MLPerf Habana Benchmark Application

Revision 0.1

October 2019

Contents

List of Tables

No table of figures entries found.

List of Figures

1. Audience

The documents should be used by Habana customers who intend to run the MLPerf benchmark on the Habana Goya HW accelerator. It describes the connections between the different SW and HW blocks that compose the running environment.

2. Test Environment

The following test environment was used:

- HW
  - Host HW
    - CPU - Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz (2 sockets)
    - Core count – 20
    - RAM – 64GB
    - HD – SSD 1TB
  - Goya – single card with 16GB onboard memory
- SW
  - OS - Ubuntu 16.04
  - Compiler - GCC 5.4 or CLANG8
  - Python 3.6
  - Onnx package - 1.4.1
  - Numpy package - 1.16
  - Habana tool set - V0.2.0

3. Habana MLPerf benchmark environment settings

Figure 1 describes the SW and HW environment needed to build and run the Habana MLPerf benchmark application. Except for Loadgen library, that comes as a part of MLPerf collateral, all other components supplied by Habana.

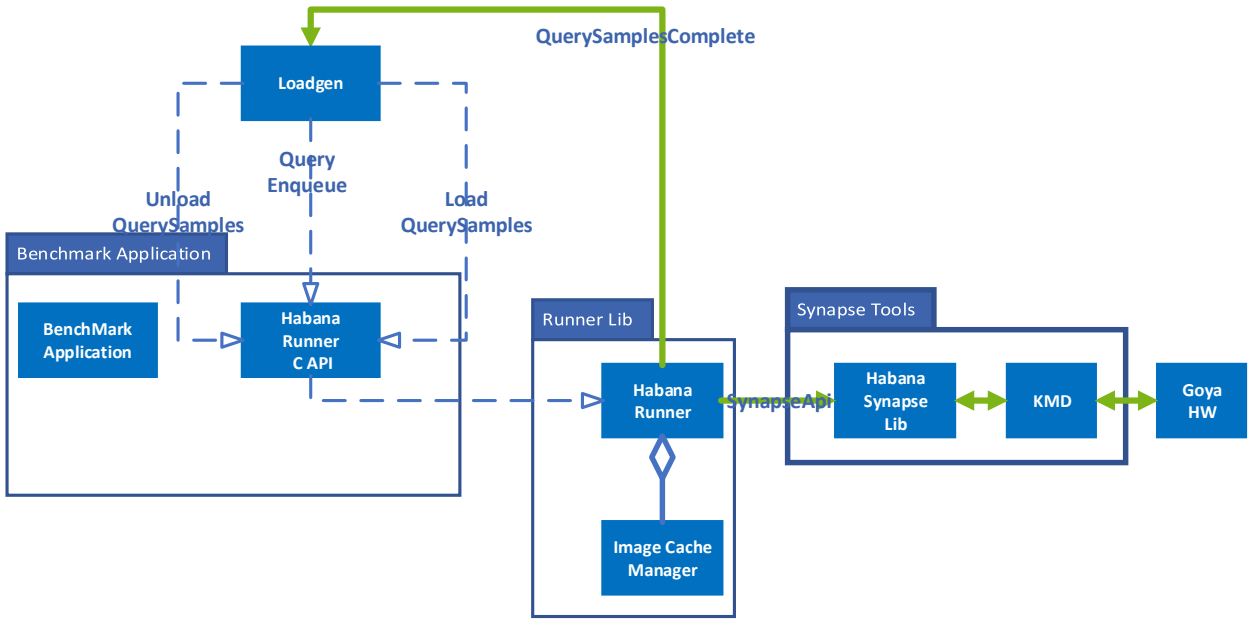

Figure 1- Habana MLPerf benchmark setup

3.1. Loadgen

The Habana benchmark application links a static library generated from the MLPerf Loadgen code. The Loadgen code is responsible for running and synchronizing the test and calling the supplied Habana callbacks

- EqueueQuery
- LoadQueySamples
- UnloadQuerySamples

3.2. Habana Runner

The HabanaRunner is the main engine that drives the benchmark application. It performs the following tasks:

- Implements the needed callbacks of the Loadgen.
- Holds image cache manager that is responsible for image load/unload.
- Interacts with Habana SynapseAI library to allocate needed memory, enqueue tasks and get outputs from the Goya card.

- Calls *MLPerf ::QuerySamplesComplete()* from the Loadgen library, as an indication that the runner has a new result received from the Goya HW via SynapseAI API. Issuing this call marks the query as done and results are available to be used by Loadgen logger.

The runner is implemented hierarchically, to address all requirements stemming from MLPerf inference scenarios (SingleStream, MultiStream, Server and Offline). The implementation uses C++ inheritance to address the requirements for all the scenarios, ensure robustness and avoid code replication. Basic SW architecture is shown in Figure 2.

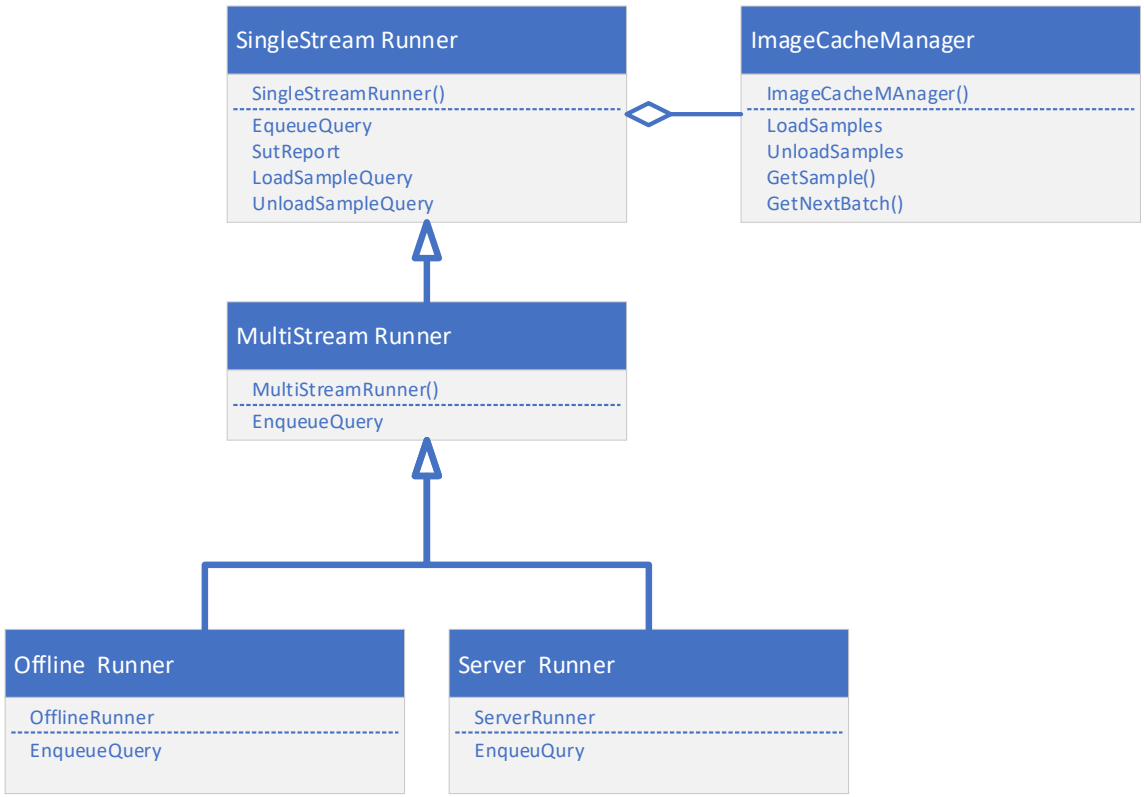

Figure 2- Runner inheritance scheme

3.3. Image Cache Manager

Image cache manager class is responsible for allocating memory for the image cache (in host memory), and to load/unload images to and from host memory. Loading/unloading is synchronized by Loadgen, the images are loaded from disk according to the *sampleIdx* provided by Loadgen.

3.4. SynapseAI Library

SynapseAI library implements the SynapseAI API, which allows the host application to interact with Goya card by compiling and/or running compiled recipes (“programs” that encapsulate optimized DL topologies suitable for running on Goya cards).

3.5. Benchmark Application

The benchmark application implements the main code flow as demonstrated in Figure 3.

- INI and MLPerf files initializations

The main functions start by reading an ini files containing needed parameters from benchmark code initialization, the same function that reads the benchmark data also initializes the TestSettings structure from data taken from the following configuration files:

1. MLPerf .conf – general configuration file
2. user.conf – user specific configuration file (Habana specific)
3. audit.conf – to be used for audit runs, for example enable accuracy sampling during performance runs.

The INI files contain general setup parameters, e.g. image directory location, multithreaded scheme (if applicable), number of images in the DB etc.

- Construction order:
    1. Habana runner allocation.
    2. SUT Allocation
    3. SQL

- Destruction order
    4. QSL
    5. SUT
    6. Habana runner

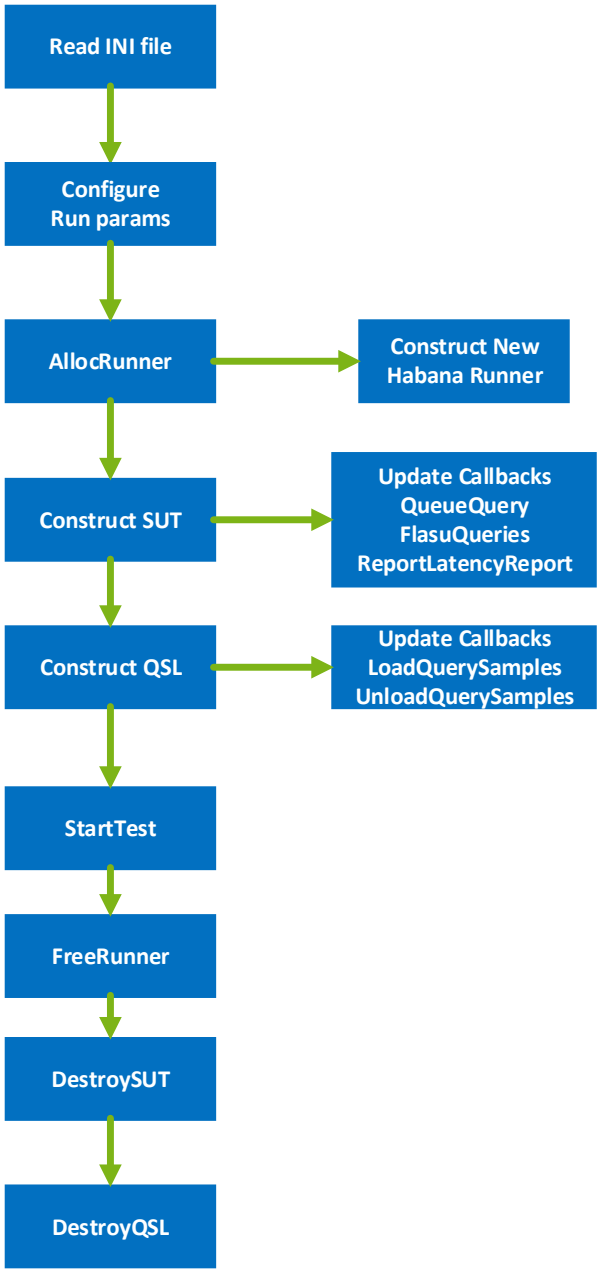

Figure 3- Benchmark code flow

3.5.1. Habana Runner C API

Implementation of plain C API, the implemented API functions are passed to the Loadgen as callback functions during construction of SUT and SQL, the following function (callbacks) are implemented:

- runnerAllocRunner
  the function constructs a Habana runner as needed by the scenario been used (SingleStream, MultiStream, Server and Offline) as part of the Habana runner construction the internal image cache is constructed and initialized.

```
void  runnerAllocRunner(mlperf::TestScenario                          runnerType,
                        const std::string                             recipeName,
                        const std::string                             imageDirePath,
                        const std::string                             imagListFile,
                        uint64_t                                      maxNumquerySamples,
                        uint64_t                                      numberOFImagesToLoad,
                        uint64_t                                      batchSize,
                        std::vector<std::unique_ptr<baseFunctor>>     &outputTransProc,
                        bool                                          enableOutputDump,
                        uint32_t                                      outputDumpHaltCnt,
                        uint32_t                                      numOfTasks,
                        uint32_t                                      breakBatchSize,
                        std::chrono::duration<int,std::micro>         expectedImageProc,
                        std::chrono::duration<int,std::micro>         procSlotDurationTime,
                        bool                                          isEnforcedResBuf=false,
                        uint64_t                                      enforcedResBuf = 0);
```

- runnerLoadSamplesToRam
  the function is been called by the Loadgen whenever there is a need to load images to the host memory (RAM) for the hard drive. The function is redirected to the Image cache manager load function which load images from the hard drive using the supplied path to a directory containing preprocessed images. Each image is loaded to memory to a specific location as designated by the query index buffer pointed by QuerySampleIndex (queryIndex is been used a and index to get the address of the loaded image in the input buffer allocated in the RAM).

```
void      runnerLoadSamplesToRam(uintptr_t clientData, const mlperf::QuerySampleIndex*, size_t);
```

- runnerUnloadSamplesFromRam – the function marks all loaded images in RAM as unusable.

```
Void runnerUnloadSamplesFromRam(uintptr_t clientData, const mlperf::QuerySampleIndex*, size_t);
```

- runnerQueueQuery
  the function is been called by Loadgen whenever there is a new task to be sent to the Habana runner.

```
void      runnerQueueQuery(uintptr_t ClientData, const mlperf::QuerySample*, size_t);
```

  The function calls the Habana runner queueQuery method implementation, user should note that Habana runner has different implementation each one suitable for a specific scenario

demanded by the MLPerf defined scenarios, each different implementation of Habana runner class has a different implementation of queueQuery function.

```
virtual void       queueQuery(uintptr_t ClientData, const mlperf::QuerySample*, size_t);
```