NPU: an AI example design for Intel® Stratix® 10 NX FPGA

Version 1.1

1. Project Description and Directories

This project is an example design for Intel® Stratix® 10 NX FPGA. The example design implements an AI soft processor called Neural Processing Unit (NPU), which was developed as part of FPGA AI research conducted at the Intel PSG CTO Office.

The repository is organized as follows:

- `compiler`: includes the NPU front-end where the user can write NPU workloads as Tensorflow Keras Sequential models
- `scripts`: includes testing scripts for a set of benchmarking workloads from the FPT'20 paper
- `simulator`: includes the NPU C++ simulator for getting fast NPU performance estimates
- `rtl`: includes the RTL implementation of the NPU hardware for the Stratix 10 NX FPGA
- `patch`: code patches for system-level demonstration, which connects NPU on FPGA to CPU host.

2. NPU installation dependencies

These dependencies need to be installed in the user's system in order to build/simulate NPU.

- Python 3.7.9 or newer
- NumPy 1.15.0 or newer
- TensorFlow 2
- GCC 4.7.2 or newer (to compile the C++ performance simulator)
- Synopsys VCS (for RTL simulation)
- Quartus Prime Pro 20.4 or newer (for RTL simulation of Intel IP cores, bitstream generation)

3. NPU Simulation Flow

The user specifies the NPU workload using a sub-class of the Keras Sequential class. This is a list of layers that are currently supported by the NPU:

- tf.keras.layers.Dense: Fully-connected layer (matrix-vector/matrix-matrix multiplication + activation).
- tf.keras.layers.Embedding: Word embedding for language models (implemented as matrix-vector multiplication on the NPU.
- tf.keras.layers.SimpleRNN: Vanilla recurrent neural network model (2x matrix-vector multiplications + activations).
- tf.keras.layers.GRU: Vanilla gated recurrent unit model (6x matrix-vector multiplications + activations).
- tf.keras.layers.LSTM: Vanilla long short-term memory model (8x matrix-vector multiplications + activations).

Writing AI program for NPU. Users should write their workloads in the provided `compiler/driver.py` file. Example AI programs for testing are also provided. See "NPU Testing Scripts" sub-section below.

Running NPU simulations. There are two types of simulations possible. First, fast performance modeling can be done using C++ simulator that is cycle-approximate to produce quick cycle count estimates. Second, more precise cycle-accurate simulation can be done by simulating the NPU RTL-level implementation.

To run NPU C++ performance simulation and RTL simulation for the specified workload, use the `-perfsim` and `-rtlsim` options, respectively.

```
$ cd compiler
$ python driver.py -perfsim -rtlsim
```

This will compile the specified workload into NPU instructions and simulate it using our C++ performance simulator and also launch a cycle-accurate RTL-level simulation.

If the C++ performance or RTL simulations are stuck, please refer to the `simulator/perf_sim_log` and `rtl/rtl_sim_log` files for a log of errors causing the problem.

NPU Testing Scripts. You can use the provided testing scripts to make sure that any modifications implemented still pass the functionality testing through C++ and RTL simulation. These scripts also provide a comparison to the baseline results of the workloads from the FPT'20 paper.

You can perform the C++ and RTL simulation testing by simply running the `perf_tests.py` and `rtl_tests.py` files from `scripts` directory, respectively.

```
$ cd scripts
$ python perf_tests.py
```

You should see an output similar to the following:

```
WORKLOAD                              TEST    TOPS      QoR
01_gemv_512x512                       PASS    5.65      +0.00%
02_gemv_1024x1024                     PASS    13.06     +0.00%
03_gemv_1152x1152                     PASS    13.38     +0.00%
04_gemv_1536x1536                     PASS    18.34     +0.00%
05_gemv_1792x1792                     PASS    20.04     +0.00%
06_rnn_512_8                          PASS    10.86     +0.00%
07_rnn_1024_8                         PASS    25.11     +0.00%
08_rnn_1152_8                         PASS    25.59     +0.00%
09_rnn_1536_8                         PASS    33.38     +0.00%
10_rnn_1792_8                         PASS    34.25     +0.00%
11_gru_512_8                          PASS    10.89     +0.00%
12_gru_1024_8                         PASS    24.32     +0.00%
13_gru_1152_8                         PASS    25.29     +0.00%
14_lstm_512_8                         PASS    15.16     +0.00%
15_lstm_1024_8                        PASS    31.85     +0.00%
16_mlp5_512                           PASS    6.86      +0.00%
17_mlp5_1024                          PASS    16.29     +0.00%
18_mlp3_1024_512_256_256              PASS    6.02      +0.00%
19_mlp3_1024_512_256_256_batched      PASS    7.26      +0.00%
```

4. Generating NPU Bitstream for Deploying and Testing on an FPGA

We provide bitstream generation flow for testing a standalone NPU on hardware, as follows. For the user-specified workload, the MRF content, inputs, and NPU instructions are all initialized using memory initialization files (MIFs). A hardware self-tester is also provided, which supplies NPU with inputs, launch start signal, receive outputs and compare them against MIF-initialized golden results. The self-tester produces a 3-bit test status that indicates success or failure as follows: (Idle = 000), (Running = 001), (Success = 010), and (Failure = 100). Success indicates that the NPU runs to completion and produces results that are matching the golden results (i.e., known correct results).

To generate the required MIF files and the proper Verilog header file to be later compiled using Intel Quartus Prime Pro, use the `-mif` option.

```
$ cd compiler
$ python driver.py -mif
```

The user can then instantiate the `self_tester_shim` module in a top-level Verilog/SystemVerilog file along with reset gate and PLL for clock generation. Please follow the detailed instructions provided in the `how_to_run_npu_on_fpga.pdf`.

5. System Demo (CPU + FPGA)

We also provide a system-level demonstration, where an NPU on a Stratix 10 NX FPGA communicates with a host CPU in a server system via PCIe link using DMA transfers. The CPU software driver uses DMA transfers to load an AI program and provide inputs to the NPU on the FPGA. Then, the NPU executes the AI program with provided inputs, and produces resulting outputs that is then transferred to the CPU. Finally, the CPU validates NPU outputs against golden reference result to check for correctness. This demo shows that NPU can be controlled by CPU, and new AI program can be loaded to the NPU purely via CPU software, i.e., using same NPU bitstream pre-programmed on the FPGA, no need for Quartus run.

Installing and running the system demo. Please refer to `how_to_run_cpu_fpga_system_demo.pdf`

Generating Input Files for System Demo. The demo has included input AI programs that are immediately ready to use for demo runs (see `how_to_run_cpu_fpga_system_demo.pdf`). However, for any new AI programs, the following details describe how to prepare such input files (i.e., .dat files) for the demo system using the NPU front-end.

The NPU front-end can be used to generate `.dat` files for instructions, MRF contents, input vectors, and golden results. These files can then be used to run the PCIe demo of the NPU. Assuming that your NPU application is written in the `driver.py` file, you can generate the required `.dat` files for it using the following compiler flags:

```
$ cd compiler
$ python driver.py -pcie -loop 171
```

The loop compiler flag determines how many times you will run this NPU program back-to-back. For example, if your NPU program is written as the normal batch-6 execution of the NPU, then looping over it for 171 times is equivalent to running it on a batch size = 6 x 171 = 1026. The front-end will generate input vectors and output golden results for that many execution iterations.

Citation and References

The following paper may be used as a citation for this NPU work:

- Boutros A., Nurvitadhi E., Ma R., Gribok S., Zhao Z., Hoe J.C., Betz V., Langhammer M., "Beyond Peak Performance: Comparing the Real Performance of AI-Optimized FPGAs and GPUs", IEEE International Conference on Field-Programmable Technology (ICFPT), 2020. Available at: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/a1153843-beyond-peak-performance-white-paper.pdf

The following provide further references on Intel Stratix 10 NX FPGA and our prior NPU research.

- Intel Stratix 10 NX Website: https://www.intel.com/content/www/us/en/products/details/fpga/stratix/10/nx.html
- Langhammer M., Nurvitadhi E., Pasca B., Gribok S., "Stratix 10 NX Architecture and Applications", 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), 2021. Available at: https://dl.acm.org/doi/10.1145/3431920.3439293
- Nurvitadhi E., D'Souza R., Martin W., "Real Performance of FPGAs Tops GPUs in the Race to Accelerate AI", Intel White Paper. Available at: https://www.intel.com/content/www/us/en/products/docs/programmable/fpga-performance-tops-gpus-white-paper.html

- Nurvitadhi E., Boutros A., Budhkar P., Jafari A., Kwon D., Sheffield D., Prabhakaran A., Gururaj K., Pranavi A., Mishali N., “Scalable Low-Latency Persistent Neural Machine Translation on CPU Server with Multiple FPGAs”, IEEE International Conference on Field-Programmable Technology (ICFPT), 2019. Available at: https://ieeexplore.ieee.org/document/8977886
- Nurvitadhi E., Kwon D., Jafari A., Boutros A., Sim J., Tomson P., Huseyin S., Chen G, Knag P., Raghavan K. Krishnamurthy R., Gribok S., Pasca B., Langhammer M., Marr D., Dasu A., “Why Compete When You Can Work Together: FPGA-ASIC Integration for Persistent RNNs” IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2019. Available at: https://ieeexplore.ieee.org/document/8735536