gem5-ndp: Near-Data Processing Architecture Simulation From Low Level Caches to DRAM

João Vieira*, Nuno Roma*, Gabriel Falcao†, Pedro Tomás*

*INESC-ID, Instituto Superior Técnico, University of Lisbon, Portugal
†Instituto de Telecomunicações, University of Coimbra, Portugal

***Abstract*—Unlike standard accelerators, the performance of Near-Data Processing (NDP) devices highly depends on the operation of the surrounding system, namely, the Central Processing Unit (CPU) and the memory hierarchy. Therefore, to accurately evaluate the gain provided by such devices, the entire processing system must be considered. Recent proposals redesigned existing architectural simulators to estimate the performance of NDP devices. However, the conclusions that can be drawn from using these frameworks are limited, and they fail to provide full support to simulate these devices (e.g., most simulators do not allow simultaneous operation of the CPU and the NDP device). In this paper, a novel framework (called gem5-ndp) based on the gem5 architectural simulator is proposed, providing full support to the development, validation, and evaluation of novel NDP architectures. To illustrate the process of developing and integrating an NDP device with a processing system using the proposed framework, as well as to demonstrate its viability and benefits, two case studies are also proposed and thoroughly discussed. gem5-ndp significantly improves the performance evaluation confidence of NDP devices with results showing that classical approaches lead to a deviation of up to $54.9\,\%$ when compared with results obtained with gem5-ndp.**

***Index Terms*—Near-Data Processing, Multi-Level Memory Hierarchies, Simulation Framework, Data-Parallel Processing.**

I. Introduction

The emergence of data-intensive algorithms in several application domains (e.g., for image and video processing, bioinformatics, machine learning, and particularly deep learning), has highlighted the inability of memory subsystems to feed data at the rate required by standard parallel computing units, including Central Processing Units (CPUs) [1]. Moreover, on big data applications where the data re-utilization in cache is low, data transfers pose significant energy consumption overheads. To overcome these limitations, accelerators and co-processors operating near-memory have been proposed. Such solutions aim to benefit from a much higher memory bandwidth (compared to the CPU), allowing to accelerate workloads that would otherwise be memory-bound.

Firstly introduced during the '80s, the Processing in Memory (PIM) paradigm [2], [3] aims at re-purposing existing Dynamic Random Access Memory (DRAM) resources to enable bit-line computation. Later, this paradigm was extended to Static Random Access Memory (SRAM), allowing to perform active computation on caches [4]. While devices using this technique usually offer massive parallelism, being able to process whole memory lines in a single clock cycle, they are only capable of performing very simple bit-wise operations [5], and may require thousands of computation cycles (and complex control circuitry) to implement more elaborate arithmetic operations using bit-line computation [6]. Moreover, due to its analog nature, bit-line computation is error-prone, and is only viable if a small number of memory lines are combined during an operation [7]. As a result, PIM-based solutions usually do not significantly benefit data-parallel algorithms, being only suitable for a small set of programs whose majority of operations are efficiently implemented by PIM accelerators. Furthermore, enabling PIM requires to change the internal architecture of memory devices. Hence, as PIM accelerators often target a single algorithm [8], it seems counter-intuitive to re-design memory chips to benefit a specific workload, while risking decreasing the overall performance of the system.

Another similar processing paradigm is Resistive Random Access Memory (RRAM)-PIM [9]. Solutions using this computation technique rely on the resistive properties of RRAM to implement analog or digital computation. By controlling the impedance of RRAM cells and the input voltage, it is possible to implement analog logic operations as well as analog Multiply–accumulate (MAC)–the main operation required by Neural Networks (NNs) [10]. On the other hand, by combining several memristors, it is possible to implement digital bit-wise operations [11], [12]. However, RRAM fabrication is known to have significant process variations, which makes memristors uneven within the same device and across devices [13]. Therefore, a significant error is introduced in RRAM-based analog computations, ultimately invalidating the results. On the other hand, digital computations using RRAM are limited to the same simple logic instructions achieved with PIM, worsened by the fact that several memristors are required to implement a single bit-wise operation between two bits (reducing the parallelism that can be achieved).

To circumvent these limitations, a different approach has been adopted by Near-Data Processing (NDP) solutions, featuring fully digital architectures directly connected to the memory devices [14]–[16]. Although these solutions usually benefit from a lower bandwidth than those implemented within the memory chips, their architectures are not restricted by the memory resources, resulting in more versatile devices, capable of efficiently processing complex workloads, while still benefiting from a higher bandwidth to the memory compared to the CPU. Moreover, these devices already reduce significantly

energy consumption across wires, as they are physically closer to the memory than conventional CPUs. Naturally, these devices have a broader scope than PIM-based solutions, and general-purpose co-processors can be implemented using this paradigm.

However, the performance evaluation of NDP devices is critically different from that of standard accelerators, since the behavior of the memory devices and the cooperation with the host CPU highly affects their performance. Classical evaluation approaches can hardly take into account important factors such as the entropy generated at the memory hierarchy level for being shared among multiple devices, or the synchronization overheads between the CPU and the NDP device. Therefore, simulating the architecture of an NDP device alone tends to poorly reproduce its real performance when integrated with a standard processing system. In fact, the most viable and accurate approach to validate and predict the performance benefits of NDP devices is often the simulation of the entire processing system. However, such task is not simple, and there is no standard tool to simulate systems that include NDP devices. Some works re-purpose existing tools to estimate the performance of custom systems. For example, in [17], the authors use a Graphics Processing Unit (GPU) simulator to estimate the performance of a PIM device. Other works extend existing simulators to partially support specific NDP architectures [18]–[20]. In contrast, this paper proposes a novel simulation framework, called gem5-ndp, that provides full support to develop, validate and evaluate the performance of new NDP architectures. Furthermore, gem5-ndp is based on the widely established gem5 architectural simulator [21], and provides full support for simultaneous operation of the CPU and NDP devices, simple integration of NDP devices with processing systems, and memory configuration mechanisms.

To validate and evaluate the proposed simulation framework, the conducted experimental procedure considered two NDP devices: (1) a simple square root calculation circuit, which was used to illustrate the development and integration of a custom NDP architecture; and (2) a data-parallel NDP device to execute several benchmarks from data-intensive application domains, showing the viability of the devised framework and its benefits when comparing with classical evaluation scenarios.

All in all, the contributions of this paper are the following:

- A Gem5-based NDP simulation framework providing full support to develop, validate and evaluate NDP devices;
- Two case studies for demonstrating and validating the proposed framework, including the creation of simulation models for the two considered systems, their integration with the surrounding processing system, and the development of host code to communicate with the NDP devices;

The rest of this paper is organized as follows. Section II details the proposed NDP simulation framework. Section III presents two case studies: (III-A) a simple near-data square root calculation mechanism, to illustrate the process of developing and integrating an NDP device using gem5-ndp; and (III-B) a more complex bidimensional NDP array, to demonstrate the benefits of using the proposed framework for developing and evaluating new NDP architectures. In Section IV, the main experimental results are presented. Section V summarizes relevant related work. Finally, Section VI concludes this paper.

II. NDP Simulation Framework

The proposed gem5-ndp is based on the gem5 architectural simulator, allowing an accurate event-driven simulation on a clock cycle basis. Accordingly, the latency associated with data movements or hardware operations is simulated through triggering events after the number of cycles corresponding to the latency of the simulated components. However, while gem5 allows simulating standard processing systems out of the box (using already-implemented simulation models), simulating novel devices requires the definition of their architectural models and connecting them with the remaining system. This task can be particularly complex for NDP devices, since they have to be integrated with the CPU (for control purposes), and the memory hierarchy (to directly fetch and store data from/to one or more memory devices), and to enable the translation between the virtual and physical address spaces.

To facilitate the process of integrating these devices with standard processing systems using gem5, gem5-ndp devises three main mechanisms. First, an architectural model implementing a programming interface and a data gather/scatter mechanism is provided to easily connect the custom NDP devices with the existing CPU and memory hierarchy. A custom NDP device based on this provided model has access to functions that allow to communicate with these structures, making the complex protocols required for implementing control and data transfers transparent to the developer, while still allowing fine-grained control over data movements. Second, convenient gem5 simulation scripts are also provided to automatically connect NDP devices to the CPU, map their programming interfaces into the CPU address space, and connect them to one or more levels of the memory hierarchy. Finally, gem5-ndp includes a simple and organized programming paradigm for developing applications using NDP devices based on the provided architectural model.

A. gem5-ndp Architectural Model

The proposed architectural model consists of a layer on top of gem5 that deals with the complex architecture of the whole processing system, allowing users to easily integrate their NDP architectures. It does not only generate the required control circuitry to automatically deal with memory requests, but also provides a programming interface to allow the necessary management by the CPU. It is important to mention that the gem5-ndp architectural model does not restrict in any way the internal architecture of the NDP devices. In particular, it is unaware of their Instruction Set Architecture (ISA) (and corresponding implementation) and the internal control required for their proper function (e.g., scheduling hardware). Moreover, although it provides the hardware infrastructure required to implement the communication between the CPU and the NDP, it is up to the developer to establish the communication protocol between them. Furthermore, gem5-ndp allows to connect an NDP device to any level of the memory hierarchy or even to

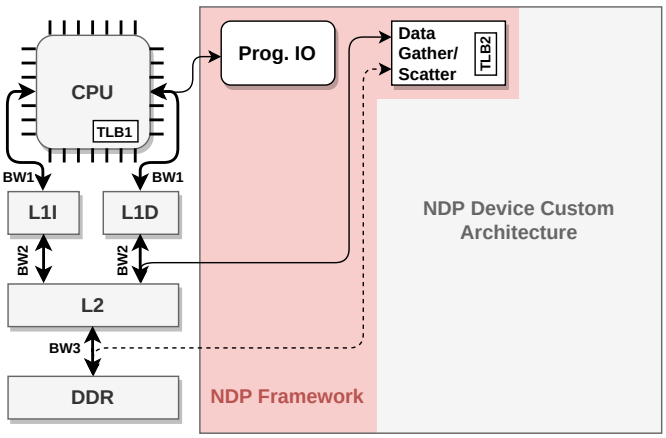


Fig. 1: Physical representation of the proposed NDP simulation framework. BW1, BW2, and BW3 represent configurable gem5-ndp parameters that define the bandwidths between the different levels of the memory hierarchy.

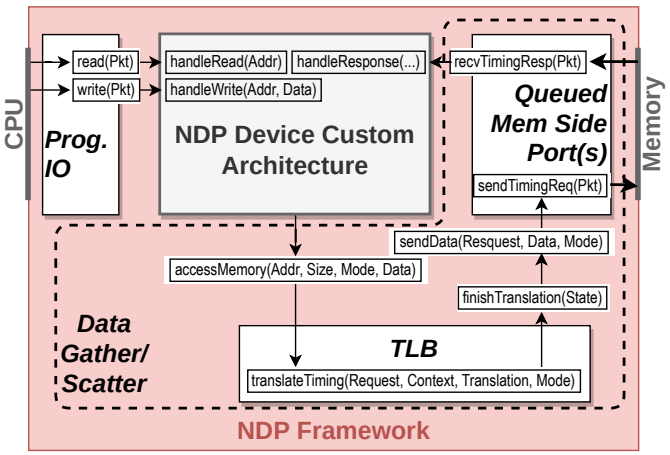


Fig. 2: Function diagram of gem5-ndp showing the relation between the framework software routines and the corresponding physical components illustrated in Fig. 1.

multiple levels, and it includes mechanisms to configure the bandwidths of each connection (see Fig. 1). The simulation of systems featuring multiple NDP devices is also supported.

The architectural model of gem5-ndp uses two existing structures of gem5: the Programmable IO (PIO) and the Queued Memory Port, as shown in Fig. 2. The PIO allows the creation of a programmable interface through which the NDP device can be managed by the CPU (Prog. IO block). This interface consists of a register bank that can be addressed by the CPU as a memory-mapped device. The register bank can be as large as required by the communication protocol established by the architect of the NDP device. At simulation level, reading or writing from/to addresses within the programmable interface will trigger a `read` or `write` procedure in the simulation framework, emulating the circuitry to deal with the request and pass any relevant data to the functions within the custom NDP device model (`handleRead` and `handleWrite`).

The Queued Memory Ports allow the connection between the NDP device and one or more memory devices from the hierarchy, for direct access. Additionally, gem5-ndp implements mechanisms to easily fetch the operands and store the results required/produced by the NDP device (Data Gather/Scatter block). For example, for fetching an entire vector from memory, the custom NDP device model only has to pass a descriptor to the framework using the `accessMemory` method, indicating the base address, the number of bytes to retrieve, and, optionally, a stride. The framework will then schedule the necessary memory requests to retrieve the data as fast as possible (`sendData` method), filtering out any extra bytes not fitting the descriptor (e.g., when a stride is specified). Optionally, the developer can also configure the latency and overall behaviour of the memory devices, by completely bypassing the memory controller. This option allows the developer to load or store data using a custom latency, which can be useful for NDP (and PIM) design exploration. As soon as the memory device finishes dealing with all the requests generated by the framework, the `handleResponse` method is called, returning the requested data to the NDP device (on loads) or acknowledging a successful write (on stores).

gem5-ndp also implements convenient structures to guarantee the translation of the virtual memory addresses accessed by the NDP device into their corresponding physical addresses. Whenever the custom NDP device issues a memory request, the framework asynchronously translates the base address of the descriptor using a dedicated secondary Translation Lookaside Buffer (TLB) (`translateTiming`) and sends the request to memory with the corresponding physical address (`sendData`). Furthermore, gem5-ndp reduces the number of accesses to the TLB by translating subsequent addresses within the same page through adding an offset to a physical address translated previously. Only when a page boundary is crossed an address is translated by explicitly accessing the TLB again. In the event of a page fault, gem5-ndp implements a page walker to fetch the new page from memory.

Memory models in gem5 follow typical hardware limitations and have a maximum request size determined by the architecture of its corresponding physical device. Therefore, when requesting/sending bursts of data from/to a memory device larger than the allowed request size, they must be split into smaller requests. To simulate this behavior, the Data Gather/Scatter module implemented by gem5-ndp automatically divides large memory requests into smaller ones, suitable to be sent to the memory device, and sends them sequentially, as shown in Fig. 3. First, the framework translates the base address of the descriptor passed by the custom NDP device, and sends a request with a size equal or smaller than the maximum allowed size by the targeted memory device. If the size of the data requested by the NDP device is larger than the maximum allowed by the memory device for a single request, the framework prepares a new request, translating its virtual address by adding an offset to the base address of the previous request if both addresses are within the same page, or

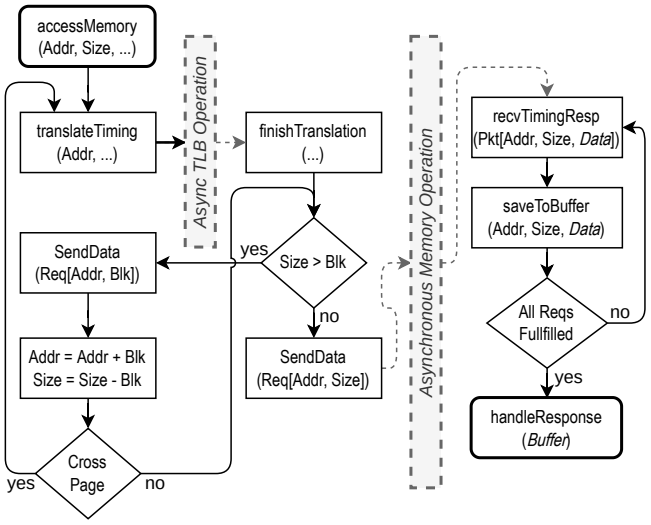


Fig. 3: Diagram showing how the gem5-ndp Data Gather/Scatter module unfolds large requests into smaller ones suitable to be sent to the memory devices.

accessing the TLB when the addresses are in different pages. This process is repeated until the original request made by the custom NDP device is entirely fulfilled.

Since gem5 is an event-driven simulator, the memory requests are resolved asynchronously by the memory device (similarly to a real system). Therefore, to ensure that all memory requests are resolved correctly, gem5-ndp implements a callback mechanism, which only makes the data available (in case of a read request) or acknowledges a successful write when all the requests issued to the memory device are fulfilled.

B. Gem5 System Emulation Scripts

gem5-ndp includes a comprehensive set of Python scripts that support the integration of the simulated NDP devices in a standard processing system, as illustrated in Listing 1. This example is logically divided into four parts, where Part 1 describes the targeted processing system, in this case featuring an Out-of-Order (OoO) CPU with two levels of cache and a DDR memory. Part 2 instantiates the included NDP device using the gem5-ndp architectural model, and connects it to the CPU and memory devices using crossbars (Part 3), whose bandwidth can also be adjusted to reflect the proximity of this device to memory through parameters such as the bus width of the interconnects (BW1, BW2, and BW3 in Fig. 1). It is worth noting that although the devised framework was developed targeting NDP devices, this feature also enables the simulation of PIM accelerators, by configuring the allowed bandwidth according to the one commonly found inside memory chips. Furthermore, gem5-ndp also provides detailed information regarding the memory transactions between the NDP devices and the memory hierarchy, which allows to determine the effective bandwidth between the two.

The provided scripts also exemplify how to map the NDP devices programming interface (provided by the framework

Listing 1: Partial gem5 simulation script integrating a standard processing system and memory hierarchy with an NDP device. In this example, the NDP device is connected to the L2 cache. Nevertheless, gem5-ndp allows to connect NDP devices to any memory device or even multiple devices.

```
# Standard processing system
system.cpu = DerivO3CPU()
system.cpu.icache = L1Cache()                                    (Part 1)
system.cpu.dcache = L1Cache()
system.l2cache = L2Cache()
system.mem_ctrl = DDR3_1600_8x8()

# NDP device
system.ndp_device = NDPDevice(                                   (Part 2)
    ndp_device_addr = 0x0
)

class CustomL1XBar(L2XBar):
    def __init__(self):
        super(CustomL1XBar, self).__init__()
        width = 32  # 256-bit bus width (BW1)

class CustomL2XBar(L2XBar):
    def __init__(self):
        super(CustomL2XBar, self).__init__()
        width = 32  # 256-bit bus width (BW2)
                                                                 (Part 3)
# Connect NDP device to CPU
system.l1bus = CustomL1XBar()
system.cpu.dcache_port = system.l1b.slave
system.ndp_device.cpu_port = system.l1b.master

# Connect NDP device to L2
system.l2xb = CustomL2XBar()
system.ndp_device.mem_port = system.l2b.slave
system.l2cache.cpu_side = system.l2bus.master

# Map NDP device into host address space
system.cpu.workload[0].map(
    0x200000000,  # Host address space
    0x0,          # NDP device address space
    4096          # Address range
)                                                                (Part 4)
# Emulated driver
system.ndp_driver = NDPAccelDriver(
    hardware = system.ndp_device,
    filename = "ndp_device" # file descriptor
)

# Start simulation
exit_event = m5.simulate()
```

architectural model) into the CPU address space, and configures an emulated driver (also provided by the framework architectural model) to implement the system calls required by virtual memory translation mechanisms (Part 4).

C. Programming and Operation of an NDP Device

To control the NDP devices from the program being executed in the CPU, gem5-ndp also includes a straightforward programming paradigm, which allows to communicate with the NDP devices by reading and writing from/to their programming interfaces (provided by gem5-ndp architectural model).

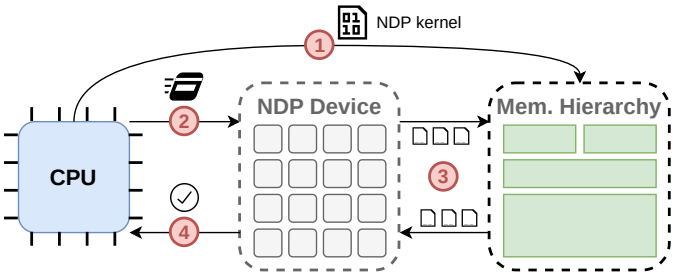


Fig. 4: Operation of an NDP device and interaction with the surrounding system: (1) the CPU stores the NDP kernel in memory; (2) the CPU triggers the NDP device to start operating; (3) the NDP device exchanges operands and results with the memory hierarchy; (4) the CPU polls the NDP device.

Fig. 4 exemplifies a possible execution flow of a prototyped NDP device using gem5-ndp. Since in this example the NDP device and the CPU share the same addressing space, the CPU starts by storing the NDP kernel (i.e., the sequence of instructions that will be decoded and executed by the NDP device circuitry) into memory (step 1). Then, it writes the memory address where the NDP kernel was stored into the NDP device programming registers (step 2), and triggers the NDP device to fetch the kernel from memory and start executing it (step 3). While the kernel is being executed, the NDP input data is fetched from the memory hierarchy, and the results are stored back. During this time, the CPU is free to execute a different workload, since both devices can simultaneously operate on different data. Finally, the CPU polls the NDP device programming register, waiting for the kernel completion (step 4). On the application side, the programmer manages the NDP device by simply reading and writing the programming registers, as shown in Listing 2. Whenever writing to an address corresponding to the programming interface of the NDP device, a `handleWrite` call is triggered in the gem5 model, while a read command triggers a `handleRead` call.

It is important to mention that while the NDP device is processing data, it is up to the developer to enforce that no invalid data is accessed, i.e., the CPU does not try to access memory positions being read or written by the NDP device. To ensure this, programs must be aware of what data is being used by the NDP devices, and implement proper synchronization using either high-level synchronization mechanisms (such as mutexes and barriers), or use the value of status registers in the NDP devices programming interfaces. Furthermore, thanks to the existence of virtual memory and process isolation mechanisms, it is also enforced that processes other than the ones that have hold of the NDP devices cannot access the data being processed by them. All in all, gem5-ndp offers guarantees that it is always possible to prevent the CPU to access memory regions involved in NDP computations.

Due to operating at several levels of the memory hierarchy, the use of NDP devices also requires awareness of data coherence across the several memory devices. The gem5-ndp programming paradigm also describes how to deal with this

Listing 2: Example of gem5-ndp-compatible C code including routines to initialize and control the NDP device.

```
int main() {
    // NDP driver initialization
    int fd = open("/dev/ndp_device", 0);
    volatile uint64_t *driver = NDP_ADDR;

    // Data is prepared; pointers are added to kernel
    DATA_TYPE dataset = ...
    void *kernel = ...

    // CPU launches kernel on NDP device
    *driver = (uint64_t) kernel;

    // CPU processes tasks in background
    ...

    // CPU waits for NDP to finish
    while (!(*driver));

    // CPU post-processes the results
    ...

    return 0;
}
```

important requirement of computing systems. When operating at cache level, coherence is enforced by cache coherence mechanisms, thus, it is transparent to the programmer. However, when working at the main memory level, the programmer has to manually flush the operands from the caches before the NDP device can read them, and invalidate the memory region of the results in cache before accessing them again from the CPU (can be done using the cache management instructions provided by the CPU). One can argue that manually evicting data from cache deteriorates performance, which goes against the purpose of NDP. However, if the majority of the operands are stored in the main memory, evicting the few that are in cache to process them near the main memory is preferred. Ideally, computation should be performed wherever the majority of the operands are present. But that is a problem that falls under NDP design exploration, which is out of the scope of this paper.

III. Case Studies

To demonstrate the viability of gem5-ndp, the following subsections present and discuss two case studies: a simple near-data square root circuit, to exemplify the process of developing and integrating an NDP architecture using gem5-ndp; and a bidimensional NDP array, used in the experimental procedure.

A. Simple square root circuit

To ilustrate the development process of an NDP device and its integration with a standard processing system using gem5-ndp, a simple near-memory square root module based on the circuit proposed in [22] is considered. This circuit receives as input a 32-bit floating-point number and calculates its square root in two steps: First, the exponent of the output is calculated by shifting the exponent of the input to the right by one bit; Second, the mantissa of the output is calculated by interpolating

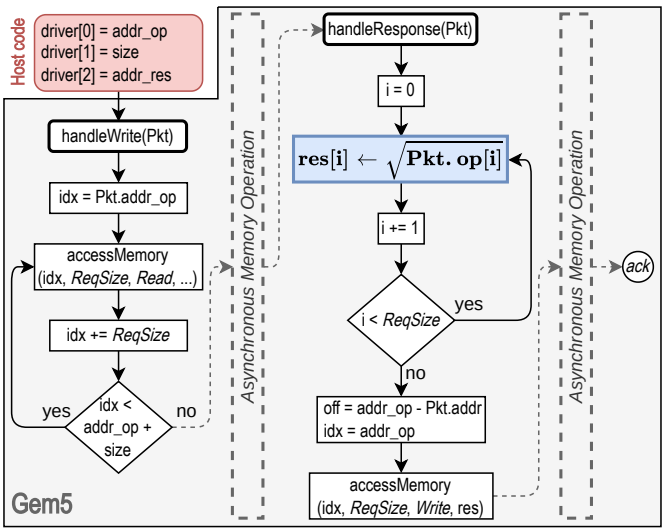


Fig. 5: Behavioral model of the simple near-data square root module described in terms of calls to the methods defined by the devised framework architectural model.

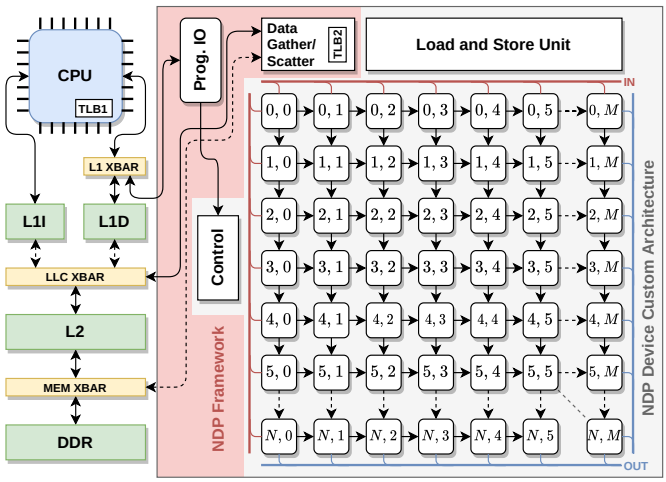


Fig. 6: NDP architecture considered in the validation and evaluation of gem5-ndp. The considered data path consists of a bidimensional array of similar Processing Elements (PEs).

the coefficients stored in one of two local memories, depending whether the exponent of the input is even or odd.

To simulate this module in gem5 using gem5-ndp, the first step consists of creating a new C++ class (an architectural model) that inherits from the class representing the gem5-ndp architectural model. As previously shown in Fig. 2, gem5-ndp requires the implementation of three main functions: `handleRead` and `handleWrite`, which deal with read and write operations from/to the NDP programming interface, and `handleResponse`, which deals with incoming packets from memory. To send requests to the memory device, the gem5-ndp architectural model implements the routine `accessMemory`, which receives a descriptor, an operation type (read or write), and the data to be written in case of a write operation.

Accordingly, the behavioral model of the simple near-data square root module is illustrated in Fig. 5. First, the `handleWrite` method is called (multiple times), to transmit the following data to the NDP device: base address of the operands, number of operands, base address to store the results. Then, the circuit will fetch batches of operands from memory, according to the availability of internal registers to store them, using the `accessMemory` method of the framework. The `handleResponse` method will be called as soon as the memory device returns a batch of operands to the NDP device, which schedules them to be processed sequentially and stores the results in an output buffer. After all operands of a batch have been processed, the results are sent back to memory. Finally, an acknowledgement is received by the NDP device through the `handleResponse` method indicating that the batch of results was successfully stored in memory.

Integrating the square root module with an existing processing system (at the L2 cache) is achieved with the steps illustrated in Listing 1, while the host code that makes use of the NDP device follows the structure described in Section II-C.

B. Bidimensional NDP Array on LLC

In this second case study, a bidimensional processing array was implemented and connected to the memory hierarchy of a processing system, as shown in Fig. 6. The considered NDP device was integrated with the CPU through the memory-mapped interface and connected to the memory hierarchy through the mechanisms provided by the architectural model of gem5-ndp. Although the devised architecture was designed to be integrated with any level of the memory hierarchy, for this particular scenario, it was decided to couple it with the Last Level Cache (LLC) to automatically maintain data coherence through the existing cache protocols.

The considered NDP device features a dedicated Load and Store Unit (LSU), responsible for fetching the input data and sending the results from/to the LLC through the data gather/scatter mechanisms provided by gem5-ndp. The control module implements all the logic that allows the CPU to manage the NDP device. It is also responsible for scheduling the NDP instructions to the PEs and keeping track of their status. The considered data path consists of a grid of similar PEs, capable of atomically processing a pair of vectors. PEs communicate with their neighbors through uni-directional buses: PE (i, j) communicates with PEs $(i+1, j)$ and $(i, j+1), i < N \wedge j < M$. Thus, the vector operands are loaded into the first row/column of PEs and propagated through the grid until reaching their destination. When available, the results are sent through the grid to the last row/column, and then collected by the LSU.

As shown in Fig. 7, it is assumed that each PE is equipped with a 64-bit integer arithmetic unit, a 64-bit floating-point arithmetic unit (both implementing addition, subtraction, and multiplication), and a 64-bit logic unit. Additionally, the arithmetic units can be configured to process a pair of 64-bit operands, two pairs of 32-bit operands, four pairs of 16-bit operands, or eight pairs of 8-bit operands. Each PE also features four 64-bit internal registers, with three of them

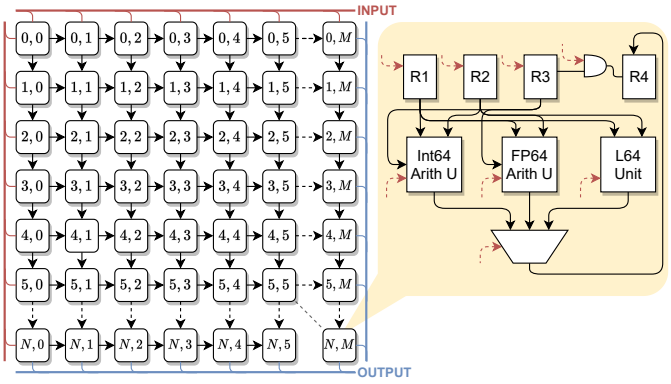


Fig. 7: Considered internal architecture of each PE of the proposed NDP device. Each PE includes Integer and Floating-Point Arithmetic Units and a Logic Unit.

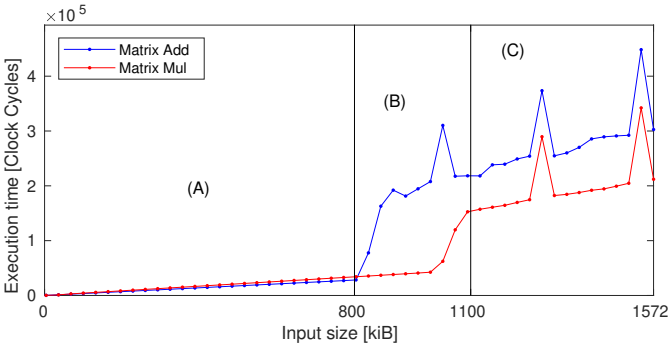


Fig. 8: Performance of the NDP device executing the Matrix Addition and Matrix Multiplication benchmarks for different matrix sizes (matrices that fit entirely in cache correspond to zone (A), and zones (B) and (C) involve matrices that do not fit entirely in cache).

serving exclusively to store operands, and an output register that can also store temporary results to be reused in subsequent operations. Since all PEs have the same internal configuration, a workload can theoretically be assigned to any cluster of PEs.

IV. Results and Discussion

To validate and evaluate the proposed framework, it was considered a processing system composed by a general-purpose CPU operating at 3.6 GHz, whose model in gem5 accurately describes a high-performance ARM core. The caches were modeled after the ARM Cortex-A72 specification, featuring a 32 kB L1 data cache and a 1 MB L2 shared cache, connected through 256-bit-wide crossbars. The main memory consists of a DDR3 module operating at 1600 MHz, with a 64-bit wide interface. For the majority of the experiments, the NDP device considered in the second case study (see Fig. 6) was coupled to the shared L2 cache. However, for the last experiment (Section IV-C) it was also connected to the DDR memory.

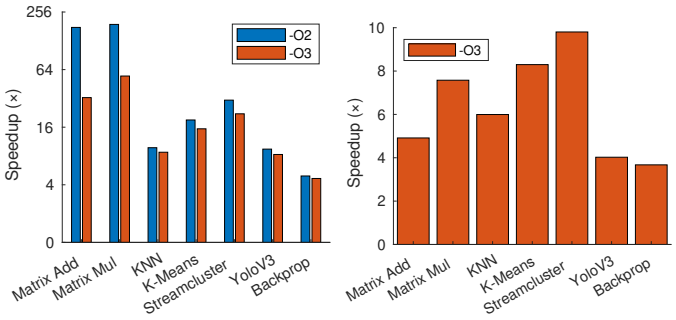
To evaluate the advantages of the proposed simulation framework, three sets of experiments were performed. First, two arithmetic benchmarks (Matrix Addition and Matrix Multiplication) were executed by the NDP device for multiple data sizes, allowing to observe its performance for different sets of data. Second, seven benchmarks from different application domains were executed by the processing system with and without the proposed NDP device connected to the L2 cache, in order to evaluate the attained performance benefits. Finally, the obtained results using gem5-ndp were compared with extrapolated performance values calculated using classical evaluation models (see Section IV-C).

A. Simple Arithmetic Benchmarks

The results of the first experiment are presented in Fig. 8, showing the performance of the NDP device while executing two regular arithmetic benchmarks: Matrix Addition and Matrix Multiplication. For this experiment each row of the matrices was 256 floating-point values long, and the second matrix of the Matrix Multiplication benchmark was transposed in memory. As it can be observed, the resulting performance is not proportional to the size of the data being processed. In particular, for matrices larger than 1 MiB (which do not entirely fit in the L2 cache), the performance of the NDP device depends on the DDR operation and cache eviction protocols (zones (A) and (B) of Fig 8). Furthermore, when executing the Matrix Addition benchmark simultaneously in the CPU and the NDP device using 4 KiB matrices, a performance decrease of 2× was observed, with the average bandwidth between the NDP and the LLC decreasing from 173 bit/cycle to 84 bit/cycle. This effect is explained by the data contention caused by the CPU and the NDP, since both require large amounts of data from the memory subsystem. This shows that the performance benefits of the NDP device can only be accurately predicted if the whole system is considered, including not only the NDP device but also the memory hierarchy (which may not have a constant throughput), and the CPU (which manages the NDP device and shares the same memory hierarchy, partially occupying its bandwidth). It is also noticeable that the Matrix Addition benchmark shows a significant performance degradation for smaller input matrices when compared with Matrix Multiplication. This is explained by the involved matrix sizes (not square), which impact the output matrices. In Matrix Addition, the output matrix size matches the input size filling up the cache sooner, as the L2 cache uses a write-back policy. In contrast, in Matrix Multiplication, the output matrix is smaller which allows using larger input matrices before triggering cache evictions that would otherwise penalize the performance. In zones (B) and (C) there is also the presence of local peaks of poor performance. These correspond to specific data sizes that, due to the cache eviction policy, generate more block evictions. Naturally, that leads to more requests being sent to the DDR, degrading performance.

B. Mixed Benchmark Evaluation

The set of benchmarks used in the second experiment consist of seven applications from three different domains, as described in Table I. For simplicity, single-precision floating-point format is used across all the benchmarks, and all the vector operands



(a) In-order CPU (b) OoO CPU

Fig. 9: Performance benefits provided by the NDP device for several benchmarks from different application domains.

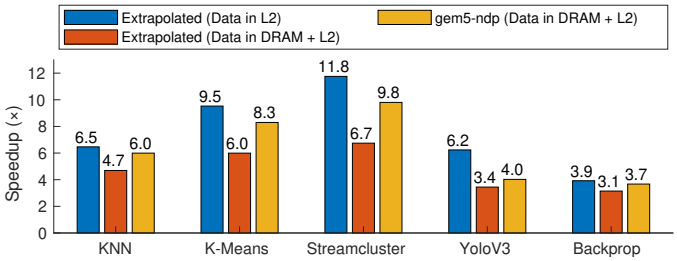


Fig. 10: Performance benefits provided by the bidimensional NDP array considering: (blue and orange bars) the CPU and the NDP device operating separately (extrapolated results); and (yellow bars) using gem5-ndp. In this experiment, the NDP device was connected to the L2 cache, the CPU and the NDP device operate simultaneously, and the data is present in both the DRAM and the L2 cache.

involved in each accelerated kernel invocation are $4\,\mathrm{KiB}$ each, with two operands being required for each kernel. In the particular case of Matrix Addition and Matrix Multiplication, each row of the matrices is the same size as a cache line ($64\,\mathrm{B}$ or 16 floating-point values), and the second matrix of the Matrix Multiplication benchmark is transposed in memory. Furthermore, three baselines are considered for calculating the performance benefits introduced by the NDP device: two using a high-performance in-order ARM CPU, with the benchmarks compiled with -O2 and -O3 optimization flags (Fig. 9a); and one considering an OoO ARM core, with the benchmarks compiled with -O3 (Fig. 9b). Despite their differences, all the seven benchmarks are highly parallelizable, with over $80\,\%$ of the workload being suitable to be executed in parallel. Due to its data-parallel compute capabilities and higher memory bandwidth (compared to the CPU), the NDP device is able to significantly accelerate these regions, with overall performance speedups ranging from 3.7× up to 189.9×, as shown in Fig. 9.

The performance improvements allowed by the NDP device mainly depend on two factors: the fraction of the workload that can be accelerated by the NDP device (the greater the accelerated workload, the higher the performance improvements–Amdahl's law); and the complexity of the kernel being executed by the NDP device (kernels that reuse the outputs of the PEs in subsequent operations tend to allow higher speedups). For example, the parallelizable region of Streamcluster accounts for over $97\,\%$ of its total workload, and its corresponding NDP kernel involves five distinct arithmetic operations over the same input data. On the other hand, Matrix Multiplication, despite being $100\,\%$ parallelizable, only requires a single arithmetic operation per pair of operands (and a final reduction). Thus, when considering the OoO CPU baseline and compilation with -O3 (which includes vectorization), the performance benefits attained by the NDP device for the Streamcluster benchmark are higher than those for Matrix Multiplication, since it can be efficiently executed by the CPU. As expected, the main advantage of NDP over traditional CPU-only processing is its higher bandwidth to the memory. Adding multiple cores may increase the CPU compute capabilities, but its limited memory bandwidth does not allow to fully exploit such capabilities.

C. Comparison with a Classical Evaluation Approach

The performance speedups obtained with gem5-ndp were compared with the corresponding results when considering the absence of an integrated simulator. In such case, to estimate the performance, one would consider the execution times of the CPU and the NDP device operating separately disregarding the access contention in the memory hierarchy. Hence, the Amdahl's law can be used by considering that the global speedup (S) can be obtained from the speedup of the workload accelerated by the NDP device (s), and the proportion of execution time that the accelerated workload originally occupied (A) on the CPU.

$$S = \frac{1}{(1 - A) + A/s}$$

Fig. 10 illustrates the performance benefits provided by the NDP device obtained through: (1) extrapolation, considering that the operands of the NDP kernel are present in the L2 cache; (2) extrapolation, considering that the operands of the NDP kernel are distributed across the DRAM and the L2 cache with a realistic L2 hit rate obtained through simulation; (3) gem5-ndp, with the operands being distributed across the DDR and the L2 cache, the NDP device coupled to the L2 cache, and both the CPU and the NDP operating simultaneously, which corresponds to an accurate simulation of the actual system. In

TABLE I: Set of seven benchmarks from three different application domains used for evaluation. The used implementations of K-Nearest Neighbors (KNN), K-Means, Streamcluster, and Backprop are from the Rodinia benchmark suite [23].

Benchmark	Domain	Approx. accel. kernel time [%]
Matrix Add.	Algebra	100 %
Matrix Mul.		100 %
K-Nearest Neighbors	Clustering	90 %
K-Means		95 %
Streamcluster		97 %
YoloV3 [24], [25]	Machine Learning	90 %
Backprop		80 %

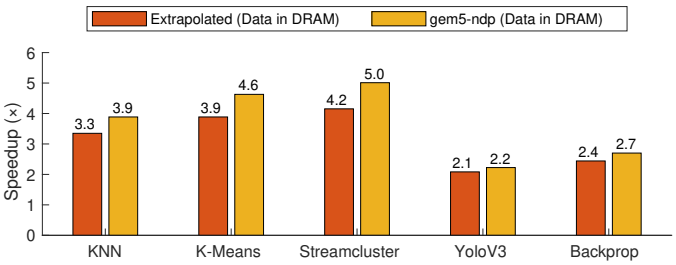


Fig. 11: Performance benefits provided by the bidimensional NDP array considering: (orange bars) the CPU and the NDP device operating separately (extrapolated results); and (yellow bars) using gem5-ndp. In this experiment, the NDP device was connected to the DRAM, the CPU and the NDP device operate simultaneously, and the data present in the DRAM.

addition, Fig. 11 shows a similar evaluation where the NDP device is coupled to the DDR memory.

As it can be observed, extrapolating the performance from the execution times of the CPU and the NDP alone is only useful to obtain the order of magnitude of the actual performance benefits, with errors ranging from $6.4\,\%$ to $54.9\,\%$. Even the results considering a realistic L2 hit rate incur in errors as high as $31.2\,\%$, which may be prohibitive when evaluating the viability of new NDP architectures. It is also noticeable that the errors are significantly lower when assuming the NDP device connected directly to the DDR memory comparing to when the NDP device is connected to the L2 memory. This effect can be explained by the fact that the latency of DDR accesses is much easier to predict than the average latency of the L2 cache, which is affected by the hit rate for a given data access pattern and the traffic generated by the CPU.

V. Related Work

Previous NDP proposals mostly resource to PIM [4] and RRAM-PIM [10] to move computation near to where the data is stored. However, these techniques present serious drawbacks: either they only implement very simple bit-wise operations, or they may introduce significant errors in the computation due to their analog nature. Therefore, most of such NDP proposals are only suitable for particular problems and applications. Naturally, due to their heterogeneity, each solution tends to use its very own evaluation method, which does not only make it difficult to establish fair comparisons between them, but also raises important questions regarding the adequacy of such procedures.

More recently, Lockerman *et al.* [26] proposed a general-purpose NDP system capable of performing computation at multiple levels of the memory hierarchy, reducing moderately both the execution time and the dynamic energy consumption when executing challenging irregular workloads, while keeping the area overhead below $3\,\%$. However, their system is based on existing NDP solutions and their specific ISAs, and it is not capable of accommodating different NDP devices. Moreover, their evaluation method involves an implementation on an Field-Programmable Gate Array (FPGA), which can hardly fit such a complex system featuring high-performance OoO CPUs and NDP-enabled memory devices, leading to the conclusion that only a part of the system was actually implemented. Therefore, their evaluation neglects (to some extent) the impact of the interaction between the different components of the system in the performance of the NDP devices, which may affect the accuracy of their results. In contrast, our simulation-based solution offers a more reliable approach, by using precise simulation models of each component to accurately predict the performance of the overall system. Moreover, our solution only requires the implementation of the simulation model of the NDP architecture being developed, which arguably requires a fraction of the time to describe the entire system in any Register-Transfer Level (RTL) language.

Following an approach closer to our proposal, Qureshi *et al.* [20] presented a set of extensions to enable in-cache processing in gem5-X. To achieve this, they proposed a modified L1 cache model that accommodates the computing architecture presented in [27]. Additionally, they validated their simulation model by implementing it using hardware synthesis tools. Although the attained results show that their artifact accurately predicts the operation of its hardware equivalent, the offered NDP capabilities are limited to a single architecture that performs computation in cache. Moreover, they only support one cache level with computation capabilities: the L1 data cache, which is the smallest cache and closest to the CPU. Furthermore, gem5-X is based on an obsolete version of gem5, being incompatible with most recent versions. In contrast, gem5-ndp supports NDP at all levels of the memory hierarchy, and can be ported to any version of gem5.

Another work targeting the simulation of NDP-enabled systems was proposed by Singh *et al.* [18], [19]. Their artifact is based on the ZSim architectural simulator [28] and Ramulator [18]. First, a given workload is executed by generating both performance statistics (with ZSim) and memory traces (with Ramulator). Then, a post-simulation step uses these intermediate results to determine a second set of performance statistics corresponding to a NDP-enabled system. Since the performance benefits provided by the existence of the NDP component of the system are only predicted in a post-simulation step, this solution does not support the parallel operation of the CPU and the NDP device. In contrast, gem5-ndp allows the CPU and NDP device to operate simultaneously, and does not require any post-simulation steps.

This considerable limitation in the work of Singh *et al.* was later solved by Yu *et al.* [29]. Similarly to the previous work, their proposal also uses traces generated by Ramulator to predict the performance benefits of NDP devices. The main difference relies on the support for the existence of multiple threads, which enables to simulate both the CPU and the NDP devices operating simultaneously. However, this artifact is still limited to DRAM-based PIM, which is the only memory technology supported by Ramulator. Moreover, the same ISA is used for the CPU and the NDP devices, which can arguably result in non-optimal NDP solutions. In contrast, gem5-ndp not only supports a wide range of memory devices, but it is also ISA independent, which confers a much ampler opportunity to

optimize the architecture of NDP devices, not only in terms of performance, but also in terms of resources and energy.

VI. Conclusions and Future Work

In this paper, a novel simulation framework targeting NDP devices is proposed. gem5-ndp aims at satisfying the need for a standard simulation tool to easily model and assess the performance benefits of new NDP architectures, without which developing such devices can be complex and time-consuming. Two case studies were presented to illustrate the process of developing and evaluating new NDP devices using gem5-ndp, as well as to show the benefits of using the framework compared to extrapolating performance gains from the execution time of the CPU and NDP devices.

The analysis of the obtained results allows to conclude that neglecting the influence of the surrounding processing system (i.e., considering only the CPU and NDP devices alone for evaluation purposes) may incur in significant errors when estimating the attained performance benefits. In our experiments, those errors were as high as $54.9\,\%$. On the other hand, by simulating the entire processing system, gem5-ndp eliminates those limitations, accurately predicting the real gains provided by NDP devices.

Presently, gem5-ndp targets gem5 System Emulation (SE) mode, with no guarantees regarding its compatibility with Full System (FS) simulation. Notwithstanding, FS support is already under development. Also, although gem5-ndp supports all CPU models provided by gem5 (In-order and OoO), implementation constraints dictate that it can only be used with the ARM ISA. Nevertheless, support for other ISAs is being developed. Furthermore, there is also some work in progress to extend the proposed framework to allow more complex descriptors to be passed to the data gather/scatter engine of the architectural model of gem5-ndp, which adds support to NDP devices targeting more complex workloads (e.g., operations with triangular matrices). In addition, we plan on making the framework openly available under permissive license in the near future.

Acknowledgements

Work supported by national funds through Fundação para a Ciência e a Tecnologia (FCT), under projects 2022.06780.PTDC, UIDB/50021/2020, and PTDC/EEI-HAC/30485/2017–HAnDLE (INESC-ID), UIDB/EEA/50008/2020 and EXPL/EEI-HAC/1511/2021 (Instituto de Telecomunicações), research grant SFRH/BD/144047/2019, and funds from the European Union Horizon 2020 Research and Innovation programme under grant agreement No. 101036168.

References

[1] W. A. Wulf, S. A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[2] S. Ghose, K. Hsieh, A. Boroumand, R. Ausavarungnirun,, O. Mutlu. Enabling the Adoption of Processing-in-Memory: Challenges, Mechanisms, Future Research Directions. *CoRR*, abs/1802.00320, 2018.

[3] J. Dinis Ferreira, G. Falcao, J. Gómez-Luna, M. Alser, L. Orosa, M. Sadrosadati, J. S. Kim, G. F. Oliveira, T. Shahroodi, A. Nori, et al. pluto: Enabling massively parallel computation in dram via lookup tables. *arXiv e-prints*, arXiv–2104, 2021.

[4] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. T. Blaauw,, R. Das. Compute Caches. In *HPCA*, 481–492. IEEE Computer Society, 2017.

[5] V. Seshadri, K. Hsieh, A. Boroumand, D. Lee, M. A. Kozuch, O. Mutlu, P. B. Gibbons,, T. C. Mowry. Fast bulk bitwise AND and OR in DRAM. *Computer Architecture Letters*, 14(2):127–131, 2015.

[6] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, D. Sylvester, D. T. Blaauw, R. Das,, R. R. Iyer. Neural cache: Bit-serial in-cache acceleration of deep neural networks. *IEEE Micro*, 39(3):11–19, 2019.

[7] S. Angizi, D. Fan. Redram: A reconfigurable processing-in-dram platform for accelerating bulk bit-wise operations. In *ICCAD*, 1–8. ACM, 2019.

[8] A. Subramaniyan, J. Wang, E. R. M. Balasubramanian, D. T. Blaauw, D. Sylvester,, R. Das. Cache automaton. In *MICRO*, 259–272. ACM, 2017.

[9] B. Li, P. Gu, Y. Shan, Y. Wang, Y. Chen,, H. Yang. Rram-based analog approximate computing. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 34(12):1905–1917, 2015.

[10] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams,, V. Srikumar. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. In *ISCA*, 14–26. IEEE Computer Society, 2016.

[11] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny,, U. C. Weiser. MAGIC - memristor-aided logic. *IEEE Trans. Circuits Syst. II Express Briefs*, 61-II(11):895–899, 2014.

[12] J. Vieira, E. Giacomin, Y. M. Qureshi, M. Zapater, X. Tang, S. Kvatinsky, D. Atienza,, P. Gaillardon. Accelerating inference on binary neural networks with digital RRAM processing. In *VLSI-SoC (Selected Papers)*, volume 586 of *IFIP Advances in Information and Communication Technology*, 257–278. Springer, 2019.

[13] P. Pouyan, E. Amat, S. Hamdioui,, A. Rubio. RRAM variability and its mitigation schemes. In *PATMOS*, 141–146. IEEE, 2016.

[14] J. Vieira, N. Roma, P. Tomás, P. Ienne,, G. F. P. Fernandes. Exploiting compute caches for memory bound vector operations. In *SBAC-PAD*, 197–200. IEEE, 2018.

[15] J. Vieira, N. Roma, G. Falcão,, P. Tomás. Processing convolutional neural networks on cache. In *ICASSP*, 1658–1662. IEEE, 2020.

[16] J. Vieira, N. Roma, G. Falcão,, P. Tomás. A compute cache system for signal processing applications. *J. Signal Process. Syst.*, 93(10):1173–1186, 2021.

[17] D. Fujiki, S. Mahlke,, R. Das. Duality cache for data parallel acceleration. In *Proceedings of the 46th International Symposium on Computer Architecture*, 397–410, 2019.

[18] Y. Kim, W. Yang,, O. Mutlu. Ramulator: A fast and extensible DRAM simulator. *IEEE Comput. Archit. Lett.*, 15(1):45–49, 2016.

[19] G. Singh, J. Gómez-Luna, G. Mariani, G. F. Oliveira, S. Corda, S. Stuijk, O. Mutlu,, H. Corporaal. NAPEL: near-memory computing application performance prediction via ensemble learning. In *DAC*, 27. ACM, 2019.

[20] Y. M. Qureshi, W. A. Simon, M. Zapater, D. Atienza,, K. Olcoz. Gem5-X: A Gem5-Based System Level Simulation Framework to Optimize Many-Core Platforms. In *SpringSim*, 1–12. IEEE, 2019.

[21] N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. S. B. Altaf, N. Vaish, M. D. Hill,, D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, 2011.

[22] H. C. Neto, M. P. Véstias. Very low resource table-based FPGA evaluation of elementary functions. In *ReConFig*, 1–6. IEEE, 2013.

[23] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee,, K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, 44–54. IEEE Computer Society, 2009.

[24] J. Redmon, S. K. Divvala, R. B. Girshick,, A. Farhadi. You only look once: Unified, real-time object detection. In *CVPR*, 779–788. IEEE Computer Society, 2016.

[25] J. Redmon, A. Farhadi. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767, 2018.

[26] E. Lockerman, A. Feldmann, M. Bakhshalipour, A. Stanescu, S. Gupta, D. Sánchez,, N. Beckmann. Livia: Data-centric computing throughout the memory hierarchy. In *ASPLOS*, 417–433. ACM, 2020.

[27] W. A. Simon, Y. M. Qureshi, A. Levisse, M. Zapater,, D. Atienza. BLADE: A bitline accelerator for devices on the edge. In *ACM Great Lakes Symposium on VLSI*, 207–212. ACM, 2019.

[28] D. Sánchez, C. Kozyrakis. Zsim: fast and accurate microarchitectural simulation of thousand-core systems. In *ISCA*, 475–486. ACM, 2013.

[29] C. Yu, S. Liu,, S. M. Khan. Multipim: A detailed and configurable multi-stack processing-in-memory simulator. *IEEE Comput. Archit. Lett.*, 20(1):54–57, 2021.