Accelerating Database Analytic Query Workloads Using an Associative Processor

Helena Caminal*
hc922@cornell.edu
Cornell University
Ithaca, NY, USA

Yannis Chronis*
chronis@cs.wisc.edu
University of Wisconsin-Madison
Madison, WI, USA

Tianshu Wu
tw387@cornell.edu
Cornell University
Ithaca, NY, USA

Jignesh M. Patel
jignesh@cs.wisc.edu
University of Wisconsin-Madison
Madison, WI, USA

José F. Martínez
martinez@cornell.edu
Cornell University
Ithaca, NY, USA

ABSTRACT

Database analytic query workloads are heavy consumers of datacenter cycles, and there is constant demand to improve their performance. Associative processors (AP) have re-emerged as an attractive architecture that offers very large data-level parallelism that can be used to implement a wide range of general-purpose operations. Associative processing is based primarily on efficient search and bulk update operations. Analytic query workloads benefit from data parallel execution and often feature both search and bulk update operations. In this paper, we investigate how amenable APs are to improving the performance of analytic query workloads. For this study, we use the recently proposed Content-Addressable Processing Engine (CAPE) framework. CAPE is an AP core that is highly programmable via the RISC-V ISA with standard vector extensions. By mapping key database operators to CAPE and introducing AP-aware changes to the query optimizer, we show that CAPE is a good match for database analytic workloads. We also propose a set of database-aware microarchitectural changes to CAPE to further improve performance. Overall, CAPE achieves a 10.8× speedup on average (up to 61.1×) on the SSB benchmark (a suite of 13 queries) compared to an iso-area aggressive out-of-order processor with AVX-512 SIMD support.

CCS CONCEPTS

• **Computer systems organization → Single instruction, multiple data**; • **Information systems → Query operators**; Query optimization.

KEYWORDS

Associative Processors, Databases, Analytic Workloads, Analytics, Acceleration, Codesign, Vector Architectures

*Yannis Chronis and Helena Caminal contributed equally to this research.

ISCA '22, June 18–22, 2022, New York, NY, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-8610-4/22/06…$15.00
https://doi.org/10.1145/3470496.3527435

ACM Reference Format:
Helena Caminal, Yannis Chronis, Tianshu Wu, Jignesh M. Patel, and José F. Martínez. 2022. Accelerating Database Analytic Query Workloads Using an Associative Processor. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3470496.3527435

1 INTRODUCTION

Data analysis is a critical tool that drives business decision-making. The value of data-driven decisions is undeniable, and as a result, both the volume of data and the associated computing needs are continuously increasing [7]. The architecture community has proposed a wide variety of solutions with the potential to bring large performance benefits to database platforms [8–13, 31, 32, 39, 41, 43, 47, 53, 56, 60, 64, 65]. By and large, custom designs trade generality for performance.

Associative Processors (AP), first introduced in the 1970s [22], have recently re-emerged as an attractive concept for general purpose applications that exhibit data-level parallelism [15, 46, 66, 68]. In addition to their very long vector-style compute capabilities, APs provide efficient multi-row *search* and *update* operations. As it turns out, database systems that run analytic workloads not only structure operands in vector-like columns but also contain searches and updates at the core of their operators. Thus, we hypothesize that APs may have the potential to accelerate analytic query workloads.

In this paper, we investigate how amenable APs are to delivering high performance on analytic query workloads. We build a research prototype database system called Castle that executes end-to-end analytic query workloads on the recently proposed Content-Addressable Processing Engine (CAPE) [15]. CAPE is an associative processor core that is highly programmable via the RISC-V ISA with standard vector extensions. By updating database operator algorithms and query optimization strategies, we show that CAPE is a strong match for database analytic workloads. We also propose database-aware microarchitectural changes to CAPE to further improve the performance of CAPE for our target workloads.

Overall, our contributions are:

(1) An in-depth analysis and evaluation of the fit between APs and analytic query workloads. To the best of our knowledge, this is the first paper to explore this synergy for end-to-end database analytic query workloads.

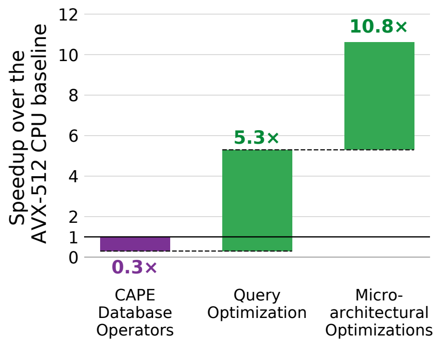


Figure 1: Waterfall chart of the speedup geomean of CAPE compared to an AVX-512 baseline for the SSB benchmark queries.

(2) An efficient mapping of database operators using the standard RISC-V vector abstraction and query optimization strategies for CAPE.
(3) A set of database-aware microarchitectural changes for CAPE to further improve performance on analytic query workloads while maintaining its programmability and performance as a general-purpose core.

To evaluate the performance of database analytic queries on CAPE, we compare to an area-equivalent aggressive out-of-order processor with AVX-512 SIMD extensions using a widely used end-to-end benchmark suite. Our results highlight the importance of a codesign approach to effectively improve performance. Figure 1 is a waterfall chart of the speedup geometric mean achieved by Castle on SSB (a benchmark suite of 13 queries), as obtained in our evaluations (Sections 4 and 6). This type of chart visualizes the contribution of different components to a total. Interestingly, we observe a slowdown of 0.3× by simply using CAPE-adapted database operators (Section 3). However, by adding CAPE-specific query optimization strategies (Section 3.4), the speedup improves to 5.3× over the baseline. Additional database-aware microarchitectural changes to CAPE (Section 5) further increase the overall speedup to 10.8×.

2 BACKGROUND

In this section, we introduce the three main components of the Castle framework: associative processors, the Content-Addressable Processing Engine (CAPE), and relational databases.

2.1 Associative Processors

An associative processor (AP) [22] 1) stores data in vector form, 2) can compare a key against all vector elements in parallel (*search*), and 3) can update all matching elements in bulk with a new value (*update*). These two primitive operations, search and update, are supported via associative memories and can each be applied simultaneously to an entire vector of elements. The two primitives are typically arranged in search-update pairs, which are sometimes bit-serial (e.g., for arithmetic operations) and element-parallel—i.e., a search-update pair operates on the same bit of all the elements of a vector (or several vectors). The sequence of search-update pairs that operates sequentially on all the bits of each vector value constitutes an *associative algorithm*.

We describe how associative algorithms implement higher-level operations and how APs execute them by using vector increment as an example (Figure 2). A vector increment increases all vector elements in value by one. To perform the operation, first, 1 is added to the least significant bit of all vector elements, and any carry is remembered. Then, for each element, the corresponding carry is added to the next bit, and so forth. Of course, an associative processor cannot "add" bits per se. Instead, it implements bitwise addition through a sequence of search-update pairs that follow the truth tables for a half adder (Figure 2 left), one bit position at a time. For each bit position i. At a high level, the logic is as follows: 1) Search vector elements for which the i^{th} bit is 0 and the running carry for that element (an extra bit of storage) is 1, then update the i^{th} bit of matching elements to 1 and their running carry to 0. 2) Search vector elements whose i^{th} bit is 1 and the running carry for that element is also 1, then update the i^{th} bit of matching elements to 0 and the running carry to 1.

Note that, in this example, we do not perform search-update pairs for the two cases where the output is the same as the input—namely, neither the element's bit nor the running carry flip as a result of applying the half adder truth table (crossed-out entries in the truth tables of Figure 2). Also, some additional support beyond

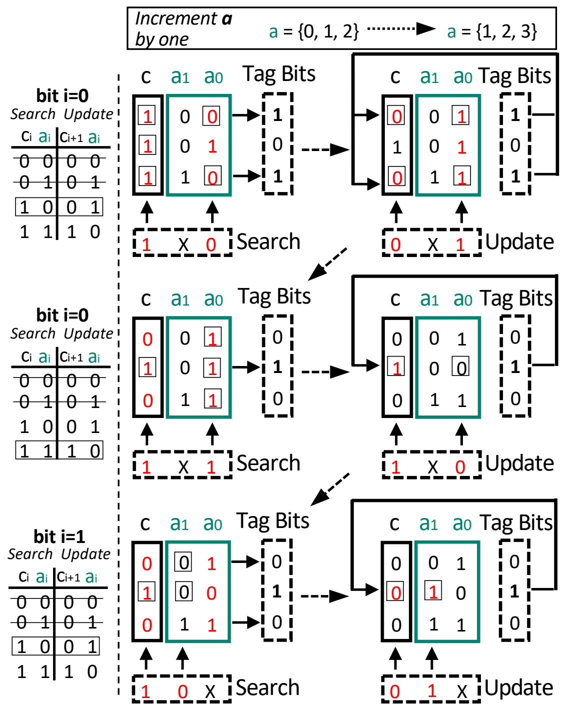


Figure 2: Associative increment instruction as a sequence of search-update operations to a vector of three two-bit elements (left to right and top to bottom). Carry bit column c is initialized to 1, as seen in [15]. The operation is done once all carry bits are zero (bottom-right).

search/update is needed: 1) We need two bits of additional storage per vector element: One bit serves as the running carry (initialized to 1 at the beginning of the instruction with a single bulk-update), and another bit "tags" matching elements (*tag bits*) in each of the searches. 2) We also need to *mask out* the bits not participating in the search or update (indicated by "X" in Figure 2) in order to constrain searches and updates to the i^{th} bit of each element. 3) The sequence of operations that implement the increment instruction needs to be "stored" somewhere (e.g., the micro-memory of a sequencer).

As stated before, most arithmetic operations on APs operate bit-serially; in fact, they often require multiple search and update operations per bit. As a result, even a relatively simple increment instruction on a 32-bit value requires over 100 such operations. The key to AP performance is that operations are carried out *simultaneously on an extremely large number of vector elements, typically tens to hundreds of thousands.*

2.2 A Content-Addressable Processing Engine

A *Content-Addressable Processing Engine (CAPE)* [15] is a general-purpose AP core that adopts a contemporary ISA abstraction, namely RISC-V with standard vector extensions, and can be readily integrated into a tiled architecture. CAPE has been shown to accelerate a broad range of data-parallel workloads [15].

CAPE's architecture comprises four main blocks (Figure 3). The Control Processor (CP) is a tiny in-order core that runs RISC-V code with standard vector extensions [62]. It processes scalar instructions locally and offloads vector instructions to the Compute-Storage Block (CSB), which acts as a coprocessor and is CAPE's associative engine. The CSB is content-addressable memory that is organized into large vectors (order of 10^4 32-bit vector elements). The maximum degree of data parallelism for each associative operation is the length of these vectors (also denoted as `MAXVL` in this paper). A vector instruction commits in the CP only after it completes in the CSB. In the shadow of an outstanding vector instruction, subsequent scalar logic/arithmetic ALU instructions may issue and execute (if not data-dependent with the vector instruction) but not commit. The increment example described in Section 2.1 maps to standard RISC-V vector instructions.

Load/store vector instructions en route to the CSB go through the Vector Memory Unit (VMU). Other vector instructions go through

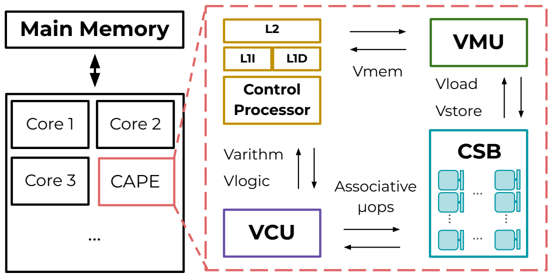


Figure 3: CAPE functional blocks (not drawn to scale): Control Processor, Compute-Storage Block (CSB), Vector Memory Unit (VMU), and Vector Control Unit (VCU). CAPE can be used as a plug-and-play core in a multicore CPU.

	Instructions	Steps (n bits)	Mode
Arithmetic	vv add	$8n + 2$	Bit-serial
	vv subtraction	$8n + 2$	
	vv multiplication	$4n^2 + 4n$	
	vv reduction sum	$\sim n$	
Logic	vv logical and	3	Bit-parallel
	vv logical or	3	
	vv logical xor	4	
Comparison	vs equality (search)	$n + 1$	Bit-serial
	vv equality	$n + 4$	
	vv inequality	$3n + 6$	

Table 1: An illustrative subset of associative operations, their cost model for n bits, and their compute mode. The"vv" and "vs" refer to vector-vector and vector-scalar, respectively, to denote the type of their input operands. The result is always a vector operand.

the Vector Control Unit (VCU), which generates microcode sequences to drive the CSB. VMU and VCU generate/transfer control/-data signals to the CSB, respectively. The RISC-V vector register names in each instruction are used to index the appropriate vector operands in the CSB.

The CSB is composed of tens of thousands of associative subarrays, which can perform massively parallel operations (shown as blue boxes in Figure 3). Each subarray is made up of 6T bitcells that can readily support the four microoperations used in CAPE's computational model: single-element reads and writes, as well as highly-efficient multi-element (vector) searches and updates. CAPE's associative arrays are built out of SRAM technology. A CAPE with a 4 MB CSB is comparable in area to a conventional out-of-order core [15].

As described in [15], the CSB is organized in *chains* of 32×32 (rows×columns) memory subarrays. Each of these subarrays i propagates the tag bits to their next $i + 1$ neighbor, hence, forming a chain. The CSB comprises thousands of chains that operate in lockstep, executing the same command independently (some chains may be disabled if the program requires less parallelism than the one supported by the hardware).

Associative algorithms were originally proposed as bit-serial operations [22], and CAPE performs arithmetic operations in this way. CAPE's array geometry and bitslicing of data values allow *logical* associative algorithms (e.g., vector XOR) to be performed in a bit-parallel fashion [15] (Table 1).

2.3 Relational Database Management Systems

Relational Database Management Systems (RDBMS) are software systems that store, manage and query collections of data organized as relations (also called tables). A defining characteristic of databases is the declarative interface. Queries submitted to the system are expressed in a declarative language (e.g., SQL). As a result, a user does not have to be aware of the database internals to use it; a declarative query defines the result of a computation but does not describe the computation. Further, the declarative interface allows the database system to organize data, build indices (auxiliary data structures that speed up data retrieval), and choose the best algorithm for each operation without the user's involvement. A critical component that enables a database system to be performant is the query optimizer. The query optimizer produces an execution plan for each query by considering data properties and internal system information that are not exposed to the user.

The workflow for using a database includes loading collections of data organized as relations, submitting queries, and collecting results also organized as relations. Relational databases include many components; a parser that transforms queries into trees of operators; a query optimizer that transforms the parser output into an execution plan; an execution engine that executes the optimized execution plan using a library of operators; a storage engine that stores the relations and manages secondary structures (e.g., indices).
Analytic Query Processing— is a class of workloads that analyzes large volumes of data and produces summary statistics (also known as Online Analytical Processing or OLAP). Analytic queries are long-running and apply complex analyses to the data. There are three core operators used in analytic queries: selection, join, and aggregation. Selections retrieve the qualifying rows of a relation that match a set of predicates. Joins combine two relations based on the values in a set of columns. Aggregations group rows of a relation based on the value of a set of columns and calculate a summary statistic for each group (e.g., SUM, MAX, STDDEV).

3 CASTLE: A DATABASE SYSTEM FOR ASSOCIATIVE PROCESSING

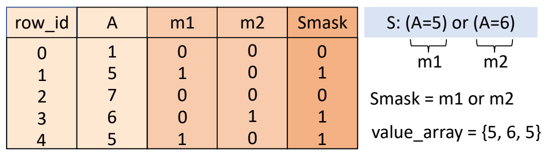
In this section, we present Castle, a research prototype database system built for a CAPE core. Castle follows a columnar design, which is the preferred option for analytic workloads [58]. Vectorization and CAPE naturally align with columnar processing: data is stored in vectors and operations produce masks (associative search) or consume them (bulk update). In this section, we describe the implementation of Castle's database operators for CAPE. Then, we adapt query optimization for multi-join queries. The optimized query plans for CAPE differ from those produced by traditional optimizers (see Section 3.4 for details), which points to the importance of changing conventional database optimization strategies for CAPE.

The algorithms presented in this section are vector-length agnostic. CAPE exposes its parallelism via RISC-V's variable vector length mechanism. Data analytic workloads process datasets typically much larger than that, which requires data movement between a CAPE (or conventional) core and main memory. Castle used the variable vector length mechanism to load partitions of the input columns to CAPE's CSB that are automatically sized to the largest between the hardware vector length (e.g., `MAXVL`) and the remaining input, until the entire input has been processed.[1]. Often, Castle retains intermediate results in-situ for subsequent operators and reduces data movement to/from main memory to further improve performance.

3.1 Indexing and Selection

Indexing: A database index on a relation maps a value to a relation row (specifically a row id). Indices are used to speed up query processing by retrieving the rows that match a value without the need

[1]Similar to other RISC-V vectorized binaries, Castle binaries can be run on CAPE cores with different MAXVL without the need to modify or recompile them.



row_id	A	m1	m2	Smask
0	1	0	0	0
1	5	1	0	1
2	7	0	0	0
3	6	0	1	1
4	5	1	0	1

Figure 4: Example of selection and mask compaction.

to scan the entire relation. In CAPE, searches are single-cycle primitives. Data loaded in CAPE is effectively indexed by virtue of being accessible as a RISC-V vector operand, without requiring building or maintaining an additional data structure. Drawing a parallel to existing indexing structures, a vector in Castle is a hashmap with no collisions that uses the identity function as a hash function.
Selection: A Castle selection operator applies predicates to one or more column(s) loaded in vectors in parallel, and produces a mask to identify the matching rows. The output mask is stored in a vector register. Masks can be logically combined as defined by a predicate.
Compaction after selection: Figure 4 presents an example of a selection operation that filters column A. The selection predicate is ($A = 5$ *or* $A = 6$). A search is performed for each equality predicate and outputs a mask ($m1$ and $m2$). The two masks are combined into the final selection result mask *Smask*. Depending on the upstream usage, we can opt to *compact* the output to a (software data structure) *values array* that can be accessed by subsequent operators. A values array contains the values whose mask bit is set. The figure presents the output of the selection in the two data formats (mask and a value array).

3.2 Join

A join (denoted by ⋈) finds matching rows from two relations according to a predicate. Traditional join algorithms (i.e., nested-loops join, hash join, lookup join [27]) iterate over the first relation and probe the second relation to find matching rows (probe phase). Some algorithms (i.e., hash join) accelerate probes with indices, which are built as part of the join operation (build phase). Traditionally, the index is built on the smaller relation, and the larger relation is used for probing (probe side) since creating an index is more expensive than probing it.

CAPE has dedicated support to search all elements of a vector for a matching key simultaneously and, therefore, can skip the index build phase. By eliminating the index build cost, Castle can choose the smaller relation as the probe side and reduce the number of probes. In the common case for analytic workloads, there is a size imbalance between the relations to be joined [14], which favors the Castle join algorithm.

Algorithm 1 Castle Join

```
Rel1 ⋈ (=join) Rel2 on Rel1.ColA=Rel2.ColB
1: while ColA is not consumed do
2:     v0 = vload_vector(next ColA partition)
3:     for i=0; i < |ColB|; ++i do
4:         v1 = search(v0, ColB[i])
5:         construct the join result in the desired format
```

Algorithm 1 presents the pseudocode for the Castle join operator that joins Rel1 and Rel2. The two relations are joined on ColA from Rel1 and ColB from Rel2. We iteratively load all partitions of ColA in vector register v0 (line 2). For each tuple of the ColB relation we search v0, and get back a mask of matches in v1 (lines 3-4). Depending on how the result of a join is going to be used, we have two options: a) keep a mask of the joined tuples that corresponds to the relation loaded in CAPE (used for semi-joins as their result only includes data from one of the joined relations) b) materialize the full result (or the part of the result that is needed by upstream operators) in the CSB or main memory. Bulk updates are used to materialize the join result in the CSB.

3.3 Aggregation

An aggregation operation is used to produce summaries for groups of rows. Modern data processing engines use sort-based or hash-based approaches [27] that rearrange rows into their respective groups to evaluate an aggregation.

Castle's aggregation operator uses an associative search instruction to discover the rows that belong to the same group in a data-parallel way and can calculate group-wide statistics via vector reductions using predication.[2] Predication allows selecting a subset of a vector to be processed, in this case the rows that belong to the same group can be reduced without physically rearranging the data. This process is repeated for each group present in the data. We illustrate Castle's aggregate operator implementation using the following example query:

```
SELECT GCol, SUM(SCol)  FROM R  GROUP BY GCol;
```

Algorithm 2 shows the pseudocode for the example aggregate query that sums the values of column *SCol* for each unique value of the column *GCol* value and returns the result to main memory. Algorithm 2 allocates two parallel arrays in main memory to store the final result: one for the group keys (*mm_arr_g*) and one for the group aggregate values (*mm_arr_sum*). The result could alternatively be stored in associative vectors depending on the downstream usage. The two relevant columns *GCol*, *SCol* are loaded in vectors (lines 3-4). An input mask with the same length as the columns is initialized to all 1s (line 5). The input mask keeps track of all the rows that we have not been processed so far. If the aggregation is performed on the output of another operator (e.g., a selection), then the input mask is the previous operator's output mask. Iteratively, we discover each group and calculate its sum aggregate value (lines 7-11). In each iteration, we find the index of the first set bit in the input mask ($v2$). Retrieving the first set bit of a mask in Castle is performed using CAPE's priority encoder tree (using the `vmfirst` instruction, lines 6, 14) [15]. Using the row index (*idx*), we retrieve the corresponding grouping column value (line 8). The next step is to find all other rows that belong to the same group by performing a search on the *GCol* vector for the value of the current group key (line 9). The mask produced by the search is used to calculate the sum per group and unset the processed rows from the input mask ($v2$). To sum up the values of *SCol* that belong to the current group, we use the predicated *vsum* operation, which sums all values of a vector indicated by a mask (line 10) using a hardware reduction tree [15]. To unselect the elements of the current group that have already been processed, a vector *xor* operation is used (line 11). When this index (*idx*) becomes -1, then the entire input is processed and we exit the loop and the aggregation is completed. The cost of Castle's aggregation correlates with the number of unique groups found.

[2]Predication in vector architectures refers to the ISA feature that allows an instruction to disable certain vector elements.

Algorithm 2 Castle Aggregation

```
SELECT sum(SCol), GCol ... GROUP BY GCol
1:  mm_arr_g = new mm_array()          ▷ array in main memory
2:  mm_arr_sum = new mm_array()        ▷ array in main memory
3:  v0 = vload_vector(GCol)            ▷ loads a column into a vector
4:  v1 = vload_vector(SCol)
5:  v2 = mask_init(1)                  ▷ replicates a constant in a vector
6:  idx = vmfirst(v2)                  ▷ get the index of the the first set bit
7:  while idx != -1 do
8:      groupkey = GCol[idx]
9:      v3 = vsearch(v0, groupkey)     ▷ vector search
10:     s = vsum(v3, v1)               ▷ predicated (by mask v3) sum v1
11:     v2 = vxor(v2, v3)              ▷ unselect elements of current group
12:     mm_array_g .append(groupkey)
13:     mm_array_sum.append(s)
14:     idx = vmfirst(v2)
15: Return mm_array_g, mm_array_sum
```

3.4 Query Optimization – Join Ordering

Database systems employ a module called query optimizer, and a key goal of an optimizer is to explore plans with different join orders. An optimized plan can be far more efficient than a naive plan [40]. In addition to the order of execution of joins, query optimizers choose the shape of the query plan, which dictates the probe side of joins (Section 3.2). Databases designed for analytic workloads organize data in a star (or a snowflake) schema, where one relation (*fact*) is much larger than the rest (*dimensions*) [48]. The information contained in the *fact* relation connects (joins) the smaller relations. Therefore, columns from *fact* relation are involved in nearly every join.

Traditional database systems favor plans with a left-deep shape. In a left-deep plan, indices are built on the *dimension* relations and the *fact* relation is the probe side. The alternative plan shapes are right-deep and zig-zag. Right-deep plans use the *dimension* relations as the probe sides and build indices on the *fact* relation and the intermediate result of each join. Zig-zag plans change the direction of the data flow through the sequence of join operators, and are essentially a combination of right-deep and left-deep subplans.

Left-deep plans allow for deeper *pipelining* of joins in a traditional database system. Pipelined join execution means that a tuple from the *fact* relation is used to probe all the *dimension* relations before the following fact tuple is joined. Right-deep plans are generally not preferred in a traditional system as intermediate results have to be materialized (to build an index on them) [40] before the next join, making them more expensive compared to other plans.

CAPE implicitly indexes data loaded into a vector register, so the Castle join operator changes the relation of the join that is traditionally the probe side to increase performance by reducing the number of probes (Section 3.2). As a result, right-deep plans that use the *dimension* relations as the probe sides are more efficient in Castle compared to left-deep plans.

As joins are performed in a multi-join query the intermediate join result may become smaller than the yet unjoined dimension relations. At that point, it is more cost-effective to switch the join direction. Castle also has the ability to evaluate zig-zag join plans, which allow for a change in the probe side mid-plan. This is a surprising result, the data flow direction change mid-plan is expensive in traditional systems, as it breaks the join pipeline. In CAPE, there is no need for an index-building step between joins, and thus the switch can be made without breaking the existing join pipeline.

Let us use an example to illustrate the implications of the three plan shapes in Castle, namely left-deep, right-deep, and zig-zag, and present the cost estimation formula used by Castle's optimizer. Further, we will use this example to illustrate an interesting side-effect of right-deep plans when executed on CAPE. The example query joins two *dimension* relations *d1*, *d2* with the *fact* relation *f*:

```
SELECT *
FROM fact as f, dimension1 as d1, dimemsion2 as d2
WHERE fact.c1 = d1.c1 AND d2.c2 = dim2.c2 AND ...
```

To join two relations in Castle, we probe one relation (*relation1*) for every tuple in the second relation (*relation2*), which requires $|relation2| * |Part(relation1)|$[2] searches, where *Part(X)* splits relation *X* in MAXVL-sized partitions.[3] The probing for all tuples of *relation2* is repeated for every partition of *relation1* loaded to CAPE.

Figure 5 visualizes the three plans using our example query. Using sample query statistics lifted from the SSB benchmark, we calculate the cost for each plan. Since the *dimension* relations are smaller than the *fact* relation, a right-deep plan decreases the execution cost in Castle. The right-deep plan for our example query is cheaper than the left-deep plan, as *d1* is smaller than the *f* relation. As an added benefit for right-deep plans, the execution cost is independent of the join order. The execution cost only depends on the sizes of the *dimension* relations because we load the whole *fact* relation (as shown by the cost formula). Thus, a right-deep plan in Castle generally safeguards us from executing a plan with a bad join order [40] due to inaccurate intermediate result size estimations [33]. This behavior is not the case in a traditional database system since in the case of a hash join, hash maps would have to be built on the intermediate join results (the size of which could be underestimated [40] during query optimization where estimates are used to decide the join order).

Although right-deep plans in Castle are generally more performant compared to left-deep plans, they are not necessarily optimal. As the joins are executed, if the probe side becomes smaller than the yet unjoined *dimension* relations, then a zig-zag plan is more efficient. In our example, the result of *f* joined with *d1* ($f \bowtie d1$) is smaller than *d2*; thus, using a zig-zag shape, which makes $f \bowtie d1$ the probe size, yields a significant cost improvement.

Building on the observations described above, Castle employs an optimization rule to decide the join order together with the plan shape so that the number of searches is minimized. The rule

[2]The vertical lines denote the relation size measured in number of rows.

[3]Here, MAXVL is used to denote the degree of parallelism offered by the CSB exposed as the maximum length of vector registers.

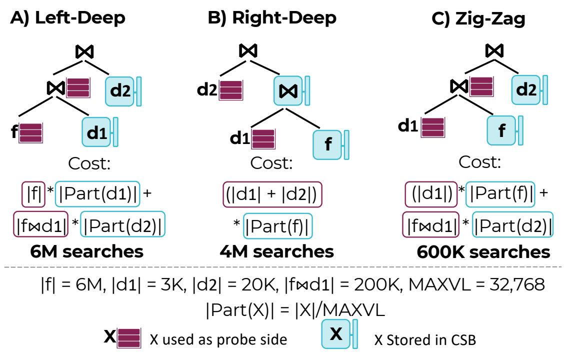


Figure 5: Query plans with different plan shapes for a query that joins (⋈) a fact relation (f) with two dimension relations (d1 and d2). Under each plan, we calculate the execution cost in terms of searches. Operators are evaluated bottom-up as shown in the query plans and the leaf nodes constitute the input relations to the query. Next to each node, we indicate if they are used as probe side (green symbol) or stored (in MAXVL-sized partitions) in the CSB (blue box).

considers the vector length of CAPE, as well the estimated query statistics (sizes of intermediate results).

In our evaluation (Section 4.2), we found that the best execution plans for Castle are usually zig-zag shaped and, less often, right-deep shaped. This is a surprising result and emphasizes the need for a hardware-aware strategy to design database systems.

4 EVALUATION OF DATABASE ANALYTICS ON CAPE

In this section, we show a first end-to-end performance evaluation of a database analytic workload executed by Castle running on unmodified CAPE. We evaluate the CAPE optimized database operators and query optimization strategies presented in Section 3, and analyze the usage of vector instructions.

4.1 Experimental Methodology and Setup

Castle System Configuration— We run experiments for a CAPE core with a CSB that supports a maximum vector length (MAXVL) of 32,768 vector elements[4] and has an effective capacity of 4 MB (32 32-bit vectors of 32,768 elements). The control processor (CP) is modeled using the RISC-V RV64G MinorCPU gem5 model, running at 2.7 GHz, and it is configured as a dual-issue, in-order, five-stage pipeline. The simulated system uses a DDR4 main memory with 64 GB of capacity, eight channels, and a maximum bandwidth of 153.6 GB/s. CAPE does not have an L3 cache, although past work has suggested that the CSB could be configured as one [15]. Table 2 summarizes the architectural configuration. To compile the vectorized Castle code (written in C) we use the RISC-V toolchain [5].

Baseline System Configuration— We choose an out-of-order, superscalar core based on a high-end Intel Skylake processor modeled with the DerivO3CPU gem5 model, which is similar in area

[4]For reference, an Intel AVX512 [4] vector coprocessor can support a vector length of 16 (32-bit elements).

	Baseline Core	**CAPE's Ctrl Processor**
System configuration	out-of-order core, 2.7GHz	in-order core, 2.7GHz
	32 kB/32kB/1MB L1D/L1I/L2	32 kB/32kB/1MB L1D/L1I/L2
	5.5MB L3 (shared), 512B cache line	512B cache line
Core configuration	8-issue, 224 ROB, 72 LQ, 56 SQ	2-issue in-order, 5 LSQ
	4/4/4/3/1 IntAdd/IntMul/FP/Mem/Br units	4/1/1/1 Int/FP/Mem/Br units
	TournamentBP, 4,096 BTB, 16 RAS	TournamentBP, 4,096 BTB, 16 RAS
L1 D/I cache	8-way, LRU,MESI, 2 tag/data latency	8-way, LRU, 2 tag/data latency
L2 cache	16-way,LRU,MESI,14 tag/data latency	16-way,LRU,14 tag/data latency
L3 cache	11-way, LRU, 50 tag/data latency, shared	N./A.
Main memory	64GB DDR4, 8 channels, 19GBps per channel	

Table 2: Experimental setup

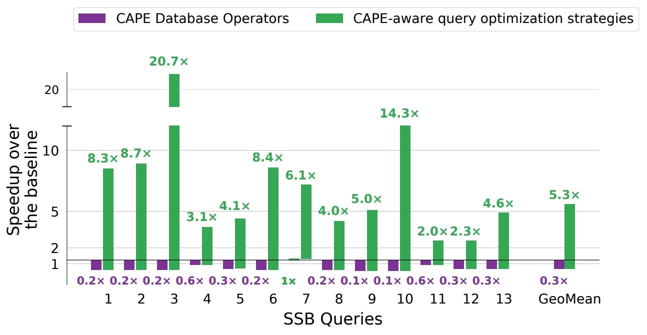


Figure 6: Speedup of Castle for SSB queries SF=1, visualized with a waterfall chart. For each query the left bar shows the speedup over our baseline using only the CAPE database operators, while the right bar is the speedup when Castle also uses the CAPE-aware query optimization strategies.

to one CAPE core when scaled to the same technology node and frequency [15]. To compare Castle against an AVX-512 equipped core, we first measure performance on a real state-of-the-art Intel Xeon CPU using code with and without AVX-512 instructions[5] and calculate the performance ratio. We then use that ratio to scale up the performance of our gem5 CPU baseline, and that becomes the AVX-512 baseline system we compare against.

SSB Benchmark— To evaluate the end-to-end performance of Castle, we use the Star Schema Benchmark (SSB) [48]. We have modified the benchmark in two ways: (1) we compress string columns that are used in selection and join predicates using standard encoding techniques to 32-bit values, which is CAPE's default data size (compression is a commonly used technique in data analytic workloads [28, 59, 63]) 2) For simplicity, we omit the sorting of the final result in all configurations, which amounts to a negligible processing time in the SSB queries. We present results for SSB scale factor (SF) 1 (~600MB of raw data). We have also used the simulation framework to run experiments for scale factors from 0.5 up to 10 and the results are similar. Note that, at SF1 the capacity of CAPE's CSB is 150× smaller than the input relations of the benchmark. Our simulation infrastructure models the data movement to and from main memory faithfully.

Reference Codebases— To evaluate Castle, we use two highly optimized reference codebases that implement the SSB benchmark. First, a scalar codebase that targets a traditional core. Second, a codebase vectorized specifically for AVX-512 [4].

To confirm that the reference codebases are competitive against state-of-the-art production databases, we compared them against MonetDB v11.41.11 (Jul2021-SP1 https://www.monetdb.org/) [23, 61]. For MonetDB, we set the number of worker threads to 1 to compare against our single threaded reference codebases, as the focus of this paper is on a single-core scale. Running natively on a Xeon CPU, the scalar reference codebase is on average 2.1× faster than MonetDB and the AVX-512 reference codebase is 3.8× faster than MonetDB. For the rest of the paper, we will use the AVX-512 vectorized reference codebase running on a AVX-512 equipped core as our baseline.

4.2 Results of the SSB Queries

Figure 6 shows the performance of Castle running on a CAPE core compared to our baseline for all 13 queries in the SSB benchmark at scale factor 1 (Section 4.1). We observe speedups between 1.4× and 20.7×, with a geometric mean of 5.3×.

[5] We disabled the automatic compiler vectorizer for the scalar binary.

CAPE Database Operators— As shown in Figure 6, simply vectorizing database operators for CAPE does not yield a performance improvement when executing most end-to-end queries. In fact, there is a slowdown of 70% on average. In this setting, the query optimizer uses the same optimization strategies as traditional database systems, which produce left-deep plans for all SSB queries (Section 3.4).

Query Optimization for Joins— The unique performance characteristics of associative processors drastically change the traditional cost behavior of database operators (Section 3.2). Analytic queries usually execute multiple joins (SSB queries 4-13 execute two to four joins). In Section 3.4, we proposed modifying the join optimization strategy of query optimization to consider additional plan shapes. Using the CAPE-aware query optimizer, Castle optimizes query plans specifically for associative processing and achieves a 5.3× speedup over our baseline.

For SSB, right-deep plans on CAPE are almost always more performant than left-deep plans. Out of the 13 best performing plans (one for each query), 8 are zig-zag, and the remaining five are right-deep. Note that queries 1-3 have only one join, and for queries 4 and 7, a right-deep plan is the most performant option.

As discussed in Section 3.4, left-deep plans are favored by conventional state-of-the-art database systems due to their superior performance, but in Castle, they are the worst-performing option in the 13 SSB queries. All plan shapes access the same amount of data (the relations to be joined). However, depending on the plan shape, a different number of searches is performed, which translates to the processing effort required. Figure 5 shows an example of a query where right-deep plans and zig-zag plans can significantly reduce the number of searches. Our evaluation confirms that our proposed optimization strategy picks the best plan shape for each query and improves performance.

4.3 Vector Instruction Usage for Analytics

Where Does the Time Go? Figure 7 shows a breakdown of the percentage of cycles spent for each type of vector instruction per

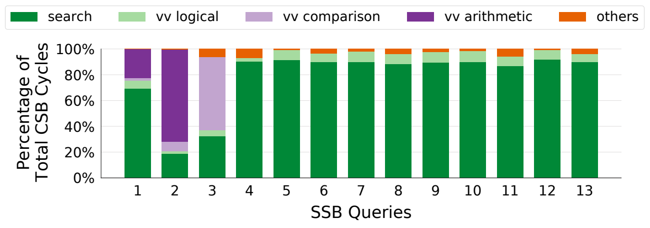


Figure 7: Breakdown of CSB cycles spent for different types of vector instructions ("vv" indicates vector-vector instructions, as opposed to vector-scalar, "vs", like searches).

SSB query. We observe two clear cases: queries 2 and 3 are dominated by *arithmetic* and *comparison* instructions, while queries 1 and 4-13 are dominated by *search* instructions.

What is the Most Expensive Database Operator? We grouped the cycles spent by database operator (join, aggregation, selection) across the 13 SSB queries. The execution cost of the SSB queries is dominated by joins: 96% of the cycles required to execute all 13 queries are spent on performing joins (this behavior is common in many analytic workloads).

Given the usage pattern of vector instructions by Castle, we can improve the overall performance of analytics on CAPE by optimizing the vector instructions and operators most commonly used by analytic queries. In Section 5, we propose three database-aware microarchitectural improvements motivated by the results presented in this section.

5 MICROARCHITECTURAL ENHANCEMENTS FOR DATABASE PROCESSING

In this section, we discuss three CAPE microarchitectural enhancements motivated by the results of Section 4.

5.1 Adaptive Bitwidth for Arithmetic

As shown in Section 4.3, vector-vector arithmetic and comparison instructions dominate the performance of queries 2 and 3. Arithmetic and comparison vector instructions are generally bit-serial (see Table 1). Thus, when operands can be represented using smaller bitwidths, the cost of arithmetic and comparison instructions is reduced. In SSB, most relation columns are stored in a 32-bit representation, but often values do not use the full range (e.g., require 5 bits instead of 32).

To improve the performance of these instruction types, we propose an *Adaptive Bitwidth Arithmetic* (ABA) scheme that dynamically discovers and uses the minimum bitwidth required for each vector instruction. Our approach has two phases: Phase 1 discovers the bitwidth required. Phase 2 configures CAPE's microcode sequencer[6] to use the new discovered bitwidth and sign-extends the results after completing the operation. In the worst case, the minimum required bitwidth is the same as the bitwidth defined by the relation column representation (in SSB, 32 bits, most of the time).

[6]In CAPE, the vector control unit contains a microcode sequencer that produces search-update pairs to implement vector instructions [15].

Reducing the required bitwidth can significantly reduce the cost of expensive bit-serial operations. For example, ABA reduces the cost of a 32-bit multiplication from 4,224 to 80 (or 976) cycles if the discovered required bitwidth for both (or one of the) operands is 4 bits. We support mixed bitwidths, where different operands in an instruction require different bitwidths (that is, in a ``vmul v2<-v0*v1'', v0 requires a different bitwidth than v1). ABA produces an exact result and does not incur any precision loss by reducing the bitwidth. CAPE offers a set of Control Status Registers (CSRs) that are used for various settings, using configuration instructions (e.g., `vsetvl`). ABA uses one of the CSRs to store the required bitwidth used by the critical bit-serial operations.

Phase 1, the discovery of the required bitwidth, can be performed in two ways. The first option is collaborative with the database system, which calculates min/max statistics per column at ingestion time (most database systems do it by default [54]). These statistics can be used to set the bitwidth of the vector instructions appropriately. The second option is to embed the discovery phase in the instruction itself, necessary when operating on columns for which statistics may not have been collected (i.e., columns generated as intermediate results in a query). If the DBMS does not set a bitwidth, the bit-serial instructions may embed the discovery phase every time they are executed. In this case, we perform a sequence of parallel searches to discover the required bitwidth for our operands.

To reduce the bitwidth used for an operation to b operand bits, in position b and higher should all be 1 (a negative number) or all 0 (a non-negative number). Bitwidth discovery is made by searching for all zeroes `0...0XXXX` (and all ones `1...1XXXX`), masking the lower bits. We keep trying lower bitwidths until the search for all 1s or 0s is not successful. To reduce the overhead of bitwidth discovery, we limit the number of guesses to a subset of possible bitwidths: 4, 8, 16, and 32 bits.

Finally, the operands are sign-extended by performing a series of bit-serial updates. First, we check the most significant bit of the required bitwidth (bit 3 for a 4-bit, 3 to 0, configuration) by searching for 0 (and 1) and updating with 0 (and 1) all the rows that match in the most significant bits.

In summary, ABA reduces the cost of bit-serial vector instructions. Section 6 shows that ABA achieves speedups between 1.03× and 1.2× for queries 1-3, compared to a rigid 32-bit scheme.

5.2 Adaptive Data Layout

This section presents a method for optimizing the delay of search instructions. CAPE uses a bitsliced data layout which, among other benefits, guarantees operand locality of vector operands required by most types of vector instructions [15]. This design choice targets general-purpose applications that involve a significant amount of vector-vector instructions (e.g., `vadd.vv`) rather than vector-scalar instructions (e.g. `vmseq.vx` aka search) [15]. Unlike in general-purpose applications, search instructions dominate the performance of analytic workloads (Figure 7). We propose a new data layout to accelerate search instructions and an adaptive method that switches between data layouts.

Bitslicing to Guarantee Operand Locality— As covered in Section 2.2, CAPE's CSB is organized in memory subarrays (shown as blue boxes in Figure 8) [15]. CAPE's subarrays are organized in

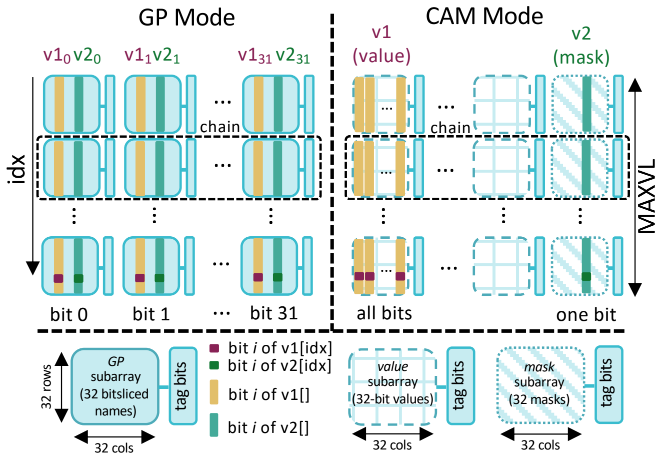


Figure 8: Physical mapping to the associative arrays found in the CSB on a General-Purpose Mode (left) and on a CAM Mode (right). On GP Mode, all subarrays store one bit of each vector register name. On CAM Mode, value subarrays may store any contiguous 32-bit value while the mask subarray is logically allocated to store only mask operands.

chains that include a chain logic that enables the accumulation of intermediate tag bits for various instructions. A search microoperation generates a mask stored in the subarray's tag bits to indicate math/mismatch per row.

CAPE bitslices vector elements: bit *i* of all vector register names is stored in the same subarray [15]. Figure 8-left shows how vectors `v1` and `v2` are mapped to the subarrays in the CSB. The rest of the vector register names in the RISC-V alphabet (`v0, v3-v31`) are mapped to the CSB the same way. Bitslicing guarantees that the operands of any vector instruction will be stored in the same subarray. For example, bit 0 of `v1` and `v2` will be in the left-most subarray, bit 1 in the next subarray, and so on. In this paper, we call the execution mode using this bitsliced layout *General-Purpose (GP) Mode*.

Search in a Bitsliced Layout is Expensive— Figure 9-left shows a search (vector-scalar equality comparison) of a key in a vector in CAPE (`vmseq.vx` instruction). Each bit of the key *i* is distributed to the appropriate subarray that contains the same bit index *i* of the vector register names. The search initially produces partial results as each subarray generates a bit-level result: is bit *i* of the key equal to bit *i* of the column operand? But to determine a full value match, these partial results must be combined. CAPE's solution is to leverage a shared tag bit bus that bit-serially accumulates the partial results into a final mask using shared chain logic, leading to a performance cost relative to the value bitdwith (33 cycles on a 32-bit configuration) [15].

To Bitslice or Not To Bitslice— Analytic queries are highly sensitive to the cost of searches. An alternative data layout that better matches the processing pattern of searches is to store all bits of a value contiguously in one subarray (Figure 9-right). Using this *contiguous* layout, a search produces the final result in the tag bits of a subarray in a single cycle and takes three cycles to complete in the CSB.

The drawback of this contiguous data layout is that it is not performant for bit-serial instructions as two operands would be stored in different subarrays. Each search/update microoperation would now require three additional cycles to transfer the two intermediate tag bits to the chain logic and the resulting tag bits back to the destination subarray, and most bit-serial instructions perform hundreds or thousands of the search/update microoperations (see Table 1).

Hence, we propose an *Adaptive Data Layout* (ADL) that switches between:

- **GP Mode:** values are bitsliced to preserve operand locality and achieve optimal performance for arithmetic operations (Figure 8, left) [15]. The VCU interprets values in bitsliced data layout and produces control signals to each subarray accordingly.
- **CAM Mode:** values are stored contiguously in a single subarray. Searches are performed efficiently in a single *value subarray* (1 cycle), followed by a copy of the final mask from the local tag bits to the chain register (1 cycle), and a final transfer to the *mask subarray*, which contains all the masks produced until now (1 cycle) (Figure 8, right).

Programs can switch back and forth between these two modes with minimal overhead while leveraging the full potential of the hardware for all instructions.

Switching Modes— Similar to other configuration instructions found in the standard RISC-V vector extension, such as `vsetvl` [62], we propose `vsetdl`, which allows for switching between the GP and CAM modes. We envision the database system to insert the switch instructions based on the upcoming operators in the execution flow. If the microarchitecture does not support layout switching, the instruction is decoded into a no-operation (*no-op*), and CAPE remains in GP Mode. This instruction is completed in a single cycle after it graduates from the CP's pipeline.

Preserving Information Across Layouts— When a `setdl` instruction is committed, the values inside the CSB are interpreted in the new layout. Therefore, data columns present in the memory before the `setdl` instruction will contain corrupted data after

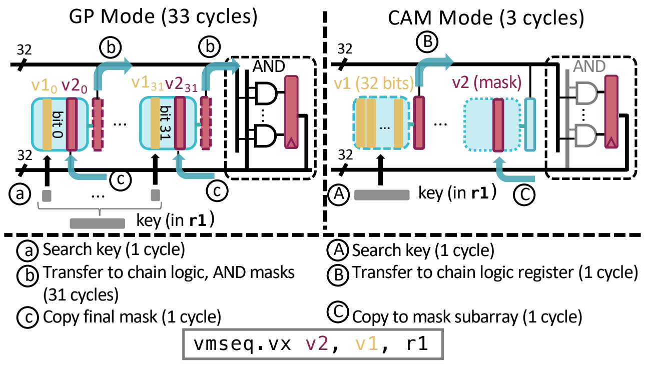


Figure 9: Illustrative chain of subarrays with the sequence of steps to realize a search instruction (via `vmseq.vx`) on GP Mode (left), and CAM Mode (right).

the change. Columns loaded after the `setdl` instructions will be loaded following the new data layout. To maintain data across switches, column operands need to be stored back to main memory and loaded again with the new layout—this strategy incurs a data movement cost. However, we have observed that, in analytic query processing, most of the time, data columns are not reused across computations that are performed in different modes, but only masks are. Therefore, data backup and restoration through main memory is, in practice, rarely needed.

To support the preservation of mask information across modes, we propose a new instruction, `vrelayout`, which copies a mask from mode A to B. After a `vrelayout` instruction is executed, subsequent instructions operating in the new execution mode can use the mask. The cost of this new instruction is two cycles regardless of the source and destination mode.

A `vrelayout` instruction that transferring mask from GP Mode to CAM Mode is done in two steps: 1) search 1 on the column where tha mask is stored (and mask out the rest) to echo the mask to the tag bits. Note that, in GP mode, all subarrays have a replica of the mask, and any replica can be used for the relayout. Within the same cycle time, we can transfer the tag bits to the chain logic. Finally, 2) transfer the tag bits from the chain logic to the appropriate mask subarray and use it to perform a predicated update (copy tag bits into the subarray). The reverse operation, to relayout a mask from CAM Mode to GP Mode would perform the same steps in exact reverse order. The only difference would be that the final update would happen in lock-step to all subarrays in a chain.

In CAM Mode, vector loads move vector operands into one subarray of every chain. Since we reserve one subarray per chain for mask storage, the rest of the subarrays can flexibly be used to store on any vector register. We use a simple register renaming scheme that includes a free list of subarrays and assigns physical subarrays to vector registers as instructions are sent to the VCU. The subarrays are freed when there is a layout switch to GP Mode or when there is a new vector load to a used vector register. In the unusual case of using more vector registers than memory subarrays, the compiler will employ traditional register spilling techniques[7]. The hardware support consists of a small 64-byte CAM and the associated logic to manage it.

In summary, ADL transforms the search instructions from bit-serial to bit-parallel, reducing their delay significantly. Configuration instructions for switching between layouts or to preserve masks across layouts are lightweight (1 cycle), and `vrelayout` can be used to preserve masks across layout switches (2 cycles). In Section 6, we show that, on average, SSB is 1.5× faster with ADL.

5.3 Multi-Key Search for Joins

In this section, we propose a technique to optimize the execution of joins via a dedicated multi-key search vector instruction. Joins in Castle search for multiple keys from the probe side relation in a vector containing a partition of the other relation by calling the `vmseq.vx` instruction for each probe key (Section 3.2). We propose a new multi-key search instruction that exposes this pattern to CAPE, specifically "`vmks v4, v3, keys[], numkeys`", searches for `numkeys` keys (stored consecutively starting at memory address `keys[]`) in vector `v3`, and produces a mask in `v4`.

[7]In our benchmark suite, we have observed that there are no more than four vector loads within a CAM Mode region, which does not register spill in a conventional CAPE design point.

The vmks instruction flows through all the pipeline stages of the control processor, like any other vector instruction, and is sent to the vector memory unit (VMU) [15], which initiates a request for the `numkeys` keys starting at address `keys[]` from main memory. Similar to the other vector memory instructions in CAPE, the CSB waits until the memory request is completely served. Keys are stored in a buffer in the VMU before they are transferred to the CSB subarrays, which perform the searches. Once `numkeys` keys are fetched to the buffer, they are distributed to all the subarrays that contain `v3`, using the existing global distribution network [15].

The internal bandwidth for transferring search keys to CSB subarrays is limited to a single key per cycle, thus search keys are consumed sequentially by the CSB. The memory request is not overlapped with the distribution of search keys and searches. The cost of `vmks` is: $Cycles(vmks) = M + numkeys + 2$. Here, `M` cycles are the delay of the memory request, `numkeys` cycles are needed to distribute and search `numkeys` keys, and two additional cycles are needed to copy the mask from the tag bits to the destination vector in the subarrays. Normally, after each search instruction, the tag bits would be moved to the destination vector (here, `v4`), which would result in each search instruction costing one cycle (to search) and two cycles (to move the tag bits to a vector). Since vmks will perform a sequence of searches (for `numkeys` keys) in the same subarray consecutively, we can leverage the existing OR functionality in the local tag bits [15] to combine the intermediate masks, in-situ, at no extra performance cost and only move the result bit mask once to `v4`, for each `numkeys` keys we search for.

The multi-key search instruction embeds more processing per fetch-decode overhead compared to conventional search instructions `vmseq.vx`. Also, joins that use `vmks` need fewer transfers of intermediate masks (tag bits to destination vector) because the intermediate masks are combined (ORed) in the same subarray. The `numkeys` factor should be decided carefully. One important design parameter is the buffer size: larger `numkeys` factors require larger buffers. However, a buffer smaller than a cacheline underutilizes the system’s data transfers since CAPE’s VMU operates at a cacheline size. Moreover, as we will see in Section 6, the number of keys from the probe side in end-to-end queries can be fewer than a cacheline. In those cases, the database system will use conventional search instructions `vmseq.vx`.

To conclude, multi-key searches pack more compute per fetch-decode instance, which leads to an average speedup on SSB of 1.4× over the queries with conventional search instructions. In Section 6, we discuss the design considerations of the buffer required to support `vmks`.

6 EVALUATION OF CAPE’S MICROARCHITECTURAL ENHANCEMENTS

In this section, we evaluate the database-aware microarchitectural changes of CAPE that we presented in Section 5. We use the experimental setup and methodology described in Section 4.

6.1 Experimental Setup

Area Considerations– For multi-key search support (Section 5.3), we evaluate buffer sizes of 64 bytes, 512 bytes, and 2 Kbytes which can store up to 16, 128, and 512 32-bit keys. The buffer is integrated as part of the vector memory unit (VMU), thus, on the same SRAM technology node as the CAPE core [15]. We estimate the area of the buffer storage using the High Performance SRAM bitcell size in 7 nm technology (0.032 μm^2) [1]. As a result, the buffers take 16.384 , 131.072 , and 1048.576 μm^2. Compared to a CAPE core area, 8.8 mm^2 [15], we consider the area overhead of the considered buffer sizes negligible.

Power Considerations– Similarly to [15], we break down each microoperation into the several steps required (wordline activation, cell access, bitline level amplification, etc.) and obtain the delay and energy (including both dynamic and leakage) using SPICE and CACTI [2]. CSB's leakage power is 0.48 W [15]. The CP power is estimated to be similar to a 20nm Cortex-A53 core with 256 kB L2 [6] (269 mW at 1.3 GHz [3]). Since CAPE is designed in 7nm, we scale the power by multiplying with the frequency ratio 2.7/1.3 (using CAPE's and Cortex-A53's frequencies, respectively), resulting in CP's power being 155 mW. We expect that the power burned by the VCU and VMU is negligible compared to the CP and even more so compared to the CSB. In conclusion, we find that CAPE's TDP is 16.39 W, including both leakage and dynamic power. [8] The microarchitectural enhancements proposed in Section 5 do not increase CAPE's power estimations, as we discuss below.

The discovery phase of ABA (Section 5.1) realizes a sequence of bit-serial searches which do not surpass the worst-case micro-operation's power (16.23 W). ABA may reduce the execution time of some instructions, leading to a reduction in the overall energy consumed. ABA uses updates to sign extend the final result in a bit-serial way (rather than updating all sign bits in a single cycle which burns significantly more power) to maintain CAPE's original TDP at a minimal performance overhead (e.g., up to 16 cycles on instructions that take hundreds or thousands of cycles).

ADL (Section 5.2) allows search operations to be executed in a single subarray while the rest of the subarrays in a chain are idle (and their peripherals can be power-gated). Inside each sub-array, more columns are searched, which does not increase the TDP (compared to single-column searches used in bitsliced mode) as the peripherals cannot be disabled at such low granularity. In total, ADL saves power by power-gating the peripherals of the idle subarrays.

Finally, MKS (Section 5.3) adds a buffer for the probe keys but performs searches in the same way as ADL, resulting in the same TDP. MKS actually reduces the number of fetches and decodes for joins; therefore, it reduces the overall energy consumed.

Overall, the microarchitectural enhancements proposed in Section 5 do not increase CAPE's original TDP (16.39 W) [15]. While this is less than 3× higher than the TDP reported for the isoarea baseline (5.63 W), Castle obtains 10.8× speedup on average using the same silicon area. Moreover, the actual power consumption during regular operation is significantly lower.

[8]TDP estimates in the original CAPE paper were intentionally very conservative upper bounds; here we use tighter upper-bound estimates.

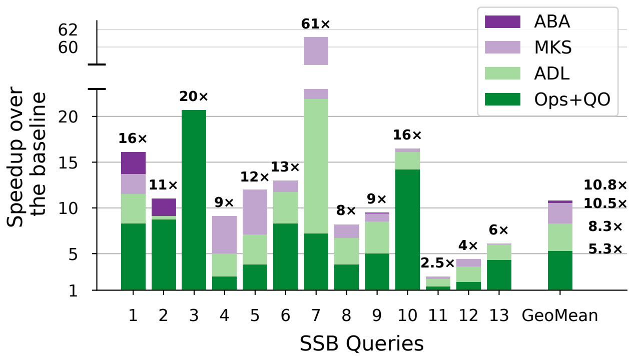


Figure 10: Speedup of Castle for SSB queries SF=1 compared to the baseline. Individual query speedups are truncated to improve the figure's readability.

Software Changes– We extend Castle to set the required bitwidth when it is found to be smaller than the default on the input columns, and to insert layout switching (`setdl`), mask relayout (`vrelayout`) when appropriate, as presented in Section 5.2. We changed Castle's join operator to consider multi-key search instructions (`vmks`) when appropriate.

Model of the Microarchitectural Enhancements– We extend the instruction-level model used in [15] to model the bitwidth discovery and the new cost of the reduced bitwidth instructions of the ABA scheme (Section 5.1) and incorporated the new costs into the gem5 model. We also extend the gem5 to model the new instructions for the ADL and MKS optimizations presented in Section 5.

6.2 Results on the SSB Benchmark

Figure 10 shows the improvement in Castle's performance with the microarchitectural enhancements compared to the baseline for all queries of the SSB benchmark at scale factor 1 (Section 4.1). We observe speedups between 2.5× and 61.1×, while the geometric mean of the speedups is 10.8×. We now break down this speedup according to each microarchitectural enhancement's contribution.

Adaptive Data Layout (ADL)— With ADL, Castle is 8.2× faster than the baseline. The overhead from `vrelayout` (updates the layout of masks present in CAPE), `setdl` (changes the data layout) instructions, and from reloading vector registers is offset by the performance improvement of the search instructions.

Multi-Key Search (MKS)— Figure 10 shows that, by adding multi-key searches in joins, the speedup of SSB further increases to 10.5× over the baseline. These results use a buffer size of 512 bytes which matches the cacheline size. The Castle join operator calls `vmks` to search for a set of `numkeys` keys, where the size of `numkeys` keys matches the size of the buffer. We have experimented with buffer sizes of 64 bytes and 2 kilobytes that lead to relative performance of 0.8× and 2.0× on average for all SSB queries, compared to the 512 bytes-sized buffer results. A smaller than cacheline buffer will waste memory bandwidth since CAPE's VMU requests are at cacheline size granularity.

We should note that not all SSB joins will take advantage of multi-key search instructions. For example, queries 2 and 3 sometimes probe for less than cacheline size number of keys and, as a result, would slow down with vmks due to unnecessary data movement. In these cases, vmks is not used. Further, SSB queries perform an aggregation after the joins, aggregations take as input the join output mask. Applying vmks to left-deep join subplans pivots the output mask and makes it incompatible with aggregations. Therefore we use vmks only on right-deep subplans.

Adaptive Bitdwith for Arithmetic (ABA)— Queries 1-2 show a significant benefit from ABA as they execute many expensive vector multiplications and comparisons (Figure 7). ABA speeds up queries 1-3 by 1.13× on average and increases the overall speedup to 10.8× of the SSB benchmark. Query 3 multiplies only a small number of elements and, thus, the Castle chooses not to use the expensive vector multiplication in favor of scalar multiplication (executed in the CP) most of the time; consequently, ABA has a small effect on this query (speedup of 1.03×).

6.3 Discussion on Data Movement

The unique performance characteristics of CAPE enable different execution strategies compared to traditional database systems (Section 3). These execution strategies, along with the ability of CAPE to retain intermediate results as masks, may lead to reduced memory usage. To execute SSB, the baseline transfers 1.51× more bytes compared to Castle. Thus, the reduced memory usage is beneficial to Castle, but it is not the only contributor to its performance advantage.

7 MICROBENCHMARK STUDY

In this section, we evaluate Castle using three microbenchmarks one for each of the three core operators found in analytic workloads: selection, aggregation, and join (Section 3). We use microbenchmarks to understand the different factors that affect Castle's performance by comparing them to highly-optimized baselines vectorized for AVX-512 (Section 4.1). We assume the same experimental setup as described in Section 4 for Castle as well as the baseline.

7.1 Selection

A selection operator filters rows based on a predicate (i.e. equality, inequality, etc). Our selection microbenchmark applies an equality predicate (relation column = val) to a column of 32-bit integers. We study the performance of Castle while varying the input size and the selectivity of the selection (how many rows satisfy the predicate). The baseline iterates over all rows of the input column to apply the predicate. Both the Castle and the baseline selection operators produce a bitmask that indicates if a row satisfied the predicate.

We experimented with inputs containing from 10^3 to 10^9 rows and varied selectivity from 1% up to 90%. Castle's speedup over the baseline ranges from 13× up to 22×, and it increases with the input size. The cost to load data from main memory to Castle or to the CPU dominates the running time of a selection. Castle's speedup increases for larger input sizes as CAPE's dedicated VMU efficiently performs data transfers compared to a CPU (as shown in [15]'s Figure 9). The CPU performs more work for larger selectivities (update the corresponding bit in the result bitmask per matching row), thus the overall speedup of Castle slightly increases with selectivity.

7.2 Join

To evaluate the performance of the Castle join operator, we join two relations (*fact* and *dimension*) on 32-bit integer join keys and vary the *fact* relation size ([1K, 100M] rows) and *dimension* relation size ([100, 20M] rows). We compare Castle's performance to a highly optimized hash join operator. The output of the join is a mask indicating the matching tuples of the *fact* relation (e.g. a semi-join). In Figure 11, each line corresponds to a different *fact* relation size and we vary the *dimension* relation size (x axis). In general, in analytic workloads one relation (the *fact* relation) is much larger than the other relation, typical size ratios are 1:12 or 1:16 [14]. We present two groups of lines, the dashed lines correspond to Castle without the micro-architectural optimizations of Section 5, the solid lines include the optimizations.

We make three key observations from the data shown in Figure 11. First, the Castle micro-architectural optimizations that target joins (ADL and MKS) result in significant speedups (5× in many cases). Second, the relative speedup over the CPU baseline is not significantly impacted by the *fact* relation size, the 1M and 10M fact relation size lines with the same optimization strategy are close to each other. Finally, for very large *dimension* relation sizes, such as 250K rows, the optimized Castle implementation and the optimized CPU implementation have similar performance. Essentially, for some join operations, and as we discuss below, these are likely to be rare in workloads that are similar to SSB, one may want to evaluate a join on the CPU. CAPE being closely integrated in a tiled architecture along other cores allows for a software architecture in which such decisions are made dynamically.

Next, let's discuss what portion of the space shown in Figure 11 is actually of interest when running SSB queries in our experiments. At a high level, most of the individual join operations in our workload experience performance gains that ranged from 2× to 70×. Interestingly, within a given query, separate join operations in the same query may see different individual gains, as different relations are involved in different joins each with its individual filter/selection predicate. For example, query 10 has three join operations.

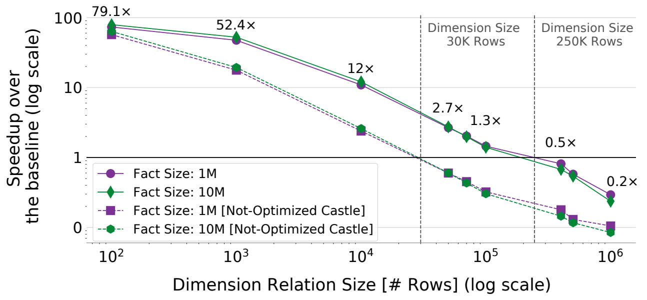


Figure 11: Speedup of Castle Join. The dimension relation size, measured in rows, is varied. Each line corresponds to a different fact relation size, measured in rows. Axes are in log scale. The speedup labels correspond to an input of 1M rows.

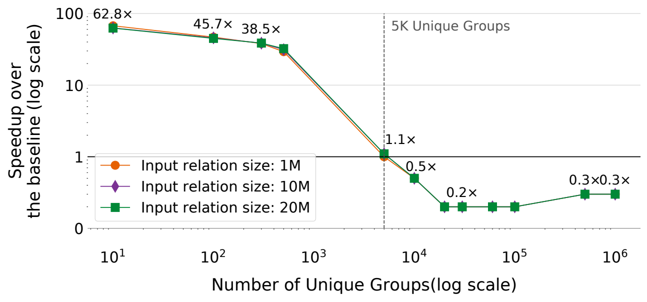


Figure 12: Speedup of Castle aggregation. The number of unique groups is varied. Each line corresponds to a different input relation size. Both axis are in logarithmic scale. The labels above the marker correspond to an input relation size of 15M rows.

At scale factor 1, the first join experiences a speedup of 2.4×, the second join experiences a 56× speedup, and the third join sees a speedup of 77×. (The query shows an overall speedup of 16×.) This behavior is quite natural (and is seen across other SSB queries too) as each join operation corresponds to a join with a probe side relation with a different size (here the *dimension*), resulting in different speedups for each join.

7.3 Aggregation

The aggregation microbenchmark implements an aggregation operator that performs a sum reduction for each group, similar to the example shown in Section 3.3. In Figure 12, we show the speedup of the Castle aggregation operator compared to a baseline hash-based aggregation operator, while varying the number of unique groups and input size.

The Castle aggregate operator performs a constant amount of work per group using CAPE's data-level parallelism (discovery of rows that belong to a group and calculation of the aggregate value, Section 3.3). As the number of groups increases each group contains less rows, thus the vector utilization is reduced, which translates into more work for the same input size. Thus, relative performance (speedup in the figure) is higher for small number of unique groups. For the SSB queries, the number of groups is within the range 3 to 600 and in this range the speedup of this microbenchmark is between 62× and 30×.

As shown in figure 12 when the number of groups is greater than 5,000, the baseline catches up to Castle, and beyond that the baseline is faster (so such aggregates are better evaluated on the CPU). Note that the speedup curve flattens to around 0.3× for very large number of groups because at this parameter space, the baseline CPU algorithm experiences a large (and increasing) number of cache misses in a core hash table data structure that it uses to compute the aggregate. Such very large number of unique groups is unusual in analytic queries, and as noted above in the SSB benchmark the maximum number of groups is 600.

7.4 Analysis

While microbenchmarks allow us to understand the sensitivity to different parameters in isolation, they do not represent how operators are organized in an end-to-end query. A critical additional factor that contributes to the end-to-end query performance is how a *sequence* of operators is executed. In Castle, the query execution methods deliberately aim to fuse consecutive operators to increase the "operational intensity" of data that is loaded in CAPE. So, if there a join operator followed by an aggregation operator, the aggregate computation is carried out eagerly on the (partial) join result. Such operator fusion strategy may result in the query achieving higher speedup than each individual operator running in isolation.

8 RELATED WORK

There is a large body of work that bridges architecture and database methods to speed up database workloads [8–13, 18, 31, 32, 39, 41, 43, 47, 53, 56, 60, 64, 65], which includes using the use of SIMD extensions [16, 37, 51, 52, 69] and repurposing GPUs for database workloads [24, 26, 30, 36, 44, 55]. There is also work on building specialized accelerators for database workloads [13, 35, 64, 65]. From a market adoption perspective, however, most database systems today run on commodity processors. Due to the importance of database workloads, it is possible that multiple hardware solutions may eventually find market adoption. In this paper, we offer CAPE as an attractive alternative that provides a balance between a generalized and a specialized approach: First, by leveraging CAPE's large vector primitives that database workloads need, and then by using a co-design approach that requires changes to database internals and CAPE.

Specialized solutions map high-level database operators to silicon, enabling very high throughput [13, 35, 64, 65]. Such approaches can support a predefined set of analytic operators very efficiently by trading off generality, as extending their capabilities requires a new hardware design. Further, highly specialized designs lack support for other workloads, which may hinder their potential adoption in datacenters. Castle leverages CAPE, which is a general-purpose core programmable via the RISC-V ISA. CAPE can accelerate a wide range of applications [15] and as shown in this paper it also accelerates database processing. The proposed microarchitectural additions (Section 5) do not sacrifice the generality or performance for non-data analytics workloads.

Other designs propose to offer flexible architecture templates that can be configured dynamically [9, 17, 18, 34, 43, 60]. These designs bring an interesting trade-off as they offer high throughput and generality. Some are based on reconfigurable hardware [17, 34], which requires programming in a hardware-descriptive language. Others use alternative languages and compilers for spatial architectures [60]. CPU-based solutions offer processor ISA as programming interfaces [9, 43], but none are based on associative processing, which brings unique features that match the core operations found inside data analytics (search and update). ReSQM proposes to use AP to map selection, sorting, and join [42] operators. However, the evaluation only considers the operators in isolation in comparison to a sub-optimal baseline and no end-to-end queries. Queries require updated query optimization strategies (Section 4) to leverage the potential of AP.

The processing-in-memory (PIM) and processing-in-storage (PIS) paradigms are promising concepts for data analytics [19–21, 25, 29, 38, 41, 42, 45, 49, 50, 57, 67]. While it is not the scope of this paper, associative processors could be designed as PIM or PIS and have the potential to leverage similar trade-offs as the mentioned proposals. The CAPE framework takes the same role as any conventional core in a tiled architecture, given its implementation in SRAM technology. Even at the moderate capacity design point studied, which operates by loading large datasets in MAXVL-sized partitions, we see speedups of order-of-magnitude scale compared to an comparable area CPU. We leave the design considerations of PIM or PIS flavors of associative processors for future work.

9 CONCLUSION

In this paper, we have explored efficient mappings of database operators used in analytics to associative processors (AP). We have shown that simply mapping those operators does not lead to performance improvements compared to a highly-optimized database running on an area-comparable out-of-order superscalar CPU with AVX-512. We have proposed an AP-aware query optimizer that produces optimal join orders, leading to a performance improvements of 5.3× compared to the AVX-512 baseline. Guided by an extensive study of bottlenecks of the optimized operators and query optimizer on a CAPE core, we have proposed microarchitectural enhancements to a CAPE core. The proposed additions to CAPE do not sacrifice the generality or performance of non-data analytics workloads. With the proposed software-hardware co-design, we demonstrate an overall performance improvement of 10.8× over the AVX-512 baseline.

ACKNOWLEDGMENTS

This work was supported in part by NSF and the Semiconductor Research Corporation (SRC) through the DEEP3M Center, part of the E2CDA program; and by DARPA and SRC through the CRISP Center, part of the JUMP program. Yannis Chronis is supported by a Facebook fellowship.

REFERENCES

[1] [n.d.]. 7 nm lithography process. https://en.wikichip.org/wiki/7_nm_lithography_process

[2] [n.d.]. CACTI 6.5. https://github.com/HewlettPackard/cacti. Accessed: 2019-08-12.

[3] [n.d.]. Cortex-A53 - Microarchitectures - ARM. https://en.wikichip.org/wiki/arm_holdings/microarchitectures/cortex-a53.

[4] [n.d.]. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference. Intel Corporation. 2015.

[5] [n.d.]. riscv-toolchain GitHub repository. https://github.com/riscv/riscv-gnu-toolchain.

[6] [n.d.]. Samsung Exynos 5433. https://en.wikichip.org/wiki/samsung/exynos/5433.

[7] 2021. Data Age 2025: The datasphere and data-readiness from edge to core. https://www.i-scoop.eu/big-data-action-value-context/data-age-2025-datasphere/

[8] Sandeep R. Agrawal, Christopher M. Dee, and Alvin R. Lebeck. 2016. Exploiting Accelerators for Efficient High Dimensional Similarity Search. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. Association for Computing Machinery, New York, NY, USA, Article 3, 12 pages. https://doi.org/10.1145/2851141.2851144

[9] Sandeep R Agrawal, Sam Idicula, Arun Raghavan, Evangelos Vlachos, Venkatraman Govindaraju, Venkatanathan Varadarajan, Cagri Balkesen, Georgios Giannikis, Charlie Roth, Nipun Agarwal, et al. 2017. A many-core architecture for in-memory data processing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 245–258.

[10] S. R. Agrawal, S. Idicula, A. Raghavan, E. Vlachos, V. Govindaraju, V. Varadarajan, C. Balkesen, G. Giannikis, C. Roth, N. Agarwal, and E. Sedlar. 2017. A Many-core Architecture for In-Memory Data Processing. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 245–258.

[11] Sandeep R. Agrawal, Valentin Pistol, Jun Pang, John Tran, David Tarjan, and Alvin R. Lebeck. 2014. Rhythm: Harnessing Data Parallel Hardware for Server Workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 19–34. https://doi.org/10.1145/2541940.2541956

[12] Cagri Balkesen, Nitin Kunal, Georgios Giannikis, Pit Fender, Seema Sundara, Felix Schmidt, Jarod Wen, Sandeep R. Agrawal, Arun Raghavan, Venkatanathan Varadarajan, Anand Viswanathan, Balakrishnan Chandrasekaran, Sam Idicula, Nipun Agarwal, and Eric Sedlar. 2018. RAPID: In-Memory Analytical Query Processing Engine with Extreme Performance per Watt. In *SIGMOD Conference*. ACM, 1407–1419.

[13] Jayanta Banerjee, David K. Hsiao, and Krishnamurthi Kannan. 1979. DBC: A Database Computer for Very Large Databases. *IEEE Computer Architecture Letters* 28, 06 (1979), 414–429.

[14] Spyros Blanas, Yinan Li, and Jignesh M. Patel. 2011. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-Core CPUs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/1989323.1989328

[15] H. Caminal, K. Yang, S. Srinivasa, A. K. Ramanathan, K. Al-Hawaj, T. Wu, V. Narayanan, C. Batten, and J. F. Martínez. 2021. CAPE: A Content-Addressable Processing Engine. In *2021 IEEE 27th International Symposium on High Performance Computer Architecture*.

[16] Jatin Chhugani, Anthony D Nguyen, Victor W Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. 2008. Efficient implementation of sorting on multi-core SIMD CPU architecture. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1313–1324.

[17] Eric S. Chung, John D. Davis, and Jaewon Lee. 2013. LINQits: Big Data on Little Clients. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. Association for Computing Machinery, New York, NY, USA, 261–272. https://doi.org/10.1145/2485922.2485945

[18] D.J. DeWitt, S. Ghandeharizadeh, D.A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. 1990. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering* 2, 1 (1990), 44–62. https://doi.org/10.1109/69.50905

[19] Jeff Draper, Jacqueline Chame, Mary Hall, Craig Steele, Tim Barrett, Jeff LaCoss, John Granacki, Jaewook Shin, Chun Chen, Chang Woo Kang, Ihn Kim, and Gokhan Daglikoca. 2002. The Architecture of the DIVA Processing-in-memory Chip. In *Proceedings of the 16th International Conference on Supercomputing*.

[20] D. G. Elliott, W. M. Snelgrove, and M. Stumm. 1992. Computational RAM: A Memory-SIMD Hybrid and its Application to DSP. In *1992 Proceedings of the IEEE Custom Integrated Circuits Conference*.

[21] D. G. Elliott, M. Stumm, W. M. Snelgrove, C. Cojocaru, and R. Mckenzie. 1999. Computational RAM: implementing processors in memory. *IEEE Design Test of Computers* (1999).

[22] Caxton C. Foster. 1976. *Content Addressable Parallel Processors*. John Wiley & Sons, Inc.

[23] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting worst-case optimal joins in relational database systems. *Proceedings of the VLDB Endowment* 13, 12 (2020), 1891–1904.

[24] Emily Furst, Mark Oskin, and Bill Howe. 2017. Profiling a GPU database implementation: a holistic view of GPU resource utilization on TPC-H queries. In *Proceedings of the 13th International Workshop on Data Management on New Hardware*. 1–6.

[25] M. Gokhale, B. Holmes, and K. Iobst. 1995. Processing in memory: the Terasys massively parallel PIM array. *Computer* (1995).

[26] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. 2006. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 325–336.

[27] Goetz Graefe. 1993. Query evaluation techniques for large databases. *ACM Computing Surveys (CSUR)* 25, 2 (1993), 73–169.

[28] Goetz Graefe and Leonard D Shapiro. 1990. *Data compression and database performance*. University of Colorado, Boulder, Department of Computer Science.

[29] Nastaran Hajinazar, Geraldo F. Oliveira, Sven Gregorio, João Dinis Ferreira, Nika Mansouri Ghiasi, Minesh Patel, Mohammed Alser, Saugata Ghose, Juan Gómez-Luna, and Onur Mutlu. 2021. SIMDRAM: A Framework for Bit-Serial SIMD Processing Using DRAM. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 329–345. https://doi.org/10.1145/3445814.3446749

[30] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. 2008. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 511–524.

[31] Tayler H. Hetherington, Mike O'Connor, and Tor M. Aamodt. 2015. MemcachedGPU: Scaling-up Scale-out Key-Value Stores. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15)*. Association for Computing

Machinery, New York, NY, USA, 43–57. https://doi.org/10.1145/2806777.2806836

[32] T. H. Hetherington, T. G. Rogers, L. Hsu, M. O'Connor, and T. M. Aamodt. 2012. Characterizing and evaluating a key-value store application on heterogeneous CPU-GPU systems. In *2012 IEEE International Symposium on Performance Analysis of Systems Software*. 88–98. https://doi.org/10.1109/ISPASS.2012.6189209

[33] Yannis E Ioannidis and Stavros Christodoulakis. 1991. On the propagation of errors in the size of join results. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of data*. 268–277.

[34] Zsolt István. 2019. The Glass Half Full: Using Programmable Hardware Accelerators in Analytics. *IEEE Data Eng. Bull.* 42, 1 (2019), 49–60.

[35] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. 2015. BlueDBM: An Appliance for Big Data Analytics. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/2749469.2750412

[36] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. 2012. GPU join processing revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*. 55–62.

[37] Changkyu Kim, Tim Kaldewey, Victor W Lee, Eric Sedlar, Anthony D Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1378–1389.

[38] Peter M. Kogge. 1994. EXECUBE-A New Architecture for Scaleable MPPs. In *Proceedings of the 1994 International Conference on Parallel Processing - Volume 01 (ICPP '94)*. IEEE Computer Society, USA, 77–84. https://doi.org/10.1109/ICPP.1994.108

[39] G. Koo, K. K. Matam, T. I., H. V. K. G. Narra, J. Li, H. Tseng, S. Swanson, and M. Annavaram. 2017. Summarizer: Trading Communication with Computing Near Storage. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 219–231.

[40] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 204–215. https://doi.org/10.14778/2850583.2850594

[41] Marzieh Lenjani, Patricia Gonzalez, Elaheh Sadredini, Shuangchen Li, Yuan Xie, Ameen Akel, Sean Eilert, Mircea R. Stan, and Kevin Skadron. 2020. Fulcrum: A Simplified Control and Access Mechanism Toward Flexible and Practical In-Situ Accelerators. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 556–569. https://doi.org/10.1109/HPCA47549.2020.00052

[42] Huize Li, Hai Jin, Long Zheng, and Xiaofei Liao. 2020. ReSQM: Accelerating Database Operations Using ReRAM-Based Content Addressable Memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 4030–4041.

[43] A. Lottarini, J. P. Cerqueira, T. J. Repetti, S. A. Edwards, K. A. Ross, M. Seok, and M. A. Kim. 2019. Master of None Acceleration: A Comparison of Accelerator Architectures for Analytical Query Processing. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. 762–773.

[44] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1633–1649. https://doi.org/10.1145/3318464.3389705

[45] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. 2000. Smart Memories: a modular reconfigurable architecture. In *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*. 161–171. https://doi.org/10.1109/ISCA.2000.854387

[46] Amir Morad, Leonid Yavits, Shahar Kvatinsky, and Ran Ginosar. 2016. Resistive GP-SIMD Processing-In-Memory. *ACM Trans. Archit. Code Optim.* (2016).

[47] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. 2020. A Modern Primer on Processing in Memory. *arXiv preprint arXiv:2012.03112* (2020).

[48] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The star schema benchmark and augmented fact table indexing. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer.

[49] M. Oskin, F. T. Chong, and T. Sherwood. 1998. Active Pages: a computation model for intelligent memory. In *Proceedings. 25th Annual International Symposium on Computer Architecture.*

[50] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. 1997. A case for intelligent RAM. *IEEE Micro* (1997).

[51] Orestis Polychroniou, Arun Raghavan, and Kenneth A Ross. 2015. Rethinking SIMD vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1493–1508.

[52] Orestis Polychroniou and Kenneth A Ross. 2013. High throughput heavy hitter aggregation for modern SIMD processors. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*. 1–6.

[53] V. Salapura, T. Karkhanis, P. Nagpurkar, and J. Moreira. 2012. Accelerating business analytics applications. In *IEEE International Symposium on High-Performance Comp Architecture*. 1–10. https://doi.org/10.1109/HPCA.2012.6169044

[54] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. 1989. Access path selection in a relational database management system. In *Readings in Artificial Intelligence and Databases*. Elsevier, 511–522.

[55] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1617–1632. https://doi.org/10.1145/3318464.3380595

[56] Malcolm Singh and Ben Leonhardi. 2011. Introduction to the IBM Netezza warehouse appliance. In *CASCON*. IBM / ACM, 385–386.

[57] H. S. Stone. 1970. A Logic-in-Memory Computer. *IEEE Trans. Comput.* (1970).

[58] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. 2018. C-store: a column-oriented DBMS. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*. 491–518.

[59] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. 2018. C-store: a column-oriented DBMS. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*. 491–518.

[60] Matthew Vilim, Alexander Rucker, Yaqi Zhang, Sophia Liu, and Kunle Olukotun. 2020. Gorgon: Accelerating Machine Learning from Relational Data. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 309–321. https://doi.org/10.1109/ISCA45697.2020.00035

[61] Benjamin Wagner, André Kohn, and Thomas Neumann. 2021. Self-Tuning Query Scheduling for Analytical Workloads. In *Proceedings of the 2021 International Conference on Management of Data*. 1879–1891.

[62] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanovic. 2014. The RISC-V Instruction Set Manual. Volume I User-level ISA. https://www.amd.com/en/products/cpu/amd-epyc-7502.

[63] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. 2000. The Implementation and Performance of Compressed Databases. *SIGMOD Rec.* 29, 3 (sep 2000), 55–67. https://doi.org/10.1145/362084.362137

[64] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross. 2015. The Q100 Database Processing Unit. *IEEE Micro* 35, 3 (2015), 34–46. https://doi.org/10.1109/MM.2015.51

[65] S. Xu, T. Bourgeat, T. Huang, H. Kim, S. Lee, and A. Arvind. 2020. AQUOMAN: An Analytic-Query Offloading Machine. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 386–399. https://doi.org/10.1109/MICRO50266.2020.00041

[66] L. Yavits, A. Morad, and R. Ginosar. 2015. Computer Architecture with Associative Processor Replacing Last-Level Cache and SIMD Accelerator. *IEEE Trans. Comput.* (2015).

[67] Yi Kang, Wei Huang, Seung-Moon Yoo, D. Keen, Zhenzhou Ge, V. Lam, P. Pattnaik, and J. Torrellas. 1999. FlexRAM: toward an advanced intelligent memory system. In *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors*.

[68] Y. Zha and J. Li. 2020. Hyper-AP: Enhancing Associative Processing Through A Full-Stack Optimization. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture*.

[69] Jingren Zhou and Kenneth A Ross. 2002. Implementing database operations using SIMD instructions. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. 145–156.