Functionality is Finally Free!

Robert Bernecky

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

January 19, 2010

Abstract

Functional array languages, such as APL, J, SAC, and SISAL, are computer languages with their roots in linear algebra. Being abstract, functional, and array-based, they are excellent tools of thought. As well as being amenable to formal analysis and algebraic simplification, they exhibit parallelism at levels including primitive verbs, expressions, defined verbs, compositions of verbs, and verbs derived from adverbs and conjunctions.

Until recently, these benefits have been sullied by poor performance, compared to C and Fortran, their scalar-oriented, imperative cousins, with array-based code often executing several orders of magnitude slower.

We present recent research results in functional array language optimization that show a closing of that performance gap, with array language code matching or beating scalar code on single core benchmarks, as well as offering significantly better performance on multi-core systems and GPU-based systems, all achieved without application code alterations.

Talk Outline

- Characteristics of Functional Array Languages (FAL)
- Code size: FAL *vs.* imperative languages (F77, C)
- Compiled APL performance
- Where does the performance come from? With-loop folding
- Serial performance results
- Parallel performance results
- Summary

Functional Array Languages (FAL) Characteristics

- Typified by APL, J, SAC, SISAL, Qube
- Tools of thought, designed for the human brain
- No concept of computer ``memory"
- Derived from linear algebra
- Arbitrary-rank arrays as first-class objects
- **NOT** vectors of vectors!
- Primitive operations on arrays; call by value

```
      1 2 3 × 10 15 20
 10 30 60
```

- Multiple return values from verbs (functions)

```
+/  % #
```

- Massive available parallelism
- Safe - no array bounds violations
- Usually terse, *e.g.*: arithmetic mean in J:
- Traditionally orders of magnitude slower than C/Fortran

Code Size

LOGD2: Acoustic signal shaping, delta modulation, first-difference

- Fortran 77 diff function

```
      subroutine diff(wv,siz)
      double precision wv(1),t,t2
      integer siz,i
      do 6 i= siz,2,-1
    6 wv(i) = wv(i) - wv(i-1)
      return & end
```

- Fortran 95 diff function

```
      subroutine diff(wv,siz)
      wv = wv - eoshift(wv,-1)
      return & end
```

- Dyalog APL diff function

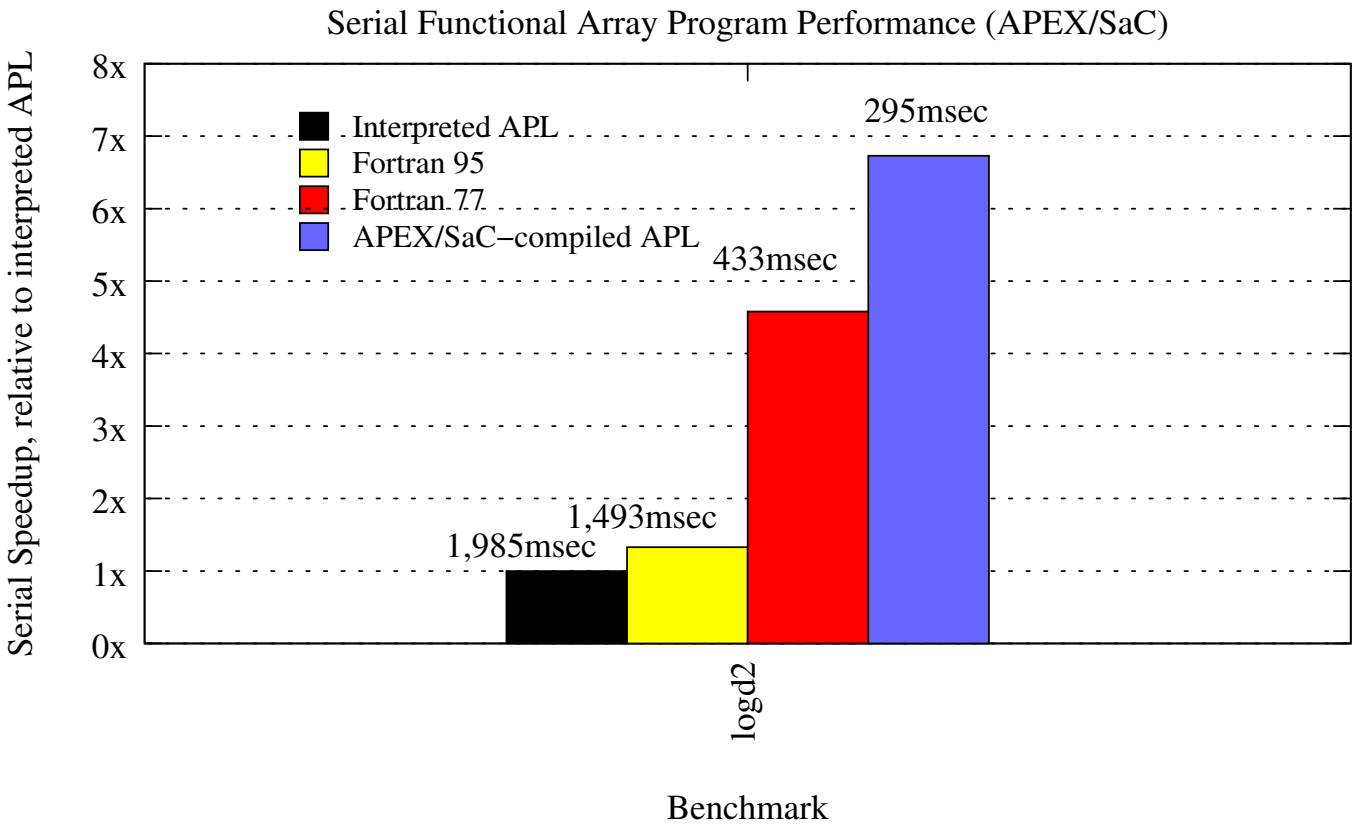
```
diff←{⍵-¯1⌽⍵}
```

SAC - Single Assignment C

- Invented by Sven-Bodo Scholz (U Hertfordshire, UK)
- Functional C subset - no pointers or globals
- Arrays as first-class objects.
- Functions for dim, shape, array generate, with-loop
- With-loop: data-parallel array generator
- With-loop folding: data-flow loop folding, *a la* loop fusion
- Had APL-like primitives, *e.g.* transpose, reshape, reverse
- Now primitive capabilities provided by stdlib, written in SAC
- You can write your own stdlib and get same performance!
- Parallel back ends
- LOGD2 example: `S-rotate(-1,S)`
- More info: www.sac-home.org

APEX/SAC Functional Array Language Serial Performance

LOGD2: Acoustic signal shaping, delta modulation, first-difference



Compiled APEX Performance

- APL source code for `logd2`:

```
main:  +/logderiv 0.5+⍳⍵
logderiv:  ¯50⌈50⌊50×(diff2 ⍵)÷⍵+0.01
diff2:  ⍵-0,¯1↓⍵
```

- AKS: Arrays of Known Shape (F77)
- AKD: Arrays of Known Dimension (APL)
- SaC AKS With-Loop Folding (WLF) - Sven-Bodo Scholz
- The above code folds into ONE data-parallel with-loop
- Symbiotic Expressions & Algebraic WLF handle AKD arrays
- How do they work? Let's walk through an example.

Sum of first 50 integers in SAC

- SAC: `z = sum(iota(50))`
- No with-loop folding

```
ints = with { ([0] <= [i] < [50]) : i;
            } : genarray( [50], 0);

z = with { ([0] <= iv < [50]) {
             curz = _accu_( iv);
             el = ints[iv];
             newz = curz + el;
             } : newz ;
         } : fold( +, 0);
```

- Array-valued intermediate result `ints`!
- Lots of memory subsystem traffic.

Sum of first 50 integers in SAC

- SAC: `z = sum(iota(50))`
- Using with-loop folding

```
z = with { ([0] <= iv=[i] < [50]) {
        curz = _accu_( iv);
        newz = curz + i;
        } : newz ;
    } : fold( +, 0);
```

- No array-valued intermediate results!
- Major reduction in memory subsystem traffic.
- This is as good as it can get.

Sum of first 50 integers in APL

- APL: `z←+/⍳50`
- No with-loop folding:

```
ints = with { ([0] <= [i] < [50]) : i;
            : genarray( [50], 0);

lim = [50] - 1;
z = with { ([0] <= iv < [50]) {
        curz = _accu_( iv);
        iv2 = lim - iv;
        el = ints[iv2];
        newz = curz + el;
        : newz ;
    } : fold( +, 0);
```

- NB. APL reduction semantics are right-to-left!!
- Array-valued intermediate result `ints`!
- Lots of memory subsystem traffic.

Sum of first 50 integers in APL

- APL: `z←+/⍳50`
- Using with-loop folding

```
lim = [50] - 1;
z = with { ([0] <= iv=[i] < [50]) {
        curz = _accu_( iv);
        i2 = lim - i;
        newz = curz + i2;
        } : newz ;
    } : fold( +, 0);
```

- Good performance, but SAC is faster.
- Can we improve the APL code?

Sum of first 50 integers in APL

- Suppose I do not care about the reduction order?
- Reverse the vector before summation!
- APL: `z←+/⌽⍳50`
- Without with-loop folding: 3 with-loops
- Using with-loop folding

```
z = with { ([0] <= iv=[i] < [50]) {
        curz = _accu_( iv);
        newz = curz + i;
        } : newz ;
    } : fold( +, 0);
```

- APL now runs as fast as SAC.
- Lesson: Functions may be free in compiled FAL!
- This would run at same speed:
  `z←+/⌽⌽⌽⌽⌽⌽⌽⌽⌽⌽⌽⌽⌽⌽⌽⌽⌽⌽⌽⍳50`
- There is NO funny business in the compiler to do this!

Symbiotic Expressions

- Consider: shape analysis for `diff2: ω-0,¯1↓ω`
- Subtract: `shape(ω) = shape(0,¯1↓ω)`
- Consider `shape(0,¯1↓ω)` is ...
- `(1 + (shape(ω) - 1)`, giving...
- `shape(ω)`, which is what we need.
- We're done!
- Can the compiler do this algebra for us?
- Now it can!
- Symbiotic expressions to the rescue.

Symbiotic Expressions

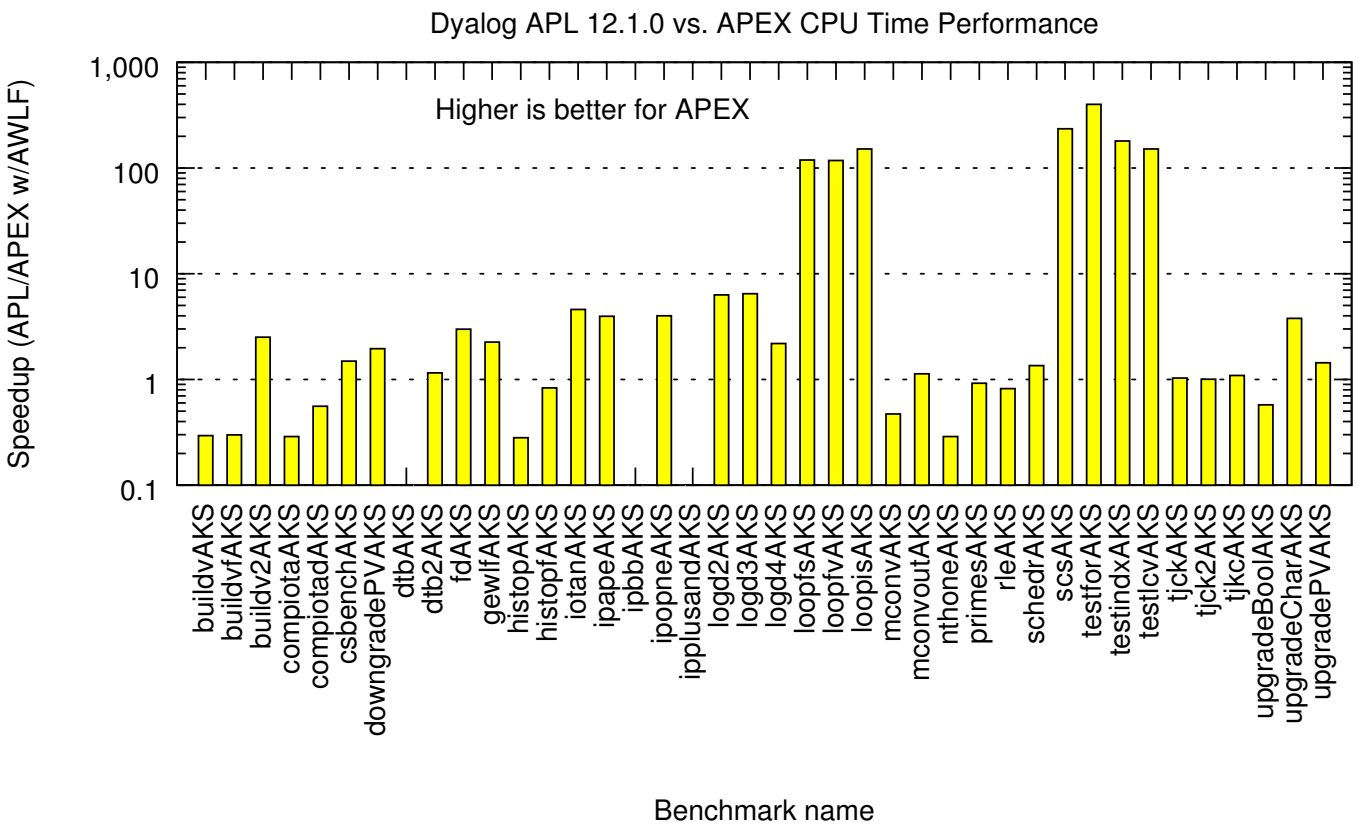
- Generate symbolic expressions as IL or source code
- Attach to Abstract Syntax Tree (AST) of code being compiled
- Let optimizers(CF, AL, AS, VP, CSE. . .) simplify them
- Extract simplified expressions from AST; optimize code
- Delete expressions from AST
- Basically, let the compiler use itself to solve its problems
- IFL2009 paper, *Symbiotic Expressions*, in press.

NAS Multi-Grid Benchmark

This kernel with-loop folds into a single data-parallel loop.

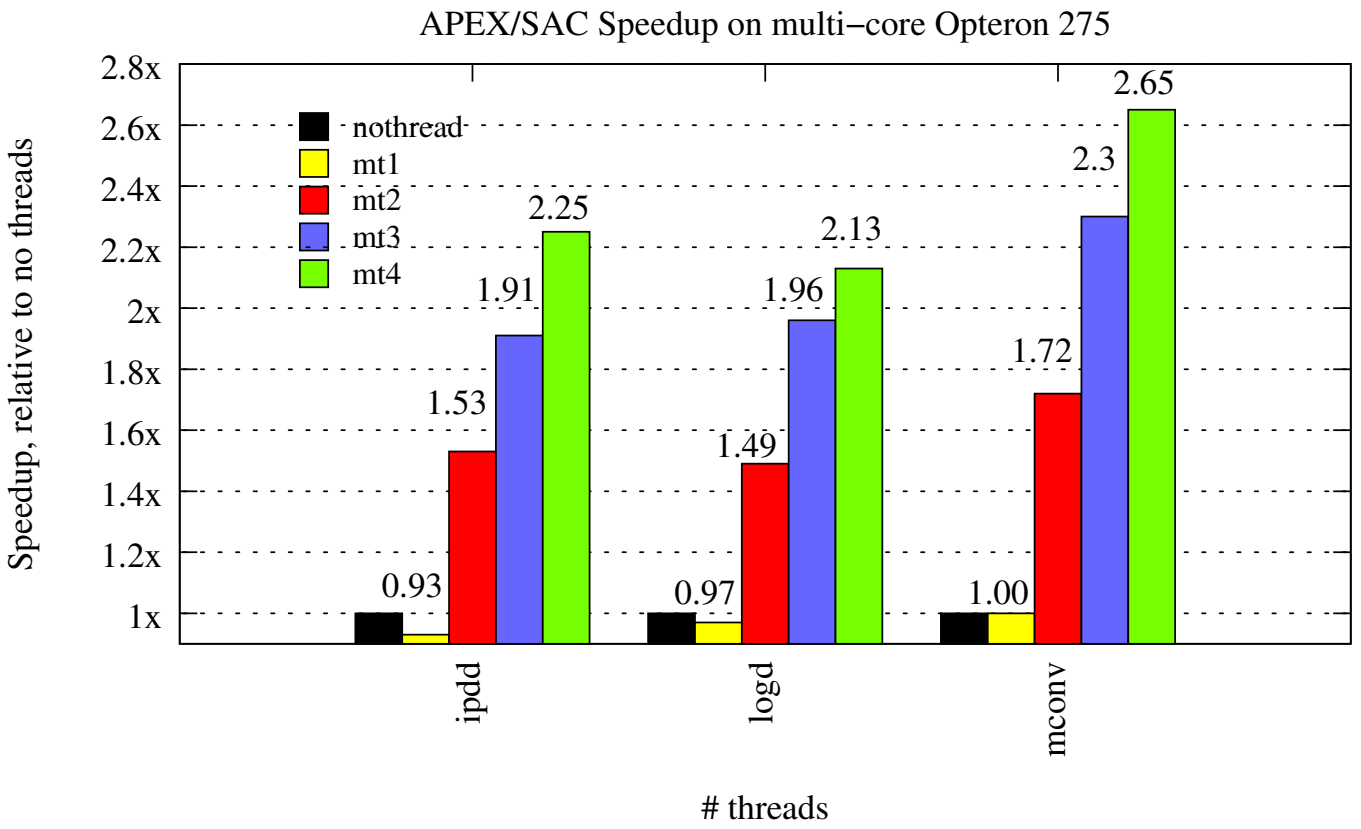
```
double[.,.] relax( double[.,.] A)
{
  m = shape(A)[0];
  n = shape(A)[1];
  leftA = drop( [1,0], take( [m-1,1], A));
  innerB = take( [m-2,n-2], drop( [1,1], A));
  middle = cat( innerB, leftA);
  result = middle ++ A;
  return(result);
}
```

APEX Performance vs. APL



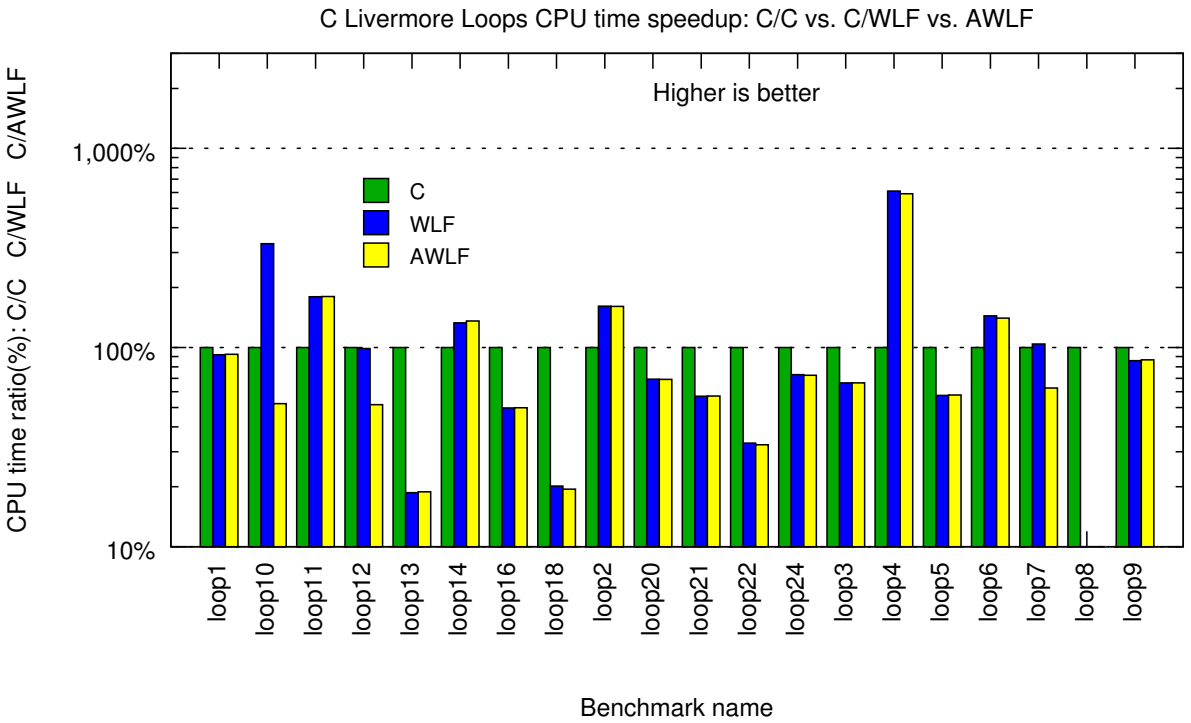
- Highly iterative code (dynamic programming `scs`, `sdyn4`) performs very well.
- FOR-loops (`buildv`, `histgrade`) & with-loops within conditionals need help.

Multi-thread APEX Performance on Opteron



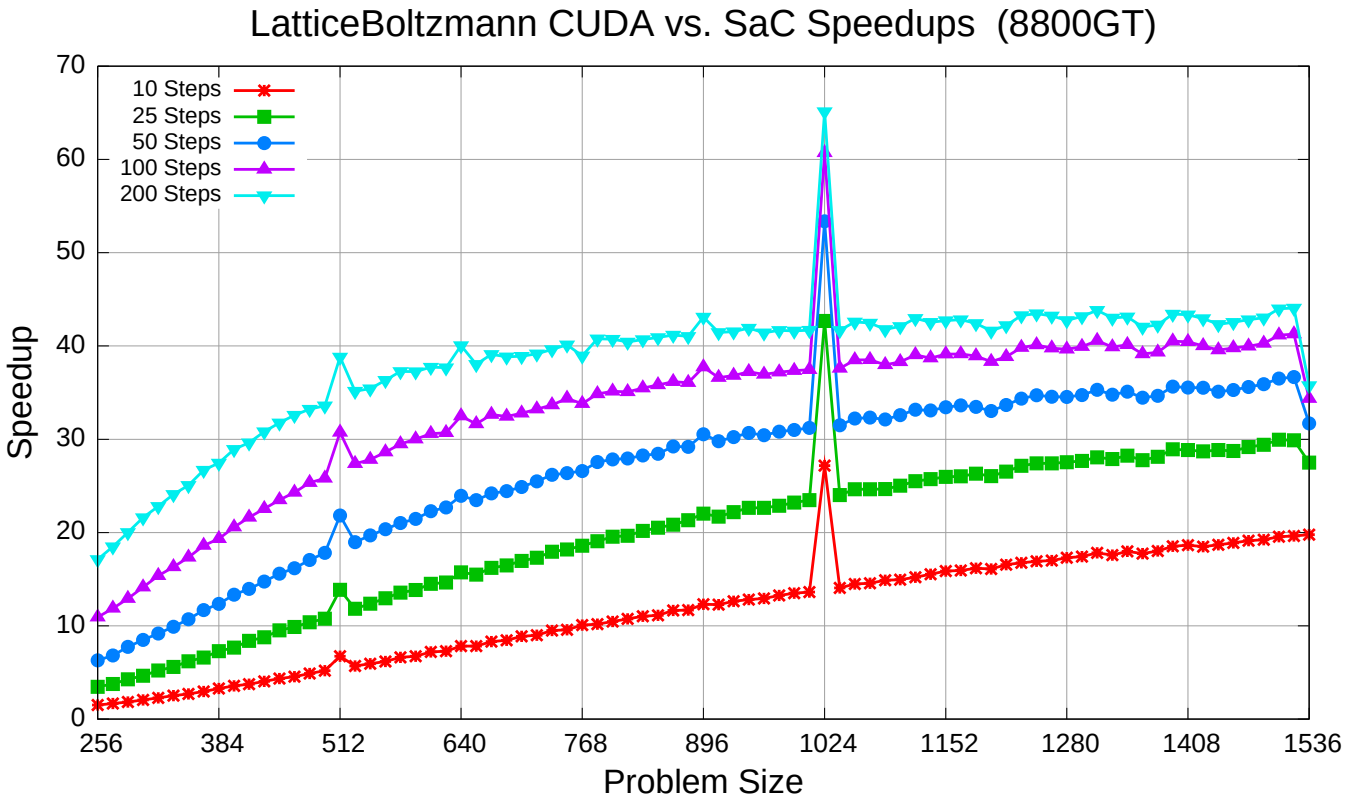
- Matrix product (`ipdd`)
- Acoustic signal processing (`logd`)
- Geophysics 1-D convolution (`mconv`)
- Today, `logd2` about 4.4X faster than APL on a 4-core box
- There are more optimizations to come. Soon.

Livermore Loops



- Livermore Loops speedup *vs.* vanilla serial SAC
- The Bad News: We still have work to do. . .
- The Good News: Lots of performance opportunities remain!

Computational Fluid Dynamics With GPU (CUDA)



- Two-dimensional flow using Lattice Boltzmann method
- Old, cheap Nvidia graphics card

Latent Parallelism

Observation: Array language programs have lots of parallelism

- More, perhaps, than we realize:
- Primitive function level (vector + vector)
- Derived function level (sum reduction on rows)
- Rank conjunction makes anything parallel: `foo"1 ⍵`
- Array expression level: ((A+B) * (C-D))
- Do old APL programs parallelize well?

Summary

- FAL programs short and lucid
- FAL serial performance is competitive with imperative languages
- FAL parallel performance beats imperative languages
- FAL do not require alteration for parallel operation

Thank you!

Questions?

References

Robert Bernecky.

Fortran 90 arrays.
ACM SIGPLAN Notices, 26(2), February 1991.

Robert Bernecky.

The role of APL and J in high-performance computation.
ACM SIGAPL Quote Quad, 24(1):17--32, August 1993.

Robert Bernecky.

APEX: The APL Parallel Executor.
Master's thesis, University of Toronto, 1997.

Robert Bernecky and R.K.W. Hui.

Gerunds and representations.
ACM SIGAPL Quote Quad, 21(4), July 1991.

Sven-Bodo Scholz.

Single Assignment C.
PhD thesis, Christian-Albrechts-Universität zu Kiel, 1996.

S.-B. Scholz.

With-loop-folding in SAC--Condensing Consecutive Array Operations.
In C. Clack, K.Hammond, and T. Davie, editors, *Implementation of Functional Languages, 9th International Workshop, IFL'97, St. Andrews, Scotland, UK, September 1997, Selected Papers*, volume 1467 of *LNCS*, pages 72--92. Springer, 1998.