APL SIMD Boolean Array Algorithms

Robert Bernecky

Snake Island Research Inc, Canada
bernecky@snakeisland.com

October 5, 2016
Abstract

Computation on large Boolean arrays is becoming more prevalent, due to applications such as cryptography, data compression, and image analysis and synthesis. The advent of bit-oriented vector extensions for microprocessors and of GPUs presents opportunities for significant performance improvements in such Boolean-dominated applications. Since APL is one of the few computer languages that supports dense (one bit per element, eight bits per byte), multi-dimensional Boolean arrays as first-class objects, it has naturally attracted research into optimizations for improved performance of Boolean array operations. This paper presents some of the Single Instruction, Multiple Data (SIMD) Boolean-related optimizations that have appeared in APL implementations, and suggests ways in which those optimizations might be exploited using contemporary hardware.
A BIT of Introduction

- The bit: the fundamental unit of digital computing
A BIT of Introduction

- The bit: the fundamental unit of digital computing
- Yet, few computer languages treat bits as basic data types
A BIT of Introduction

- The bit: the fundamental unit of digital computing
- Yet, few computer languages treat bits as basic data types
- Fewer support multi-dimensional bit arrays (8 bits/byte)
A BIT of Introduction

- The bit: the fundamental unit of digital computing
- Yet, few computer languages treat bits as basic data types
- Fewer support multi-dimensional bit arrays (8 bits/byte)
- Fewer yet provide array operations on Boolean arrays
A BIT of Introduction

- The bit: the fundamental unit of digital computing
- Yet, few computer languages treat bits as basic data types
- Fewer support multi-dimensional bit arrays (8 bits/byte)
- Fewer yet provide array operations on Boolean arrays
- Boolean arrays appear in image analysis, cryptography, data compression...
A BIT of Introduction

- The bit: the fundamental unit of digital computing
- Yet, few computer languages treat bits as basic data types
- Fewer support multi-dimensional bit arrays (8 bits/byte)
- Fewer yet provide array operations on Boolean arrays
- Boolean arrays appear in image analysis, cryptography, data compression…
- The burden of bit twiddling is left to the programmer
The bit: the fundamental unit of digital computing
Yet, few computer languages treat bits as basic data types
Fewer support multi-dimensional bit arrays (8 bits/byte)
Fewer yet provide array operations on Boolean arrays
Boolean arrays appear in image analysis, cryptography, data compression...
The burden of bit twiddling is left to the programmer
APL, however, simply treats Booleans as the integers 0 and 1
A BIT of Introduction

- The bit: the fundamental unit of digital computing
- Yet, few computer languages treat bits as basic data types
- Fewer support multi-dimensional bit arrays (8 bits/byte)
- Fewer yet provide array operations on Boolean arrays
- Boolean arrays appear in image analysis, cryptography, data compression...
- The burden of bit twiddling is left to the programmer
- APL, however, simply treats Booleans as the integers 0 and 1
- Boolean arrays are grist to APL’s data-parallel, expressive mill!
Why Does APL have One-bit Booleans?

- Blame Larry Breed: while designing APL\360,
Why Does APL have One-bit Booleans?

- Blame Larry Breed: while designing APL\360,
- Breed decided to store Booleans densely, eight bits/byte
Why Does APL have One-bit Booleans?

- Blame Larry Breed: while designing APL\360,
- Breed decided to store Booleans densely, eight bits/byte
- Booleans were stored in row-major order, as are other arrays
Why Does APL have One-bit Booleans?

- Blame Larry Breed: while designing $APL\backslash360$,
- Breed decided to store Booleans densely, eight bits/byte
- Booleans were stored in row-major order, as are other arrays
- This eased indexing, structural and selection verbs, etc.
Why Does APL have One-bit Booleans?

- Blame Larry Breed: while designing APL\360,
- Breed decided to store Booleans densely, eight bits/byte
- Booleans were stored in row-major order, as are other arrays
- This eased indexing, structural and selection verbs, etc.
- Single-bit indexing was more expensive than word indexing...
Why Does APL have One-bit Booleans?

- Blame Larry Breed: while designing APL\360,
- Breed decided to store Booleans densely, eight bits/byte
- Booleans were stored in row-major order, as are other arrays
- This eased indexing, structural and selection verbs, etc.
- Single-bit indexing was more expensive than word indexing...
- But it opened the door to SIMD Boolean array optimizations
Why Does APL have One-bit Booleans?

- Blame Larry Breed: while designing *APL\360*,
- Breed decided to store Booleans densely, eight bits/byte
- Booleans were stored in row-major order, as are other arrays
- This eased indexing, structural and selection verbs, etc.
- Single-bit indexing was more expensive than word indexing... 
- But it opened the door to SIMD Boolean array optimizations
- Those optimizations are the subject of this talk
Why Does APL have One-bit Booleans?

- Blame Larry Breed: while designing $APL\backslash 360$,
- Breed decided to store Booleans densely, eight bits/byte
- Booleans were stored in row-major order, as are other arrays
- This eased indexing, structural and selection verbs, etc.
- Single-bit indexing was more expensive than word indexing... 
- But it opened the door to SIMD Boolean array optimizations
- Those optimizations are the subject of this talk
- Speedups were usually 8X or 32X, but sometimes even more
Why Does APL have One-bit Booleans?

- Blame Larry Breed: while designing APL\360,
- Breed decided to store Booleans densely, eight bits/byte
- Booleans were stored in row-major order, as are other arrays
- This eased indexing, structural and selection verbs, etc.
- Single-bit indexing was more expensive than word indexing. . .
- But it opened the door to SIMD Boolean array optimizations
- Those optimizations are the subject of this talk
- Speedups were usually 8X or 32X, but sometimes even more
- A half century later, Breed’s decision remains brilliant
Why Does APL have One-bit Booleans?

- Blame Larry Breed: while designing \texttt{APL\textbackslash 360},
- Breed decided to store Booleans densely, eight bits/byte
- Booleans were stored in row-major order, as are other arrays
- This eased indexing, structural and selection verbs, etc.
- Single-bit indexing was more expensive than word indexing. . .
- But it opened the door to SIMD Boolean array optimizations
- Those optimizations are the subject of this talk
- Speedups were usually 8X or 32X, but sometimes even more
- A half century later, Breed’s decision remains brilliant
- These optimizations are still important and relevant
Why Does APL have One-bit Booleans?

- Blame Larry Breed: while designing APL\360,
- Breed decided to store Booleans densely, eight bits/byte
- Booleans were stored in row-major order, as are other arrays
- This eased indexing, structural and selection verbs, etc.
- Single-bit indexing was more expensive than word indexing. . .
- But it opened the door to SIMD Boolean array optimizations
- Those optimizations are the subject of this talk
- Speedups were usually 8X or 32X, but sometimes even more
- A half century later, Breed’s decision remains brilliant
- These optimizations are still important and relevant
- GPU and SIMD vector facilities can exploit them
Scalar Verbs

- Breed optimized many rank-0 (scalar) Boolean verbs e.g.
Scalar Verbs

- Breed optimized many rank-0 (scalar) Boolean verbs e.g.
- Boolean verbs: $\land, \lor, \neg, \ominus, \mp \ldots$
Scalar Verbs

- Breed optimized many rank-0 (scalar) Boolean verbs e.g.
- Boolean verbs: $\land, \lor, \neg, \cong, \forall, \exists$.
- Relational verbs: $<, \leq, =, \geq, >, \neq$.
Scalar Verbs

- Breed optimized many rank-0 (scalar) Boolean verbs e.g.
- Boolean verbs: ∧, ∨, ~, ⊤, ⊥...
- Relational verbs: <, ≤, =, ≥, >, ≠
- SIMD application, a word at a time (32 bits on S/360)
Scalar Verbs

- Breed optimized many rank-0 (scalar) Boolean verbs e.g.
- Boolean verbs: \( \land, \lor, \neg, \land, \lor \ldots \)
- Relational verbs: \( <, \leq, =, \geq, >, \neq \)
- SIMD application, a word at a time (32 bits on S/360)
- One or more of us optimized scalar extension, e.g.
Scalar Verbs

- Breed optimized many rank-0 (scalar) Boolean verbs e.g.
- Boolean verbs: $\land$, $\lor$, $\sim$, $\neg$, $\forall$, $\exists$, etc.
- Relational verbs: $<$, $\leq$, $=$, $\geq$, $>$, $\neq$
- SIMD application, a word at a time (32 bits on S/360)
- One or more of us optimized scalar extension, e.g.
- $1 \land B$ would produce $B$, 
Scalar Verbs

- Breed optimized many rank-0 (scalar) Boolean verbs e.g.
- Boolean verbs: \(\land, \lor, \sim, \&\&\), \(\&\) 
- Relational verbs: \(<, \leq, =, \geq, >, \neq\)
- SIMD application, a word at a time (32 bits on S/360)
- One or more of us optimized scalar extension, e.g.
  - \(1 \land B\) would produce \(B\),
  - without doing any element-wise computations
Strength Reduction: a Classic Compiler Optimization

- *Strength reduction*: replace one operation by a cheaper one
Strength Reduction: a Classic Compiler Optimization

- **Strength reduction**: replace one operation by a cheaper one
- *E.g.*, replace multiply by a power of two with a shift
Strength Reduction: a Classic Compiler Optimization

- *Strength reduction*: replace one operation by a cheaper one
- *E.g.*, replace multiply by a power of two with a shift
- In APL, Boolean $B1 \times B2$ becomes $B1 \wedge B2$
Strength Reduction: a Classic Compiler Optimization

- *Strength reduction:* replace one operation by a cheaper one
- *E.g.*, replace multiply by a power of two with a shift
- In APL, Boolean $B1 \times B2$ becomes $B1 \wedge B2$
- In APL, Boolean $B1 \backslash B2$ becomes $B1 \wedge B2$
Strength Reduction: a Classic Compiler Optimization

- *Strength reduction*: replace one operation by a cheaper one
- *E.g.*, replace multiply by a power of two with a shift
- In APL, Boolean $B1 \times B2$ becomes $B1 \wedge B2$
- In APL, Boolean $B1 \mathbin{\text{|}} B2$ becomes $B1 \wedge B2$
- In APL, Boolean $B1 \mathbin{\text{⋆}} B2$ becomes $B1 \geq B2$
Strength Reduction: a Classic Compiler Optimization

- **Strength reduction**: replace one operation by a cheaper one
- *E.g.*, replace multiply by a power of two with a shift
- In APL, Boolean \( B_1 \times B_2 \) becomes \( B_1 \land B_2 \)
- In APL, Boolean \( B_1 \land B_2 \) becomes \( B_1 \land B_2 \)
- In APL, Boolean \( B_1 \lor B_2 \) becomes \( B_1 \lor B_2 \)
- Performance boosts: simpler verbs, SIMD operation, no conditionals, stay in Boolean domain
Strength Reduction: a Classic Compiler Optimization

- **Strength reduction**: replace one operation by a cheaper one
- *E.g.*, replace multiply by a power of two with a shift
- In APL, Boolean $B_1 \times B_2$ becomes $B_1 \wedge B_2$
- In APL, Boolean $B_1 \land B_2$ becomes $B_1 \wedge B_2$
- In APL, Boolean $B_1 \ast B_2$ becomes $B_1 \geq B_2$
- Performance boosts: simpler verbs, SIMD operation, no conditionals, stay in Boolean domain
- **Performance boosts**: In a compiler, opportunity for other optimizations
Structural and Selection Verbs I

- catenate, laminate, rotate, reverse, rank, from,
Structural and Selection Verbs I

- catenate, laminate, rotate, reverse, rank, from,
- merge, take, drop...
Structural and Selection Verbs I

- catenate, laminate, rotate, reverse, rank, from,
- merge, take, drop...
- These verbs, e.g., 1 0 1, 0 1 1 0 have to handle array indices that are not byte-aligned
Structural and Selection Verbs I

- catenate, laminate, rotate, reverse, rank, from,
- merge, take, drop...
- These verbs, e.g., 1 0 1, 0 1 1 0 have to handle array indices that are not byte-aligned
- We would like these to run SIMD, word-at-a-time, on Booleans
Structural and Selection Verbs I

- catenate, laminate, rotate, reverse, rank, from,
- merge, take, drop...
- These verbs, e.g., 1 0 1, 0 1 1 0 have to handle array indices that are not byte-aligned
- We would like these to run SIMD, word-at-a-time, on Booleans
- We introduced rbemove: generalized stride-1 (ravel order) copier verb
Structural and Selection Verbs I

- catenate, laminate, rotate, reverse, rank, from,
- merge, take, drop . . .
- These verbs, e.g., 1 0 1, 0 1 1 0 have to handle array indices that are not byte-aligned
- We would like these to run SIMD, word-at-a-time, on Booleans
- We introduced rbemove: generalized stride-1 (ravel order) copier verb
- \( \text{snk}[\text{sni}+\text{rk}] \leftarrow \text{src}[\text{sri}+\text{rk}] \)
- catenate, laminate, rotate, reverse, rank, from,
- merge, take, drop...
- These verbs, e.g., 1 0 1, 0 1 1 0 have to handle array indices that are not byte-aligned
- We would like these to run SIMD, word-at-a-time, on Booleans
- We introduced rbemove: generalized stride-1 (ravel order) copier verb
- \text{snk}[\text{sni+\text{i}}k] \leftarrow \text{src}[\text{sri+\text{i}}k]
- Does not corrupt out-of-bounds array elements
Structural and Selection Verbs I

- catenate, laminate, rotate, reverse, rank, from,
- merge, take, drop...
- These verbs, e.g., 1 0 1, 0 1 1 0 have to handle array indices that are not byte-aligned
- We would like these to run SIMD, word-at-a-time, on Booleans
- We introduced rbemove: generalized stride-1 (ravel order) copier verb
- \( \text{snk}[\text{sni} + \text{\text{i}k}] \rightarrow \text{src}[\text{sri} + \text{\text{i}k}] \)
- Does not corrupt out-of-bounds array elements
- Operates in SIMD mode(s) whenever possible
Structural and Selection Verbs I

- catenate, laminate, rotate, reverse, rank, from,
- merge, take, drop...
- These verbs, e.g., 1 0 1, 0 1 1 0 have to handle array indices that are not byte-aligned
- We would like these to run SIMD, word-at-a-time, on Booleans
- We introduced rbemove: generalized stride-1 (ravel order) copier verb
- \( \text{snk}[\text{sni+1k}] \leftarrow \text{src}[\text{sri+1k}] \)
- Does not corrupt out-of-bounds array elements
- Operates in SIMD mode(s) whenever possible
- **Supports all type conversions**
Operation on non-trailing array axes:
Operation on non-trailing array axes:

- SIMD copy entire subarrays at once, e.g.

  `1 2 3 4 1 2 4`
Operation on non-trailing array axes:

SIMD copy entire subarrays at once, e.g.

\[ 1 \ 2 \ 3 \ 4 \ \Theta \ 24 \]

rbemove will copy 12 adjacent array elements at once
Structural and Selection Verbs III

\[ 0 \quad 1 \quad 2 \quad 3 \]
\[ 4 \quad 5 \quad 6 \quad 7 \]
\[ 8 \quad 9 \quad 10 \quad 11 \]
\[ 12 \quad 13 \quad 14 \quad 15 \]
\[ 16 \quad 17 \quad 18 \quad 19 \]
\[ 20 \quad 21 \quad 22 \quad 23 \]
Structural and Selection Verbs III

- $\texttt{2 3 4\&\textbackslash 24}$
  - 0 1 2 3
  - 4 5 6 7
  - 8 9 10 11

- $\texttt{1\@2 3 4\&\textbackslash 24}$
  - 12 13 14 15
  - 16 17 18 19
  - 20 21 22 23

- $\texttt{1\@2 3 4\&\textbackslash 24}$
  - 0 1 2 3
  - 4 5 6 7
  - 8 9 10 11
Bernecky, 1979: fast algorithms for $\phi \omega$ & $\alpha \phi \omega$
Reverse and Rotate on Booleans

- Bernecky, 1979: fast algorithms for $\phi \omega$ & $\alpha \phi \omega$
- last-axis Boolean $\phi \omega$ did a byte at a time, w/table lookup... 
  \[ \text{RevTab}[\text{uint8 } \omega] \]
Reverse and Rotate on Booleans

- Berneky, 1979: fast algorithms for $\phi \omega$ & $\alpha \phi \omega$
- last-axis Boolean $\phi \omega$ did a byte at a time, w/table lookup...
  RevTab[uint8 $\omega$]
- then byte-aligned the resulting vector, SIMD, a word at a time
Berneky, 1979: fast algorithms for $\phi \omega$ & $\alpha \phi \omega$

last-axis Boolean $\phi \omega$ did a byte at a time, w/table lookup...
RevTab[uint8 $\omega$]

then byte-aligned the resulting vector, SIMD, a word at a time

All non-last-axis operations copied entire cells at once, using rbemove
Reverse and Rotate Performance on Booleans

FIG 1
LAST AXIS 1 ROTATE
ON SQUARE MATRICES

SHARP APL TORONTO
TIME=04/21/79 21:11
USER LOAD=34

- - - - - OLD BOOLEAN
- - - OLD INTEGER
- - OLD FLOATING
- OLD CHARACTER
NEW BOOLEAN
NEW INTEGER
NEW FLOATING
NEW CHARACTER

NUMBER OF ELEMENTS IN ARRAY

Robert Bernecky
APL SIMD Boolean Array Algorithms
Reshape allows element reuse, e.g.:

8ρ1 0 0
1 0 0 1 0 0 1 0
Reshape allows element reuse, e.g.:

```
8 1 0 0
1 0 0 1 0 0 1 0
```

Breed optimized Boolean reshape this way:
Reshape

- Reshape allows element reuse, e.g.:
  \[
  \begin{pmatrix}
  8 & 1 & 0 & 0 \\
  1 & 0 & 0 & 1 & 0 & 0 & 1 & 0
  \end{pmatrix}
  \]

- Breed optimized Boolean reshape this way:

- Copy the argument to the result
Reshape

- Reshape allows element reuse, e.g.:
  \[ 8 \rho 1 \ 0 \ 0 \]
  \[ 1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \]

- Breed optimized Boolean reshape this way:
- Copy the argument to the result
- Catenate the partial result to itself, doubling its length, until its tail is byte-aligned.
Reshape

- Reshape allows element reuse, e.g.:
  \[
  8 \rho 1 \ 0 \ 0 \\
  1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0
  \]
- Breed optimized Boolean reshape this way:
- Copy the argument to the result
- Catenate the partial result to itself, doubling its length, until its tail is byte-aligned.
- Do an overlapped move, or “smear" of the result to its tail
An unlikely candidate for SIMD, it would seem...
An unlikely candidate for SIMD, it would seem...

αωω with unchanged trailing axes
An unlikely candidate for SIMD, it would seem...

αω with unchanged trailing axes

T←2 2 2 3 1 2 4

0 1 2
3 4 5

6 7 8
9 10 11

12 13 14
15 16 17

18 19 20
21 22 23
Transpose II

- Transpose with unchanged trailing axes
Transpose II

- Transpose with unchanged trailing axes
- SIMD copy six elements at once (rbemove)
Transpose II

- Transpose with unchanged trailing axes
- SIMD copy six elements at once (rbemove)

```
1  0  2  3
0  1  2
3  4  5
12 13 14
15 16 17
6  7  8
9 10 11
18 19 20
21 22 23
```
▶ **Hacker’s Delight**: fast 8×8 Boolean matrix transpose
Boolean Transpose III

- *Hacker’s Delight*: fast 8×8 Boolean matrix transpose
- Kernel: 16 logical & shift operations on 64-bit ravel
- *Hacker’s Delight:* fast 8×8 Boolean matrix transpose
- *Kernel:* 16 logical & shift operations on 64-bit ravel
- Uses *perfect shuffle (PDEP)* on any power of two shape
Boolean Transpose III

- *Hacker’s Delight*: fast 8×8 Boolean matrix transpose
- Kernel: 16 logical & shift operations on 64-bit ravel
- Uses *perfect shuffle* (PDEP) on any power of two shape
- Dyalog APL (Foad): 10X speedup on large Boolean array transpose
Boolean Transpose III

- *Hacker’s Delight*: fast 8×8 Boolean matrix transpose
- Kernel: 16 logical & shift operations on 64-bit ravel
- Uses *perfect shuffle* (PDEP) on any power of two shape
- Dyalog APL (Foad): 10X speedup on large Boolean array transpose
- Kernel generalizes to any power of two, e.g., 16×16, 32×32
Search Verbs

- Bernecky, 1971: fast `indexOf` and `set membership`
Search Verbs

- Bernecky, 1971: fast \textit{indexOf} and \textit{set membership}
- All data types except reals with $\square\circ t \neq 0$
Search Verbs

- Bernecky, 1971: fast `indexOf` and `set membership`
- All data types except reals with \( \subset \neq 0 \)
- \((a \text{;} 0 \text{;} 1)[\omega]\)
Search Verbs

- Bernecky, 1971: fast `indexof` and `set membership`
- All data types except reals with `∪ct≠0`
- `(αi0 1)[ω]`
- `(αi□av)[ω]`
Search Verbs

- Bernecky, 1971: fast `indexOf` and `set membership`
- All data types except reals with `ct ≠ 0`
- `(a ∘ 0 1)[ω]`
- `(a ∘ □av)[ω]`
- Booleans: Vector search for first byte of interest
Search Verbs

- Bernecky, 1971: fast `indexOf` and `set membership`
- All data types except reals with `ct≠0`
- `(α₀ 1)[ω]`
- `(α₀av)[ω]`
- Booleans: Vector search for first byte of interest
- Then, table lookup to get bit offset
Search Verbs

- Bernecky, 1971: fast *indexOf* and *set membership*
- All data types except reals with □c≠0
- (αι0 1)[ω]
- (αιιav)[ω]
- Booleans: Vector search for first byte of interest
- Then, table lookup to get bit offset
- **Speedup:** lots - linear time vs. quadratic time
Search Verbs

- Bernecky, 1971: fast *indexOf* and *set membership*
- All data types except reals with $\Box \epsilon \neq 0$
- $(a \downarrow 0 \ 1)[\omega]$
- $(a \downarrow \downarrow a \neg)[\omega]$
- Booleans: Vector search for first byte of interest
- Then, table lookup to get bit offset
- Speedup: lots - linear time vs. quadratic time
- Created *indexOf* kernel utility for interpreter use
Roger D. Moore, 1971: fast +/ω Boolean vector
Reduce

- Roger D. Moore, 1971: fast $+/\omega$ Boolean vector
- Initial use was *compress* and *expand* setup:
  $+/\alpha$ was taking longer than *compress/expand*
Reduce

- Roger D. Moore, 1971: fast $+/\omega$ Boolean vector
- Initial use was *compress* and *expand* setup:
  $+/\alpha$ was taking longer than *compress/expand*
- S/360 translate vector op: Boolean bytes into population counts, SIMD 124 bytes per segment
Reduce

- Roger D. Moore, 1971: fast $+\omega$ Boolean vector
- Initial use was *compress* and *expand* setup:
  $+/\alpha$ was taking longer than *compress/expand*
- S/360 translate vector op: Boolean bytes into population counts, SIMD 124 bytes per segment
- SIMD integer sum of 4-byte words gave 4-element partial sum
Roger D. Moore, 1971: fast $+\omega$ Boolean vector

Initial use was *compress* and *expand* setup:
$+\alpha$ was taking longer than *compress/expand*

S/360 translate vector op: Boolean bytes into population counts, SIMD 124 bytes per segment

SIMD integer sum of 4-byte words gave 4-element partial sum

Shift-and-add gave final result
Roger D. Moore, 1971: fast +/ω Boolean vector
Initial use was *compress* and *expand* setup:
+/
 was taking longer than *compress/expand*
S/360 translate vector op: Boolean bytes into population
counts, SIMD 124 bytes per segment
SIMD integer sum of 4-byte words gave 4-element partial sum
Shift-and-add gave final result
Segment size limited to prevent inter-byte carries
Reduce

- Roger D. Moore, 1971: fast $+/\omega$ Boolean vector
- Initial use was *compress* and *expand* setup:
  $+/\alpha$ was taking longer than *compress/expand*
- S/360 translate vector op: Boolean bytes into population counts, SIMD 124 bytes per segment
- SIMD integer sum of 4-byte words gave 4-element partial sum
- Shift-and-add gave final result
- Segment size limited to prevent inter-byte carries
- Larry Breed haiku:
  $+/+\neq 4$ resh PopcountTab[uint8 $\omega$]
Reduce

- Roger D. Moore, 1971: fast \(+/\omega\) Boolean vector
- Initial use was compress and expand setup:
  \(+/\alpha\) was taking longer than compress/expand
- S/360 translate vector op: Boolean bytes into population counts, SIMD 124 bytes per segment
- SIMD integer sum of 4-byte words gave 4-element partial sum
- Shift-and-add gave final result
- Segment size limited to prevent inter-byte carries
- Larry Breed haiku:
  \(+/+/\neq 4\text{ resh PopcountTab[uint8 } \omega]\)
- Algorithm used briefly for \(\vee/\omega\) and \(\wedge/\omega\)
Reduce and Scan

- E.E. McDonnell, 1974: elegant APL models of Boolean *scan* and *reduction* for relationals
Reduce and Scan

▶ E.E. McDonnell, 1974: elegant APL models of Boolean scan and reduction for relationals
▶ Result was catenation of prefix, infix, & suffix expressions
Reduce and Scan

- E.E. McDonnell, 1974: elegant APL models of Boolean *scan* and *reduction* for relationals
- Result was catenation of prefix, infix, & suffix expressions
- Used Bernecky’s fast indexof
Reduce and Scan

- E.E. McDonnell, 1974: elegant APL models of Boolean scan and reduction for relationals
- Result was catenation of prefix, infix, & suffix expressions
- Used Bernecky’s fast indexof
- Result: linear-time, word-at-a-time, SIMD Boolean scan & reduce
John Heckman, 1970 or 1971: user-defined APL scan verbs
Scan

- John Heckman, 1970 or 1971: user-defined APL scan verbs
- Now widely used in GPUs
Scan

- John Heckman, 1970 or 1971: user-defined APL scan verbs
- Now widely used in GPUs
- Recursive doubling:
  \[ r \leftarrow \text{nescanall } y ; s ; \text{biw} \]
  \[ \triangleright \text{Not-equal scan} \]
  \[ r \leftarrow y \]
  \[ \text{biw} \leftarrow [2 \otimes 1 \otimes y] \]
  \[ : \text{For } s : \text{In } 2 \uparrow \text{biw } \triangleright \text{Heckman} \]
  \[ r \leftarrow r \neq (-\rho r) \uparrow (-s) \downarrow r \]
  \[ : \text{EndFor} \]
Scan

- John Heckman, 1970 or 1971: user-defined APL scan verbs
- Now widely used in GPUs
- Recursive doubling:
  - r←nescanall y;s;biw
  - a Not-equal scan
  - r←y
  - biw←(2↓0)↓por y
  - :For s :In 2*↓biw a Heckman
  - r←r≠(-pr)↑(-s)↓r
  - :EndFor
- SIMD, word-at-a-time algorithm for Boolean $\neq \omega$ and $=\omega$ along last axis
John Heckman, 1970 or 1971: user-defined APL scan verbs
Now widely used in GPUs
Recursive doubling:
\[ r \leftarrow \text{nescanall y} ; s ; \text{biw} \]
\[ \triangleq \text{Not-equal scan} \]
\[ r \leftarrow y \]
\[ \text{biw} \leftarrow [2 \otimes 1 \otimes y] \]
\[ : \text{For } s : \text{In } 2 \times \text{biw} \triangleq \text{Heckman} \]
\[ r \leftarrow r \neq (-r) \uparrow (-s) \downarrow r \]
\[ : \text{EndFor} \]
SIMD, word-at-a-time algorithm for Boolean \( \neq \omega \) and \( = \omega \) along last axis
Bernecky's simple C Heckman implementation is about 3X faster than Dyalog APL 15.0 (vector only)
John Heckman, 1970 or 1971: user-defined APL scan verbs

Now widely used in GPUs

Recursive doubling:
\[
\begin{align*}
r &\leftarrow \text{nescanall } y; s; biw \\
& \quad \text{a Not-equal scan} \\
r &\leftarrow y \\
biw &\leftarrow 2 \otimes 1 \otimes \rho y \\
& \quad : \text{For } s : \text{In } 2 \times \downarrow biw \text{ a Heckman} \\
& \quad \quad \quad r \leftarrow r \neq (-\rho r) \uparrow (-s) \downarrow r \\
& \quad : \text{EndFor}
\end{align*}
\]

SIMD, word-at-a-time algorithm for Boolean \( \neq \omega \) and \( = \omega \) along last axis

Bernecky’s simple C Heckman implementation is about 3X faster than Dyalog APL 15.0 (vector only)

So far, no X86 vectorization; perhaps we can do even better
IPSA, 1973: Boolean array inner products were painfully slow
IPSA, 1973: Boolean array inner products were painfully slow

Control Data (CDC) wanted APL for their new STAR-100 vector supercomputer
- IPSA, 1973: Boolean array inner products were painfully slow
- Control Data (CDC) wanted APL for their new STAR-100 vector supercomputer
- Group from Toronto I.P. Sharp Associates hired to work on the interpreter
IPSAS, 1973: Boolean array inner products were painfully slow
Control Data (CDC) wanted APL for their new STAR-100 vector supercomputer
Group from Toronto I.P. Sharp Associates hired to work on the interpreter
Memory-to-memory vector instructions needed stride-1 access for good performance
STAR Inner Product I

- IPSA, 1973: Boolean array inner products were painfully slow
- Control Data (CDC) wanted APL for their new STAR-100 vector supercomputer
- Group from Toronto I.P. Sharp Associates hired to work on the interpreter
- Memory-to-memory vector instructions needed stride-1 access for good performance
- Bernecky: heard about STAR APL stride-1 inner-product algorithm;
IPSA, 1973: Boolean array inner products were painfully slow
Control Data (CDC) wanted APL for their new STAR-100 vector supercomputer
Group from Toronto I.P. Sharp Associates hired to work on the interpreter
Memory-to-memory vector instructions needed stride-1 access for good performance
Bernecky: heard about STAR APL stride-1 inner-product algorithm;
redesigned Boolean inner product to use STAR algorithm
Classic Inner Product Algorithm

Z←X ipclassic Y;RX;CX;CY;I;J;K
RX←(ρX)[0]
CX←(ρX)[1]
CY←(ρY)[1]
Z←(RX,CY)ρ0.5
:For I :In I RX
  :For J :In I CY
    Z[I;J]←0
    :For K :In I CX
      Z[I;J]←Z[I;J]+X[I;K]×Y[K;J]
    :EndFor
  :EndFor
:EndFor
STAR Inner Product Algorithm

\[
\begin{align*}
Z &\leftarrow X \text{ ipstar } Y; RX; CX; CY; I; J; Xel \\
RX &\leftarrow (\rho X)[0] \\
CX &\leftarrow (\rho X)[1] \\
CY &\leftarrow (\rho Y)[1] \\
Z &\leftarrow (RX, CY) \rho 0 \\
: &\text{For } I : \text{In } \triangleright RX \\
: &\text{For } J : \text{In } \triangleright CX \\
&Xel \leftarrow X[I; J] \\
&Z[I; ] \leftarrow Z[I; ] + Xel \times Y[J; ] \\
: &\text{EndFor} \\
: &\text{EndFor}
\end{align*}
\]
Inner product loops reordered; key benefits, for α, f, g, ω
Inner product loops reordered; key benefits, for $\alpha \neq g \neq \omega$

Each $\alpha$ element, $x \in l$, fetched only once
STAR Inner Product II

- Inner product loops reordered; key benefits, for $\alpha \in \mathbb{F} \cdot g \cdot \omega$
- Each $\alpha$ element, $xel$, fetched only once
- Type conversion of $xel$ no longer time-critical
STAR Inner Product II

- Inner product loops reordered; key benefits, for $\alpha \in \mathbb{F} \cdot \mathbb{G} \cdot \mathbb{W}$
- Each $\alpha$ element, $x_{el}$, fetched only once
- Type conversion of $x_{el}$ no longer time-critical
- $x_{el}$ analysis amortized over entire row: $Y[J;]$
Inner product loops reordered; key benefits, for $\alpha \neq g \neq \omega$

Each $\alpha$ element, $Xel$, fetched only once

Type conversion of $Xel$ no longer time-critical

$Xel$ analysis amortized over entire row: $Y[J;]$

Scalar-vector application of $g$

$\text{tmp} \leftarrow Xel \ g \ Y[J;]$
Inner product loops reordered; key benefits, for \( \alpha \ f \cdot g \ \omega 
\)

- Each \( \alpha \) element, \( xel \), fetched only once
- Type conversion of \( xel \) no longer time-critical
- \( xel \) analysis amortized over entire row: \( Y[J;] \)
- Scalar-vector application of \( g \)
  \[ \text{tmp} \leftarrow xel \ g \ Y[J;] \]
- Vector-vector \( f \)-reduce into result row \( Z[I;] \)
  \[ Z[I;] \leftarrow Z[I;] \ f \ \text{tmp} \]
Scalar-vector $X e l \ g \ Y[J;]$ is word-at-a-time Boolean SIMD
SIMD Boolean STAR Inner Product Basics

- Scalar-vector $X \in \mathbb{g} \ Y[J;]$ is word-at-a-time Boolean SIMD
- Vector-vector $Z[I;] \leftarrow Z[I;]$ is word-at-a-time Boolean SIMD
SIMD Boolean STAR Inner Product Basics

- Scalar-vector $X \text{el } Y[J;]$ is word-at-a-time Boolean SIMD
- Vector-vector $Z[I;] \leftrightarrow Z[I;]$ is word-at-a-time Boolean SIMD
- We are already a lot faster
SIMD Boolean STAR Inner Product Basics

- Scalar-vector $X e l g Y[J;]$ is word-at-a-time Boolean SIMD
- Vector-vector $Z[I;] \leftarrow Z[I;]$ is word-at-a-time Boolean SIMD
- We are already a lot faster
- The STAR APL model does $+.\times 90\times$ faster than the Classic model, on $200\times200$ real matrices
SIMD Boolean STAR Inner Product Basics

- Scalar-vector $X \in g \ Y[J;]$ is word-at-a-time Boolean SIMD
- Vector-vector $Z[I;] \leftarrow Z[I;]$ is word-at-a-time Boolean SIMD
- We are already a lot faster
- The STAR APL model does $+.\times$ 90X faster than the Classic model, on 200\times200 real matrices
- Unfortunately, the APL primitive is still 30X faster than the APL model
Consider $\text{tmp} \leftarrow \text{Xel} \land \text{RO}$ in $\alpha \lor \omega$
Boolean STAR Inner Product Optimizations

- Consider $\text{tmp} \leftarrow \text{Xel} \land \text{RO} \in \alpha v. \land \omega$

- If Xel is 0, then tmp is all zeros: no g computation
Consider `tmp←Xel ∧ RO in αν.∧ω`

- If `Xel` is 0, then `tmp` is all zeros: no `g` computation
- If `Xel` is 1, then `tmp` is just `RO`: no `g` computation
Consider $\text{tmp} \leftarrow \text{Xel} \land RO$ in $\alpha \lor \cdots \land \omega$

- If $\text{Xel}$ is 0, then $\text{tmp}$ is all zeros: no $g$ computation
- If $\text{Xel}$ is 1, then $\text{tmp}$ is just $RO$: no $g$ computation
- $f$ is $\lor$, so its identity element is 0
Boolean STAR Inner Product Optimizations

- Consider \( \text{tmp} \leftarrow \text{Xel} \land \text{RO} \) in \( \alpha \lor . \land \omega \)
- If \( \text{Xel} \) is 0, then \( \text{tmp} \) is all zeros: no \( g \) computation
- If \( \text{Xel} \) is 1, then \( \text{tmp} \) is just \( \text{RO} \): no \( g \) computation
- \( f \) is \( \lor \), so its identity element is 0
- Hence, if \( \text{Xel} \) is 0, we can skip the \( g \)-reduction
Boolean STAR Inner Product Optimizations

- Consider $\text{tmp} \leftarrow \text{Xel} \land \text{RO}$ in $\alpha \lor . \land \omega$
- If $\text{Xel}$ is 0, then $\text{tmp}$ is all zeros: no $g$ computation
- If $\text{Xel}$ is 1, then $\text{tmp}$ is just $\text{RO}$: no $g$ computation
- $f$ is $\lor$, so its identity element is 0
- Hence, if $\text{Xel}$ is 0, we can skip the $g$-reduction
- Similarly, if $\text{Xel}$ is 1, we can do the $g$-reduction using $\text{RO}$
Boolean STAR Inner Product Optimizations

- Consider \( \text{tmp} \leftarrow \text{Xel} \land \text{RO} \) in \( \alpha \lor \cdot \land \omega \)
- If \( \text{Xel} \) is 0, then \( \text{tmp} \) is all zeros: no \( g \) computation
- If \( \text{Xel} \) is 1, then \( \text{tmp} \) is just \( \text{RO} \): no \( g \) computation
- \( f \) is \( \lor \), so its identity element is 0
- Hence, if \( \text{Xel} \) is 0, we can skip the \( g \)-reduction
- Similarly, if \( \text{Xel} \) is 1, we can do the \( g \)-reduction using \( \text{RO} \)
- This gives us poor man’s sparse arrays, which works on other data types, too
Consider $\text{tmp} \leftarrow \text{Xel} \land \text{RO}$ in $\alpha \lor \land \omega$

- If $\text{Xel}$ is 0, then $\text{tmp}$ is all zeros: no $g$ computation
- If $\text{Xel}$ is 1, then $\text{tmp}$ is just $\text{RO}$: no $g$ computation
- $f$ is $\lor$, so its identity element is 0
- Hence, if $\text{Xel}$ is 0, we can skip the $g$-reduction
- Similarly, if $\text{Xel}$ is 1, we can do the $g$-reduction using $\text{RO}$
- This gives us poor man’s sparse arrays, which works on other data types, too

Final result: Boolean inner products on SHARP APL/PC ran much faster than APL2 on huge mainframe
Control Flow Becomes Data Flow

- Booleans as arithmetic: replace control flow by data flow
Control Flow Becomes Data Flow

▶ Booleans as arithmetic: replace control flow by data flow
▶ Conditionals can often be removed, e.g.:
Control Flow Becomes Data Flow

- Booleans as arithmetic: replace control flow by data flow
- Conditionals can often be removed, e.g.:
- Give those with salary, \( S \), less than Tiny a raise of \( R \):
  \[ S \leftarrow S + R \times S < \text{Tiny} \]
Control Flow Becomes Data Flow

- Booleans as arithmetic: replace control flow by data flow
- Conditionals can often be removed, e.g.:
  - Give those with salary, $S$, less than $\text{Tiny}$ a raise of $R$
    $S \leftarrow S + R \times S < \text{Tiny}$
- Knuth calls this capability \textit{Iverson’s convention for characteristic functions}
Control Flow Becomes Data Flow

- Booleans as arithmetic: replace control flow by data flow
- Conditionals can often be removed, e.g.:
  - Give those with salary, $S$, less than $\text{Tiny}$ a raise of $R$
    
    $S \leftarrow S + R \times S < \text{Tiny}$
  - Knuth calls this capability \textit{Iverson’s convention for characteristic functions}
- See also the verb $\text{mq}s$, for finding quoted text
Control Flow Becomes Data Flow

- Booleans as arithmetic: replace control flow by data flow
- Conditionals can often be removed, e.g.:
- Give those with salary, $S$, less than $\text{Tiny}$ a raise of $R$
  
  $S \leftarrow S + R \times S < \text{Tiny}$
- Knuth calls this capability $\text{l}verson’s$ $\text{c}onvention$ $\text{f}or$
  $\text{c}haracteristic$ $\text{f}unctions$
- See also the verb $\text{mq}S$, for finding quoted text
- See also the Bernecky-Scholz PLDI2014 Arrays Workshop
  paper: $Abstract$ $Expressionism$
Boolean Sort

- Sort ascending for Booleans:
  \[ \text{SortAscending} \leftarrow \{(-\rho \omega) \uparrow (+/\omega) \rho 1\} \]
Boolean Sort

- Sort ascending for Booleans:
  \[ \text{SortAscending} \leftarrow \{ \neg \omega \uparrow (+/\omega) \rho 1 \} \]

- Boolean sort can use Moore’s SIMD +/-Boolean in its first phase of execution
Boolean Sort

- Sort ascending for Booleans:
  \[ \text{SortAscending} \leftarrow \{ (-\rho \omega) \uparrow (+/\omega) \rho 1 \} \]
- Boolean sort can use Moore’s SIMD +/- Boolean in its first phase of execution
- Second phase can be performed in SIMD, e.g., by a single SAC data-parallel with-loop.
- SIMD Boolean upgrade:
  \[ \nu g \leftarrow \{(\neg \omega) / \nu \omega, \omega / \nu \omega\} \]
SIMD Boolean upgrade:
\[ \mu \sigma \left\{ \frac{\neg \omega}{\omega} , \frac{\omega}{\neg \omega} \right\} \]

Not stunningly SIMD, though.
Boolean Matrix Operations

- **Shard**: For byte-oriented algorithms, a possibly empty sub-byte fragment of a matrix row, extending from the start of the row to next byte, or from the last byte in the row to the end of the row.
Boolean Matrix Operations

- **Shard**: For byte-oriented algorithms, a possibly empty sub-byte fragment of a matrix row, extending from the start of the row to next byte, or from the last byte in the row to the end of the row.

- Handling shards is a nuisance; it destroys algorithmic beauty
Boolean Matrix Operations

- **Shard**: For byte-oriented algorithms, a possibly empty sub-byte fragment of a matrix row, extending from the start of the row to next byte, or from the last byte in the row to the end of the row.
- Handling shards is a nuisance; it destroys algorithmic beauty
- Handling shards is also slower than beautiful algorithms
**Boolean Matrix Operations**

- **Shard**: For byte-oriented algorithms, a possibly empty sub-byte fragment of a matrix row, extending from the start of the row to next byte, or from the last byte in the row to the end of the row.
- Handling shards is a nuisance; it destroys algorithmic beauty
- Handling shards is also slower than beautiful algorithms
- Consider:
  
  1 1 0 0, 0 0 0 0, 1 0 1 0, 1 1 1 0
Boolean Matrix Operations

- **Shard**: For byte-oriented algorithms, a possibly empty sub-byte fragment of a matrix row, extending from the start of the row to next byte, or from the last byte in the row to the end of the row.

- Handling shards is a nuisance; it destroys algorithmic beauty

- Handling shards is also slower than beautiful algorithms

- Consider:

  1 1 0 0, 0 0 0 0, 1 0 1 0, 1 1 1 0

- The vector 1 0 1 is a shard, because its elements start at a byte boundary, but end in mid-byte.
Boolean Matrix Operations

- **Shard**: For byte-oriented algorithms, a possibly empty sub-byte fragment of a matrix row, extending from the start of the row to next byte, or from the last byte in the row to the end of the row.

- Handling shards is a nuisance; it destroys algorithmic beauty

- Handling shards is also slower than beautiful algorithms

- Consider:
  
  \[
  \begin{array}{ccccccc}
  1 & 1 & 0 & 0 & 0 & 0 & 0 \\
  1 & 0 & 1 & 0 & 1 & 1 & 1
  \end{array}
  \]

- The vector 1 0 1 is a shard, because its elements start at a byte boundary, but end in mid-byte.

- The vector 0 1 1 1 0 is a shard, because it starts in mid-byte, and ends on a byte boundary.
Boolean Matrix Operations

- **Shard**: For byte-oriented algorithms, a possibly empty sub-byte fragment of a matrix row, extending from the start of the row to next byte, or from the last byte in the row to the end of the row.
- Handling shards is a nuisance; it destroys algorithmic beauty
- Handling shards is also slower than beautiful algorithms
- Consider:
  
  1 1 0 0, 0 0 0 0, 1 0 1 0, 1 1 1 0

- The vector 1 0 1 is a shard, because its elements start at a byte boundary, but end in mid-byte.
- The vector 0 1 1 1 0 is a shard, because it starts in mid-byte, and ends on a byte boundary.
- A similar definition holds for word-oriented algorithms
Acknowledgements

My sincere thanks to James A. Brown, Larry M. Breed, Walter Fil, Jay Foad, Roger K.W. Hui, Roger D. Moore, and Bob Smith, for their constructive suggestions and ideas regarding this paper. The APL programs in this paper were executed on Dyalog APL Version 15.0. Dyalog provides free downloads of their interpreters for educational use; they also offer a free download of their Raspberry Pi edition, at www.dyalog.com. The British APL Association (BAA) provided the author with financial assistance for attending this conference; their thoughtfulness and generosity is greatly appreciated.