Exploiting Compute Caches for Memory Bound Vector Operations

João Vieira*, Paolo Ienne†, Nuno Roma*, Gabriel Falcao‡, Pedro Tomás*

*INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal
†École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
‡Instituto de Telecomunicações, Universidade de Coimbra, 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 was validated by integrating it with an MB-Lite soft-core 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 4x to 408x, and energy efficiency gains from 6x to 328x.**

***Index Terms*—Compute caches, Memory bound operations, Vectorization**

I. Introduction

Memory-bound applications usually transfer large amounts of data to the processing core just to apply rather simple transformations such as map and reduce. This renders useless the CPU resources as the memory subsystem cannot feed the data at the desired rate.

Given the low complexity of such operations, Processing in Memory (PIM) approaches, which move the computation closer to the operands natural location have become increasingly popular [2], [5]–[8]. Recently, some similar solutions based on pushing the processing to the cache subsystem have also been proposed [1], [3], [9], [10]. 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. However, such solutions are often limited to a few elementary bitwise logic operations, constraining its adoption for real-world applications. Furthermore, this type of technology only allows map-type operations.

In this paper, an ambitious acceleration infrastructure is proposed by integrating a stream vector compute unit within a conventional cache: the Cache Compute System (CCS). The developed vector compute unit supports 48 arithmetic, shift and logic operations that match most common parallel processing patterns, namely, map and reduce.

In accordance, this manuscript presents the following novel aspects and contributions:

- architecture of a cache compute unit that implements 48 arithmetic, shift and logic vector operations;
- feasible mechanism that takes advantage of long cache lines and operands locality, exploiting massive parallelism and avoiding moving data to the processing core;
- simple framework to program and control the CCS;
- independent operation (regarding the aggregated CPU), allowing the CPU to concurrently operate over other data.

Each one of the previous features will be explained in detail in the next section, followed by experimental results and some conclusions.

II. Cache Compute system architecture

The proposed CCS is depicted in Figure 1. It is defined as a logical block that includes a conventional cache and the tightly interconnected cache compute unit.

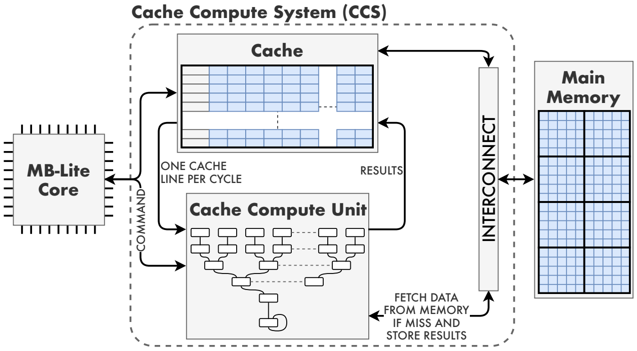

Fig. 1. System overview integrating the processor, the CCS and the main memory. The CCS is defined as the logic block that includes the cache and the cache compute unit.

From the perspective of the CPU, the CCS behaves like a memory device, while from the memory side it acts 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 cache compute unit programming; the CCS has

978-1-5386-7769-8/18/$31.00 ©2018 IEEE
DOI 10.1109/SBAC-PAD.2018.00041

two interfaces to the memory, one for the cache system and another for the cache compute unit. Within the CCS, the cache has two memory interfaces: one to the processor and another to the cache compute unit. The processor and the CCS operate independently, and while the cache compute unit reads the operands or writes the results of a command, the processor is free to operate over other data.

Since the processor has a single interface to the CCS, the cache compute unit control registers are memory-mapped in the processor's addressing space. To program the cache compute unit, the processor simply writes its control registers and signals the unit to start. One of the cache compute unit control registers indicates its readiness, making it possible to know when it is idle and when the last issued command has finished. The other programming registers provide the CCS with information about: the command ID, the operands length, an optional constant, the start addresses of the operands and the result, the stride and the compute mask. Finally, there is also a programming register responsible for starting the execution.

The devised CCS supports 48 arithmetic, shift and logic vector operations that were chosen by taking into consideration algorithms used in a wide range of application domains including clustering, machine learning, cryptography, image processing, biomedicine, and linear algebra.

A. Data processing structure

Figure 2 depicts the integration of the cache compute unit within the CCS. To allow the implementation of both map and reduce operations, the cache compute unit 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; the remaining levels adopt 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 cache compute unit have to support more operations (e.g., the second level of the cache compute unit 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.

B. Operands fetch and result storage

The cache compute unit operands and result are described by 1-D descriptors composed of base address, stride, and length. To execute a given operation, the operands are divided into groups of N elements, where N corresponds to the vector unit length. The cache compute unit loads each one of these partitions in a sequential fashion: for VOP2 commands, it first loads one operand, storing it in an input buffer, before proceeding to the second operand; for VOP1 and VCOP, it skips the second operand and iteratively fetches data from a single data input stream. When fetching data, the cache compute unit uses 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 forwarded to the cache compute unit and are not stored in the cache. This avoids cache pollution, by reducing conflicts between cache compute unit and processor data. To store the command output, the CCS again relies on the cache existing structures and policies for data allocation and write.

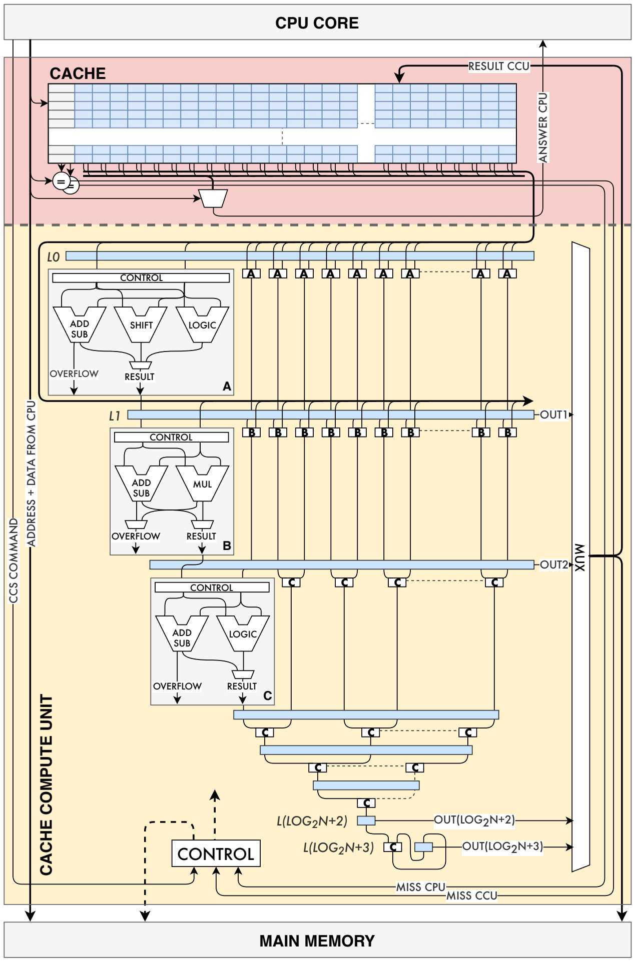

Fig. 2. Datapath of the cache compute unit. The first two levels of the binary structure are meant for maps and the remaining structure acts as a reduction tree. Consequently, different levels are equipped with different units: type-A unit–implements an adder/subtracter, a shift unit and a logic unit; 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.

C. 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 need to have the same alignment in the cache for the command to succeed. For striding and alignment correction purposes, two 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. The stride mask is generated by software and each one of its bits is set to one when the index is multiple of the stride. In the CCS, both masks are combined, generating the execution mask, that determines which elements of the operands will be considered when executing the command, and which results are to be stored in memory. Map-type commands only need the execution mask, but reduction-type commands require a submask to be generated at each level of the reduction, determining which elements are to be reduced to the next level.

D. CCS programming and operation

The implemented communication protocol allows the CPU to interact with the CCS for three main purposes: (1) to program a command, (2) to order the start of 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 cache compute unit are mapped. To ease this procedure, a framework was developed in order to release the programmer from the need of understanding 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 commands:

- `__ccs_setup(<args>)`: programs the CCS with the required parameters;
- `__ccs_start()`: signals the CCS to start the execution of the command previously programmed;
- `__ccs_check()`: (nonblocking) checks if the CCS is ready to execute and returns a logic value;
- `__ccs_wait()`: blocks execution until the CCS is ready to accept a new command.

Independently of the considered commands, the operand length is not limited by the number of functional units of the first level. Instead, larger operands can be specified and hardware loops will process them in parts until they are all consumed. For map-type commands, each partition of the operands is fetched, processed and stored independently, while for reduce-type commands, the partitioned operands are fetched in sequence and the execution is pipelined.

III. Experimental results

The proposed CCS was prototyped in a Xilinx VC709 Development Board equipped with an XC7VX690T Virtex-7 FPGA, using Vivado 2018.1 software suite.

The base system considered an MB-Lite soft-core [4] with a memory subsystem composed by: (1) the main memory, implemented with Field-Programmable Gate Array (FPGA) native Block RAMs (BRAMs), and (2) a single level cache. The cache was designed to be direct-mapped with write-through and write-not-allocate policies from the CPU point of view, and write/read-not-allocate from the cache compute unit perspective. The cache is composed of 16 lines, each with 2048-bits, plus the tag and one validity bit per line. The size of the implemented cache compute unit 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).

Table I presents the required resources, frequency of operation and power of the base system, the proposed CCS, and the complete system. The presented values show that the proposed CCS increases the hardware resources requirements, which is justified by its greater amount of functional units. Nevertheless, the system still uses less than 25% of the total resources available in the considered device.

To measure the performance and energy efficiency improvements relative to the base system, a set of microbenchmarks was considered. For each microbenchmark, its sequential equivalent was run in the MB-Lite and the execution time was measured. All results were analyzed independently of the operating frequency, being the latencies measured in clock cycles.

As shown in Figure 3, the use of the CCS provides considerable speedups ranging from $4\times$ (INITC) to $66\times$ (ROLVV, RORVV), and energy efficiency improvements between $6\times$ and $90\times$. This is mainly due to three factors explored by the CCS: (1) intrinsic parallelism of the commands, (2) in-place processing of data, and (3) mitigation of the loop control required by the processor. Furthermore, both speedups and energy efficiency improvements scale with the size of the operands, as shown in Figure 4. In particular, for 32-bit vectors of 1024 elements, the speedups scale up to $408\times$.

As shown in table I, the dynamic power requirements of the CCS are significantly higher than the equivalent for the MB-Lite. However, due to the order of magnitude of the achieved speedups, there are still significant energy savings. For 1024 32-bit vector operands, the energy efficiency improvements can be as high as $328\times$.

To complement this analysis, three different benchmarks were adapted to use the proposed CCS: the k-Nearest Neighbors, the integer Matrix Multiplication and the Linear Regression. The overall speedups obtained for these three algorithms reach $68\times$, $54\times$ and $3\times$, respectively, as shown in Figure 5. While the parallelized phases of the Matrix Multiplication and

TABLE I
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.

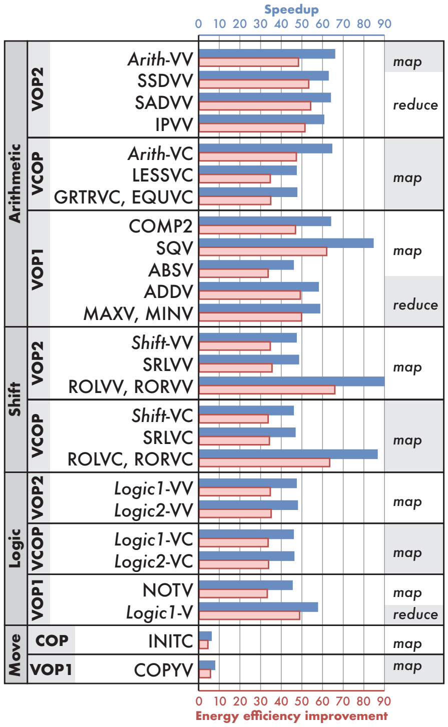

Fig. 3. Speedup and energy efficiency improvement provided by the CCS when compared to equivalent sequential versions of the CCS commands running in the MB-Lite. For comparison purposes, the execution latencies were measured in clock cycles. The commands are classified accordingly to different criteria: (1) Mathematical operation: arithmetic, logic, shift, or move; (2) Operands: two vectors (VOP2), a vector and a constant (VCOP), a single vector (VOP1), or only a constant (COP); (3) Functional operation: map or reduce. The designated groups of commands include: *Arith*: ADD, SUB, MUL; *Shift*: SLL, SLA, SRA; *Logic1*: AND, OR, XOR; *Logic2*: NAND, NOR, XNOR. The operations SSDVV, SADVV and IPVV represent sum of square differences, sum of absolute differences and dot product. The commands LESSVC, GRTRVC and EQUVC produce a mask (same size of the vector operand) with the result of the evaluated condition (less than, greater than or equal to).

the k-Nearest Neighbors consist of a single CCS command, the Linear Regression sequentially takes 5 distinct commands, which not only increases the programming overhead but also does not allow to fully explore the CCS pipeline capabilities.

IV. Conclusions

A novel Cache Compute System (CCS) was introduced to allow data processing and manipulation directly on-cache, avoiding moving data 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 integrated with an MB-Lite soft-core using a memory subsystem consisting of a single level cache and a main memory built with the FPGA native BRAMs. For operations that directly map to CCS commands, the obtained results show a maximum speedup of 408× (for 1024 32-bit word vector operands). When executing real-world application benchmarks (kNN, Matrix Multiplication, and Linear Regression) the obtained speedups range from 3× to 68×. The achieved energy efficiency improvements are also remarkable and can be as high as 328× for operations that directly map to CCS commands.

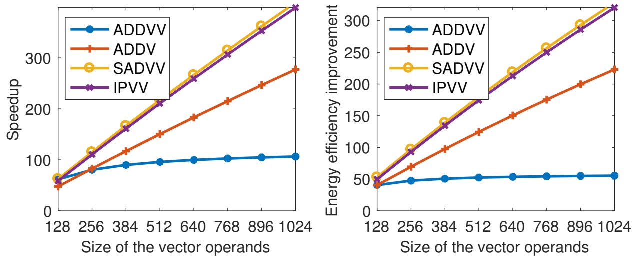

Fig. 4. Obtained speedups and energy efficiency improvements with 4 CCS commands when compared to their sequential versions executing in the MB-Lite for different vector operands sizes.

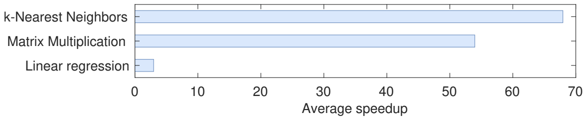

Fig. 5. Obtained speedups when running three application benchmarks partially executed in the CCS: the k-Nearest Neighbors (with 64 coordinates per sample, $k = 4$, 64 control samples and 1 sample to classify), the integer Matrix Multiplication (64 × 64 matrices), and the Linear Regression (64 2D points).

Acknowledgements

The authors would like to acknowledge the financial support from Fundação para a Ciência e a Tecnologia (FCT) under projects UID/CEC/50021/2013, UID/EEA/50008/2013, and PTDC/EEI-HAC/30485/2017 (HAnDLE).

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] Reetuparna Das. Blurring the lines between memory and computation. *IEEE Micro*, 37(6):13–15, 2017.
[4] 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.
[5] 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.
[6] Shuangchen Li et al. DRISA: a dram-based reconfigurable in-situ accelerator. In *MICRO*, pages 288–301. ACM, 2017.
[7] Vivek Seshadri et al. Fast bulk bitwise AND and OR in DRAM. *Computer Architecture Letters*, 14(2):127–131, 2015.
[8] Vivek Seshadri et al. Ambit: in-memory accelerator for bulk bitwise operations using commodity DRAM technology. In *MICRO*, pages 273–287. ACM, 2017.
[9] Arun Subramaniyan et al. Cache automaton. In *MICRO*, pages 259–272. ACM, 2017.
[10] Arun Subramaniyan et al. Parallel automata processor. In *ISCA*, pages 600–612. ACM, 2017.