Accelerating Database Analytics
Using an Associative Processor

Helena Caminal   Yannis Chronis   Tianshu Wu   Jignesh Patel   José Martínez

This work was supported in part by the Semiconductor Research Corporation (SRC) and DARPA.
A continuous effort to process **more data** and answer **complex queries**
Domain Specific

Efficient

ASIC

General-Purpose

Flexible, programmable

CPU
Domain Specific

Efficient

ASIC

Near-data

General-Purpose

Flexible, programmable

CPU

Efficient

ASIC

+ ASIC

+ SanDisk

+ SanDisk

+ CPU
Domain Specific

Efficient

ASIC

General-Purpose

Flexible, programmable

SIMD

CPU

Near-data

SanDisk

ASIC

SanDisk

ASIC
Domain Specific

- Efficient
- ASIC
- Near-data

General-Purpose

- Flexible, programmable
- GPU
- SIMD
- CPU
- Near-data

Efficient

Flexible, programmable

ASIC

GPU

SIMD

CPU

Near-data

SanDisk

SanDisk
Desiderata

Performance

Process complex queries on very large volumes of data

General programming abstraction

Allow SW innovation and future-proofing
Target evolving domains

Integration into existing systems

Accelerates adoption
Encourages heterogeneous computing
Associative Processors

**Large Data-Level Parallelism**
Massively data-parallel
General-purpose operations (e.g. vadd)

**Processing Using Memory**
Based on two primitive operations: search, update
Analytical processing backed by a CAPE core
Associative Processing 101
Associative Processing Primitives

**Memories**
Indexing by address

**Read**

**Write**
Associative Processing Primitives

Memories
Indexing by address

Read

Write

Associative Memories
Indexing by value

Search

Update
Vector Instructions

Craft more sophisticated operations:

- vector add
- vector mul
- ... vector XOR
- vector AND
- ... vector find MIN
- vector N-way split
- ...

Indexing by value

- Search
- Update

0 1 3

2
Addition Using Associative Processing

\[ \text{addi } x5, x6, 1 \]

(\(d\)) (\(a\))

<table>
<thead>
<tr>
<th>(c_i)</th>
<th>(a_i)</th>
<th>(c_{i+1})</th>
<th>(d_i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1 0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1 1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

(\(AND\))(\(XOR\))

Two-bit incrementer (adder)

most-significant bit (lsb) \(\rightarrow\) least-significant bit (lsb)

Carry bit propagation
Addition Using Associative Processing

\textit{vaddi} v5, v6, 1

(d) (a)

<table>
<thead>
<tr>
<th>search</th>
<th>update</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_i a_i</td>
<td>c_{i+1} d_i</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>1 0</td>
<td>0 1</td>
</tr>
<tr>
<td>1 1</td>
<td>1 0</td>
</tr>
</tbody>
</table>

Combinational incrementer

Associative array

<table>
<thead>
<tr>
<th>c</th>
<th>a_1</th>
<th>a_0</th>
<th>d_1</th>
<th>d_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

Tag Bits

Search/Update

\[ a = \{0, 1, 2\} \]
\[ d = \{1, 2, 3\} \]

\[ d[k] = a[k] + 1; \text{ } k = 0, \ldots, 2 \]
Addition Using Associative Processing

\[ \text{vaddi} \ v5, \ v6, \ 1 \]

<table>
<thead>
<tr>
<th>d</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td>update</td>
</tr>
<tr>
<td>c_i a_i</td>
<td>c_{i+1} d_i</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>1 0</td>
<td>0 1</td>
</tr>
<tr>
<td>1 1</td>
<td>1 0</td>
</tr>
</tbody>
</table>

Optimizations:
- initialize \( d_i = 0 \)

Associative array

<table>
<thead>
<tr>
<th>c</th>
<th>a_1</th>
<th>a_0</th>
<th>d_1</th>
<th>d_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Tag Bits

Search/Update

vaddi v5, v6, 1

\( a = \{0, 1, 2\} \)

\( d = \{1, 2, 3\} \)

\( d[k] = a[k] + 1; k = 0, \ldots, 2 \)
Addition Using Associative Processing

\[ \text{vaddi } v5, v6, 1 \]

<table>
<thead>
<tr>
<th>(c_i)</th>
<th>(a_i)</th>
<th>(c_{i+1})</th>
<th>(d_i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Optimizations:
- initialize \(d_i=0\)
- \(c_i = c_{i+1}\)

Associative array

\[ \begin{array}{ccc}
  c & a_1 & a_0 \\
  d & d_1 & d_0 \\
  1 & 0 & 0 \\
  1 & 0 & 1 \\
  1 & 1 & 0 \\
\end{array} \]

Search/Update

Opt: initialize \(d_i=0\)
Addition Using Associative Processing

\[ \text{vaddi } v5, v6, 1 \]

\[
\begin{array}{c|c|c}
\text{c} & \text{a} & \text{d} \\
\hline
0 & 0 & 0 \\
0 & 1 & 0 \\
1 & 0 & 0 \\
1 & 1 & 1 \\
\end{array}
\]

two search/update pairs

Optimizations:
- initialize \( d_i = 0 \)
- \( c_i = c_{i+1} \)
## Addition Using Associative Processing

**vaddi** \( v_{5}, v_{6}, 1 \)

(d) (a)

<table>
<thead>
<tr>
<th>( c_{i} )</th>
<th>( a_{i} )</th>
<th>( c_{i+1} )</th>
<th>( d_{i} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>0 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>0 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>1 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Optimizations:
- Initialize \( d_{i} = 0 \)
- \( c_{i} = c_{i+1} \)

**Associative array**

<table>
<thead>
<tr>
<th>( c )</th>
<th>( a_1 )</th>
<th>( a_0 )</th>
<th>( d_1 )</th>
<th>( d_0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0</td>
<td>0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 1</td>
<td>0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 0</td>
<td>0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Tag Bits**

- \( 0 \)
- \( 0 \)
- \( 0 \)

**Search**

- 0
- X
- 1
- X
- X

Operating on bit 0
Addition Using Associative Processing

\[ \text{vaddi } v5, v6, 1 \]

Optimizations:
- initialize \( d_i = 0 \)
- \( c_i = c_{i+1} \)

Operate on bit 0

Update

Search

Associative array

Truth Table:

<table>
<thead>
<tr>
<th>( c )</th>
<th>( a_1 )</th>
<th>( a_0 )</th>
<th>( d_1 )</th>
<th>( d_0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(No matches)
Addition Using Associative Processing

$\text{vaddi v5, v6, 1}$

$\begin{array}{c|c|c|c|c}
\text{c}_i & \text{a}_i & \text{c}_{i+1} & \text{d}_i \\
0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 \\
\end{array}$

Optimizations:
- initialize $d_i=0$
- $c_i = c_{i+1}$

Associative array

$\begin{array}{c|c|c|c|c|c|c|c|c|c}
\text{c} & \text{a}_1 & \text{a}_0 & \text{d}_1 & \text{d}_0 \\
1 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 \\
\end{array}$

Operating on bit 0

Associative array

Tag
Bits

Search

Addition Using Associative Processing

\[ \text{vaddi } v5, v6, 1 \]

\[
\begin{array}{c|c|c|c|c|c|c}
\text{c} & \text{a} & \text{d} \\
\hline
0 & 0 & 0 \\
0 & 1 & 0 \ 
1 & 0 & 1 \\
1 & 1 & 1 \\
\end{array}
\]

Optimizations:
- initialize \( d_i = 0 \)
- \( c_i = c_{i+1} \)

Associative array

Bit 0 of all elements done!

Associative Addition Using Associative Processing
Addition Using Associative Processing

```plaintext
vaddi v5, v6, 1
(d) (a)
```

<table>
<thead>
<tr>
<th>$c_i$</th>
<th>$a_i$</th>
<th>$c_{i+1}$</th>
<th>$d_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Optimizations:
- $d_i=0$
- $c_i = c_{i+1}$

Operating on bit 1

```
Optimizations:
• initialize $d_i=0$
• $c_i = c_{i+1}$
```
Addition Using Associative Processing

\[ \text{vaddi } v5, v6, 1 \]

<table>
<thead>
<tr>
<th>( c_i )</th>
<th>( a_i )</th>
<th>( c_{i+1} d_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0 1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0 1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1 1</td>
</tr>
</tbody>
</table>

Optimizations:
- initialize \( d_i = 0 \)
- \( c_i = c_{i+1} \)

Associative array

Operating on bit 1

Addition Using Associative Processing

\[ \text{vaddi } v5, v6, 1 \]

<table>
<thead>
<tr>
<th>( c )</th>
<th>( a_1 )</th>
<th>( a_0 )</th>
<th>( d_1 )</th>
<th>( d_0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Update

Operating on bit 1
Addition Using Associative Processing

\[ \text{vaddi } v5, v6, 1 \]

<table>
<thead>
<tr>
<th>( c_i )</th>
<th>( a_i )</th>
<th>( c_{i+1} )</th>
<th>( d_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

Optimizations:
• initialize \( d_i = 0 \)
• \( c_i = c_{i+1} \)

Associative array

- \( c \), \( a_1 \), \( a_0 \), \( d_1 \), \( d_0 \)
- Tag
- Bits
- Search

Operating on bit 1

\( a = \{0, 1, 2\} \)
\( d = \{1, 2, 3\} \)
\( d[k] = a[k] + 1; \) for \( k = 0, \ldots, 2 \)
Addition Using Associative Processing

\[ \text{vaddi } v5, v6, 1 \]

<table>
<thead>
<tr>
<th>( c_i )</th>
<th>( a_i )</th>
<th>( c_{i+1} )</th>
<th>( d_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>0 1</td>
<td>0 1</td>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>1 0</td>
<td>0 1</td>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
</tr>
</tbody>
</table>

**Optimizations:**
- initialize \( d_i = 0 \)
- \( c_i = c_{i+1} \)

**Associative array**

All bits completed!
Castle

Data Analytics on a Modern Associative Processor
CAPE A modern associative processor

Content-Addressable Processing Engine

General-purpose

RISC-V RV64G

Area comparable to one OoO core

Vector Architecture of $2^{15}$ vector elements
CAPE A modern associative processor

Content-Addressable Processing Engine

[HPCA21: CAPE- A Content-Addresable Processing Engine]
Castle Database Operators

Selection  Aggregation  Join
Castle Database Operators

Selection   Aggregation   Join

HW/SW Codesign Optimizations

Query Optimization   Vector vs Scalar Processing   Data Format
Adaptive Arithmetic   Adaptive Data Layout   Multi-Key Search
Evaluation Setup

gem5: cycle-accurate simulator

Area-equivalent baseline

SSB: 13 queries
Scale Factor 1 (~600MB)
183x larger than CAPE’s CSB

64GB DDR4 Main memory (153.6 GB/s)
Castle vs AVX-512 [SSB SF 1]

![Castle Speedup with HW/SW codesign](chart)

**Speedup**

- Castle Speedup with HW/SW codesign

**SSB Queries**

1. 16×
2. 11×
3. 20×
4. 12×
5. 13×
6. 61×
7. 8×
8. 9×
9. 16×
10. 2.5×
11. 4×
12. 6×
13. 10.8×

GeoMean
Castle vs AVX-512 [SSB SF 1]

![Graph showing speedup for Castle vs AVX-512 for SSB SF 1 queries]

- **Castle Speedup with HW/SW codesign**
- **Castle database operators**

**SSB Queries**

- **Speedup**
  - 16×
  - 11×
  - 20×
  - 9×
  - 12×
  - 13×
  - 61×
  - 8×
  - 9×
  - 16×
  - 2.5×
  - 4×
  - 6×
  - 10.8×

The GeoMean is represented at the bottom of the graph.
Castle Database Operators

Selection  Aggregation  Join

HW/SW Codesign Optimizations

Query Optimization  Vector vs Scalar Processing  Data Format
Adaptive Arithmetic  Adaptive Data Layout  Multi-Key Search
Join

R Relation  S Relation

R ⋈ S
Joins are accelerated by using indexes
Fast searches vs hashtable build

CPU

Castle

loading

hashtable

${\text{Cost(build)}} + |\text{probe side}| \text{ searches}$

${\text{Load}} + |\text{probe side}| \text{ searches}$

search($R \bowtie S$)

vector

search($R \bowtie S$)
Join

search( )

vector

R \bowtie S

Castle
Join

search(Castle)

vector

#partitions = \frac{|\text{larger relation}|}{\text{vector size}}

|probe side| * #partitions

searches

Block-Nested Loop like
Castle Database Operators

Selection    Aggregation    Join

HW/SW Codesign Optimizations

Query Optimization

Vector vs Scalar Processing

Adaptive Arithmetic

Adaptive Data Layout

Data Format

Multi-Key Search
Query Optimization

Left-deep

fact
[6M rows]

⋈

d1
[3K rows]

⋈

d2
[20K rows]
Query Optimization

The smaller relation should be the probe side
Query Optimization

The smaller relation should be the probe side

cost: #searches depends on relation sizes and vector size (#partitions)

*assume a vector size of 32,768
Query Optimization

Left-deep

- `fact` [6M rows]
  - `d1` [3K rows]
  - `d2` [20K rows]
  - Cost: 6,200,000 searches

Right-deep

- `fact`
  - `d1`
  - `d2`
  - `|d1| * (fact partitions)`
  - 6,000,000 searches
  - 3,680,000 searches
  - 552,000 searches
  - Cost: 4,232,000 searches

Cost does not depend on the order of joins

*assume a vector size of 32,768*
Query Optimization

Decide join order + join direction -> # searches is minimized

Cost: 6,200,000 searches

Cost: 4,232,000 searches
does not depend on the order of joins

Cost: 752,000 searches

*assume a vector size of 32,768
Castle Database Operators

Selection    Aggregation    Join

HW/SW Codesign Optimizations

Query Optimization    Vector vs Scalar Processing    Data Format
Adaptive Arithmetic    Adaptive Data Layout    Multi-Key Search
Performance Evaluation

Instructions type used (%)

Bit-serial instructions
- vector multiplications
- vector comparisons

Searches

SSB Queries

1 2 3 4 5 6 7 8 9 10 11 12 13

25 % 25 % 25 % 25 % 25 % 25 % 25 % 25 % 25 % 25 % 25 % 25 % 25 %

75 % 75 % 75 % 75 % 75 % 75 % 75 % 75 % 75 % 75 % 75 % 75 % 75 %

50 % 50 % 50 % 50 % 50 % 50 % 50 % 50 % 50 % 50 % 50 % 50 % 50 %

100 % 100 % 100 % 100 % 100 % 100 % 100 % 100 % 100 % 100 % 100 % 100 % 100 %
Adaptive Data Layout

Bit-sliced Mode (General-Purpose Layout)

Contiguous Mode

associative array

Logical View

v1

v2

32,768 rows

v1[idx]

v2[idx]

Physical View

vseq.vx v2, v1, key (33 cycles)

Chain Merge Logic

Chain Bus

32

intermediate results

32 cycles

trace tag bits

copy back (1 cycle)

bit 0

bit 1

bit 31

associative arithmetic requires locality at the bit level

search (vseq.vx) serially merge (AND) intermediate results

49
Adaptive Data Layout

Bit-sliced Mode (General-Purpose Layout)

Contiguous Mode (Improves search performance)

Logical View

Physical View

vseq.vx v2, v1, key (3 cycles)

copy tag bits (2 cycles)  Chain Merge Logic

search (vseq.vx) copies tag bits (final result)
Programmability of Adaptive Layout

Adaptive Data Layout (ADL) switches between:

**bitsliced mode**
efficient arithmetic / logic operations

**contiguous mode**
efficient search / logic operations

**Instructions:**
Reconfiguration: `setdl(mode)` (1 cycle)
Mask preservation: `vrelayout(v2)` (3 cycles)
Value preservation: Regular vector stores/loads (spilling)
HW/SW Codesign Optimizations

Query Optimization

Adaptive Data Layout

Multi-Key Search

Adaptive Bitwidth for Arithmetic

Data Format

Vector vs Scalar Processing
HW/SW Codesign Optimizations

Query Optimization

Adaptive Data Layout

**Multi-Key Search**

Make the HW aware of SW processing patterns

Adaptive Bitwidth for Arithmetic

Data Format

Vector vs Scalar Processing
HW/SW Codesign Optimizations

Query Optimization

Adaptive Data Layout

**Multi-Key Search**
Make the HW aware of SW processing patterns

**Adaptive Bitwidth for Arithmetic**
Adapt precision to the data types dynamically

Data Format

Vector vs Scalar Processing
Evaluation Recap
Castle vs AVX-512 [SSB SF 1]

- Adaptive Bitwidth Arithmetic
- Multi-key Search
- Adaptive Data Layout
- Castle operators + Query optimization

Speedup

SSB Queries

- Castle vs AVX-512
- SSB SF 1
- GeoMean
Analytics constantly need more performance
Golden era of architecture, HW-SW Codesign

Associative processors can accelerate multiple applications and offer analytic-“friendly” massively parallel primitives. A domain specific architecture for data processing?

Castle: prototype associative data processing system

**HW-SW codesign** opportunities

Up to 61X faster on SSB - GeoMean 11X
Questions
The Castle Team

Helena Caminal    Yannis Chronis    Tianshu Wu    Jignesh Patel    José Martínez