Astounding Performance Looms!

Robert Bernecky

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

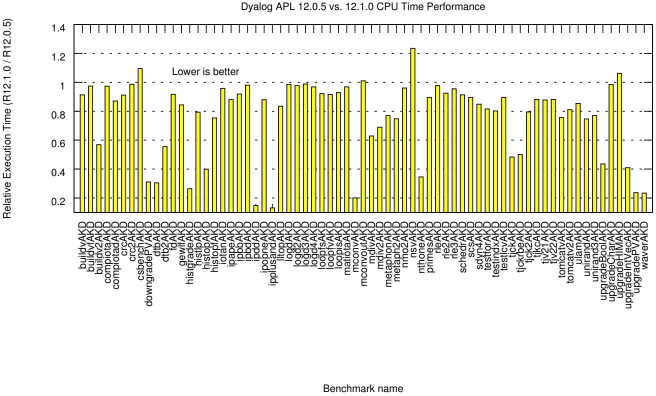
September 16, 2009

Abstract

Array languages, despite their proven advantages in time-to-solution and terse expression, continue to have a reputation for poor performance compared to imperative languages, such as C and Fortran. That reputation is about to change, thanks to recent advances in array compilation theory, APL's inherent parallelism, and the many-core computers that are now commonplace.

We showcase the state of the art of array languages, pitting interpreted APL code against compiled APL against Fortran 77 and Fortran 95, in both serial and parallel environments. We also outline how we propose to close the remaining performance gap between interpreted APL and compiled array languages.

Dyalog APL Performance: State of the Art



- Inner product speedups (ipdd, ipbd, mconv, waver)
- Grade speedups (downgradePV, upgradeHIM, upgradeIntVec)
- Generally, decent improvement across the board
- A few losers (nsv, csbench), to keep implementers humble

Compiled Array Languages

- SAC: Research Array language: Extended functional C
  - Language research projects
  - Serial performance projects (AWLF, WLF. . .)
  - Parallel performance projects
  - About 15 people working on compiler now
  - Compiler undergoing major refactoring (function spine, SAA opts)
- APEX: Research compiler: Extended flat APL, generates SAC or SISAL
- Fortran 9X: Fortran 77 with array extensions

Signal Processing

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

- Dyalog APL diff function

```
diff←{⍵-¯1⌽⍵}
```

Signal Processing

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

- Dyalog APL diff function

```
diff←{ω-¯1⌽ω}
```

- Fortran 95 diff function

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

Signal Processing

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

- Dyalog APL diff function

```
diff←{ω-¯1⌽ω}
```

- Fortran 95 diff function

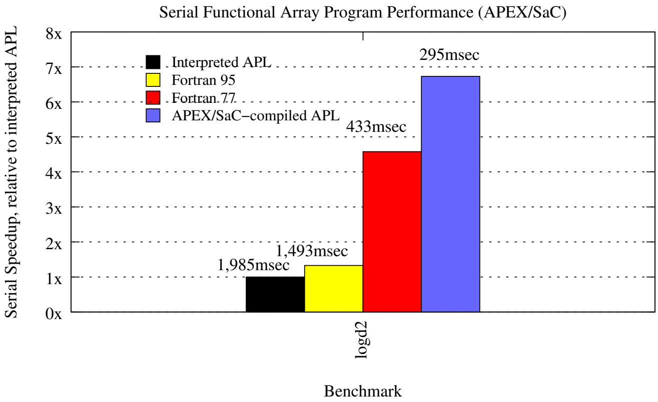
```
subroutine diff(wv,siz)
wv = wv - eoshift(wv,-1)
return & end
```

- 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
```

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

Compiled APEX Performance

- APL source code for `logd2`:

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

- SaC AKS With-Loop Folding (WLF) - Sven-Bodo Scholz

Compiled APEX Performance

- APL source code for `logd2`:
```
main:  +/logderiv 0.5+⍳ω
logderiv:  ¯50⌈50⌊50×(diff2 ω)÷ω+0.01
diff2:  ω-0,¯1↓ω
```
- SaC AKS With-Loop Folding (WLF) - Sven-Bodo Scholz
- The above code folds into ONE parallel with-loop

Compiled APEX Performance

- APL source code for `logd2`:
```
main:  +/logderiv 0.5+⍳⍵
logderiv:  ¯50⌈50⌊50×(diff2 ⍵)÷⍵+0.01
diff2:  ⍵-0,¯1↓⍵
```
- SaC AKS With-Loop Folding (WLF) - Sven-Bodo Scholz
- The above code folds into ONE parallel with-loop
- With Symbiotic Expressions & Algebraic WLF, also handles AKD arrays

Compiled APEX Performance

- APL source code for `logd2`:
```
main:  +/logderiv 0.5+⍳ω
logderiv:  ¯50⌈50⌊50×(diff2 ω)÷ω+0.01
diff2:  ω-0,¯1↓ω
```
- SaC AKS With-Loop Folding (WLF) - Sven-Bodo Scholz
- The above code folds into ONE parallel with-loop
- With Symbiotic Expressions & Algebraic WLF, also handles AKD arrays
- BENEFIT: Abstract expressionism.

APL Reduction

- +/⍳N

APL Reduction

- `+/⍳N`
- Generated code includes subtract - reduction order

APL Reduction

- +/⍳N
- Generated code includes subtract - reduction order
- +/⌽⍳N

APL Reduction

- +/⍳N
- Generated code includes subtract - reduction order
- +/⌽⍳N
- Generated code has NO subtract, no temps

APL Reduction

- +/⍳N
- Generated code includes subtract - reduction order
- +/⌽⍳N
- Generated code has NO subtract, no temps
- This is NOT idiom detection!

Finite Element Analysis - 2D Jacobi Relaxation

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

- SAC function

Finite Element Analysis - 2D Jacobi Relaxation

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

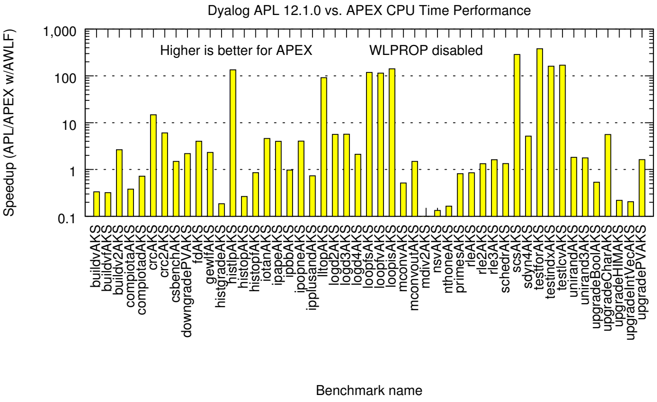
- SAC function
- This compiles into two data-parallel loops:

Finite Element Analysis - 2D Jacobi Relaxation

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

- SAC function
- This compiles into two data-parallel loops:
- It should compile into one loop, but not this week

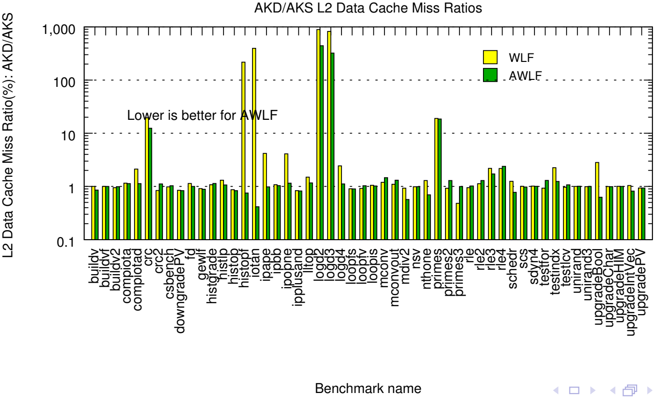
APEX Performance vs. APL



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

APEX Cache Performance

- L2 cache miss rates
- AKS - Arrays of Known Shape (Fortran 77)
- AKD - Arrays of Known Dimension (APL)
- WLF - With-Loop Folding (AKS-only)
- AWLF - Algebraic With-Loop Folding (AKS and AKD)



AWLF and Symbiotic Expressions

- Problem: Optimizers have to perform algebra

AWLF and Symbiotic Expressions

- Problem: Optimizers have to perform algebra
- *E.g.*, AWLF array index set intersection calculation

AWLF and Symbiotic Expressions

- Problem: Optimizers have to perform algebra
- *E.g.*, AWLF array index set intersection calculation
- Traditional solution I: Limit problem to simple cases

AWLF and Symbiotic Expressions

- Problem: Optimizers have to perform algebra
- *E.g.*, AWLF array index set intersection calculation
- Traditional solution I: Limit problem to simple cases
- Traditional solution II: Use SMT solver - Yices, Omega Calculator

AWLF and Symbiotic Expressions

- Problem: Optimizers have to perform algebra
- *E.g.*, AWLF array index set intersection calculation
- Traditional solution I: Limit problem to simple cases
- Traditional solution II: Use SMT solver - Yices, Omega Calculator
- Symbiotic Expressions: Glue problem onto program's AST

AWLF and Symbiotic Expressions

- Problem: Optimizers have to perform algebra
- *E.g.*, AWLF array index set intersection calculation
- Traditional solution I: Limit problem to simple cases
- Traditional solution II: Use SMT solver - Yices, Omega Calculator
- Symbiotic Expressions: Glue problem onto program's AST
- Abstract Syntax Tree gets lamprey-like code hanging from it

AWLF and Symbiotic Expressions

- Problem: Optimizers have to perform algebra
- *E.g.*, AWLF array index set intersection calculation
- Traditional solution I: Limit problem to simple cases
- Traditional solution II: Use SMT solver - Yices, Omega Calculator
- Symbiotic Expressions: Glue problem onto program's AST
- Abstract Syntax Tree gets lamprey-like code hanging from it
- Compiler's optimizers (CF, AL, AS, DL, CSE, CVP . . .) simplify

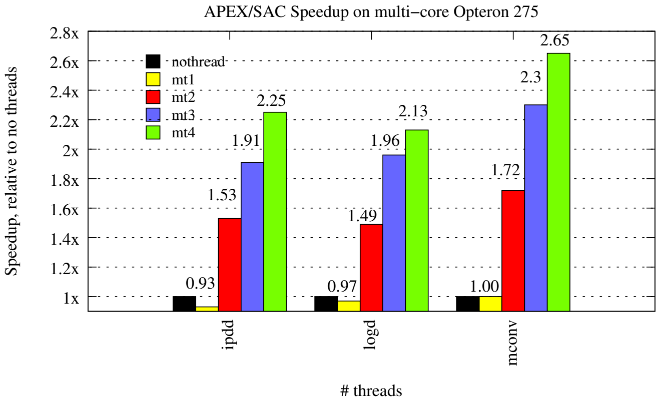
AWLF and Symbiotic Expressions

- Problem: Optimizers have to perform algebra
- *E.g.*, AWLF array index set intersection calculation
- Traditional solution I: Limit problem to simple cases
- Traditional solution II: Use SMT solver - Yices, Omega Calculator
- Symbiotic Expressions: Glue problem onto program's AST
- Abstract Syntax Tree gets lamprey-like code hanging from it
- Compiler's optimizers (CF, AL, AS, DL, CSE, CVP ...) simplify
- If suitably simplified, answer allows optimization to proceed

AWLF and Symbiotic Expressions

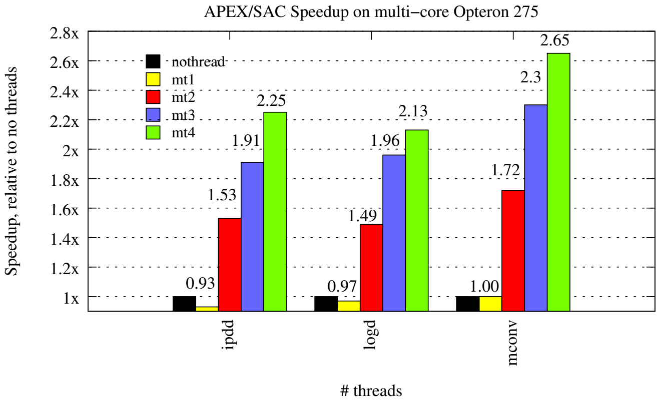
- Problem: Optimizers have to perform algebra
- *E.g.*, AWLF array index set intersection calculation
- Traditional solution I: Limit problem to simple cases
- Traditional solution II: Use SMT solver - Yices, Omega Calculator
- Symbiotic Expressions: Glue problem onto program's AST
- Abstract Syntax Tree gets lamprey-like code hanging from it
- Compiler's optimizers (CF, AL, AS, DL, CSE, CVP ...) simplify
- If suitably simplified, answer allows optimization to proceed
- Unlike the lamprey, both compiler and program benefit

Multi-thread APEX Performance on Opteron



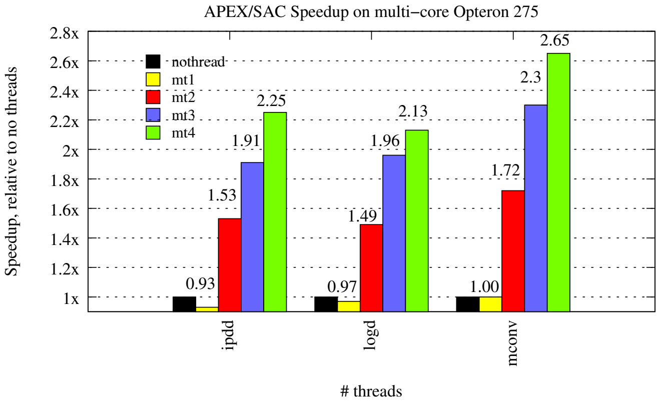
- Matrix product (ipdd)

Multi-thread APEX Performance on Opteron



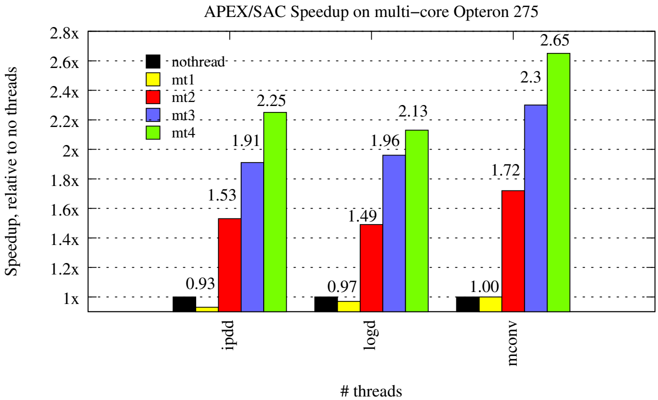
- Matrix product (`ipdd`)
- Acoustic signal processing (`logd`)

Multi-thread APEX Performance on Opteron



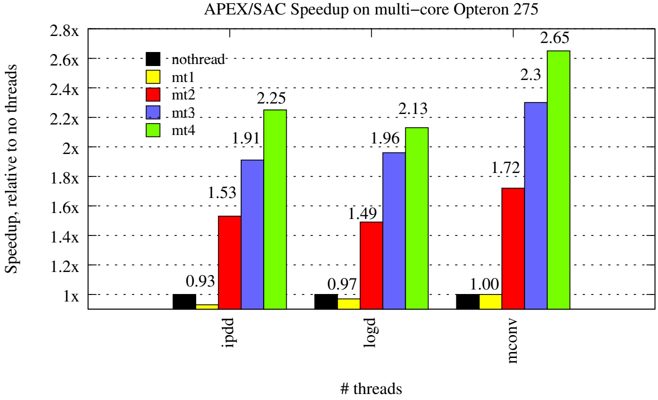
- Matrix product (`ipdd`)
- Acoustic signal processing (`logd`)
- Geophysics 1-D convolution (`mconv`)

Multi-thread APEX Performance on Opteron



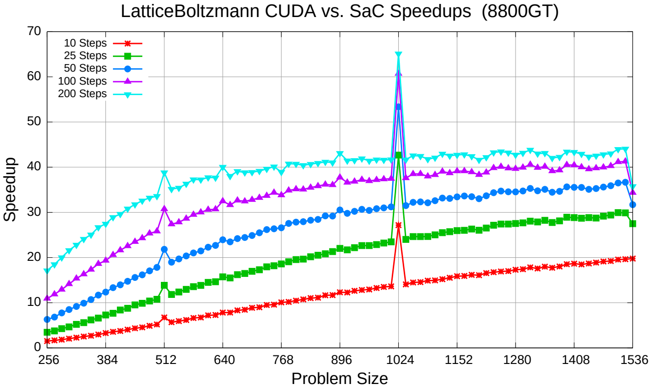
- Matrix product (`ipdd`)
- Acoustic signal processing (`logd`)
- Geophysics 1-D convolution (`mconv`)
- Today, `logd2` about 12X faster than APL on a 4-core box

Multi-thread APEX Performance on Opteron



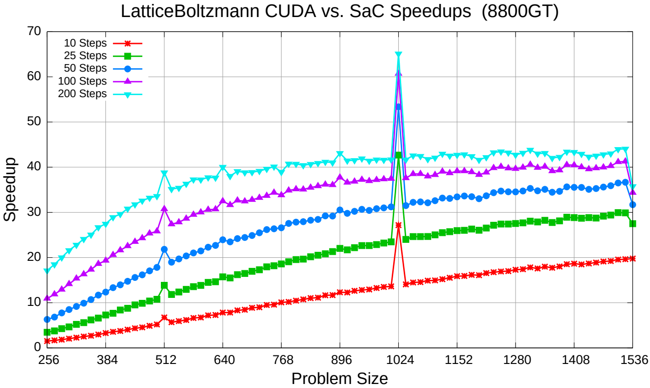
- Matrix product (**ipdd**)
- Acoustic signal processing (**logd**)
- Geophysics 1-D convolution (**mconv**)
- Today, **logd2** about 12X faster than APL on a 4-core box
- There are more optimizations to come. Soon.

Computational Fluid Dynamics With CUDA Back End



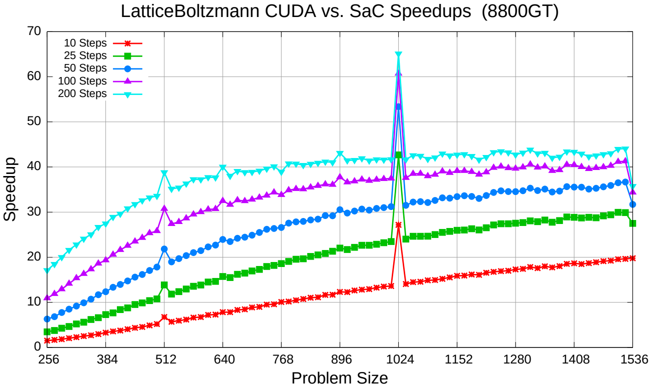
- Two-dimensional flow using Lattice Boltzmann method
- MuTC project in EU, UK
  - Single-tick thread initiation

Computational Fluid Dynamics With CUDA Back End



- Two-dimensional flow using Lattice Boltzmann method
- MuTC project in EU, UK
  - Single-tick thread initiation
  - Simulated linear speedup with 50K threads

Computational Fluid Dynamics With CUDA Back End



- Two-dimensional flow using Lattice Boltzmann method
- MuTC project in EU, UK
  - Single-tick thread initiation
  - Simulated linear speedup with 50K threads
  - Prototype SAC MuTC back-end exists

Bridging the Interpreter-Compiler Performance Gap

- Today: ⎕na calls to APEX
- access to optimized code

- Today: `⎕na` calls to APEX
- access to optimized code
  - multi-core execution

Bridging the Interpreter-Compiler Performance Gap

- Today: ⎕na calls to APEX
- access to optimized code
  - multi-core execution
  - access to CUDA

Bridging the Interpreter-Compiler Performance Gap

- Today: `⎕na` calls to APEX
- access to optimized code
  - multi-core execution
  - access to CUDA
  - Some overhead due to array copying across interface

Bridging the Interpreter-Compiler Performance Gap

- Today: `⎕na` calls to APEX
- access to optimized code
  - multi-core execution
  - access to CUDA
  - Some overhead due to array copying across interface
  - $\longrightarrow$ Slower for very small computations

- Tomorrow:
  - APEX performance improvements continue (2X-20X)

Bridging the Interpreter-Compiler Performance Gap

- Tomorrow:
  - APEX performance improvements continue (2X-20X)
  - Fastpath ⎕na call from APL to compiled code
    - Reduce and/or eliminate array copying across interface

Bridging the Interpreter-Compiler Performance Gap

- Tomorrow:
  - APEX performance improvements continue (2X-20X)
  - Fastpath `⎕na` call from APL to compiled code
    - Reduce and/or eliminate array copying across interface
  - JIT compiler for interpreted APL:
    - `A + B × ⍳ C` $\longrightarrow$ One parallel loop, no temp arrays
    - Reduce "each" hell: less memory fragmentation, much faster
    - Perhaps compile some class of dynamic functions
    - Compiled function cache

Bridging the Interpreter-Compiler Performance Gap

- Day after tomorrow:
- Possible APEX compiler extensions

Bridging the Interpreter-Compiler Performance Gap

- Day after tomorrow:
- Possible APEX compiler extensions
  - Nested arrays, structures

Bridging the Interpreter-Compiler Performance Gap

- Day after tomorrow:
- Possible APEX compiler extensions
  - Nested arrays, structures
  - Vendor-specific features, *e.g.*, dynamic functions

Bridging the Interpreter-Compiler Performance Gap

- Day after tomorrow:
- Possible APEX compiler extensions
  - Nested arrays, structures
  - Vendor-specific features, *e.g.*, dynamic functions
- Optimistic Algebraic With-Loop Folding

Joining Forces Could be Neat

- Traditional approach: One compiler for each language, each target system
- The GCC approach:
  - Compile F77, C, C++, F95 to common intermediate language (IL)
- Our approach:

Joining Forces Could be Neat

- Traditional approach: One compiler for each language, each target system
- The GCC approach:
  - Compile F77, C, C++, F95 to common intermediate language (IL)
  - Perform optimizations on that IL.
- Our approach:

Joining Forces Could be Neat

- Traditional approach: One compiler for each language, each target system
- The GCC approach:
  - Compile F77, C, C++, F95 to common intermediate language (IL)
  - Perform optimizations on that IL.
  - Generate code from the IL for specific target machine.
- Our approach:

Joining Forces Could be Neat

- Traditional approach: One compiler for each language, each target system
- The GCC approach:
  - Compile F77, C, C++, F95 to common intermediate language (IL)
  - Perform optimizations on that IL.
  - Generate code from the IL for specific target machine.
- Our approach:
  - Compile APL, J, A+, ... to common IL.

Joining Forces Could be Neat

- Traditional approach: One compiler for each language, each target system
- The GCC approach:
  - Compile F77, C, C++, F95 to common intermediate language (IL)
  - Perform optimizations on that IL.
  - Generate code from the IL for specific target machine.
- Our approach:
  - Compile APL, J, A+, ... to common IL.
  - Perform optimizations, perhaps SAC-based, on that IL.

Joining Forces Could be Neat

- Traditional approach: One compiler for each language, each target system
- The GCC approach:
  - Compile F77, C, C++, F95 to common intermediate language (IL)
  - Perform optimizations on that IL.
  - Generate code from the IL for specific target machine.
- Our approach:
  - Compile APL, J, A+, ... to common IL.
  - Perform optimizations, perhaps SAC-based, on that IL.
  - Generate code from the IL for specific target machine.

Joining Forces Could be Neat

- Traditional approach: One compiler for each language, each target system
- The GCC approach:
  - Compile F77, C, C++, F95 to common intermediate language (IL)
  - Perform optimizations on that IL.
  - Generate code from the IL for specific target machine.
- Our approach:
  - Compile APL, J, A+, . . . to common IL.
  - Perform optimizations, perhaps SAC-based, on that IL.
  - Generate code from the IL for specific target machine.
  - Need consortium of array language vendors, to produce:

Joining Forces Could be Neat

- Traditional approach: One compiler for each language, each target system
- The GCC approach:
  - Compile F77, C, C++, F95 to common intermediate language (IL)
  - Perform optimizations on that IL.
  - Generate code from the IL for specific target machine.
- Our approach:
  - Compile APL, J, A+, ... to common IL.
  - Perform optimizations, perhaps SAC-based, on that IL.
  - Generate code from the IL for specific target machine.
  - Need consortium of array language vendors, to produce:
  - New Extensible Array Translator:

Joining Forces Could be Neat

- Traditional approach: One compiler for each language, each target system
- The GCC approach:
  - Compile F77, C, C++, F95 to common intermediate language (IL)
  - Perform optimizations on that IL.
  - Generate code from the IL for specific target machine.
- Our approach:
  - Compile APL, J, A+, ... to common IL.
  - Perform optimizations, perhaps SAC-based, on that IL.
  - Generate code from the IL for specific target machine.
  - Need consortium of array language vendors, to produce:
  - New Extensible Array Translator:
  - *NEAT!*

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.

Thank you!

Questions?