Approximate computation of post-synaptic spikes reduces bandwidth to synaptic storage in a model of cortex

Dimitrios Stathis*, Yu Yang*, Ahmed Hemani*, Anders Lansner§
*KTH Royal Institute of Technology, Stockholm, Sweden. Email: stathis, yuyang2, hemani@kth.se
§Stockholm University and KTH Royal Institute of Technology, Sweden. Email: ala@kth.se

***Abstract*— The Bayesian Confidence Propagation Neural Network (BCPNN) is a spiking model of the cortex. The synaptic weights of BCPNN are organized as matrices. They require substantial synaptic storage and a large bandwidth to it. The algorithm requires a dual access pattern to these matrices, both row-wise and column-wise, to access its synaptic weights. In this work, we exploit an algorithmic optimization that eliminates the column-wise accesses. The new computation model approximates the post-synaptic spikes computation with the use of a predictor. We have adopted this approximate computational model to improve upon the previously reported ASIC implementation, called eBrainII. We also present the error analysis of the approximation to show that it is negligible. The reduction in storage and bandwidth to the synaptic storage results in a 48% reduction in energy compared to eBrainII. The reported approximation method also applies to other neural network models based on a Hebbian learning rule.**

Keywords—Approximate computing, Neuromorphic Hardware, ASIC, 3D DRAM, Bandwidth optimization

I. Introduction

In this paper, we focus on applying approximate computing to a complex model of the cortex to improve its efficiency by reducing its bandwidth requirement to the synaptic storage by optimizing the access pattern and reducing the synaptic storage itself. We apply this approximation to a biologically plausible model of the cortex called Bayesian Confidence Propagation Neural Network **(BCPNN)**. We build upon and compare the benefits of the approximation algorithm to a previous ASIC design [1], called eBrainII. The approximation algorithm has been proposed in [2]. Yu Yang, et al. [2] have shown that at the algorithmic level, the approximation does not change the behaviour of the network. They proved that when applied to a GPU implementation, the algorithmic optimization can reduce the memory bandwidth and storage requirement.

The BCPNN is organized as a network of hyper-column units **(HCUs)**. Each HCU has its own synaptic data stored in a matrix format. The number of HCUs and the size of each HCU depends on the complexity and size of the network. We briefly introduce BCPNN and its computation model in Section II. The rows and columns of the synaptic matrix are updated in response to pre- and post-synaptic spikes, respectively. Because of this organization, the BCPNN requires a dual access pattern to the memory. This dual access pattern is very inefficient for memory since memories such as DRAMs can only be accessed in a row-wise manner.

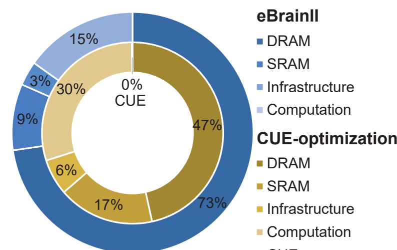

Fig. 1: Energy breakdown of eBrainII (blue) and eBrain with CUE opt. (gold)

In eBrainII [1], the authors identified the DRAM used for synaptic storage as the leading source of power consumption in the design, as shown in Fig. 1. We note that this dominant power consumption is despite leveraging the efficiency of 3D integrated DRAM, whose dimensions were customized for BCPNN. Fig. 1 shows that 73% of the total energy is consumed by the DRAM. An approximation scheme has been proposed [2], to reduce the bandwidth to the synaptic storage. We refer to this scheme as Column Update Elimination or **CUE**. The core idea behind CUE is to remember the recent output (post-synaptic) spikes in a history buffer and update the column cells when updating the rows, thereby avoid column-wise access to the synaptic storage. Since the buffer is finite, some output spikes are lost. The impact of the lost spikes is approximated by statistically predicting the number and instance of the output spikes that were dropped. In this work, we focus on the hardware implementation of the BCPNN with the CUE model of computation. We compare the benefits of using this approximation against the ASIC implementation of BCPNN without CUE [1].

Although we focus on BCPNN in this paper, these optimizations are also possible to apply to other spiking neural networks (**SNNs**), such as networks using spike-timing-dependent plasticity (**STDP**) for learning rule and leaky integrate and fire (**LIF**) neurons.

II. The BCPNN Model

The BCPNN is a spiking neural network model with Hebbian-Bayesian synaptic plasticity. Simpler brain models, like the STDP-LIF, use individual neurons as their atomic neuronal unit. In contrast, the atomic neuronal unit in BCPNN is called a *Mini-Column Unit* (**MCU**). An MCU mimics 100 neurons in a cortical column whose outputs are strongly correlated. The MCUs model leaky, integrate, and fire behaviour. Multiple MCUs are encapsulated in a Hyper-Column Unit (**HCU**). The BCPNN is a network of HCUs that trigger each other with spikes. The MCUs in an HCU compete in a soft winner-take-all (WTA) fashion. The winning MCU(s) generate an output spike *(S_j)*. This output spike arrives at the inputs of other HCUs as an input spike *(S_i)*. Each HCU has its own

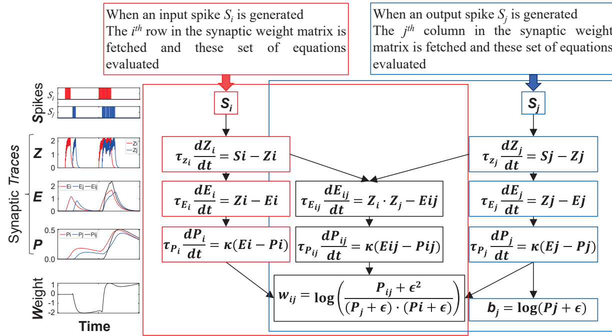

Fig. 2: BCPNN model of computation. Modified from [3].

synaptic storage, and the HCUs operate in a parallel non-deterministically concurrent fashion.

The input and output spikes (S_j, S_j) go through a cascade of low-pass filters, shown in Fig. 2. This cascade generates intermediate traces, *Z, E* and *P*. The pre- and post-synaptic *Z* traces go to a common cascade that correlates the two spike trains. The background for these traces from a neuroscience perspective is discussed in [3]. The traces are categorized into three types, *i*, *j* and *ij* traces. These traces constitute the synaptic storage of each HCU. They are organized as a column vector, row vector and matrix, as shown in Fig. 3a. The rows represent the state of connection to the other HCUs. Columns represent the state of individual MCU in an HCU.

A. Human scale BCPNN dimensions & requirements

Human scale BCPNN requires 2 million HCUs. Each HCU has 100 MCUs and 10 000 input connections, i.e., the synaptic storage has 100 columns and 10 000 rows. Each HCU receives on an average 10 000 pre-synaptic spikes/s. The storage, bandwidth and computational requirements resulting from these dimensions are shown in Table 1.

B. The lazy evaluation computational model of BCPNN

The original BCPNN model required decaying the traces every *ms* and reinforcing the traces in response to spikes. Accessing the entire synaptic storage every *ms* is very expensive, so a new model was proposed [4], [5]. In the lazy evaluation model, the cells in the synaptic storage and the spikes are timestamped. The timestamp allows applying an integrated decay along with the reinforcement triggered by the input and

TABLE 1: BCPNN HUMAN SCALE REQUIREMENTS FOR ONE HCU AND FULL HUMAN SCALE NETWORK 2×106 HCUS

	Single HCU	BCPNN
Computation	81 MFlop/s	162 TFlop/s
Storage	25 MB	50 TB
Bandwidth	100 MB/s	200 TB/s
Spike propagation	100 kB/s	200 GB/s

output spikes. In the lazy evaluation, there are three types of operations and illustrated in Fig. 3b:
<u>*Column update*</u> is triggered when an MCU or MCUs win the soft WTA competition generating output spike(s).
<u>*Row update*</u> is triggered by the arrival of input spikes.
<u>*Periodic & Support Update*</u> is performed every *ms.*

III. COLUMN UPDATE ELIMINATION COMPUTATION MODEL

In this section, we introduce the column update elimination **(CUE)** model, for details see [2]. The CUE method is an algorithmic optimization that primarily aims to reduce the required bandwidth to the synaptic storage.

In the lazy evaluation scheme of the BCPNN, the synaptic storage is organized in a matrix. The DRAM storage is accessible row-wise, but the BCPNN requires both row-wise and column-wise access. As a result, row-wise accesses are efficient, but column-wise accesses are very inefficient. The authors in [1], try to balance the column and row access by interleaving the synaptic rows. This data shuffling does improve the overall utilization of the channel to DRAM, but it does not eliminate the troublesome column access.

The CUE optimization solves this problem algorithmically by eliminating the column access. The ideal CUE method is functionally equivalent to the lazy evaluation scheme briefly described in section II.B. The ideal method requires an infinite buffer and is not practical. For this reason, an approximate method that results in a small acceptable deviation from the lazy evaluation BCPNN scheme has been proposed. The innovative thing about the approximation method is that it ensures that errors do not accumulate with time, see [2] for details.

The core idea behind CUE is illustrated in Fig. 3c and can be seen in contrast to the lazy evaluation scheme shown in Fig. 3b. When an output spike is generated, the column is neither fetched nor updated; instead, the spike is buffered. When input spikes arrive, the rows are fetched and updated, and the row's cells that correspond to the columns identified by the output spikes in the buffer are also updated.

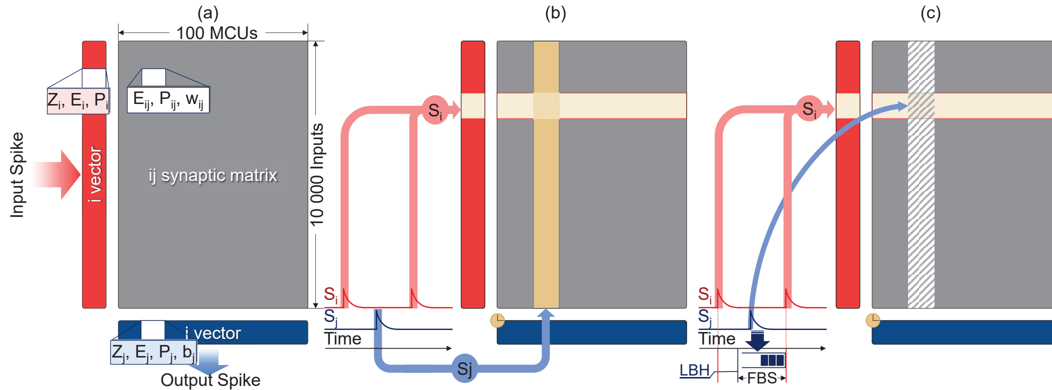

Fig. 3: (a) BCPNN data structure, (b) lazy evaluation update model, and (c) CUE update model

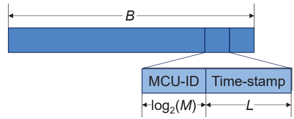

Fig. 4: Structure of the history buffer, the size of the buffer is given in cells and the size of the cells in bits

In a realistic scenario, a history buffer with a finite size would be used, marked as a finite buffer size **(FBS)** in Fig. 3c. Any spikes beyond the lookback horizon **(LBH)** of the buffer would be dropped since they cannot be recorded. The dropped spikes introduce a systematic error that accumulates over time, resulting in divergence from the ideal BCPNN. To avoid this systematic and accumulative error, the authors in [2] have proposed an approximation scheme that predicts the presence of the dropped spikes, i.e., the spikes beyond the LBH.

IV. The CUE Implementation

The CUE update mechanism requires two main components, the spike history buffer, and the approximation function.

A. The output-spike history buffer

The spike history buffer must record two types of information, which MCU has spiked and at which instance – the time-step. In BCPNN, one time-step is 1 *ms*, and one epoch is 1 sec. In the algorithmic CUE paper [2], the authors opted for a software style convenience to create a buffer of $M \times T$, where M is the number of MCUs and T is the number of time-steps. This one hot encoded two-dimensional buffer is excellent for programming convenience, but not the most efficient from a hardware implementation perspective. Since typically only one MCU per HCU spikes per time-step, such a buffer would be massively under-utilized. An alternative buffer structure is proposed to avoid this under-utilization.

The MCUs in an HCU can share the buffer among them. Rather than one-hot encoding the ID of MCUs as is done in [2], we adopted a binary encoding. This would reduce the buffer size per HCU to $Log_2(M) \cdot T$; where M is the number of MCUs per HCU and T is the history length in time-steps. While this is a significant improvement, it still would result in under-utilization. Because the spiking activity is very sparse; it is extremely rare to have one spike per time-step.

Based on this analysis, we have adopted a buffer, in which each cell stores two objects: The binary encoded ID of the spiking MCU and timestamp of the time-step at which it spiked. In this scheme, the dimensions of the buffer would be equal to $(Log_2(M) + L) \cdot B$. Where L is the number of bits allocated for recording the timestamp when the spike occurred, and B is the size of the buffer. Fig. 4 illustrates such a buffer. This scheme has the advantage that it results in no cells being left empty. In our experiments we use $Log_2(M) = 7$, $L = 32$ and $B = 100$. We use $B = 100$ to make sure that the buffer history will be at least as long as the one reported in [2].

B. Approximation function

We have modified the approximation function proposed in [2] for hardware implementation. The approximation function predicts the number and time instance (timestamp) of the output spikes beyond the LBH; i.e., the spikes that *might* have been dropped. The authors in [2] have proposed two strategies, both based on the firing rate (r) of the output spikes. In the static version, the rate r is constant. The adaptive version continuously samples and records the r. The approximate function is invoked once the LBH is reached. The last input

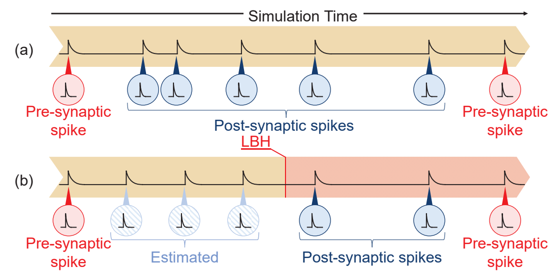

Fig. 5: Illustration of a) real distribution of spikes generated by an MCU and b) estimated spike distribution using the buffer and the estimation function.

spike marks the end of the approximation interval for the specific row that is being updated, shown as dt in Fig. 5b.

The approximation that we implement in hardware differs compared to the one used in [2]. In the software implementation, the spikes generated in the interval dt is Poisson distributed with a mean of $1/r$. The implementation of such distribution requires a randomization process to be invoked for every time-step. We have simplified the distribution to a hardware friendly uniform distribution, with $1/r$ intervals, Fig. 5. A small degree of randomization is introduced by predicting the position of the first spike randomly in the interval $[1, 2/r]$. Fig. 6 shows the comparison between the two schemes, the one with the Poisson distribution (A) vs the one with the uniform distribution (B) adopted in this paper. The two functions are compared in terms of the probability of introducing unacceptable errors by the approximation function. As can be seen, this probability is very low, and the adaptations, we have done for hardware implementation does not significantly change it. The comparison was made for a range of two critical parameters to establish the validity and generality of the simplification. The first parameter M decides the number of MCUs in an HCU. The second is α that decides how active an HCU is. When $\alpha = 0$, the HCU is silent and has a very low probability of firing, when $\alpha = 1$ the HCU is active and has high probability of firing. We refer to [2] for more details about how these parameters affect the behaviour of the network.

C. Hardware Architecture

The schematic of the proposed architecture is shown in Fig. 7. Each HCU has one such unit. The synaptic update process

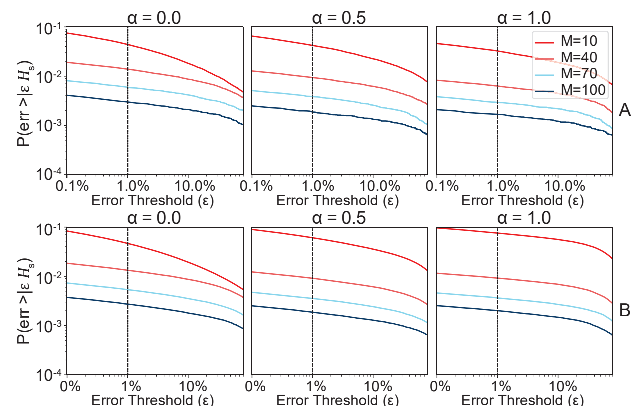

Fig. 6: Comparing the probability of introducing unacceptable errors by the Poisson distributed predicted spikes in (A) vs the uniform distributed predicted spikes in (B).

gets its input from either the CUE history buffer or from the approximation function. The buffer has been implemented as a circular buffer in a register file. The CUE finite state machine (**FSM**) is the main controller that has three principal functions. 1) Enqueue the output spikes in the buffer generated by the MCUs, storing the MCU ID and timestamp. 2) Dequeue these spikes from the buffer when the row update is triggered. 3) Invoke the approximation function if the last input spike is older than the LBH. The CUE FSM needs a random number, that comes from the LFSR, to generate the random timestamp for the first spike. The subsequent spikes are uniformly placed, as shown in Fig. 5b.

V. Implementation Results

The metrics used for the comparison are the area, latency, and energy. The results for the CUE come from an implementation in 28 nm technology. The design was clocked at 200 MHz, and the area and energy numbers are from post-layout data. The metrics for the rest of the design and the DRAM are based on the results reported in [1]. The DRAM energy reported in [1] includes energy for core and I/O energy, which we estimated using DRAMPower [6] integrated into DRAMSys [7]. Since the computation amount does not change with CUE, we assume that the energy required for the computation remains unchanged. The rest of the functions of the system also remain the same, so we can directly use the results from [1].

A. Area and Energy Overhead of CUE

The post-layout design of CUE, as shown in Fig. 7, occupies an area of 0.024 mm^2. Each HCU in the BCPNN requires one such CUE module. The total overhead compared to eBrainII is 5.89%. This overhead is not only small, but it does not result in any increase in the footprint of the design. This is because, as reported in [1], the area occupied by the logic underneath is 48% smaller than the custom 3D DRAM.

The energy overhead of CUE is negligible. The average measured energy of the CUE module was 0.467 nJ per row update. The energy per HCU per second can be calculated using this average energy per row. Since there are on an average 10 000 row update/sec, the energy consumption of CUE is equal to 4.67 µJ/sec. The reason behind this minimal energy overhead is that the CUE module can be heavily clock-gated. The history buffer gets, on an average only 100 inputs/sec. As a result, the write energy to the buffer is minimal. The FSM uses most of the energy followed from the read access to the buffer. The CUE module only operates for a small amount of time, on average, 236 clock cycles for every row update. And it remains dormant and gated for the rest of the time.

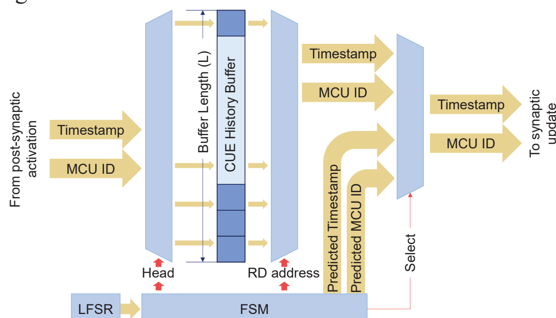

Fig. 7: CUE history buffer and prediction architecture

B. Storage, Bandwidth and Energy savings due to CUE

1) Reduction in Storage: Many of the timestamps necessary for the column updates are no longer required, see [2]. This removal has resulted in a reduction in storage requirement from ~25 MB/HCU to ~15,5 MB/HCU, drop of ~33%.

2) Reduction in Bandwidth: The reduction in bandwidth comes from two sources. The primary source is the elimination of column access. The secondary source is the reduction in synaptic storage. Together, the bandwidth is reduced from ~100 MB/s/HCU to ~31 MB/s/HCU, a drop of ~67%.

3) Reduction in Energy: The basis for measuring energy consumption is the same as that used by the authors of eBrainII [1]. We assume the same 3D DRAM design and energy/bit cost of 7 pJ/bit. Further, we consider the rest of the eBrainII design to be still valid for the CUE implementation. We justify this assumption because the amount of computation and real-time requirements of human-scale BCPNN remains the same. The savings in energy comes from the reduction in DRAM energy because of the reduced storage and access. As a result, the DRAM energy is reduced by 67%, Fig. 1. Consequently, the total energy drops to 1.57 kW compared to 3 kW in eBrainII. As expected, the percentage of energy in DRAM goes down sharply from 73% in eBrainII to 47% with CUE.

VI. Conclusion & Future Work

In this work, we explore the improvements of a new algorithmic optimization called CUE, when applied to a custom hardware design. The optimization reduces the overall required DRAM bandwidth by 67% and the power consumption by 48%. In this paper, we only explore the benefits of CUE optimization; we keep the 3D-DRAM design the same as in the previous work. The CUE optimization provides with ample opportunities to improve the design of the 3D-DRAM itself. We also plan to explore the inherent locality of the BCPNN to reduce the bandwidth to the DRAM further using a custom cache system. We believe that such optimization can reduce the power consumption of a human scale BCPNN to 700 W.

References

[1] D. Stathis *et al.*, “eBrainII: a 3 kW Realtime Custom 3D DRAM Integrated ASIC Implementation of a Biologically Plausible Model of a Human Scale Cortex,” *J. Signal Process. Syst.*, pp. 1–21, Jul. 2020, doi: 10.1007/s11265-020-01562-x.

[2] Y. Yang *et al.*, “Optimizing BCPNN Learning Rule for Memory Access,” *Front. Neurosci.*, vol. 14, p. 878, Aug. 2020, doi: 10.3389/fnins.2020.00878.

[3] P. J. Tully *et al.*, “Synaptic and nonsynaptic plasticity approximating probabilistic inference,” *Front. Synaptic Neurosci.*, vol. 6, no. APR, p. 8, Apr. 2014, doi: 10.3389/fnsyn.2014.00008.

[4] A. Lansner *et al.*, “Spiking brain models: Computation, memory and communication constraints for custom hardware implementation,” in *Asia and South Pacific Design Automation Conference, ASP-DAC*, 2014, pp. 556–562, doi: 10.1109/ASPDAC.2014.6742950.

[5] B. Vogginger *et al.*, “Reducing the computational footprint for real-time BCPNN learning,” *Front. Neurosci.*, vol. 9, no. JAN, p. 2, Jan. 2015, doi: 10.3389/fnins.2015.00002.

[6] K. Chandrasekar *et al.*, “DRAMPower: Open-source DRAM power & energy estimation tool,” *http://www.drampower.info*, 2012.

[7] M. Jung *et al.*, “DRAMSys: A Flexible DRAM Subsystem Design Space Exploration Framework,” *IPSJ Trans. Syst. LSI Des. Methodol.*, vol. 8, no. 0, pp. 63–74, Feb. 2015, doi: 10.2197/ipsjtsldm.8.63.