Abstract Expressionism for Parallel Performance

Robert Bernecky[1] Sven-Bodo Scholz[2]

[1]Snake Island Research Inc, Canada
bernecky@snakeisland.com

[2]Heriot-Watt University, UK
S.Scholz@hw.ac.uk

This paper was presented at PLDI 2015, Portland, OR.

August 31, 2015

Abstract

Optimizing Functional Array Language (FAL) compilers for languages such as APL (APEX) and SAC (sac2c), now produce code that outperforms hand-optimized C in both serial and parallel arenas, while retaining the abstract expressionist nature of well-written FAL code.
In this talk, we demonstrate how FAL can now outperform C, in both serial and OpenMP variants, by up to a third, with *no* source code modifications. We also show that modern optimizers can sometimes generate identical loops from substantially different FAL source code.

Talk Layout

- Serial performance: physics relaxation benchmark

Talk Layout

- Serial performance: physics relaxation benchmark
- Parallel performance: physics relaxation benchmark

Talk Layout

- Serial performance: physics relaxation benchmark
- Parallel performance: physics relaxation benchmark
- Wild applause

A Physics Benchmark: Vector Relaxation

- Inputs: temperatures (fixed) at each end of N-element rod

Dyalog APL/S-64 Version 14.1.25324
8-core AMD FX-8350 (Piledriver) @ 4013MHz, 32GB DRAM
Ubuntu 14.04LTS, sac2c Build #18605, gcc 4.8.2-19ubuntu1
100000 iterations of relaxation kernel
100001-element vector argument, N

A Physics Benchmark: Vector Relaxation

- Inputs: temperatures (fixed) at each end of *N*-element rod
- Output: End element temperatures remain unchanged;
  Other element temps are arithmetic mean of neighbors

Dyalog APL/S-64 Version 14.1.25324
8-core AMD FX-8350 (Piledriver) @ 4013MHz, 32GB DRAM
Ubuntu 14.04LTS, sac2c Build #18605, gcc 4.8.2-19ubuntu1
100000 iterations of relaxation kernel
100001-element vector argument, *N*

A Physics Benchmark: Vector Relaxation

- Inputs: temperatures (fixed) at each end of N-element rod
- Output: End element temperatures remain unchanged;
  Other element temps are arithmetic mean of neighbors
- Application: image processing, *e.g.*, dust removal (2D)

Dyalog APL/S-64 Version 14.1.25324
8-core AMD FX-8350 (Piledriver) @ 4013MHz, 32GB DRAM
Ubuntu 14.04LTS, sac2c Build #18605, gcc 4.8.2-19ubuntu1
100000 iterations of relaxation kernel
100001-element vector argument, N

A Physics Benchmark: Vector Relaxation

- Inputs: temperatures (fixed) at each end of *N*-element rod
- Output: End element temperatures remain unchanged;
  Other element temps are arithmetic mean of neighbors
- Application: image processing, *e.g.*, dust removal (2D)
- Application: temperature distribution in a rod

Dyalog APL/S-64 Version 14.1.25324
8-core AMD FX-8350 (Piledriver) @ 4013MHz, 32GB DRAM
Ubuntu 14.04LTS, sac2c Build #18605, gcc 4.8.2-19ubuntu1
100000 iterations of relaxation kernel
100001-element vector argument, *N*

Three Ways to do Vector Relaxation in Dyalog APL

- Abstract: No tinkering of “memory"

Three Ways to do Vector Relaxation in Dyalog APL

- Abstract: No tinkering of “memory"
- Expressions: No need for variables (convenience only)

Abstract Expressionism in Dyalog APL

Three Ways to do Vector Relaxation in Dyalog APL

- Abstract: No tinkering of “memory"
- Expressions: No need for variables (convenience only)
- `TD←{(1↑ω),(((2↓ω)+¯2↓ω)÷2.0),¯1↑ω}`

Three Ways to do Vector Relaxation in Dyalog APL

- Abstract: No tinkering of “memory"
- Expressions: No need for variables (convenience only)
- `TD←{(1↑ω),(((2↓ω)+¯2↓ω)÷2.0),¯1↑ω}`
-
```
ROT←{N←ρω
      m←(0=ιN)∨(N−1)=ιN
      (m×ω)+(~m)×((1ϕω)+¯1ϕω)÷2.0}
```

Abstract Expressionism in Dyalog APL

Three Ways to do Vector Relaxation in Dyalog APL

- Abstract: No tinkering of "memory"
- Expressions: No need for variables (convenience only)
- `TD←{(1↑ω),(((2↓ω)+¯2↓ω)÷2.0),¯1↑ω}`
- ROT:

```
ROT←{N←ρω
      m←(0=ιN)∨(N−1)=ιN
      (m×ω)+(~m)×((1⌽ω)+¯1⌽ω)÷2.0}
```

- SHF:

```
SHF←{N←ρω
      m←(0=ιN)∨(N−1)=ιN
      (m×ω)+(~m)×((1 shift ω)+¯1 shift ω)÷2}
      shift←{((×α)×ρω)↑α↓ω}
```

Serial Relaxation Timings in Dyalog APL

```
TD←{(1↑ω),(((2↓ω)+¯2↓ω)÷2.0),¯1↑ω}
ROT←{N←ρω
     m←(0=ιN)∨(N−1)=ιN
     (m×ω)+(~m)×((1⌽ω)+¯1⌽ω)÷2.0}
SHF←{N←ρω
     m←(0=ιN)∨(N−1)=ιN
     (m×ω)+(~m)×((1 shift ω)+¯1 shift ω)÷2}
     shift←{((×α)×ρω)↑α↓ω}
```

- Timings:

APL TD	82.6s
APL ROT	203.9s
APL SHF	236.9s

Serial Relaxation in C Using IF/THEN/ELSE

```
for( j=0; j<N; j++) {
   if(0==j) {
     res[j] = v[j];
   } else if((N-1)==j) {
     res[j] = v[j];
   } else {
     res[j] = (v[j-1] + v[j+1])/2.0;
   }
}
```

- Timings:

APL TD	82.6s
APL ROT	203.9s
APL SHF	236.9s

Serial Relaxation in C Using IF/THEN/ELSE

```
for( j=0; j<N; j++) {
   if(0==j) {
     res[j] = v[j];
   } else if((N-1)==j) {
     res[j] = v[j];
   } else {
     res[j] = (v[j-1] + v[j+1])/2.0;
   }
}
```

- Timings:

APL TD	82.6s
APL ROT	203.9s
APL SHF	236.9s
C IF/THEN/ELSE	16.3s

Serial Relaxation in C Using Conditional Expressions

```
for( j=0; j<N; j++) {
  res[j] = (0==j)      ? v[j] :
           ((N-1)==j) ? v[j] :
             (v[j-1] + v[j+1])/2.0;
}
```

- Timings:

APL TD	82.6s
APL ROT	203.9s
APL SHF	236.9s
C IF/THEN/ELSE	16.3s
C COND	16.4s

Serial Relaxation in SAC Using Conditional Expressions

```
res = with {
        ([0] <= [j] < [N]) :
          (0==j)      ? v[j] :
          ((N-1)==j) ? v[j] :
            (v[j-1] + v[j+1])/2.0;
      } : modarray( v);
```

- Timings:

APL TD	82.6s
APL ROT	203.9s
APL SHF	236.9s
C IF/THEN/ELSE	16.3s
C COND	16.4s
SAC COND	12.0s

Serial Relaxation in SAC, Hand-Optimized

Can SAC do better?
Three data-parallel With-Loop partitions:

```
res = with {
      ([0]   <= [j] < [1]) : v[j];
      ([1]   <= [j] < [N-1]) :
         (v[j-1] + v[j+1])/2.0;
      ([N-1] <= [j] < [N]) : v[j];
    } : modarray( v);
```

- Timings:

APL TD	82.6s
APL ROT	203.9s
APL SHF	236.9s
C IF/THEN/ELSE	16.3
C COND	16.4
SAC COND	12.0s
SAC HAND	5.9s

Serial Relaxation using Abstract Expressionism and APEX

- Take and drop algorithm in APEX

Serial Relaxation using Abstract Expressionism and APEX

- Take and drop algorithm in APEX
- `TD←{(1↑ω),(((2↓ω)+¯2↓ω)÷2.0),¯1↑ω}`

Serial Relaxation using Abstract Expressionism and APEX

- Take and drop algorithm in APEX
- `TD←{(1↑ω),(((2↓ω)+¯2↓ω)÷2.0),¯1↑ω}`
- Approximate APEX-generated SAC code

```
mid = (drop([2],v)+drop([-2],v))/2.0;
res = take([1],v)++mid++take([-1],v);
```

Serial Relaxation using Abstract Expressionism and APEX

- Take and drop algorithm in APEX
- `TD←{(1↑ω),(((2↓ω)+¯2↓ω)÷2.0),¯1↑ω}`
- Approximate APEX-generated SAC code

```
mid = (drop([2],v)+drop([-2],v))/2.0;
res = take([1],v)++mid++take([-1],v);
```

- Timings:

APL TD	82.6s
SAC HAND	5.9s
APEX TD	5.9s

Serial Relaxation using Abstract Expressionism and APEX

- Take and drop algorithm in APEX
- `TD←{(1↑ω),(((2↓ω)+¯2↓ω)÷2.0),¯1↑ω}`
- Approximate APEX-generated SAC code

```
mid = (drop([2],v)+drop([-2],v))/2.0;
res = take([1],v)++mid++take([-1],v);
```

- Timings:

APL TD	82.6s
SAC HAND	5.9s
APEX TD	5.9s

- *Identical* inner loops for `APEX TD` and `SAC HAND`

Serial Relaxation using Abstract Expressionism and APEX

```
ROT←{N←ρω
      m←(0=ιN)∨(N−1)=ιN
      (m×ω)+(~m)×((1⌽ω)+¯1⌽ω)÷2.0}
```

```
m = (0 == iota(N)) | ((N-1) == iota(N));
res = (tod(m) * v) + tod(!m) *
  ((rotate([1], v) + rotate([-1], v)))/2.0;
```

- Rotate algorithm in APEX, generated SAC code

Serial Relaxation using Abstract Expressionism and APEX

```
ROT←{N←⍴⍵
     m←(0=⍳N)∨(N-1)=⍳N
     (m×⍵)+(~m)×((1⌽⍵)+¯1⌽⍵)÷2.0}
```

```
m = (0 == iota(N)) | ((N-1) == iota(N));
res = (tod(m) * v) + tod(!m) *
  ((rotate([1], v) + rotate([-1], v)))/2.0;
```

- Rotate algorithm in APEX, generated SAC code
- Timings:

APL ROT	82.6s
SAC HAND	5.9s
APEX ROT	5.9s

Serial Relaxation using Abstract Expressionism and APEX

```
ROT←{N←ρω
     m←(0=ιN)∨(N−1)=ιN
     (m×ω)+(~m)×((1⌽ω)+¯1⌽ω)÷2.0}
```

```
m = (0 == iota(N)) | ((N-1) == iota(N));
res = (tod(m) * v) + tod(!m) *
    ((rotate([1], v) + rotate([-1], v)))/2.0;
```

- Rotate algorithm in APEX, generated SAC code
- Timings:

APL ROT	82.6s
SAC HAND	5.9s
APEX ROT	5.9s

- *Identical* inner loops for APEX ROT and SAC HAND

Serial Relaxation using Abstract Expressionism and APEX

```
SHF←{N←⍴⍵
     m←(0=⍳N)∨(N−1)=⍳N
     (m×⍵)+(~m)×((1 shift ⍵)+¯1 shift ⍵)÷2}
     shift←{((×⍺)×⍴⍵)↑⍺↓⍵}
```

```
m = (0 == iota(N)) | ((N-1) == iota(N));
res = (tod(m) * v) + tod(!m) *
 ((shift([1],v) + shift([-1],v)))/2.0;
```

- Shift algorithm in APEX-generated SAC code

Serial Relaxation using Abstract Expressionism and APEX

```
SHF←{N←⍴⍵
     m←(0=⍳N)∨(N−1)=⍳N
     (m×⍵)+(~m)×((1 shift ⍵)+¯1 shift ⍵)÷2}
     shift←{((×⍺)×⍴⍵)↑⍺↓⍵}
```

```
m = (0 == iota(N)) | ((N-1) == iota(N));
res = (tod(m) * v) + tod(!m) *
 ((shift([1],v) + shift([-1],v)))/2.0;
```

- Shift algorithm in APEX-generated SAC code
- Timings:

APL TD	82.6s
APL ROT	203.9s
APL SHF	236.9s
SAC HAND	5.9s
APEX TD	5.9s
APEX ROT	5.9s
APEX SHIFT	5.9s

Serial Relaxation using Abstract Expressionism and APEX

```
SHF←{N←ρω
     m←(0=ιN)∨(N−1)=ιN
     (m×ω)+(~m)×((1 shift ω)+¯1 shift ω)÷2}
     shift←{((×α)×ρω)↑α↓ω}
```

```
m = (0 == iota(N)) | ((N-1) == iota(N));
res = (tod(m) * v) + tod(!m) *
 ((shift([1],v) + shift([-1],v)))/2.0;
```

- Shift algorithm in APEX-generated SAC code
- Timings:

APL TD	82.6s
APL ROT	203.9s
APL SHF	236.9s
SAC HAND	5.9s
APEX TD	5.9s
APEX ROT	5.9s
APEX SHIFT	5.9s

- *ALL* inner loops are identical!

Why are Identical Inner Loops Noteworthy?

- APL source codes differ substantially

Why are Identical Inner Loops Noteworthy?

- APL source codes differ substantially
- Very different SAC stdlib code for rotate, shift, take/drop

Why are Identical Inner Loops Noteworthy?

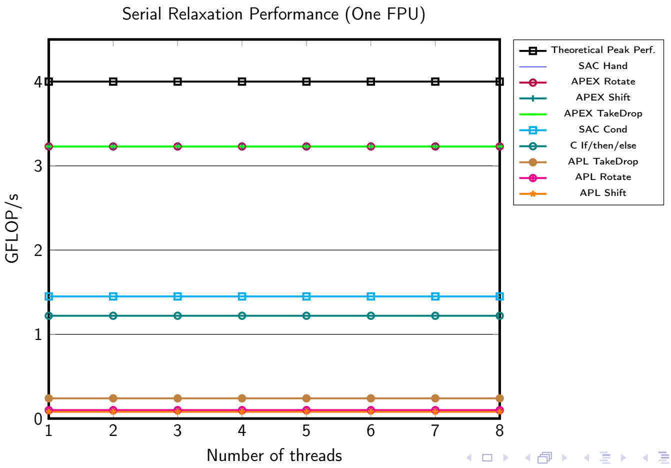
- APL source codes differ substantially
- Very different SAC stdlib code for rotate, shift, take/drop
- *E.g.*, number of With-Loops, setup code style

Why are Identical Inner Loops Noteworthy?

- APL source codes differ substantially
- Very different SAC stdlib code for rotate, shift, take/drop
- *E.g.*, number of With-Loops, setup code style
- See paper for stdlib code, here: `http://www.snakeisland.com/abstractexpressionism.pdf`

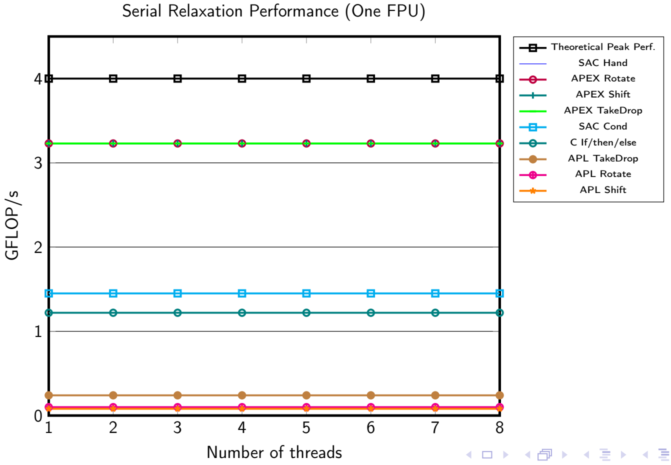
Serial Performance GFLOPS

- Hard to do better? SAC/APEX approach maximum GFLOPS rate



Serial Performance GFLOPS

- Hard to do better? SAC/APEX approach maximum GFLOPS rate
- Let's look at parallel execution



Parallel Relaxation Speedup in C

- Open MP

Parallel Relaxation Speedup in C

- Open MP
- Basic idea: Introduce ceremonial rubbish into **SOURCE** code

Parallel Relaxation Speedup in C

- Open MP
- Basic idea: Introduce ceremonial rubbish into **SOURCE** code
- See paper for ceremonial details

Parallel Relaxation Speedup in C

- Open MP
- Basic idea: Introduce ceremonial rubbish into **SOURCE** code
- See paper for ceremonial details
- Basic idea: Introduce pragmas into **SOURCE** code

  ```
  #pragma omp parallel for
  ```

  after *SOME* `for` statements.

Parallel Relaxation Speedup in C

- Open MP
- Basic idea: Introduce ceremonial rubbish into **SOURCE** code
- See paper for ceremonial details
- Basic idea: Introduce pragmas into **SOURCE** code

  ```
  #pragma omp parallel for
  ```

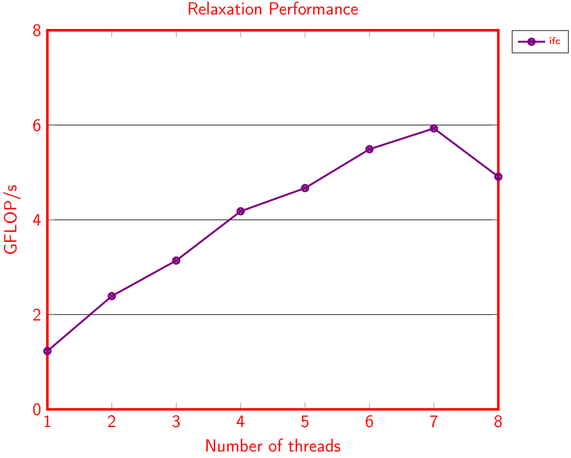
  after *SOME* `for` statements.
- Compile with `-fopenmp`

Parallel Relaxation Speedup in C Performance

- Timings: (higher is better)

Parallel Relaxation Speedup in C Performance

- Timings: (higher is better)



Optimized Parallel Relaxation in C

```
for( j=0; j<N; j++) {
   if(0==j) {
     res[j] = v[j];
   } else if((N-1)==j) {
     res[j] = v[j];
   } else {
     res[j] = (v[j-1] + v[j+1])/2.0;
   }
}
```

- Bright idea: Replace multiple "`res[j] =`" by "`el =`"

Optimized Parallel Relaxation in C

```
for( j=0; j<N; j++) {
   if(0==j) {
     res[j] = v[j];
   } else if((N-1)==j) {
     res[j] = v[j];
   } else {
     res[j] = (v[j-1] + v[j+1])/2.0;
   }
}
```

- Bright idea: Replace multiple "`res[j] =`" by "`el =`"
- Bright idea: and add "`res[j] = el;`" after IF-statement

Optimized Parallel Relaxation in C

```
for( j=0; j<N; j++) {
   if(0==j) {
     res[j] = v[j];
   } else if((N-1)==j) {
     res[j] = v[j];
   } else {
     res[j] = (v[j-1] + v[j+1])/2.0;
   }
}
```

- Bright idea: Replace multiple "`res[j] =`" by "`el =`"
- Bright idea: and add "`res[j] = el;`" after IF-statement
- Rationale: Eliminate multiple indexed assigns into "`res`"

Optimized Parallel Relaxation in C

```
for( j=0; j<N; j++) {
   if(0==j) {
     res[j] = v[j];
   } else if((N-1)==j) {
     res[j] = v[j];
   } else {
     res[j] = (v[j-1] + v[j+1])/2.0;
   }
}
```

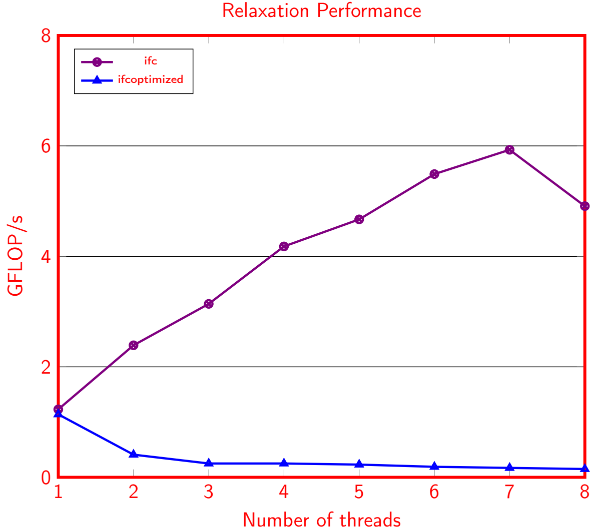
- Bright idea: Replace multiple "`res[j] =`" by "`el =`"
- Bright idea: and add "`res[j] = el;`" after IF-statement
- Rationale: Eliminate multiple indexed assigns into "`res`"
- This should improve instruction cache use

Pessimized Parallel Relaxation in C

- Timings: (higher is better)

Pessimized Parallel Relaxation in C

- Timings: (higher is better)



Parallel Relaxation Slowdown in C Post-mortem

```
for( j=0; j<N; j++) {
   if(0==j) {
     el = v[j];
   } else if((N-1)==j) {
     el = v[j];
   } else {
     el = (v[j-1] + v[j+1])/2.0;
   }
   res[j] = el;
}
```

- What went wrong?

Parallel Relaxation Slowdown in C Post-mortem

```
for( j=0; j<N; j++) {
    if(0==j) {
      el = v[j];
    } else if((N-1)==j) {
      el = v[j];
    } else {
      el = (v[j-1] + v[j+1])/2.0;
    }
    res[j] = el;
}
```

- What went wrong?
- `el` is shared, so it hops among all threads

Parallel Relaxation Slowdown in C Post-mortem

```
for( j=0; j<N; j++) {
   if(0==j) {
     el = v[j];
   } else if((N-1)==j) {
     el = v[j];
   } else {
     el = (v[j-1] + v[j+1])/2.0;
   }
   res[j] = el;
}
```

- What went wrong?
- `el` is shared, so it hops among all threads
- Bottom line: Bright idea not so bright (watch system monitor!)

Parallel Relaxation Slowdown in C Post-mortem

```
for( j=0; j<N; j++) {
   if(0==j) {
     el = v[j];
   } else if((N-1)==j) {
     el = v[j];
   } else {
     el = (v[j-1] + v[j+1])/2.0;
   }
   res[j] = el;
}
```

- What went wrong?
- `el` is shared, so it hops among all threads
- Bottom line: Bright idea not so bright (watch system monitor!)
- Bottom line: Writing parallel C code is **NOT** trivial

Serial and Parallel Relaxation Performance

- Abstract expressionist APL matches best SAC code

Serial and Parallel Relaxation Performance

- Abstract expressionist APL matches best SAC code
- SAC and APL beat C by 2.75X in serial environment

Serial and Parallel Relaxation Performance

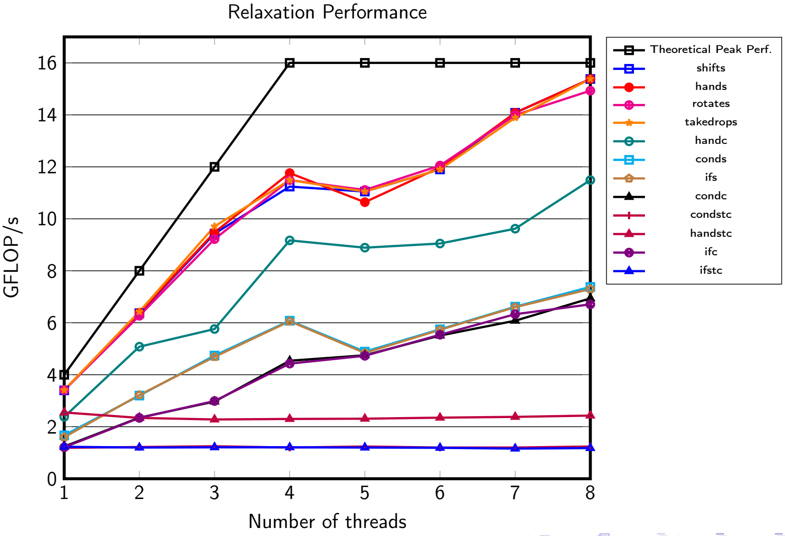
- Abstract expressionist APL matches best SAC code
- SAC and APL beat C by 2.75X in serial environment
- SAC and APL beat Open MP C by 1/3 in parallel environment

Serial and Parallel Relaxation Performance

- Abstract expressionist APL matches best SAC code
- SAC and APL beat C by 2.75X in serial environment
- SAC and APL beat Open MP C by 1/3 in parallel environment
- *NO* changes to APL code for parallel execution, unlike C

Serial and Parallel Relaxation Performance

Higher is better



SAC Keys to High-Performance FAL Compilation

- Provide purely functional Intermediate Language (IL)

SAC Keys to High-Performance FAL Compilation

- Provide purely functional Intermediate Language (IL)
- Preserve array semantics in IL

SAC Keys to High-Performance FAL Compilation

- Provide purely functional Intermediate Language (IL)
- Preserve array semantics in IL
- Scalarize small arrays, *e.g.*:

SAC Keys to High-Performance FAL Compilation

- Provide purely functional Intermediate Language (IL)
- Preserve array semantics in IL
- Scalarize small arrays, *e.g.*:
- in Gaussian Elimination pivot, replacing:

```
mat[rowa,rowb;]←mat[rowb,rowa;]
```

by

```
trow←mat[rowa;] ⋄ mat[rowa;]←mat[rowb;] ⋄
mat[rowb;]←trow
```

SAC Keys to High-Performance FAL Compilation

- Provide purely functional Intermediate Language (IL)
- Preserve array semantics in IL
- Scalarize small arrays, *e.g.*:
- in Gaussian Elimination pivot, replacing:

```
mat[rowa,rowb;]←mat[rowb,rowa;]
```

by

```
trow←mat[rowa;] ◊ mat[rowa;]←mat[rowb;] ◊
mat[rowb;]←trow
```

- . . . gives 2X speedup!

SAC Keys to High-Performance FAL Compilation

- Provide purely functional Intermediate Language (IL)
- Preserve array semantics in IL
- Scalarize small arrays, *e.g.*:
- in Gaussian Elimination pivot, replacing:

  `mat[rowa,rowb;]←mat[rowb,rowa;]`

  by

  `trow←mat[rowa;] ⋄ mat[rowa;]←mat[rowb;] ⋄ mat[rowb;]←trow`
- . . . gives 2X speedup!
- Do scalarization in the compiler, *NOT* in app source code.

SAC Keys to High-Performance FAL Compilation

- Provide purely functional Intermediate Language (IL)
- Preserve array semantics in IL
- Scalarize small arrays, *e.g.*:
- in Gaussian Elimination pivot, replacing:

  `mat[rowa,rowb;]←mat[rowb,rowa;]`

  by

  `trow←mat[rowa;] ⋄ mat[rowa;]←mat[rowb;] ⋄ mat[rowb;]←trow`
- ... gives 2X speedup!
- Do scalarization in the compiler, *NOT* in app source code.
- Use array-based optimizations, *e.g.*, with-loop folding (WLF)

SAC Keys to High-Performance FAL Compilation

- Provide purely functional Intermediate Language (IL)
- Preserve array semantics in IL
- Scalarize small arrays, *e.g.*:
- in Gaussian Elimination pivot, replacing:
  `mat[rowa,rowb;]←mat[rowb,rowa;]`
  by
  `trow←mat[rowa;] ◊ mat[rowa;]←mat[rowb;] ◊ mat[rowb;]←trow`
- …gives 2X speedup!
- Do scalarization in the compiler, *NOT* in app source code.
- Use array-based optimizations, *e.g.*, with-loop folding (WLF)
- and others…

SAC Keys to High-Performance FAL Compilation

- Provide purely functional Intermediate Language (IL)
- Preserve array semantics in IL
- Scalarize small arrays, *e.g.*:
- in Gaussian Elimination pivot, replacing:

```
mat[rowa,rowb;]←mat[rowb,rowa;]
```

by

```
trow←mat[rowa;] ⋄ mat[rowa;]←mat[rowb;] ⋄
mat[rowb;]←trow
```

- ...gives 2X speedup!
- Do scalarization in the compiler, *NOT* in app source code.
- Use array-based optimizations, *e.g.*, with-loop folding (WLF)
- and others...
- Stay tuned for the book!

This work was supported in part by grant EP/L00058X/1, from the UK Engineering and Physical Sciences Research Council (EPSRC). The late Ken Iverson, an Albertan farm boy, had many excellent insights, for which we are grateful. The excellent performance of the `sac2c` compiler is due to the diligence of many researchers, whose contributions can be found on the SaC web site at `http:sac-home.org`. Our thanks to Philip Mucci and John D. McCalpin for answering our AMD architecture questions. We also thank the anonymous referees for their thoughtful comments.

Thank you! Questions?