PUMICE: Processing-using-Memory Integration with a Scalar Pipeline for Symbiotic Execution

Socrates S. Wong
Computer Systems Laboratory
Cornell University
Ithaca, NY 14853 USA
ssw96@cornell.edu

Cecilio C. Tamarit
Computer Systems Laboratory
Cornell University
Ithaca, NY 14853 USA
ct652@cornell.edu

José F. Martínez
Computer Systems Laboratory
Cornell University
Ithaca, NY 14853 USA
martinez@cornell.edu

***Abstract*—Existing SIMD extensions in scalar CPUs (e.g., SSE, AVX, etc.) can leverage instruction-level parallelism (ILP) because of their tight integration with the CPU pipeline. However, the vectors they employ are quite short, and this limits their ability to exploit data-level parallelism (DLP). On the other hand, processing-using-memory (PUM) accelerators are capable of exploiting massive amounts of DLP, as they typically perform computation on very long vectors (tens of thousands of elements) within the memory itself. Recent work demonstrates that order-of-magnitude speedups can be achieved by these architectures for a variety of workloads over area-equivalent multicore CPUs with SIMD extensions. Still, PUM architectures are largely decoupled from the CPU itself, thereby limiting their ability to tap the CPU's ILP the way SIMD extensions do.**

In this paper, we propose PUMICE, a tightly integrated CPU-PUM architecture that simultaneously exploits DLP and ILP for very long vector operations. As a result of this tight integration, PUMICE delivers significant performance gains: Our experimental results show speedups of up to 2.2× (1.4× on average) over a state-of-the-art decoupled approach.

***Index Terms*—Associative processing, associative memory, vector processors**

I. Introduction

Processing-in-memory (PIM) has gained attention in the last few years as a way to tackle the von Neumann bottleneck by bringing computational units closer to data. Among them, processing-using-memory (PUM) is an *in situ* approach that focuses on performing operations within the memory structure itself. Notable recent work in general-purpose PUM involves bitline computation [7], [10] and associative processing [1], [3], [4], [6], [13]. Evaluations of these solutions report very large speedups, as they exploit massive data-level parallelism (DLP) over very long vectors of data (tens of thousands of elements).

One limitation of these proposals is that they are envisioned as largely decoupled accelerators that process one vector instruction at a time, with no meaningful support for instruction-level parallelism (ILP). At the same time, SIMD short-vector extensions (e.g., Intel's SSE or AVX) in general-purpose CPUs are typically tightly integrated with the CPU pipeline, and this enables them to exploit ILP. However, SIMD extensions are very limited in the amount of DLP they can exploit, primarily because of their small vector sizes.

This paper presents the first general-purpose architecture that tightly integrates a CPU with a PUM-based very long vector unit. The resulting processing element can exploit DLP and ILP simultaneously and highly effectively. Our results show speedups for the Phoenix benchmark suite of up to 2.2× (1.4× on average) over PUM architectures that follow a decoupled accelerator model. This represents a speedup of over two orders of magnitude over a server-class CPU with SIMD extensions.

The main contributions of this paper are:

1) An analysis of the performance limitations that adopting a decoupled accelerator model brings to PUM very long vector architectures.
2) A microarchitectural approach to tightly integrate a PUM-based very long vector architecture and a CPU to exploit both DLP and ILP.
3) An experimental evaluation of the performance of our integrated approach compared against a state-of-the-art decoupled PUM accelerator.

II. Background

A. Processing-using-Memory

Processing-using-memory (PUM) leverages memory structures to perform computation in place, with the goal of extracting large amounts of data-level parallelism (DLP) efficiently. Some general-purpose PUM approaches [1], [6] are based on bitline computation [7], [10], relying on the existing precharging and sensing circuitry to perform bitwise logic operations in place, serially across the bits of an element, and in parallel over many elements across bitlines. Associative processing (AP), on the other hand, is a PUM architecture that makes use of CAM arrays to perform bulk searches and

Algorithm 1 AP-based Vector OR (vor.vv v3, v1, v2)

Input: `v1`, `v2` ▷ Vector operands
Output: `v3` ▷ Output vector
1: **Search** 1 in `v1` ▷ Set tag bits if v1[i] == 1
2: **Search-Accum** 1 in `v2` ▷ Set tag bits if v2[i] == 1
3: **Update** v3 ← `tags` ▷ Set v3 to tags

This work was supported in part by the Semiconductor Research Corporation (SRC) through the Center for Research on Intelligent Storage and Processing-in-memory (CRISP, part of the JUMP program) and the ACE Center for Evolvable Computing (ACE, part of the JUMP 2.0 program), both co-sponsored by DARPA; and by the NSF Science and Technology Center for Programmable Plant Systems (CROPPS, award #2019674). Cecilio Tamarit was also partially sponsored by the Fulbright Program.

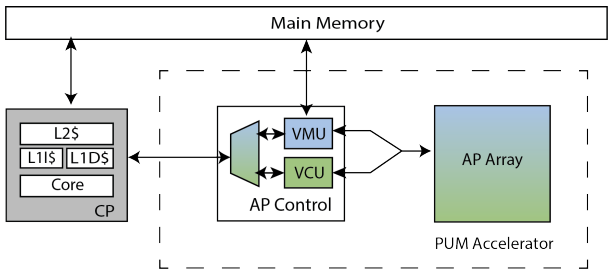


Fig. 1: The CAPE very long vector PUM architecture [4]. The control processor (CP) is a tiny in-order RISC-V CPU with standard vector extensions. Vector operations are relayed to the AP Array (also referred to as Compute-Storage Block, or CSB) through the Vector Control Unit (VCU), which converts them into sequences of PUM microinstructions. A Vector Memory Unit (VMU) facilitates data movement between CAPE and main memory.

updates that effectively apply a truth table, often bit-serially over thousands of elements at once [3], [4], [13]. Algorithm 1 shows the example of a vector OR instruction, implemented as a bit-parallel associative algorithm. Assuming vectors with tens of thousands of 32-bit integers, in lines 1 and 2, all bits of every element are checked simultaneously to determine whether their value is 1, and a tag is set for every match. In line 3, following the truth table of the OR operation, the output vector is set to the result of the searches. Unlike AP logic instructions, AP arithmetic instructions (not shown in Algorithm 1) process one bit at a time. However, being able to operate simultaneously on tens of thousands of elements more than makes up for their bit-serial nature.

Recent AP architecture proposals [3], [4], [13] execute such algorithms over massive amounts of data in parallel (tens to hundreds of thousands of elements). Works like Hyper-AP [13] focus on exploring AP array-level optimizations for CMOS and emerging memory technologies, whereas works like CAPE [3], [4] explore full-system implementations. At a high level, CAPE leverages the RISC-V length-agnostic standard vector extensions [12] to provide the programmer with *direct* access to very long vector processing capabilities with native ISA support, without the use of specific APIs or accelerator-specific programming models. Both approaches yield very high performance running a broad variety of applications.

At the architecture level, PUM-based very long vector architectures are ultimately subordinated to the CPU, which offloads vector instructions to a decoupled accelerator. A control unit translates the instructions into a series of bulk searches and updates, to be performed on the AP array.

B. Vector architectures

Vector and SIMD architectures both have the capability to exploit data-level parallelism by operating over multiple vector elements simultaneously, the main difference being flexibility in terms of bitwidth and vector length. Traditional long vector architectures (typ. 32-128 elements) have been popular in the context of high-performance computing (HPC), and commodity off-the-shelf microprocessors often boast short vector SIMD extensions (e.g., Intel AVX) able to operate over a small amount of elements simultaneously by means of replicating existing pipeline resources.

Previous works have studied the importance of decoupling memory and computation in traditional long vector architectures to increase ILP, as well as operand chaining of intermediate results for decreased data movement [5]. There have been attempts to integrate traditional long vector architectures with microprocessor pipelines [8], and more recent work studies the energy efficiency potential of such an approach in the mobile landscape [11]. In this paper, we study the integration of *PUM-based very long vector architectures* (tens of thousands of elements) with a scalar processor pipeline. To the best of our knowledge, this is the first proposal to integrate tightly a CPU and a massively parallel PUM-based vector unit.

III. Bottleneck Analysis

In this section, we lay out the limitations that the decoupled accelerator model imposes on very long vector PUM architectures. To do so, we model two different design points based on the CAPE architecture with the parameters specified in its original publication [4].

A. Experimental setup

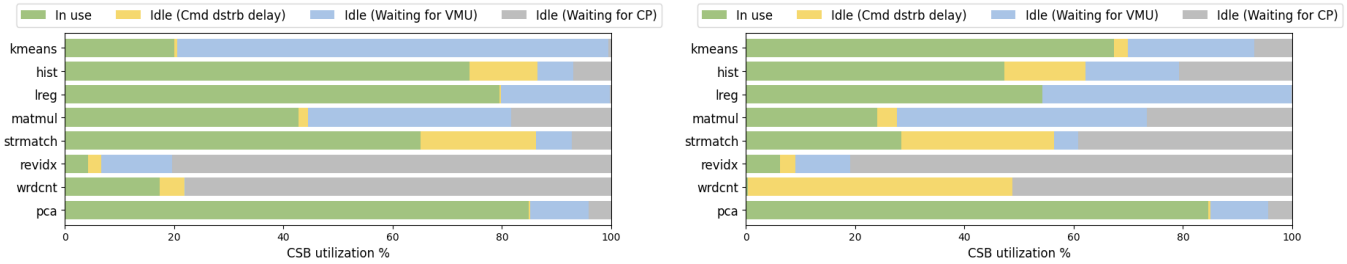
We replicate the gem5 cycle-approximate simulation environment described in the original CAPE paper [4] to model the CAPE PUM architecture and run the same workloads as the original CAPE work (the entire Phoenix benchmark suite [9]), with the input sizes specified in Table II. We run experiments with the two AP array sizes in the original CAPE paper, 32k and 131k vector elements, and the corresponding delays. We then break down whole-program execution cycles into four categories depending on AP array (in)activity. The results for both CAPE32k and CAPE131k are shown in Figure 2.

TABLE I: Control processor and memory system

Core configuration	2-issue in-order core, 2.7GHz 4/1/1/1 Int/FP/Mem/Br units 5+5-entry LSQ Tournament BP, 4,096-entry BTB, 16-entry RAS
L1 D/I cache	32 kB, 8-way, 64 B block, LRU 2-cycle tag/data latency
L2 cache	1 MB, 16-way, 512 B block, LRU 14-cycle tag/data latency
Memory	4H HBM, 8 channels 16 GBps/512 MB (per channel)

TABLE II: Description of the evaluated workloads

Application	Input Size	O3CPU Runtime
lreg	500 MB	$4.4 \cdot 10^9$ cycles
hist	1.4 GB	$1.4 \cdot 10^{10}$ cycles
kmeans	100 k	$5.0 \cdot 10^9$ cycles
matmul	1,000×1,000	$7.0 \cdot 10^9$ cycles
pca	1,500×1,500	$1.7 \cdot 10^{10}$ cycles
strmatch	500 MB	$6.8 \cdot 10^{10}$ cycles
wrdcnt	10 MB	$4.9 \cdot 10^9$ cycles
revidx	100 MB	$6.0 \cdot 10^8$ cycles



(a) CAPE (32k elements)

(b) CAPE (131k elements)

Fig. 2: Per-workload breakdown of the AP Array utilization for two PUM accelerator configurations of different sizes running the Phoenix benchmark suite. Green indicates the AP is in use, yellow or blue that it is waiting for commands or data that are being offloaded to it, and gray that it is waiting for the scalar CPU.

B. Discussion

Figure 2 depicts the utilization of the PUM array for the two baseline accelerator designs when running their respective benchmarks. It is clear that the AP array is being underutilized across all workloads, and in most cases severely so. This is due to three main issues: On the one hand, moving data introduces significant delays, whether due to (1) load/store instructions (blue) or (2) the control signals themselves (yellow). On the other hand, a non-negligible amount of cycles is attributed mostly to the nature of the workloads, but also to (3) stalls in the scalar processor (gray).

This suboptimal utilization leads to an effective performance that is far lower than the theoretically achievable limit permitted by the memory bandwidth and the computational throughput. Indeed, an associative PUM architecture like CAPE will mitigate the von Neumann bottleneck because increasingly larger arrays enable massively parallel computation with far less data movement after the initial compulsory access. However, at these utilization rates and with this degree of decoupling, the true potential of such an approach remains untapped.

By mitigating the effect of the bottlenecks described in this section, the worst-case scenario (`revidx`) shows potential for up to 4x performance improvement if a better utilization of the associative processor can be achieved, provided the given workload possesses the required degree of data-level parallelism. In the following sections, we describe the design and evaluate the effectiveness of an architecture that aims to achieve a higher degree of ILP by means of tight integration, enabling the overlap of AP array computation cycles with command distribution and data transfer latencies.

IV. Proposed Architecture

We have empirically demonstrated how PUM very long vector architectures that follow the accelerator model suffer from severe underutilization. In this section, we introduce PUMICE, an architecture that tackles this issue by means of tight integration of a PUM very long vector unit with a general-purpose CPU pipeline. This reduces data movement, ALU and control complexity, and maximizes both DLP and ILP to increase performance.

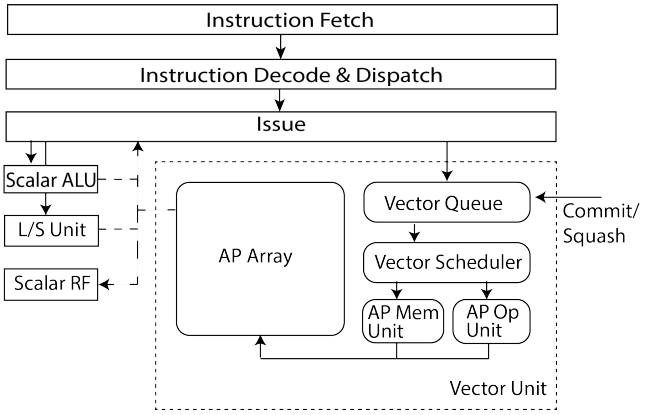


Fig. 3: PUMICE architecture diagram. The associative processor (AP Array) and its related structures are integrated into the pipeline as a Vector Unit, as opposed to the more conventional decoupled accelerator model.

A. Overview of the pipeline

We envision integrating the PUM very long vector unit as a functional unit that requires its own scheduling logic. Furthermore, the CPU pipeline has to support mixed speculation: Scalar instructions can be executed speculatively while vector instructions cannot, as any rollback would be too costly due to the sheer length of the vectors. To accommodate this, we propose a Vector Queue to interface with the Vector Unit and guarantee that it will only schedule vector instructions once the architectural state can be updated safely. Once dependencies are resolved, the committed instructions in the Vector Queue are scheduled by the Vector Scheduler. Unlike previous PUM accelerator designs, the queueing and scheduling capabilities enable two important behaviors of traditional long vector architectures that are not present in proposed PUM designs and improve instruction-level parallelism:

(1) *Operand chaining and back-to-back execution*, by allowing several vector instructions to be in flight simultaneously, so that they can be executed once their operands are ready. This is in contrast to previous decoupled PUM accelerators that process them one at a time. The goal is to hide the command distribution delay (Figure 2, yellow) that

arises when the scalar CPU offloads the instructions to the accelerator.

(2) *Decoupling of memory and computation*, achieved by dividing the control of the PUM array in two separate units: The AP Operation Unit receives all non-memory operations from the scheduler and translates them into the microcode that will drive the PUM array so that it performs the searches and updates that implement said operation. Concurrently, the AP Memory Unit receives memory operations and interfaces with main memory in a decoupled manner, buffering the loads to (and stores from) the PUM array while it is busy performing other operations (Figure 2, blue).

Overall, these two mechanisms allow operations of all kinds to execute back-to-back, memory and computation to overlap, and also the running ahead of independent arithmetic operations during the long latencies of memory accesses (Figure 2, overlapping blue with green).

B. Mixed speculation

Integrating a very long vector unit involves supporting four classes of instructions that interact with the pipeline and each other in a variety of ways. Most importantly, the scalar processor will have to handle some of them non-speculatively while retaining speculation in some aspects for others:

(1) *Scalar read and write*: These are the scalar-only instructions from the base ISA. Their execution in the CPU pipeline remains unchanged, and speculative execution is permitted as usual.

(2) *Scalar read and write with vector/status register read*: Some RISC-V vector instructions that fall in this category include `setvl`, `vextract`, and `setvb`, which produce a single return value that is sent to the scalar pipeline. Speculative execution is also safe. This is due to the fact that all modifications to the microarchitectural state occur in the scalar side of the pipeline, which can readily support checkpointing/rollback mechanisms to deal with exceptions without as much of an impact in area, performance, and energy. However, all vector instructions must be dispatched non-speculatively to the queue in the Vector Unit. As many of the produced values are often used by later scalar instructions or resolving branches, by allowing some of the values produced by these instructions to be utilized speculatively we prevent stalls present in the accelerator model, where the CPU has to wait for said instruction to finish its execution and send back the result.

(3) *Scalar read, vector register write*: These are the vector instructions that take a scalar value as an input. An example of this would be `vadd.vx`, which adds a scalar value to all elements in a vector. This type of instruction is not issued until all the scalar values it depends on are ready, and then it is kept in the Vector Queue until the scalar instructions it depends on are retired. This queue is similar to a store queue in a traditional processor, and will squash any pending vector instructions when branch mispredictions or exceptions occur.

(4) *Non-scalar, vector register write*: Considered to be a specialization of the latter, with the major difference being that they do not have a scalar dependency and thus can be issued immediately to the Vector Queue. Examples of this would be vector bitwise and arithmetic instructions, such as `vand.vv` or `vadd.vv`.

V. Evaluation

In this section, we describe our experimental methodology and analyze the results we obtained for PUMICE, the integrated architecture we proposed in Section IV.

A. Methodology

As was the case in Section III, we utilized gem5 [2], a cycle-approximate simulator, with the same configuration described in Table I, to simulate PUMICE over the two PUM accelerator design points based on the orginal CAPE paper [4]. We then evaluate our architecture as in the bottleneck analysis, with the same set of workloads from the Phoenix benchmark suite, again with the specific parameters and baselines shown in Table II. Furthermore, we use two small microbenchmarks to highlight the main individual benefits of our proposal separately.

B. Results

1) Microbenchmarks: We use two small microbenchmarks to study the impact of our architecture on computational throughput and the degree of compute-memory overlap that can be achieved.

Vector Throughput Microbenchmark — The vector throughput microbenchmark measures the number of cycles it takes for the evaluated architectures to execute 1,000 of the instructions listed. The microbenchmark contains ten vector arithmetic or bitwise logic operations. The instructions are executed in a loop; therefore, this is a measure of how efficient each of the architectures is at issuing and completing the instructions.

TABLE III: Vector Throughput Microbenchmark Results (cycles per 1000 instructions)

Instr.	CAPE32k	PUMICE32k	CAPE131k	PUMICE131k
vadd.vv	259,157	257,026	260,158	257,027
vmul.vv	3,044,157	3,042,026	3,045,158	3,042,027
vsub.vv	259,157	257,026	260,158	257,027
vand.vv	6,157	4,026	7,158	4,027
vor.vv	6,157	4,026	7,158	4,027
vxor.vv	7,157	5,026	8,194	5,027

The results in terms of vector throughput match our previous observation (see Section III) that, depending on the workload, the benefit that can be gained by hiding the command distribution delay and by increasing concurrency between the VU and the CP in general can be significant for operations such as `vor` and `vand`, which result in up to 43% fewer cycles needed for completion. However, as expected, it also reveals that for longer instructions such as `vmul` it can have a negligible impact.

Mixed Load-Compute Microbenchmark — To test the overlap of memory and computation enabled by PUMICE, the following microbenchmark executes different proportions of memory and computation instructions back-to-back. In particular, we run 1,000 random vector instructions, which can be vector multiplications (`vmul`) or vector load instructions

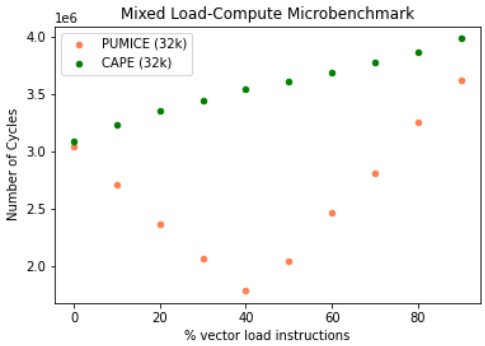


Fig. 4: Mixed Load-Compute microbenchmark results.

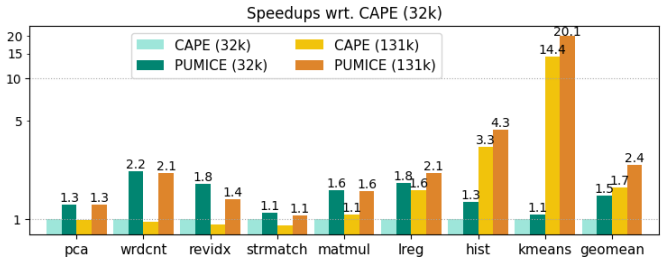


Fig. 5: Performance of the Phoenix benchmarks when ran on our two proposed architectures, normalized to CAPE32k.

(`vld`). The relative amount of each of the instructions varies with each iteration.

As shown in the plot in Figure 4, PUMICE performs better when there is a similar amount of vector memory and compute operations. However, at the extreme end of either spectrum, the performance of PUMICE becomes similar to that of CAPE– i.e., no overlap between memory and computation.

2) Phoenix Benchmark Suite: The Phoenix Benchmark Suite was also used to evaluate the CAPE [4] architecture, the only difference in our setup being the newer compiler and that we perform manual loop unrolling to reduce branch penalties and help expose the available ILP to the hardware. Considering this, in Figure 5 we can observe that we were able to largely replicate the CAPE work, modeling their two different design points in terms of vector length: A 32k-element design, area-equivalent to a single out-of-order scalar core, and a longer 131k-element variant, comparable to 2 cores. As in the original work, we observe that when scaling the CAPE baseline from 32k to 131k AP Array size, greater performance is not guaranteed. For applications like `wrdcnt`, `revidx`, or `strmatch` it can actually decrease. On the other hand, other programs like `kmeans` and `hist` see very significant speedups. The most aggressive CAPE design obtains a 1.7x speedup over its smaller counterpart, which was already reported to be 14x (up to 254x) faster than an out-of-order CPU [4].

With the smallest of our proposed designs, when looking at the results for PUMICE32k, we observe an average speedup of 1.5x over the CAPE32k baseline. As expected, `revidx` and `wrdcnt`, the applications where the AP Array starved the most due to unnecessary idling and the effect of command distribution delay (see Figure 2) are also among the ones that see the most benefit, with performance increases of 1.8x and 2.2x, respectively. This result is the opposite of the previous situation, where it seemed like running these workloads on (increasingly large) CAPEs was not beneficial or even detrimental, confirming that their low performance was indeed not only due to the nature of the applications and their memory access patterns, but also because of the inability of CAPE32k to exploit the intrinsic ILP of those specific workloads.

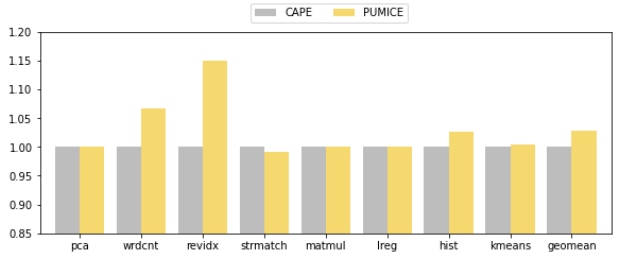
In the case of the larger PUMICE configuration with an AP Array size of 131k, we observe consistent increases too, especially in the workloads most sensitive to this. We thus conclude that **even though the command distribution delay grows with the size of the AP array, we are still able to hide it successfully**, even though higher delays also require exposing a larger degree of ILP to be hidden, which is the reason why on average the relative speedup is sometimes not as high with longer vector sizes. It is also interesting to note that **some workloads benefit more from our optimizations at the 32k size than from a much larger and more expensive CAPE131k** (area about two full CPU cores as opposed to one). For example in the case of `lreg`. On the other hand, this is not the case for `kmeans`, as the particular bottleneck for that one was its low data reuse, so being able to store a larger working set is more helpful for this workload than our optimizations.

The ideal applications that leverage everything PUMICE has to offer should possess a healthy mix of compute and memory operations and be able to overlap scalar and vector operations. Applications that follow this pattern include, for example, `matmul` and `lreg`, reaching between 1.6x and 1.8x performance, respectively, for our 32k configurations. In the latter, about 1.6x of that 1.8x speedup can be attributed to the overlap of memory and computation alone.

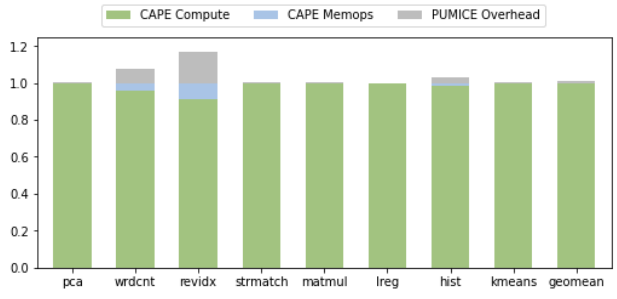
Overall, although the performance benefit of hiding the aforementioned delays is highly dependent on how long the vector instructions make use of the AP Array, **an integrated design results in speedups across the board with no performance decreases**. No matter what the workload is, PUMICE will provide increased performance with no additional optimizations needed on the software side. Furthermore, by hiding and lessening the impact of the command distribution delay, we have significantly reduced the trade-offs of using larger vector sizes.

3) Energy and Area Overhead: Using the per-microinstruction energy figures in the original CAPE paper [4], we inspect the traces of CAPE and PUMICE to determine how much energy is required to execute each application. In Figure 6 we observe that there is actually a minimal decrease (less than 0.5%) when comparing CAPE to PUMICE. This is primarily due to **savings in leakage power**, as PUMICE is able to finish running the workloads in a significantly shorter amount of time.

Although the energy overhead for memory operations in PUMICE is roughly double than that of CAPE due to the buffering required for overlap, the additional energy cost is only 2% on average. Depending on the application, it can range from minor savings (less than 0.1%) in `strmatch` to to an increase of 14.8% in `revidx` and `wrdcnt` as shown in Figure 6b. However, memory operations remain a small subset of all the AP instructions, so this overhead is modest in most cases.



(a) Normalized Energy Usage



(b) Energy Overhead Breakdown for PUMICE

Fig. 6: The energy consumption of CAPE and PUMICE with a vector length of 32k.

PUMICE involves a **modest area overhead**, as most of the proposed changes involve adding control circuitry and a small amount of extra buffering. The Load-Store Buffer requires the capacity to store an entire vector, and a conservative area estimation for this is 1/32 the total area of the AP, which stores 32 vector registers. Therefore, the total overhead of our proposed changes should be well below 5% over the CAPE design.

VI. Conclusion

Previous work had already shown the effectiveness of PUM-based very-long-vector architectures in exploiting data-level parallelism, but much performance remained untapped by regarding them as accelerators instead of functional units. While tight integration is not always the answer, our results confirm that such an approach in this context increases performance significantly, as it enables exploitation of instruction-level parallelism as well, by decoupling memory and computation and allowing for a smooth back-to-back execution of operations which maximizes utilization. Particularly, for the applications studied, our tightly-integrated approach results in speedups of up to 2.2x over the original accelerator design (1.4x on average), transparently to the programmer. Performance has improved across the board, even for workloads that were previously thought to be unfavorable for the vector and AP paradigms.

Acknowledgments

The authors would like to thank Helena Caminal for her advice, support, and assistance in reproducing the results from previous work, Kailin Yang for his early discussion and help in reproducing the results from previous works, and Michael Woodson for his technical support.

References

[1] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das. Compute Caches. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 481–492, February 2017. ISSN: 2378-203X.

[2] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator, aug 2011.

[3] Helena Caminal, Yannis Chronis, Tianshu Wu, Jignesh M Patel, and José F Martínez. Accelerating Database Analytic Query WorkloadsUsing an Associative Processor. *New York*, page 15, 2022.

[4] Helena Caminal, Kailin Yang, Srivatsa Srinivasa, Akshay Krishna Ramanathan, Khalid Al-Hawaj, Tianshu Wu, Vijaykrishnan Narayanan, Christopher Batten, and José F. Martínez. CAPE: A Content-Addressable Processing Engine. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 557–569, February 2021. ISSN: 2378-203X.

[5] R. Espasa and M. Valero. Decoupled vector architectures. In *Proceedings. Second International Symposium on High-Performance Computer Architecture*, pages 281–290, San Jose, CA, USA, 1996. IEEE Comput. Soc. Press.

[6] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. Duality cache for data parallel acceleration. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 397–410, New York, NY, USA, June 2019. Association for Computing Machinery.

[7] Shuangchen Li, Dimin Niu, Krishna T Malladi, Bob Brennan, and Hongzhong Zheng. DRISA : A DRAM-based Reconfigurable In-Situ Accelerator. *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 288–301, 2017. Publisher: ACM ISBN: 978-1-4503-4952-9.

[8] Francisca Quintana, Jesus Corbal, Roger Espasa, and Mateo Valero. Adding a vector unit to a superscalar processor. In *Proceedings of the 13th international conference on Supercomputing*, ICS '99, pages 1–10, New York, NY, USA, May 1999. Association for Computing Machinery.

[9] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, Scottsdale, AZ, USA, 2007. IEEE.

[10] Vivek Seshadri, Todd C. Mowry, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A. Kozuch, Onur Mutlu, and Phillip B. Gibbons. Ambit: An In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-50 '17*, pages 273–287, New York, New York, USA, 2017. ACM Press.

[11] Milan Stanic, Oscar Palomar, Timothy Hayes, Ivan Ratkovic, Adrian Cristal, Osman Unsal, and Mateo Valero. An Integrated Vector-Scalar Design on an In-Order ARM Core. *ACM Transactions on Architecture and Code Optimization*, 14(2):17:1–17:26, May 2017.

[12] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. The RISC-V Instruction Set Manual. I:1356, 2011. ISBN: 9781467303422.

[13] Yue Zha and Jing Li. Hyper-Ap: Enhancing Associative Processing through A Full-Stack Optimization. *Proceedings - International Symposium on Computer Architecture*, 2020-May:846–859, 2020. ISBN: 9781728146614.