PROCESSING CONVOLUTIONAL NEURAL NETWORKS ON CACHE

João Vieira[⋆] *Nuno Roma*[⋆] *Gabriel Falcao*[†] *Pedro Tomás*[⋆]

[⋆] INESC-ID, Instituto Superior Técnico, University of Lisboa, Portugal
[†] Instituto de Telecomunicações, University of Coimbra, Portugal

ABSTRACT

With the advent of Big Data application domains, several Machine Learning (ML) signal-processing algorithms, such as Convolutional Neural Networks (CNNs), are required to process progressively larger datasets at a great cost in terms of both compute power and memory bandwidth. Although dedicated accelerators have been developed targeting this issue, they usually require moving massive amounts of data across the memory hierarchy to the processing cores and low-level knowledge of how data is stored in the memory devices to enable in-/near-memory processing solutions. In this paper, we propose and assess a novel mechanism that operates at cache level, leveraging both data-proximity and parallel processing capabilities, enabled by dedicated fully-digital vector Functional Units (FUs). We also demonstrate the integration of this mechanism in a conventional Central Processing Unit (CPU). The obtained results show that our engine provides performance improvements on CNNs ranging from 3.92× to 16.6×.

Index Terms— CNNs, SIMD, Near-cache processing

1. INTRODUCTION

Convolutional Neural Networks (CNNs) are an important class of non-supervised Machine Learning (ML) applications frequently used for data categorization [1–3] and detection [4–6]. Some of these applications, such as object detection, are performed close to where the data is acquired. Therefore, such applications are targeted in embedded systems featuring low-end Central Processing Units (CPUs) such as ARM cores. CNNs are known to be data-driven, requiring the processing of large amounts of data. Transferring such datasets across the memory hierarchy imposes significant performance and power overheads [7], which limits system's performance.

Furthermore, most signal-processing operations performed by CNNs are computationally simple. For example, convolutional and fully-connected layers (shown in Figure 1) only apply vector multiplication and addition, while most of the time is spent fetching the operands from the memory and storing back the results. Hence, the CPU spends most of the time waiting for the operands to be retrieved and a small percentage of the execution time is actually spent on processing. This raises the question of whether more efficient approaches, closer to where the data is stored, can be employed to deal with these workloads.

Work supported by national funds through Fundação para a Ciência e a Tecnologia (FCT), under projects UIDB/50021/2020 and PTDC/EEI-HAC/30485/2017–HAnDLE (INESC-ID), UIDB/EEA/50008/2020 (Instituto de Telecomunicações), and research grant SFRH/BD/144047/2019.

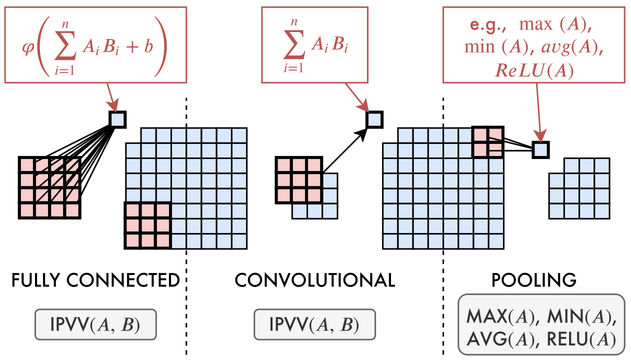
$\varphi\left(\sum_{i=1}^{n} A_i B_i + b\right)$
$\sum_{i=1}^{n} A_i B_i$
e.g., max (A), min (A), $avg(A)$, $ReLU(A)$
FULLY CONNECTED
CONVOLUTIONAL
POOLING
IPVV(A, B)
IPVV(A, B)
MAX(A), MIN(A), AVG(A), RELU(A)

Fig. 1: Most common CNN layers: Fully Connected, Convolutional and Pooling. The gray boxes show the commands of the devised engine that are used to implement these layers.

In this work, we propose the CCS, a system based on [8] that couples a Single Instruction Multiple Data (SIMD) unit to the Last-Level Cache (LLC) of an ARM CPU, as shown in Figure 2. The CCS operates near the memory device where the operands are actually stored, reducing the data movements. Furthermore, it processes an entire cache line per cycle, enabling massive parallelism. We assess the performance benefits of the devised mechanism by offloading the execution of convolutional and pooling layers, which are responsible for over 90% of most CNN's workload [9] (e.g., ResNet, AlexNet and VGGNet), to the CCS and comparing the results to those obtained with an optimized ARM Cortex-A53 implementation.

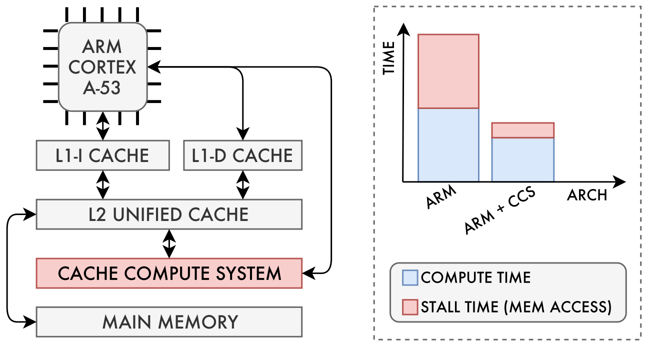

Fig. 2: Integration of the devised Compute Cache System (CCS) with the memory hierarchy of a conventional ARM Cortex-A53 core.

All in all, the contributions of this work are:

- Simulation model of the CCS, a system based on the work presented in [8], for the gem5 simulator;
- Library written in C to program and control the CCS;
- Convolutional kernels adapted to use the CCS;
- Analysis of performance improvements provided by the CCS over an ARM Cortex-A53 CPU for CNNs.

2. RELATION TO PRIOR WORK

Some previous proposals aiming at accelerating the execution of CNNs, by moving the processing near to where the operands are stored, include in-memory [10–17] and in-/near-cache [18–20] computing approaches, where specific modifications to the memory structures are performed to enable near-data processing. Such solutions leverage the data proximity and memory organization to perform multiple operations in parallel, allowing significant performance boosts. However, these solutions often impose selective data placements, which requires the programmer to know how data is stored in the physical memory device. Furthermore, these solutions are limited in terms of the number of operations they implement, and the simple arithmetic routines required by CNNs may take hundreds of cycles to complete.

More recently, in-memory processing using emerging Non-Volatile Memory (NVM) technologies also appeared, in the context of CNNs that take advantage of the analog computing capabilities of Resistive Random-Access Memory (RRAM) to perform the convolutions between input data and fixed kernels encoded in the memristors [21–24]. However, since these operations take place in the analog domain, costly Analog-to-Digital Converters (ADCs) and Digital-to-Analog Converters (DACs) need to be added to the memory chips. Furthermore, the error introduced by these memristors technologies to perform analog convolutions is substantial, mostly due to process variations [25] and temperature fluctuations [26], severely compromising the reliability of the results.

In contrast, the engine devised in [8] makes use of a conventional arithmetic and logic vector unit supporting a much wider range of operations than conventional in-memory computing structures, while leveraging the parallelism enabled by accessing an entire cache row per cycle. Among the supported vector operations, some particular ones such as the dot-product are commonly used in the context of CNNs. Furthermore, the engine is fully digital, which is an advantage over error-prone analog RRAM-based solutions. In [8], a proof-of-concept of the designed architecture was implemented in a Register Transfer Level (RTL) description and evaluated using a simple soft-CPU and a single-level cache structure. Although such an approach is convenient for a preliminary proof-of-concept, it hardly tells anything about how such engine would impact the performance of a realistic hard-CPU, equipped with its own SIMD unit and connected to a multi-level cache hierarchy.

In this work, we re-engineer the original architecture of the mechanism presented in [8] to optimally execute CNN's convolutional and pooling layers, by adding specific commands to its Instruction Set Architecture (ISA). Furthermore, we evaluate its performance when connected to a conventional ARM Cortex-A53 CPU core equipped with the NEON SIMD engine and a multi-level cache hierarchy. We compared the obtained results with those that were obtained by executing the same workload in an equivalent system without the CCS, with remarkable performance gains.

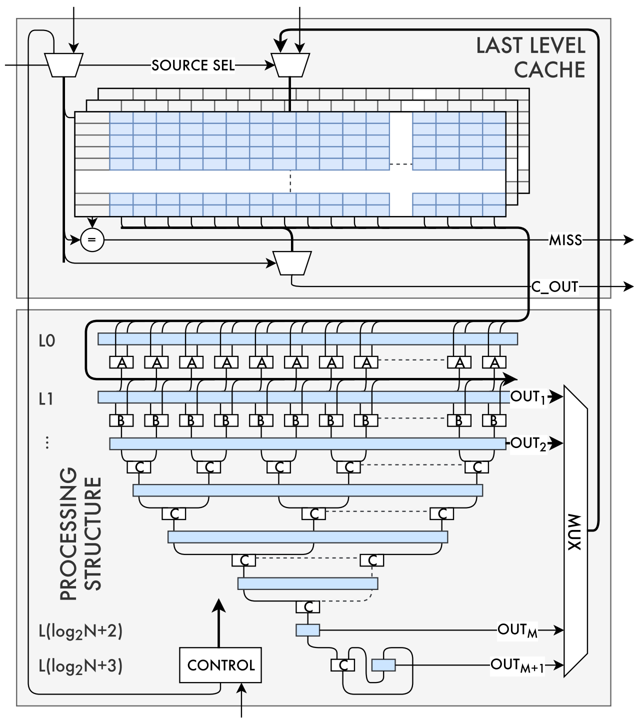

Fig. 3: Processing structure of the CCS.

The next section details the architecture of the re-engineered CCS, together with the developed mechanisms to support the execution of convolutional and pooling layers of CNNs.

3. CACHE COMPUTE ARCHITECTURE

3.1. Processing structure

The proposed CCS is composed of several levels of quasi-generic functional units arranged in a binary tree, as shown in the Figure 3. Each level of the binary tree contains processing nodes capable of executing a specific number (and type) of arithmetic, logic and shift operations. The different operations that are supported at each level are combined through commands that perform a whole complex signal-processing function. Each command specifies the operands, the operation to be performed, and the destination addresses of the results.

In addition to the commands supported by the engine devised in [8], the re-engineered CCS extends the former set of commands in order to further facilitate the implementation of signal-processing functions, such as those used in the pooling layers of CNNs. To accommodate the new functions, the functional units at each level were modified accordingly. For example, to implement the Rectified Linear Unit (ReLU), the functional units of the first level were modified such that there is a control combination that allows the output to be equal to the input, when it is positive, or zero, otherwise. The commands used to implement each layer are listed in gray boxes

```
#define L_SIZE          // size of cache line
#define DATA_WIDTH      // width of data matrix
#define DATA_HEIGHT     // height of data matrix
#define KERNEL_LENGTH   // size of the kernel

external int **data;    // data addr
external int *kernel;   // kernel addr
external int **result;  // result addr
int a[L_SIZE];          // data buffer

// 2-D convolution
for (int i = 0; i < DATA_WIDTH; i++) {
  for (int j = 0; j < DATA_HEIGHT; j++) {
    gather_data(a, i, j);
    ccs_setup(IPVV, KERNEL_LENGTH, 0, a, kernel,
        result + i * DATA_WIDTH + j, 1);
    ccs_start();
  }
}
ccs_wait();
```

Listing 1: 2-D convolution using the CCS.

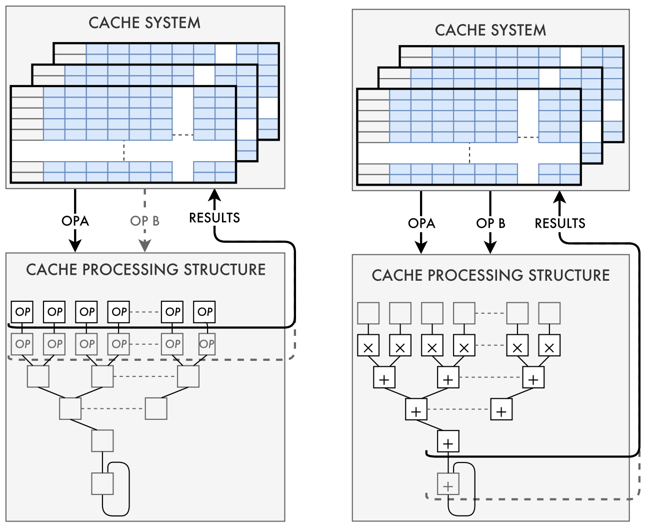

Fig. 4: Execution of two different instructions in the CCS. Figure 4a: ReLU; Figure 4b: dot-product.

in Figure 1. Accordingly, the CCS implements a total of 47 distinct commands. Nevertheless, the CCS can be easily modified to implement other signal-processing functions used in the context of CNNs or other application domains.

3.2. Integration with the data cache

Unlike a CPU, the CCS does not access the cache word-by-word. Instead, it accesses an entire cache line. Furthermore, the CCS is connected to a dedicated port of the cache, not compromising the bandwidth between the cache and the CPU.

When the operands are not in the cache, they are fetched from the main memory, similarly to what would happen if they were requested by the CPU. In that case, the CCS issues a read request to the memory subsystem and waits until it is finished.

The operands (fetched from the data cache) are delivered to the first level of the CCS and the computation starts as soon as all the operands are available (see Figure 3).

3.3. Pipeline Dataflow

There are four levels of the CCS pipeline that produce output, depending on the command being executed. For example, a dot-product of two vectors has to go all the way through the CCS and the result is collected in the last level, whilst a ReLU of a vector is entirely executed in the first level. Hence, the execution time of the CCS only depends on the type of command being issued and on the locality of the operands.

Independently of the considered operation, the designed pipeline binary tree ensures the maximum processing throughput and allows more than one command to be simultaneously under execution (in different pipeline stages).

3.4. Programming the CCS

To execute commands in the CCS, an interface between the CPU and the CCS is required. This interface is composed of memory-mapped registers through which the CPU specifies the operation code, an optional constant, the operands and result vector dimension, stride and start address. As soon as the operands are available, the computation starts. When it finishes, a write request is issued to the memory subsystem and the processor is notified that the results are available.

To facilitate the process of writing the parameters into the programming registers and controlling the CCS, a library was written, using ARM assembly. This allows spending the least number of cycles on configuring the CCS, minimizing the overhead of doing so. The library contains the following four routines: `ccs_setup(<args>)`, `ccs_start()`, `ccs_check()`, and `ccs_wait()`.

Furthermore, a software framework was developed to easily offload convolutional and pooling layers of CNNs to the CCS. This framework provides routines to gather the operands and to align them within memory so that optimal results can be achieved when using the CCS. Besides gathering the operands, these routines also program the CCS to operate over the data that was previously gathered. The example in Listing 1 shows the implementation of a 2-D convolution using the CCS.

3.5. Summary example

Figure 4 provides two examples to show how the mechanism implements operations with different requirements. Figure 4a) illustrates the execution of an operation flow typical of a ReLU. After fetching the operands from the cache, the processing structure simultaneously executes a given instruction over all the elements of the vector. The result has the same size as a cache line and the results are collected from the first or the second level of the processing structure. Figure 4b) depicts the execution of a dot-product operation. This example illustrates how different levels of the CCS are programmed to operate differently, by performing complex operations together. The first level of the processing structure does not perform any operation, the second multiplies the elements of both vectors, and the remaining levels sum all the outputs of level two.

4. SIMULATING THE CCS

For assessing the performance improvements attained by the CCS over a conventional CPU equipped with a SIMD unit, a simulation model of the re-engineered CCS for gem5 was developed. The developed CCS model was coupled to a CPU through a memory-mapped interface. Since the CCS's architecture is highly pipelined and stages can be added without affecting its throughput, it was considered that the CCS is capable of operating at the CPU's frequency. An additional module was used to forward memory requests between the CPU and the memory subsystem or between the CPU and the CCS memory-mapped programming registers, depending on the target address. If the target address of a CPU's memory request is reserved to the CCS programming registers, the request is forwarded to the CCS, otherwise it is forwarded to the memory subsystem. The CCS was coupled to the L2 cache (in the reference system, the LLC). For translating the virtual addresses of the operands, the CCS was also equipped with dedicated translation lookaside buffer (TLB) and page-walker. The model used for the CPU was an in-order ARM core refined with the gem5-X [27] framework for obtaining reliable performance results corresponding to an ARM Cortex-A53.

The conceived CCS model was designed to target 32-bit fixed-point vector operations. The size of a cache line in the reference CPU is 64 bytes wide. Hence, the developed model of the CCS can process 16 words of 32 bits per clock cycle (i.e., the input of the CCS is 16 32-bit words wide).

5. EXPERIMENTAL RESULTS AND DISCUSSION

To assess the performance benefits offered by the CCS in the context of CNNs, five kernels that are in the basis of CNNs were built and executed using only the ARM Cortex-A53 CPU core and both the CPU core and the CCS. Three of the five kernels that were tested are convolutions (1-D, 2-D and 3-D), whereas the remaining two kernels are max-pooling and ReLU. Table 1 lists the parameters of each kernel, which were picked accordingly to data formats frequently used in the context of widely used CNNs (e.g., ResNet, AlexNet and VGGNet).

Figure 5 shows the obtained execution times and attained speedups. As expected, the time spent accessing the memory was drastically reduced for all kernels when executing in the CCS, which is due to processing data closer to where it

Table 1: Parameters of the kernels used to assess the benefits offered by the CCS to execute CNN-related workload.

	Parameter	Value
Conv 1-D	*Data size*	1000000
	Kernel size	15
Conv 2-D	*Data size*	$\{1000, 1000\}$
	Kernel size	$\{3, 3\}$
Conv 3-D	*Data size*	$\{100, 100, 100\}$
	Kernel size	$\{3, 3, 3\}$
Max Pool	*Data size*	$\{999, 999\}$
	Patch size	$\{3, 3\}$
	Stride size	$\{3, 3\}$
ReLU	*Data size*	$\{1000,1000\}$

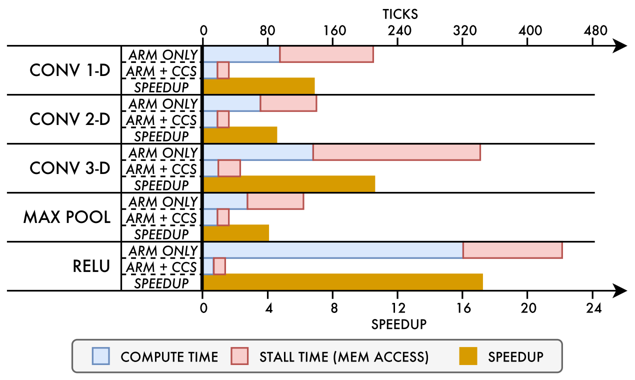

Fig. 5: Execution time and speedup of the five selected kernels. The blue bars represent processing time, whilst red bars show the time that processing system stalls waiting for the operands.

is stored. Furthermore, the CCS fetches an entire cache line at once, which significantly reduces the number of memory accesses. The time spent on processing the data is also lower when using the CCS, which is mainly due to the parallelism enabled by its processing structure. Overall, both the reduction of memory accesses and the faster processing due to the parallelism enabled by the CCS contribute to the performance improvements offered by the CCS. The attained speedups range from 3.92× to 16.6×.

The speedup attained for the ReLU kernel is significantly higher than for the remaining kernels. A possible explanation for this involves the complexity of executing the ReLU in the CPU. While all other kernels only involve vector arithmetic operations, that can be often parallelized using the ARM NEON engine, the ReLU involves assessing whether a value is positive or negative. Not only this may not be parallelized using NEON, but also each assessment may take more time than a single arithmetic operation. On the other hand, the CCS takes even less time performing the ReLU operation (as the results are extracted from the first level) than executing the dot-product, which is how convolution is implemented.

6. CONCLUSIONS

This paper proposes a novel method for exploiting near-data processing for CNNs. The mechanism devised in this work, the CCS, operates near the LLC of the CPU, leveraging both the proximity to the data (fetching and storing data becomes less time consuming) and the access to an entire cache line per cycle. The CCS was integrated with an ARM Cortex-A53 CPU and simulated using the gem5 architectural simulator. Five relevant signal-processing kernels in the context of CNNs were executed and the performance benefits derived from using the CCS were evaluated. The speedups obtained by using the CCS over using only the CPU range from 3.92× to 16.6×. Furthermore, the attained results show that using the CCS does not only reduce the time spent accessing memory, but also the time spent doing actual processing, which is due to the parallelism offered by the CCS processing structure.

7. REFERENCES

[1] A. Karpathy et al., "Large-Scale Video Classification with Convolutional Neural Networks," in *CVPR*. 2014, pp. 1725–1732, IEEE Computer Society.

[2] J. Pons and X. Serra, "Randomly Weighted CNNs for (Music) Audio Classification," in *ICASSP*. 2019, pp. 336–340, IEEE.

[3] T. Agrawal et al., "On Evaluating CNN Representations for Low Resource Medical Image Classification," in *ICASSP*. 2019, pp. 1363–1367, IEEE.

[4] R. Girshick et al., "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation," in *CVPR*. 2014, pp. 580–587, IEEE Computer Society.

[5] J. Sun et al., "Topic Detection in Conversational Telephone Speech Using CNN with Multi-stream Inputs," in *ICASSP*. 2019, pp. 7285–7289, IEEE.

[6] P. Barmpoutis et al., "Fire Detection from Images Using Faster R-CNN and Multidimensional Texture Analysis," in *ICASSP*. 2019, pp. 8301–8305, IEEE.

[7] W. Wulf and S. McKee, "Hitting the memory wall: implications of the obvious," *SIGARCH Computer Architecture News*, vol. 23, no. 1, pp. 20–24, 1995.

[8] J. Vieira et al., "Exploiting Compute Caches for Memory Bound Vector Operations," in *SBAC-PAD*. 2018, pp. 197–200, IEEE.

[9] J. Cong et al., "Minimizing Computation in Convolutional Neural Networks," in *ICANN*. 2014, vol. 8681 of *Lecture Notes in Computer Science*, pp. 281–290, Springer.

[10] S. Li et al., "DRISA: a DRAM-based reconfigurable in-situ accelerator," in *MICRO*. 2017, pp. 288–301, ACM.

[11] V. Seshadri et al., "Ambit: in-memory accelerator for bulk bitwise operations using commodity DRAM technology," in *MICRO*. 2017, pp. 273–287, ACM.

[12] V. Seshadri et al., "Fast Bulk Bitwise AND and OR in DRAM," *Computer Architecture Letters*, vol. 14, no. 2, pp. 127–131, 2015.

[13] S. Yitbarek et al., "Exploring specialized near-memory processing for data intensive operations," in *DATE*. 2016, pp. 1449–1452, IEEE.

[14] P. Chi et al., "PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory," in *ISCA*. 2016, pp. 27–39, IEEE Computer Society.

[15] M. Cheng et al., "TIME: A Training-in-memory Architecture for Memristor-based Deep Neural Networks," in *DAC*. 2017, pp. 26:1–26:6, ACM.

[16] Y. Wang et al., "Towards Memory-Efficient Allocation of CNNs on Processing-in-Memory Architecture," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 6, pp. 1428–1441, 2018.

[17] J. Liu et al., "Processing-in-Memory for Energy-Efficient Neural Network Training: A Heterogeneous Approach," in *MICRO*. 2018, pp. 655–668, IEEE Computer Society.

[18] S. Aga et al., "Compute Caches," in *HPCA*. 2017, pp. 481–492, IEEE Computer Society.

[19] X. Wang et al., "Bit Prudent In-Cache Acceleration of Deep Convolutional Neural Networks," in *HPCA*. 2019, pp. 81–93, IEEE.

[20] C. Eckert et al., "Neural Cache: Bit-Serial In-Cache Acceleration of Deep Neural Networks," *IEEE Micro*, vol. 39, no. 3, pp. 11–19, 2019.

[21] A. Shafiee et al., "ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars," in *ISCA*. 2016, pp. 14–26, IEEE Computer Society.

[22] M. Imani et al., "FloatPIM: in-memory acceleration of deep neural network training with high precision," in *ISCA*. 2019, pp. 802–815, ACM.

[23] D. Fujiki et al., "In-Memory Data Parallel Processor," in *ASPLOS*. 2018, pp. 1–14, ACM.

[24] J. Vieira et al., "A Product Engine for Energy-Efficient Execution of Binary Neural Networks Using Resistive Memories," in *VLSI-SoC*, 2019.

[25] P. Pouyan et al., "RRAM variability and its mitigation schemes," in *PATMOS*. 2016, pp. 141–146, IEEE.

[26] X. Liu et al., "HR^{3}AM: A Heat Resilient Design for RRAM-based Neuromorphic Computing," in *ISLPED*. 2019, pp. 1–6, IEEE.

[27] Y. Qureshi et al., "Gem5-X: A Gem5-Based System Level Simulation Framework to Optimize Many-Core Platforms," in *SpringSim*. 2019, pp. 1–12, IEEE.