The Three Beaars

Basically, Every Array Allocation Reduces Speed

Robert Bernecky

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

October 10, 2012

Abstract

Functional array language compiler and interpreter designers try to reduce the number of arrays created during application execution, because the negative impact of arrays on performance is so dramatic.
Just as The Three Bears had different requirements for their own satisfaction, so do differing array shapes have different requirements for their elimination. The problem itself is a bear: scalar operations are the baby bear, typified here by dynamic programming and the Floyd-Warshall algorithm; operations on small arrays, such as numerically intense computations on complex arrays, is the mama bear; operations on large arrays, typified by acoustic signal processing, is the papa bear.
We compare interpreted to compiled APL performance for several applications with different array shapes, and give an overview of the various optimizations that enable those speedups, in both serial and parallel contexts.

The APEX-SaC tool chain

- APEX: APL-to-SaC compiler (R. Bernecky)

The APEX-SaC tool chain

- APEX: APL-to-SaC compiler (R. Bernecky)
- SaC: SaC-to-C compiler (S.B. Scholz)

The APEX-SaC tool chain

- APEX: APL-to-SaC compiler (R. Bernecky)
- SaC: SaC-to-C compiler (S.B. Scholz)
- APEX & SaC preserve arrays throughout compilation

The APEX-SaC tool chain

- APEX: APL-to-SaC compiler (R. Bernecky)
- SaC: SaC-to-C compiler (S.B. Scholz)
- APEX & SaC preserve arrays throughout compilation
- SaC is a purely functional compiler

The APEX-SaC tool chain

- APEX: APL-to-SaC compiler (R. Bernecky)
- SaC: SaC-to-C compiler (S.B. Scholz)
- APEX & SaC preserve arrays throughout compilation
- SaC is a purely functional compiler
- SaC represents control structures as functions

The APEX-SaC tool chain

- APEX: APL-to-SaC compiler (R. Bernecky)
- SaC: SaC-to-C compiler (S.B. Scholz)
- APEX & SaC preserve arrays throughout compilation
- SaC is a purely functional compiler
- SaC represents control structures as functions
- These characteristics are a mixed blessing...

Why are arrays expensive?

- Consider the cost to perform `Z←X+Y`:

Why are arrays expensive?

- Consider the cost to perform `Z←X+Y`:
- (Interpreter) Parse code to find expression: 200ops

Why are arrays expensive?

- Consider the cost to perform `Z←X+Y`:
- (Interpreter) Parse code to find expression: 200ops
- Increment reference counts on `X,Y`: 50ops

Why are arrays expensive?

- Consider the cost to perform `Z←X+Y`:
- (Interpreter) Parse code to find expression: 200ops
- Increment reference counts on `X,Y`: 50ops
- Conformance checks (type, rank, shape) for addition: 200ops

Why are arrays expensive?

- Consider the cost to perform `Z←X+Y`:
- (Interpreter) Parse code to find expression: 200ops
- Increment reference counts on `X,Y`: 50ops
- Conformance checks (type, rank, shape) for addition: 200ops
- Allocate temp for result from heap: 200ops

Why are arrays expensive?

- Consider the cost to perform `Z←X+Y`:
- (Interpreter) Parse code to find expression: 200ops
- Increment reference counts on `X,Y`: 50ops
- Conformance checks (type, rank, shape) for addition: 200ops
- Allocate temp for result from heap: 200ops
- Initialize temp: 100ops

Why are arrays expensive?

- Consider the cost to perform `Z←X+Y`:
- (Interpreter) Parse code to find expression: 200ops
- Increment reference counts on `X,Y`: 50ops
- Conformance checks (type, rank, shape) for addition: 200ops
- Allocate temp for result from heap: 200ops
- Initialize temp: 100ops
- Perform actual additions: 200ops

Why are arrays expensive?

- Consider the cost to perform `Z←X+Y`:
- (Interpreter) Parse code to find expression: 200ops
- Increment reference counts on `X,Y`: 50ops
- Conformance checks (type, rank, shape) for addition: 200ops
- Allocate temp for result from heap: 200ops
- Initialize temp: 100ops
- Perform actual additions: 200ops
- Decrement reference counts on `X,Y`: 50ops

Why are arrays expensive?

- Consider the cost to perform `Z←X+Y`:
- (Interpreter) Parse code to find expression: 200ops
- Increment reference counts on `X,Y`: 50ops
- Conformance checks (type, rank, shape) for addition: 200ops
- Allocate temp for result from heap: 200ops
- Initialize temp: 100ops
- Perform actual additions: 200ops
- Decrement reference counts on `X,Y`: 50ops
- Deallocate old `Z`, if any: 100ops

Why are arrays expensive?

- Consider the cost to perform `Z←X+Y`:
- (Interpreter) Parse code to find expression: 200ops
- Increment reference counts on `X,Y`: 50ops
- Conformance checks (type, rank, shape) for addition: 200ops
- Allocate temp for result from heap: 200ops
- Initialize temp: 100ops
- Perform actual additions: 200ops
- Decrement reference counts on `X,Y`: 50ops
- Deallocate old `Z`, if any: 100ops
- Assign `Z←temp`: 50ops

Why are arrays expensive?

- Consider the cost to perform `Z←X+Y`:
- (Interpreter) Parse code to find expression: 200ops
- Increment reference counts on `X,Y`: 50ops
- Conformance checks (type, rank, shape) for addition: 200ops
- Allocate temp for result from heap: 200ops
- Initialize temp: 100ops
- Perform actual additions: 200ops
- Decrement reference counts on `X,Y`: 50ops
- Deallocate old `Z`, if any: 100ops
- Assign `Z←temp`: 50ops
- TOTAL: 1150ops

Why are arrays expensive?

- Consider the cost to perform `Z←X+Y`:
- (Interpreter) Parse code to find expression: 200ops
- Increment reference counts on `X,Y`: 50ops
- Conformance checks (type, rank, shape) for addition: 200ops
- Allocate temp for result from heap: 200ops
- Initialize temp: 100ops
- Perform actual additions: 200ops
- Decrement reference counts on `X,Y`: 50ops
- Deallocate old `Z`, if any: 100ops
- Assign `Z←temp`: 50ops
- TOTAL: 1150ops
- *vs.* compiled scalar code: 10ops

Baby Bear - Eliminating Scalar Arrays in a Compiler

- Use classical static data flow analysis to find scalars

Baby Bear - Eliminating Scalar Arrays in a Compiler

- Use classical static data flow analysis to find scalars
- Traditional optimization methods: CSE, VP, CP, etc.

Baby Bear - Eliminating Scalar Arrays in a Compiler

- Use classical static data flow analysis to find scalars
- Traditional optimization methods: CSE, VP, CP, etc.
- Allocate scalars on stack, instead of heap

Baby Bear - Eliminating Scalar Arrays in a Compiler

- Use classical static data flow analysis to find scalars
- Traditional optimization methods: CSE, VP, CP, etc.
- Allocate scalars on stack, instead of heap
- Generate scalar-specific code

Baby Bear - Eliminating Scalar Arrays in a Compiler

- Use classical static data flow analysis to find scalars
- Traditional optimization methods: CSE, VP, CP, etc.
- Allocate scalars on stack, instead of heap
- Generate scalar-specific code
- This is challenging to do in an interpreter

Baby Bear - Eliminating Scalar Arrays in a Compiler

- Use classical static data flow analysis to find scalars
- Traditional optimization methods: CSE, VP, CP, etc.
- Allocate scalars on stack, instead of heap
- Generate scalar-specific code
- This is challenging to do in an interpreter
- Experimental platform: AMD 1075T 6-core CPU, 3.2GHz

Baby Bear - Eliminating Scalar Arrays in a Compiler

- Use classical static data flow analysis to find scalars
- Traditional optimization methods: CSE, VP, CP, etc.
- Allocate scalars on stack, instead of heap
- Generate scalar-specific code
- This is challenging to do in an interpreter
- Experimental platform: AMD 1075T 6-core CPU, 3.2GHz
- (cheap ASUS M4A88T-M desktop machine)

Baby Bear Problem: Floyd-Warshall Algorithm

```
z←floyd D;i;j;k
siz←⍳(⍴D)[0]
:For k :In siz
  :For i :In siz
    :For j :In siz
      D[i;j]←D[i;j]⌊D[i;k]+D[k;j]
    :EndFor
  :EndFor
:EndFor
```

- Problem size: 3000x3000 graph

Baby Bear Problem: Floyd-Warshall Algorithm

```
z←floyd D;i;j;k
siz←⍳(⍴D)[0]
:For k :In siz
  :For i :In siz
    :For j :In siz
      D[i;j]←D[i;j]⌊D[i;k]+D[k;j]
    :EndFor
  :EndFor
:EndFor
```

- Problem size: 3000x3000 graph
- Dyalog APL, J interpreters: one week-ish; APEX/SAC: 103sec

Baby Bear Problem: Floyd-Warshall Algorithm

```
z←floyd D;i;j;k
siz←⍳(⍴D)[0]
:For k :In siz
  :For i :In siz
    :For j :In siz
      D[i;j]←D[i;j]⌊D[i;k]+D[k;j]
    :EndFor
  :EndFor
:EndFor
```

- Problem size: 3000x3000 graph
- Dyalog APL, J interpreters: one week-ish; APEX/SAC: 103sec
- Dynamic programming (string shuffle): >1000X speedup

Baby Bear Problem: Floyd-Warshall Algorithm

```
z←floyd D;i;j;k
siz←⍳(⍴D)[0]
:For k :In siz
  :For i :In siz
    :For j :In siz
      D[i;j]←D[i;j]⌊D[i;k]+D[k;j]
    :EndFor
  :EndFor
:EndFor
```

- Problem size: 3000x3000 graph
- Dyalog APL, J interpreters: one week-ish; APEX/SAC: 103sec
- Dynamic programming (string shuffle): >1000X speedup
- Lesson: Interpreters dislike scalar-dominated algorithms

Baby Bear Problem: Floyd-Warshall Algorithm

```
z←floyd D;i;j;k
siz←⍳(⍴D)[0]
:For k :In siz
  :For i :In siz
    :For j :In siz
      D[i;j]←D[i;j]⌊D[i;k]+D[k;j]
    :EndFor
  :EndFor
:EndFor
```

- Problem size: 3000x3000 graph
- Dyalog APL, J interpreters: one week-ish; APEX/SAC: 103sec
- Dynamic programming (string shuffle): >1000X speedup
- Lesson: Interpreters dislike scalar-dominated algorithms
- Lesson: Compilers are not fussy; Baby bear problem solved!

Baby Bear Problem: Floyd-Warshall Algorithm

```
z←floyd D;i;j;k
siz←⍳(⍴D)[0]
:For k :In siz
  :For i :In siz
    :For j :In siz
      D[i;j]←D[i;j]⌊D[i;k]+D[k;j]
    :EndFor
  :EndFor
:EndFor
```

- Problem size: 3000x3000 graph
- Dyalog APL, J interpreters: one week-ish; APEX/SAC: 103sec
- Dynamic programming (string shuffle): >1000X speedup
- Lesson: Interpreters dislike scalar-dominated algorithms
- Lesson: Compilers are not fussy; Baby bear problem solved!
- But, no parallelism: Adding threads just makes it slower!

Baby Bear Problem: Floyd-Warshall Algorithm

```
z←floyd D;i;j;k
siz←⍳(⍴D)[0]
:For k :In siz
  :For i :In siz
    :For j :In siz
      D[i;j]←D[i;j]⌊D[i;k]+D[k;j]
    :EndFor
  :EndFor
:EndFor
```

- Problem size: 3000x3000 graph
- Dyalog APL, J interpreters: one week-ish; APEX/SAC: 103sec
- Dynamic programming (string shuffle): >1000X speedup
- Lesson: Interpreters dislike scalar-dominated algorithms
- Lesson: Compilers are not fussy; Baby bear problem solved!
- But, no parallelism: Adding threads just makes it slower!
- What about array-based solutions? It's papa bear time!

Array-based Floyd-Warshall Algorithm

- j64-602, from J Essays (CDC STAR APL algorithm variant)

```
floyd=: 3 :
('for_j. i.#y';'do.
        y=. y <. j ({"1 +/ {) y';'
      end.';'y')
```

Array-based Floyd-Warshall Algorithm

- j64-602, from J Essays (CDC STAR APL algorithm variant)

```
floyd=: 3 :
('for_j. i.#y';'do.
        y=. y <. j ({"1 +/ {) y';'
      end.';'y')
```

- SAC: Scholz & Bernecky (Classic matmul variant)

```
inline int[.,.] floydSbs1(int[.,.] D ) {
  DT = transpose(D);
  res = with
          (. <= [i,j] <= .) :
              min( D[i,j], minval( D[i] + DT[j]));
: modarray(D);
  return( res);
}
```

Array-based Floyd-Warshall Algorithm

- j64-602, from J Essays (CDC STAR APL algorithm variant)

```
floyd=: 3 :
('for_j. i.#y';'do.
        y=. y <. j ({"1 +/ {) y';'
      end.';'y')
```

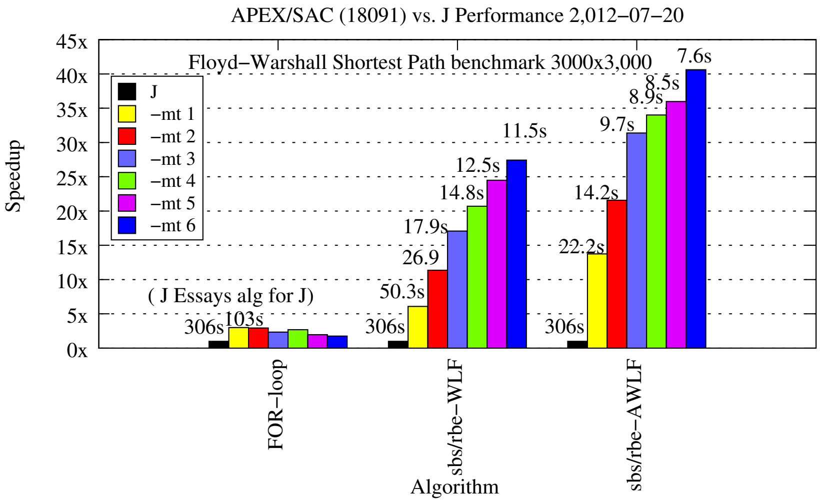
- SAC: Scholz & Bernecky (Classic matmul variant)

```
inline int[.,.] floydSbs1(int[.,.] D ) {
  DT = transpose(D);
  res = with
          (. <= [i,j] <= .) :
                min( D[i,j], minval( D[i] + DT[j]));
: modarray(D);
  return( res);
}
```

- A "with-loop" is a nested data-parallel FORALL loop

Array-based Floyd-Warshall Algorithm Speedup

Lesson: Array-based code and optimizers are good for you



Loop Fusion

- Z = V1 + (V2 * V3)

```
for( i=0; i<n; i++) {
    tmp[i] = V2[i] * V3[i] ; }
for( j=0; j<n; j++) {
    Z[j] = V1[j] + tmp[j]; }
```

Loop Fusion

- `Z = V1 + (V2 * V3)`

```
for( i=0; i<n; i++) {
    tmp[i] = V2[i] * V3[i]; }
for( j=0; j<n; j++) {
    Z[j] = V1[j] + tmp[j]; }
```

- Loop fusion transforms this into:

```
for( j=0; j<n; j++) {
    Z[j] = V1[j] + (V2[j] * V3[j]); }
```

Loop Fusion

- `Z = V1 + (V2 * V3)`

  ```
  for( i=0; i<n; i++) {
      tmp[i] = V2[i] * V3[i]; }
  for( j=0; j<n; j++) {
      Z[j] = V1[j] + tmp[j]; }
  ```

- Loop fusion transforms this into:

  ```
  for( j=0; j<n; j++) {
      Z[j] = V1[j] +  (V2[j] * V3[j]); }
  ```

- Benefit: Array-valued `tmp` removed (DCR)

Loop Fusion

- `Z = V1 + (V2 * V3)`
  ```
  for( i=0; i<n; i++) {
     tmp[i] = V2[i] * V3[i]; }
  for( j=0; j<n; j++) {
     Z[j] = V1[j] + tmp[j]; }
  ```
- Loop fusion transforms this into:
  ```
  for( j=0; j<n; j++) {
     Z[j] = V1[j] +  (V2[j] * V3[j]);  }
  ```
- Benefit: Array-valued `tmp` removed (DCR)
- Benefit: Reduced memory subsystem traffic

Loop Fusion

- Z = V1 + (V2 * V3)

  ```
  for( i=0; i<n; i++) {
     tmp[i] = V2[i] * V3[i]; }
  for( j=0; j<n; j++) {
     Z[j] = V1[j] + tmp[j]; }
  ```
- Loop fusion transforms this into:

  ```
  for( j=0; j<n; j++) {
     Z[j] = V1[j] +  (V2[j] * V3[j]); }
  ```
- Benefit: Array-valued `tmp` removed (DCR)
- Benefit: Reduced memory subsystem traffic
- Benefit: Reduced loop overhead

Loop Fusion

- `Z = V1 + (V2 * V3)`
  ```
  for( i=0; i<n; i++) {
     tmp[i] = V2[i] * V3[i]; }
  for( j=0; j<n; j++) {
     Z[j] = V1[j] + tmp[j]; }
  ```
- Loop fusion transforms this into:
  ```
  for( j=0; j<n; j++) {
     Z[j] = V1[j] +  (V2[j] * V3[j]);  }
  ```
- Benefit: Array-valued `tmp` removed (DCR)
- Benefit: Reduced memory subsystem traffic
- Benefit: Reduced loop overhead
- Benefit: Improved parallelism, in some compilers

With-Loop Folding (WLF) and Algebraic With-Loop Folding (AWLF)

- WLF (S.B. Scholz) - a generalization of loop fusion

With-Loop Folding (WLF) and Algebraic With-Loop Folding (AWLF)

- WLF (S.B. Scholz) - a generalization of loop fusion
- Handles Arrays of Known Shape (AKS) only

With-Loop Folding (WLF) and Algebraic With-Loop Folding (AWLF)

- WLF (S.B. Scholz) - a generalization of loop fusion
- Handles Arrays of Known Shape (AKS) only
- AWLF (R. Bernecky)

With-Loop Folding (WLF) and Algebraic With-Loop Folding (AWLF)

- WLF (S.B. Scholz) - a generalization of loop fusion
- Handles Arrays of Known Shape (AKS) only
- AWLF (R. Bernecky)
- Handles AKS arrays & Arrays of Known Dimension (AKD)

With-Loop Folding (WLF) and Algebraic With-Loop Folding (AWLF)

- WLF (S.B. Scholz) - a generalization of loop fusion
- Handles Arrays of Known Shape (AKS) only
- AWLF (R. Bernecky)
- Handles AKS arrays & Arrays of Known Dimension (AKD)
- Acoustic signal processing (delta modulation):

```
logd←{¯50⌈50⌊50×(DIFF 0,ω)÷0.01+ω}
DIFF←{¯1↓ω−¯1⌽ω}
```

WLF/AWLF example: Acoustic Signal Processing

- `logd` on 200E6-element double-precision vector

sac2c options	Serial elapsed time sec	Parallel (-mt 6) elapsed time sec	Speedup
APL	7.8s	n/a	n/a
-nowlf -O3	10.7s	5.5s	1.9X
-doawlf -O3	3.2s	0.7s	4.5X
Speedup	3.3X	7.8X	15X

WLF/AWLF example: Acoustic Signal Processing

- `logd` on 200E6-element double-precision vector
- Sixteen with-loops are folded into two WLs!

sac2c options	Serial elapsed time sec	Parallel (-mt 6) elapsed time sec	Speedup
APL	7.8s	n/a	n/a
-nowlf -O3	10.7s	5.5s	1.9X
-doawlf -O3	3.2s	0.7s	4.5X
Speedup	3.3X	7.8X	15X

WLF/AWLF example: Acoustic Signal Processing

- `logd` on 200E6-element double-precision vector
- Sixteen with-loops are folded into two WLs!
- WLF/AWLF increase available parallelism

sac2c options	Serial elapsed time sec	Parallel (-mt 6) elapsed time sec	Speedup
APL	7.8s	n/a	n/a
-nowlf -O3	10.7s	5.5s	1.9X
-doawlf -O3	3.2s	0.7s	4.5X
Speedup	3.3X	7.8X	15X

With-Loop Scalarization (WLS)

- With-Loop Scalarization: (C. Grelck, S.B. Scholz, K. Trojahner)

With-Loop Scalarization (WLS)

- With-Loop Scalarization: (C. Grelck, S.B. Scholz, K. Trojahner)
- Operates on nested-WLs in which inner loop creates non-scalar cells

With-Loop Scalarization (WLS)

- With-Loop Scalarization: (C. Grelck, S.B. Scholz, K. Trojahner)
- Operates on nested-WLs in which inner loop creates non-scalar cells
- WLS to merge loop-nest pairs, forming a single WL

```
A = with ([0] <= iv < [4]) {
B = with ([0] <= jv < [4])
     genarray( [4], iv[0] + 2 * jv[0]);
}
     genarray( [4], B);
```

With-Loop Scalarization (WLS)

- With-Loop Scalarization: (C. Grelck, S.B. Scholz, K. Trojahner)
- Operates on nested-WLs in which inner loop creates non-scalar cells
- WLS to merge loop-nest pairs, forming a single WL

```
A = with ([0] <= iv < [4]) {
B = with ([0] <= jv < [4])
     genarray( [4], iv[0] + 2 * jv[0]);
}
     genarray( [4], B);
```

- WLS transforms this into:

```
A = with ([0,0] <= iv < [4,4])
     genarray( [4,4], iv[0] + 2 * iv[1]);
```

With-Loop Scalarization (WLS)

- With-Loop Scalarization: (C. Grelck, S.B. Scholz, K. Trojahner)
- Operates on nested-WLs in which inner loop creates non-scalar cells
- WLS to merge loop-nest pairs, forming a single WL

```
A = with ([0] <= iv < [4]) {
B = with ([0] <= jv < [4])
     genarray( [4], iv[0] + 2 * jv[0]);
}
     genarray( [4], B);
```

- WLS transforms this into:

```
A = with ([0,0] <= iv < [4,4])
     genarray( [4,4], iv[0] + 2 * iv[1]);
```

- Mandatory for good performance: array-valued temps removed

WLF/AWLF/WLS example: Poisson 2-D Relaxation Kernel

From Sven-Bodo Scholz: *With-Loop-Folding in Sac*

A good argument for Ken Iverson's mask verb!

```
z←relax A;m;n. . .
 m←(⍴A)[0]
 n←(⍴A)[1]
 B←((1⊖A)+(¯1⊖A)+(1⌽A)+(¯1⌽A))÷4
 upperA←(1,n)↑A
 lowerA←((m−1),0)↓A
 leftA←1 0↓((m−1),1)↑A
 rightA←((m−2),1)↑(1,n−1)↓A
 innerB←((m−2),n−2)↑1 1↓B
 middle←leftA,innerB,rightA
 z←upperA⍪middle⍪lowerA
```

Poisson 2-D Relaxation: Multi-thread, Various Grid Sizes

- AWLF, aided by WLS, folds `relax` function into 1 loop!

Poisson 2-D Relaxation: Multi-thread, Various Grid Sizes

- AWLF, aided by WLS, folds `relax` function into 1 loop!
- 20K iterations, 250x250 grid: Dyalog APL: CPU time = 47.4s

Poisson 2-D Relaxation: Multi-thread, Various Grid Sizes

- AWLF, aided by WLS, folds `relax` function into 1 loop!
- 20K iterations, 250x250 grid: Dyalog APL: CPU time = 47.4s
- APEX/SAC 18091, single-thread: CPU time = 3.65s

Poisson 2-D Relaxation: Multi-thread, Various Grid Sizes

- AWLF, aided by WLS, folds `relax` function into 1 loop!
- 20K iterations, 250x250 grid: Dyalog APL: CPU time = 47.4s
- APEX/SAC 18091, single-thread: CPU time = 3.65s
- APEX/SAC 18091: multi-threaded (no source code changes!)

Poisson 2-D Relaxation: Multi-thread, Various Grid Sizes

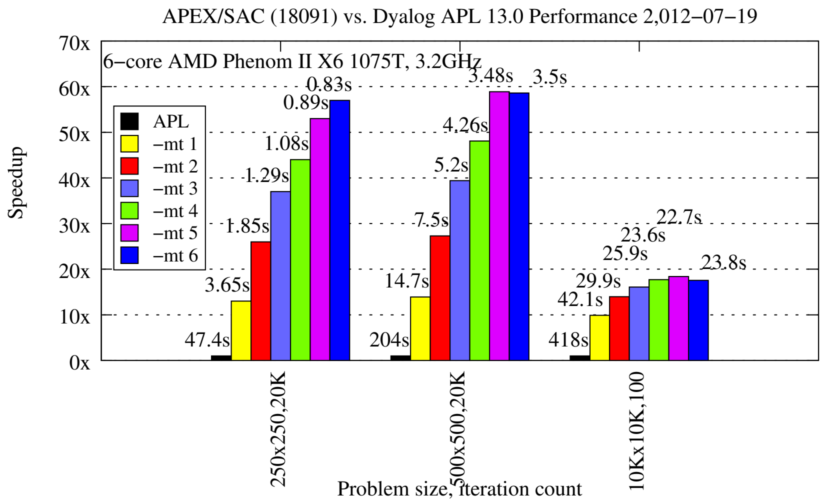


Figure: APEX vs. APL CPU time performance

Poisson 2-D Relaxation: Memory footprint

- Why poor speedup on 10Kx10K test?

Poisson 2-D Relaxation: Memory footprint

- Why poor speedup on 10Kx10K test?
- Dyalog APL 13.0, 10Kx10K grid: 8.5GB footprint

Poisson 2-D Relaxation: Memory footprint

- Why poor speedup on 10Kx10K test?
- Dyalog APL 13.0, 10Kx10K grid: 8.5GB footprint
- APEX/SAC 18091: 10Kx10K grid: 3.4GB footprint

Poisson 2-D Relaxation: Memory footprint

- Why poor speedup on 10Kx10K test?
- Dyalog APL 13.0, 10Kx10K grid: 8.5GB footprint
- APEX/SAC 18091: 10Kx10K grid: 3.4GB footprint
- Memory subsystem bandwidth: 4464MB/s

Poisson 2-D Relaxation: Memory footprint

- Why poor speedup on 10Kx10K test?
- Dyalog APL 13.0, 10Kx10K grid: 8.5GB footprint
- APEX/SAC 18091: 10Kx10K grid: 3.4GB footprint
- Memory subsystem bandwidth: 4464MB/s
- Grid is 800MB $\rightarrow$ 5 writes of grid to/from memory/s

Poisson 2-D Relaxation: Memory footprint

- Why poor speedup on 10Kx10K test?
- Dyalog APL 13.0, 10Kx10K grid: 8.5GB footprint
- APEX/SAC 18091: 10Kx10K grid: 3.4GB footprint
- Memory subsystem bandwidth: 4464MB/s
- Grid is 800MB → 5 writes of grid to/from memory/s
- Therefore, speedup is eventually memory-limited on cheapo system

Poisson 2-D Relaxation: Memory footprint

- Why poor speedup on 10Kx10K test?
- Dyalog APL 13.0, 10Kx10K grid: 8.5GB footprint
- APEX/SAC 18091: 10Kx10K grid: 3.4GB footprint
- Memory subsystem bandwidth: 4464MB/s
- Grid is 800MB $\rightarrow$ 5 writes of grid to/from memory/s
- Therefore, speedup is eventually memory-limited on cheapo system
- Scholz sees linear speedup on 48-core system

Poisson 2-D Relaxation: Memory footprint

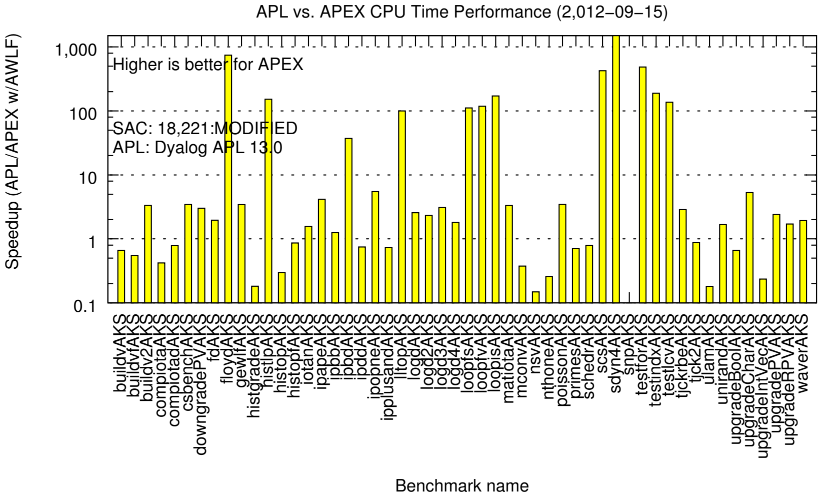
- Why poor speedup on 10Kx10K test?
- Dyalog APL 13.0, 10Kx10K grid: 8.5GB footprint
- APEX/SAC 18091: 10Kx10K grid: 3.4GB footprint
- Memory subsystem bandwidth: 4464MB/s
- Grid is 800MB → 5 writes of grid to/from memory/s
- Therefore, speedup is eventually memory-limited on cheapo system
- Scholz sees linear speedup on 48-core system
- Lesson: High memory bandwidth is good for you.

Poisson 2-D Relaxation: Memory footprint

- Why poor speedup on 10Kx10K test?
- Dyalog APL 13.0, 10Kx10K grid: 8.5GB footprint
- APEX/SAC 18091: 10Kx10K grid: 3.4GB footprint
- Memory subsystem bandwidth: 4464MB/s
- Grid is 800MB $\rightarrow$ 5 writes of grid to/from memory/s
- Therefore, speedup is eventually memory-limited on cheapo system
- Scholz sees linear speedup on 48-core system
- Lesson: High memory bandwidth is good for you.
- Lesson: Array optimizations are VERY good for you.

Mama Bear Motivation

Why is interpreted APL faster than compiled code for some tests?



Mama Bear Motivation

Some reasons for poor performance of compiled SAC code:

- Index vector generation for indexed assign

Mama Bear Motivation

Some reasons for poor performance of compiled SAC code:

- Index vector generation for indexed assign
- Shape vector generation for variable result shapes

Mama Bear Motivation

Some reasons for poor performance of compiled SAC code:

- Index vector generation for indexed assign
- Shape vector generation for variable result shapes
- Generation of small arrays, *e.g.*, complex scalars

Mama Bear Motivation

Some reasons for poor performance of compiled SAC code:

- Index vector generation for indexed assign
- Shape vector generation for variable result shapes
- Generation of small arrays, *e.g.*, complex scalars
- No SaC FOR-loop analog to with-loop

Mama Bear - Small Array Scalarization

- Replace small arrays by their scalarized form

Mama Bear - Small Array Scalarization

- Replace small arrays by their scalarized form
- Optimization: Primitive Function Unrolling (Classic)

Mama Bear - Small Array Scalarization

- Replace small arrays by their scalarized form
- Optimization: Primitive Function Unrolling (Classic)
- Optimization: Index Vector Elimination (IVE) (sacdev)
  2–16X speedup observed

Mama Bear - Small Array Scalarization

- Replace small arrays by their scalarized form
- Optimization: Primitive Function Unrolling (Classic)
- Optimization: Index Vector Elimination (IVE) (sacdev)
  2–16X speedup observed
- Optimizations: LS, LACSI, LACSO (S.B. Scholz, R. Bernecky)

Mama Bear - Small Array Scalarization

- Mandelbrot set computation performance

Mama Bear - Small Array Scalarization

- Mandelbrot set computation performance
- `mandelbrot`: Uses complex numbers

```
int calc( complex z, int maxdepth) {...
while(real(z)*real(z)+imag(z)*imag(z)<=4.0)...
```

Mama Bear - Small Array Scalarization

- Mandelbrot set computation performance
- `mandelbrot`: Uses complex numbers

```
int calc( complex z, int maxdepth) {...
while(real(z)*real(z)+imag(z)*imag(z)<=4.0)...
```

- Complex scalars, under the covers:

```
complex z ↔ double(2) z
   real(z) ↔ z[0]
   imag(z) ↔ z[1]
```

Mama Bear - Small Array Scalarization

- Mandelbrot set computation performance
- `mandelbrot`: Uses complex numbers

```
int calc( complex z, int maxdepth) {...
while(real(z)*real(z)+imag(z)*imag(z)<=4.0)...
```

- Complex scalars, under the covers:

```
complex z ↔ double(2) z
   real(z) ↔ z[0]
   imag(z) ↔ z[1]
```

- `mandelbrot_opt`: Hand-scalarized - pair of scalars

```
int calc( double zr, double zi, int maxdepth) {...
while( zr * zr + zi * zi <= 4.0)...
```

Mama Bear - Small Array Scalarization

- Execution times, with LS,LACSI,LACSO opts enabled/disabled

Test	Opts	-mt 1	-mt 2	-mt 3	-mt 4	-mt 5	-mt 6
mandelbrot	off	1508.9s	956.0s	828.7s	676.8s	655.7s	635.2s
mandelbrot_opt	off	71.8s	48.4s	35.2s	28.1s	23.0s	19.8s
mandelbrot	on	69.9s	46.1s	34.6s	28.1s	23.0s	21.9s
mandelbrot_opt	on	70.7s	46.7s	34.7s	28.2s	22.9s	19.6s

Mama Bear - Small Array Scalarization

- Execution times, with LS,LACSI,LACSO opts enabled/disabled

Test	Opts	-mt 1	-mt 2	-mt 3	-mt 4	-mt 5	-mt 6
`mandelbrot`	off	1508.9s	956.0s	828.7s	676.8s	655.7s	635.2s
`mandelbrot_opt`	off	71.8s	48.4s	35.2s	28.1s	23.0s	19.8s
`mandelbrot`	on	69.9s	46.1s	34.6s	28.1s	23.0s	21.9s
`mandelbrot_opt`	on	70.7s	46.7s	34.7s	28.2s	22.9s	19.6s

- Lesson: No more suffering for being elegant

Mama Bear - Small Array Scalarization

- Execution times, with LS,LACSI,LACSO opts enabled/disabled

Test	Opts	-mt 1	-mt 2	-mt 3	-mt 4	-mt 5	-mt 6
`mandelbrot`	off	1508.9s	956.0s	828.7s	676.8s	655.7s	635.2s
`mandelbrot_opt`	off	71.8s	48.4s	35.2s	28.1s	23.0s	19.8s
`mandelbrot`	on	69.9s	46.1s	34.6s	28.1s	23.0s	21.9s
`mandelbrot_opt`	on	70.7s	46.7s	34.7s	28.2s	22.9s	19.6s

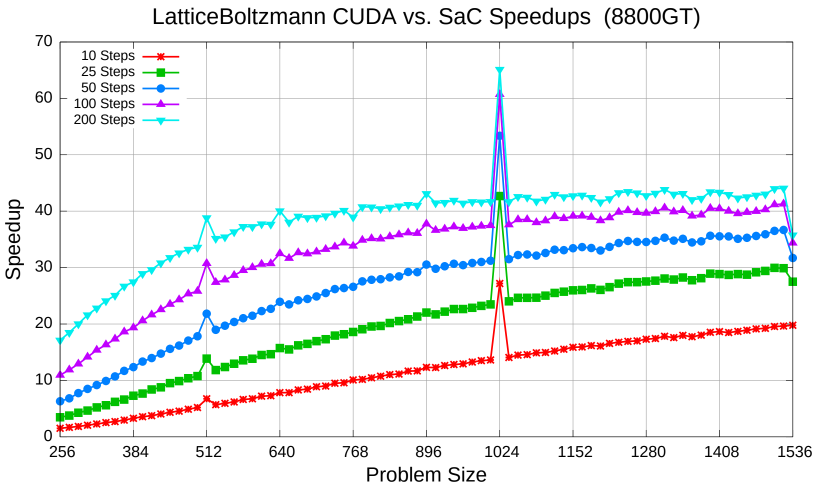
- Lesson: No more suffering for being elegant
- Well, less suffering for being elegant...

GPU (CUDA) Support Without Suffering

- SaC generates CUDA code automatically: `-target cuda`

GPU (CUDA) Support Without Suffering

- SaC generates CUDA code automatically: `-target cuda`
- Physics experiment



Goldilocks - Nested Arrays in APEX/SAC

- Nested arrays are alive and living in SAC! (R. Douma)

Goldilocks - Nested Arrays in APEX/SAC

- Nested arrays are alive and living in SAC! (R. Douma)
- APL convolution kernel using *EACH*:

```
convn←{fi←α ⋄ (⍳⍴ω)con¨⊂ω}
con←{fi+.×(⍴fi)↑α↓ω}
```

Goldilocks - Nested Arrays in APEX/SAC

- Nested arrays are alive and living in SAC! (R. Douma)
- APL convolution kernel using *EACH*:

```
convn←{fi←α ⋄ (ιρω)con¨⊂ω}
con←{fi+.×(ρfi)↑α↓ω}
```

- SAC convolution kernel using *EACH*:

```
nested double[.] NDV;
nested double NDS;
pt=trace++(filter*0.0); NB. No overtake in SAC
z=convn(iota(shape(tr)[0]),fi,enclose_NDV(pt));
convn: z=with{ ( . <= iv <= .) :
           con(dc[iv],fi,disclose_NDV(tr));
           } : genarray(shape(dc),0.0);
con: matmul(fi,take(shape(fi),drop([dc],tr)))
```

Goldilocks - Nested Arrays in APEX/SAC

- Nested arrays are alive and living in SAC! (R. Douma)
- APL convolution kernel using *EACH*:

```
convn←{fi←α ⋄ (⍳⍴ω)con¨⊂ω}
con←{fi+.×(⍴fi)↑α↓ω}
```

- SAC convolution kernel using *EACH*:

```
nested double[.] NDV;
nested double NDS;
pt=trace++(filter*0.0); NB. No overtake in SAC
z=convn(iota(shape(tr)[0]),fi,enclose_NDV(pt));
convn: z=with{ ( . <= iv <= .) :
            con(dc[iv],fi,disclose_NDV(tr));
            } : genarray(shape(dc),0.0);
con: matmul(fi,take(shape(fi),drop([dc],tr)))
```

- Performance is so-so: Optimistic optimizations required

Summary and Future Work

- Status:

Bear	Array size	Optimizers	Serial speedup	Parallel speedup
Baby	scalars	mature	up to 1300X	none
Mama	small	developing	up to 20X	enables other opts
Papa	large	nearly done	up to 10X	2X-50X

Summary and Future Work

- Status:

Bear	Array size	Optimizers	Serial speedup	Parallel speedup
Baby	scalars	mature	up to 1300X	none
Mama	small	developing	up to 20X	enables other opts
Papa	large	nearly done	up to 10X	2X-50X

- All optimizations are critical for getting excellent performance

Summary and Future Work

- Status:

Bear	Array size	Optimizers	Serial speedup	Parallel speedup
Baby	scalars	mature	up to 1300X	none
Mama	small	developing	up to 20X	enables other opts
Papa	large	nearly done	up to 10X	2X-50X

- All optimizations are critical for getting excellent performance
- Array-based algorithms will win, and scale well

Summary and Future Work

- Status:

Bear	Array size	Optimizers	Serial speedup	Parallel speedup
Baby	scalars	mature	up to 1300X	none
Mama	small	developing	up to 20X	enables other opts
Papa	large	nearly done	up to 10X	2X-50X

- All optimizations are critical for getting excellent performance
- Array-based algorithms will win, and scale well
- Nested arrays: APEX, SAC both require work

Summary and Future Work

- Status:

Bear	Array size	Optimizers	Serial speedup	Parallel speedup
Baby	scalars	mature	up to 1300X	none
Mama	small	developing	up to 20X	enables other opts
Papa	large	nearly done	up to 10X	2X-50X

- All optimizations are critical for getting excellent performance
- Array-based algorithms will win, and scale well
- Nested arrays: APEX, SAC both require work
- Small arrays: Needs scalarized index-vector-to-offset primitive

Summary and Future Work

- Status:

Bear	Array size	Optimizers	Serial speedup	Parallel speedup
Baby	scalars	mature	up to 1300X	none
Mama	small	developing	up to 20X	enables other opts
Papa	large	nearly done	up to 10X	2X-50X

- All optimizations are critical for getting excellent performance
- Array-based algorithms will win, and scale well
- Nested arrays: APEX, SAC both require work
- Small arrays: Needs scalarized index-vector-to-offset primitive
- Small arrays: Perhaps (likely!), additional work will be needed

Summary and Future Work

- Status:

Bear	Array size	Optimizers	Serial speedup	Parallel speedup
Baby	scalars	mature	up to 1300X	none
Mama	small	developing	up to 20X	enables other opts
Papa	large	nearly done	up to 10X	2X-50X

- All optimizations are critical for getting excellent performance
- Array-based algorithms will win, and scale well
- Nested arrays: APEX, SAC both require work
- Small arrays: Needs scalarized index-vector-to-offset primitive
- Small arrays: Perhaps (likely!), additional work will be needed
- And, they lived more or less happily ever after! Thank you!