Journal of Signal Processing Systems (2021) 93:1173–1186
https://doi.org/10.1007/s11265-020-01626-y

A Compute Cache System for Signal Processing Applications

João Vieira[1] · Nuno Roma[1] · Gabriel Falcao[2] · Pedro Tomás[1]

Received: 13 July 2020 / Revised: 3 November 2020 / Accepted: 10 December 2020 / Published online: 12 April 2021

Abstract

Nowadays, processing systems are constrained by the low efficiency of their memory subsystems. Although memories evolved into faster and more efficient devices through the years, they were still unable to keep up with the computational power offered by processors, i.e., feed the processors with the data they require at the rhythm it is consumed. Consequently, with the advent of Big Data, the need for fetching large amounts of data from memory became the most prominent performance bottleneck. Naturally, several approaches seeking to mitigate this problem have arisen through the years, such as application-specific accelerators and Near Data Processing (NDP) solutions. However, none were capable to offer a satisfactory general-purpose solution without imposing rather limiting constraints. For instance, NDP solutions often require the programmer to have low-level knowledge of how data is physically stored in memory. In this paper, we propose an alternative mechanism that operates at the cache level, leveraging both proximity to the data and the parallelism enabled by accessing an entire cache line per cycle. We detail the internal architecture of the Cache Compute System (CCS) and demonstrate its integration with a conventional high-performance ARM Cortex-A53 Central Processing Unit (CPU). Furthermore, we assess the performance benefits of the novel CCS using an extensive set of microbenchmarks as well as six kernels widely used in the context of Convolutional Neural Networks (CNNs) and clustering algorithms. Results show that the CCS provides performance improvements ranging from 3.9× to 40.6× regarding the six tested kernels.

Keywords Near data processing · Cache · Single instruction multiple data · Memory-bound

1 Introduction

General-purpose processors have evolved to offer significant processing throughput by integrating multiple cores and efficient vector Functional Units (FUs). However, the memory subsystem is often incapable of providing enough bandwidth to the cores, which negatively affects the performance of the overall system. This widely-known phenomenon was thoroughly addressed by William et al. [1] and became known as the memory wall. Although its effects are horizontal to most algorithms and applications, this issue is most relevant in the context of data-centric applications, which are common in several domains, including Machine Learning (ML) and signal-processing. For example, in [2], the authors accelerated the k-Nearest Neighbors (kNN) algorithm by designing custom hardware accelerators and reducing drastically data movements. Their results show that their architecture can speed up the execution of the kNN algorithm almost two orders of magnitude, with proportional energy gains. In their analysis, the authors conclude that the significant reduction of data movements (and memory accesses) is the key that led to such advantages.

Moreover, due to the complexity of the memory hierarchies and the time spend on fetching data, a significant part of the spent energy is due to moving data across the memory subsystem. In their work, Aga et al. [3] show that, on average, 25% of the energy spent by a processing

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.

✉ João Vieira
joaomiguelvieira@tecnico.ulisboa.pt

Nuno Roma
nuno.roma@inesc-id.pt

Gabriel Falcao
gff@co.it.pt

Pedro Tomás
pedro.tomas@inesc-id.pt

[1] INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal

[2] Instituto de Telecomunicações, University of Coimbra, Coimbra, Portugal

system is associated with the data movements within the memory hierarchy, and between the memory subsystem and the processing cores (see Fig. 1a). For example, in [4], the authors report that the energy required by the Dynamic Random Access Memory (DRAM) is one order of magnitude superior to the energy spent by the processing cores while executing a Convolutional Neural Network (CNN) (see Fig. 1b).

Naturally, several alternative processing paradigms to circumvent the limitations imposed by the memory subsystems have arisen. One of the most common approaches is to analyze the data pattern of an algorithm and design a custom accelerator to be implemented in an Application Specific Integrated Circuit (ASIC) or a Field Programmable Gate Array (FPGA) that somehow enables more efficient memory management than that of general-purpose Central Processing Units (CPUs). However, such solutions are usually application-specific and are hardly usable for different workloads.

Another still prominent solution that was first proposed during the '80s is called Processing In Memory (PIM) [5]. Its core idea is to move computations closer to the memory by repurposing existing DRAM resources to enable active computation using a technique named bit-line computation. Due to the internal architecture of DRAMs, PIM solutions enable massive parallelism. However, they are only capable of executing simple logic operations atomically, requiring hundreds of cycles to execute more complex arithmetic calculations based on those atomic operations. Moreover, early PIM solutions had limited compilation support, requiring the programmer to know low-level details of the memory architecture, and explicitly write the code of the kernels to be executed in memory.

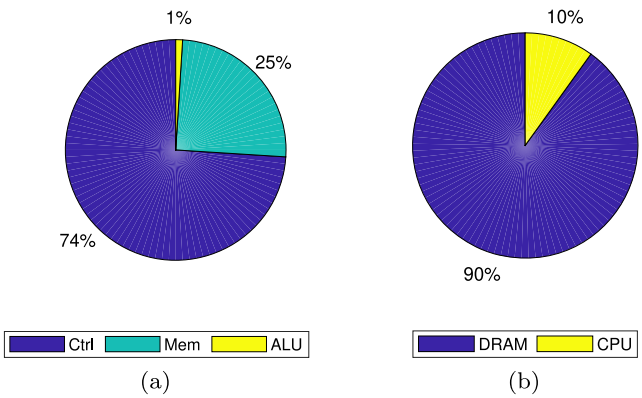

Figure 1 Average energy distribution: **a** general-purpose processing system [3] and **b** general-purpose processing system while executing a CNN [4].

In addition, operating data directly in the main memory may cause coherence issues since some of data may be cached. Therefore, the operands involved in the PIM operation have to be flushed from cache before computation and will remain inaccessible until the computation finishes.

Another factor that made the adoption of early PIM solutions difficult was the requirement for significant modifications to the architecture of memory chips, which generated resistance from the companies fabricating these devices.

Although PIM has had its ups and downs since the beginning, it ended up being an important alternative processing solution for memory-bound applications and gave origin to the Near Data Processing (NDP) paradigm [6]. The NDP paradigm is a super-set that includes all PIM solutions. However, unlike PIM, it is not limited to performing computation using exclusively the memories' internal resources and allows to use computing elements near (but outside) the memory devices. During the last decades, several NDP solutions have emerged and a lot has been learned. For instance, the complexity of modifying memory devices to accommodate computing structures has been mitigated by recent 3D-stacking techniques, which allow placing logic layers on top of memory cells connected by Through-Silicon-Vias (TSVs). An example of a device using this technique is the Hybrid Memory Cube (HMC). Moreover, the original principles of PIM have been transposed to different levels of the memory hierarchy, originating sub-paradigms such as in-/near-cache computing, and different related technologies, such as Resistive Random Access Memory (RRAM)-based processing [7].

In this work, the constrained performance of memory-bound ML and signal processing algorithms is revisited and new alternatives resourcing to NDP solutions are envisaged. The proposed approach consists of a Cache Compute System (CCS) based on the solutions presented in [8, 9] that couples a Single Instruction Multiple Data (SIMD) unit to the Last Level Cache (LLC) of a general-purpose CPU. Differently from previous proposals, this CCS architecture is based on robust fully-digital FUs, capable of implementing complex arithmetic, shift, and logic operations atomically. Furthermore, by being coupled to the LLC, it takes advantage of data locality and automatically deals with coherence issues that arise from doing computation directly in the main memory (updating data in the main memory may conflict with data present in cache). The presented research also includes a convenient library to program and control the CCS from plain C code.

All in all, the main contributions of this work are the following:

1. Detailed architecture of a CCS based on [8, 9], aiming at accelerating ML and signal processing algorithms using the NDP paradigm;
2. Cycle-accurate simulation model of the CCS for the gem5 architectural simulator;
3. Comprehensive library written in C to program, control and synchronize the CCS;
4. Thorough performance assessment of the CCS, including microbenchmarks and application benchmarks.

The rest of this paper is organized as follows. Section 2 summarizes the most relevant previous work in the field of NDP. Section 3 provides an overview of the proposed system, and Section 4 details the novel architecture. Section 5 explains the simulation and prototyping methodology and discusses the main experimental results. Finally, Section 6 concludes this paper.

2 Related Work

Several approaches have recently been considered for NDP, including PIM [10–17] and in-/near-cache [3, 18–21] processing solutions.

In such works, the authors take advantage of the memories' high bandwidth and parallelism to perform in-place computation, therefore reducing both execution time and energy consumption. As an example, Seshadri et al. [12] proposed a fast bulk-wise `AND` and `OR` mechanism to be implemented directly in the DRAM chip. The working principle of this mechanism lies in bit-line computing, where multiple lines of the memory array are simultaneously activated, and the sensed output is the result of a bit-wise operation. This provides the possibility to perform bit-wise `AND` and `OR` operations over the bits of an entire DRAM line.

Another example is Pinatubo, proposed by Shuangchen et al. [13], which is an in-memory processing scheme for emerging nonvolatile memories. Although the computation principle also relies on bit-line computation (supporting `AND`, `OR`, `XOR`, and `INV` operations), several functions and libraries are also supplied to take advantage of the computing capabilities. Pinatubo also attempts to bypass complex issues such as the operands being in different sub-arrays or banks. However, no solutions are proposed when the operands are in different higher levels of the memory architecture, such as different ranks. Furthermore, all of these solutions rely on bit-line computation, which limits their Instruction Set Architecture (ISA) to a few bit-wise operations. In contrast, the work presented in this paper offers a rich ISA currently composed of 47 instructions, allowing to perform a large set of operations.

Ahn et al. [22] introduced the concept of PIM-Enabled Instructions (PEIs). Their work relies on existing PIM-enabled architectures, therefore requiring no modifications to the compiler or the programming paradigm. They propose adding dedicated hardware structures to decide whether PIM instructions should be executed in memory or near the host processor (in specific hardware units), depending on the locality of the operands. To solve data dependencies, they rely on the existing cache coherence mechanisms (a strategy that is also adopted in this work).

Under a different approach, works such as [15, 16] leverage the processing capabilities of emerging RRAM technologies to accelerate the execution of CNNs. CNNs are a subclass of ML algorithms whose workload is over 90% composed of convolutions [23]. Therefore, accelerating the convolution operation through the analog computing capabilities of RRAM arrays results in significant performance enhancement and energy reduction. The working principle of analog RRAM-aided convolution relies on unrolling multi-dimensional convolutions in one-dimension operations (dot products). Then, half of the operands are encoded in the memristors as impedance values, and the other half is encoded as voltage levels that are imposed on the word lines. The impedance values encoded in the memristors weight the voltages in the word lines, resulting in an output current that represents the analog dot product of two vectors, which is then read by Analog to Digital Converters (ADCs). Although RRAM-based processing enables massive advantages in terms of performance and energy efficiency, it also presents serious drawbacks: (1) it requires expensive ADCs that do not only significantly increase the chip area but also augment the energy requirements [24]; (2) the error introduced during the computation is substantial, mostly due to process variations [25] and temperature fluctuations [26], which severely compromises the reliability of the results.

Aga et al. [3] propose a cache computing architecture where the cache resources are re-purposed to perform active computation. Unlike PIM solutions, where the computation takes place in the main memory, this solution takes advantage of the locality of the operands, present in many data-centric applications. This architecture provides two different computing mechanisms: one in-cache that can only be used when the operands share the same cache line, and another near-cache. It supports ten operations in total and provides the programmer with explicit functions and libraries to make use of the compute cache mechanisms. The computation can occur in any of the cache levels.

Although representing important contributions, these previously proposed architectures show significant performance degradation when the operands are misaligned or on

different levels of the memory hierarchy. Furthermore, their analog computing approach may also introduce significant errors that compromise the results. In contrast, the proposed solution minimizes the impact of the operands being split across different lines in the cache or even when the cache alignment between operands is different. This is achieved by loading them into an input buffer and selecting only the valid items from the cache line. Also, since the proposed architecture is fully digital, it delivers robustness and is less error-prone.

Also following this paradigm, Subramaniyan et al. [18] suggest exploiting the last level caches to perform active computation optimized for the Non-deterministic Finite Automata (NFA) computational model. Differently from traditional PIM approaches, including Micron's DRAM-based automata processor [27], the Cache Automaton relies on cutting edge Static Random Access Memory (SRAM) cache technology, which provides faster and more energy-efficient memories. The switch architecture, responsible for coordinating the access to the common buses, is redesigned to support the automata processing features, as well as the line multiplexing for sense-amplifying. Compiler support is also provided, which automates the process of mapping real-world NFAs to the Cache Automaton. Although this architecture provides an astonishing peak speedup of 3840× over a conventional x86 CPU, it is application-specific and cannot be used for other purposes.

Despite all the efforts, there are still two important issues that were not addressed by any of the previous works: (1) as these solutions are mostly application-specific, they provide small ISAs mostly limited to bit-wise operations, which limits the range of applications for which they are effective; (2) the performances of these architectures are severely degraded when the operands are in different levels of the memory hierarchy. In contrast, the proposed engine, based on the works devised in [8, 9], makes use of a conventional arithmetic and logic vector unit supporting a much wider range of operations than conventional NDP solutions while enabling massive parallelism by processing up to an entire cache row per cycle. Furthermore, the proposed CCS is fully digital, which represents an advantage over error-prone analog RRAM-aided processing. In [8], a proof-of-concept of the proposed architecture was fully implemented and validated in Hardware Description Language (HDL) using a simple soft-core and a minimalist (non-virtual) memory hierarchy. The work presented in [9] re-purposes the previous architecture, extending it to support operations commonly used by CNNs. In this work, the architecture of the proposed CCS is explained in much greater detail, a complete flow to integrate it with a general-purpose CPU is presented (including the details of how it is coupled to the remainder system), an optimized software framework to control the CCS is proposed, and a set of benchmarks is explored to evaluate its performance benefits.

3 System Overview

From the system point of view, the CCS consists of a device that is tightly connected to the memory hierarchy of a conventional processing system. As shown in Fig. 2, the CCS is connected to the processing system at two levels: (1) at the processor level, through which the processor communicates with the CCS by reading and writing the internal registers of the CCS' Programming Interface (PI); (2) at the LLC level, using the memory bus of the shared LLC, from where the CCS issues data requests and outputs the commands' results. The following subsections detail the internal architecture of the proposed solution as well as how it operates.

3.1 Near-Cache Compute Structure

The proposed CCS is composed of several independent FUs that are managed by a common control unit, which allows operating over data directly fetched from the cache using a SIMD paradigm. The CCS is meant to be integrated with general-purpose (possibly multi-core) processors and their respective memory subsystems. Therefore, it includes all the necessary mechanisms to support the complexity of such systems: (1) from the processor side, the CCS receives commands from (any of) the processing core(s), indicating the addresses of the input and output data, the size of the operands, and the operation to be performed; (2) given the existence of a virtual memory subsystem, the CCS translates the virtual addresses that were provided by the CPU using its own independent Translation Lookaside

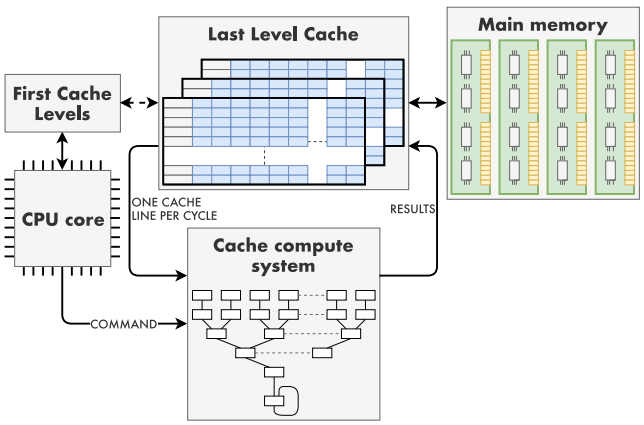

Figure 2 Block diagram of the proposed system integrating a processing core with the CCS.

Buffer (TLB), which is synchronized with the processor's TLB; (3) the commands are executed by the order of arrival, and the CCS offers synchronization mechanisms to ensure that two sequential commands do not conflict; (4) the CCS gathers the data required by the executing command by issuing read requests to the cache, receiving an entire cache line atomically; (5) after executing the command, the CCS writes the results directly into the cache, one line at a time.

After receiving a command for execution, the CCS sends a data read request to the cache. Since most systems feature inclusive caches, it is guaranteed that the retrieved data is up-to-date. If the requested data is present in the cache, it is retrieved immediately. Otherwise, the request is forwarded to the main memory. Naturally, while not being able to receive the data, the CCS enters an idle state, which (depending on the implementation) can be either executing no instruction or clock gating, disabling the CCS while it is not in use. For operations requiring two vector operands, the CCS fetches the operands sequentially. After gathering the first operand, the CCS stores it in a buffer and waits for the second operand. As soon as all the required data is available, the CCS starts executing the command.

Depending on the operation type, the execution can take from one to several clock cycles to complete. However, since the CCS is fully pipelined, it becomes available for receiving the next command right after all the previous operands were fetched and the previous command started executing. Figure 3 depicts two execution examples of commands that require different numbers of clock cycles to complete ((Rectified Linear Unit (ReLU) and dot-product), as well as a top-level schematics of the CCS tree-shaped internal architecture composed of several levels of FUs

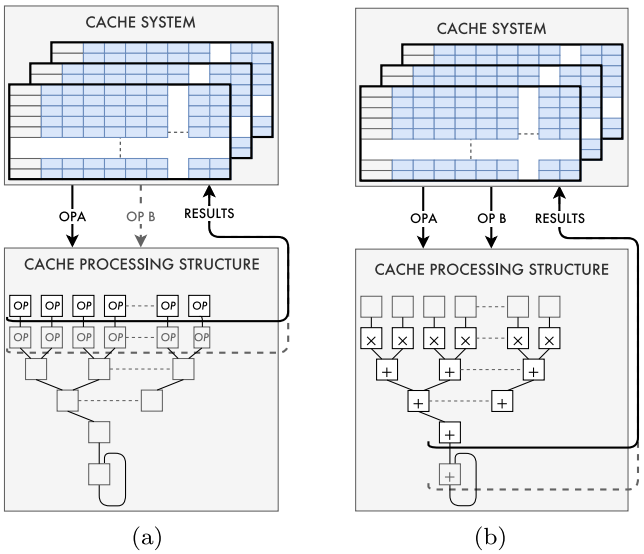

Figure 3 Execution of two different instructions in the CCS: **a** ReLU; **b** dot-product.

(which is discussed in detail in Section 4). The first command (a) performs an element-wise operation over all the elements of the vector operand, taking at most two clock cycles to complete. On the other hand, the second command (b) combines all the elements of the operands into a single one and takes a number of cycles at least equal to the number of levels of the tree-shaped CCS. When the execution terminates, the results are stored again in the LLC, invalidating any outdated data in the upper cache levels, and the processor is notified. If the operation executed by the CCS produces a vector as result (which is at most the size of a cache line), the elements are written back to the cache in parallel. For the currently supported instructions, the size of the output vector is the same as the operands'.

The proposed CCS essentially presents four advantages when compared to existing scalar or vector processors: (1) it makes extensive use of data locality, which is particularly advantageous in data-centric applications with regular memory access patterns; (2) since data is not moved all the way from the cache to the processing cores, less energy is spent on data transfers; (3) by exploiting the high bandwidth that is available near the cache, it is possible to achieve high-performance levels resulting from vectorization, by operating in up to an entire cache line in parallel; (4) due to the proposed CCS structure, FUs at different levels can be programmed to perform different operations, making it possible to execute complex commands.

3.2 Concurrent Operation

From the point of view of the host CPU, the CCS behaves like an independent co-processor that fetches, operates, and writes data in parallel without run-time intervention of the CPU. The only interactions between the CCS and the CPU happen when programming the CCS and inquiring its state (to determine whether the execution has finished and the results are available). Naturally, while the CCS is operating, the processor is free to execute a different workload, as long as there are no dependencies with the data being computed by the CCS. From a programming perspective, these dependencies are left to the programmer to resolve. Before reading or writing data that conflicts with the results calculated by the CCS, the programmer has to ensure that those results are available for being used. Otherwise, write-after-read and write-after-write conflicts may take place. Luckily, standard synchronization mechanisms can be used to work around this issue, such as mutexes, semaphores, and barriers.

Since the CCS operates at cache level, all the inherent coherence mechanisms are used. For instance, when writing the results to the cache, any outdated data in upper cache levels is automatically invalidated. Therefore, as long as the

programmer guarantees correct synchronization between the commands running in the CCS and the workload being executed in the CPU, no other data hazards are possible.

3.3 Integration with Hierarchical Caches

On multi-level cache systems, the CCS may be integrated at any level of the cache hierarchy or even at multiple levels. For instance, several instances of the CCS can be coupled to each level of cache, maximizing the attained parallelism. On the other hand, a single instance of the CCS can be used to minimize the hardware and power overhead. In this respect, integrating the CCS with the LLC seems to be the most interesting option since: (1) it provides the maximum proximity with the data that resides in the main memory (allowing to reduce the traffic across the upper memory hierarchy); (2) usually, the LLC is the largest level of cache, which positively affects the number of expected cache hits; (3) since the LLC is usually shared by the several cores, all the issues that could arise from writing to the cache are automatically solved by the existing coherence protocol, which ensures that upper cache levels are updated. Moreover, operating at the LLC level also reduces contention with the processing cores, which is useful for task-level parallelism.

4 The CCS Compute Architecture

The internal architecture of the proposed CCS is composed of several FUs, each one implementing a set of atomic arithmetic, shift, and logic operations. These FUs are micro-programmed by a central control unit, allowing to execute complex commands using multiple FUs in a SIMD manner.

4.1 Programming Interface

To instruct the CCS to execute a command, the processor programs the CCS by writing in the memory-mapped registers of its PI. Each command contains information about the operands to be processed, the computation to be performed, and the destination in memory of the result. The CCS supports 47 distinct commands (which are summarized in Table 1) that can be classified according to three different criteria:

1. The class of mathematical operations: integer arithmetic, shift, or logic;
2. The type of the operands: two vectors (VOP2), a vector and a constant (VCOP), or a single vector (VOP1);
3. The class of functional operation: *map* or *reduce*.

Table 1 CCS' currently supported commands classified regarding their class of mathematical operations, type of operands, and class of functional operation.

		Instr	Description	
Arithmetic	VOP2	ADDVV	$r[i] = a[i] + b[i]$	map
		SUBVV	$r[i] = a[i] - b[i]$	
		MULVV	$r[i] = a[i] \times b[i]$	
		SSDVV	$r = \sum_i (a[i] - b[i])^2$	reduce
		SADVV	$r = \sum_i \lvert a[i] - b[i] \rvert$	
		IPVV	$r = \sum_i a[i] \times b[i]$	
	VCOP	ADDVC	$r[i] = a[i] + k$	map
		SUBVC	$r[i] = a[i] - k$	
		MULVC	$r[i] = a[i] \times k$	
	VOP1	COMP2V	$r[i] = -a[i]$	
		ADDV	$r[i] = \sum_i a[i]$	reduce
		MAXV	$r[i] = \max(a)$	
		MINV	$r[i] = \min(a)$	
		LESSVC	$r[i] = \begin{cases} 1 & , a[i] < k \\ 0 & , otherwise \end{cases}$	map
		GRTRVC	$r[i] = \begin{cases} 1 & , a[i] > k \\ 0 & , otherwise \end{cases}$	
		EQUVC	$r[i] = \begin{cases} 1 & , a[i] = k \\ 0 & , otherwise \end{cases}$	
		SQV	$r[i] = a[i]^2$	
		ABSV	$r[i] = \lvert a[i] \rvert$	
		RELUV	$r[i] = \begin{cases} a[i] & , a[i] > 0 \\ 0 & , otherwise \end{cases}$	
Shift	VOP2	SLLVV	$r[i] = \text{sll}(a[i], b[i])$	
		SRLVV	$r[i] = \text{sll}(a[i], b[i])$	
		SLAVV	$r[i] = \text{sla}(a[i], b[i])$	
		SRAVV	$r[i] = \text{sra}(a[i], b[i])$	
		ROLVV	$r[i] = \text{rol}(a[i], b[i])$	
		RORVV	$r[i] = \text{ror}(a[i], b[i])$	
	VCOP	SLLVC	$r[i] = \text{sll}(a[i], k)$	
		SRLVC	$r[i] = \text{srl}(a[i], k)$	
		SLAVC	$r[i] = \text{sla}(a[i], k)$	
		SRAVC	$r[i] = \text{sra}(a[i], k)$	
		ROLVC	$r[i] = \text{rol}(a[i], k)$	
		RORVC	$r[i] = \text{ror}(a[i], k)$	
Logic	VOP2	ANDVV	$r[i] = a[i]$ and $b[i]$	
		NANDVV	$r[i] = a[i]$ nand $b[i]$	
		ORVV	$r[i] = a[i]$ or $b[i]$	
		NORVV	$r[i] = a[i]$ nor $b[i]$	
		XORVV	$r[i] = a[i]$ xor $b[i]$	
		XNORVV	$r[i] = a[i]$ xnor $b[i]$	
	VCOP	ANDVC	$r[i] = a[i]$ and k	
		NANDVC	$r[i] = a[i]$ nand k	
		ORVC	$r[i] = a[i]$ or k	
		NORVC	$r[i] = a[i]$ nor k	
		XORVC	$r[i] = a[i]$ xor k	
		XNORVC	$r[i] = a[i]$ xnor k	
	VCOP	NOTV	$r[i] = \text{not } a[i]$	
		ANDV	$r[i] = a[0]$ and $\cdots$ and $a[B-1]$	reduce
		ORV	$r[i] = a[0]$ or $\cdots$ or $a[B-1]$	
		XORV	$r[i] = a[0]$ xor $\cdots$ xor $a[B-1]$	

4.2 Compute Infrastructure

The CCS is composed of several levels of quasi-generic FUs arranged in a pipelined binary tree, as shown in Fig. 4. To ensure the aimed performance and energy gains, some architectural aspects have been considered concerning the integration of the CCS with the conventional memory subsystem of a processing system and the CCS' internal compute structure.

4.2.1 Integration with the Data Cache

Differently from most general-purpose processors, the CCS does not read/write a single word from/to the cache. Instead, it fetches/stores an entire cache line atomically. For the VOP2 commands, the CCS sequentially fetches the two vector operands using the same input bus. First, one of the operands is fetched from the cache and stored in a buffer. Then, the other operand is fetched, fulfilling all the requirements for starting execution.

Whenever the operands are not in cache, they are automatically fetched from the main memory to the cache, similarly to what would happen if they were requested by a general-purpose processor. Since the CCS is coupled to the LLC, the largest cache level, loading the vector operands from the main memory does not necessarily increase the number of collisions between the data being used by the processing cores and those used by the CCS. In fact, most LLCs are not only larger but also have higher associativity than upper cache levels. Thus, the data being evicted due to loading the operands required by the CCS will be (most probably) the least used, minimizing the negative effects on the processor's performance.

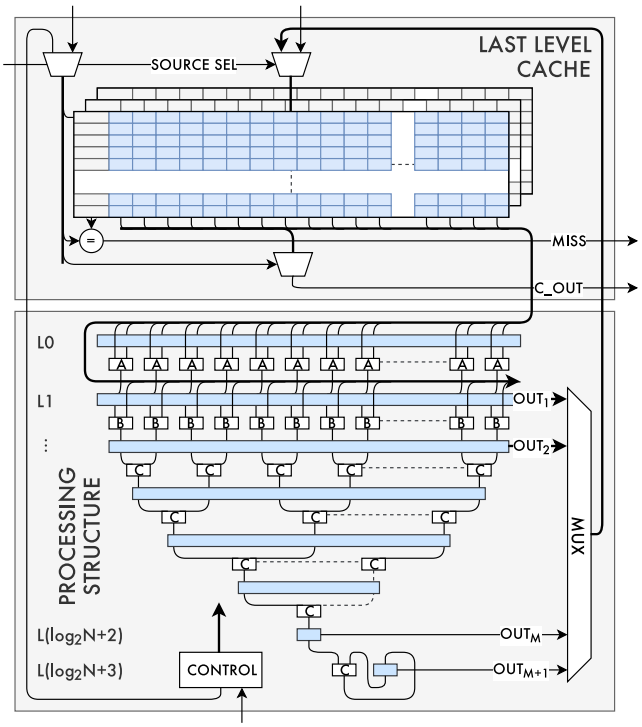

Figure 4 Processing structure of the CCS.

4.2.2 Pipeline Functional Units

To maximize the CCS' throughput, the processing structure integrates a pipelined binary tree, with each level implementing different arithmetic, shift, and logic operations, depending on the command being executed.

The operands (fetched from the cache) are delivered to the first level FUs and the computation starts as soon as all the operands are available (see Fig. 4).

The pipelined CCS datapath is composed of three different types of FUs, as shown in Fig. 5. The first level is composed by type A FUs (see Fig. 5a), integrating an adder/subtracter, a shifter and a logic unit. The first level of the CCS is responsible for several element-by-element operations, including: `ADD`, `SUB`, `COMP2`, `SLL`, `SRL`, `SRA`, `ROL`, `ROR`, `AND`, `NAND`, `OR`, `NOR`, `XOR`, `XNOR`, `NOT`, and element-wise comparison.

The second level of the CCS is composed by type B FUs (see Fig. 5b), being capable of computing the following operations: `MUL`, `SQ`, and `ABS`. Type B units are the only that include an integer multiplier, since many commands supported by the proposed CCS require, at most, one multiplication. The multiplier was included in the second (rather than the first) level because commands such as `SSDVV` require one operation before multiplication. Other common operations that require multiplication (e.g. `IPVV`) do not require any operation before multiplication (thus, skipping the entire first level of the CCS), but require a reduction step afterward, which is compatible with the used approach.

All the other pipeline stages of the CCS are only used by commands of type *reduce*. These computing stages are equipped with type C FUs (see Fig. 5c), which only implement arithmetic (except multiplication) and logic instructions: `ADD`, `MAX`, `MIN`, `AND`, `OR`, and `XOR`. For these units, only an adder/subtracter and a logical unit are required.

The operation at each level is controlled by control signals that are decoded by the CCS from the command issued by the processor.

4.2.3 Pipeline Dataflow

There are four computing levels of the CCS that produce output. The command being executed determines which of the four outputs is valid and when. The first level produces the output for all operations of type *map* that do not involve multiplication (which is only implemented in

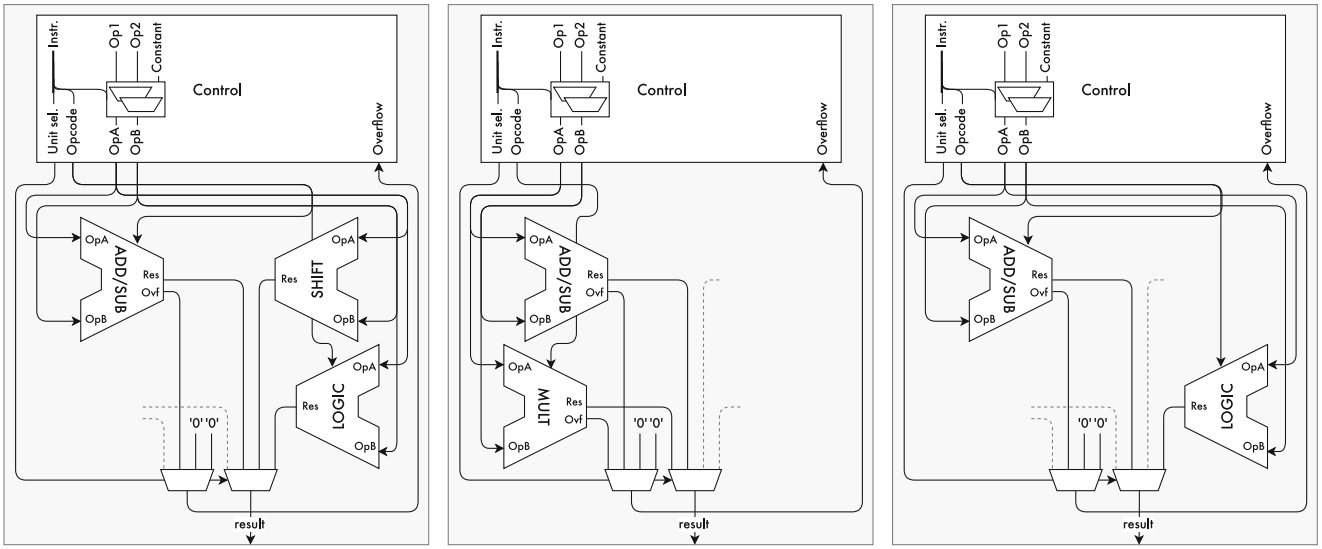

Figure 5 FUs that compose the proposed CCS: **a** first level units, integrating an adder/subtracter, a shifter and a logical unit, **b** second level units containing an adder/subtracter and a multiplier, and **c** remaining FUs, including only an adder/subtracter and a logical unit.

the second level). Therefore, the output of *map* operations involving integer multiplication may only be collected in the second level. The output of operations of type *reduce* can be produced in two levels of the CCS: the last and the second to last. If the vector operands involved in the *reduce* operation fit within a line of cache (i.e., the operands are entirely processed in a single run of the CCS), then the output is collected in the second to last level. Otherwise, the operands cannot be processed in a single run of the CCS, since there are not enough resources to process all the elements simultaneously (the operands are too large). In that case, the command is automatically split into multiple pipelined operations, and the last level of the CCS is used to accumulate the sub-results of each operation until all elements are processed.

4.3 Programming the CCS

The programming of the CCS by the processor is done using the provided PI. This interface is composed of memory-mapped registers through which the CPU specifies the command to be executed, an optional constant, and the descriptors for the operands and the result. Additionally, there is a memory-mapped register in the CCS' PI through which the processor instructs the CCS to start executing the command. As soon as all the operands required by the command are fetched to the CCS, the computation starts. When it finishes, the results are written back to the cache and the processor is informed that the results are available through another register in the CCS' PI.

To facilitate the process of writing the parameters into the programming registers and controlling the CCS, a fully optimized library is provided that exports a set of routines that can be straightforwardly used by the programmer. The library has the following four routines:

1. `ccs_setup(<args>)` writes the following parameters (`<args>`) of a command to the CCS' PI:
   - `cmd_id`: command identifier;
   - `op_len`: length of the vector operands;
   - `k`: optional constant;
   - `opa_addr`: start address of the first vector operand;
   - `opb_addr`: start address of the second vector operand;
   - `res_addr`: start address of the result;
   - `stride`: stride of the operands (and the result for applications of type *map*).
2. `ccs_start()` instructs the CCS to start executing the command previously configured in its PI;
3. `ccs_check()` verifies if the previous command has finished and the CCS is ready to receive a new command;
4. `ccs_wait()` hangs until the previous command has finished and the CCS becomes ready to receive a new command.

Listing 1 shows the code of a two-dimensional convolution using the library provided with the CCS.

```
#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 Implementation of a 2D convolution using the provided CCS library

All the developed routines were tuned at assembly level to ensure that the maximum optimization is achieved. For example, Listing 2 illustrates the implementation of these low-level routines using ARM assembly.

5 System Evaluation

To assess the performance benefits of the proposed CCS, a gem5-based simulation environment was developed. The following subsections explain the evaluation methodology and discuss the main experimental results.

5.1 Methodology

An accurate model of the proposed CCS was developed and integrated with the gem5 model of an in-order ARM CPU equipped with a NEON SIMD unit. The CPU model was further improved with the parameters found in the gem5-X [28] framework to produce reliable results regarding the ARM Cortex-A53 CPU. The CCS was connected to the CPU through its data port and to the LLC (in this case the L2 cache) using the existing memory bus. Since the architecture of the CCS is highly pipelined (and additional pipeline stages can still be added if necessary to further reduce the critical path without affecting performance), it was considered that the CCS can operate at the same frequency of the CPU core.

An additional module was also used to forward the memory requests from the CPU to the memory subsystem or between the CPU and the CCS' PI, depending on the target address. If the target address of the request is reserved to the CCS' PI, the module forwards the request to the CCS. Otherwise, the request is forwarded to the memory subsystem. The translation of the virtual addresses of the operands into their correspondent physical addresses (at the cache) is ensured by a dedicated TLB that is synchronized with the processor's TLB, as explained in Section 3.

```
void ccs_setup (uint64_t cmd_id,
  uint64_t op_len,
  uint64_t k,
  uint64_t opa_addr,
  uint64_t opb_addr,
  uint64_t res_addr,
  uint64_t stride) {
  // store parameters in the programming registers
  __asm( "str %[cmd_id],    [%[addr_cmd_id]]\n\t"
         "str %[op_len],    [%[addr_op_len]]\n\t"
         "str %[k],         [%[addr_k]]\n\t"
         "str %[opa_addr],  [%[addr_opa_addr]]\n\t"
         "str %[opb_addr],  [%[addr_opb_addr]]\n\t"
         "str %[res_addr],  [%[addr_res_addr]]\n\t"
         "str %[stride],    [%[addr_stride]]\n\t"
   : : [cmd_id]        "r" (cmd_id),
       [addr_cmd_id]   "r" (CCS_CMD_ID),
       [op_len]        "r" (op_len),
       [addr_op_len]   "r" (CCS_OP_LEN),
       [k]             "r" (k),
       [addr_k]        "r" (CCS_K),
       [opa_addr]      "r" (opa_addr),
       [addr_opa_addr] "r" (CCS_OPA_ADDR),
       [opb_addr]      "r" (opb_addr),
       [addr_opb_addr] "r" (CCS_OPB_ADDR),
       [res_addr]      "r" (res_addr),
       [addr_res_addr] "r" (CCS_RES_ADDR),
       [stride]        "r" (stride),
       [addr_stride]   "r" (CCS_STRIDE)
   );
}

void ccs_start () {
  const uint64_t start = 0x1;

  __asm( "str %[start], [%[addr_start]]\n\t"
   : : [start]      "r" (start),
       [addr_start] "r" (CCS_START)
   );
}

uint64_t ccs_check () {
  volatile uint64_t ready;

  __asm( "ldr %[ready], [%[readiness_addr]]\n\t"
    : : [ready]          "r" (ready),
        [readiness_addr] "r" (CCS_READINESS)
    );

  return ready;
}

void ccs_wait () {
  return;
  while (check() != 0x1);
}
```

Listing 2 Low-level routines written in ARM assembly to program and control the CCS from plain C code

The conceived CCS model was designed to target fixed-point vector operations. Since the size of a cache line in the reference CPU is 64 bytes wide, the developed model of the CCS can process up to 64 bytes of data per clock cycle. In this evaluation, the CCS was configured as a vector FU capable of processing up to 16 words of 32 bit each, 32 16-bit words or 64 8-bit words.

5.2 Experimental Results and Discussion

To assess the performance of the proposed CCS, two different evaluation procedures were considered. First, a set

of microbenchmarks was executed, each corresponding to a single CCS command involving vector operands of the size of a cache line, and the results were compared with those obtained using the CPU core. Second, six kernels commonly used by CNNs and clustering algorithms were evaluated. Five of the kernels are commonly used by CNNs. The sixth kernel consists of the distance computation phase of the kNN clustering algorithm, which is also common to other algorithms (e.g., *k*-Means). The tested kernels as well as the used parameters are presented in Table 2.

The CPU baseline was fully optimized using all the compiler optimizations, including vectorization support. Additionally, it was considered that the operands were stored in the memory hierarchy level closer to where the computation took place (L1 cache when using the CPU and LLC when using the CCS).

Figure 6 shows the results obtained for each microbenchmark.

As expected, commands of type *map* take fewer cycles to complete than commands of type *reduce*, and, overall, the CCS shows significant performance benefits across all the microbenchmarks. The microbenchmark associated with the highest performance improvement was the `ROLVV` command, which requires several operations when executed by the processor's Arithmetic and Logic Unit (ALU), but a single one when implemented by the CCS.

Also, Fig. 7 shows the variation of the performance gains with the size of the operands for a command of type *map* (`ADDVV`) and a command of type reduce (`ADDV`). As expected, the performance gains increase with the size of the operands due to the pipeline architecture of the CCS. When the vector operands are too big, requiring a command to be split in several runs, the CCS issues the requests for the operands in sequence, one per cycle, and starts performing the computation as soon as the operands of the first run arrive. Since the requests for the operands of the second run were issued right after the requests for the operands of the first run, the second run can be executed immediately after the first, in a pipelined manner. Therefore, the executing time of the two runs is much lower than twice the execution

Table 2 Parameters of the kernels used to assess the benefits offered by the CCS.

		Parameter	Value	CCS/GPU Iterations
Conv	1D	Data size *(length)*	1000 *(1000)*	1000
		Kernel size	15	
	2D	Data size *(length)*	{100, 100} *(10000)*	10000
		Kernel size	{3, 3}	
	3D	Data size *(length)*	{10, 10, 10} *(1000)*	1000
		Kernel size	{3, 3, 3}	
Max Pool		Data size *(length)*	{99, 99} *(9801)*	1089
		Patch size	{3, 3}	
		Stride size	{3, 3}	
ReLU		Data size *(length)*	{100, 100} *(10000)*	10000
kNN		*k*	4	1000
		Distance	`SSDVV`	
		Training	1000	
		Testing	1	
		Features	16	
		Classes	8	

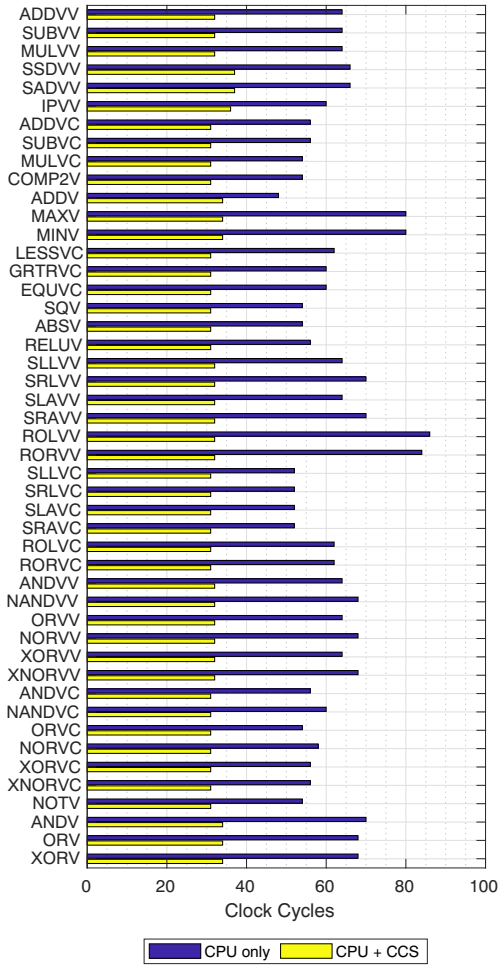

Figure 6 Performance results of a set of microbenchmarks, each one corresponding to a CCS command, when executed using only the reference CPU or the CCS.

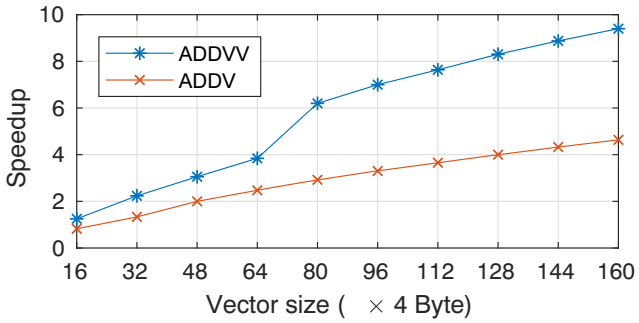

Figure 7 Variation of performance gains with the size of the operands for a command of type *map* (ADDVV) and a command of type reduce (ADDV).

time of a single run. On the other hand, processing a vector operand twice as big in the CPU takes approximately twice as many cycles.

Figures 8 and 9 depict the execution time associated with the processing of the six kernels using vector operands with 32-bit elements. For visualization purposes, the results are normalized to the number of cycles required to process 64 bytes of data. As expected, the CCS allows for significant performance benefits on all tested kernels. In particular, the ReLU kernel presents the highest speedup of 40.6× (for vectors of 8-bit elements, as shown in Fig. 10) due to its greater complexity when executed on a general-purpose CPU, contrasting with the single execution cycle required in the CCS. It is also worth mentioning that, although processing times are still dominated by memory accesses, the time spent by the CCS accessing memory is significantly lower than that of the CPU. Therefore, the performance benefits associated with the CCS have two complementary sources: (1) fewer memory accesses when compared to a conventional general-purpose CPU; and (2) more efficient implementation of complex operations that may otherwise take tens of cycles on a conventional general-purpose CPU.

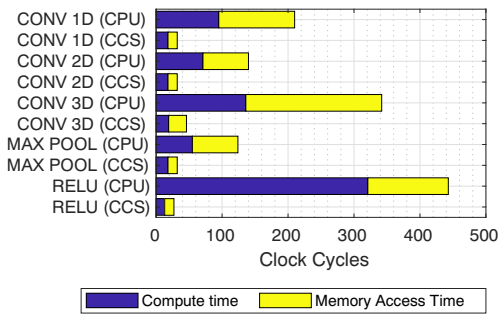

Figure 8 Performance results regarding five kernels commonly used by CNNs, showing the time dedicated to accessing the memory and the actual computation time.

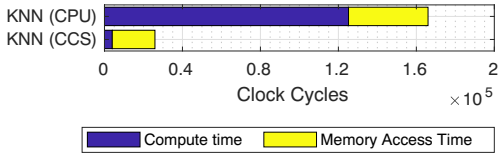

Figure 9 Performance results regarding the distance computation phase of the kNN algorithm, showing the time dedicated to accessing the memory and the actual computation time.

Furthermore, it is also noticeable that the convolution benchmarks do not attain higher speedups when using vectors of smaller elements (8-bit and 16-bit) (see Fig. 10). This is due to the fact that the NEON SIMD engine (used when executing kernels on the CPU) is capable of operating four 32-bit elements, eight 16-bit elements or sixteen 8-bit elements simultaneously. Therefore, both the CCS and the NEON engine consume the same amount of data per cycle regardless of the operands' representation. However, the Max Pool and the ReLU benchmarks do not use the SIMD unit of the processor due to their complex implementation. Therefore, reducing the size of the elements to half increases the number of clock cycles required to process a 64-byte vector. Naturally, reducing the size of the elements to 16-bit for the Max Pool and the ReLU benchmarks increases the speedup attained by the CCS 1.7× and 1.5×, respectively. Reducing the precision of the operands even further to 8-bit increases the performance benefits of the CCS 3.7× and 2.5×, respectively.

Finally, to further evaluate the advantages brought by the novel CCS, the proposed experimental evaluation also considered an alternative accelerating infrastructure similar to those that are typically found in a general-purpose computer, namely Graphics Processing Units (GPUs). In accordance, parallelized versions of the previously discussed benchmarks were produced and executed in two different GPU devices (GPU 1: GeForce GTX 1080 Ti; GPU 2:

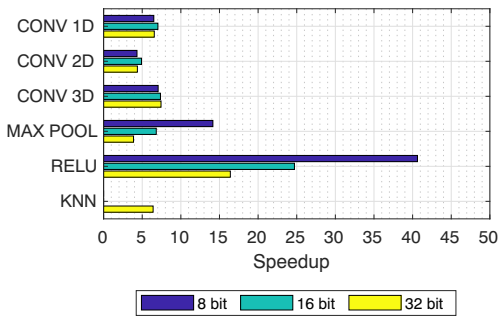

Figure 10 Speedup allowed by executing six kernels using the CCS.

Table 3 Thousands of cycles spent by the studied six kernels when executed using only the CPU, the CCS, or a GPU device (GPU 1: GeForce GTX 1080 Ti; GPU 2: TITAN RTX).

	CPU only	CPU + CCS	CPU + GPU 1			CPU + GPU 2		
			Kernel	Transfers	Total	Kernel	Transfers	Total
Conv 1D	210	32	236	2124	2360	42	504	546
2D	1406	320	108	2952	3060	20	408	428
3D	343	46	278	1882	2160	38	330	368
Max Pool	135	35	94	1744	1838	16	304	320
ReLU	4440	270	26	946	972	12	308	320
kNN	166000	25891	62000	194000	456000	56000	96000	152000

TITAN RTX). Table 3 shows the performance results (in thousands of clock cycles) obtained while executing the kernels using these GPUs and compares them with the CPU and the proposed CCS.

In general, it was observed that the GPUs perform significantly worse than the CPU-only and the CCS systems, which can be explained by the fact that the tested kernels are DRAM-bound. Hence, while the CCS focus on processing at the DRAM's peak bandwidth with a low latency, the GPU's performance is highly limited by the additional overheads of transferring data between the external DRAM and the GPU's global memory. Therefore, even though the GPU's relative performance improves for larger datasets, it is always bound by data transfers for the CCS' application use cases. Furthermore, processing on the GPUs leads to an overhead associated with the launching of threads in the GPU.

On the other hand, the CCS allows a much higher bandwidth to the memory, spends virtually no time transferring the operands and storing the results, and does not have an overhead associated with launching threads.

Furthermore, the available hardware in GPU devices highly surpasses that of the CCS. Naturally, this imposes a significant overhead in terms of both the global system's hardware and energy requirements, which is not always compatible with the system's constraints (e.g., edge devices which are highly constrained in terms of both hardware and energy supply).

6 Conclusions and Future Work

This paper proposes a novel architecture to explore the performance benefits of NDP. The devised CCS is meant to be integrated with a common general-purpose CPU, and operates over data directly fetched from the cache, leveraging both proximity to the data (fetching and storing data becomes less time consuming) and the access to an entire cache line simultaneously. For evaluation purposes, the CCS was integrated with a high-performance ARM Cortex-A53 CPU and simulated using the gem5 architectural simulator. A comprehensive software framework was developed to program and synchronize the CCS from plain C code. Results show that the CCS provides significant performance benefits in the form of reduced memory communications and increased processing power. Six kernels commonly used in the context of CNNs and clustering algorithms were tested, with the obtained speedups ranging from 3.9× to 40.6×. Additionally, GPU-parallelized versions of the tested kernels were used to compare the performance of the CCS with conventional GPU devices, which allowed to conclude that the CCS presents a better alternative than a GPU whenever the application is DRAM-bound.

6.1 Future Work

The evaluation of the CCS architecture has shown the benefits of offloading certain memory-bound workloads to a NDP co-processor. However, multiple optimizations are still possible to maximize the applications and the performance of this mechanism. Presently, the CCS only supports the execution of a single command each time it is programmed. A natural development is to allow the CCS to execute kernels composed of multiple commands without the need for reprogramming between them. This would not only reduce the programming overhead but also allow to explore other optimization techniques such as command rescheduling and data prefetching. Another possible research direction is to extend the CCS and couple it to several levels of the memory hierarchy, allowing the same co-processor to fetch data from several sources depending on the locality of the operands.

References

1. Wulf, W.A., & McKee, S.A. (1995). Hitting the memory wall: implications of the obvious. *SIGARCH Computer Architecture News*, *23*(1), 20–24.
2. Vieira, J., Duarte, R.P., Neto, H.C. (2019). kNN-STUFF: kNN streaming unit for Fpgas. *IEEE Access*, *7*, 170864–170877.
3. Aga, S., Jeloka, S., Subramaniyan, A., Narayanasamy, S., Blaauw, D.T., Das, R. (2017). Compute caches. In *HPCA* (pp. 481–492): IEEE Computer Society.

4. Vieira, J., Giacomin, E., Qureshi, Y.M., Zapater, M., Tang, X., Kvatinsky, S., Atienza, D., Gaillardon, P. (2019). A product engine for energy-efficient execution of binary neural networks using resistive memories (pp. 160–165): IEEE.
5. Ghose, S., Hsieh, K., Boroumand, A., Ausavarungnirun, R., Mutlu, O. (2018). Enabling the adoption of processing-in-memory: challenges, mechanisms, future research directions. arXiv:1802.00320.
6. Kim, N.S., & Mehra, P. (2019). Practical near-data processing to evolve memory and storage devices into mainstream heterogeneous computing systems. In *DAC* (p. 22): ACM.
7. Shafiee, A., Nag, A., Muralimanohar, N., Balasubramonian, R., Strachan, J.P., Hu, M., Williams, R.S., Srikumar, V. (2016). ISAAC: a convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *ISCA* (pp. 14–26): IEEE Computer Society.
8. Vieira, J., Roma, N., Tomás, P., Ienne, P., Falcao, G. (2018). Exploiting compute caches for memory bound vector operations. In *SBAC-PAD* (pp. 197–200): IEEE.
9. Vieira, J., Roma, N., Falcao, G., Tomás, P. (2020). Processing convolutional neural networks on cache. In *ICASSP 2020-2020 IEEE international conference on acoustics, speech and signal processing (ICASSP)* (pp. 1658–1662): IEEE.
10. Li, S., Niu, D., Malladi, K.T., Zheng, H., Brennan, B., Xie, Y. (2017). DRISA: a DRAM-based reconfigurable in-situ accelerator. In *MICRO* (pp. 288–301): ACM.
11. Seshadri, V., Lee, D., Mullins, T., Hassan, H., Boroumand, A., Kim, J., Kozuch, M.A., Mutlu, O., Gibbons, P.B., Mowry, T.C. (2017). Ambit: in-memory accelerator for bulk bitwise operations using commodity DRAM technology. In *MICRO* (pp. 273–287): ACM.
12. Seshadri, V., Hsieh, K., Boroumand, A., Lee, D., Kozuch, M.A., Mutlu, O., Gibbons, P.B., Mowry, T.C. (2015). Fast bulk bitwise AND and OR in DRAM. *IEEE Computer Architecture Letters*, *14*(2), 127–131.
13. Li, S., Xu, C., Zou, Q., Zhao, J., Lu, Y., Xie, Y. (2016). Pinatubo: a processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *DAC* (pp. 173:1–173:6): ACM.
14. Yitbarek, S.F., Yang, T., Das, R., Austin, T.M. (2016). Exploring specialized near-memory processing for data intensive operations. In *DATE* (pp. 1449–1452): IEEE.
15. Chi, P., Li, S., Xu, C., Zhang, T., Zhao, J., Liu, Y., Wang, Y., Xie, Y. (2016). PRIME: a novel processing-in-memory architecture for neural network computation in ReRAM-based main memory. In *ISCA* (pp. 27–39): IEEE Computer Society.
16. Cheng, M., Xia, L., Zhu, Z., Cai, Y., Xie, Y., Wang, Y., Yang, H. (2019). TIME: a training-in-memory architecture For RRAM-based deep neural networks. *IEEE Trans. on CAD of Integrated Circuits and Systems*, *38*(5), 834–847.
17. Wang, Y., Chen, W., Yang, J., Li, T. (2018). Towards memory-efficient allocation of CNNs on processing-in-memory architecture. *IEEE Trans. Parallel Distrib. Syst.*, *29*(6), 1428–1441.
18. Subramaniyan, A., Wang, J., Balasubramanian, E.R.M., Blaauw, D.T., Sylvester, D., Das, R. (2017). Cache automaton. In *MICRO* (pp. 259–272): ACM.
19. Wang, X., Yu, J., Augustine, C., Iyer, R.R., Das, R. (2019). Bit prudent in-cache acceleration of deep convolutional neural networks. In *HPCA* (pp. 81–93): IEEE.
20. Eckert, C., Wang, X., Wang, J., Subramaniyan, A., Sylvester, D., Blaauw, D.T., Das, R., Iyer, R.R. (2019). Neural cache: bit-serial in-cache acceleration of deep neural networks. *IEEE Micro*, *39*(3), 11–19.
21. Nag, A., Ramachandra, C.N., Balasubramonian, R., Stutsman, R., Giacomin, E., Kambalasubramanyam, H., Gaillardon, P. (2019). GenCache: leveraging in-cache operators for efficient sequence alignment. In *MICRO* (pp. 334–346): ACM.
22. Ahn, J., Yoo, S., Mutlu, O., Choi, K. (2015). PIM-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In *ISCA* (pp. 336–348): ACM.
23. Cong, J., & Xiao, B. (2014). Minimizing computation in convolutional neural networks. In *ICANN. Volume 8681 of Lecture Notes in Computer Science* (pp. 281–290): Springer.
24. Giacomin, E., Greenberg-Toledo, T., Kvatinsky, S., Gaillardon, P. (2019). A robust digital rram-based convolutional block for low-power image processing and learning applications. *IEEE Trans. Circuits Syst. I Regul. Pap.*, *66-I*(2), 643–654.
25. Pouyan, P., Amat, E., Hamdioui, S., Rubio, A. (2016). RRAM variability and its mitigation schemes. In *2016 26th international workshop on power and timing modeling, optimization and simulation (PATMOS)* (pp. 141–146): IEEE.
26. Liu, X., Zhou, M., Rosing, T.S., Zhao, J. (2019). HR^{3}AM: a heat resilient design for RRAM-based neuromorphic computing. In *ISLPED* (pp. 1–6): IEEE.
27. Bo, C., Wang, K., Fox, J.J., Skadron, K. (2016). Entity resolution acceleration using the automata processor. In *BigData* (pp. 311–318): IEEE Computer Society.
28. Qureshi, Y.M., Simon, W.A., Zapater, M., Atienza, D., Olcoz, K. (2019). Gem5-X: a gem5-based system level simulation framework to optimize many-core platforms. In *SpringSim* (pp. 1–12): IEEE.

João Vieira received his M.Sc. degree in Electrical and Computer Engineering from Instituto Superior Técnico, Portugal in 2018. He is now pursuing a Ph.D. at the same University while being a research assistant at the High-Performance Computing Architectures and Systems research group at Instituto de Engenharia de Sistemas e Computadores - Investigação e Desenvolvimento, Lisbon, Portugal. In 2018, he performed a research internship at the Processor Architecture Laboratory at the Swiss Federal Institute of Technology Lausanne, Switzerland, and from January until August 2019 he did another research internship at the Laboratory for NanoIntegrated Systems at the University of Utah, USA. His research interests include High-Performance Computing Architectures, Near-Data Processing, and Hardware/Software Co-Design.

Nuno Roma received the Ph.D. degree in electrical and computer engineering from Instituto Superior Técnico (IST), Universidade Técnica de Lisboa, Lisbon, Portugal, in 2008. Currently, he is an Associate Professor with the Department of Electrical and Computer Engineering of IST and a Senior Researcher of the High Performance Computing Architectures and Systems (HPCAS) research area of Instituto de Engenharia de Sistemas e Computadores R&D (INESC-ID). His research interests include computer architectures, specialized and dedicated structures for digital signal processing, energy-aware computing, parallel processing and high-performance computing systems. He contributed to more than 100 manuscripts to journals and international conferences and served as a Guest Editor of Springer Journal of Real-Time Image Processing (JRTIP) and of EURASIP Journal on Embedded Systems (JES). He has also acted as the organizing chair of several workshops and special sessions. He has a consolidated experience on funded research projects leadership and he is member of several research Networks of Excellence (NoE), including HiPEAC (European Network of Excellence on High Performance and Embedded Architecture and Compilation). Dr. Roma is a Senior Member of IEEE and ACM.

Gabriel Falcao received the Ph.D. degree from the University of Coimbra in 2010. He is currently an Assistant Professor with the Department of Electrical and Computer Engineering of the University of Coimbra and a Researcher with the Instituto de Telecomunicações. His research interests include parallel computer architectures, energy-efficient processing, GPU- and FPGA-based accelerators, and compute-intensive signal processing applications. In 2011/12 and again in 2017/18 he was a Visiting Professor with EPFL, and in the summer of 2018 he was a Visiting Academic at ETHZ, Switzerland. Gabriel is a member of the IEEE Signal Processing Society, a Senior member of the IEEE and a full member the HiPEAC network of excellence.

Pedro Tomás received the Ph.D. in Electrical and Computer Engineering (ECE) from Instituto Superior Técnico (IST), Technical University of Lisbon, Portugal, in 2009. He is an assistant professor in the Dept. of ECE, IST, and a senior researcher at Instituto de Engenharia de Sistemas e Computadores R&D (INESC-ID). His research activities include computer microarchitectures, specialized computational structures, and high-performance computing. He is also interested in artificial intelligence models and algorithms. He is a member of the IEEE Computer Society and has contributed to more than 60 papers to international peer-reviewed journals and conferences.