Scalarization of Index Vectors in Compiled APL

Robert Bernecky

Snake Island Research Inc
18 Fifth Street, Ward's Island
Toronto, Canada
tel: +1 416 203 0854
bernecky@snakeisland.com

September 30, 2011

Abstract

High-performance for array languages offers several unique challenges to the compiler writer, including fusion of loops over large arrays, detection and elimination of scalars as arbitrary arrays, and eliminating or minimizing the run-time creation of index vectors.

We introduce one of those challenges in the context of SAC, a functional array languge, and give preliminary results on the performance of a compiler that eliminates index vectors by scalarizing them within the optimization cycle.

The Question

- How much faster is compiled APL than interpreted APL?

The Question

- How much faster is compiled APL than interpreted APL?
- The answer is **NOT** a scalar.

Environment

- Dyalog APL 13.0 *vs.* APEX/SAC

Environment

- Dyalog APL 13.0 *vs.* APEX/SAC
- The current SAC compiler
  - a functional array language
  - data-parallel nested loops: *With-Loop*
  - array-based optimizations
  - functional loops and conditionals as functions

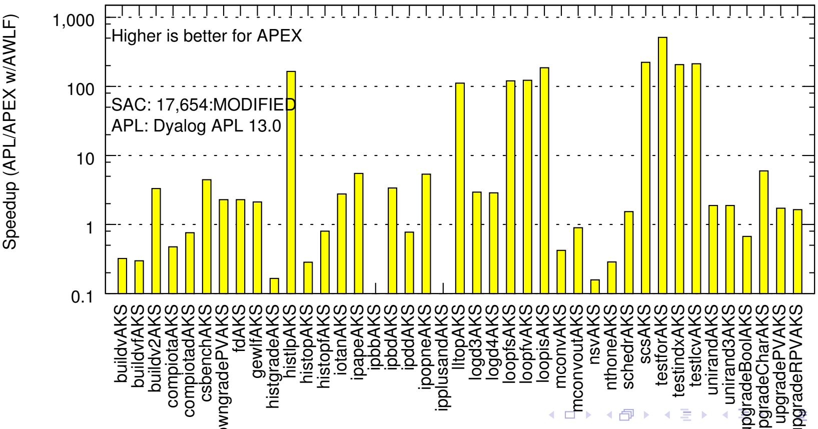
Environment

- Dyalog APL 13.0 *vs.* APEX/SAC
- The current SAC compiler
  - a functional array language
  - data-parallel nested loops: *With-Loop*
  - array-based optimizations
  - functional loops and conditionals as functions
- Goal: Compiled APL performance competitive with hand-coded C

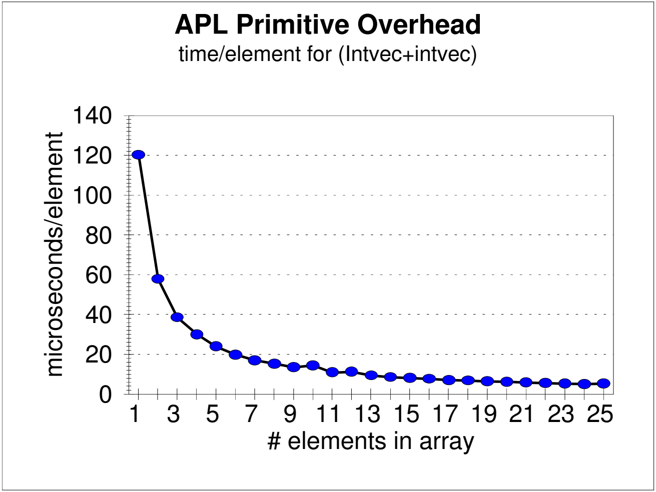
Some Reasons Why APL is Slow

- Fixed per-primitive overheads: Syntax analysis, conf checks, fn dispatch, mem mgmt
- Variable per-primitive overheads
- Index vector materialization

APL vs. APEX CPU Time Performance (2,011–09–30)

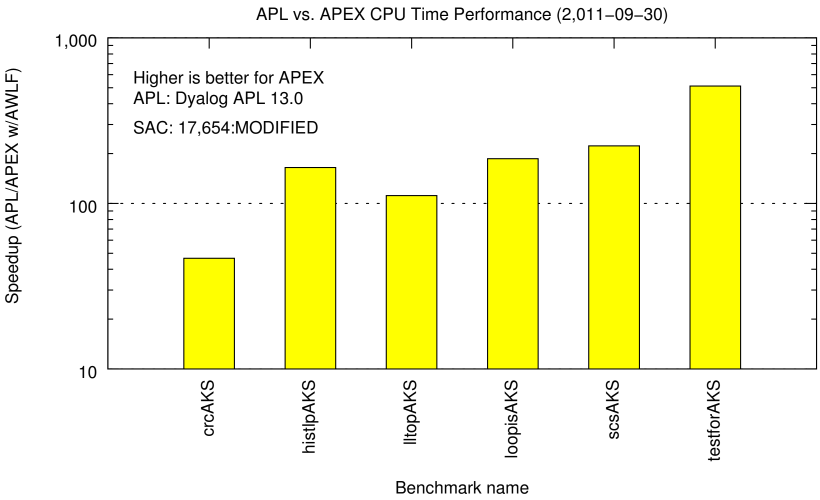


Why APL is Slow: Fixed Per-Primitive Overheads



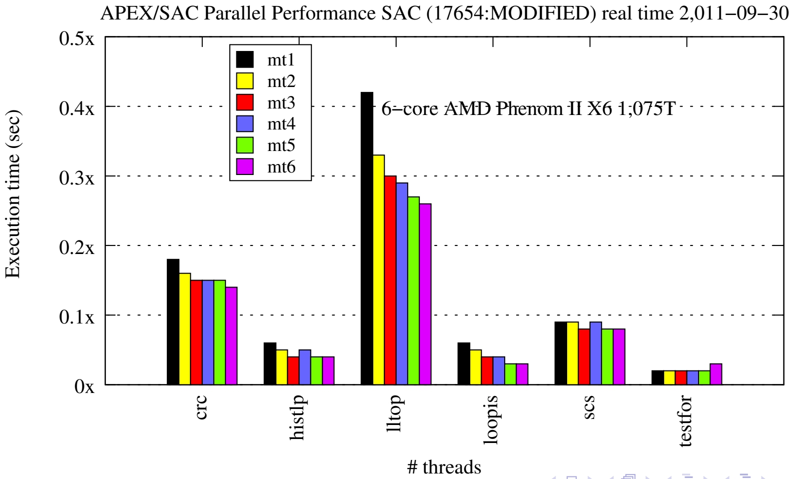
Who suffers? Apps dominated by operations on scalars: CRC, loopy histograms, dynamic programming, RNG

Why APL is Slow: Fixed Per-Primitive Overheads



Why APL is Slow: Fixed Per-Primitive Overheads

- Scalar-dominated apps have good serial speedup...
- but poor parallel speedup



Why is APL Slow? Variable Per-Primitive Overheads

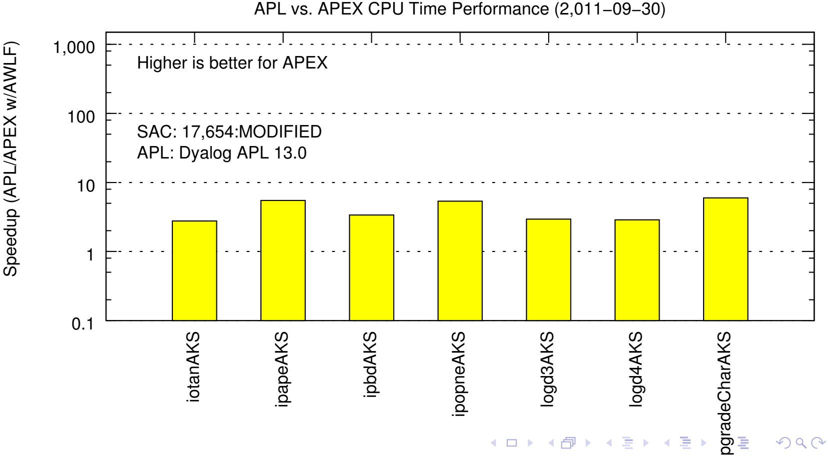
- Naive execution: Limited fn composition, *e.g.* sum(iota(N))

Why is APL Slow? Variable Per-Primitive Overheads

- Naive execution: Limited fn composition, *e.g.* sum(iota(N))
- Array-valued intermediate results: memory madness

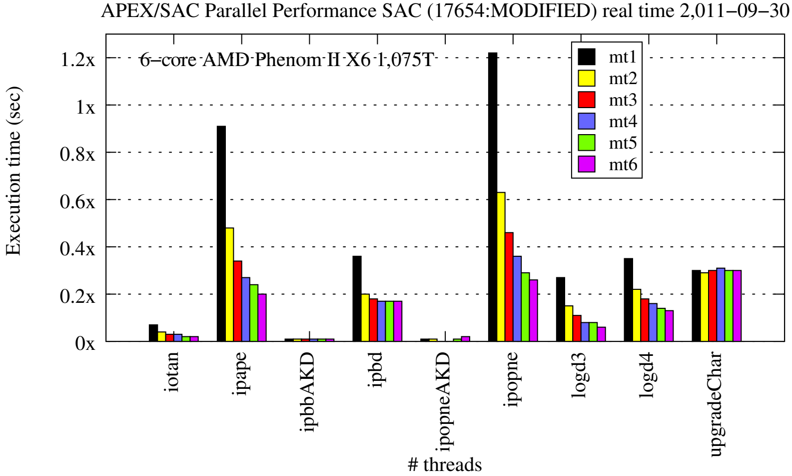
Why is APL Slow? Variable Per-Primitive Overheads

- Naive execution: Limited fn composition, *e.g.* sum(iota(N))
- Array-valued intermediate results: memory madness
- Who suffers? Apps dominated by operations on large arrays: Signal processing, convolution, normal move-out



Why is APL Slow? Variable Per-Primitive Overheads

- Who suffers? Apps dominated by operations on large arrays: Signal processing, convolution, normal move-out



Why is APL Slow? Materialized Index Vectors

- Mike Jenkins' matrix divide model
  `a[;i,pi] = a[;pi,i]`

Why is APL Slow? Materialized Index Vectors

- Mike Jenkins' matrix divide model
  `a[;i,pi] = a[;pi,i]`
- `[i,pi]` and `[pi,i]` are materialized index vectors

Why is APL Slow? Materialized Index Vectors

- Mike Jenkins' matrix divide model
  `a[;i,pi] = a[;pi,i]`
- `[i,pi]` and `[pi,i]` are materialized index vectors
- A few simple changes to scalarize index vectors:

```
tmp = a[;i]
a[;i] = a[;pi]
a[;pi] = tmp
```

Why is APL Slow? Materialized Index Vectors

- Mike Jenkins' matrix divide model
  `a[;i,pi] = a[;pi,i]`
- `[i,pi]` and `[pi,i]` are materialized index vectors
- A few simple changes to scalarize index vectors:
  ```
  tmp = a[;i]
  a[;i] = a[;pi]
  a[;pi] = tmp
  ```
- Matrix divide model now runs twice as fast!

Why Materialized Index Vectors are Expensive

- Materialization of `[i,pi]` and `[pi,i]`:

Why Materialized Index Vectors are Expensive

- Materialization of `[i,pi]` and `[pi,i]`:
- (* for indexing part)
  *Increment reference counts on `i` and `pi`
  Allocate 2-element temp vector
  Initialize temp vector descriptor
  Initialize temp vector elements
  *Perform indexing
  Deallocate 2-element temp vector
  *Decrement reference counts on `i` and `pi`

Why Materialized Index Vectors are Expensive

- Who suffers? Apps using **explicit** array indexing

Why Materialized Index Vectors are Expensive

- Who suffers? Apps using **explicit** array indexing
- *e.g.*, many apps dominated by indexed assign

Why Materialized Index Vectors are Expensive

- Who suffers? Apps using **explicit** array indexing
- *e.g.*, many apps dominated by indexed assign
- Who suffers? Matrix divide, compress, deal, dynamic programming

Why Materialized Index Vectors are Expensive

- Who suffers? Apps using **explicit** array indexing
- *e.g.*, many apps dominated by indexed assign
- Who suffers? Matrix divide, compress, deal, dynamic programming
- Who suffers? Inner products that use the CDC STAR-100 algorithm

Why Materialized Index Vectors are Expensive

- Who suffers? Apps using **explicit** array indexing
- *e.g.*, many apps dominated by indexed assign
- Who suffers? Matrix divide, compress, deal, dynamic programming
- Who suffers? Inner products that use the CDC STAR-100 algorithm
- `800x800 ipplusandAKD` CPU time: 45 minutes!

Eliminating Materialized Index Vectors With IVE

- IFL2006 paper: *Index Vector Elimination* (IVE)
  Bernecky, Grelck, Herhut, Scholz, Trojahner, and Schafarenko

Eliminating Materialized Index Vectors With IVE

- IFL2006 paper: *Index Vector Elimination* (IVE)
  Bernecky, Grelck, Herhut, Scholz, Trojahner, and Schafarenko
- Post-optimization transformation

Eliminating Materialized Index Vectors With IVE

- IFL2006 paper: *Index Vector Elimination* (IVE)
  Bernecky, Grelck, Herhut, Scholz, Trojahner, and Schafarenko
- Post-optimization transformation
- Start with:

```
IV = [ i, j, k]
z = sel( IV, M)
```

Eliminating Materialized Index Vectors With IVE

- IFL2006 paper: *Index Vector Elimination* (IVE)
  Bernecky, Grelck, Herhut, Scholz, Trojahner, and Schafarenko
- Post-optimization transformation
- Start with:

```
IV = [ i, j, k]
z = sel( IV, M)
```

- IVE: Replace: `z = M[ IV]` by:

```
IV = [ i, j, k]
offset = vect2offset( shape(M), IV)
z = idxsel( offset, M)
```

Eliminating Materialized Index Vectors With IVE

- IFL2006 paper: *Index Vector Elimination* (IVE)
  Bernecky, Grelck, Herhut, Scholz, Trojahner, and Schafarenko
- Post-optimization transformation
- Start with:
  ```
  IV = [ i, j, k]
  z = sel( IV, M)
  ```
- IVE: Replace: `z = M[ IV]` by:
  ```
  IV = [ i, j, k]
  offset = vect2offset( shape(M), IV)
  z = idxsel( offset, M)
  ```
- Implication: `IV` is a materialized index vector!

Eliminating Materialized Index Vectors With IVE

- IFL2006 paper: *Index Vector Elimination* (IVE)
  Bernecky, Grelck, Herhut, Scholz, Trojahner, and Schafarenko
- Post-optimization transformation
- Start with:
  ```
  IV = [ i, j, k]
  z = sel( IV, M)
  ```
- IVE: Replace: `z = M[ IV]` by:
  ```
  IV = [ i, j, k]
  offset = vect2offset( shape(M), IV)
  z = idxsel( offset, M)
  ```
- Implication: `IV` is a materialized index vector!
- Implication: `offset` calculation may be liftable (LIR)

Eliminating Materialized Index Vectors With IVE

- IVE: If `IV` can be represented as scalars, eliminate `vect2offset`. Before:

```
IV = [ i, j, k]
offset = vect2offset( shape(M), IV)
z = idxsel( offset, M)
```

Eliminating Materialized Index Vectors With IVE

- IVE: If `IV` can be represented as scalars, eliminate `vect2offset`. Before:

```
IV = [ i, j, k]
offset = vect2offset( shape(M), IV)
z = idxsel( offset, M)
```

- After:

```
IV = [ i, j, k]
offset = idxs2offset( shape(M), i, j, k)
z = idxsel( offset, M)
```

Eliminating Materialized Index Vectors With IVE

- IVE: If `IV` can be represented as scalars, eliminate `vect2offset`. Before:

```
IV = [ i, j, k]
offset = vect2offset( shape(M), IV)
z = idxsel( offset, M)
```

- After:

```
IV = [ i, j, k]
offset = idxs2offset( shape(M), i, j, k)
z = idxsel( offset, M)
```

- IV is now dead code, and can be eliminated!

Eliminating Materialized Index Vectors With IVE

- IVE: If `IV` is **NOT** scalars, scalarize it. Before:

```
offset = vect2offset( shape(M), IV)
z = idxsel( offset, M)
```

Eliminating Materialized Index Vectors With IVE

- IVE: If `IV` is **NOT** scalars, scalarize it. Before:

```
offset = vect2offset( shape(M), IV)
z = idxsel( offset, M)
```

- After:

```
I = IV[0]
J = IV[1]
K = IV[2]
JV = [ I, J, K]
offset = vect2offset( shape(M), JV)
```

Eliminating Materialized Index Vectors With IVE

- IVE: If `IV` is **NOT** scalars, scalarize it. Before:

```
offset = vect2offset( shape(M), IV)
z = idxsel( offset, M)
```

- After:

```
I = IV[0]
J = IV[1]
K = IV[2]
JV = [ I, J, K]
offset = vect2offset( shape(M), JV)
```

- IV is now dead code, and can be eliminated!

Eliminating Materialized Index Vectors With IVE

- IVE: If IV is **NOT** scalars, scalarize it. Before:

```
offset = vect2offset( shape(M), IV)
z = idxsel( offset, M)
```

- After:

```
I = IV[0]
J = IV[1]
K = IV[2]
JV = [ I, J, K]
offset = vect2offset( shape(M), JV)
```

- IV is now dead code, and can be eliminated!
- Earlier substitution by idxs2offset now feasible.

Eliminating Materialized Index Vectors With IVE

- IVE: If `IV` is **NOT** scalars, scalarize it. Before:

```
offset = vect2offset( shape(M), IV)
z = idxsel( offset, M)
```

- After:

```
I = IV[0]
J = IV[1]
K = IV[2]
JV = [ I, J, K]
offset = vect2offset( shape(M), JV)
```

- IV is now dead code, and can be eliminated!
- Earlier substitution by `idxs2offset` now feasible.
- Unfortunately...

Dueling Optimizations: IVE *vs.* LIR

- IVE introduces `JV = [ I, J, K]`

Dueling Optimizations: IVE *vs.* LIR

- IVE introduces `JV = [ I, J, K]`
- Loop-Invariant Removal (LIR) lifts `I, J, K, JV` out of the function

Dueling Optimizations: IVE *vs.* LIR

- IVE introduces `JV = [ I, J, K]`
- Loop-Invariant Removal (LIR) lifts `I, J, K, JV` out of the function
- Constant Folding (CF) replaces `JV = [ I, J, K]` by `JV = IV`

Dueling Optimizations: IVE *vs.* LIR

- IVE introduces `JV = [ I, J, K]`
- Loop-Invariant Removal (LIR) lifts `I, J, K, JV` out of the function
- Constant Folding (CF) replaces `JV = [ I, J, K]` by `JV = IV`
- Common-subexpression elimination (CSE) replaces `JV` by `IV`

Dueling Optimizations: IVE *vs.* LIR

- IVE introduces `JV = [ I, J, K]`
- Loop-Invariant Removal (LIR) lifts `I, J, K, JV` out of the function
- Constant Folding (CF) replaces `JV = [ I, J, K]` by `JV = IV`
- Common-subexpression elimination (CSE) replaces `JV` by `IV`
- This is where we came in!

Dueling Optimizations: IVE *vs.* LIR

- IVE introduces `JV = [ I, J, K]`
- Loop-Invariant Removal (LIR) lifts `I, J, K, JV` out of the function
- Constant Folding (CF) replaces `JV = [ I, J, K]` by `JV = IV`
- Common-subexpression elimination (CSE) replaces `JV` by `IV`
- This is where we came in!
- So, what can we do?

Dueling Optimizations: IVE *vs.* LIR

- IVE introduces `JV = [ I, J, K]`
- Loop-Invariant Removal (LIR) lifts `I, J, K, JV` out of the function
- Constant Folding (CF) replaces `JV = [ I, J, K]` by `JV = IV`
- Common-subexpression elimination (CSE) replaces `JV` by `IV`
- This is where we came in!
- So, what can we do?
- A kludged LIR to deal with this was deemed tasteless

Biting the Bullet

- Another approach: move IVE into the optimization cycle

Biting the Bullet

- Another approach: move IVE into the optimization cycle
- Having IVE in opt cycle enables other optimizations!

Biting the Bullet

- Another approach: move IVE into the optimization cycle
- Having IVE in opt cycle enables other optimizations!
- But, it also breaks many optimizations

Biting the Bullet

- Another approach: move IVE into the optimization cycle
- Having IVE in opt cycle enables other optimizations!
- But, it also breaks many optimizations
- To prevent dueling, we must scalarize all index vector ops!

Biting the Bullet

- Another approach: move IVE into the optimization cycle
- Having IVE in opt cycle enables other optimizations!
- But, it also breaks many optimizations
- To prevent dueling, we must scalarize all index vector ops!
- *e.g.*, unroll `IV+1`, guard and extrema functions..

Biting the Bullet

- Another approach: move IVE into the optimization cycle
- Having IVE in opt cycle enables other optimizations!
- But, it also breaks many optimizations
- To prevent dueling, we must scalarize all index vector ops!
- *e.g.*, unroll `IV+1`, guard and extrema functions..
- *e.g.*, extend existing optimizations, such as CF

Biting the Bullet

- Another approach: move IVE into the optimization cycle
- Having IVE in opt cycle enables other optimizations!
- But, it also breaks many optimizations
- To prevent dueling, we must scalarize all index vector ops!
- *e.g.*, unroll `IV+1`, guard and extrema functions..
- *e.g.*, extend existing optimizations, such as CF
- The Good News: I had already scalarized many index vectors for Algebraic With-Loop Folding!

Current Status

- Moving IVE into optimization cycle worked, sort of:

Current Status

- Moving IVE into optimization cycle worked, sort of:
- The Good News: `ipplusandAKD` CPU time: 8 seconds, instead of 45 minutes

Current Status

- Moving IVE into optimization cycle worked, sort of:
- The Good News: `ipplusandAKD` CPU time: 8 seconds, instead of 45 minutes
- The Bad News: New primitives in AST broke some optimizations!

Current Status

- Moving IVE into optimization cycle worked, sort of:
- The Good News: `ipplusandAKD` CPU time: 8 seconds, instead of 45 minutes
- The Bad News: New primitives in AST broke some optimizations!
- More Bad News: I am still fixing them!

Current Status

- Moving IVE into optimization cycle worked, sort of:
- The Good News: `ipplusandAKD` CPU time: 8 seconds, instead of 45 minutes
- The Bad News: New primitives in AST broke some optimizations!
- More Bad News: I am still fixing them!
- The Good News: Performance is improving daily

Current Status

- Moving IVE into optimization cycle worked, sort of:
- The Good News: `ipplusandAKD` CPU time: 8 seconds, instead of 45 minutes
- The Bad News: New primitives in AST broke some optimizations!
- More Bad News: I am still fixing them!
- The Good News: Performance is improving daily
- More Good News: Many opportunities exist for further optimization

Current Status

- Moving IVE into optimization cycle worked, sort of:
- The Good News: `ipplusandAKD` CPU time: 8 seconds, instead of 45 minutes
- The Bad News: New primitives in AST broke some optimizations!
- More Bad News: I am still fixing them!
- The Good News: Performance is improving daily
- More Good News: Many opportunities exist for further optimization
- Even More Good News: More parallelism is coming

How Fast Is Compiled APL?

- Array sizes affect performance

How Fast Is Compiled APL?

- Array sizes affect performance
- Iteration counts affect performance

How Fast Is Compiled APL?

- Array sizes affect performance
- Iteration counts affect performance
- Indexed assigns affect performance

How Fast Is Compiled APL?

- Array sizes affect performance
- Iteration counts affect performance
- Indexed assigns affect performance
- Characterize your application, then we can provide an answer.