Exploiting Near-Cache Processing for Memory Bound Vector Operations

João Vieira
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal

***Abstract*—To reduce the average memory access time, most current processors make use of a multilevel cache subsystem. However, despite the proven benefits of such cache structures in the resulting throughput, conventional operations such as copy, simple maps and reductions still require moving large amounts of data to the processing cores. This imposes significant energy and performance overheads, with most of the execution time being spent moving data across the memory hierarchy. To mitigate this problem, a Cache Compute System (CCS) that targets memory-bound kernels such as map and reduce operations is proposed. The developed CCS takes advantage of long cache lines and data locality to avoid data transfers to the processor, and exploits the intrinsic parallelism of vector compute units to accelerate a set of 48 operations commonly used in map and reduce patterns. The CCS architecture was validated by simulation using gem5 and compared with a PULPino soft-core running on a Xilinx ZYNQ-7 ZC706 Evaluation Board. Furthermore, the CCS was integrated in a system featuring an MB-LITE soft-core and a memory subsystem, and the system was implemented in a Xilinx Virtex-7 VC709 Development Board. When compared to the MB-Lite core, the proposed CCS presents performance improvements in the execution of the commands ranging from $18\times$ to $94\times$, and energy efficiency gains from $11\times$ to $67\times$. On the other hand, the CCS can perform between $132\times$ and $401\times$ better than PULPino for 1024 32-bit elements operands.**

***Index Terms*—Near Data Processing, *Near-cache* processing, Memory subsystem, Memory bound operations, Cache Compute System, RTL implementation**

I. Introduction

Memory-bound applications spend most of the time transferring data between the processor and the memory subsystem, often rendering useless the CPU resources dedicated to instruction level parallelism. Although cache systems aim to mitigate this issue, a significant overhead is still imposed by transferring data through the entire memory hierarchy.

In particular, many of those applications are often characterized by low computational intensities, requiring data to be moved to the processing core just to apply a rather limited set of transformations (e.g., map and reduce). However, while the processing workload of these operations is almost negligible, the memory subsystem easily becomes saturated with the amount of data that has to be moved, and cannot feed the processor at the desired rate. In fact, given the low complexity of these operations, Processing in Memory (PIM) approaches based on moving the computation resources closer to where the operands are stored have become increasingly popular [2]–[8], [10]–[12], [14]–[16], [18]–[22], [31], [33]. Recently, some rather similar solutions based on pushing the processing to the cache subsystem have also been proposed [1], [9], [23], [24]. Both topics rely on performing some modifications to the memory structures to enable *near-data* computing, taking advantage of the high internal bandwidth of the memory devices and enabling massive parallelism by applying extensive vector processing techniques. However, such solutions are often limited in terms of what can be directly calculated in memory, allowing only a few and rather elementary bitwise logic operations, often of limited interest for the majority of real-world applications. Furthermore, this type of technology only allows map-type operations, meaning that reductions cannot be implemented at this level.

On this thesis, a somewhat more ambitious acceleration infrastructure is proposed by integrating a stream vector compute unit within a conventional cache system. The vector compute unit supports 48 different arithmetic, shift and logic operations that match most common parallel processing patterns, namely, map and reduce. Furthermore, it does not require any significant modifications to the base system other than internal connections with a conventional cache system. Differently from the most common solutions, the proposed Cache Compute System (CCS) also implements a stride functionality, which allows operating over stridden vectors without the need of reallocating them in advance. It also offers elementary hardware loop capabilities, which means that the operands of the CCS are not limited in length by the physical implementation of the architecture. When the operands are too long to be operated at once, they are partitioned and the CCS will operate one partition at a time, eliminating the overhead of doing this by software.

In accordance, the following contributions are offered:

- architecture of a stream-based Compute Unit (CU) that implements 48 arithmetic, shift and logic vector operations, particularly useful for a diversity of applications that include map and reduce processing patterns;
- behavioral simulation and validation of the developed architecture using gem5;
- Register Transfer Level (RTL) implementation of the CCS in VHSIC Hardware Description Language (VHDL);
- simple framework to program and control the CCS;
- comparison and integration of the devised architecture with several processing systems;
- performance and energy efficiency improvements evaluation of the considered setups, each one featuring a processing system and the proposed mechanism.

To validate its behaviour, the developed architecture was simulated using gem5 and its performance was compared to a PULPino soft-core [25], [26] running on a Xilinx

ZYNQ-7 ZC706 Evaluation Board. Furthermore, it was prototyped using fixed-point precision on a Xilinx Virtex-7 Field-Programmable Gate Array (FPGA) based on a VC709 Development Board, alongside with an MB-Lite soft-core [13]. When compared with the PULPino standalone system, it provides a processing speedup as high as $401\times$ and energy efficiency improvements of $328\times$.

II. Cache Compute system architecture

The proposed Cache Compute System (CCS) is depicted in Figure 1. It is defined as a logical block that includes a conventional cache and the tightly interconnected CU.

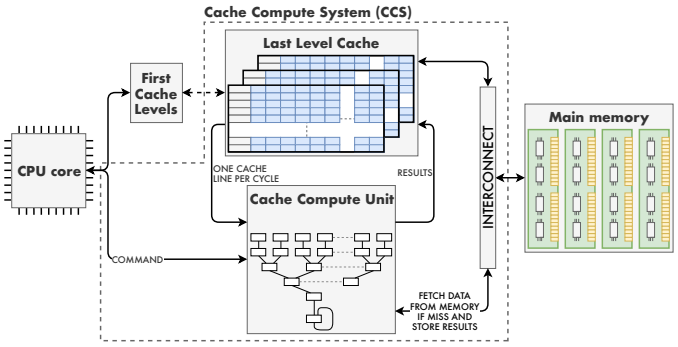


Fig. 1: System overview integrating the processor, the CCS and the main memory. The CCS is defined as the logic block that includes the cache and the CU.

From the perspective of the CPU, the CCS behaves just like a memory device, while from the memory side it performs simply as a conventional data requester. As such, standard memory interfaces are used for both communication channels: the CPU core has a single interface with the CCS, used for data transfers and for the CU programming. The processor and the CCS operate independently, and while the CU reads the operands or writes the results of a command, the processor is free to operate over other data. Within the CCS, the cache also has two memory interfaces: one to the processor and another to the CU—for the same reason stated above.

Since the processor has a single interface to the CCS, the CU control registers are memory-mapped in the processor's addressing space. To program the CU, the processor simply writes its control registers and signals the unit to start. One of the CU control registers indicates its readiness, making it possible to know when it is idle and when has the last issued command finished. Table I details the control registers of the CCS's programming interface.

A. Supported operations

The 48 arithmetic, shift and logic vector operations supported by the devised CU are presented in Table II. These operations were chosen by taking into consideration the algorithms used in a wide range of application domains including clustering, machine learning, cryptography, image processing, biomedicine, and linear algebra. Table III lists some example applications that can benefit from the proposed CCS, together with some of the commands likely to be used to parallelize parts of those algorithms.

These commands can be classified accordingly to different criteria: (1) The mathematical operation: arithmetic, logic, shift, or move; (2) The operands: two vectors (VOP2), a vector and a constant (VCOP), a single vector (VOP1), or only a constant (COP); (3) The functional operation: map or reduce.

B. Data processing structure

Figure 2 depicts the integration of the CU within the CCS. To allow the implementation of both map and reduce operations, the CU has two processing parts: the first part corresponds to the two first levels of the compute unit (L0 and L1), providing a parallel-input parallel-output capability required to implement the map operations (two levels are needed for the map-reduction-type operations, such as the sum of square differences, which needs two consecutive map operations, difference and square, before the reduction); the second part comprehends the remaining levels and adopts a binary-tree shaped structure, required to implement the reduce-type commands.

To satisfy all the commands supported by the proposed architecture, the first two levels of the CU have to support more operations (e.g., the second level of the CU is the only one equipped with integer multipliers, and the first level implements shift operations), as shown in Figure 2. The levels responsible for the reduction-type commands only implement subsets of the total arithmetic and logic operations.

C. Operand fetch and result store

The CCS operates as a stream processor with the operands and result being described as a 1-D descriptor composed of base address, stride, and length. To execute a given operation, the CCS divides the operands into groups of N elements, where N corresponds to the vector unit length, which should be adjusted according to the size of the cache line. Naturally, since the first and last elements may not be aligned in the cache, specific masks are generated to guarantee a correct operation over the described arrays (see subsection II-D).

When executing commands, the CCS loads the operands in a sequential fashion (see also Fig. 3): for VOP2 commands, it first loads one operand, storing it in an input buffer, before

TABLE I: Control registers offered by the CCS' programming interface mapped in the processor's addressing space.

Relative address	Function	Register mode	Required
0x00	Command ID	Read/write	Always
0x04	Operands length	Read/write	VOP2, VOP1, VCOP
0x08	Constant	Read/write	VCOP, COP
0x0c	Op. A address	Read/write	VOP2, VOP1, VCOP
0x10	Op. B address	Read/write	VOP2
0x14	Result address	Read/write	Always
0x18	Op(s). stride	Read/write	Always
0x1c	Exec. mask	Read/write	Always
0x20			
0x24	*Reserved*		
0x28	Exec. start	Read/write	Always
0x2c	Readiness	Read-only	*Does not apply*

proceeding to the second operand; for VOP1 and VCOP, where a single operand exists, it skips the second operand and iteratively fetches data only from a single data input stream. At each data fetch, the CCS makes use of the existing cache structures. Hence, it first checks for a hit on the cache, before issuing a cache line request to the main memory. However, when loading data from memory, the operands are directly forwarded to the CU and are not stored in the cache. This avoids cache pollution, by reducing conflicts between CCS operands and processor data.

To store the command output, the CCS again relies on the cache existing structures and policies for data allocation and write. Hence, for write-back write-allocate caches, data is always written on the cache, whereas for write-through write-not-allocate, the result is always stored on the memory, with the cache being updated if the corresponding line(s) already exist on the cache.

TABLE II: Supported commands by the proposed CCS, comprehending four classes of mathematical operations: integer arithmetic, logic, shift, and move.

		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	$r[i] = a[i] \times k$	
		LESSVC	$r[i] = \begin{cases} 1 & , a[i] < k \\ 0 & , otherwise \end{cases}$	
		GRTRVC	$r[i] = \begin{cases} 1 & , a[i] > k \\ 0 & , otherwise \end{cases}$	
		EQUVC	$r[i] = \begin{cases} 1 & , a[i] = k \\ 0 & , otherwise \end{cases}$	
	VOP1	COMP2	$r[i] = -a[i]$	
		SQV	$r[i] = a[i]^2$	
		ABSV	$r[i] = \lvert a[i] \rvert$	
		ADDV	$r[i] = \sum_i a[i]$	*reduce*
		MAXV	$r[i] = \max(a)$	
		MINV	$r[i] = \min(a)$	
Shift	VOP2	SLLVV	$r[i] = \text{sll}(a[i], b[i])$	*map*
		SRLVV	$r[i] = \text{srl}(a[i], b[i])$	
		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 $b[i]$	*map*
		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	
	VOP1	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 $a[B-1]$	
Move	COP	INITC	$r[i] = k$	*map*
	VOP1	COPYV	$r[i] = a[i]$	

TABLE III: Examples of applications likely to benefit from the proposed CCS and the respective useful commands.

Area	Algorithm	CCS commands
Clustering	kNN	SSDVV
	k-Means	
	Mean-Shift	
	DBSCAN	
Machine Learning	CNN	IPVV
	Linear Regression	ADDV, MULVV, SUBVV
	Linear Discriminant Analysis	ADDV, MULVV, SUBVV
Cryptography	Md5 Checksum	ANDVV, ORVV, XORVV, ROLVV
	Serpent	XORV, ROLVC
	Twofish	INITC, XORVC, SLLVV, ANDVV, XORVV
	Blowfish	XORV, INITC
	Cast-128/256	ADDVC, XORVV
Image/Video Processing	Convolutional filters	IPVV
	Pixel matching	SADVV
Biomedicine	CAST [17]	MAXV, INITC
Linear Algebra	Intel BLAS	COPYV, MULVC, ADDVV, IPVV, SSDVV, SADVV

D. Mask generation and stride processing

When a command operates on vector operands, the stride functionality is enabled. As long as the step is a power of two smaller than the size of the cache line, a mask can be specified in such a way that the only considered elements of the operands will be the indexes that are multiples of the stride. However, for all commands that involve multiple vectors (e.g., two vector operands or one vector operand and a vector result), all the elements have to be aligned in the cache for the command to succeed. For striding and alignment correction purposes, two distinct masks are produced before the CCS command is executed. The boundary mask, generated directly by the CCS, is used to align the operands with the beginning and the end of a cache line (all the words in the cache line out of the operand's range are discarded). The stride mask is generated by software and each one of its bits is set to one when its index is multiple of the stride. In the CCS, both masks are combined, generating the execution mask. The execution mask determines which elements of the operands will be considered when executing the command, and which results are to be stored in memory. For map commands, only the execution mask is needed; however, for reduction commands, a new submask has to be generated at each level of the reduction three, determining which elements are to be reduced to the next level. Each submask is generated by performing the logic OR operation on the mask bits associated with the two operands of a given unit. Additionally, the instruction performed by each unit of the reduction tree is influenced by the bits of the mask associated with its operands. When both operands are to be considered, the instruction performed by the unit corresponds to the one that is meant to be executed in that level, but in case

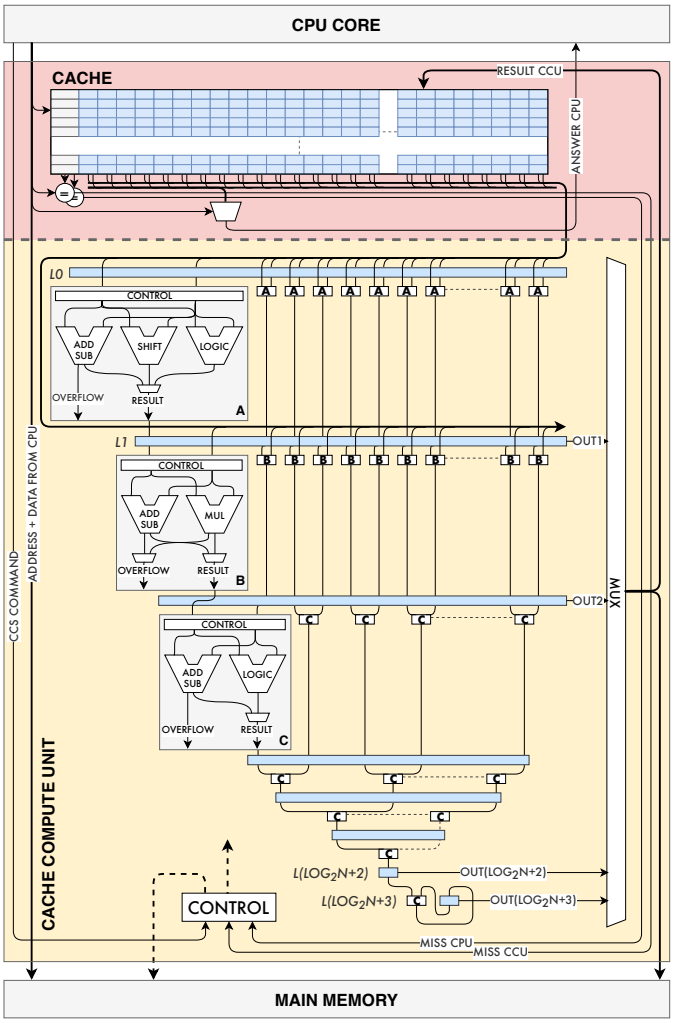


Fig. 2: Datapath of the CU. The first two levels of the binary structure are meant for maps and the remaining structure acts as a reduction tree. Consequently, different levels are equipped with different units: type-A unit–implements an adder/subtracter, a shift unit and a logic unit; type-B unit–implements an adder/subtracter and an integer multiplier; type-C unit–implements an adder/subtracter and a trimmed logic unit, capable of performing a subset of the total logic instructions.

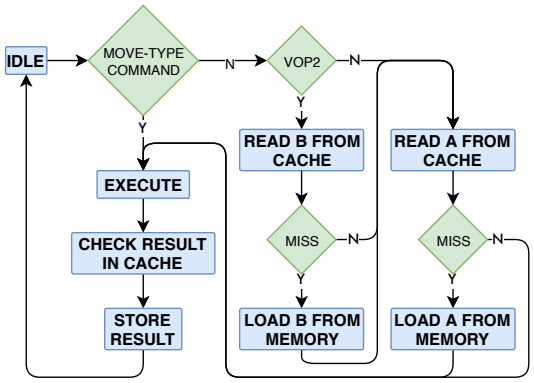


Fig. 3: Execution flow graph of a command by the CCS, showing the different states of operation by the CU.

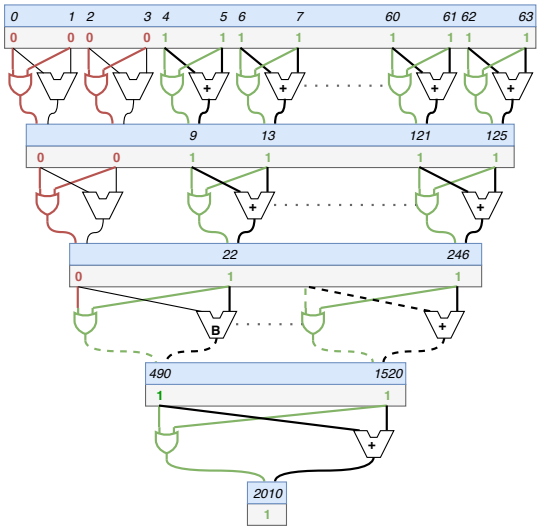


Fig. 4: Sum of all elements of a misaligned vector with 60 elements and generation of the intermediary execution masks for the reduction three from the main execution mask.

only one of the operands is valid, the output of the unit will be that same operand. Figure 4 illustrates an example of all the elements of a vector being summed, and also the submasks of each level being generated from the submask of the previous reduction level, when the execution mask excludes the first 4 elements from the cache line in a reduction.

III. CCS Programming and Operation

The communication protocol between the CPU and the CCS allows the CPU to interact with the CCS for three main purposes: (1) to program a command, (2) to order the start of the execution of the previously programmed command, and (3) to check for completion/readiness. The process of issuing a command to the CCS consists of saving the configuration parameters in the memory locations where the control registers of the CU are mapped. To ease this procedure, a convenient framework was developed requiring the programmer to know the minimum about the low-level programming flow of the CCS. This framework offers the following functions to program and control the CCS, together with macros to define the CCS commands:

- `__ccs_setup()`: takes as arguments the command identifier, the length of the operands, an optional constant, the addresses of the operands and of the result, and the stride; it also calculates the software mask and programs the CCS with the required parameters;
- `__ccs_start()`: signals the CCS to start the execution of the command previously programmed;
- `__ccs_check()`: checks (nonblocking) if the CCS is ready to execute, i.e., if it is not busy executing a command, and returns a logic value;
- `__ccs_wait()`: blocks the execution until the CCS is

```
int opa[VEC_SIZE], opb[VEC_SIZE], res[VEC_SIZE];

// program the CCS
__ccs_setup(ADDVV, VEC_SIZE, 0, opa, opb, res, 1);

// start the execution
__ccs_start();

routine1();

// wait until previous command completes
__ccs_wait();
```

Listing 1: Example of command being issued to the CCS. After programming the CCS, the processor executes some other routine that does not require to access its results, and before accessing the memory positions were the results are stored, a blocking call is executed that checks if the CCS is ready (if the previous command has finished).

ready to accept a new command.

Listing 1 illustrates a C code example used to program the CCS to add two vector operands, execute a routine in the processor simultaneously and independently of the CCS and to wait for the CCS to complete.

The execution of a command by the CCS is independent of the CPU. First, the CCS control unit checks the type of issued command. If the command is type VOP2, VOP1 or VCOP, the CCS will switch to a reading state, where it checks if the operands are in cache. If one or more operand(s) are not in cache, the CCS enters an idle state while they are fetched from the main memory to an input buffer. However, if the command type is COP, the CCS changes its state directly from idle to execute, since the only operand is the constant passed by the CPU. When it reaches the execution state, the CCS will keep executing for the predetermined number of cycles that a given command requires to complete. Finishing the execution of the command, the CCS returns to the idle state and notifies the processor about its readiness, by writing to the corresponding interface register.

Independently of the considered commands, the operands' length is not limited by the number of functional units of the first level. Instead, larger operands can be specified and hardware loops will take care of processing them in parts until they are all consumed. Depending on the type of the command, these loops are executed differently, as shown in Figure 5. For map-type commands, each partition of the operands is processed independently and the results are stored while the next partition is fetched. However, the prototyped architecture does not allow storing results in memory while reading operands for the next iteration, as only one half duplex memory channel is implemented. As such, the fetch of the operands for the next iteration can only occur after the result of the previous one is written. For reduces, the operands are entirely fetched and the execution happens in pipeline.

IV. Experimental Results

To validate the behavior of the proposed system, a comprehensive simulation framework was developed, using gem5.

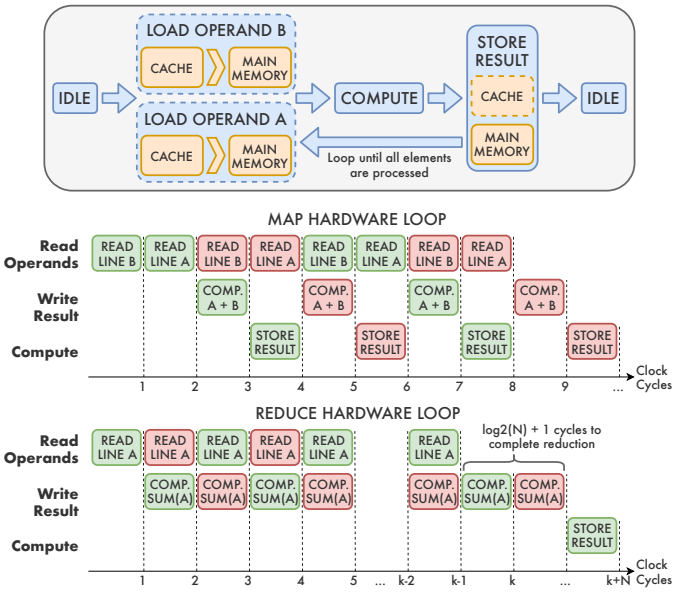


Fig. 5: Finite state machine that describes the control flow of the CU. The examples below describe the flow of the hardware loops. For map commands, the partitions of the operands are fully processed and the results are stored for each iteration. Differently, for reduce commands, the operands are fully loaded and the execution happens in pipeline.

However, architectural simulators do not allow to evaluate, by default, the hardware and energy requirements of a system. Thus, to evaluate the performance and energy efficiency gains that may be provided by the presented CCS, a comprehensive experimental procedure was conducted based on an hardware prototype of the proposed structures. In particular, this section presents the obtained experimental results in what concerns the resulting CCS operating frequency, occupied hardware resources, throughput (performance) and power consumptions. In accordance, two setups are considered: one which considers a soft-core (PULPino) as reference [25], [26]; and another where the CCS is tightly coupled with an MB-LITE soft-core. Due to compatibility constraints, while the configuration featuring PULPino was synthesized and implemented using Xilinx Vivado 2016.4 and a Xilinx ZYNQ-7 ZC706 Evaluation Board, for the system integrating the CCS with the MB-LITE [13], Xilinx Vivado 2018.1 and a Xilinx VC709 Development Board were used.

The setup featuring an MB-LITE soft-core and the CCS also included a memory subsystem composed by the main memory (made of FPGA native Block RAMs (BRAMs)) and a single level cache. The cache was designed to be direct-mapped with write-through and write-not-allocate policies from the Central Processing Unit (CPU) point of view, and write/read-not-allocate from the CU point of view. Although the cache was designed to be fully reconfigurable, it was implemented having 16 lines, each with 2048-bits, plus the tag and 1 validity bit per line. Naturally, the size of the implemented CU matches the size of the cache lines, and the reduction tree (levels 3 to $\log_2(N)+3$) has 7 computation levels (6 for reduction and 1

for accumulation).

A. Area and power requirements

The maximum operating frequency of PULPino and the CCS implementations, together with the occupied hardware resources and resulting power consumption, when considering an implementation based on a ZYNQ-7 FPGA, are presented in Table IV.

The presented results show that PULPino is capable of working at a maximum frequency of 40 MHz on the target ZYNQ-7 device, while the CCS is able to operate at a much higher frequency (132 MHz). Consequently, the limited operating frequency of PULPino represents an drawback of using it as host processor, since it would slow down the processes that require the CPU and the CCS to communicate (e.g. the programming process, that, as it will be proven later, represents a significant overhead).

The hardware resources used to implement the CCS are significantly higher than the resources occupied by PULPino. The CCS implements a massive quantity of functional units, which justifies the area overhead. We consider it a fair price, given the achieved speed-up and the energy gains only possible when using the CCS. The total power consumption of the CCS is lower than the base system, and given the smaller execution time, the energy savings due to the CCS are even higher.

Table V presents the required resources, frequency of operation and power of a base system featuring an MB-LITE and a memory subsystem (composed by a single level cache and a main memory), and a setup also featuring the CCS. The presented values show that the proposed CCS exceeds the area resources occupied by the base system by a factor of 10. Nevertheless, the complete system still uses less than 25% of the total resources available in the considered device.

B. Microbenchmarks

To further assess the performance of the CCS, a battery of microbenchmarks was developed to evaluate the performance of each arithmetic, logic and shift command. Each microbenchmark calls the respective command to be executed in the CCS and implements the correspondent loop to be executed in the reference processor. Moreover, when comparing against the PULPino soft-core, the size of the vector operands was varied between 64 and 1024, by considering 32-bit data elements.

The speedup of the CCS over PULPino for these different microbenchmarks is depicted in Figure 6. The obtained speedups range from 132× to 401× when comparing to PULPino.

A similar set of microbenchmarks (the commands INITC and COPYV were added) was used to assess the performance of the CCS against MB-LITE. Similarly to what was done for PULPino, all the results were analyzed independently of the operating frequency, being the latencies measured in clock cycles.

As shown in figure 7, the use of the CCS provides considerable speedups ranging from 4× (INITC) to 66× (ROLVV, RORVV), and energy efficiency improvements between 6×

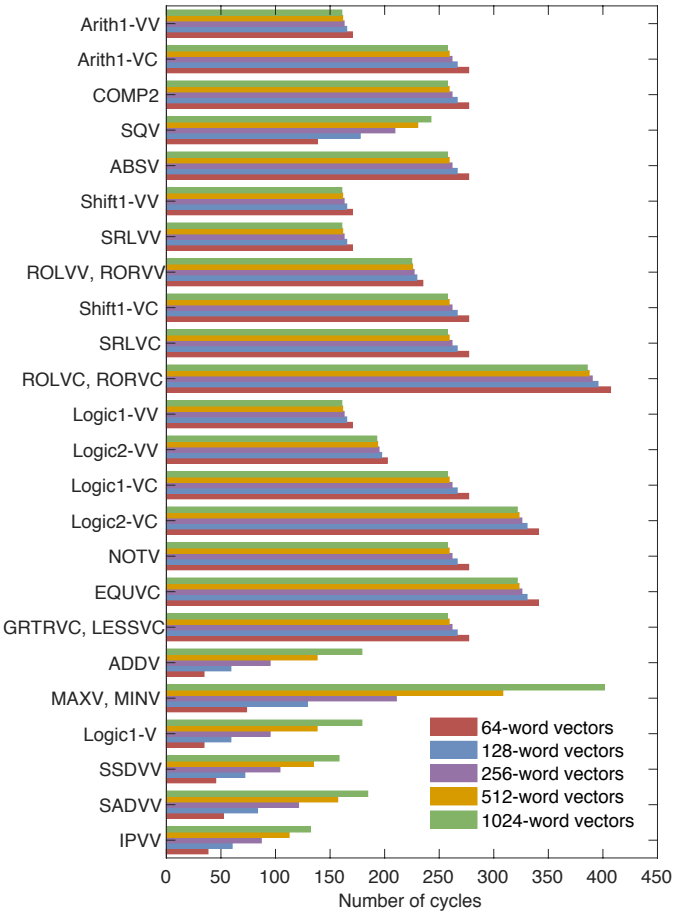


Fig. 6: Speedup of the proposed CCS when compared to PULPino, when executing vector operations with 1024 data elements (includes overheads for kernel launch). Both PULPino and CCS operate at 40 MHz.

and 90×. This is mainly due to three factors explored by the CCS: (1) intrinsic parallelism of the commands (SIMD-like commands), (2) in-place processing of data (avoiding moving it to the processing core), and (3) mitigation of the loop control required by the processor. Furthermore, both the speedups and energy efficiency improvements were shown to scale with the size of the operands, as shown in Figure 8 for 4 different CCS commands. In particular, for 32-bit vectors of 1024 elements, the obtained speedups scale up to 408×.

While the static power is similar for both parts of the complete system (the base system and the CCS), the dynamic power of the CCS is about 5× higher than the equivalent for the MB-LITE (see Table V). However, due to the order of magnitude of the achieved speedups, there are still significant energy savings. For 1024 32-bit vector operands, the theoretical energy efficiency improvements can be as high as 328×.

C. Application benchmarks

To further assess the performance impact of the proposed CCS, several application benchmarks were considered. For comparing with PULPino, the K-Nearest Neighbors (KNN) and KMeans applications were used, whereas KNN, Matrix

TABLE IV: Timing and hardware resources summary of the reference PULPino processor and of the proposed CCS, obtained using Xilinx Vivado 2016.4 when using a ZYNQ-7 ZC706 Evaluation Board.

System	Timing	Resources				Total Power consumption
		Slice LUTs	Slice Registers	Memory	DSP	
PULPino	40MHz	22193 (10.15%)	13472 (3.08%)	16 (2.94%)	10 (1.11%)	1.91W
CCS	132MHz	59749 (27.33%)	10464 (2.39%)	0 (0.00%)	192 (21.33%)	5.05W
	40MHz					1.69W

TABLE V: Occupied hardware resources, operation frequencies and power requirements of the considered setups.

		Base System		CCS		Complete System	
		Util.	%	Util.	%	Util.	%
Resources	LUT	6351	2	87675	21	91596	22
	LUTRAM	2053	2	7	1	7	1
	FF	2483	1	13014	2	13368	2
	BRAM	34	3	128	9	162	11
	DSP	3	1	192	6	195	6
Frequency [MHz]		100		100		67*	
P [W]	Static	0.329		0.345		0.345	
	Dinamic	0.150		0.793		0.612	

* The complete system maximum clock frequency decreases $\sim 30\%$ (regarding the individual systems) because of routing issues. Additional CCS-to-MB-LITE pipeline stages were not introduced because it would impact the performance of the MB-LITE.

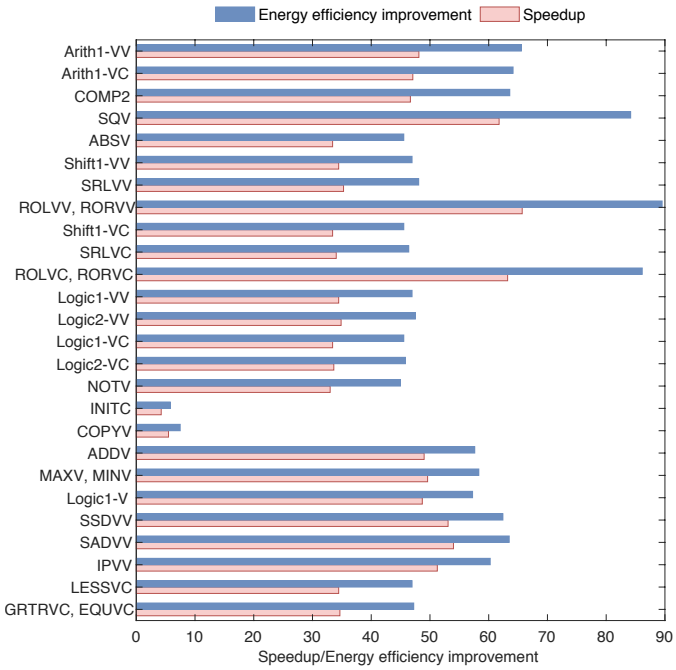
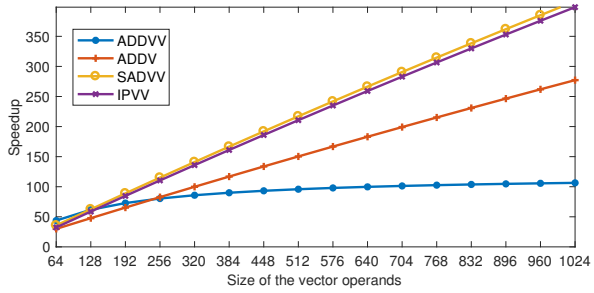
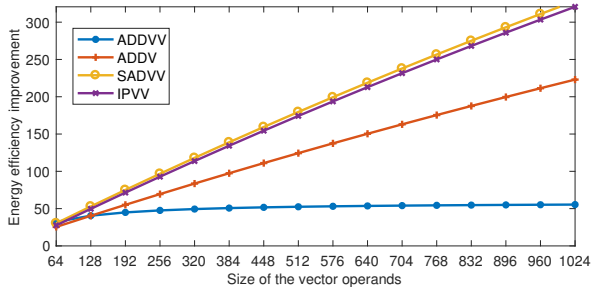


Fig. 7: Speedup and energy efficiency improvement provided by the CCS versus the MB-LITE for 64 32-bit element vector operands. For comparison purposes, the execution latencies were measured in clock cycles.

Multiplication and Linear Regression were used to evaluate the performance of the CCS over MB-LITE. In these applications, the code segments that can be accelerated by the developed CCS were firstly identified. For both the k-Nearest Neighbors (kNN) and KMeans, the euclidean distance phase are



(a) Speedup



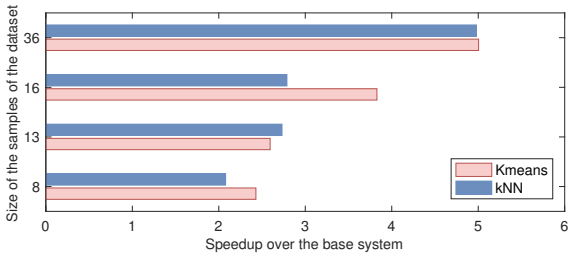
(b) Energy efficiency improvement

Fig. 8: Speedups and energy efficiency improvements with 4 CCS commands when compared to their sequential versions executing in the MB-LITE for several vector operands sizes.

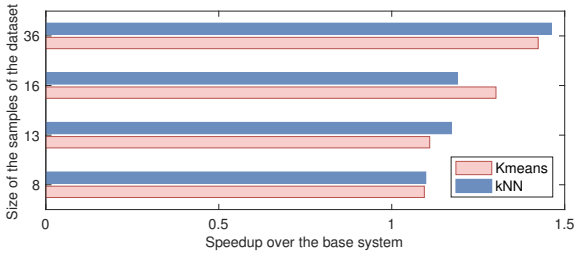
performed in the CCS, while the Matrix Multiplication and the Linear Regression are fully performed in the CCS.

KNN and KMeans were ran on PULPino by considering four data-sets chosen from the University of California, Irvine, Machine Learning Repository [27]–[30], with a different number of features. For each run, it was measured: (1) the number of cycles of the run; (2) the number of cycles corresponding to each section to be executed in the CCS; (3) the number of times the code to be executed in the CCS was run.

By applying Amdahl's law, and assuming that the PULPino processor idles while the CCS operates (and ignoring the CCS programming time), it is possible to calculate the theoretical speed-up of the developed system integrating the PULPino soft-core and the proposed CCS, depicted in Figure 9, alongside with the corresponding energy savings. The obtained speed-ups are related with the number of features of each data-set. When the number of features approaches 64 (the width of the top-level of the developed CCS), all resources are used with the maximum parallelism that can be achieved. For the kNN algorithm, the speed-ups range from $2\times$ to $6.8\times$, and



(a) Scalar PULPino



(b) PULPino featuring a 256-bit SIMD unit

Fig. 9: Speedup of the proposed CCS when implementing the KMeans and KNN algorithms.

for the KMeans algorithm, the speed-ups range from $2.4\times$ to $4.7\times$. The energy savings range from 54% to 76% for the KMeans algorithm and from 46% to 83% for the kNN. When considering the CCS performing at 132 MHz, the achieved gains in terms of the energy savings are similar.

For the system featuring the MB-LITE, the following benchmarks were adapted to use the proposed CCS (using the provided programming framework): KNN (with 64 coordinates per sample, $k = 4$, 64 control samples and 1 sample to classify); the integer Matrix Multiplication (64×64 matrices); and the Linear Regression (64 2D points). Differently from what was done for PULPino, these results were not extrapolated, but obtained directly by measuring the execution time of each benchmark. According to the workload that is delegated to the CCS (Amdahl's law) the overall speedups obtained for these three algorithms reach $68\times$, $54\times$ and $3\times$, respectively, as shown in Figure 10. The parallelized versions of the Matrix Multiplication and the KNN show better performances than the Linear Regression because their parallelized phases consist of a single CCS command each. For the Linear Regression, the parallelized phase takes 5 distinct commands, which not only increases the programming overhead but also does not allow to fully explore the CCS pipeline capabilities.

V. Related work

The Near Data Processing (NDP) paradigm was first introduced in the 70's, mainly to help mitigating the memory wall problem [32]. The principle behind PIM architectures relies on the main memory's internal high bandwidth, which allows massive parallelism. Moreover, by performing computation directly in the memory, the data transfers between the memory and the cores are avoided, reducing both execution time and energy consumption. Recent changes to the common memory

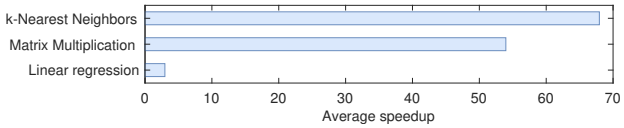


Fig. 10: Obtained speedups when running three application benchmarks partially executed in the proposed CCS: the KNN (with 64 coordinates per sample, $k = 4$, 64 control samples and 1 sample to classify), the integer Matrix Multiplication (64×64 matrices), and the Linear Regression (64 2D points).

hierarchies made possible such solutions to become viable in terms of implementation cost, such as the introduction of 3D-stacking. Naturally, several new approaches in PIM were also considered (e.g., [14], [15], [18], [19]). Seshadri et al. [19] proposed a fast bulk bitwise AND and OR mechanism to be implemented in the DRAM chip. This technology uses bit-line computation, which consists of simultaneously activating multiple lines of the memory. The several loads of each memory cell are combined and the sensed value (in a given) line is the result of a logic operation. Another approach that relies on bit-line computation is Pinatubo [14], although specifically targeting nonvolatile memories. It also supports XOR and INV operations (in addition to AND and OR) and attempts to solve conflicts such as the operands not being in the same subarray, by performing computation near-memory. However, no solutions are provided when the operands are in different higher levels of the memory subsystem.

However, in-memory processing does not always pay off. When the data is closer to the cores than to the main memory (e.g., L1 cache), it is usually more efficient to use the core than to move the data back to memory. This same principle corresponds to the concept of PIM-enabled-instructions (PEI), introduced by Ahn et al. [2]. Their work uses architectures with PIM support and does not require any modifications to the compiler. The main novelty of their work is a specialized hardware unit that decides whether the operation is to be issued to the memory processing unit or to dedicated accelerators near the core based on the locality of the operands.

Meanwhile, processing in cache approaches started to be adopted as an alternative to PIM solutions. Processing in cache also relies on the memory's internal bandwidth and avoids transferring data between the cache and the cores by performing the computation in-place. However, because the computation is performed in cache rather than in the main memory, it is not certain that the operands may be present, and the performance of such architectures is tightly coupled with the locality of the operands. Aga et al. [1] re-purpose the cache resources to perform active computation. Their architecture provides two independent computing mechanisms: (1) one *in-cache* meant for when the operands are memory aligned, and (2) another *near-cache* that is used when the operands are not aligned. By supporting ten distinct operations, this solution provides a programming mechanism to explicitly program the computing mechanisms in the cache. Another in-cache processing solution was proposed by Subramaniyan et al. [24]

to perform computation optimized for the Non-deterministic Finite Automata (NFA): the Cache Automaton. This approach relies on the SRAM cutting-edge cache technology, much faster than the DRAM, which allows achieving speedups up to $3840\times$ when compared with a conventional x86 CPU. Compiler support is also provided, which automates the process of mapping real-world NFAs to Cache Automaton.

However, despite the important contributions of the previous proposals, these architectures mainly rely on bitwise operations to perform computation in-memory/in-cache. This reduces the scope of what can be achieved as they hardly support the implementation of arithmetic, shift, and more complex operations. In contrast, the proposed CCS offers an ISA with 48 distinct instructions, it is not limited by the size of the cache lines (as it implements hardware loops) and it also supports a limited range of strides in the operands.

VI. Conclusions

A novel Cache Compute System (CCS) was introduced to allow data processing and manipulation directly on-cache, avoiding moving it from the memory hierarchy to the processor core just to perform simple operations (such as map and reduce). Furthermore, the developed CCS allows the massive exploitation of parallelism and eliminates the need for software-side loop control, which increases even further its efficiency. It also provides support for hardware loops, in case the vector operands are longer than cache lines. The proposed CCS was validated by simulation using gem5, compared with a PULPino soft-core, and integrated with an MB-Lite CPU using a memory subsystem consisting of a single level cache and a main memory built with the FPGA native BRAMs. Considering the results obtained with MB-LITE, for operations that directly map to CCS commands, a maximum speedup of $408\times$ (for 1024 32-bit word vector operands) is shown. When executing real-world application benchmarks (kNN, Matrix Multiplication, and Linear Regression) the obtained speedups range from $3\times$ to $68\times$. The achieved energy efficiency improvements are also remarkable and can be as high as $328\times$ for operations that directly map to CCS.

References

[1] Shaizeen Aga et al. Compute caches. In *HPCA*, pages 481–492. IEEE Computer Society, 2017.

[2] Junwhan Ahn et al. Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In *ISCA*, pages 336–348. ACM, 2015.

[3] Junwhan Ahn et al. A scalable processing-in-memory accelerator for parallel graph processing. In *ISCA*, pages 105–117. ACM, 2015.

[4] Amirali Boroumand et al. Lazypim: An efficient cache coherence mechanism for processing-in-memory. *Computer Architecture Letters*, 16(1):46–50, 2017.

[5] Wei-Hao Chen et al. A 16mb dual-mode reram macro with sub-14ns computing-in-memory and memory functions enabled by self-write termination scheme. In *Electron Devices Meeting (IEDM), 2017 IEEE International*, pages 28–2. IEEE, 2017.

[6] Ming Cheng et al. TIME: A training-in-memory architecture for memristor-based deep neural networks. In *DAC*, pages 26:1–26:6. ACM, 2017.

[7] Ping Chi et al. PRIME: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *ISCA*, pages 27–39. IEEE Computer Society, 2016.

[8] Guohao Dai et al. Graphh: A processing-in-memory architecture for large-scale graph processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[9] Reetuparna Das. Blurring the lines between memory and computation. *IEEE Micro*, 37(6):13–15, 2017.

[10] Daichi Fujiki et al. In-memory data flow processor. In *PACT*, page 375. IEEE Computer Society, 2017.

[11] Lei Han et al. A novel reram-based processing-in-memory architecture for graph computing. In *NVMSA*, pages 1–6. IEEE, 2017.

[12] Sangwoo Han et al. PIM architecture exploration for HMC. In *APCCAS*, pages 635–636. IEEE, 2016.

[13] Tamar Kranenburg et al. MB-LITE: A robust, light-weight soft-core implementation of the microblaze architecture. In *DATE*, pages 997–1000. IEEE Computer Society, 2010.

[14] Shuangchen Li et al. Pinatubo: a processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *DAC*, pages 173:1–173:6. ACM, 2016.

[15] Shuangchen Li et al. DRISA: a dram-based reconfigurable in-situ accelerator. In *MICRO*, pages 288–301. ACM, 2017.

[16] Lifeng Nai et al. Graphpim: Enabling instruction-level PIM offloading in graph computing frameworks. In *HPCA*, pages 457–468. IEEE Computer Society, 2017.

[17] Agathoklis Papadopoulos et al. Towards systolic hardware acceleration for local complexity analysis of massive genomic data. In *ACM Great Lakes Symposium on VLSI*, pages 339–344. ACM, 2012.

[18] Ashutosh Pattnaik et al. Scheduling techniques for GPU architectures with processing-in-memory capabilities. In *PACT*, pages 31–44. ACM, 2016.

[19] Vivek Seshadri et al. Fast bulk bitwise AND and OR in DRAM. *Computer Architecture Letters*, 14(2):127–131, 2015.

[20] Vivek Seshadri et al. Ambit: in-memory accelerator for bulk bitwise operations using commodity DRAM technology. In *MICRO*, pages 273–287. ACM, 2017.

[21] Linghao Song et al. Graphr: Accelerating graph processing using reram. In *HPCA*, pages 531–543. IEEE Computer Society, 2018.

[22] Fang Su et al. A 462gops/j rram-based nonvolatile intelligent processor for energy harvesting ioe system featuring nonvolatile logics and processing-in-memory. In *VLSI Circuits, 2017 Symposium on*, pages C260–C261. IEEE, 2017.

[23] Arun Subramaniyan et al. Cache automaton. In *MICRO*, pages 259–272. ACM, 2017.

[24] Arun Subramaniyan et al. Parallel automata processor. In *ISCA*, pages 600–612. ACM, 2017.

[25] Andreas Traber et al. Pulpino: A risc-v based single-core system. In *OpenRISC Conference, ORCONF2015*. ORCONF2015, 2015.

[26] Andreas Traber et al. Pulpino: A small single-core risc-v soc. In *3rd RISCV Workshop*, 2016.

[27] UCI Center for Machine Learning and Intelligent Systems. Letter Recognition Data Set, 1991.

[28] UCI Center for Machine Learning and Intelligent Systems. Wine Data Set, 1991.

[29] UCI Center for Machine Learning and Intelligent Systems. Statlog (Landsat Satellite) Data Set, 1993.

[30] UCI Center for Machine Learning and Intelligent Systems. Wholesale customers Data Set, 2014.

[31] Yi Wang et al. Towards memory-efficient allocation of cnns on processing-in-memory architecture. *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[32] William A. Wulf et al. Hitting the memory wall: implications of the obvious. *SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[33] Salessawi Ferede Yitbarek et al. Exploring specialized near-memory processing for data intensive operations. In *DATE*, pages 1449–1452. IEEE, 2016.