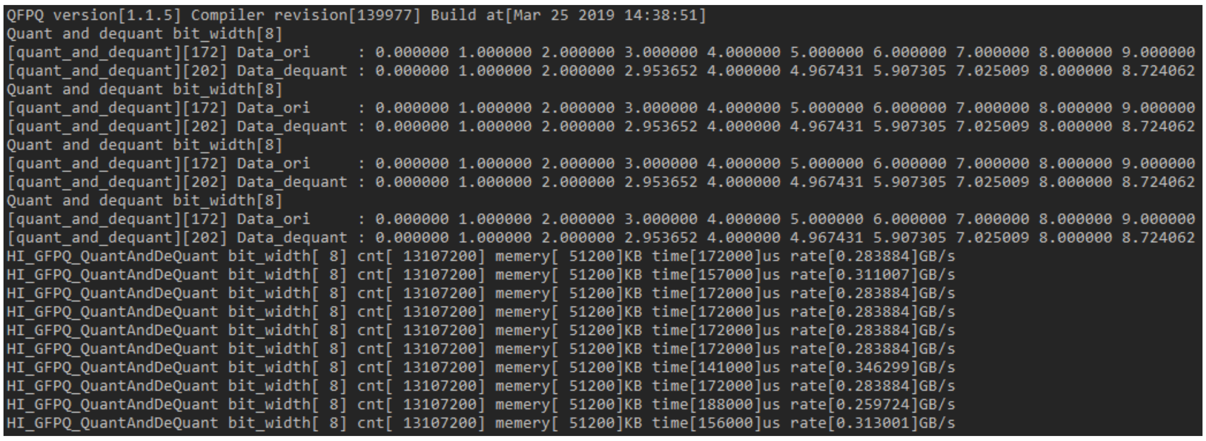


```
> python sample_on_python_gpu.py
Origin data:     0.0000000000  1.0000000000  2.0000000000  3.0000000000  4.0000000000  5.0000000000  6.0000000000  7.0000000000  8.0000000000  9.0000000000
QuantAndDeQuant: 0.0000000000  1.0000000000  2.0000000000  2.9536523819  4.0000000000  4.9674310684  5.9073047638  7.0250086784  8.0000000000  8.7240619659
Data expected:   0.0000000000  1.0000000000  2.0000000000  2.9536523819  4.0000000000  4.9674310684  5.9073047638  7.0250086784  8.0000000000  8.7240619659
HI_GFPQ_QuantAndDeQuant_PY time:0:00:00.001461
HI_GFPQ_QuantAndDeQuant success
GFPQ PARAM:  0x60 0x62 -0x7e 0x69 -0x4d -0x1 -0x1 -0x1 -0x3b -0x39 0x27 -0x34 0x3a 0x38 -0x28 0x33
```

2.6 Usage of the GFPQ Library (Windows)

Decompress the GFPQ library package, open the DOS command prompt, go to the **sample** directory, and run **run_windows.bat** to generate the **build-windows\Release\sample.exe** file. If the CUDA driver is supported, **sample_gpu.exe** is also generated. **Figure 2-5** shows the running result of **sample.exe**.

Figure 2-5 sample.exe running result



```
QFPQ version[1.1.5] Compiler revision[139977] Build at[Mar 25 2019 14:38:51]
Quant and dequant bit_width[8]
[quant_and_dequant][172] Data_ori     : 0.000000 1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.000000 9.000000
[quant_and_dequant][202] Data_dequant : 0.000000 1.000000 2.000000 2.953652 4.000000 4.967431 5.907305 7.025009 8.000000 8.724062
Quant and dequant bit_width[8]
[quant_and_dequant][172] Data_ori     : 0.000000 1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.000000 9.000000
[quant_and_dequant][202] Data_dequant : 0.000000 1.000000 2.000000 2.953652 4.000000 4.967431 5.907305 7.025009 8.000000 8.724062
Quant and dequant bit_width[8]
[quant_and_dequant][172] Data_ori     : 0.000000 1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.000000 9.000000
[quant_and_dequant][202] Data_dequant : 0.000000 1.000000 2.000000 2.953652 4.000000 4.967431 5.907305 7.025009 8.000000 8.724062
Quant and dequant bit_width[8]
[quant_and_dequant][172] Data_ori     : 0.000000 1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.000000 9.000000
[quant_and_dequant][202] Data_dequant : 0.000000 1.000000 2.000000 2.953652 4.000000 4.967431 5.907305 7.025009 8.000000 8.724062
HI_GFPQ_QuantAndDeQuant bit_width[ 8] cnt[ 13107200] memery[ 51200]KB time[172000]us rate[0.283884]GB/s
HI_GFPQ_QuantAndDeQuant bit_width[ 8] cnt[ 13107200] memery[ 51200]KB time[157000]us rate[0.311007]GB/s
HI_GFPQ_QuantAndDeQuant bit_width[ 8] cnt[ 13107200] memery[ 51200]KB time[172000]us rate[0.283884]GB/s
HI_GFPQ_QuantAndDeQuant bit_width[ 8] cnt[ 13107200] memery[ 51200]KB time[172000]us rate[0.283884]GB/s
HI_GFPQ_QuantAndDeQuant bit_width[ 8] cnt[ 13107200] memery[ 51200]KB time[172000]us rate[0.283884]GB/s
HI_GFPQ_QuantAndDeQuant bit_width[ 8] cnt[ 13107200] memery[ 51200]KB time[172000]us rate[0.283884]GB/s
HI_GFPQ_QuantAndDeQuant bit_width[ 8] cnt[ 13107200] memery[ 51200]KB time[141000]us rate[0.346299]GB/s
HI_GFPQ_QuantAndDeQuant bit_width[ 8] cnt[ 13107200] memery[ 51200]KB time[172000]us rate[0.283884]GB/s
HI_GFPQ_QuantAndDeQuant bit_width[ 8] cnt[ 13107200] memery[ 51200]KB time[188000]us rate[0.259724]GB/s
HI_GFPQ_QuantAndDeQuant bit_width[ 8] cnt[ 13107200] memery[ 51200]KB time[156000]us rate[0.313001]GB/s
```

NOTICE

sample run_windows.bat of the sample depends on CMake 2.6 or later and VS2015 or later.

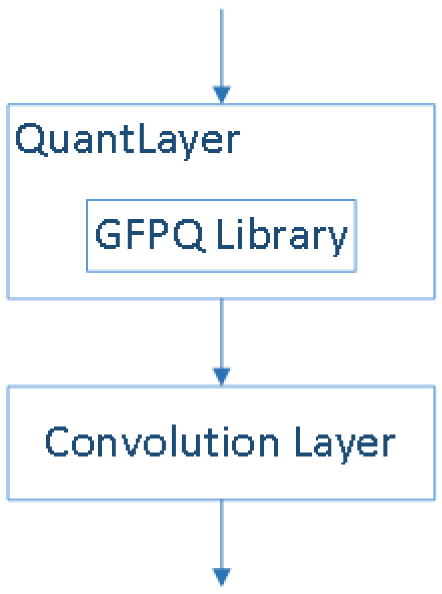
3 Integrating the GFPQ Library in Caffe

During the fine-tuning of the quantization model in Caffe, the data and weights in the network need to be quantized and dequantized.

3.1 Data Quantization

When a quantization layer needs to be added, the interface of the GFPQ library is called at the quantization layer for data quantization and dequantization.

Figure 3-1 Position of the quantization layer



3.1.1 Downloading and Compiling Caffe

Download the official Caffe source code from **https://github.com/BVLC/caffe** and do not integrate the GFPQ library so that the native Caffe code can be compiled.

3.1.2 Adding the Quantization Layer QuantLayer

For details about adding a quantization layer in Caffe, see the **caffe_patch.tgz** package. The following files need to be added:

- include/caffe/layers/quant_layer.hpp
- src/caffe/layers/quant_layer.cpp
- src/caffe/layers/quant_layer.cu

The files to be modified include the **src/caffe/proto/caffe.proto** file in the native Caffe environment. Modify the file in incremental mode and add the parameter settings related to QuantLayer.

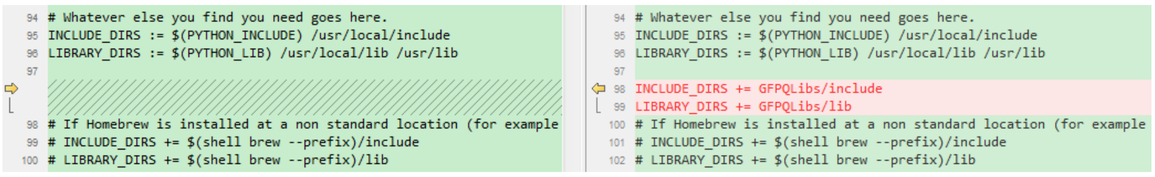
To add a quantization layer to the .prototxt file of Caffe, add the following content:

```
layer {
  name: 'quant_layer_sample'
  type: 'Quant'
  bottom: 'data'
  top: ' quant_layer_sample '
  quant_param {
        quant_bits: 8
        quant_en: 1
  }
}
```

3.1.3 Linking the GFPQ Library

- Decompress the GFPQ library package to the Caffe directory and name it **GFPQLibs**.
- Modify the **Makefile.config** file. After the initial value of **LIBRARY_DIRS** is set, include the header file and library paths of **GFPQLibs**, as shown in Figure 3-2.

Figure 3-2 Modifying Makefile.config

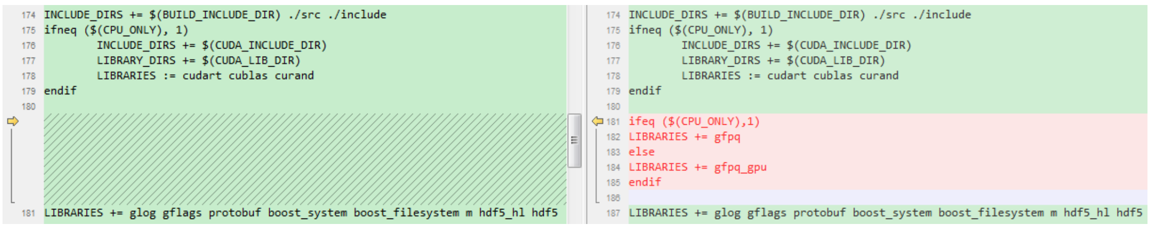


```
# Whatever else you find you need goes here.
INCLUDE_DIRS := $(PYTHON_INCLUDE) /usr/local/include
LIBRARY_DIRS := $(PYTHON_LIB) /usr/local/lib /usr/lib

# If Homebrew is installed at a non standard location (for example
# INCLUDE_DIRS += $(shell brew --prefix)/include
# LIBRARY_DIRS += $(shell brew --prefix)/lib
```

```
# Whatever else you find you need goes here.
INCLUDE_DIRS := $(PYTHON_INCLUDE) /usr/local/include
LIBRARY_DIRS := $(PYTHON_LIB) /usr/local/lib /usr/lib

INCLUDE_DIRS += GFPQLibs/include
LIBRARY_DIRS += GFPQLibs/lib
# If Homebrew is installed at a non standard location (for example
# INCLUDE_DIRS += $(shell brew --prefix)/include
# LIBRARY_DIRS += $(shell brew --prefix)/lib
```

- Modify **Makefile** and link the GFPQ library, as shown in Figure 3-3.

Figure 3-3 Modifying Makefile — 1

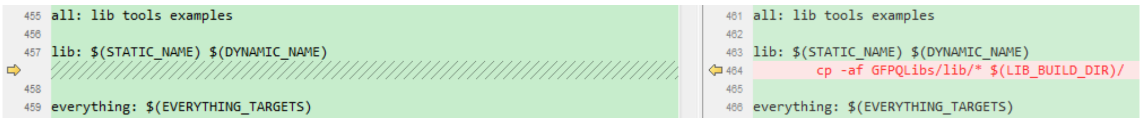


```
INCLUDE_DIRS += $(BUILD_INCLUDE_DIR) ./src ./include
ifneq ($(CPU_ONLY), 1)
        INCLUDE_DIRS += $(CUDA_INCLUDE_DIR)
        LIBRARY_DIRS += $(CUDA_LIB_DIR)
        LIBRARIES := cudart cublas curand
endif

LIBRARIES += glog gflags protobuf boost_system boost_filesystem m hdf5_hl hdf5
```

```
INCLUDE_DIRS += $(BUILD_INCLUDE_DIR) ./src ./include
ifneq ($(CPU_ONLY), 1)
        INCLUDE_DIRS += $(CUDA_INCLUDE_DIR)
        LIBRARY_DIRS += $(CUDA_LIB_DIR)
        LIBRARIES := cudart cublas curand
endif

ifeq ($(CPU_ONLY),1)
LIBRARIES += gfpq
else
LIBRARIES += gfpq_gpu
endif

LIBRARIES += glog gflags protobuf boost_system boost_filesystem m hdf5_hl hdf5
```

- Modify **Makefile** and copy the GFPQ library to the **build** directory of Caffe, as shown in Figure 3-4.

Figure 3-4 Modifying Makefile — 2



```
all: lib tools examples                        all: lib tools examples

lib: $(STATIC_NAME) $(DYNAMIC_NAME)            lib: $(STATIC_NAME) $(DYNAMIC_NAME)
                                                       cp -af GFPQLibs/lib/* $(LIB_BUILD_DIR)/

everything: $(EVERYTHING_TARGETS)              everything: $(EVERYTHING_TARGETS)
```

- Run the **make clean; make pycaffe -j** command to recompile the Caffe code.

3.2 Weight Quantification

The fine-tuning script needs to be manually modified for weight quantization according to the fine-tuning environment.

3.2.1 Weight Quantification Process

The fine-tuning process of a convolutional neural network (CNN) is as follows:

Step 1 Randomly initialize the weights of the network.

Step 2 Perform forward propagation on the network. That is, the input data is forward propagated trough the network layer by layer (including the convolutional, BatchNorm, Scale, activation function, and pooling layers) to give the final output.

Step 3 Calculate the error between the network output and the expected value (ground truth).

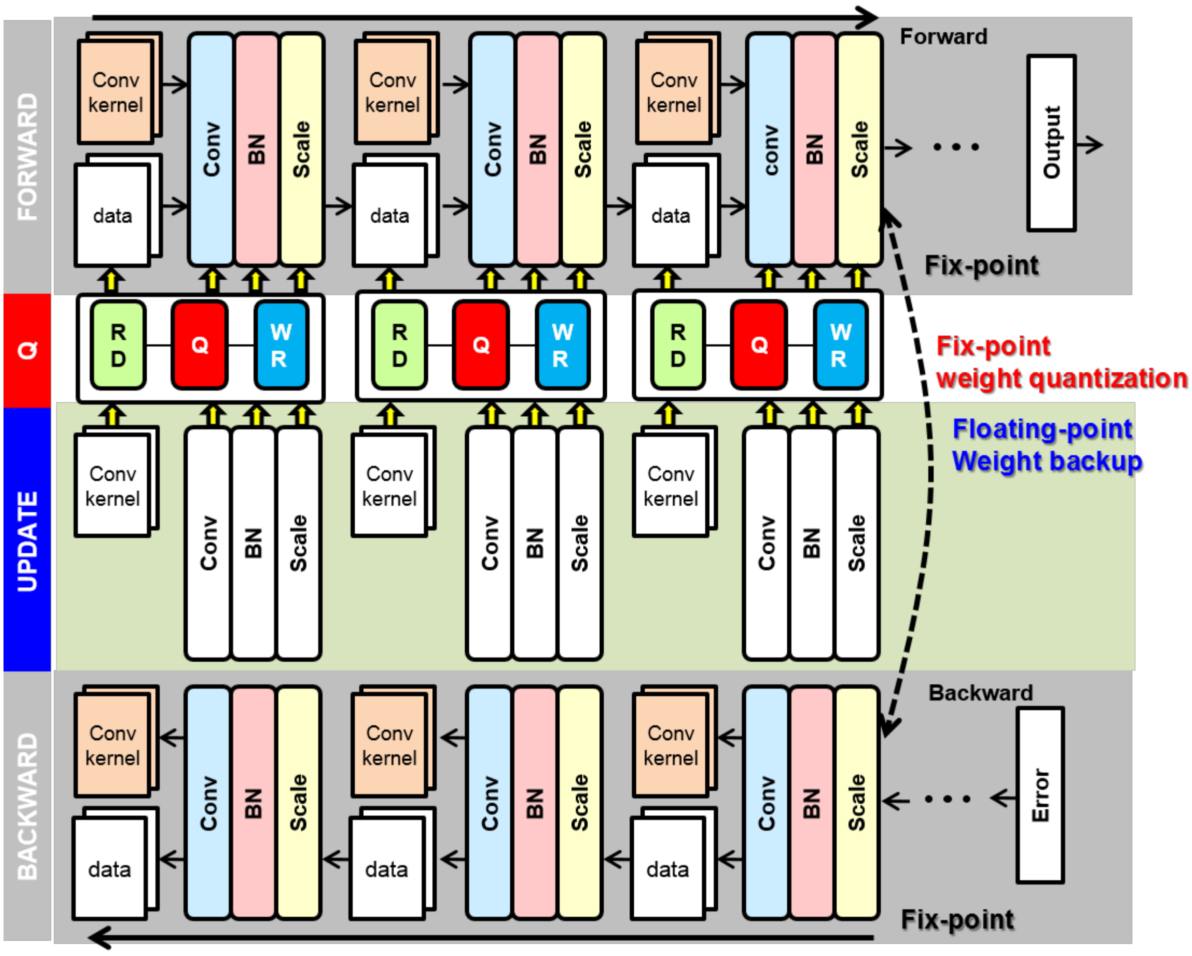
Step 4 When the error is greater than expected, back-propagate the error through the network layer by layer (including the pooling, activation function, Scale, BatchNorm, and convolutional layers), and obtain the error at each layer.

Step 5 Calculate the gradient of each layer according to the error of each layer, and then update the parameters of specific layers (for example, the weight of the convolutional layer) based on the gradients.

Step 6 Repeat **Step 2** to **Step 5** until the network converges to the target.

----End

Figure 3-5 Weight quantification process



Among the optimization algorithms of deep learning, a typical weight optimization approach is the momentum update algorithm based on stochastic gradient descent. The equation is as follows:

$$w_{iter+1} = w_{iter} + \Delta history_{iter+1}$$

where, $\Delta history$ indicates the accumulated historical gradients in network fine-tuning to the current iteration:

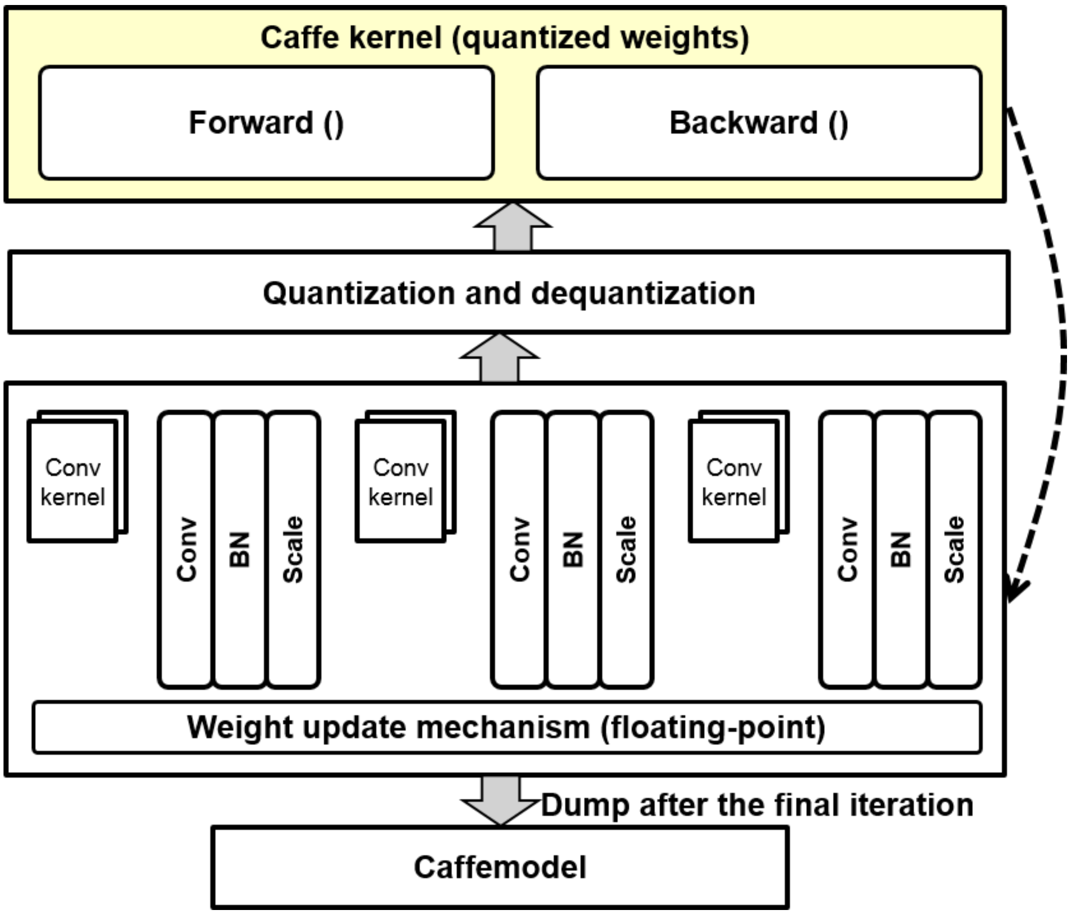
$$\Delta_{iter+1} = momentum \cdot \Delta history_{iter} + learnRate \cdot \frac{\partial loss}{\partial w}$$

3.2.2 Quantization with Weight Update in Fine-tuning

Weight quantization is a process of approximating a continuous set of weight values to a finite set of discrete weight values. The quantization precision is in *n*-bit (binary) format. The precision of the NNIE fixed-point operations is 8-bit. Quantization, each amplitude value is mapped into an approximation from a finite set of discrete values.

From the perspective of CNN fine-tuning (as described in **3.2.1 Weight Quantification Process**), in each fine-tuning iteration, the weight of each layer is adjusted in a gradual manner according to a preset learning rate. When the learning rate or the gradient of Caffe layers is small, the weight adjustment amplitude in each iteration may be smaller than the quantization step. If the updated weights obtained after each iteration are quantized and then used to rewrite the CNN model, the weight values of the model do not change and the convergence becomes impossible.

Figure 3-6 Fine-tuning process

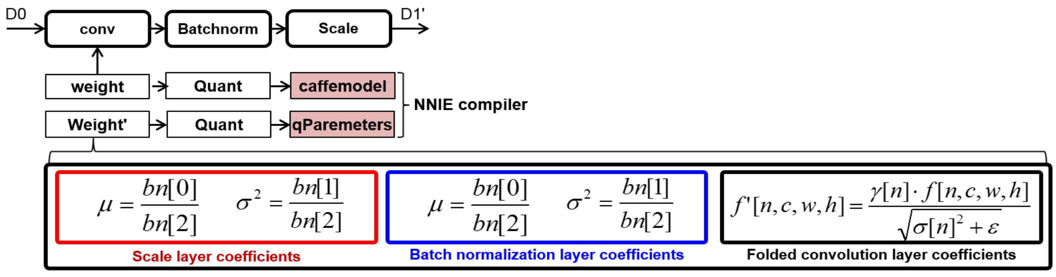

Caffe kernel (quantized weights)
Forward ()
Backward ()
Quantization and dequantization
Conv kernel
Conv
BN
Scale
Conv kernel
Conv
BN
Scale
Conv kernel
Conv
BN
Scale
Weight update mechanism (floating-point)
Dump after the final iteration
Caffemodel

Therefore, a floating-point Caffe model needs to be backed up for the network fine-tuning. All the weight updates in fine-tuning are performed on the backup floating-point model. The floating-point model is used for network quantization and forward and backward propagation, as shown in the figure above.

3.2.3 Weight Quantization in Layer-Folding

During network fine-tuning, in most cases (especially for complex networks), the convolutional layer is followed by the BatchNorm and the Scale layers. A normalized feature map can facilitate network convergence. However, in actual inference, to reduce the calculation pressure of the board, the convolutional, BatchNorm, and Scale layers are folded in mathematics.

Figure 3-7 Folding the BatchNorm and Scale layers


D0
conv
Batchnorm
Scale
D1'
weight
Quant
caffemodel
Weight'
Quant
qParemeters
NNIE compiler
$\mu = \frac{bn[0]}{bn[2]}$ $\sigma^2 = \frac{bn[1]}{bn[2]}$
Scale layer coefficients
$\mu = \frac{bn[0]}{bn[2]}$ $\sigma^2 = \frac{bn[1]}{bn[2]}$
Batch normalization layer coefficients
$f'[n,c,w,h] = \frac{\gamma[n]\cdot f[n,c,w,h]}{\sqrt{\sigma[n]^2+\varepsilon}}$
Folded convolution layer coefficients

In the network quantization and fine-tuning, pay special attention to the floating-point and fixed-point hybrid operations. The NNIE performs fixed-point operations on the network layers that require a large number of multiplication operations such as the convolutional and fully connected layers. Fewer multiplication operations are performed on BatchNorm and Scale layers. Therefore, these operations are implemented with high precision by the NNIE. In network fine-tuning involving the BatchNorm and Scale layers, the convolutional layer is quantized, and FP32 operations are performed on BatchNorm and Scale layers. After convergence, a Caffe model (.caffemodel file) is obtained. In dumping of the Caffe model, the weights of the convolutional, BatchNorm , and Scale layers are quantized to obtain the quantization parameter **qParameters**. The generated .caffemodel file and **qParameter** parameter serve as the knowledge base for the NNIE compiler.

3.3 Fine-Tuning

This topic describes the building of the NNIE fine-tuning environment and fine-tuning methods.

3.3.1 Building the Fine-Tuning Environment

The fine-tuning environment can be a floating-point environment. Build the environment as follows:

Step 1 Integrate the NNIE GFPQ library into the Caffe environment by referring to the preceding topics.

Step 2 Recompile the Caffe environment.

Step 3 Add a quantization layer (QuantLayer) to the .prototxt files for network fine-tuning and testing respectively according to **3.1 Data Quantization**.

Step 4 Compile the fine-tuning script according to **3.2.1 Weight Quantification Process** and **3.2.2 Quantization with Weight Update in Fine-tuning**. For details, see the quantization fine-tuning case released by HiSilicon.

Step 5 Use a smaller learning rate to fine-tune the network. The convergent network model obtained by floating-point fine-tuning is the initial model.

----End

3.3.2 Fine-Tuning Methods and Precautions

Build the environment and compile the fine-tuning script according to this document. Generally, the precision loss caused by quantization can be restored to that of the floating-point model by fine-tuning. If the precision of the quantization model fails to be restored after fine-tuning, or the restored precision still differs greatly from that of the floating-point model, you can try the following methods:

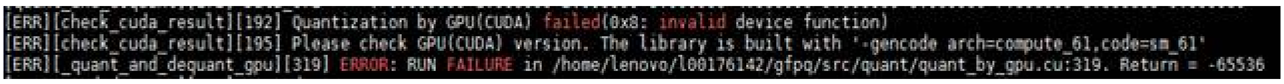
1. Quantize only the data instead the weights. If the precision is restored to that of the floating-point model, it can be determined that the problem is caused by the script.
2. Start from the data layer, compare the feature map of the floating-point output with the feature map of the fine-tuned quantized output, and analyze the network layers of low similarities one by one (for example, a cosine similarity lower than 0.98).
3. In the weight update part of the fine-tuning script, exclude the BatchNorm and quantization layers because the parameters of these layers are obtained from statistics, not learning. Therefore, no update is required.

4 FAQs

4.1 GFPQ Library Error

The following GFPQ library error is reported: **Quantization by GPU(CUDA) failed(0x8: invalid device function)**

Figure 4-1 GFPQ library error



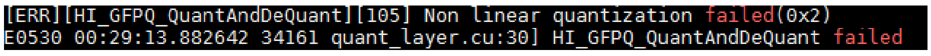
```
[ERR][check_cuda_result][192] Quantization by GPU(CUDA) failed(0x8: invalid device function)
[ERR][check_cuda_result][195] Please check GPU(CUDA) version. The library is built with '-gencode arch=compute_61,code=sm_61'
[ERR][_quant_and_dequant_gpu][319] ERROR: RUN FAILURE in /home/lenovo/l00176142/gfpq/src/quant/quant_by_gpu.cu:319. Return = -65536
```

- Cause analysis: The GFPQ library of the GPU version does not support the server GPU architectures. It only supports only the following architectures: sm_30, sm_35, sm37, sm_50, sm_52, sm_60, and sm61.
- Solution: Use the library in **GFPQLibs_xxx.tgz** and run **get_gpu_version.sh** to confirm the CUDA gencode of the server.

NOTICE

In the sample of network fine-tuning in Caffe, the **GFPQLibs** file may not be the latest. Decompress **GFPQLibs.tgz** to the Caffe directory.

4.2 GFPQ Library Calling Error During Fine-Tuning



```
[ERR][HI_GFPQ_QuantAndDeQuant][105] Non linear quantization failed(0x2)
E0530 00:29:13.882642 34161 quant_layer.cu:30] HI_GFPQ_QuantAndDeQuant failed
```

- Cause analysis: The GFPQ library interfaces are misused.
- Solution: Check the parameters in the codes for calling the quantization interfaces in **caffe/src/caffe/layers/quant_layer.cpp**, **caffe/src/caffe/layers/quant_layer.cu**, and **wkQuant.py**. Run the sample code to ensure that the sample code can be properly executed. Then, modify the error code by referring to the sample code.

NOTICE

Transfer the memory allocated by the CPU to the CPU interface, and transfer the memory allocated by the GPU to the GPU interface.