APL SIMD Boolean Array Algorithms

Robert Bernecky

Snake Island Research Inc, Canada
bernecky@snakeisland.com

June 10, 2021

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

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

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 $APL\backslash 360$,
- Breed decided to store Booleans densely, eight bits/byte

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

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.

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*\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 *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

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: ∧ , ∨ , ∼ , ⍲ , ⍱ . . .

Scalar Verbs

- Breed optimized many rank-0 (scalar) Boolean verbs *e.g.*
- Boolean verbs: $\wedge, \vee, \sim, \not\wedge, \not\vee \ldots$
- Relational verbs: $<, \leq, =, \geq, >, \neq$

Scalar Verbs

- Breed optimized many rank-0 (scalar) Boolean verbs *e.g.*
- Boolean verbs: $\wedge, \vee, \sim, \barwedge, \bar{\vee} \ldots$
- Relational verbs: $<, \leq, =, \geq, >, \neq$
- 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: $\wedge, \vee, \sim, \not\wedge, \not\vee \ldots$
- 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: ∧ , ∨ , ∼ , ⍲ , ⍱ . . .
- Relational verbs: < , ≤ , = , ≥ , > , ≠
- SIMD application, a word at a time (32 bits on S/360)
- One or more of us optimized scalar extension, *e.g.*
- `1∧B` would produce `B`,

Scalar Verbs

- Breed optimized many rank-0 (scalar) Boolean verbs *e.g.*
- Boolean verbs: ∧ , ∨ , ∼ , ⍲ , ⍱ . . .
- 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∧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×B2` becomes `B1∧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×B2` becomes `B1∧B2`
- In APL, Boolean `B1⌊B2` becomes `B1∧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×B2` becomes `B1∧B2`
- In APL, Boolean `B1⌊B2` becomes `B1∧B2`
- In APL, Boolean `B1*B2` becomes `B1≥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×B2` becomes `B1∧B2`
- In APL, Boolean `B1⌊B2` becomes `B1∧B2`
- In APL, Boolean `B1*B2` becomes `B1≥B2`
- 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 `B1×B2` becomes `B1∧B2`
- In APL, Boolean `B1⌊B2` becomes `B1∧B2`
- In APL, Boolean `B1*B2` becomes `B1≥B2`
- 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
- `snk[sni+⍳k]←src[sri+⍳k]`

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
- `snk[sni+⍳k]←src[sri+⍳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
- `snk[sni+⍳k]←src[sri+⍳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
- `snk[sni+⍳k]←src[sri+⍳k]`
- Does not corrupt out-of-bounds array elements
- Operates in SIMD mode(s) whenever possible
- Supports all type conversions

Structural and Selection Verbs II

- Operation on non-trailing array axes:

Structural and Selection Verbs II

- Operation on non-trailing array axes:
- SIMD copy entire subarrays at once, *e.g.*
  `1⊖2 3 4⍴⍳24`

Structural and Selection Verbs II

- Operation on non-trailing array axes:
- SIMD copy entire subarrays at once, *e.g.*
  1⊖2 3 4⍴⍳24
- rbemove will copy 12 adjacent array elements at once

Structural and Selection Verbs III

- 2 3 4⍴⍳24

```
 0  1  2  3
 4  5  6  7
 8  9 10 11

12 13 14 15
16 17 18 19
20 21 22 23
```

Structural and Selection Verbs III

- ```
  2 3 4⍴⍳24
   0  1  2  3
   4  5  6  7
   8  9 10 11

  12 13 14 15
  16 17 18 19
  20 21 22 23
  ```
- ```
  1⊖2 3 4⍴⍳24
  12 13 14 15
  16 17 18 19
  20 21 22 23

   0  1  2  3
   4  5  6  7
   8  9 10 11
  ```

Reverse and Rotate on Booleans

- Bernecky, 1979: fast algorithms for ⌽ω & α⌽ω

Reverse and Rotate on Booleans

- Bernecky, 1979: fast algorithms for ⌽ω & α⌽ω
- last-axis Boolean ⌽ω did a byte at a time, w/table lookup. . .

```
RevTab[uint8 ω]
```

Reverse and Rotate on Booleans

- Bernecky, 1979: fast algorithms for ⌽ω & α⌽ω
- last-axis Boolean ⌽ω did a byte at a time, w/table lookup...

```
RevTab[uint8 ω]
```

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

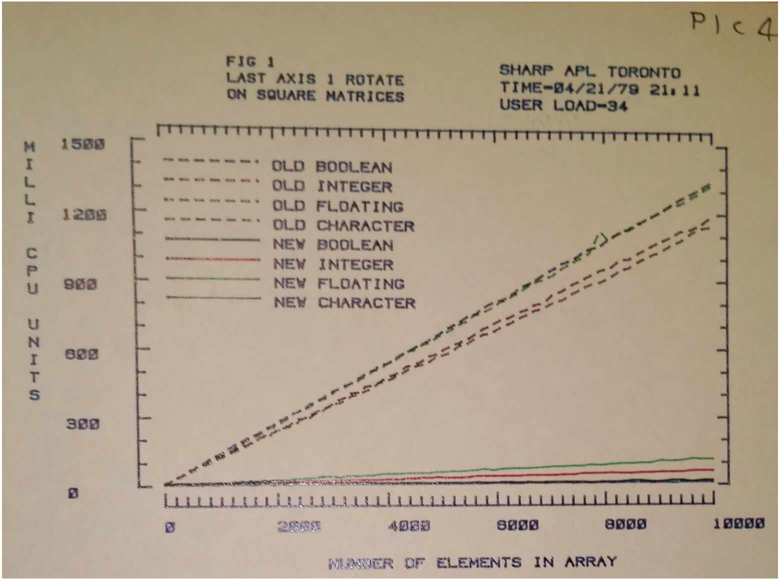
Reverse and Rotate on Booleans

- Bernecky, 1979: fast algorithms for ⌽ω & α⌽ω
- last-axis Boolean ⌽ω did a byte at a time, w/table lookup...

  ```
  RevTab[uint8 ω]
  ```

- 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



Reshape

- Reshape allows element reuse, *e.g.*:

```
      8⍴1 0 0
1 0 0 1 0 0 1 0
```

Reshape

- 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.*:

```
      8⍴1 0 0
1 0 0 1 0 0 1 0
```

- Breed optimized Boolean reshape this way:
- Copy the argument to the result

Reshape

- Reshape allows element reuse, *e.g.*:

```
      8⍴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⍴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

Transpose I

- An unlikely candidate for SIMD, it would seem. . .

Transpose I

- An unlikely candidate for SIMD, it would seem. . .
- α⍉ω with unchanged trailing axes

Transpose I

- An unlikely candidate for SIMD, it would seem. . .
- α⍉ω with unchanged trailing axes
- T←2 2 2 3⍴⍳24

```
   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⍉T
   0  1  2
   3  4  5

  12 13 14
  15 16 17


   6  7  8
   9 10 11

  18 19 20
  21 22 23
  ```
  ```

Boolean Transpose III

- *Hacker's Delight*: fast 8x8 Boolean matrix transpose

Boolean Transpose III

- *Hacker's Delight*: fast 8x8 Boolean matrix transpose
- Kernel: 16 logical & shift operations on 64-bit ravel

Boolean Transpose III

- *Hacker's Delight*: fast 8x8 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 8x8 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 8x8 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.*, 16x16, 32x32

Search Verbs

- Bernecky, 1971: fast *indexof* and *set membership*

Search Verbs

- Bernecky, 1971: fast *indexof* and *set membership*
- All data types except reals with ⎕ct≠0

Search Verbs

- Bernecky, 1971: fast *indexof* and *set membership*
- All data types except reals with ⎕ct≠0
- (α⍳0 1)[ω]

Search Verbs

- Bernecky, 1971: fast *indexof* and *set membership*
- All data types except reals with `⎕ct≠0`
- `(⍺⍳0 1)[⍵]`
- `(⍺⍳⎕av)[⍵]`

Search Verbs

- Bernecky, 1971: fast *indexof* and *set membership*
- All data types except reals with ⎕ct≠0
- `(αι0 1)[ω]`
- `(αι⎕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`
- `(α⍳0 1)[ω]`
- `(α⍳⎕av)[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 `⎕ct≠0`
- `(α⍳0 1)[ω]`
- `(α⍳⎕av)[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 `⎕ct≠0`
- `(⍺⍳0 1)[⍵]`
- `(⍺⍳⎕av)[⍵]`
- 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

Reduce

- Roger D. Moore, 1971: fast $+/\omega$ 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 +/ω Boolean vector
- Initial use was *compress* and *expand* setup:
  +/α 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

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

Reduce

- Roger D. Moore, 1971: fast $+/\omega$ 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 +/ω Boolean vector
- Initial use was *compress* and *expand* setup:
  +/α 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:
  `+/+⌿4 resh PopcountTab[uint8 ω]`

Reduce

- Roger D. Moore, 1971: fast +/ω Boolean vector
- Initial use was *compress* and *expand* setup:
  +/α 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:
  `+/+⌿4 resh PopcountTab[uint8 ω]`
- Algorithm used briefly for ∨/ω and ∧/ω

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

Scan

- 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←nescanall y;s;biw
  ⍝ Not-equal scan
  r←y
  biw←⌈2⍟1⌈⍴y
  :For s :In 2*⍳biw ⍝ Heckman
    r←r≠(-⍴r)↑(-s)↓r
  :EndFor
```

Scan

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

```
r←nescanall y;s;biw
  ⍝ Not-equal scan
  r←y
  biw←⌈2⍟1⌈⍴y
  :For s :In 2*⍳biw ⍝ Heckman
    r←r≠(-⍴r)↑(-s)↓r
  :EndFor
```

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

Scan

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

```
r←nescanall y;s;biw
  ⍝ Not-equal scan
  r←y
  biw←⌈2⍟1⌈⍴y
  :For s :In 2*⍳biw ⍝ Heckman
    r←r≠(-⍴r)↑(-s)↓r
  :EndFor
```

- SIMD, word-at-a-time algorithm for Boolean $\neq\backslash\omega$ and $=\backslash\omega$ along last axis
- Bernecky's simple C Heckman implementation is about 3X faster than Dyalog APL 15.0 (vector only)

Scan

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

```
r←nescanall y;s;biw
  ⍝ Not-equal scan
  r←y
  biw←⌈2⍟1⌈⍴y
  :For s :In 2*⍳biw ⍝ Heckman
    r←r≠(-⍴r)↑(-s)↓r
  :EndFor
```

- SIMD, word-at-a-time algorithm for Boolean ≠\⍵ and =\⍵ 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

STAR Inner Product I

- IPSA, 1973: Boolean array inner products were painfully slow

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

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

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

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;

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;
- redesigned Boolean inner product to use STAR algorithm

Classic Inner Product Algorithm

```
Z←X ipclassic Y;RX;CX;CY;I;J;K
 RX←(⍴X)[0]
 CX←(⍴X)[1]
 CY←(⍴Y)[1]
 Z←(RX,CY)⍴0.5
 :For I :In ⍳RX
  :For J :In ⍳CY
   Z[I;J]←0
   :For K :In ⍳CX
    Z[I;J]←Z[I;J]+X[I;K]×Y[K;J]
   :EndFor
  :EndFor
 :EndFor
```

STAR Inner Product Algorithm

```
Z←X ipstar Y;RX;CX;CY;I;J;Xel
 RX←(⍴X)[0]
 CX←(⍴X)[1]
 CY←(⍴Y)[1]
 Z←(RX,CY)⍴0
 :For I :In ⍳RX
  :For J :In ⍳CX
   Xel←X[I;J]
   Z[I;]←Z[I;]+Xel×Y[J;]
  :EndFor
 :EndFor
```

STAR Inner Product II

- Inner product loops reordered; key benefits, for `α f.g ω`

STAR Inner Product II

- Inner product loops reordered; key benefits, for `α f.g ω`
- Each `α` element, `Xel`, fetched only once

STAR Inner Product II

- Inner product loops reordered; key benefits, for α `f.g` ω
- Each α element, `Xel`, fetched only once
- Type conversion of `Xel` no longer time-critical

STAR Inner Product II

- Inner product loops reordered; key benefits, for `α f.g ω`
- Each `α` element, `Xel`, fetched only once
- Type conversion of `Xel` no longer time-critical
- `Xel` analysis amortized over entire row: `Y[J;]`

STAR Inner Product II

- Inner product loops reordered; key benefits, for `α f.g ω`
- Each `α` 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`
  `tmp←Xel g Y[J;]`

STAR Inner Product II

- Inner product loops reordered; key benefits, for `α f.g ω`
- Each `α` 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`
  `tmp←Xel g Y[J;]`
- Vector-vector `f`-reduce into result row `Z[I;]`
  `Z[I;]←Z[I;] f tmp`

SIMD Boolean STAR Inner Product Basics

- Scalar-vector `Xel g Y[J;]` is word-at-a-time Boolean SIMD

SIMD Boolean STAR Inner Product Basics

- Scalar-vector `Xel g Y[J;]` is word-at-a-time Boolean SIMD
- Vector-vector `Z[I;]←Z[I;]` is word-at-a-time Boolean SIMD

SIMD Boolean STAR Inner Product Basics

- Scalar-vector `Xel g Y[J;]` is word-at-a-time Boolean SIMD
- Vector-vector `Z[I;]←Z[I;]` is word-at-a-time Boolean SIMD
- We are already a lot faster

SIMD Boolean STAR Inner Product Basics

- Scalar-vector `Xel g Y[J;]` is word-at-a-time Boolean SIMD
- Vector-vector `Z[I;]←Z[I;]` is word-at-a-time Boolean SIMD
- We are already a lot faster
- The STAR APL model does `+.×` 90X faster than the Classic model, on 200x200 real matrices

SIMD Boolean STAR Inner Product Basics

- Scalar-vector `Xel g Y[J;]` is word-at-a-time Boolean SIMD
- Vector-vector `Z[I;]←Z[I;]` is word-at-a-time Boolean SIMD
- We are already a lot faster
- The STAR APL model does `+.×` 90X faster than the Classic model, on 200x200 real matrices
- Unfortunately, the APL primitive is still 30X faster than the APL model

Boolean STAR Inner Product Optimizations

- Consider `tmp←Xel ∧ RO` in `α∨.∧ω`

Boolean STAR Inner Product Optimizations

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

Boolean STAR Inner Product Optimizations

- Consider `tmp←Xel ∧ RO` in `α∨.∧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

Boolean STAR Inner Product Optimizations

- Consider `tmp←Xel ∧ RO` in `α∨.∧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
- `f` is `∨`, so its identity element is `0`

Boolean STAR Inner Product Optimizations

- Consider `tmp←Xel ∧ RO` in `α∨.∧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
- `f` is `∨`, so its identity element is `0`
- Hence, if `Xel` is `0`, we can skip the `g`-reduction

Boolean STAR Inner Product Optimizations

- Consider `tmp←Xel ∧ RO` in `α∨.∧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
- `f` is `∨`, so its identity element is `0`
- Hence, if `Xel` is `0`, we can skip the `g`-reduction
- Similarly, if `Xel` is `1`, we can do the `g`-reduction using `RO`

Boolean STAR Inner Product Optimizations

- Consider `tmp←Xel ∧ RO` in `α∨.∧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
- `f` is `∨`, so its identity element is `0`
- Hence, if `Xel` is `0`, we can skip the `g`-reduction
- Similarly, if `Xel` is `1`, we can do the `g`-reduction using `RO`
- This gives us poor man's sparse arrays, which works on other data types, too

Boolean STAR Inner Product Optimizations

- Consider `tmp←Xel ∧ RO` in `α∨.∧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
- `f` is `∨`, so its identity element is `0`
- Hence, if `Xel` is `0`, we can skip the `g`-reduction
- Similarly, if `Xel` is `1`, we can do the `g`-reduction using `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←S+R×S<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 `Tiny` a raise of `R`
  `S←S+R×S<Tiny`
- Knuth calls this capability *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 `Tiny` a raise of `R`
  `S←S+R×S<Tiny`
- Knuth calls this capability *Iverson's convention for characteristic functions*
- See also the verb `mqs`, 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 `Tiny` a raise of `R`
  `S←S+R×S<Tiny`
- Knuth calls this capability *Iverson's convention for characteristic functions*
- See also the verb `mqs`, for finding quoted text
- See also the Bernecky-Scholz PLDI2014 Arrays Workshop paper: *Abstract Expressionism*

Boolean Sort

- Sort ascending for Booleans:
  ```
  SortAscending←{(-⍴⍵)↑(+/⍵)⍴1}
  ```

Boolean Sort

- Sort ascending for Booleans:
  `SortAscending←{(-ρω)↑(+/ω)ρ1}`
- Boolean sort can use Moore's SIMD `+/Boolean` in its first phase of execution

Boolean Sort

- Sort ascending for Booleans:
  `SortAscending←{(-⍴⍵)↑(+/⍵)⍴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.

Boolean Grade

- SIMD Boolean upgrade:
  `ug←{((~⍵)/⍳⍴⍵),⍵/⍳⍴⍵}`

Boolean Grade

- SIMD Boolean upgrade:
  `ug←{((~⍵)/⍳⍴⍵),⍵/⍳⍴⍵}`
- 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:
  `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.

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.