article

DBScan Embedded Improvements

October 21, 2024

I. Why

Our original goal was to run a systolic DBScan on the embedded neuromorphic kit, with an epsilon of 1, and min. points of 4.

After spending 2–3 weeks optimizing the embedded neuromorphic code to run DBScan at a reasonable framerate we came to the conclusion that the hardware was simply not powerful enough to run the model. We were eventually able to get the model to run at a theoretical max of 1 FPS, at a quarter of the DAVIS 346's resolution. This model was nearing the limits of the Pi Pico 2040 (585 neurons, 1613 synapses), so increasing the resolution was likely not a possibility. Running a model for the full camera resolution would require 2345 neurons and 6488 synapses (3380 neurons and 15800 synapses for an epsilon of 2). Not satisfied with these results, we wanted to see other options that were available.

Our attention shifted from the kit to a more performant yet still embedded platform. In an attempt to keep similar power consumption, and availability to interface with the world we landed on the Milk-V Duo 256M, an eSBC running a single 1Ghz RISC-V core, with 256Mb of DRAM, all while consuming 0.5W. The board offers 8x the CPU speed of a single Pico 2040, or 2.6x the 3 Picos of the kit, yet it consumes only 0.5W, while the kit consumes 0.6W under load. The Duo 256M has around 1000x more RAM than a Pico, meaning much larger networks are supported.

The next step was to begin porting the embedded neuromorphic kit code to a single monolithic application which we can run on the Duo.

II. Architecture

The Duo features a similar SDK to the Pico, providing its own toolchain to allow cross compilation of RISC-V binaries (https://github.com/milkv-duo/duo-examples?tab=readme-ov-file).

A. Porting the RISP Processor

Embedded neuromorphic (and by extension the app-compile tool) create three separate binaries that the user can flash to the respective pico. The first step was to combine the functionality of each component into a single executable.

We decided to dedicate a simple function for both observations and actions, keeping a similar API to the kit code. The user-defined `observations` function is called for each input neuron at the start of a time step. Similarily, the `actions` function is called at the end of each timestep, with an array indicating which of the output neurons spiked given to `actions`.

Any setup code that was previously split across the two files can simply be brought over, and called before the processor begins running.

This simple port allowed us to run the full scale model, and we were able to achieve a theoretical max of 7.6 FPS.

B. Encoding Limits in the Processor

Similar to the original Embedded Neuromorphic Kit, our port avoids dynamic allocation of memory. This means that the limits (or maximums) of the network must be known at compile time. For instance the number of neurons and synapses is statically defined, which requires the user to extract these values from their network ahead of time.

If we can count on the user to define these values correctly we can remove error checking from our code that enforces these limits. Future work is needed to extract these parameters from the network to eliminate a potential for user error.

Removing error checking does not net much a benefit (0.7%), but if the network is sized properly it will never be needed.

C. Reworking RISP for Better Spatial Locality

On the Duo 256M we have 32Kb of L1, and 128Kb of L2 data cache. Of course, running Linux we have almost no guarantee that our process will be able to occupy the cache entirely on its own, so it is a good idea to minimze our application's footprint.

The smallest footprint we were able to achieve on the ported kit code was ~150Kb (which includes neurons, synapses, and $\text{charge}_{\text{changes}}$). After considering different methods to improve memory usage we found a solution that both decreases memory usage, and greatly simplifies the processor. Instead of storing "events" separatly from the entity that they act upon, then applying their effects later, we can simply apply

the changes directly to a neuron.

To accomplish this, each neuron now contains a ring buffer who's size is equal to the maximum number of tracked timesteps. Now when a neuron fires, it follows each outgoing synapse and adds the associated weight to the downstream neuron. The position in the ring buffer to which this charge is added can be determined by the delay value of the synapse. This model allows for charges to accumulate for timesteps in the future, without the need to individually track charges, thus drastically decreasing memory usage from 150Kb to 68Kb.

This change nets the greatest improvement in performance, running the full model at theoretical 10.034 FPS, 31.9% faster than the originally ported code.

D. Embedding Synapses Within Neurons

On the same line of thought from above, we can further reduce memory usage by embedded synapses within their associated neurons. Because we want to support more than 255 synapses in a network each synapse has to have a 16-bit integer dedicated to its index. We can eliminate these indexes by places synapses within neurons.

> For networks with few synapses this would increase memory usage, as each neuron now has to have capacity for the maximum number of outgoing synapses.

In addition to reducing memory usage, this change further improves spatial locality, which improves the speed of the network further. The final outcome is 10.234 FPS, a full 34.4% improvement over our baseline implementation.

III. Benchmarking

A. Small

Epsilon 1, Min. Pts. 4

System Description	Mean	Max. FPS	% Gain	Observed FPS	% Gain	Approx. Size
RISP Processor	10.512	7.610	–	5.103	–	150Kb
No Error Checking	10.434	7.667	0.7%	5.161	1.1%	150Kb
Ring Buffer	7.973	10.034	31.9%	6.057	18.7%	68Kb
Embedded Synapses	7.817	10.234	34.4%	6.132	20.2%	60Kb

B. Medium

Epsilon 2, Min. Pts. 4

System Description	Mean	Median	Max. FPS	Observed Fl
Embedded Synapses	13.930	13.930	5.743	5.1