ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A Thousand Brains on a Thousand Chips

Master Thesis

Xavier Servot

Fall 2025

Advisors: Dr. M. Besta, Prof. Dr. T. Hoefler
Department of Computer Science, ETH Zürich

Abstract

The promise of Thousand Brains Systems lies in their potential to learn sensorimotor models of highly complex environments on-the-fly and continually. Thousand Brains Systems are based on the theory that mammals evolved flexible intelligence through the replication of semi-independent sensorimotor units known as cortical columns.

While today's implementations remain small in scale as the fundamentals get developed and improved upon, there is an obvious motivation to scale up such systems to the size of the human neocortex. However, the fundamental design of Thousand Brains Systems demands parallel execution with heterogeneous weights. This makes scaling such systems incompatible with the high operational intensity demanded by GPUs, and incompatible with modern core-centric computers that rely on deep cache hierarchies.

We argue that Processing-in-Memory systems offer a unique opportunity to massively scale up Thousand Brains Systems, and thus have the potential to revolutionize the applicability of these models to more complex problems. That is because:
(i) Thousand Brains Systems are scaled through increasing the number of Learning Modules, which are small, self-contained and operate in a semi-independent fashion.
(ii) In a Processing-in-Memory chip, compute and memory are co-located, leading to a system with a massive amount of independent small processing units that can concurrently work on heterogeneous data.
We call our idea *a Thousand Brains on a Thousand Chips*.

We introduce Montyll, a Thousand Brains System implemented in high performance C, and Montyll-PiM, a companion implementation on a real-world Processing-in-Memory system. We successfully scale Montyll to 2560 learning modules on a single-node CPU system an a PiM system, representing 44.5M neurons and 16.1B synapses. Montyll-PiM features a speedup of 2.2× over Montyll on a high-end CPU baseline. Our work is the first of its kind studying the impact of large-scale interconnected htm networks and column-based artificial intelligence on computing systems.

Contents

Acknowledgements

My great advisor Maciej Besta jokingly said to me one day that he rarely sees a student who is so stubborn about exploring a topic. Well that stubbornness is a direct consequence of the awfully good supervision and interest I received from Doctor Clay and the disgustingly interesting ideas and theories developed by Jeff Hawkins and everyone else over the years at Numenta and the Thousand Brains Project. I thank them profusely for having endowed me with an equal amount of headaches and amazement.

The same goes for the great Doctor Maciej Besta, my advisor at my current research group, the Scalable Parallel Computing Lab led by the also great Professor Torsten Hoefler, who I sadly haven't had the honor of interacting with much, but who has been nothing but nice to me. Doctor Besta somehow trusted me to explore this idea, and has thus burdened me with a great deal of pleasure.

I also have to give a sizeable thanks to Professor Onur Mutlu and Doctor Mohammad Sadrosadati and all the amazing folks at the SAFARI research group. It is your fault if, today, this idea flourished into anything more than some scribbles in the margin of a book. And it is Doctor Juan Gómez Luna who I will also thank for initially introducing me to programming Processing-in-Memory systems with a great amount of patience and an equal amount of Spanish accent.

Nicole and Jérôme, my parents. You guys deserve the most humorous and well thought-out thank you. It is however quite hard to put into humorous words the very serious patience, love and support that you have been providing me for all my life. You have shown to be some pretty reckless parents by choosing to trust me at a way earlier age than you should have. I love you.

To my brother Nicolas, I feel the need to write a very personal and special note. My brother and I both come from an incredibly privileged environment but it seems like we are both playing a competitive game of who can make their own life more difficult. You have won the battle, but not the war. Fair play, although you have used some borderline illegal move. I love you.

Chapter 1

Introduction

Today's leading artificial intelligence systems lack core features of biological intelligence, especially the ability to learn rapidly [1][2] and continually [3][4][5] while in deployment [6][7], in a compute and energy efficient way [8][9][10].

Building on the theory that mammalian intelligence is based on replication of a core computational unit, the cortical column [11][12], the Thousand Brains Theory [13][14][15] is a theory of intelligence in the neocortex, which argues that each cortical column learns complete predictive sensorimotor models of the world. Adopting these principles, Thousand Brains Systems [16][17] are artificial intelligence systems that aim to bridge the gap with biological intelligence.

Thousand Brains Systems are composed of many learning modules, analogous in function to columns in the neocortex. They are inherently designed to learn through interactions with an environment. Each module learns a local sensorimotor model and contributes to a collective representation through lateral communication. This design implies a distinct scaling profile, as increasing capabilities is done through the instantiation of more learning modules, thus maintaining more module-specific connections and state. The functional equivalence between learning modules and cortical columns provides a strong motivation to scale such systems toward mammalian cortices with thousands of columns, up to the human neocortex with roughly 200'000 cortical columns.

Our goal in this paper is to investigate the unique requirements of scaling Thousand Brains Systems and to investigate the ability of various computing platforms to accommodate those requirements.

In contrast to mainstream deep learning [18][19][20][21], where performance is often driven by batching and high data reuse over shared weights [22][23][24][25], Thousand Brains Systems operate in an *auto-regressive* sensorimotor loop and rely on large amounts of module-specific state and connections. In our analysis of two concrete implementations, this leads to low operational intensity and memory-dominated execution, with large per-step data movement that scales linearly with the number of learning modules.

On CPUs, scaling quickly runs into *core oversubscription* (many more learning modules than cores, forcing time-sharing within a step) and the *memory bandwidth wall*, as the combined working set overwhelms caches and must be streamed from DRAM each step. On GPUs and systolic-array accelerators [26], the lack of cross-step parallelism and the low operational intensity limit utilization. Additionally, irregular control flow (e.g., pointer-chasing) further reduces efficiency.

Processing-in-Memory (PiM) is a long-standing architectural idea [27][28][29][30][31][32][33] that has regained momentum as data movement has become a dominant cost in modern systems [34][35]. The core principle is to bring computation closer to where data resides, trading fast, complex cores for many simpler compute units embedded near memory, thus exposing high internal bandwidth and providing massive parallelism. Recent work has demonstrated PiM's potential across a broad range of workloads, and both academia and industry have recently proposed concrete PiM systems [36][37][38][39][40][41].

For Thousand Brains Systems, the key advantage of PiM is that it offers a large number of independent compute units with high aggregated memory bandwidth, allowing many learning modules to run the same step in parallel on heterogeneous weights. This observation motivates our central idea: *a Thousand Brains on a Thousand Chips*.

In this work, we put forth the following contributions:

- We identify the core architectural bottlenecks that arise when scaling Thousand Brains Systems, and explain why they are a poor fit for both CPU-centric designs and GPU-style accelerators.
- We introduce *Montyll*, a novel Thousand Brains System which integrates elements of low-level cortical processing, for which we provide a unified mathematical formulation and space/complexity analyses, as well as a high performance C implementation.
- We introduce *Montyll-PiM*, the implementation of Montyll on (i) a PiM functional simulator and (ii) a PiM cycle-level simulator.
- We demonstrate that a real-world PiM system can be used to scale Montyll-PiM to 2560 learning modules, and achieve up to a 2.2× speedup over a high-end CPU baseline.

Our goal is to highlight how Processing-in-Memory systems can revolutionize the applicability of Thousand Brains Systems through scale. We hope to give a sense that this holds true for any artificial intelligence system that operates on the principles of column-based intelligence in the neocortex, as their structure demands massive parallel computations on heterogeneous data.

We open source our code in the following repositories. Montyll at https://github.com/Xavier0301/cmontyll. Montyll-PiM on a functional simulator at https://github.com/Xavier0301/montyll-pim and Montyll-PiM on a cycle-level simulator at https://github.com/Xavier0301/montyll-upimulator.

Chapter 2

Background

2.1 Thousand Brains Systems

Despite significant progress over the last decade, current leading AI systems lack core attributes of biological intelligence, such as rapid learning (few-shot) from limited data [1][2], continuous adaptations to new situations [3][4][5], especially in robotics [7][6][42][43], robust generalizations using structured representations [44][45][46][47][48] and compute and energy efficient learning [8][9][10].

Building on the theory that mammalian intelligence is based on replication of a core computational unit, the cortical column [11][12], the Thousand Brains Theory [13][15] is a theory of intelligence in the neocortex developed by Jeff Hawkins and his research group over the past 20 years. It explains intelligence in the neocortex by arguing that each cortical column learns complete predictive sensorimotor models of the world.

The Thousand Brains Project [17] was launched in 2024 in an attempt to build AI systems that bridge the gap between artificial intelligence and biological intelligence, based on core principles of the Thousand Brains Theory. These models are composed of learning modules that mimic the function of cortical columns in mammals, where each learning modules gets sensory input from a sensor patch as highlighted in figure 2.1. Early results on these Thousand Brains Systems are promising [16]. In particular, they outperform Visual Transformers on rapid learning, continuous learning and compute-efficient training and inference on a 3d object detection benchmark.

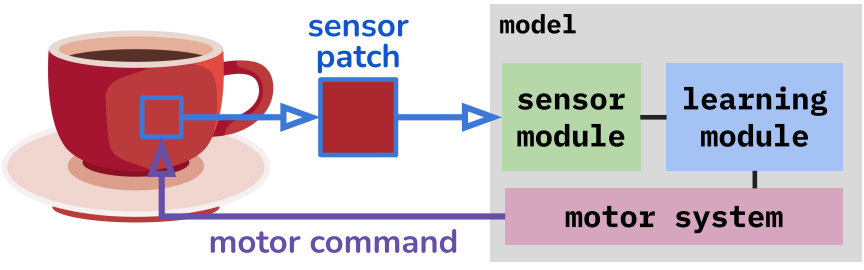
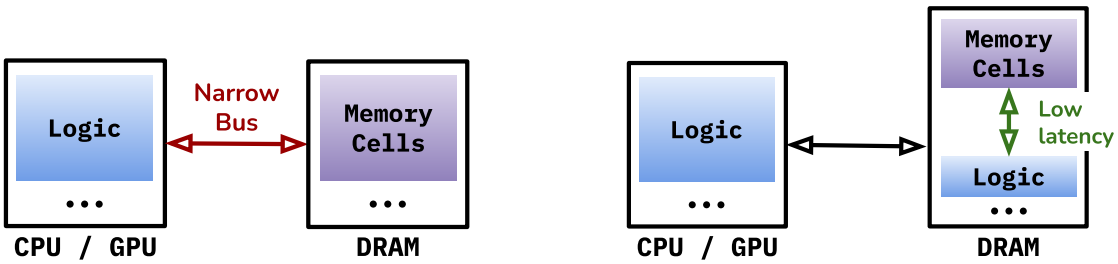


Figure 2.1: Thousand Brains Systems architecture. Thousand Brains Systems are composed of sensor modules and learning modules. Each learning module gets the transformed sensory input from a sensory patch, unlike deep learning methods that get sensory input from complete high-resolution images.

While current implementations of Thousand Brains Systems build models of the world based on explicit graphs in 3d space, one of the long term goals of the Thousand Brains Project is to integrate elements of low-level neocortical processing into Thousand Brains System, e.g. using more complete neuron models [49][50][51][52][53][54] and representing locations with grid cells [55][13][56][57][58][59][60].

2.2 Processing-in-Memory

The concept of co-locating memory and compute on a single chip has been a source of architectural debate since the 1970s [27][61][28], which resurfaced in the 1990s and 2000s [29][30][31][32][33] and has arguably never been as active as it is now [62][34][37][35]. Processing-in-Memory (PiM) remains a radical departure from the traditional Von Neumann architecture that dominates today's computing, where logic and memory are separate and communicate through a narrow data bus, as highlighted in figure 3.2.



(a) Von-Neumann Architecture **(b)** Processing-in-Memory

Figure 2.2: Core-centric architecture vs. Processing-in-Memory. Processing-in-Memory places compute closer to the data, thereby eliminating the potentially costly data movements that core-centric (Von-Neumann) architectures are susceptible to.

Historically, main memory (DRAM) and logic (CPUs) have been manufactured using different fabrication processes. Different pressures led the memory and logic fabrication processes to be vastly different. Patterson et. al's analysis from 1997 remains valid today [31]. Memory fabs offer small DRAM cells for increased memory density and low leakage current for reduced refresh rate. Logic fabs offer fast transistors for fast logic and many metal layers for complex routing.

If a single fabrication process is chosen to co-locate memory and logic, one of the two will suffer some downsides. Manufacturing memory in logic fabs leads to prohibitive costs for large memory capacities. Manufacturing logic in memory fabs leads to slow logic and an inability to easily support complex routing due to the reduced number of metal layers. The Processing-in-memory that is of interest to us is the latter, to keep memory capacity high while adding direct computing capabilities to the DRAM.

Despite the limitations, the promise of Processing-in-Memory lies in its capability to eliminate costly data movements and to provide massive internal bandwidth for parallel computation. As such, it has more recently been put forward in academia for its capabilities in general-purpose computing [36][63][64][65], graph processing [62][66][67] and neural networks inference

[68][64][69][70][71]. In parallel, industry has proposed Processing-in-Memory solutions geared at general purpose computing [38], neural network inference [39][40] and recommender systems acceleration [41].

Despite these significant advances, Processing-in-Memory has yet to establish itself as a widely-used computing platform. While there are many barriers that explain its limited adoption today [72][34], we believe that a significant reason is the lack of a killer application for Processing-in-Memory. An application that is both important and that can uniquely be scaled on Processing-in-Memory hardware. This means an application that takes advantage of the Processing-in-Memory architecture in a way that is comparatively significant with respect to other computing architectures.

We hope to give a sense that any artificial intelligence system that operates on the principles of column-based intelligence in the neocortex, like Thousand Brains Systems, constitutes a potential killer application for Processing-in-Memory systems. This is because the column-based structure induces a need for massive parallel computations on heterogeneous data, thus highlighting a unique advantage of Processing-in-Memory system that has, to the best of our knowledge, not been widely discussed and put forward in the literature.

Chapter 3

Motivating the need for Processing-in-Memory

Why can't we just use CPUs, GPUs, accelerators, neuromorphic chips or compute clusters to scale up Thousand Brains Systems?

3.1 Prelude on Thousand Brains Systems

3.1.1 The motivation to scale

*Why should one care about **scaling up** Thousand Brains Systems?*

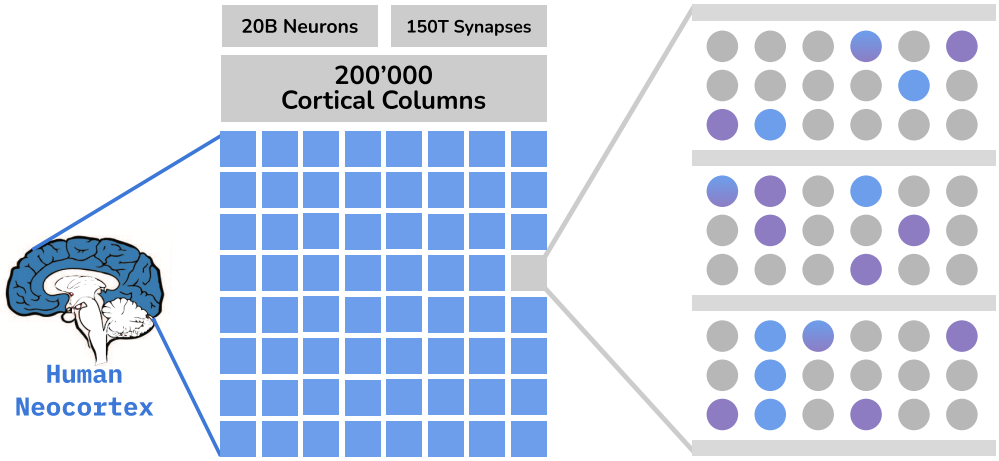


Figure 3.1: Size of the human neocortex. The human neocortex is functionally made up of cortical columns [73], about 200'000 of them, for a total of 20 billion neurons and 150 trillion synapses. Each cortical column is arranged in layers, each layer containing pyramidal neurons.

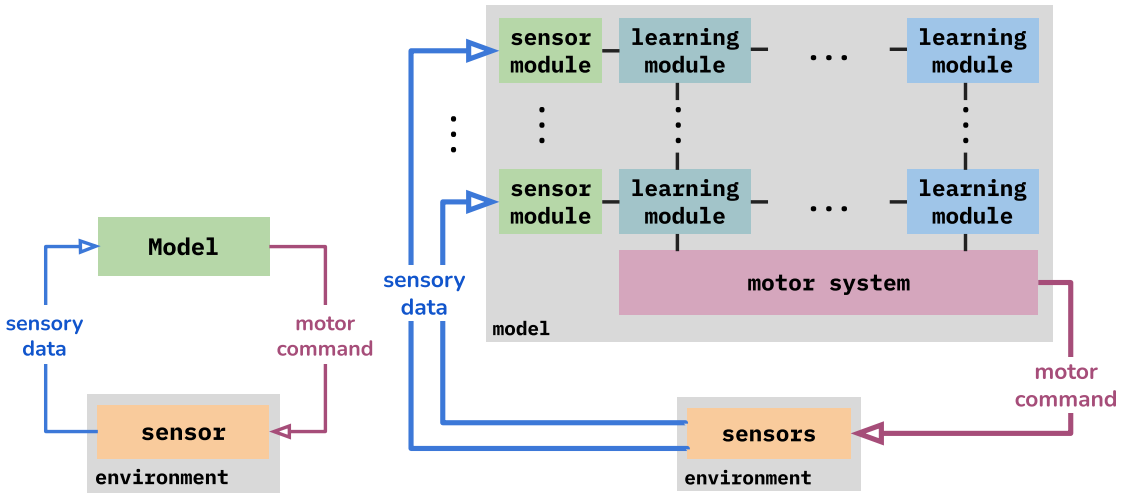
Thousand Brains Systems are based on the Thousand Brains Theory, which posits that each cortical column [12][11] in the neocortex acts as a semi-independent sensorimotor learning system. The cortical organization in the brain is highlighted in figure 3.1. The learning modules in Thousand Brains Systems would be equivalent in function as cortical columns in the neocortex. The human neocortex counts around 200'000 cortical columns [14], although the exact figure varies depending on the neocortical surface area and column diameter figures used [12]. If one believes what precedes

to be true, the motivation to scale Thousand Brains Systems to thousands of learning modules becomes evident.

Early experiments and implementations of Thousand Brains Systems demonstrate quantitative upsides to increasing the number of learning modules. Leadholm et. al [16] show improved accuracy and up to 8x faster inference on a 3d object recognition task. Further research needs to be conducted to characterize the exact benefits of scaling the number of learning modules, specifically in adding multiple modalities, speeding up inference and constructing deeper hierarchical models [17].

3.1.2 Computing Thousand Brains Systems

Thousand Brains Systems learn by interacting with their environment. At each step, the model receives input to its sensors and issues a motor command. As a consequence, Thousand Brains Systems are autoregressive in nature, during both learning and inference. That is because the model's response at step i depends on internal changes that happen at step $i-1$. This is highlighted in figure 3.2a.



(a) Environment loop **(b)** The model in detail

Figure 3.2: Thousand Brains Systems setup. Thousand Brains Systems are inherently designed to learn by interacting with an environment.

This means that we cannot parallelize the computation across steps, but we can look for parallelization opportunities within each step's computation. In particular, Thousand Brains Systems are made up of semi-independent many sensor and learning modules, which represents a parallelization opportunity, as highlighted in figure 3.2b.

Another parallelization method is to run many simulations at once and to batch over the simulations. This method has been successfully applied in the field of deep reinforcement learning [74][75][76]. However, the fundamental goal of Thousand Brains Systems is to create sensorimotor intelligence capable of continual and rapid (few-shot) learning [17]. This is antithetic to learning by running

thousands to millions of simulations at once. A benefit of pursuing the continual rapid learning goal could also be the key to the Sim-to-Real gap in robotics: the difference in performance when training on a simulator and deploying in the wild [77]. If an agent can quickly and continually learn complex models of the world while deployed, there is no need to bridge a Sim-to-Real gap. As such, a goal of the paper is to find a suitable computing platform for embodied sensorimotor intelligence.

3.1.3 Thousand Brains Systems implementations: Monty and Montyll

In this chapter, we often refer to two Thousand Brains Systems implementation. One is Monty [16], which builds object models based on explicit graphs in 3d cartesian space. The other is our custom C implementation Montyll, which we will explain in greater detail in chapter 4. Montyll integrates elements of low-level neocortical processing, e.g. using more complete neuron models [49][51] and representing locations with grid cells [55][56]. Montyll was explicitly designed to be aligned with the long term goals of the Thousand Brains Project [17].

3.2 GPUs, TPUs and Systolic Arrays

> *The use of GPUs has completely revolutionized the applicability of Deep Learning Systems, why can't it do the same for Thousand Brains Systems?*

3.2.1 GPU architecture and operational intensity

The operational intensity OI of a workload is defined as the number of operations executed per byte of accessed memory:

$$OI = Operations/Byte$$

In a way, it is a measure of data reuse: how many times is a single piece of data reused in the workload. GPUs [78], TPUs [25] and, more generally, systolic arrays [26] all provide massive performance benefits for workloads with high operational intensity. To understand why, we have to look at the following example roofline model [24] in figure 3.3 including GPUs, TPUS and CPUs.

We see that at lower operational intensities (1-10 ops/byte), GPU or TPU systems offer no significant advantage over CPU-only systems, especially once we consider the potential cost of moving data from the CPU cores to the GPU/TPU system. However, once we consider higher operational intensities (100-1000 ops/byte), we see that GPUs and TPUs offer significantly higher processing speeds. The cost being that one *needs* high operational intensity in their workload to benefit from the potential performance gains of GPUs and TPUs. Work is continually being done to push the training of modern neural network ever closer to the compute-bound region of the roofline model, by reducing data movements and thus increasing the operational intensity [81].

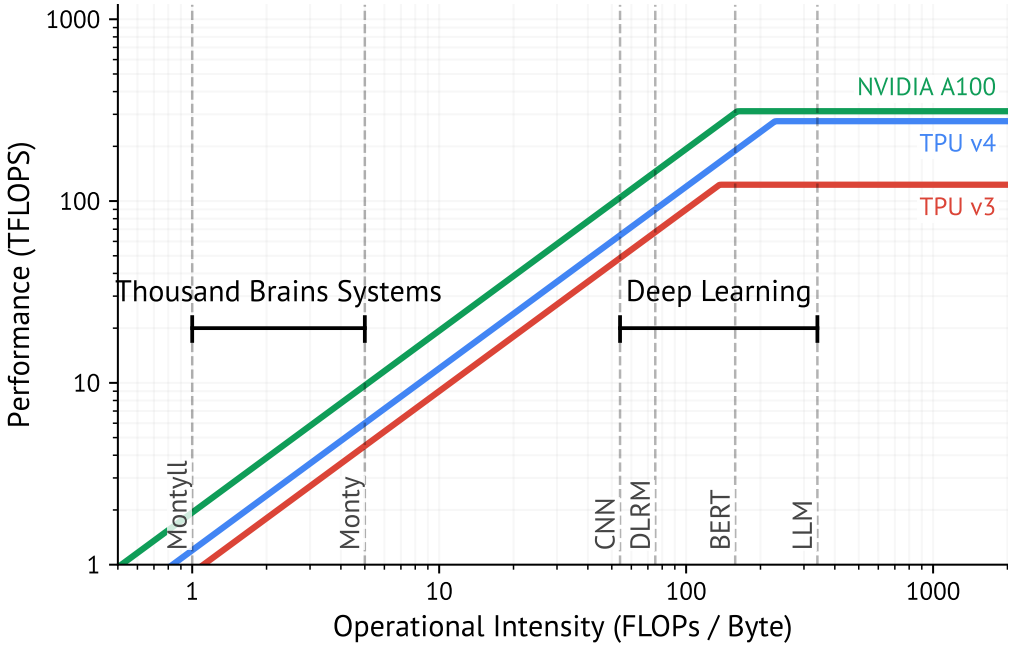


Figure 3.3: GPU/TPU roofline plot. Deep learning methods shows high operational intensity, especially as methods progressed from e.g. CNNs [22] to Transformers [79]. GPUs (e.g. A100) and TPUs exhibit high performanced (TFlops/s) at high operational intensities, and lower performance at lower operational intensities. Thousand Brains Systems exhibit low operational intensity due to their auto-regressive nature. Extended from [80].

In GPUs and TPUs, inputs are streamed from memory and reused by the same block of weights stored in shared memory, as highlighted in figure 3.4. As such, larger batches of inputs are required for high compute utilization in GPUs and TPUs.

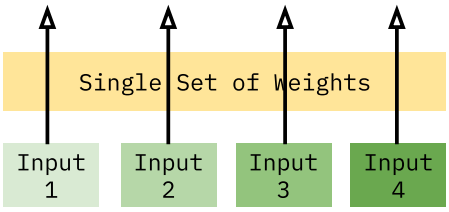


Figure 3.4: Batching in GPUs. To hide memory access latencies, the GPU architecture relies on kernels that reuse the same set of weights across many inputs, also called batching.

3.2.2 Thousand Brains Systems' operational intensity

We calculate the operational intensity of Monty and MontyII, two Thousand Brains Systems implementations mentioned in section 3.1.3. We show their standing on a roofline model in figure 3.3.

Recall from section 3.1.2 that Thousand Brains Systems computation happen in an auto-regressive loop. Thus, we cannot consider batching across steps, and have to consider the maximum that can

be reused within a step to compute the operational intensity.

Monty exhibits potential for small levels of data reuse with matrix-vector operations that can be batched over the number of objects learnt. However, the computation time is dominated by KD-Tree knn search primarily consisting of pointer-chasing operations. Such computation not only presents low operational intensity, it causes branch divergence which further presents a great barrier to GPU performance [82]. As such, the operational intensity of Monty is less than 10 ops/byte.

Montyll's operational intensity is straightforward to compute as it is based on calculating activations and predictions via a sparse connection matrix, where each connection is used once per step. Therefore, the operational intensity of Montyll is around 1 op/byte.

3.2.3 Conclusion

As we saw, GPUs, TPUs and systolic arrays suffer from low processing speed (ops/second) for workloads with low operational intensity. On the other hand, Thousand Brains Systems exhibit low operational intensity, in the 1-10 ops/byte range. As such, GPU and TPU systems have limited capacity to revolutionize the applicability of Thousand Brains Systems to larger scale and more complex problems.

3.3 CPUs

> *Learning modules of Thousand Brains Systems are semi-independent self-contained processing units. Modern multicore systems can contain up to hundreds of cores. Why can't we just use modern multicore CPUs to scale up Thousand Brains Systems?*

We will see in this section that the biggest barrier to scaling up Thousand Brains Systems on CPUs is not necessarily the number of cores and the available operations per second, but the latency and bandwidth to the main memory (DRAM).

Recall from section 3.1.1 that we wish to scale up Thousand Brains Systems to thousands of learning modules. This means that, at every step, data that is relevant to the recognition of an object, the recognition of an environment, or the generation of a motor command must be fetched and brought to the cores for computation. But how much data are we actually talking about here? We first consider the data movement costs for a single learning module and discuss the implications for a systems with thousands of learning modules.

3.3.1 Thousand Brains Systems data movements

Theoretical data movement volume. Current stereological estimates suggest the human neocortex contains approximately 20 billion neurons [83] and roughly 150 to 164 trillion synapses [84]. A single neuron in the neocortex is expected to be connected to thousands ($\approx$ 7'500) of other neurons. As mentioned in section 3.1.1, these neurons are functionally organized in 200'000 cortical columns. While it is hard to classify synaptic connection into inter-column (long range)

and intra-column, we can expect roughly 100'000 neurons and 750 million synapses per cortical column. An interesting detail is that the size of cortical columns stay approximately consistent across species whose brain differ in volume by a factor of 1000 [12]. This means that the theoretical footprint of a cortical column in the brain is around 3 GB, assuming that each connection has to address a space of 2^{24} neurons and 8 bits are used to represent the connection myelination and strength. If every connection is used to determine the state of neurons at a specific step, then the theoretical data movement volume is 3 GB per step and per column.

Monty data movement volume. Monty's current learning module implementation is based on explicit graphs in 3d space. As such, the exact footprint of a learning module in Monty depends on the underlying object model used, the number of learnt graphs and the number of learnt observations per graph. In practice, we find that the learning module footprint is around 1-100 MB, and that the data movement volume is around 5-100 MB per step per learning module.

Montyll data movement volume. Montyll was designed to strike a compromise between the theoretical footprint and the limited memory available to us in the Processing-in-Memory chips as mentioned in sections 3.7 and 4.2. Each learning module in Montyll contains 20'000 neurons, where each neuron can develop connections with up to 480 other neurons, for a total of 9.6M connections. Montyll's memory footprint is 50 MB and its data movement volume is around 100 MB per step per learning module.

Table 3.1: Data movement volume of Thousand Brains Systems *per step* for different numbers of learning modules. Biological equivalents are there to provide a rough idea, not a strict comparison.

Modules	Theoretical	Monty	Montyll	Biological Equivalent
1	3 GB	5–100 MB	100 MB	1 Cortical Column
2'500	7.5 TB	12.5–250 GB	250 GB	Cat cortex (*Felis Catus*) [85]
200'000	600 TB	1–20 TB	20 TB	Human neocortex

Data movement volume at scale. Data movement volume increases linearly with the number of learning modules in a Thousand Brains System. We see that the data movement volume per step increases to pretty hefty sizes quite quickly in table 3.1. For a system with 2'500 learning modules, akin to the size of the cerebral cortex of the cat (*Felis Catus*) with 250 million neurons [85], Thousand Brains Systems need tens to hundreds of gigabytes of data moved. To reach the full size of a human neocortex at about 200'000 learning modules, we need tens of terabytes of data movement *per step*.

3.3.2 Can CPUs scale Thousand Brains Systems?

There are two barriers that strongly limit CPUs from scaling Thousand Brains Systems to thousands of learning modules: core oversubscription and the memory bandwidth wall. To better explain

how these problems arise even on some of the most power CPU-based hardware, we include table 3.2 that presents a range of CPU-based computing platforms. We purposefully include low-end to high-end robotics hardware alongside a high-end workstation CPU, as one of our goals is to find the best computing platform for embodied sensorimotor intelligence.

Table 3.2: Robotics and Workstation Hardware Specifications. A comparison of core counts and memory bandwidth across a spectrum of CPU-based computing platforms.

Platform	Cores	LLC	Mem. Bandwidth	Power	DRAM
Jetson Nano [86]	4	2 MB	25.6 GB/s	5–10 W	4 GB
Jetson Orin Nano [87]	6	4 MB	102 GB/s	7–15 W	8 GB
Jetson AGX Orin [88]	12	6 MB	204.8 GB/s	15–60 W	64 GB
Jetson Thor [89]	14	16 MB	273 GB/s	40–130 W	128 GB
Threadripper 7995WX [90]	96	384 MB	$\approx$ 384 GB/s	$\geq$ 350 W	$\leq$ 2 TB

Core oversubscription and time-sharing. As shown in table 3.2, a high-end workstation possesses 96 cores, and a high-end robotics system (AGX Thor) possesses 14. Whether we look at Thousand Brains Systems of cat-scale (2'500 learning modules) or human-scale (200'000 learning modules), there is bound to be a massive imbalance between the number of learning modules and the number of physical cores. For example, to run 2'500 modules on 96 or 14 cores, the CPU must employ time-sharing, where each physical core is responsible for sequentially computing 26 and 178 modules respectively per time step. This serialization destroys the system's latency.

The memory bandwidth wall. The most critical bottleneck is not the compute speed, but the sheer volume of data movement. As established in table 3.1, a Montyll implementation at the scale of a cat (2'500 modules) requires moving approximately 250 GB of data *per step*.

Comparing this requirement to table 3.2, we see a gap between the required data movement and the available memory bandwidth. The NVIDIA Jetson Thor offers 273 GB/s, and the Threadripper workstation offers $\approx$ 384 GB/s. These would cap the runtime of cat-scale Montyll to around 1 step per second on account of data movements alone.

Because the combined footprint of the learning modules well exceeds the l3 caches, the CPU cannot hide this latency. Every step forces a full flush and reload of data from DRAM. No matter how fast the CPU cores can compute the relevant operations, time to bring data assuming maximum bandwidth is already a limiting factor. Scaling to a human-sized system (200'000 modules) would mean spending 53s on data movements alone on the best hardware of table 3.2 at peak bandwidth.

Multi-node systems. If single-node CPUs are not sufficient to scale to thousands of learning modules, why not use many nodes at once? The answer partly lies in our goal to find the best computing platform for real-time embedded sensorimotor learning. We talk about this in more detail in section 3.6.

3.4 Accelerators

Any modern System-on-a-Chip has AI accelerators, why can't there simply be Thousand Brains Systems accelerators in-logic?

In-logic accelerators for Thousand Brains Systems have been proposed [91][92][93], although they are not end-to-end designs and only cover specific elements of Thousand Brains Systems implementations.

Scaling issues. While these designs might help run small scale Thousand Brains Systems faster and in a more energy-efficient way, they suffer from the same data movement problems at larger scale that we mention in the section 3.3.1 on data movements in CPUs. In particular, our observations from table 3.1 about data movement volume for Thousand Brains Systems with 2'500 and 200'000 learning modules still apply here. Connection data still overwhelms the caches, and thus needs to be brought from higher-latency and lower-bandwidth main memory, as highlighted in figure 3.5. These designs are not geared to answer the problems that arise when we massively scale up Thousand Brains Systems.

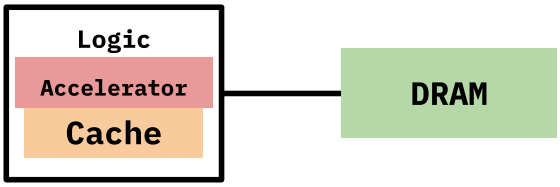


Figure 3.5: By having to bring the connection data from main memory, In-logic accelerators suffer from the same data movement bottlenecks as any core-centric architecture.

Programmability. Another problem with accelerators is that they are not programmable. However, Thousand Brains Systems are still a very active area of research that is arguably in its infancy. Researchers need programmability to allow for some amount of freedom in designing and experimenting with new algorithms and model architectures. We expect the exact underlying algorithms and methods to change massively, especially to bridge the gap between current Thousand Brains Systems [16] and the long term goal of the Thousand Brains Project [17], which aims to integrate elements of low-level neocortical processing [49][52][55] into Thousand Brains Systems.

3.5 Neuromorphic hardware

Neuromorphic computing is a computing approach inspired by the human brain's structure and function. Why can't past and current neuromorphic computing efforts be used to scale Thousand Brains Systems?

While it sounds like existing neuromorphic hardware [94] supports neuron models which align with Montyll [95], more than 90% of neuronal connections occur at distal dendrites [96] and have a completely different spiking mechanism (NMDA spikes) [97] that neuromorphic chips do not support. This significant gap can be closed by supporting multi-compartment or pyramidal [98] neuron models, but it would come at a large cost in runtime and parallelism. Some neuromorphic hardware like BrainsScaleS-2 offers limited support to multi-compartment neuron models [99], but it limits the number of connections per neurons to tens instead of the thousands required to represent its biological counterpart with fidelity [49]. Efforts also remain limited in their ability to learn and form new connections [100].

Device Limitations. While the field of neuromorphic hardware is large, the main emerging memory technologies (RRAM, PCM, MRAM) have severe limitations. Despite their potential for high density, these devices are notoriously unreliable: they often behave inconsistently from one to the next, their stored values tend to change on their own over time, and they struggle to strengthen or weaken connections with equal precision [101]. Crucially, limited write endurance (often capping between 10^6 and 10^9 cycles for PCM and RRAM) severely restricts the capacity for the continuous, rapid synaptic updates required for lifelong on-chip learning. They nonetheless represent an interesting and significant opportunity to scale workloads that can inherently tolerate noise, which we discuss in section 8.2).

Programmability. The design choices at the core of neuromorphic chips greatly constrain the applications and limit the algorithm design space in a field where algorithmic breakthroughs are needed to make significant progress. Moreover, Thousand Brains Systems like Monty are based on graphs of locations in 3d cartesian space, which is wholly incompatible with the core operations performed by neuromorphic chips.

3.6 Datacenters & Computer Clusters

Datacenters & Compute Clusters can be used to massively scale up applications and simulations, why can't it do the same for Thousand Brains Systems?

There are a couple reasons why one might search for a better scaling solution than throwing ever larger warehouse-scale computer at the problem. We will go over these reasons in this section.

Goal incompatibility. One of the goals of Thousand Brains Systems [17] is to design sensorimotor learning systems that can perform rapid continual learning. In particular, to design systems that learn by interacting with their environment rather than through back-propagating losses on a big static dataset like deep learning systems. Another goal, although similar, is to design algorithms that learn while in deployment and can be run on the edge for robotics applications. These goals are incompatible with a training and deployment model that relies on warehouse-scale computers because of real-time processing constraints.

What's needed for research. Regardless of the goal incompatibility, big computer clusters are expensive. Needing such big clusters to conduct research on Thousand Brains Systems effectively limits the pool of research groups willing and able to invest these sums. An analogy can be made with the hardware that enabled large scale deep learning systems. In 2009, researchers at Stanford where among the first to advocate for using graphics processors to train deep learning systems [102]. We know now how influential this idea has been, although it has mostly been popularized by the AlexNet paper [22], which actually demonstrated the power of GPUs to train large scale deep learning networks. There was another interesting paper from 2012, which trained large scale deep learning systems on tens of thousands of CPU cores [103]. Although successful, the general idea of the paper has been less influential: no one really trains models on clusters of CPUs. Although the success of GPUs over CPUs to train neural networks cannot be traced to a single term, it is undeniable that a big factor has been the scale to cost advantage that GPUs offer. We believe that Processing-in-Memory systems can afford researchers the same scale to cost advantage over competing hardware.

3.7 Processing-in-Memory (PiM)

How can Processing-in-Memory help scaling up Thousand Brains Systems?

3.7.1 UPMEM PiM Architecture

The Processing-in-Memory design that we consider in this paper is based on the chips commercialized by the company UPMEM, released in 2021 called variant v1B. An UPMEM module is based on a standard DDR4-2400 DIMM, and multiple memory channels can be dedicated to UPMEM modules. Within a module is a set of DPUs, which can all operate in parallel. UPMEM chips are an example of 2d-integrated PiM designs [104], where the PiM cores (called DPUs, for Data Processing Units) are located near the memory banks, as highlighted in figure 3.6. There can be up to 2560 PiM cores operating in parallel on a full system.

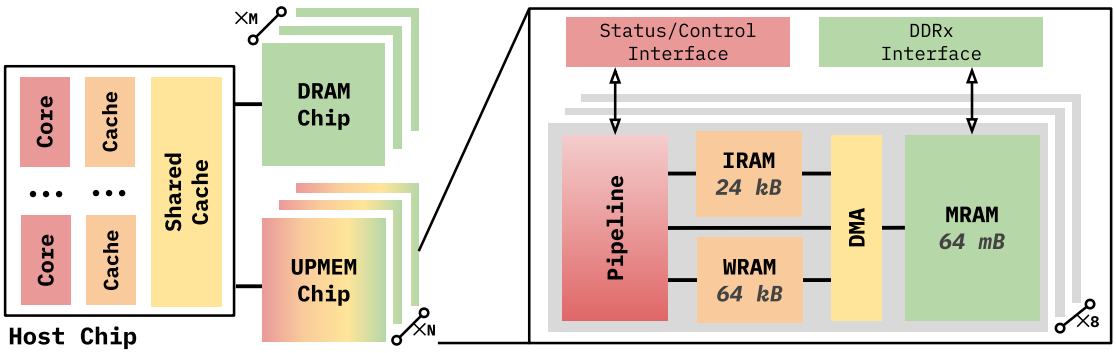


Figure 3.6: UPMEM Processing-in-Memory Architecture. UPMEM DPUs are RISC cores located on the memory chip, near the banks.

In aggregate, these chips are capable of performing 1.02 trillion operations per second, of accessing their bank memory (MRAM) at a bandwidth of 1.28 TB/s and of accessing their scratchpad memory (WRAM) at a bandwidth of 8.19 TB/s. Before looking into more detail, these are the figures that best explain how Processing-in-Memory can handle the sheer aggregated volume of data movement that large scale thousand brains systems exhibit as outlined in table 3.1, as well as the aggregated number of operations.

Each PiM core is a 32-bit in-order RISC processor, running at 400 MHz and fine-grained multi-threaded (FGMT). Multiple tasklets (i.e. threads) (from 1 to 24) can run on each PiM core. Because it is FGMT, there can only be a single instruction from the same tasklet traveling through the 14-stage pipeline at once. Therefore, it is important to parallelize any piece of code across tasklets to run at a maximal number of instructions per cycle.

A PiM core has zero-cycle latency access to the register file and single-cycle latency access to a 64 KiB scratchpad memory called WRAM. Data from the 64 MiB memory bank, called MRAM, first needs to be transported to the WRAM for access. These accesses need to be explicitly handled by the programmer through a DMA engine. What does not need to be handled by the programmer is how the instructions are fetched from the MRAM to the 24 KiB IRAM, which can hold 4000 instructions at once, each encoded on 6 bytes.

Because of the constraints of designing logic using a DRAM manufacturing process, the PiM cores suffer from a few performance issues. In his Hot Chips 31 presentation [38], F. Devaux from UPMEM cites the following figures for the DRAM manufacturing process. The transistors are three times (3x) slower than for a logic process of the same size, the logic is ten times (10x) less dense than an ASIC process and the routing density is dramatically lower with 3 metals only for routing (vs. 10+ for logic process), and pitch x4 larger. The chips thus only run at 400 MHz compared to the 2-3 GHz that any modern processor can handle. They also do not support floating-point operations and only support integer multiplications of small precision (8 bits $\times$ 8 bits).

While these PiM chips suffer some serious limitations, they have the potential to do parallel computing on heterogeneous weights on 2560 cores at once. We hope to give a sense that our target application Montyll (section 4) plays nicely with the limitations of these PiM chips. In particular, the learning and activation rules that Montyll follows (section 4.1.3) only require low-precision integer arithmetic, while the memory footprints (table 4.4) and access patterns fit nicely within the WRAM-MRAM transfer paradigm. Dividing the work into tasklets is straightforward and those activation rules that require high precision floating point operations (e.g. boosting in pooler, section 4.1.3) can be converted into DPU friendly option (section 4.2).

3.7.2 Thousand Brains Systems on PiM: a rough analytical model

The following derivations are made to give a flavor of why, without going into too much detail, the Processing-in-Memory equation works for Montyll. In the next section 4, we dive into the details and show the exact benefits in the results section 6.

In Montyll, the model footprint is around 50 MiB, with a data movement per step of 50-100 MiB and an operational intensity of 1, entailing around 50-100 million operations (MOps) per step. Let's say that on average, it is 75 MiB and 75 MOps per step. According to [37], if we transfer the

relevant data from MRAM to WRAM well, we end up doing it at around 500 MiB/s. If we apply to bandwidth figure to our 75 MiB of data movement per step, we'd end up spending 150 ms on data transfers. Furthermore, the core runs at 400 MHz and we have 75 MOps to execute, it will end up taking 188 ms. This makes a total processing time of 338 ms per step. This might seem like a lot, but it opens up an opportunity for multiple steps a second to be executed across 2560 learning modules. Most importantly, if we recall from section 3.3.2 on single-node CPUs, data movements alone for 2500 learning modules cost nearly a second for the high-end workstation.

Now we have to be careful about our estimates above. In particular, the DPUs do not necessarily execute 1 "useful" operation per cycle, it all depends on the instruction mix, how well the work is parallelized and how many accessory instructions are needed to setup the execution of said operation. Furthermore, MRAM-WRAM transfer latencies are not taken into account. We are also not taking the costs of eventually communicating states across learning modules that are located on different DPUs. Thankfully, this section only serves to give a rough idea, and our thorough implementation (section 4) and experiments (sections 5, 6) will give more precise data (section 6) to evaluate the benefit of PiM to scale Thousand Brains Systems.

Chapter 4

A Thousand Brains on a Thousand Chips

4.1 Montyll: a Thousand Brains Systems implementation in C

Montyll stands for Monty low-level, as it is inspired from N. Leadholm et al.'s implementation named *Monty* [16], but integrates elements of low-level neocortical processing e.g. more complete neuron models [49][51] and grid cells to represent location [55][56]. Montyll was explicitly designed to be aligned with the long term goals of the Thousand Brains Project [17]. It is written in C for performance and to be amenable to architectural studies. We open source Montyll at https://github.com/Xavier0301/cmontyll.

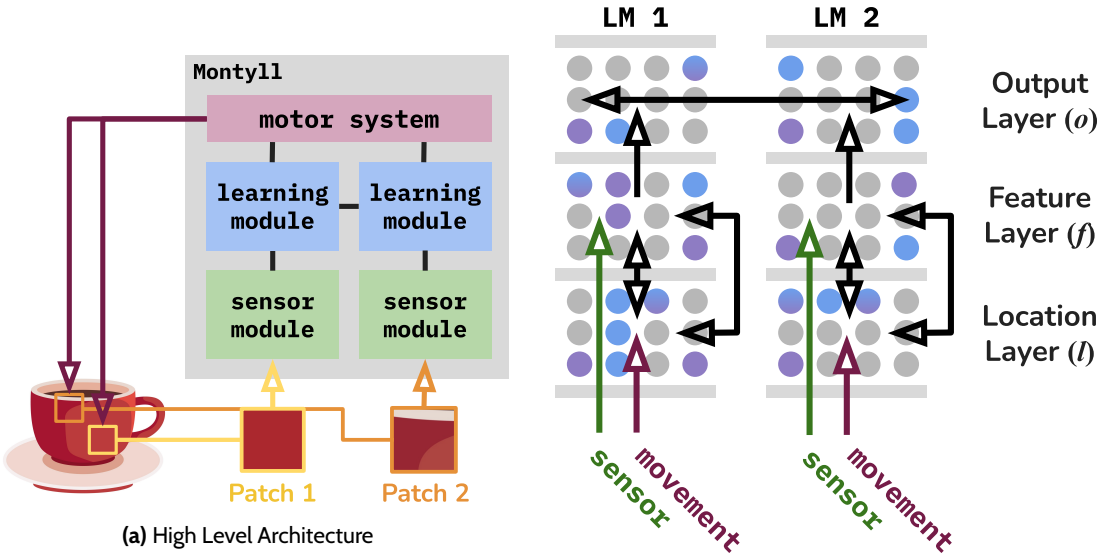
Our goal is to show that Thousand Brains Systems can be uniquely scaled on Processing-in-Memory hardware. As such, Montyll is not designed to be useful or solve a specific problem. It is rather designed to be a good representation of Thousand Brains Systems computations, especially as they integrate more low-level elements of neocortical processing. We leave a port of Monty, a Thousand Brains Systems implementation that solves concrete 3d object recognition problems, to Processing-in-Memory systems as future work.

4.1.1 Architecture

The architecture of Montyll follows the high-level architecture of Monty, with sensor modules and learning modules. In Montyll, these sensor and learning modules are implemented with low-level neocortical processing elements. What that means, we hope, will become clear in this very section. We summarize the architecture of Montyll in figure 4.1.

The first important concept to introduce here are *sparse distributed representations* (SDRs), modeled after representations in the neocortex. *Sparse* because only a small percentage of neurons are active while the rest remain inactive [105]. *Distributed* because a representation does not rely on a single neuron but on a population of neurons. As such SDRs could also be called "a sparse distributed code of cellular activations", and have been studied to have great noise tolerance and vast representational capacity in [106], [53], [54] and [49].

Let's now shift our attention to the neuron model that will underlie all the computations in Montyll. Departing from the point neuron [107] used in conventional neural networks [18] as well as spiking neural networks [108], we adopt the active dendrite model proposed by Hawkins in 2016 [49]. This model distinguished between proximal inputs, which drive action potential spikes, and distal inputs, which drive dendritic spikes. When a distal dendritic segment recognizes a pattern, it



(a) High Level Architecture

(b) Learning Module (LM) Architecture

Figure 4.1: Architecture of Montyll. The learning modules in Montyll are implemented using a collections of HTM networks. Sensor modules produce SDRs representing the sensory patch inputed to them. Horizontal arrows represent context connections, attached at distal sites of neurons and producing NMDA spikes. Vertical arrows represent feedforward connections, attached at proximal sites of neurons and producing action potentials.

generates a local NMDA spike which depolarizes the cell body without causing an action potential [109][97]. Integrating dendritic spikes in neuron models is especially important if we consider that 90% of the connections in the neocortex happen at distal locations, and only 10% at proximal locations [96]. In 2016, Hawkins proposed that a dendritic spike places the cell in a predictive state [53], which would cause it to fire earlier than its neighbors thus inhibiting their activity.

This model has been the basis for the so-called *hierarchical-temporal memory* networks developed in the same period [50][51], which we base many of our modules' components on. In particular, sensor modules are responsible for encoding the raw input data and mapping the encoding into an SDR, based on the HTM spatial pooler [52]. Learning modules are composed of a feature, a location and an output layer. The feature layer of the LMs extends works that describe layer 4 of the neocortex [49][50][51]. The location layer of the LMs extends works on integrating grid cells to represent locations in layer 6a of the neocortex [55], notably with modifications that follow the new mini-column structures outlined in [110]. The output layer of the LMs extends works that describe layer 3 of the neocortex [111]. We leave to future work the integrations of layers 5 on motor commands (i.e. displacements) and 6b on orientation representation into the network, as well as integrating more accurately the role of the thalamus as outlined in [15].

We now spend the rest of the section describing the learning and activation rules that regulate our sensor and learning modules, as well as their algorithmic and space complexity.

4.1.2 Notation

The notation used in this section, as well as the activation and learning rules used in our implementation are an extension of previous works listed in the above section 4.1.1. We unify these works into a single network, and provide novel ways to describe certain mechanisms that we hope can give a clear overview of inter-connected HTM networks. We summarize the symbols in table 4.1.

Table 4.1: Summary of symbols used in the activation and learning ruless

Property	**Feature (f)**	**Location (l)**	**Output (o)**	**Pooler (p)**
Layout	$M \times N$ cells	$M \times N$ cells	M cells	M cells
States				
Activation	$a_{ij}^{t[f]} \in \{0,1\}$	$a_{ij}^{t[l]} \in \{0,1\}$	$a_{i}^{t[o]} \in \{0,1\}$	$a_{i}^{t[p]} \in \{0,1\}$
Prediction	$\pi_{ij}^{t[f]} \in \{0,1\}$	$\pi_{ij}^{t[l]} \in \{0,1\}$	$\pi_{i}^{t[o]} \in \{0,1\}$	—
Spiking Segments	$\tau_{ij}^{t[f]}$	$\tau_{ij}^{t[l]}$	$\tau_{i}^{t[o]}$	—
Connections				
Feedforward	—	—	$\mathbf{F}_{i}^{t}$	$\mathbf{F}_{i}^{t}$
Contextual	$\mathbf{D}_{ijd}^{t}$	$\mathbf{D}_{ijd}^{t}$	$\mathbf{D}_{id}^{t}$	—
Incident Index (I)	(f,i,j)	(l,i,j)	(o,i)	(p,i)
Overlap	$\mu^{t_0}(\mathbf{G}, p^*)$ counts active connections with concomitantly active presynaptic cells on connection structure $\mathbf{G}$ (where $\mathbf{G}$ is $\mathbf{F}$ or $\mathbf{D}$).			

The feature, location and output layers as well as the pooler from the sensor module share a similar structure and hence have similar activation and learning rules. In brief, they all use similar structures for activation a_{ij}^{t} and prediction π_{ij}^{t}. The pooler and the output layer use similar structures for feedforward input $\mathbf{F}_{i}^{t}$. All layers in the learning module use similar structures for contextual input $\mathbf{D}_{ijd}^{t}$.

We borrow and slightly modify the notation used in [49]. Let M be the number of mini-columns and N be the number of cells in a layer. The feature and location layer both are a set of $M \times N$ cells (i.e. neurons), while the output layer is composed of M cells. The pooler is composed of M cells which match the number of columns in the feature layer.

Let $\pi_{ij}^{t} \in \{0,1\}$ be the prediction state and $a_{ij}^{t} \in \{0,1\}$ be the activation state of the j-th cell of the i-th mini-column of the layer at time step t. Let $\mathbf{A}^{t} = \{a_{ij}^{t}\}_{i,j}$ and $\mathbf{\Pi}^{t} = \{\pi_{ij}^{t}\}_{i,j}$ denote the activation and prediction matrices of the layer at time step t.

Let $\mathbf{F}_{ij}^{t}$ be the set of incident feedforward connections to the j-th cell of mini-column i of the layer at time step t. Each element of $\mathbf{F}_{ij}^{t}$ is of form (I, p_t) where I represents the index of the incident cell and p_t represents the permanence value, which represents how reinforced the connection is.

If s neurons are proximally connected to cell i, then $\mathbf{F}^t_{ij}$ will contain s elements. A connection is considered active if the permanence value is above a certain pre-determined threshold $p^* \in [0; 1]$.

Let $\mathbf{D}^t_{ijd}$ be the set of incident context connections to the d-th segment of the j-th cell of the i-th mini-column of the layer at time step t. Each element of $\mathbf{D}^t_{ijd}$ is of form (I, p_t) where I represents the index of the incident cell and p_t represents the permanence value. If s neurons are distally connected to segment d, then $\mathbf{D}^t_{ijd}$ will contain s elements. A connection is considered active if the permanence value is above a certain pre-determined threshold $p^* \in [0; 1]$.

The indices above I are generally of form (x, i, j), where the first dimension represents the layer, the second represents the mini-column and the third represents the cell. That is because cells in a certain layer can receive context connections from cells in another layer. For example, the location layer provides context to the feature layer and vise-versa, on top of the location and feature layers providing context to themselves.

Our notation above departs from the notation from [49], where $\mathbf{D}^t_{ijd}$ can be mathematically described as a sparse tensor. If s neurons are distally connected on the segment d of the j-th cell of mini-column i, then the tensor $\mathbf{D}^t_{ijd}$ would contain s non-zero elements. And we'd have the following equation for active connections: $\widetilde{\mathbf{D}}^t_{ijd} = \{\mathbb{I}\{(\mathbf{D}^t_{ijd})_{klm} > d_0\}\}_{klm}$.

We let $\mathbf{T}^t = \{\tau^t_{ij}\}_{i,j}$ be the number of spiking segments on a cell at time step t. The number of spiking segments $\mathbf{T}^t$ is related to the predictive state of each cell $\mathbf{\Pi}^t$ in a way that will become clearer in the next section 4.1.3 on activation rules. To further simplify the activation rules, we introduce $\mu^{t_0}(\mathbf{G}, p^*)$, which describes the number of active connections with concomitantly active incident (presynaptic) cells, where $\mathbf{G}$ can be $\mathbf{F}^t_i$ or $\mathbf{D}^t_{ijd}$. We call $\mu^{t_0}(\mathbf{G}, p^*)$ the *overlap*, to mean the overlap between connection activity and presynaptic cell activity.

Finally, when it is not clear from context, we clearly distinguish the various symbols between the feature, location and output layers. For example, the activations in the feature layer are $\mathbf{A}^{t[f]} = \{a^{t[f]}_{ij}\}_{ij}$, while they are respectively $\mathbf{A}^{t[l]} = \{a^{t[l]}_{ij}\}_{ij}$, $\mathbf{A}^{t[o]} = \{a^{t[o]}_{i}\}_{i}$ and $\mathbf{A}^{t[p]} = \{a^{t[p]}_{i}\}_{i}$ for the location layer, output layers and pooler. We drop the j index for the output layer since it can be viewed as a layer of M mini-columns with 1 cell each.

4.1.3 Activation and learning rules

In table 4.2, we try to summarize the common patterns of activation, prediction and learning accross the HTM networks in the pooler, feature layer, location layer and output layer. The activation, prediction and learning rules are largely borrowed and extended from previous works. See section 4.1.1 for more details.

SM: Pooler

The pooler is made of M cells and the activation of the pooler is a locally inhibitory process that selects the top k cells that best match the feedforward input. First, feedforward input from an encoded raw input $(a^{t[in]}_i)_i$ is integrated to determine which cell could become active. Then, only the k cells with biggest overlap become active.

Table 4.2: Summary of the rules that regulate activation, prediction and learning in HTM networks.

Mechanism	Mathematical Formulation	Remarks
Synaptic Overlap	$\mu^t(\mathbf{G}, p^*) = \lvert\{(I, p_t) \in \mathbf{G} \mid p_t \geq p^*, a_I^{t[x]} = 1\}\rvert$	$\mathbf{G}$ can be proximal ($\mathbf{F}$) or distal ($\mathbf{D}$). Counts active connections with active cells.
Segment NMDA Spike	$\tau_{ij}^t = \sum_d \mathbb{I}\{\mu^{t-1}(\mathbf{D}_{ijd}^t, p^*) \geq \theta_d\}$	**Distal Context:** Occurs locally on a segment when the contextual overlap μ exceeds threshold θ_d.
Cell Depolarization	$\pi_I^t = 1$ if Condition(τ_I^t)	**Feature/Location:** Primed if ≥ 1 segment spikes. **Output:** Primed if in Top-k cells by segment spike count.
Somatic Action Potential	$a_I^t = \text{Decision}(\mu^t(\mathbf{F}_I^t, p^*), \pi_I^{t-1})$	**Pooler:** Enough ffw overlap and competitive Top-k overlap ranking. **Feature/Location:** Bursting logic if no cell is depolarized. No ffw connection $\mathbf{F}^t$. **Output:** Enough ffw overlap and predicted.
Learning Rule	$p_{t+1} = p_t + p^+$ if incident $a_I^{t[x]} = 1$, else $p_t - p^-$. This Hebbian update is applied to all segments in the update set X^t (correctly predicting segments or best-match segments during a burst).	

We start by calculating the overlap for each cell i, which is the number of active connections that concomitantly have an active incident input:

$$\mu^t(\mathbf{F}_i^t, p^*) = \lvert\{e\colon e = ((in, i_0), p_t) \in \mathbf{F}_i^t,\ p_t \geq p^*,\ a_{i_0}^{t[in]} = 1\}\rvert \tag{4.1}$$

The set of output cells with enough feedforward input described by $\widetilde{a}_i^t \in \{0, 1\}$ is determined as follows:

$$\widetilde{a}_i^t = \begin{cases} 1 & \text{if } b_i \mu^t(\mathbf{F}_i^t, p^*) \geq \theta_f \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

where θ_f is a pre-determined threshold for the action potential, and b_i are *boosting factors* which computation will be detailed bellow. We can now calculate the activations as follows:

$$a_i^t = \begin{cases} 1 & \text{if } \widetilde{a}_i^t = 1 \text{ and } \mu^t(\mathbf{F}_i^t, p^*) \geq \mu^t(\mathbf{F}_{\xi_k^t}^t, p^*) \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

where ξ_k^t is the cell index which feedforward overlap is the k-th biggest. For learning, we update all cells that have become active $X_t = \{i\colon a_i^t = 1\}$. For all cells i in X_t, we go through the

connections $((in, i_0), p_t) \in \mathbf{F}_i^t$ and we adjust the permanence p_t by a small increment p^+ or decrement p^- depending on the state of the input $a_{i_0}^{t[in]}$ as such:

$$p_{t+1} = \begin{cases} p_t + p^+ & \text{if } a_{i_0}^{t[in]} = 1 \\ p_t - p^- & \text{if } a_{i_0}^{t[in]} = 0 \end{cases} \tag{4.4}$$

The boosting factors b_i are calculated based on the activity levels of a cell compared to other cells in the layer. The general goal is the following. We want to lower the spiking threshold for comparatively inactive cells and raise the spiking threshold for overly active cells. We first start by calculating the time-averaged activation level A_i^t for each cell i over the last T steps:

$$A_i^t = \frac{1}{T}((T-1)A_i^{t-1} + a_i^t) \tag{4.5}$$

Which gives us an estimate for the expected time-averaged activation level across the M cells of the pooler:

$$\overline{A^t} = \frac{1}{M}\sum_i A_i^{T,t} \tag{4.6}$$

Finally, the boosting factor of cell i is calculated based on the difference between $\overline{A^t}$ and its own time-averaged activation level A_i^t:

$$b_i = e^{-\beta(A_i^t - \overline{A^t})} \tag{4.7}$$

where β is a pre-determined parameter that can tune the boosting effect. According to the paper that originally derives the above formulas [52], the above boosting mechanisms have been inspired by studies on homeostatic regulation of neuronal excitability [112] and previous models of homeostatic synaptic plasticity [113][114].

LM: Location layer

We have integrated the new mechanisms outlined in [110] into previous work [55] that explain how locations are encoded in layer 6 of the neocortex through an HTM-like network with grid cells. In particular, the grid cell "modules" from [55] have been replaced by the mini-column structure that was first used to describe layer 4 in [55] and [111].

The activations are determined in two steps. The first step shifts the mini-column activity following an input movement. The second step activates specific cells of each now-active mini-column depending on their predictive state.

Let $\widetilde{\mathbf{A}}^t = \{\widetilde{a}_i^t\}_i$ describe which mini-columns are active at step t. Let $(x^t, y^t) \in \mathbb{Z}^2$ be the input movement at step t. If for a mini-column i, we have $\widetilde{a}_i^{t-1} = 1$, we need to determine the index i_* of the column which will become active following the shift from the input movement (x^t, y^t). Mini-columns can be viewed as a set of $P \times Q$ mini-columns, where $P, Q | M$. We define the mapping $\varphi : [\![0; M]\!] \to \mathbb{Z}^2$ such that for each index i the coordinate (x_i, y_i) is given by $\varphi(i) = \left(i \bmod Q, \frac{i-(i \bmod Q)}{P}\right)$. We calculated the shifted coordinates as $(x_*, y_*) = (x_i + x^t \bmod Q, y_i + y^t \mod P)$ and the resulting index as $i_* = x_* + y_* Q$. We then finally set $\widetilde{a}_{i_*}^t = 1$.

We set all mini-column that were left untouched by this process to zero. We leave to future work how the mechanism described in this paragraph can be tweaked and expanded to work for an input vector in $\mathbb{R}^3$.

In the second step, we activate only the cells in the active mini-columns that where in a predictive state. If no cell in an active mini-column is predicted, we activate all cells in the mini-column. The rule is summarized in the following equation:

$$a_{ij}^t = \begin{cases} 1 & \text{if } \widetilde{a}_i^t = 1 \text{ and } \pi_{ij}^{t-1} = 1 \\ 1 & \text{if } \widetilde{a}_i^t = 1 \text{ and } \forall j, \pi_{ij}^{t-1} = 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.8}$$

We let θ represent the NMDA spiking threshold of each segment, i.e. the minimum number of active connections with active incident cells required to cause a spike. To calculate he number of spiking segments τ_{ij}^t, we first need to calculate the overlap $\mu^t ijd$, which counts the number of active connections with concomitantly active incident cells on a specific segment:

$$\mu^{t-1}(\mathbf{D}_{ijd}^t, p^*) = |\{e \colon e = ((x, i_0, j_0), p_t) \in \mathbf{D}_{ijd}^t,\ p_t \geq p^*,\ a_{i_0 j_0}^{t-1[x]} = 1\}| \tag{4.9}$$

We can now calculate the number of spiking segments as follows:

$$\tau_{ij}^t = \sum_d \mathbb{I}\{\mu^{t-1}(\mathbf{D}_{ijd}^t, p^*) \geq \theta_d\} \tag{4.10}$$

It is the number of segments for which the count of active connections with active incident cells is above the NMDA spiking threshold θ_d. The predictive state of a cell can now be calculated as:

$$\pi_{ij}^t = \begin{cases} 1 & \text{if } \tau_{ij}^t \geq 1 \\ 0 & \text{otherwise} \end{cases} \tag{4.11}$$

In other words, if a cell had at least one of its segments spike, it is put in a predictive state. We can now shift our focus to the rules that regulate learning. There are two cases where we consider updating the permanence values of a segment d.

In the first case, if a cell was correctly predicted (i.e. was predicted and became active), we update the connections from all spiking segments, or in equation form we update all segments in the following set:

$$Y^t = \{(i, j, d) \colon \pi_{ij}^t = 1,\ a_{ij}^t = 1,\ \mu^{t-1}(\mathbf{D}_{ijd}^t, p^*) \geq \theta_d, \quad i \in [\![1; M]\!],\ j \in [\![1; N]\!]\} \tag{4.12}$$

In the second case, if a cell was active but no cell was predicted in that cell's mini-column, we pick a winning cell that will now represent this context in the future. The winning cell is the cell that had the segment that was closest to being above the NMDA spiking threshold out of all segments in the column. In equation form, we update all segments in the following set:

$$Z^t = \{(i, j, d) \colon \widetilde{a}_i^t = 0,\ \mu^t(\mathbf{D}_{ijd}^t, p^*) = \max_{i'j'd'} \mu^t(\mathbf{D}_{i'j'd'}^t, p^*), \quad i \in [\![1; M]\!],\ j \in [\![1; N]\!]\} \tag{4.13}$$

Therefore, at step t, we update the connections of all segments in $X^t = Y^t \cup Z^t$. For all segments (i, j, d) in X_t, we go through the connections $(I, p_t) \in \mathbf{D}^t_{ijd}$ and adjust the permanence p_t by a small increment p^+ or decrement p^- depending on the state of the incident cell $a_I^{t[x]}$ as such:

$$p_{t+1} = \begin{cases} p_t + p^+ & \text{if } a_I^{t[x]} = 1 \\ p_t - p^- & \text{if } a_I^{t[x]} = 0 \end{cases} \tag{4.14}$$

The location layer has context connections emanating from cells in the location layer itself as well as from cells in the feature layer. Therefore, for each segment, the set of connected cells $\mathbf{D}^{t[l]}_{ijd}$ is initialized as a random set of cells from both the location and the feature layers. Permanence values are initialized to be a random value between 0 and 1.

LM: Feature layer

The activation and learning rules in the feature layer are exactly the same as in the location layer, except for determining the active mini-columns $\widetilde{\mathbf{A}}^t$. The inhibitory process that selects the k columns that best match the sensory input happens in the pooling step of the sensor module, the learning and activation rules of which are detailed above in the pooler section 4.1.3.

The feature layer has context connections emanating from cells in the feature layer itself as well as from cells in the location layer. Therefore, for each segment, the set of connected cells $\mathbf{D}^{t[f]}_{ijd}$ is initialized as a random set of cells from both the feature and the location layers. Permanence values are initialized to be a random value between 0 and 1.

LM: Output layer

The activation of the output layer occurs in two stages and heavily resembles the locally inhibitory process described for the pooler in section 4.1.3. First, feedforward input from the feature layer is integrated to determine which cell could become active. Then, only the specific cells that were predicted from the context connections actually become active. The context is defined by cells in the output layer itself and from the output layer of other learning modules, while feedforward connections are from cells in the feature layer.

We start by calculating the overlap for each cell i, which counts the number of active feedforward connections with concomitantly active incident cells, similar to equation 4.1:

$$\mu^t(\mathbf{F}^t_i, p^*) = |\{e \colon e = ((f, i_0, j_0), p_t) \in \mathbf{F}^t_i,\ p_t \geq p^*,\ a^{t[f]}_{i_0 j_0} = 1\}| \tag{4.15}$$

The set of output cells with enough feedforward input described by $\widetilde{a}^t_i \in \{0, 1\}$ is determined as follows:

$$\widetilde{a}^t_i = \begin{cases} 1 & \text{if } \mu^t(\mathbf{F}^t_i, p^*) \geq \theta_f \\ 0 & \text{otherwise} \end{cases} \tag{4.16}$$

where θ_f is a pre-determined threshold for the action potential. We can now calculate the activations as follows:

$$a^t_i = \begin{cases} 1 & \text{if } \widetilde{a}^t_i = 1 \text{ and } \pi^{t-1}_i = 1 \\ 0 & \text{otherwise} \end{cases} \tag{4.17}$$

To calculate the predictive state π_i^t at a time step t we start by counting the active context connections with concomitantly active incident cells for each segment d:

$$\mu^{t-1}(\mathbf{D}_{id}^t, p^*) = |\{e \colon e = ((x, i_0), p_t) \in \mathbf{D}_{id}^t,\ p_t \geq p^*,\ a_{i_0}^{t-1[x]} = 1\}| \tag{4.18}$$

The number of spiking segments on a cell i is given by:

$$\tau_i^t = \sum_d \mathbb{I}\{\mu^{t-1}(\mathbf{D}_{id}^t, p^*) \geq \theta_d\} \tag{4.19}$$

where θ_d is a pre-determined threshold for the NMDA spiking threshold. The predictive state of a cell can now be calculated as:

$$\pi_i^t = \begin{cases} 1 & \text{if } \tau_i^t \geq \xi_k^t \\ 0 & \text{otherwise} \end{cases} \tag{4.20}$$

where ξ_k^t represents the k-th highest number of active context segments. k hence represents the minimum desired number of predicted neurons in the output layer, because at least k cells will be such that $\tau_i^t \geq \xi_k^t$. If the number of cells with contextual support is less than k in the layer, ξ_k^t is zero and all cells with enough feedforward input will become active.

Learning the context connections in the output layer is regulated by equation 4.14, where the segments selected for an update can be described by the following equation:

$$X^t = \{(i, j, d) \colon \pi_i^t = 1,\ a_i^t = 1,\ \mu^{t-1}(\mathbf{D}_{id}^t, p^*) \geq \theta_d, \quad i \in [\![1; M]\!]\} \tag{4.21}$$

The output layer has randomly initialized context connections $\mathbf{D}_{id}^{t[o]}$ emanating from the output layer itself and from the output layer of other learning modules. The feedforward connections $\mathbf{F}_i^{t[o]}$ are randomly initialized to cells in the feature layer. Permanence values are initialized to be a random value between 0 and 1.

4.1.4 Algorithmic Complexity

Table 4.3: Algorithmic Complexity Analysis. Breakdown of computational costs per step for the Sensor Module (SM) and the Learning Module (LM). Activation costs are negligible hence omitted.

Layer	Overlap (Prediction)	Weight Update (Learning)	Total Step Complexity
SM: Pooler	$O(MC)$	$O(MCs)$	$MC(1+s)$
LM: Feature	$O(MNSC)$	$O(MNSCs)$	$O(MNSC(1+s))$
LM: Location	$O(MNSC)$	$O(MNSCs)$	$O(MNSC(1+s))$
LM: Output	$O(MSC)$	$O(MSCs)$	$O(MSC(1+s))$

Parameters: M: Columns/Cells; N: Cells per column; S: Segments per cell; C: Connections per segment; s: Sparsity. We have dropped the superscripts in the parameters (e.g. S instead of $S^{[f]}$) for readability.
Note: The sparsity $s \in [0; 1]$ is typically very low (e.g. 0.02) in a lot of these layers.

We breakdown the algorithmic complexity analysis summarized in the table 4.3 above inside the sections below. All costs are formulated per step.

SM: Pooler

We assume that the raw sensory input is encoded on L cells and that each cell of the pooler can make connections with up to $C^{[p]}$ cells in the input. The complexity is dominated by the calculation of the overlap (eq 4.1) which takes $O(MC^{[p]})$ operations.

If we assume sparsity of $s^{[p]} \in [0; 1]$ in the pooler's activation, the learning step (eq 4.4) takes on average $O(MC^{[p]}s^{[p]})$ operations. In practice we chose k such that the top-k cells count is around $s^{[p]}$ of the total number of cells in the layer M, i.e. $\frac{k}{M} = s^{[p]}$.

The boosting rules take $O(M)$ operations.

The complexity of the pooling step is $O(MC^{[p]}(1 + s^{[p]}))$.

LM: Feature layer

We assume that there are $S^{[f]}$ context segments per cell in the layer, and that each segment in the feature can make connections with up to $C^{[f]}$ cells in the feature and location layers.

The complexity is dominated by computing the predictive state of the cells, more specifically the overlap computation (eq 4.18), which takes $O(MNS^{[f]}C^{[f]})$ operations. That is because a traversal of the connections $\mathbf{D}^t_{ijd}$ is required. In particular, for every mini-column i, for every cell j, for every segment d, we calculate whether the segment is spiking by summing over the set of connections $\mathbf{D}^t_{ijd}$.

Computing the cell activations takes $O(MN)$.

If we assume sparsity of $s^{[f]} \in [0; 1]$ in the cell activations, the learning step (eq 4.14 on cells 4.12 and 4.13) takes $O(MNS^{[f]}C^{[f]}s^{[f]})$. If the pooling layer cells have a $s^{[p]}$ sparsity, then the sparsity of the feature layer activations is $s^{[f]} \leq s^{[p]}$.

The total complexity of the feature layer is $O(MNS^{[f]}C^{[f]}(1 + s^{[f]}))$.

LM: Location layer

The complexity in the location layer is equal to the complexity in the feature layer, up to negligible terms. The total complexity of the location layer is $O(MNS^{[l]}C^{[l]}(1 + s^{[l]}))$.

LM: Output layer

We assume that there are $S^{[o]}$ context segments per cell in the layer, and that each segment in the output layer can make connections with up to $C^{[o]}$ cells in this output layer itself and cells in the output layer of other learning modules.

As for other layers, the complexity in the output layer is dominating by the computation of the predictive state of the cells, more specifically the overlap computation (eq 4.18, which takes $O(MS^{[o]}C^{[o]})$.

Computing the cell activations takes $O(M)$.

If we assume sparsity of $s^{[o]} \in [0; 1]$ in the cell activations, the learning step (eq 4.14 on cells 4.21) takes $O(MS^{[o]}C^{[o]}s^{[o]})$. Similarly to the pooling layer, we chose k such that the top-k cells count is around $s^{[o]}$ of the total number of cells in the layer M, i.e. $\frac{k}{M} = s^{[o]}$.

The total complexity of the output layer is $O(MS^{[o]}C^{[o]}(1 + s^{[o]}))$.

4.1.5 Space Complexity

We devise table 4.4 to list the space complexity, i.e. the footprint in memory, of the relevant structures, including the cell states and the connection data.

Table 4.4: Space Complexity of HTM States and Connections

Component	Pooler (p)	Feature (f)	Location (l)	Output (o)
States				
Activation (a)	$O(M)$	$O(MN)$	$O(MN)$	$O(M)$
Prediction (π)	—	$O(MN)$	$O(MN)$	$O(M)$
Connections				
Feedforward (F)	$O(MC)$	—	—	$O(MC)$
Contextual (D)	—	$O(MNSC)$	$O(MNSC)$	$O(MSC)$

Parameters: M: Mini-columns; N: Cells per column; S: Segments per cell; C: Connections per segment. We have dropped the superscripts in the parameters (e.g. S instead of $S^{[f]}$) for readability.
Note: States typically require 1 bit per element if packed, otherwise 1 byte per. Connections require storing both the index of the incident cell and the permanence value, possibly on 32 bits if packed.

4.2 Implementation on Processing-in-Memory hardware

We implement Montyll on the UPMEM Processing-in-Memory system introduced in section 3.7. This 2d-integrated PiM design is capable of running one learning module step per core, all in parallel, with up to 2560 active cores. While this ability to do parallel computation with heterogeneous weight and input is a great benefit, PiM cores suffer from drawbacks due to the logic elements being manufactured on a DRAM processed. We will briefly talk about the ways in which Montyll fits the UPMEM PiM constraints well in section 4.2.1, before talking about the ways in which it was optimized to run smoothly on the chips in section 4.3.

4.2.1 Is Montyll a good fit for UPMEM PiM?

Most importantly, Montyll is composed of a big set of sensor and learning modules, which operate semi-independently, while the UPMEM PiM system is composed of big set of independent cores, each complete with private memory that is large enough to store the weights that are private to

each learning module. This capacity to compute the activations and learning rules of thousands of learning modules in parallel is by far the most important benefit of the UPMEM PiM system in this context. How Montyll otherwise plays into the limitations of the UPMEM PiM cores can be summarized as follows.

First, only bitwise operations and integer arithmetic are natively supported, with integer multiplication up to 8 bits by 8 bits. This is on top of the usual control and load/store operations one would expect from a RISC processor. When looking at the activation and learning rules of Montyll in section 4.1.3, we see that most operations fall in the bitwise operations and integer arithmetic buckets. This a great coincidental fit of HTM networks, lucky for us and our constrained chips. In the next section, we will see how those few operations that required high precision or floating points are handled, especially the boosting factors of form $b_i = e^{-\beta(A_i^t - \overline{A^t})}$.

Second, work within a DPU needs to be divided well between multiple tasklets (threads) to maintain a high instruction per cycle (IPC). In Montyll, most work is done in nested for loops that iterate over the set of columns, cells, segments and connection. That provides an easy parallelization factor. We need to carefully decide the way in which we parallelize in relation to how the MRAM data blocks will be fetched. We talk about it in more detail in section 4.3.5.

Third, direct memory accesses to the MRAM are not possible. The relevant MRAM data needs to be brought to WRAM, preferably in blocks to maximize WRAM-MRAM bandwidth. Thankfully, our workload exhibits straightforward data access patterns to the connections data, which means that we don't have unnecessary WRAM-MRAM data movements due to over-fetching. This relates to the last point, because transfer block sizes are related to how much data is required by each tasklet within an iteration of the loop.

4.3 Optimizing Montyll: Memory & Operations

Since the memory that UPMEM PiM cores have access to is limited, we carefully design our states and connections, which space complexities are outlined in 4.4, so that all states fit in the single-cycle latency 64 KiB scratchpad memory (WRAM) and all connections fit in the 64 MiB bank (MRAM).

Moreover, as per the hardware constraints of the available Processing-in-Memory hardware, Montyll's operations were optimized and tweaked to run on these cores without sacrificing performance. We discuss here the various design choices that were made.

4.3.1 SM: Pooler

To accommodate the lack of floating-point unit, the boosting factors are calculated using a combination of fixed-point arithmetic and hybrid lookup tables (LUTs).

First, the time-averaged are calculated iteratively with $A_i^t = A_i^{t-1} + \frac{1}{T}(a_i^t - A_i^{t-1})$. It can be seen as a map $f\colon [0;1] \times \{0;1\} \to [0;1]$ with $f(X,Y) = X + (Y - X)/T$ where $X = A_i^{t-1}, Y = a_i^t$ and $f(X,Y) = A_i^t$. If we restrict T to be a power of 2, the division by T is equivalent to a shift operation. Moreover, because of the absence of floating-point operations, we don't want a map

from $[0;1]$ to $[0;1]$, and we encode the range $[0;1]$ into a range $[\![0;B]\!]$, where $B = 2^{16} - 1$. We instead use the map g, which is what we use in practice to compute the time-averaged activation encoded on $[\![0;B]\!]$:

$$g: \begin{array}{c} [\![0;B]\!] \times \{0;1\} \to [\![0;B]\!] \\ (X,Y) \mapsto Bf(X/B,Y) = X + (YB - X)/T \end{array} \tag{4.22}$$

Second, to accommodate for the lack of floating-point unit and keep the implementation space efficient, the exponential boosting function $b_i = e^{-\beta(A_i^t - \overline{A^t})}$ is pre-computed on an 8-bit integer representation in a lookup table (LUT). To maximize the representational range within the 8-bits, we use a dynamic precision encoding scheme. Standard fixed-point arithmetic typically suffers from a trade-off between range and resolution. Our implementation dynamically changes the semantic meaning of the stored byte (in the LUT) based on the magnitude of the boosting factor.

The 16-bit time-averaged activation is quantized to 8 bits to serve as the index. The stored value dictates the operation:

- For columns which response needs to be suppressed ($b_i \leq 0.5$), the LUT stores a negative value representing a logarithmic shift magnitude ($= \log_2(b_i)$). The runtime applies this as a right-shift operation (`>> boosting_factor`), effectively dividing the response.
- Otherwise if $b_i > 0.5$, the LUT stores the direct integer multiplier. The runtime applies this using the available 8x8 bit multiplication (`* boosting_factor`).

4.3.2 LM: Location, Feature and Output layers

The most relevant changes in operations in the location, feature and output layers relate to how the states are packed in integers.

Location & Feature Layers. The activations a_i^t and predictions π_i^t are arrays of integers of size M, where the exact integer type can depend on N. In practice, the number of cells per column does not exceed 32 or 64, meaning that we can afford to encode the activity and predictions of all cells within a columns on a single integer. To retrieve to `i`-th bit in integer `x`, we use `(x >> i) & 1`. Packing the values into integers for lower state footprint does not come for free, and has a penalty in runtime. In particular, computation time is dominated by the overlap calculation, which is done in a loop over the columns, cells, segments and connections with 12 instructions per loop. Inside this loop, to determine whether the incident cell is active, we don't just perform a memory access but we have to perform a `shift` and logical `and`, adding two instructions. In practice, it only incurs an additional 1 instruction since the `and` operation can be optimized away as we are performing a logical `and` with the connection activity.

Output Layer. Since the output layer is of shape (M), we store the activations a_i^t in a packed `uint32_t` representation of length $M/32$. Moreover, we don't store predictions π_i^t but the spike counts τ_i^t given by equation 4.19 instead. That is because the actual predictions are decided by a competitive top-k process. Therefore, τ_i^t is stored in an array of `uint8_t` of size M. Storing

the activations in a packed form incurs a cost in runtime. In particular, retrieving the activation of cell `i` in array `acts` is done with `GET_BIT(acts[i >> 5], i & 0b11111)`, where the first 5 bits of `i` select for the bit and the other bits select for the word and where `GET_BIT(x, i)` is defined as `(x >> i) & 1`. The cost in runtime is larger than for the location and feature layer, as index computation incurs an additional instruction for a total of 13 per loop of the overlap calculation. This number goes to 18 if the cell activity comes from an external learning modules, as 5 more instructions are required to dynamically calculate the address of the external matrix row. Fortunately, since the algorithmic complexity of the overlap calculations is $O(MSC)$ in the output layer compared to $O(MNSC)$ if the feature/location layers, these additional instructions per loop do not come at too steep of a price in total runtime.

4.3.3 Connections & Spike Count Cache

Each cell in a layer has S segments on which C incident cells can be connected. A connection is comprised of a permanence value that can be encoded on 8 bits and of the index of the incident cell can be encoded on 24 bits, for a total of 32 bits per connection. The index can be encoded on 24 bits thanks to the use of C unions and carefully crafting packed index representations for each possible incident layer. This means that the size in bytes of the context connections in the feature/location layers is $MNSC \cdot$`sizeof(uint32_t)`, while it is $MSC \cdot$`sizeof(uint32_t)` in the output layer.

To not have to recompute the overlaps in the learning step after having computed them in the prediction step, we cache the number of spiking connections per segment in an integer (`uint8_t`) structure of shape (M, N, S). This structure is significantly smaller than the connection structure.

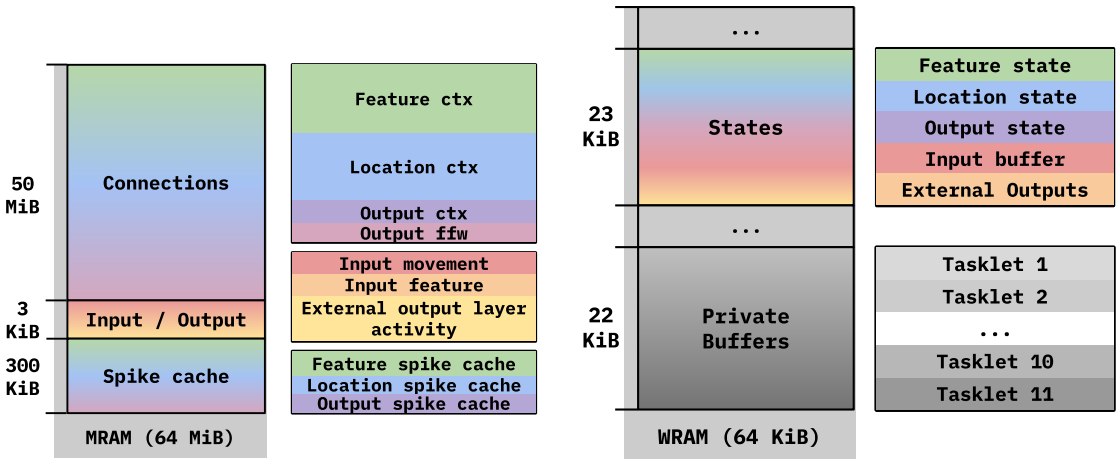
4.3.4 MRAM & WRAM content

We include an example of the WRAM and MRAM content in figure 4.2. The exact values are obtained from a network parametrized by the values in table 5.1 from the methods section 5.

4.3.5 Tasklet-level parallelism

UPMEM chips are fine-grained multi-threaded. This requires multiple threads to run on the same chip to keep the instruction per cycle figure high. This means that we need to find parallelization opportunities within each computational step of a learning module to keep the chips near the optimal performance.

Workload parallelization strategy. The bulk of workload runtime is spent on operations which happen inside `for` statements that loop over the columns and cells of the network. This represents an obvious parallelization strategy. That is because while the state of each cell at step i depends on their state at step $i-1$, they do not depend on each other's state at step i. Hence, each tasklet can independently work on a set of cells assigned to it. In practice, in the feature/location layers, work is divided among the tasklets by assigning an equal set of columns to each tasklet, while in the output layer, it is the number of cells that is divided equally between tasklets.



(a) MRAM **(b)** WRAM

Figure 4.2: DPU memory contents. Connection data lives in MRAM and is block-transfered to WRAM as needed to the private buffers. The same applies to the spike count caches. States are kept in WRAM and never flushed for constant-cycle latency access.

MRAM-WRAM block transfers. Direct access to the sparse connection matrices stored in the larger MRAM is not possible on UPMEM PiM chips. We first need to bring the relevant data to WRAM via block transfers. The same goes for the spike count cache. Following both the best practices recommendations from [37] and the limited amount of WRAM (64 kB), we chose to bring data in blocks of around 1KiB for the connections and a less ideal 50 B for the spike count cache. This corresponds to the amount of connections data needed for a single cell, and to the amount of spike count cache data needed for a whole column. That figure is obtained with equations $SC \cdot$`sizeof(segment_data)` and $S \cdot$`sizeof(uint8_t)` which can be adapted to accommodate different network parameters. There is a slight modification in the output layer, where the spike count cache is brought in in packs of 8 cells, as the data needed for a single cell instead of a whole column is too small to qualify for a WRAM-MRAM transfer (minimum 8 bytes).

4.3.6 Barriers and synchronization

Because the work is distributed across tasklets, we need to make sure that certain tasklets are not computing states that depend on states that have not finished settling and that are still being worked on by other tasklets. To this end, we introduce barriers between certain steps. For example, activations depend on predictions, hence a barrier needs to be placed between their computation. We list all barriers present in the learning module of Montyll in figure 4.3.

As a small aside, notice that, in our implementation, the predictions computed in step t are directly used for the activations within the same step t. We could have implemented Montyll such that the activations in step t use the predictions of step $t-1$, i.e. the predictions are prepared for the next step. These two implementations are equivalent and we chose the former.

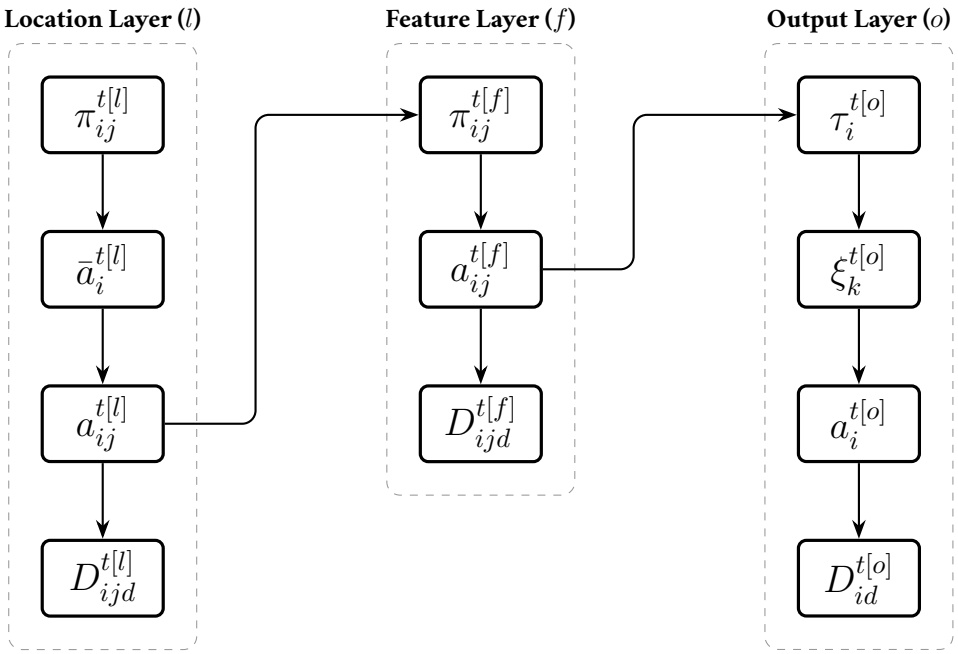


Figure 4.3: Montyll learning module dependency diagram for step t. Only the dependencies that require barriers are shown, each arrow corresponding to a barrier.

4.4 Communicating between Learning Modules

4.4.1 Pipeline parallelism

In concordance with the hierarchy in the neocortex [115][116][117], e.g. the human visual system has 4 levels of hierarchy (V1-V4), Thousand Brains Systems can be set up as a hierarchy of learning modules.

The output layer of a learning module represents an object id. Higher-level learning modules receive an object id from lower-level learning modules as their feature layer input. Furthermore, they receive skip-layer connections from the sensor modules themselves at the feature layer, as well as transformed movement vectors at the location layer. Figure 4.4 serves as an illustration.

Each processor of a PiM system runs a single learning module or a single sensor module, or a combination of both. Cores that run high-level modules receive inputs from cores that run lower-level modules. However, these higher-level cores need not sit idle while lower-level processing is done. They can be busy processing the last step. For instance, sensor modules (level 0) might be processing step i while level 1 learning modules are still processing step $i-1$, going all the way to level 4 learning modules that would still be processing step $i-4$. This way, processing is pipelined over the hierarchy of modules, as highlighted in figure 4.5.

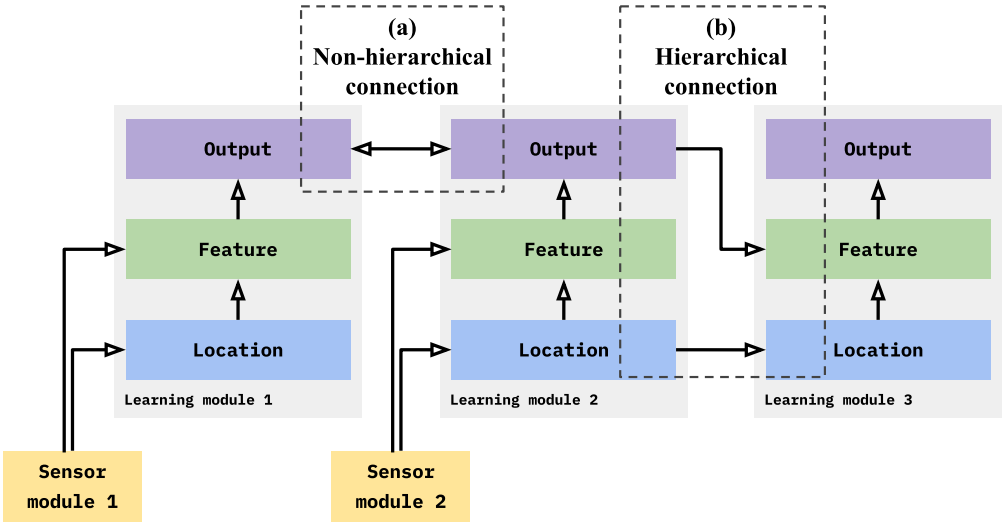


Figure 4.4: Hierarchical vs. non-hierarchical connections in Thousand Brains Systems.

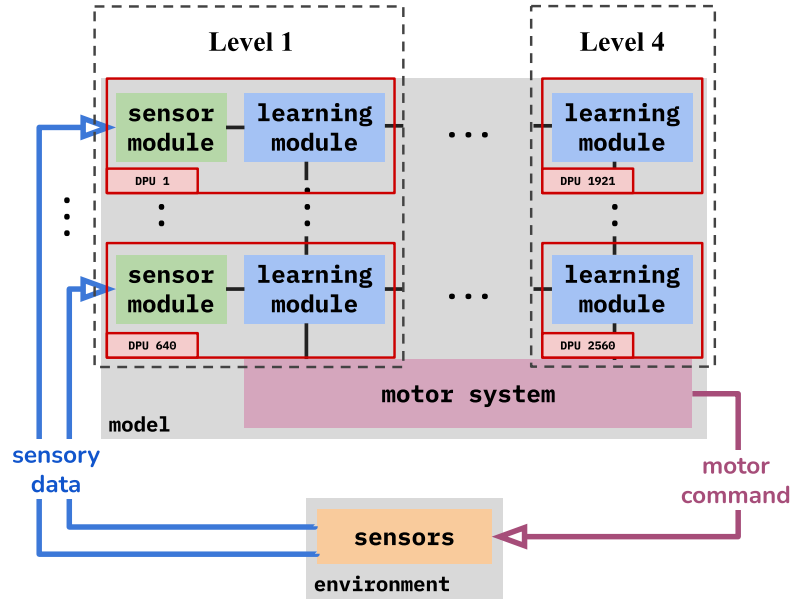


Figure 4.5: Pipeline parallelism in Thousand Brains Systems. All learning modules are mapped to a DPU. Processing is performed in as many stages as there are levels in the hierarchy.

4.4.2 Growing external state footprint

As the number of learning modules grows, the number of lateral and hierarchical connections from the output layer grows. The output layer and feature layer of learning modules get contextual signal from a growing number of learning modules' output layer, in addition to the location layer receiving feedforward input from the location layer of lower learning modules. Meanwhile, our implementation of Montyll relies on the layers' state to fit in WRAM for $O(1)$ read and write. If

the contextual states can no longer fit in WRAM (64 KiB), we have to devise new ways to deal with this data.

Inter-learning modules connection sparsity. There are limits to the combinatorial explosion since the inter-lm connections are sparse. While the exact sparsity figure is unclear, columns in the neocortex typically project very sparsely to unrelated areas [118][119], bur project less sparsely to related areas and neighboring columns [115]. The same way that columns are divided into regions in the brain (visual cortex, auditory cortex, ...), we can divide the learning modules of Thousand Brains Systems into functional regions, and enforce some sparsity within a region, and higher sparsity across regions, while still allowing connections.

State compression. The typical sparsity exhibited in the output layer is around 1-3%. By compressing the output states, we can fit orders of magnitudes more states in WRAM, although it comes at a penalty in execution time. This is because reading a specific cell in an output state in compressed form vs. uncompressed form takes respectively $O(1)$ and $O(k)$ where k is the cell's index in the state array. We leave more complex state compression mechanisms for future work, as discussed in section 8.4.

Chapter 5

Methodology

5.1 Model

Since our goal in this paper is to explore the consequence of scaling Thousand Brains Systems, we are most interested in the parameters that directly impact the algorithmic complexity, space complexity or data movements. We described these parameters in table 5.1. The other parameters that are used in the learning and activation rules are included in table 5.2 for completion.

Table 5.1: Structural Parameters. These parameters significantly impact the algorithmic and space complexity of our network.

Symbol	Parameter	Value	C Struct Field
M	Columns / Output	1024	`num_minicols`, `cols`
N	Cells per Column	8	`cells`
S	Segments	12	`*_segments`
s	Sparsity	0.02	`activation_density`
$k^{[p]}$	Min. Active	20	`top_k`
$k^{[o]}$	Min. Active	10	`min_active_cells`
$N_{ext}^{[o]}$	External Inputs	20	`external_lms`

Note: Superscript $[x]$ corresponds to a parameter in layer x. Other parameters are shared across all layers.

Our exact implementation of the learning and activation rules listed in section 4.1.3 can be found in our library `cmontyll` which we open-source at https://github.com/Xavier0301/cmontyll. We include the relevant C symbols corresponding to the mathematical symbols in both tables 5.1 and 5.2.

We also open-source `pim-montyll`, our implementation of Montyll on the UPMEM Processing-in-Memory functional simulator, at https://github.com/Xavier0301/montyll-pim. This implementation closely follows our library `cmontyll` while introducing all the necessary modifications introduced in section 4.2. In particular, we have divided the work done in a single step across multiple tasklets, have introduced inter-tasklet barriers and added all the necessary WRAM-MRAM block transfers to correctly access and update both the connection data and the

Table 5.2: Learning Parameters. These parameters regulate the internal logic (activation thresholds, learning rates) but do not significantly alter memory allocation or algorithmic complexity bounds.

Symbol	Parameter	Value	C Struct Field
Activation Thresholds			
$\theta_f^{[p]}$	Feedforward Thresh.	1	`pooler.stimulus_threshold`
$\theta_d^{[f,l]}$	Context Threshold	15	`htm.segment_spiking_threshold`
$\theta_f^{[o]}$	Feedforward Thresh.	3	`ext_htm.feedforward_activation_threshold`
$\theta_d^{[o]}$	Context Threshold	18	`ext_htm.context_activation_threshold`
Synaptic Plasticity			
p^*	Perm. Threshold	0.5	`permanence_threshold`
$p^{+[p]}$	SM Increment	0.10	`pooler.permanence_increment`)
$p^{-[p]}$	SM Decrement	0.02	`pooler.permanence_decrement`
$p^{-[o,f,l]}$	LM Increment	0.06	`htm.perm_increment`
$p^{-[o,f,l]}$	LM Decrement	0.04	`htm.perm_decrement`
p_{decay}	Decay	≈ 0.004	`htm.perm_decay`
Homeostasis			
$\beta^{[p]}$	Boosting Strength	100.0	`pooler.boosting_strength`
$T^{[p]}$	Duty Cycle Window	1024	`log2_activation_window` (2^{10})

Note: Superscript $[x, y, z]$ correspond to parameters shared across layers x ,y and z. Other parameters are shared.

spike count cache. We have also included the host-side code responsible for transferring the relevant per-step input (sensory input, movement, other lms' output layer activation) and retrieve the relevant per-step output (output layer activation).

5.2 Simulation

To run simulations of Montyll on a Processing-in-Memory system, we use the uPIMulator [120] cycle-level simulator. uPIMulator has been validated against real hardware and shown to be 98.4% correlated with the real time for the kernel execution. All the parameters used in our simulation are detailed in table 5.3.

In order to correctly and accurately transfer the input data to the DPUs, we initialize our model in C and import the data to uPIMulator (written in go) using go-C interop. We use the UPMEM functional simulator to verify our implementation. We release the relevant extensions to the uPIMulator source code in the following fork https://github.com/Xavier0301/montyll-upimulator.

We compare the runtime of Montyll on the simulated Processing-in-Memory machines against its runtime on two CPU system, all listed in table 5.4. The first CPU Low, has 12 cores, while we

Table 5.3: uPIMulator simulation configuration.

Parameter	Value
DPU Processor Architecture	
Operating frequency	400 MHz
Number of pipeline stages	14
Revolver scheduling cycles	11
WRAM / IRAM size	64 KB / 24 KB
WRAM / IRAM access latency	1 cycle
WRAM / IRAM access granularity	4 / 6 B per clock
WRAM / IRAM access bandwidth	1,400 / 2,100 MB/s
Atomic memory size	256 Bits
DRAM System	
MRAM size	64 MB
DDR specification	DDR4-2400 [121]
Memory scheduling policy	FR-FCFS
Row buffer size	1 KB
Timings ($t_{RCD}, t_{RAS}, t_{RP}, t_{CL}, t_{BL}$)	16, 39, 16, 16, 4 cycles
Software Architecture	
Number of general-purpose registers	24
Maximum number of threads	24
Stack size (per thread)	2 KB
Heap size	4 KB

use 48 cores from the CPU High machine. This is because we are using the Euler cluster at ETH Zurich as a guest, which limits the core count to 48. We use OpenMP [122][123] to program the CPU systems.

We omit the latency incurred by communications for the PiM system, assuming a low-latency interconnect. At each step, a core sends and receives about $N_{ext}M/8$ bytes of data, which is about 2.5 KB in our case, which we assume to incur a negligible increase in the time per step.

Table 5.4: Hardware Specifications. A comparison of the three computing platforms used in our methodology. CPU1 (Low) and CPU2 (High) serves as baselines with varying number of cores.

Platform	Year	Cores	Freq.	LLC	Mem. BW	Mem. Cap.
CPU1 [Low] M2 Pro 12-CPU	2023	12 (8P + 4E)	2.8/3.5† GHz	36 MB (L2)	200 GB/s	16 GB
CPU2 [High] AMD EPYC 7763	2021	48‡	2.45 GHz	256 MB (L3)	204.8 GB/s	128 GB
PiM UPMEM v1B	2021	2,560 DPUs	400 MHz	–	600 MB/s per DPU	64 MB per DPU

† 8 P Cores at 3.5 GHz, 4 E Cores at 2.8 GHz. ‡ We use 48 of the 64 cores on the machine.

Chapter 6

Results

Scaling. In practice, we scale Montyll to 2560 learning modules on CPU2 and PiM. This represents a combined 44.5 million neurons and 16.1 billion synapses, making our work the first to investigate inter-connected htm networks of this scale.

Time per step. Figure 6.1 represents the per step runtime of Montyll on the three different computing platforms: CPU1, CPU2 and PiM.

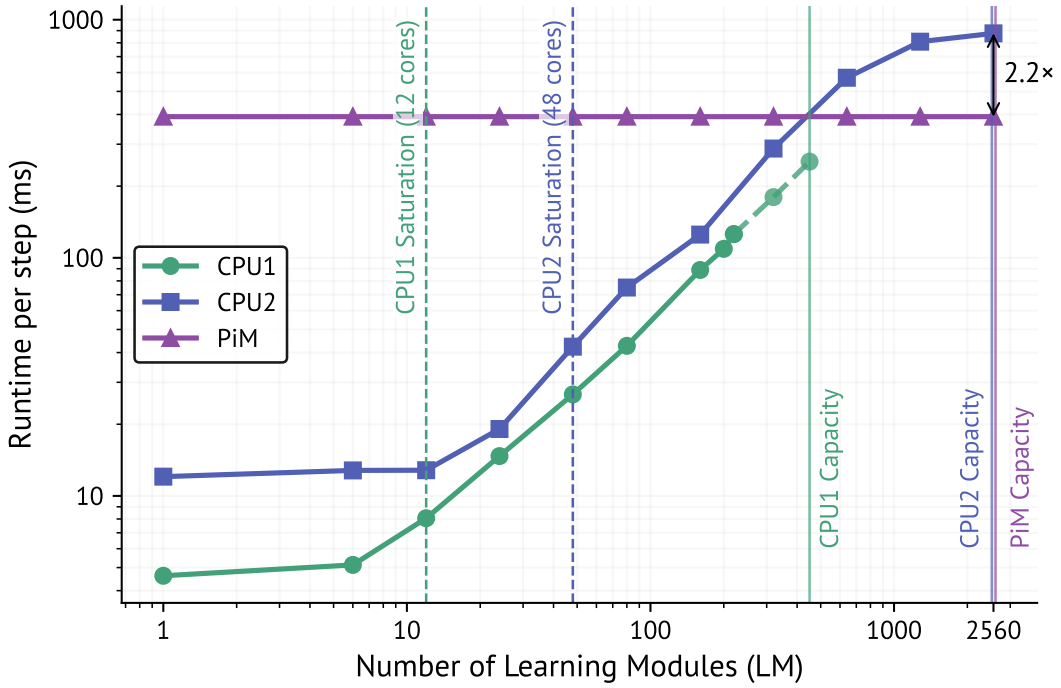


Figure 6.1: Runtime per step in ms for an increasing number of learning modules, on CPU1, CPU2 and PiM.

The most interesting comparison point is between CPU2 and PiM. We see that PiM suffers a very high runtime per step for 1 learning module of 391 ms, which is 32× that of CPU2 and 84.5× that of CPU1. This is due to the low PiM core frequency of 400 MHz, as well as the comparatively slow

memory bandwidth to MRAM of 600 MB/s. CPU1 and CPU2 both enjoy high-frequency cores, which are furthermore capable of Instruction Level Parallelism (ILP) and of prefetching the data to hide the memory latencies.

However, PiM can enable scaling to thousands of cores without degradation in time per step, unlike CPU1 and CPU2. This culminates in a 2.2× speedup over CPU2 at 2560 learning modules, for a per step runtime of 391 ms vs. 876 ms. The CPU2's increase in runtime vs. the number of learning modules is not linear. This is because it benefits from an increase in ILP when the number of threads goes from 640 to 1280 to 2560, with respectively 0.76, 1.05 and 1.87 instructions per cycle. We also see that CPU1 has a limited capacity of 16 GB, which can accommodate a maximum of 450 learning modules at a runtime per step of 125 ms.

Figure 6.2 breaks down the time spent in each layer within a step. We see that the DPUs spend 82.8% of the time in location predict and feature predict. This is unsurprising given the computational complexity of prediction vs. the rest as listed in 4.3. The complexity of learning depends on the sparsity in the network. More sparsity means less updates per step. The output layer has $N\times$ less cells than the other layers, which explains that output predict (23 ms) is 6.7× and 7.3× less expensive than location and feature predict. We also see that the activation rules for the output are more expensive than in the other layers. That is because the underlying algorithms are less amenable to parallelization over tasklets, tanking the performance of the FGMT pipeline.

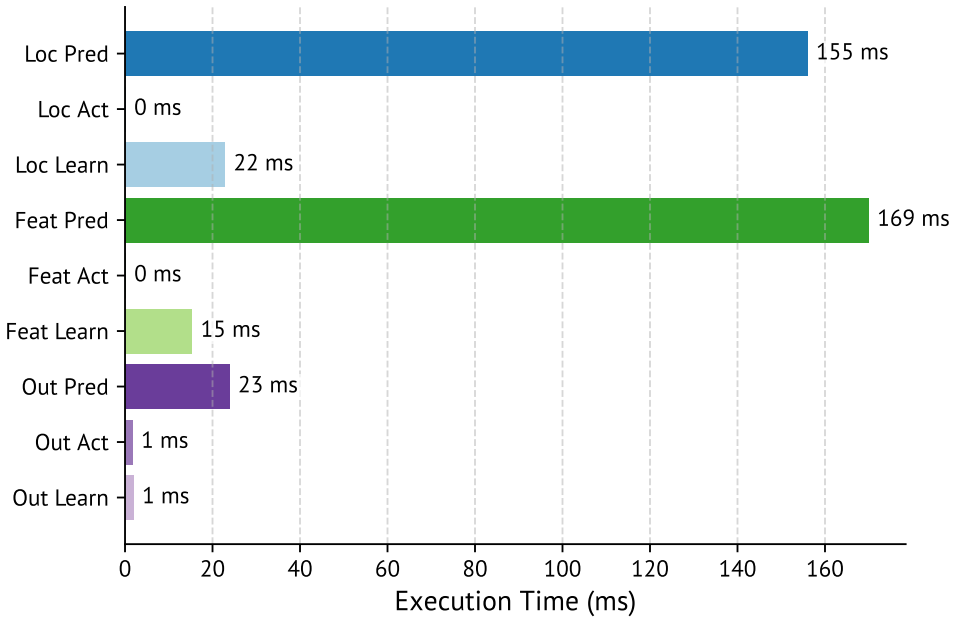


Figure 6.2: Runtime split in ms of each section of a learning module's step on the PiM system

Figure 6.3 highlights the time spent by the cores executing operations vs. waiting for memory (Memory), suffering from not enough issuable instructions (Bubble) or stalled because of register file hazard (RF). We see that the DPUs schedule 0.57 instructions per cycle on Monty, given that the

theoretical maximum is 1. We see that very little time is spent waiting for MRAM-WRAM transfers. That is because of the large MRAM-WRAM bandwidth in DPUs and because the optimal block transfer patterns that we implement in our code. We also notice that our workload is managing to fill the pipeline very well, causing very little bubbles.

The DPUs spend 36% of the time idle because of register file hazards. That is explained by the fact that the register file in an UPMEM DPU does not have enough ports to read two values at once. One port can read any even register (`r0`, `r2`, ...) and the other can read any odd register (`r1`, `r3`, ...). This causes stalls if an instruction is trying to read two even or two odd registers at once.

We see that a lot of time is spent waiting for memory in the learning stages. That is because this stage exhibits a very low operational intensity in its current implementation. In particular, if it is decided that a segment will not be reinforced, which is the case for most segments, we go onto the next segment, thus skipping a whole segment worth of connections that was brought in from MRAM without doing any operation with it. This issue would be resolved by finer-grained MRAM transfers, i.e. only loading precisely was is needed, which we leave for future work.

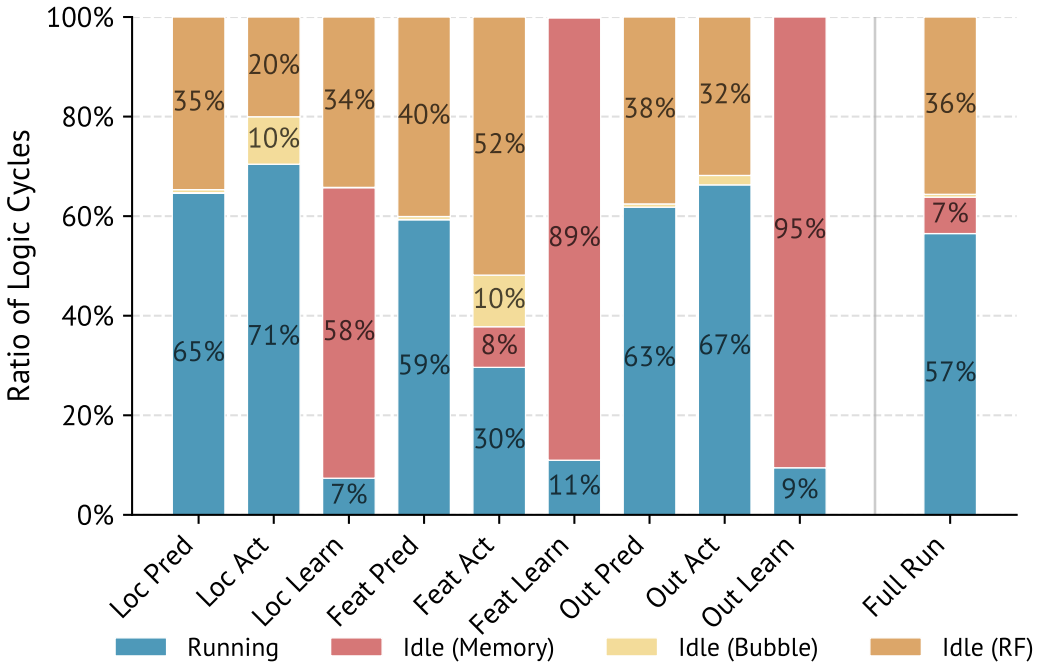


Figure 6.3: Breakdown of pipeline activity during the execution on the PiM system.

Barriers and synchronization. We run Montyll-PiM on the simulator with all barriers removed. We find that we spend 1.003% of the runtime of Montyll-PiM, or about 3.929 ms per step on waiting for other tasklets to finish, which constitutes a marginal spending.

Reset penalty. After each step, we reset the DPU. This involves (i.) loading various MRAM data like the input buffers and the external output activations into WRAM, (ii.) allocating the necessary structure on the heap, (iii.) loading the relevant model parameters on the stack, and (iv.) computing all addresses and strides used. We find that the DPUs spend 0.01% of time per step on reset, which is negligible.

Chapter 7

Related Works

This work studies Thousand Brains Systems through a systems lens, and is therefore informed by (*i*) prior work on column-based/HTM-style models and (*ii*) a broader literature on hardware and architectures for memory-bound computation. We briefly summarize the most relevant lines of work and position our work at a high level.

7.1 Thousand Brains Systems and HTM-style networks

Thousand Brains Systems build on the Thousand Brains Theory [13][15] and earlier theories on the columnar organization in the neocortex [12]. Early end-to-end results on Thousand Brains Systems are reported in [16], while the Thousand Brains Project outlines a roadmap toward larger-scale, more biologically grounded systems [17].

Montyll draws from several lines of prior HTM-related work, including active dendrite neuron models [49][50][51], HTM spatial pooling [52], and location representations inspired by grid cells [55][56].

There are efforts to unify and clarify the growing body of HTM-related mechanisms and their relationship to cortical function [124].

Our work does not aim to be a neuroscience contribution. Instead, it aims to provide a concrete systems-oriented instantiation (Montyll) that is consistent with the long-term HTM/Thousand Brains agenda and amenable to architectural study.

7.2 Hardware acceleration for Thousand Brains Systems

There is emerging work on specialized hardware for Thousand Brains Systems, largely in the form of in-logic accelerators [91][92][93]. We view these efforts as complementary: they build specific hardware components, whereas our focus is on understanding what limits scaling under an auto-regressive execution model and the architectures that best support it.

7.3 Processing-in-Memory and memory-centric computing

Most modern systems remain processor-centric: they are designed to move data to compute units. A growing body of work argues that this approach increasingly runs into fundamental bottlenecks:

many important workloads have become data-intensive and off-chip data movement is often far more expensive than executing operations in latency, bandwidth, and energy [24][64][34][35].

Processing-in-Memory (PiM) broadly captures a family of approaches that reduce or eliminate data movement by placing computation mechanisms in or near where data is stored. While the general idea is old [27][61][28][31][32][33], it has regained momentum due to both application trends and pressures to the memory system [34][125].

One of the Processing-in-Memory solutions, *processing-near-memory*, places compute units closer to memory (e.g. near-bank cores, 3D-stacking) showing performance and energy efficiency gains in general purpose processing [36][63][64][65], graph processing [62][66][67] and neural networks inference [68][64][69][70][71]. Another direction is *processing-using-memory*, which exploits DRAM/-NAND circuit properties to execute bulk operations in situ [126][127][128][129]. Across both, adoption remains a challenge that spans architecture, programming models, and system integration [72][130].

Finally, memory-centric thinking is also motivated by the growing complexity of main-memory design and scaling challenges, including reliability and security concerns such as RowHammer [131][132][133][134] and RowPress [135].

In this broader context, our work studies Processing-in-Memory as a concrete architectural substrate for computing an emerging class of AI workloads (Thousand Brains Systems) with a unique scaling profile, using a real PiM platform [37] that places RISC cores near DRAM banks.

7.4 Machine learning on specialized and near-memory hardware

The broader ML systems literature shows that mapping AI workloads to specialized hardware is often key to scaling and efficiency. GPU-centric training and inference were pivotal in the deep learning era [102][22][78][79], and dedicated accelerators such as TPUs extend this paradigm [25][80]. A recurring theme is careful management of data movement [24][81] and communication in distributed settings [136].

Processing-near-memory and accelerator-in-memory designs have also been proposed for neural workloads [68][69][39][40][41][70], and more recently for large language model inference in GPU-free systems [71].

This work distances itself from the GPU paradigm that regulates modern AI systems, as it investigates the scaling consequences of many learning modules under an auto-regressive loop, but it fits into the broader theme of algorithm-hardware co-design for AI. It can best be compared to the scaling challenges faced in decoder-only transformers inference due to the presence of an auto-regressive loop [71].

Chapter 8

Discussion

8.1 Closing the Gap: Real-Time Processing Speed

While biological intelligence is not regulated by a global clock, the brain does show signs of processing the world discretely [137], regulated by alpha waves at 8-12 Hz. Studies on eye movements show even slower frequency of perception at around 2-3 Hz [138], and studies on decision making also find a psychological refractory period corresponding to 2-3 Hz [139]. More pertinent to us is perhaps the fact that the human visual system is capable of recognizing objects in 150-250 ms [140][141][142]. The ”steps” in Monty and Montyll do not have a biological equivalent, because many steps can happen before an object can be recognized. It is the time it takes to ”recognize”, which takes multiple steps and be viewed as a ”super-step”, that can then more directly be compared to the recognition times studied in humans.

We’ve shown that our PiM system is capable of computing a single step for 2560 learning modules in 391.5 ms, corresponding to 2.55 Hz, but needs multiple steps to recognize, creating a gap in performance between our system and biological learning systems. Beyond closing the gap in frequency, there will always be a motivation to process steps faster.

8.1.1 Building logic with DRAM processes

There are obvious candidates for faster processing speeds: increasing the PiM core frequency, introducing instruction-level parallelism (ILP) in PiM cores and increasing the number of cores per bank to more than 1. These are all solution borrowed from logic-based core designs.

However, the design space of logic elements built with DRAM manufacturing processes is largely unexplored. Many breakthroughs have yet to happen, and many interesting ideas are still waiting to be discovered. For example, there might be breakthroughs at the device level, in how to use the analog computing capabilities of DRAM cells [126][127][143], in using the subarray structure of DRAM to increase PiM parallelism [144][145] and I am sure many other ideas which we have yet to see.

This is especially true if our goal is not to design commodity memory devices that are DDRx compliant [146] hence highly constrained, but rather to design Processing-in-Memory systems that can handle the internal capacity, compute and bandwidth requirements of large scale Thousand Brains Systems, or any artificial intelligence system that is built after the columnar organization of the neocortex.

8.1.2 Accelerator-like design

While there is an important need for programmability to support a large design space exploration, special domain specific instructions could be introduced. For instance, the loops that dominate the algorithmic complexity and the runtime have from 12 to 18 instructions executed per loop. Many of these instructions are about shifting addresses to access data correctly, and then shifting the data itself to access the right bits, something that could easily be done within the hardware itself. This is a great opportunity to reduce this number of instructions down to 4-6, increasing throughput by 2-4x, which we leave for future work.

8.2 Closing the Gap: Capacity

The theoretical amount of data needed to represent neocortical connections (section 3.3.1) is 7.5 TB for a cat-scale system, and 600 TB for a human-scale system. While these capacities are attainable on clusters, there is a gap between the capacities of our most powerful single-node machines and the theoretical amount of data needed. Moreover, there are reasons to believe that DRAM-based main memory will never scale to the sizes required to support human-scale Thousand Brains Systems on a single-node machine, because of various challenges as highlighted in [132][125][147].

There is no good existing compromise between the internal memory & compute parallelism offered by near-bank Processing-in-Memory and the dense capacity offered by 3d NAND flash storage. Column-based AI could massively benefit from a memory device (e.g. STT-RAM, PCM, ReRAM) denser than DRAM with nevertheless similar internal bandwidth and compute parallelism to conserve a low time per step. It might be achievable, especially if the resulting memory system is architected for the sole purpose of enabling real-time learning in columnar AI systems, instead of the usual external constraints of commodity memory (e.g. [146][148]). Other related promising technologies like carbon nanotubes might also fit the bill [149][150]. This is especially true considering that the noise tolerance of HTM networks [49] might play well into the device limitations of these emerging technologies.

8.3 Algorithm design

PiM constraints. PiM hardware imposes constraints on the algorithms that it can run. Building logic using memory processes leads to cores that are less performant and can only run a reduced ISA. For instance, UPMEM PiM chips can only run integer arithmetic, and only up to 8 bit x 8 bit integer multiplication, thus largely forgoing floating point operations. As the technology improves, constraints could gradually soften. Until then, Thousand Brains Systems researchers that want to take advantage of the scalability that PiM offers will have to work with these constraints, which would therefore shape the space of possible algorithm designs.

An analogy with GPU constraints. This is similar as to what algorithms "win" in the space of deep learning. The winning ideas are largely the ones that take full advantage of the scalability that GPUs offer, which comes with its own set of constraints. To utilize GPUs to the fullest, deep

learning algorithms need to exhibit high arithmetic intensity i.e. aggressively reuse weights across inputs. This disincentivizes all algorithms that do not fall neatly in this paradigm.

8.4 State compression strategies

Following sparsity in the neocortex representations [49][53], the state of neurons in the feature, location and output layer of a learning module of our implementation is sparse. Representing the state as an array of 1s and 0s leads to a missed opportunity to compress data. Representing the state as an array of indices leads to a significant performance hit as access to a specific element is not constant time but requires a search through an array. An interesting compromize might be to use bloom filters [151] or similar data structures [152][153][154]. That way, we both significantly lower the footprint of the states while maintaining acceptable constant time for random access patterns.

8.5 Closing the Gap: Learning Module Footprint

In this work, each learning module was mapped to a DPU. However, given the nature of learning modules, which are composed of multiple layers (feature, location, output in Montyll), each layer could be mapped to a different DPU. Furthermore, this enables the addition of more layers, i.e. those left for future work (rotation i.e. layer 6b, motor command i.e. layer 5), without incurring runtime per step cost, only capacity loss. This would be a challenge for the PiM design, since the layers need to communicate their states because of inter-layer dependencies. This would heighten the incentive of building an interconnect, at least within a rank (8 chips) to communicate the states within a step. Interconnects on near-bank PiM have been successfully implemented by the industry [40].

This strategy of mapping a learning module to multiple chips also closes the gap between the footprint of a learning module (50 MB in Montyll) and the theoretical footprint of a cortical column (3 GB). By putting multiple chips together, each having a 64 MB bank, we can increase the maximum model footprint. This strategy can be extended to dispatch layers within an LM to different DPUs, since there are more than enough columns and cells in each layer to feed the pipelines of multiple DPUs.

Chapter 9

Conclusion

This work investigated the architectural suitability of various computing platforms for large-scale *Thousand Brains Systems*. We highlighted that the columnar structure of Thousand Brains Systems imply unique requirements in memory capacity, processing speed and data movement volumes. We showed that this execution model aligns well with *Processing-in-Memory* (PiM), where compute is distributed close to memory with access to sizeable internal bandwidths and high aggregated capacity while maintaining acceptable processing speeds.

Concretely, we proposed *Montyll*, a novel Thousand Brains Systems implementation which integrates elements of low-level cortical processing and unifies previous works. We provided a unified mathematical formulation, as well as space and complexity analyses of the rules that regulate the activation and learning in Montyll. We provided a high-performance implementation in C, as well as an implementation on Processing-in-Memory hardware using a cycle-level simulator based on a real-world PiM architecture, all of which we open-sourced.

We scaled Montyll to 2560 learning modules on a CPU system and a PiM system. This represents a combined 44.5 million neurons and 16.1 billion synapses, making our work the first to investigate inter-connected htm networks of this scale.

We demonstrated that PiM can scale Montyll to 2560 learning modules and achieve up to a 2.2× speedup over a high-end CPU baseline. We further analyzed why traditional CPU- and GPU-centric designs struggle to offer the same scaling behavior under the memory-traffic and weight-heterogeneity constraints inherent to Thousand Brains Systems.

Overall, *a Thousand Brains on a Thousand Chips* provides a promising direction for bringing large-scale, continually learning, sensorimotor models closer to real-time operation.

Future work includes improving PiM core throughput, internal bandwidth and overall design to close the real-time gap, exploring denser memory technologies to close the capacity gap, and co-designing Thousand Brains Systems that explicitly embrace the constraints of the underlying computing architectures and devices.

Bibliography

[1] B. M. Lake, R. Salakhutdinov, and J. B. Tenenbaum, "Human-level concept learning through probabilistic program induction," *Science*, vol. 350, no. 6266, pp. 1332–1338, 2015.

[2] B. M. Lake, T. D. Ullman, J. B. Tenenbaum, and S. J. Gershman, "Building machines that learn and think like people," *Behavioral and brain sciences*, vol. 40, e253, 2017.

[3] M. McCloskey and N. J. Cohen, "Catastrophic interference in connectionist networks: The sequential learning problem," in *Psychology of learning and motivation*, vol. 24, Elsevier, 1989, pp. 109–165.

[4] T. Flesch, J. Balaguer, R. Dekker, H. Nili, and C. Summerfield, "Comparing continual task learning in minds and machines," *Proceedings of the National Academy of Sciences*, vol. 115, no. 44, E10313–E10322, 2018.

[5] G. I. Parisi, R. Kemker, J. L. Part, C. Kanan, and S. Wermter, "Continual lifelong learning with neural networks: A review," *Neural networks*, vol. 113, pp. 54–71, 2019.

[6] T. Lesort, V. Lomonaco, A. Stoian, D. Maltoni, D. Filliat, and N. Díaz-Rodríguez, "Continual learning for robotics: Definition, framework, learning strategies, opportunities and challenges," *Information fusion*, vol. 58, pp. 52–68, 2020.

[7] N. Sünderhauf et al., "The limits and potentials of deep learning for robotics," *The International journal of robotics research*, vol. 37, no. 4-5, pp. 405–420, 2018.

[8] M. E. Raichle and D. A. Gusnard, "Appraising the brain's energy budget," *Proceedings of the National Academy of Sciences*, vol. 99, no. 16, pp. 10 237–10 239, 2002.

[9] N. C. Thompson, K. Greenewald, K. Lee, G. F. Manso, et al., "The computational limits of deep learning," *arXiv preprint arXiv:2007.05558*, vol. 10, p. 2, 2020.

[10] J. Sevilla, L. Heim, A. Ho, T. Besiroglu, M. Hobbhahn, and P. Villalobos, "Compute trends across three eras of machine learning," in *2022 international joint conference on neural networks (IJCNN)*, IEEE, 2022, pp. 1–8.

[11] G. M. Edelman and V. B. Mountcastle, *The mindful brain: Cortical organization and the group-selective theory of higher brain function*. MIT press, 1982.

[12] V. B. Mountcastle, "The columnar organization of the neocortex.," *Brain: a journal of neurology*, vol. 120, no. 4, pp. 701–722, 1997.

[13] J. Hawkins, M. Lewis, M. Klukas, S. Purdy, and S. Ahmad, "A framework for intelligence and cortical function based on grid cells in the neocortex," *Frontiers in neural circuits*, vol. 12, p. 121, 2019.

[14] J. Hawkins, *A thousand brains: A new theory of intelligence*. Basic Books, 2021.

[15] J. Hawkins, N. Leadholm, and V. Clay, "Hierarchy or heterarchy? a theory of long-range connections for the sensorimotor brain," *arXiv preprint arXiv:2507.05888*, 2025.

[16] N. Leadholm, V. Clay, S. Knudstrup, H. Lee, and J. Hawkins, "Thousand-brains systems: Sensorimotor intelligence for rapid, robust learning and inference," *arXiv preprint arXiv:2507.04494*, 2025.

[17] V. Clay, N. Leadholm, and J. Hawkins, "The thousand brains project: A new paradigm for sensorimotor intelligence," *arXiv preprint arXiv:2412.18354*, 2024.

[18] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[19] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[20] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85–117, 2015. DOI: 10.1016/j.neunet.2014.09.003.

[21] Y. Bengio, "Learning deep architectures for ai," *Foundations and Trends in Machine Learning*, vol. 2, no. 1, pp. 1–127, 2009. DOI: 10.1561/2200000006.

[22] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[23] P. Goyal et al., "Accurate, large minibatch sgd: Training imagenet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.

[24] S. W. Williams, A. Waterman, and D. A. Patterson, "Roofline: An insightful visual performance model for floating-point programs and multicore architectures," Technical Report UCB/EECS-2008-134, EECS Department, University of …, Tech. Rep., 2008.

[25] N. P. Jouppi et al., "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[26] H.-T. Kung, *Why systolic architecture?* Design Research Center, Carnegie-Mellon University Pittsburgh, PA, USA, 1982.

[27] B. A. Crane and J. Githens, "Bulk processing in distributed logic memory," *IEEE Transactions on Electronic Computers*, no. 2, pp. 186–196, 1965.

[28] H. S. Stone, "A logic-in-memory computer," *IEEE Transactions on Computers*, vol. 100, no. 1, pp. 73–78, 1970.

[29] P. M. Kogge, "Execube-a new architecture for scaleable mpps," in *1994 International Conference on Parallel Processing Vol. 1*, IEEE, vol. 1, 1994, pp. 77–84.

[30] D. G. Elliott, "Computational ram, a memory-simd hybrid," Ph.D. dissertation, 1998.

[31] D. Patterson et al., "A case for intelligent ram," *Micro, IEEE*, vol. 17, pp. 34–44, Apr. 1997. DOI: 10.1109/40.592312.

[32] D. G. Elliott, M. Stumm, W. M. Snelgrove, C. Cojocaru, and R. McKenzie, "Computational ram: Implementing processors in memory," *IEEE Design & Test of Computers*, vol. 16, no. 1, pp. 32–41, 1999.

[33] J. Draper et al., "The architecture of the diva processing-in-memory chip," in *Proceedings of the 16th international conference on Supercomputing*, 2002, pp. 14–25.

[34] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "A modern primer on processing in memory," in *Emerging computing: from devices to systems: looking beyond Moore and Von Neumann*, Springer, 2022, pp. 171–243.

[35] O. Mutlu, A. Olgun, and İ. E. Yüksel, "Memory-centric computing: Solving computing's memory problem," in *2025 IEEE International Memory Workshop (IMW)*, IEEE, 2025, pp. 1–4.

[36] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3S, pp. 336–348, 2015.

[37] J. Gómez-Luna, I. El Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, "Benchmarking a new paradigm: Experimental analysis and characterization of a real processing-in-memory system," *IEEE Access*, vol. 10, pp. 52 565–52 608, 2022.

[38] F. Devaux, "The true processing in memory accelerator," in *2019 IEEE Hot Chips 31 Symposium (HCS)*, IEEE Computer Society, 2019, pp. 1–24.

[39] Y.-C. Kwon et al., "25.4 a 20nm 6gb function-in-memory dram, based on hbm2 with a 1.2 tflops programmable computing unit using bank-level parallelism, for machine learning applications," in *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, IEEE, vol. 64, 2021, pp. 350–352.

[40] D. Kwon et al., "A 1ynm 1.25 v 8gb 16gb/s/pin gddr6-based accelerator-in-memory supporting 1tflops mac operation and various activation functions for deep learning application," *IEEE Journal of Solid-State Circuits*, vol. 58, no. 1, pp. 291–302, 2022.

[41] D. Niu et al., "184qps/w 64mb/mm 2 3d logic-to-dram hybrid bonding with process-near-memory engine for recommendation system," in *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, IEEE, vol. 65, 2022, pp. 1–3.

[42] J. A. Marshall and A. B. Barron, "Are transformers truly foundational for robotics?" *npj Robotics*, vol. 3, no. 1, p. 9, 2025.

[43] A. Billard et al., "A roadmap for ai in robotics," *Nature Machine Intelligence*, pp. 1–7, 2025.

[44] P. Gavrikov et al., "Can we talk models into seeing the world differently?" *arXiv preprint arXiv:2403.09193*, 2024.

[45] R. Geirhos et al., "Partial success in closing the gap between human and machine vision," *Advances in Neural Information Processing Systems*, vol. 34, pp. 23 885–23 899, 2021.

[46] S. Motamed, L. Culp, K. Swersky, P. Jaini, and R. Geirhos, "Do generative video models learn physical principles from watching videos?" *arXiv e-prints*, arXiv–2501, 2025.

[47] M. Schneider, R. Krug, N. Vaskevicius, L. Palmieri, and J. Boedecker, "The surprising ineffectiveness of pre-trained visual representations for model-based reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 37, pp. 32 916–32 946, 2024.

[48] C. Szegedy et al., "Intriguing properties of neural networks," *arXiv preprint arXiv:1312.6199*, 2013.

[49] J. Hawkins and S. Ahmad, "Why neurons have thousands of synapses, a theory of sequence memory in neocortex," *Frontiers in neural circuits*, vol. 10, p. 23, 2016.

[50] Y. Cui, S. Ahmad, and J. Hawkins, "Continuous online sequence learning with an unsupervised neural network model," *Neural computation*, vol. 28, no. 11, pp. 2474–2504, 2016.

[51] S. Ahmad, A. Lavin, S. Purdy, and Z. Agha, "Unsupervised real-time anomaly detection for streaming data," *Neurocomputing*, vol. 262, pp. 134–147, 2017.

[52] Y. Cui, S. Ahmad, and J. Hawkins, "The htm spatial pooler—a neocortical algorithm for online sparse distributed coding," *Frontiers in computational neuroscience*, vol. 11, p. 111, 2017.

[53] S. Ahmad and J. Hawkins, "How do neurons operate on sparse distributed representations? a mathematical theory of sparsity, neurons and active dendrites," *arXiv preprint arXiv:1601.00720*, 2016.

[54] S. Ahmad and L. Scheinkman, "How can we be so dense? the robustness of highly sparse representations," in *ICML Workshop on Uncertainty & Robustness in Deep Learning*, 2019.

[55] M. Lewis, S. Purdy, S. Ahmad, and J. Hawkins, "Locations in the neocortex: A theory of sensorimotor object recognition using cortical grid cells," *Frontiers in neural circuits*, vol. 13, p. 22, 2019.

[56] N. Leadholm, M. Lewis, and S. Ahmad, "Grid cell path integration for movement-based visual object recognition," *arXiv preprint arXiv:2102.09076*, 2021.

[57] T. Hafting, M. Fyhn, S. Molden, M.-B. Moser, and E. I. Moser, "Microstructure of a spatial map in the entorhinal cortex," *Nature*, vol. 436, no. 7052, pp. 801–806, 2005.

[58] A. O. Constantinescu, J. X. O'Reilly, and T. E. Behrens, "Organizing conceptual knowledge in humans with a gridlike code," *Science*, vol. 352, no. 6292, pp. 1464–1468, 2016.

[59] J. B. Julian, A. T. Keinath, G. Frazzetta, and R. A. Epstein, "Human entorhinal cortex represents visual space using a boundary-anchored grid," *Nature neuroscience*, vol. 21, no. 2, pp. 191–194, 2018.

[60] T. E. Behrens et al., "What is a cognitive map? organizing knowledge for flexible behavior," *Neuron*, vol. 100, no. 2, pp. 490–509, 2018.

[61] W. H. Kautz, "Cellular logic-in-memory arrays," *IEEE Transactions on Computers*, vol. 100, no. 8, pp. 719–727, 1969.

[62] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd annual international symposium on computer architecture*, 2015, pp. 105–117.

[63] A. Boroumand et al., "Lazypim: An efficient cache coherence mechanism for processing-in-memory," *IEEE Computer Architecture Letters*, vol. 16, no. 1, pp. 46–50, 2016.

[64] A. Boroumand et al., "Google workloads for consumer devices: Mitigating data movement bottlenecks," in *Proceedings of the twenty-third international conference on architectural support for programming languages and operating systems*, 2018, pp. 316–331.

[65] K. Hsieh et al., "Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation," in *2016 IEEE 34th International Conference on Computer Design (ICCD)*, IEEE, 2016, pp. 25–32.

[66] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "Graphpim: Enabling instruction-level pim offloading in graph computing frameworks," in *2017 IEEE International symposium on high performance computer architecture (HPCA)*, IEEE, 2017, pp. 457–468.

[67] M. Besta et al., "Sisa: Set-centric instruction set architecture for graph mining on processing-in-memory systems," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 282–297.

[68] E. Azarkhish, D. Rossi, I. Loi, and L. Benini, "Neurostream: Scalable and energy efficient deep learning with smart memory cubes," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 2, pp. 420–434, 2017.

[69] M. He et al., "Newton: A dram-maker's accelerator-in-memory (aim) architecture for machine learning," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2020, pp. 372–385.

[70] G. F. Oliveira, J. Gómez-Luna, S. Ghose, A. Boroumand, and O. Mutlu, "Accelerating neural network inference with processing-in-dram: From the edge to the cloud," *IEEE Micro*, vol. 42, no. 6, pp. 25–38, 2022.

[71] Y. Gu et al., "Pim is all you need: A cxl-enabled gpu-free system for large language model inference," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2025, pp. 862–881.

[72] S. Ghose, K. Hsieh, A. Boroumand, R. Ausavarungnirun, and O. Mutlu, "Enabling the adoption of processing-in-memory: Challenges, mechanisms, future research directions," *arXiv preprint arXiv:1802.00320*, 2018.

[73] V. B. Mountcastle, "Modality and topographic properties of single neurons of cat's somatic sensory cortex," *Journal of neurophysiology*, vol. 20, no. 4, pp. 408–434, 1957.

[74] L. Espeholt et al., "Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures," in *International conference on machine learning*, PMLR, 2018, pp. 1407–1416.

[75] C. Berner et al., "Dota 2 with large scale deep reinforcement learning," *arXiv preprint arXiv:1912.06680*, 2019.

[76] O. Vinyals et al., "Grandmaster level in starcraft ii using multi-agent reinforcement learning," *nature*, vol. 575, no. 7782, pp. 350–354, 2019.

[77] W. Zhao, J. P. Queralta, and T. Westerlund, "Sim-to-real transfer in deep reinforcement learning for robotics: A survey," in *2020 IEEE symposium series on computational intelligence (SSCI)*, IEEE, 2020, pp. 737–744.

[78] D. B. Kirk and W. H. Wen-Mei, *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[79] A. Vaswani et al., "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[80] N. Jouppi et al., "Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings," in *Proceedings of the 50th annual international symposium on computer architecture*, 2023, pp. 1–14.

[81] A. Ivanov, N. Dryden, T. Ben-Nun, S. Li, and T. Hoefler, "Data movement is all you need: A case study on optimizing transformers," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 711–732, 2021.

[82] S. Brown and J. Snoeyink, "Gpu nearest neighbor searches using a minimal kd-tree," *University of North Carolina, can be found at web address: http://cs. unc. edu/ ̃shawndb*, 2013.

[83] B. Pakkenberg and H. J. G. Gundersen, "Neocortical neuron number in humans: Effect of sex and age," *Journal of comparative neurology*, vol. 384, no. 2, pp. 312–320, 1997.

[84] Y. Tang, J. R. Nyengaard, D. M. De Groot, and H. J. G. Gundersen, "Total regional and global number of synapses in the human brain neocortex," *Synapse*, vol. 41, no. 3, pp. 258–273, 2001.

[85] D. Jardim-Messeder et al., "Dogs have the most neurons, though not the largest brain: Trade-off between body mass and number of neurons in the cerebral cortex of large carnivoran species," *Frontiers in neuroanatomy*, vol. 11, p. 296 229, 2017.

[86] NVIDIA Corporation, *Nvidia jetson nano system-on-module data sheet*, Version 1.0, Mar. 2019. [Online]. Available: https://developer.nvidia.com/embedded/jetson-nano.

[87] NVIDIA Corporation, *Nvidia jetson orin*, In the entry Jetson Orin Nano 8GB of the specification sheet, Dec. 2024. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/.

[88] NVIDIA Corporation, *Nvidia jetson orin*, In the entry Jetson AGX Orin 64GB of the specification sheet, Oct. 2023. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/.

[89] NVIDIA Corporation, *Nvidia jetson thor*, Aug. 2025. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-thor/.

[90] Advanced Micro Devices, Inc., *AMD Ryzen Threadripper PRO 7995WX Drivers & Support*, Accessed: 2026-01-02, 2023. [Online]. Available: https://www.amd.com/en/support/downloads/drivers.html/processors/ryzen-threadripper-pro/amd-ryzen-threadripper-pro-7000-wx-series/amd-ryzen-threadripper-pro-7995wx.html.

[91] H. Nair, D. Barajas-Jasso, Q. Jacobson, and J. P. Shen, "Tnn-cim: An in-sram cmos implementation of tnn-based synaptic arrays with stdp learning," in *2024 IEEE 6th International Conference on AI Circuits and Systems (AICAS)*, IEEE, 2024, pp. 189–193.

[92] H. Nair, W. Leyman, A. Sampath, Q. Jacobson, and J. P. Shen, "Nertcam: Cam-based cmos implementation of reference frames for neuromorphic processors," in *2024 Neuro Inspired Computational Elements Conference (NICE)*, IEEE, 2024, pp. 1–9.

[93] D. Lister, P. Vellaisamy, J. P. Shen, and D. Wu, "Catwalk: Unary top-k for efficient ramp-no-leak neuron design for temporal neural networks," in *2025 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, IEEE, vol. 1, 2025, pp. 1–6.

[94] W. Zhang et al., "Neuro-inspired computing chips," *Nature electronics*, vol. 3, no. 7, pp. 371–382, 2020.

[95] S. Billaudelle and S. Ahmad, "Porting htm models to the heidelberg neuromorphic computing platform," *arXiv preprint arXiv:1505.02142*, 2015.

[96] M. Megıas, Z. Emri, T. Freund, and A. Gulyás, "Total number and distribution of inhibitory and excitatory synapses on hippocampal ca1 pyramidal cells," *Neuroscience*, vol. 102, no. 3, pp. 527–540, 2001.

[97] S. D. Antic, W.-L. Zhou, A. R. Moore, S. M. Short, and K. D. Ikonomu, "The decade of the dendritic nmda spike," *Journal of neuroscience research*, vol. 88, no. 14, pp. 2991–3001, 2010.

[98] P. Poirazi, T. Brannon, and B. W. Mel, "Pyramidal neuron as two-layer neural network," *Neuron*, vol. 37, no. 6, pp. 989–999, 2003.

[99] C. Pehle et al., "The brainscales-2 accelerated neuromorphic system with hybrid plasticity," *Frontiers in Neuroscience*, vol. 16, p. 795 876, 2022.

[100] D. B. Chklovskii, B. Mel, and K. Svoboda, "Cortical rewiring and information storage," *Nature*, vol. 431, no. 7010, pp. 782–788, 2004.

[101] A. Sebastian, M. Le Gallo, R. Khaddam-Aljameh, and E. Eleftheriou, "Memory devices and applications for in-memory computing," *Nature nanotechnology*, vol. 15, no. 7, pp. 529–544, 2020.

[102] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors," in *Proceedings of the 26th annual international conference on machine learning*, 2009, pp. 873–880.

[103] J. Dean et al., "Large scale distributed deep networks," *Advances in neural information processing systems*, vol. 25, 2012.

[104] J. Cheng, Z. Guan, J. Fu, H. Xu, and J. Ke, "Advanced integration-inspired process-in-memory: A comprehensive review of design, challenges, and future prospects," *IEEE Access*, 2024.

[105] B. A. Olshausen and D. J. Field, "Sparse coding of sensory inputs," *Current opinion in neurobiology*, vol. 14, no. 4, pp. 481–487, 2004.

[106] S. Ahmad and J. Hawkins, "Properties of sparse distributed representations and their application to hierarchical temporal memory," *arXiv preprint arXiv:1503.07469*, 2015.

[107] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[108] W. Maass, "Networks of spiking neurons: The third generation of neural network models," *Neural networks*, vol. 10, no. 9, pp. 1659–1671, 1997.

[109] G. Major, M. E. Larkum, and J. Schiller, "Active properties of neocortical pyramidal neuron dendrites," *Annual review of neuroscience*, vol. 36, no. 1, pp. 1–24, 2013.

[110] Thousand Brains Project, *2025/09 - HTM sequence memory and time and grid cells in the neocortex*, https://www.youtube.com/watch?v=6NSDjcR3nOk, Video file. Timestamp: 23:37, Nov. 2025.

[111] J. Hawkins, S. Ahmad, and Y. Cui, "A theory of how columns in the neocortex enable learning the structure of the world," *Frontiers in neural circuits*, vol. 11, p. 295 079, 2017.

[112] G. W. Davis, "Homeostatic control of neural activity: From phenomenology to molecular design," *Annu. Rev. Neurosci.*, vol. 29, no. 1, pp. 307–323, 2006.

[113] C. Clopath, L. Büsing, E. Vasilaki, and W. Gerstner, "Connectivity reflects coding: A model of voltage-based stdp with homeostasis," *Nature neuroscience*, vol. 13, no. 3, pp. 344–352, 2010.

[114] S. Habenschuss, H. Puhr, and W. Maass, "Emergence of optimal decoding of population codes through stdp," *Neural computation*, vol. 25, no. 6, pp. 1371–1407, 2013.

[115] D. J. Felleman and D. C. Van Essen, "Distributed hierarchical processing in the primate cerebral cortex.," *Cerebral cortex (New York, NY: 1991)*, vol. 1, no. 1, pp. 1–47, 1991.

[116] M. Riesenhuber and T. Poggio, "Hierarchical models of object recognition in cortex," *Nature neuroscience*, vol. 2, no. 11, pp. 1019–1025, 1999.

[117] N. T. Markov et al., "Anatomy of hierarchy: Feedforward and feedback pathways in macaque visual cortex," *Journal of comparative neurology*, vol. 522, no. 1, pp. 225–259, 2014.

[118] E. Bullmore and O. Sporns, "The economy of brain network organization," *Nature reviews neuroscience*, vol. 13, no. 5, pp. 336–349, 2012.

[119] N. T. Markov, M. Ercsey-Ravasz, D. C. Van Essen, K. Knoblauch, Z. Toroczkai, and H. Kennedy, "Cortical high-density counterstream architectures," *Science*, vol. 342, no. 6158, p. 1 238 406, 2013.

[120] B. Hyun, T. Kim, D. Lee, and M. Rhu, "Pathfinding future pim architectures by demystifying a commercial pim technology," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, 2024, pp. 263–279.

[121] Samsung Electronics, *8Gb C-die DDR4 SDRAM x16*, K4A8G165WC, Rev. 1.5, Samsung Semiconductor, Apr. 2017.

[122] L. Dagum and R. Menon, "Openmp: An industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[123] R. Chandra, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[124] M. Bennett, "An attempt at a unified theory of the neocortical microcircuit in sensory cortex," *Frontiers in Neural Circuits*, vol. 14, p. 40, 2020.

[125] O. Mutlu, "Memory scaling: A systems architecture perspective," in *2013 5th IEEE International Memory Workshop*, IEEE, 2013, pp. 21–25.

[126] V. Seshadri et al., "Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 273–287.

[127] N. Hajinazar et al., "Simdram: A framework for bit-serial simd processing using dram," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 329–345.

[128] G. F. Oliveira et al., "Mimdram: An end-to-end processing-using-dram system for high-throughput, energy-efficient and programmer-transparent multiple-instruction multiple-data computing," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, 2024, pp. 186–203.

[129] J. Park et al., "Flash-cosmos: In-flash bulk bitwise operations using inherent computation capability of nand flash memory," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2022, pp. 937–955.

[130] S. Ghose, A. Boroumand, J. S. Kim, J. Gómez-Luna, and O. Mutlu, "Processing-in-memory: A workload-driven perspective," *IBM Journal of Research and Development*, vol. 63, no. 6, pp. 3–1, 2019.

[131] Y. Kim et al., "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, 2014.

[132] O. Mutlu, "The rowhammer problem and other issues we may face as memory becomes denser," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, IEEE, 2017, pp. 1116–1121.

[133] O. Mutlu and J. S. Kim, "Rowhammer: A retrospective," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 8, pp. 1555–1571, 2019.

[134] O. Mutlu, A. Olgun, and A. G. Yağlıkçı, "Fundamentally understanding and solving rowhammer," in *Proceedings of the 28th Asia and South Pacific Design Automation Conference*, 2023, pp. 461–468.

[135] H. Luo et al., "Rowpress: Amplifying read disturbance in modern dram chips," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–18.

[136] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Comput. Surv.*, vol. 52, no. 4, Aug. 2019, ISSN: 0360-0300. DOI: 10.1145/3320060. [Online]. Available: https://doi.org/10.1145/3320060.

[137] R. VanRullen and C. Koch, "Is perception discrete or continuous?" *Trends in cognitive sciences*, vol. 7, no. 5, pp. 207–213, 2003.

[138] K. Rayner, "Eye movements in reading and information processing: 20 years of research.," *Psychological bulletin*, vol. 124, no. 3, p. 372, 1998.

[139] H. Pashler, "Dual-task interference in simple tasks: Data and theory.," *Psychological bulletin*, vol. 116, no. 2, p. 220, 1994.

[140] S. Thorpe, D. Fize, and C. Marlot, "Speed of processing in the human visual system," *nature*, vol. 381, no. 6582, pp. 520–522, 1996.

[141] J. J. DiCarlo, D. Zoccolan, and N. C. Rust, "How does the brain solve visual object recognition?" *Neuron*, vol. 73, no. 3, pp. 415–434, 2012.

[142] R. M. Cichy, D. Pantazis, and A. Oliva, "Resolving human object recognition in space and time," *Nature neuroscience*, vol. 17, no. 3, pp. 455–462, 2014.

[143] İ. E. Yüksel et al., "Simultaneous many-row activation in off-the-shelf dram chips: Experimental characterization and analysis," in *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, IEEE, 2024, pp. 99–114.

[144] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, "A case for exploiting subarray-level parallelism (salp) in dram," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 368–379, 2012.

[145] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, "Low-cost inter-linked subarrays (lisa): Enabling fast inter-subarray data movement in dram," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2016, pp. 568–580.

[146] JEDEC Solid State Technology Association, *DDR5 SDRAM specification (JESD79-5C)*, Available at https://www.jedec.org, JEDEC, Apr. 2024.

[147] T. Vogelsang, "Understanding the energy consumption of dynamic random access memories," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE, 2010, pp. 363–374.

[148] ONFI Workgroup, *Open NAND Flash Interface specification (revision 5.2)*, Available at http://www.onfi.org, Oct. 2024. [Online]. Available: https://onfi.org/specs.html.

[149] M. M. Shulaker et al., "Carbon nanotube computer," *Nature*, vol. 501, no. 7468, pp. 526–530, 2013.

[150] M. M. S. Aly et al., "Energy-efficient abundant-data computing: The n3xt 1,000 x," *Computer*, vol. 48, no. 12, pp. 24–33, 2015.

[151] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[152] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The bloomier filter: An efficient data structure for static support lookup tables," in *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, 2004, pp. 30–39.

[153] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, 2014, pp. 75–88.

[154] T. M. Graf and D. Lemire, "Xor filters: Faster and smaller than bloom and cuckoo filters," *Journal of Experimental Algorithmics (JEA)*, vol. 25, pp. 1–16, 2020.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every written paper or thesis authored during the course of studies. **In consultation with the supervisor**, one of the following two options must be selected:

☒ I hereby declare that I authored the work in question independently, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used no generative artificial intelligence technologies[1].

☐ I hereby declare that I authored the work in question independently. In doing so I only used the authorised aids, which included suggestions from the supervisor regarding language and content and generative artificial intelligence technologies. The use of the latter and the respective source declarations proceeded in consultation with the supervisor.

Title of paper or thesis:

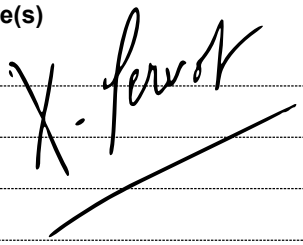
A Thousand Brains on a Thousand Chips

Authored by:
If the work was compiled in a group, the names of all authors are required.

Last name(s):	**First name(s):**
Servot	**Xavier**

With my signature I confirm the following:
- I have adhered to the rules set out in the Citation Guidelines.
- I have documented all methods, data and processes truthfully and fully.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for originality.

Place, date	**Signature(s)**
Zürich, Switzerland, 01.02.2026	

If the work was compiled in a group, the names of all authors are required. Through their signatures they vouch jointly for the entire content of the written work.

[1] For further information please consult the ETH Zurich websites, e.g. https://ethz.ch/en/the-eth-zurich/education/ai-in-education.html and https://library.ethz.ch/en/researching-and-publishing/scientific-writing-at-eth-zurich.html (subject to change).