WNN-PiM: Accelerating Weightless Neural Network Inference and Training on a Real Processing-in-Memory System

Xavier Servot
Master Student
ETH Zürich

Juan Gómez Luna
Advisor
ETH Zürich

Onur Mutlu
Advisor
ETH Zürich

Problem & Motivation

Deep Neural Networks are not suitable for **edge inference** and **training**
Because of ❶ High Flop Count and ❷ Large Memory Footprint

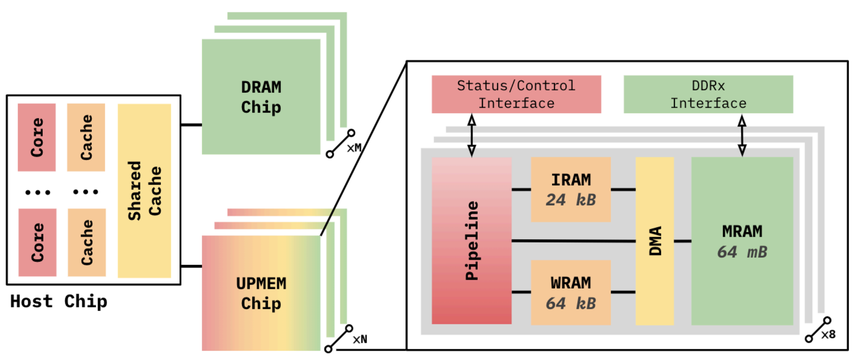
Weightless Neural Networks (WNN) emerged as a **viable alternative**
They learn non-linear activations via lookup-tables

However, **edge inference** and **training** of WNNs remain a **challenge**
Because of ❶ Random Memory Accesses with ❷ Low Reuse and ❸ High Strides

Computing systems with **Processing-in-Memory** (PiM) capabilities can alleviate this **data movement related costs** by placing **computation near the data**

State-of-the-Art PiM System

We use the **UPMEM PiM** architecture with **general-purpose processing cores** called **DPUs**



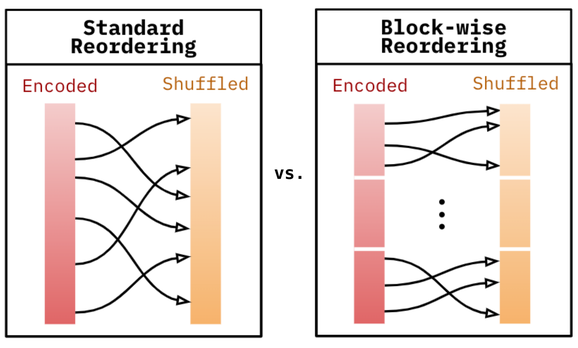
- We use ≤**256 DPUs** to represent an **edge system**
- 32-bit integer arithmetic
- 8-bit multiplication **only**
- Up to 24 PiM threads (*tasklets*)
- 64 mB DRAM bank (*MRAM*)
- 64 kB scratchpad (*WRAM*)

Proposed Solution

We implement **inference** and **end-to-end training** of WNNs on the UPMEM PiM system.
To that end, we write DPU kernel code for ❶ Encoding, ❷ Shuffling, ❸ Hashing and ❹ Filter Reduction while Pop Count and Argmax are reduction steps only executed during inference and done in host.

We partition the model *by classes* and *by filters* (i.e. *lookup tables*) so that each part of the model **fits in WRAM.**
Random memory accesses are then executed at a throughput of **1 per cycle**, thus alleviating the main data movement-related cost.

Algorithmic Improvements & Further Partitioning



Problem. Shuffling makes memory accesses throughout the WNN pipeline **unpredictable**.
In particular, any filter within a class might be accessed from any DPU, which would *not* allow us to **distribute** the model by filter.

Block-wise reordering enables us to **further distribute** the model *by filter*. This way, bigger models can be **fit in WRAM** by using more DPUs.

Moreover, *block-wise reordering* **improves model accuracy** across the models we have tested.

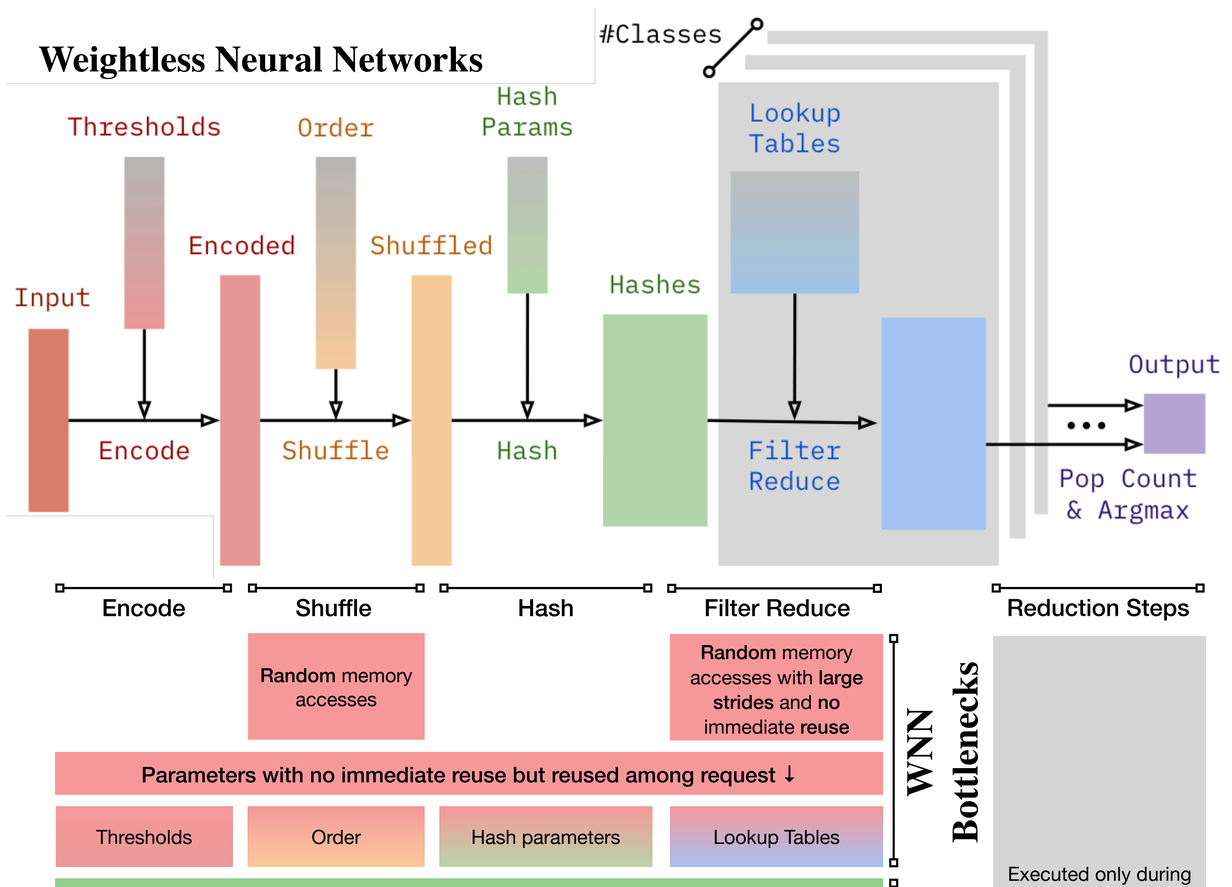
Methodology: System

We conduct our performance evaluation on a **UPMEM PIM system** that includes a 2-socket Intel Xeon Silver 4110 CPU at 2.10 GHz (host CPU), standard main memory (DDR4-2400) of 128 GB, and 20 UPMEM PIM DIMMs with 160 GB PIM-capable memory and 2560 DPUs.
As our **baseline CPU**, we use a 2015 Quad-Core Intel Core i7 at 2.50 GHz, in a system with **6 MB of combined L3 cache** and 256 KB of L2 cache per core.

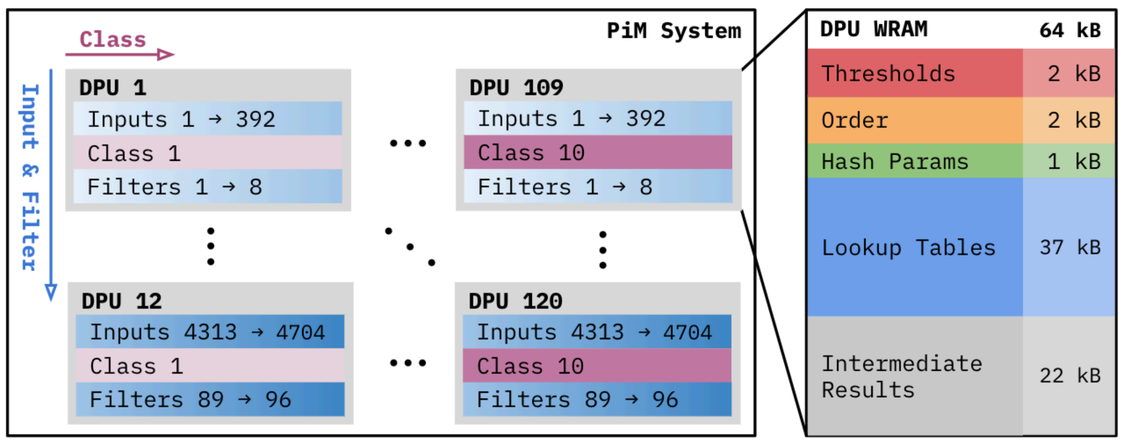
Methodology: Experiments

We conduct **weak scaling experiments** (scaling the number of DPUs while scaling the amount of work to do) to understand the performance gains at different scales, and we find that this architecture **can accelerate WNN inference and training with** only a small number of DPUs, that could be present in an **edge system**.
We perform **inference and training** on a 1000 images in each case. To scale up the number of DPUs, we let the **number of models scale up**.

Weightless Neural Networks



Model Distribution



Inference-specific optimizations

Reduces memory footprint of WNNs and enables to fit **16x larger** models in *WRAM*

❶ Bleaching `u16 → u8`

After training, the counting bloom filters are transformed into standard bloom filters.
To this end, a bleaching threshold is selected on a validation dataset.

❷ Bit-Packing `u8 → bool`

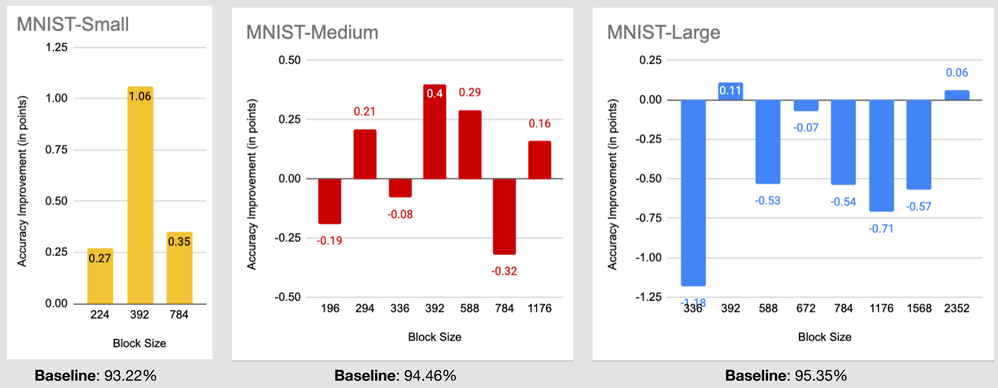
Each entry of the bloom filter tables is binary hence 8 entries can be packed into an 8-bit integer.
Memory accesses take an additional 2 cycles to decode the packed representation.

Methodology: Accuracy Gains

We write **our own C library** for Weightless Neural Networks and **validate** it against an existing python library. We use the C library for our **baseline timing**.
We evaluate our **accuracy improvements** against three WNN models of different size trained on MNIST: MNIST-Small (70 kB), MNIST-Medium (210 kB) and MNIST-Large (960 kB).

Results: Accuracy Benefits of Algorithmic Improvements

Reordering by 1D-Blocks: accuracy improvements in points



Baseline: 93.22% **Baseline**: 94.46% **Baseline**: 95.35%

Reordering by 2D-Blocks: Accuracy improvements in points

MNIST-Small

rows / cols	28	14	7
56	-0.12	0.85	-4.16
28	-0.08	0.9	-4.27
14	0.87	1.42	-0.38
8	-0.11	0.89	-7.42
7	-0.79	0.06	-4.57
4	-0.46	-0.4	-9.48

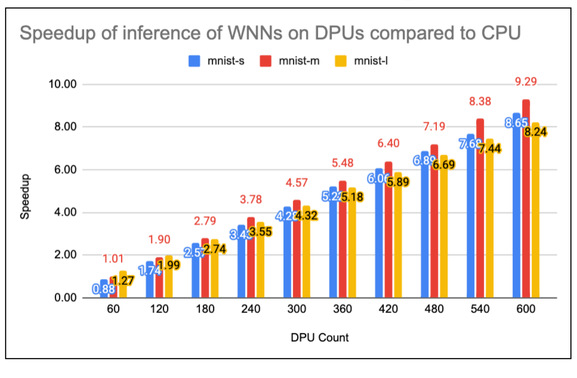
Baseline: 93.22%

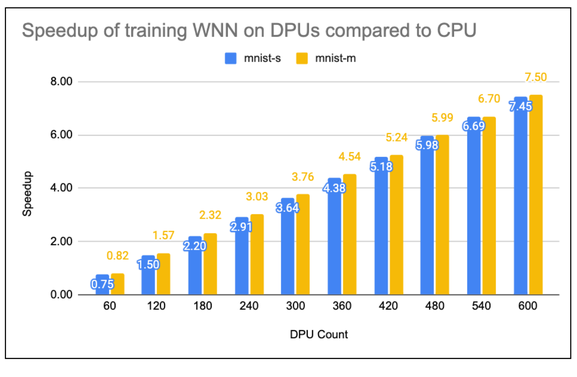
MNIST-Large

rows / cols	28	14	7
168	-0.11	0.56	-4.48
84	0.04	0.64	-2.28
56	-0.6	0.16	-1.35
42	-0.65	0.2	-3.18
28	-0.57	0.3	-7.74
24	-0.18	0.5	-3.28
21	-0.54	0.11	-3.29
14	0.12	0.46	-4.93
12	-1	0.22	-4.17
8	-0.93	0.05	-3.12
7	-3.3	-1.71	-11.07

Baseline: 95.35%

Results: Performance Improvements





Future Work

1. In general, more work needs to be done to **understand how WNNs can be scaled up** in size and adapted to **more complex datasets**. Work also needs to be done to understand the **theory behind WNNs**.
2. Evaluating the accuracy improvements of **block-wise reordering** on **WNNs trained using back-propagation**
3. **Unifying** the inference and training implementations to illustrate **on-device continual learning**

Summary

Weightless Neural Networks are emerging as a **promising alternative** to Deep Neural Networks

We implements **inference** and **training** of WNNs on the **UPMEM PiM system** and find that this architecture **can accelerate training and inference** with a careful implementation

Along the way, we find an **algorithmic modification** that **improves accuracy** and **model partitioning**

Github Links

Cbthowen. A complete and fast WNN inference and training library fully written in C and validated against its Python counterpart
`https://github.com/Xavier0301/Cbthowen`

BiM (BTHOWeN in Memory). An (incomplete) implementation of WNN inference on UPMEM DPUs
`https://github.com/Xavier0301/BiM`

BiMT (BTHOWeN in Memory - Training). An (incomplete) implementation of WNN training on UPMEM DPUs
`https://github.com/Xavier0301/BiMT`

Acknowledgements

We acknowledge the generous gifts provided by our industrial partners, including ASML, Facebook, Google, Huawei, Intel, Microsoft, and VMware. We acknowledge support from the Semiconductor Research Corporation and the ETH Future Computing Laboratory.