Abstract Expressionism for Parallel Performance

Robert Bernecky\textsuperscript{1} \quad Sven-Bodo Scholz\textsuperscript{2}

\textsuperscript{1}Snake Island Research Inc, Canada
bernecky@snakeisland.com

\textsuperscript{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.
Serial performance: physics relaxation benchmark
Talk Layout

- Serial performance: physics relaxation benchmark
- Parallel performance: physics relaxation benchmark
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)
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←{N←ρω
  m←(0=1N)∨(N-1)=1N
  (m×ω)+(~m)×((1φω)+¯1φω)÷2.0}
- SHF←{N←ρω
  m←(0=1N)∨(N-1)=1N
  (m×ω)+(~m)×((1 shift ω)+¯1 shift ω)÷2}
  shift←{(((×α)×ρω)↑α↓ω}
Serial Relaxation Timings in Dyalog APL

\[ \text{TD} \leftarrow \{ (1 \uparrow \omega), ((2 \downarrow \omega) + (-2 \downarrow \omega)) \div 2.0), (-1 \uparrow \omega) \} \]
\[ \text{ROT} \leftarrow \{ N \leftarrow p\omega \]
\[ m \leftarrow (0 \equiv N) \lor (N-1) \equiv N \\
(\text{m} \times \omega) + (\neg m) \times ((1 \phi \omega) + (-1 \phi \omega)) \div 2.0 \} \]
\[ \text{SHF} \leftarrow \{ N \leftarrow p\omega \]
\[ m \leftarrow (0 \equiv N) \lor (N-1) \equiv N \\
(\text{m} \times \omega) + (\neg m) \times ((1 \text{ shift } \omega) + (-1 \text{ shift } \omega)) \div 2 \} \]
\[ \text{shift} \leftarrow \{ ((\times \alpha) \times p\omega) \uparrow \alpha \downarrow \omega \} \]

- **APL TD**: 82.6s
- **APL ROT**: 203.9s
- **APL SHF**: 236.9s

**Timings:**
Serial Relaxation in C Using IF/THEN/ELSE

```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;
    }
}
```

**Timings:**

- APL TD: 82.6s
- APL ROT: 203.9s
- APL SHF: 236.9s
Serial Relaxation in C Using IF/THEN/ELSE

```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;
    }
}
```

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

Robert Bernecky, Sven-Bodo Scholz

Abstract Expressionism for Parallel Performance
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:

<table>
<thead>
<tr>
<th>Language</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>APL TD</td>
<td>82.6s</td>
</tr>
<tr>
<td>APL ROT</td>
<td>203.9s</td>
</tr>
<tr>
<td>APL SHF</td>
<td>236.9s</td>
</tr>
<tr>
<td>C IF/THEN/ELSE</td>
<td>16.3s</td>
</tr>
<tr>
<td>C COND</td>
<td>16.4s</td>
</tr>
<tr>
<td>SAC COND</td>
<td>12.0s</td>
</tr>
</tbody>
</table>
Serial Relaxation in SAC, Hand-Optimized

Can SAC do better?

Three data-parallel With-Loop partitions:

```plaintext
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

Robert Bernecky, Sven-Bodo Scholz

Abstract Expressionism for Parallel Performance
Take and drop algorithm in APEX
Serial Relaxation using Abstract Expressionism and APEX

- Take and drop algorithm in APEX
- \[ \text{TD} \leftarrow \{(1 \uparrow \omega), ((2 \downarrow \omega) + \neg 2 \downarrow \omega) \div 2.0), \neg 1 \uparrow \omega\} \]
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

- Take and drop algorithm in APEX
  \[ \text{TD} \leftarrow \{ (1 \uparrow \omega), ((2 \downarrow \omega) + \neg 2 \downarrow \omega) \div 2.0, \neg 1 \uparrow \omega \} \]

- Approximate APEX-generated SAC code
  \[
  \text{mid} = (\text{drop}([2], v) + \text{drop}([-2], v)) / 2.0;
  \text{res} = \text{take}([1], v) + \text{mid} + \text{take}([-1], v);
  \]

- Timings:
<table>
<thead>
<tr>
<th></th>
<th>APL TD</th>
<th>SAC HAND</th>
<th>APEX TD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>82.6s</td>
<td>5.9s</td>
<td>5.9s</td>
</tr>
</tbody>
</table>

Identical inner loops for APEX TD and SAC HAND
Serial Relaxation using Abstract Expressionism and APEX

- Take and drop algorithm in APEX
  - \( \text{TD} \leftarrow \{ (1 \uparrow \omega), (((2 \downarrow \omega) + \neg 2 \downarrow \omega) \div 2.0), \neg 1 \uparrow \omega \} \)

- Approximate APEX-generated SAC code
  
  \[
  \text{mid} = (\text{drop}([2], v) + \text{drop}([-2], v)) / 2.0;
  \text{res} = \text{take}([1], v) ++ \text{mid} ++ \text{take}([-1], v);
  \]

- 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

\[ \text{ROT} \left\{ \text{N} \leftarrow \rho \omega \right. \]

\[ m \leftarrow (0 = \iota(N)) \lor (N-1) = \iota(N) \]

\[ (m \times \omega) + (\neg m) \times ((1 \phi \omega) + \neg 1 \phi \omega) \div 2.0 \}

\[ m = (0 \equiv \iota(N)) \lor ((N-1) \equiv \iota(N)); \]

\[ \text{res} = (\text{tod}(m) \times v) + \text{tod}(\neg m) \times \]

\[ ((\text{rotate}([1], v) + \text{rotate}([-1], v)))/2.0; \]

- Rotate algorithm in APEX, generated SAC code
Serial Relaxation using Abstract Expressionism and APEX

\[ \text{ROT} \leftarrow \{ N \leftarrow \rho \omega \} \]

\[ m \leftarrow (0 = \iota N) \lor (N - 1) = \iota N \]

\[ (m \times \omega) + (\neg m) \times ((1 \phi \omega) + \neg 1 \phi \omega) \div 2.0 \}

\[ m = (0 == \text{iota}(N)) \lor ((N - 1) == \text{iota}(N)); \]

\[ \text{res} = (\text{tod}(m) \times v) + \text{tod}(!m) \times \]

\[ ((\text{rotate}([1], v) + \text{rotate}([-1], v)))/2.0; \]

- Rotate algorithm in APEX, generated SAC code

  - APL ROT 82.6s
  - Timings: SAC HAND 5.9s
  - APEX ROT 5.9s

Robert Bernecky, Sven-Bodo Scholz

Abstract Expressionism for Parallel Performance
Serial Relaxation using Abstract Expressionism and APEX

\[ \text{ROT} \leftarrow \{ \text{N} \leftarrow \rho \omega \} \]

\[ m \leftarrow (0 = \iota(N)) \lor (N-1) = \iota(N) \]
\[ (m \times \omega) + (\neg m) \times ((1 \Phi \omega) + (-1 \Phi \omega)) \div 2.0 \]

\[ m = (0 == \text{iota}(N)) \lor ((N-1) == \text{iota}(N)); \]
\[ \text{res} = (\text{tod}(m) \times v) + \text{tod}(!m) \times \]
\[ ((\text{rotate}([1], v) + \text{rotate}([-1], v)))/2.0; \]

- Rotate algorithm in APEX, generated SAC code

<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>APL ROT</td>
<td>82.6s</td>
</tr>
<tr>
<td>SAC HAND</td>
<td>5.9s</td>
</tr>
<tr>
<td>APEX ROT</td>
<td>5.9s</td>
</tr>
</tbody>
</table>

- **Identical** inner loops for APEX ROT and SAC HAND
Serial Relaxation using Abstract Expressionism and APEX

\[
\text{SHF} \leftarrow \{ N \leftarrow \rho \omega \\
\quad m \leftarrow (0 = \iota(N) \lor (N-1) = \iota N) \\\n\quad (m \times \omega) + (\neg m) \times ((1 \text{ shift } \omega) + \neg 1 \text{ shift } \omega) \div 2 \}
\]

\[
\text{shift} \leftarrow \{ ((\times \alpha) \times \rho \omega) \uparrow \alpha \downarrow \omega \}
\]

\[
m = (0 == \iota(N)) \lor ((N-1) == \iota(N));
\]

\[
\text{res} = (\text{tod}(m) \times v) + \text{tod}(\neg m) \times \\
((\text{shift}([1],v) + \text{shift}([-1],v))/2.0;
\]

- 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!
Serial Relaxation using Abstract Expressionism and APEX

SHF ← \{ N ← \rho \omega \\
            m ← (0 = \iota N) ∨ (N-1) = \iota N \\
            (m \times \omega) + (\neg m) \times ((1 \text{ shift } \omega) + ^{-1} \text{ shift } \omega) \div 2 \}

shift ← \{ ((x \alpha) \times \rho \omega) \uparrow \alpha \downarrow \omega \}

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

<table>
<thead>
<tr>
<th></th>
<th>APL TD</th>
<th>APL ROT</th>
<th>APL SHF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>82.6s</td>
<td>203.9s</td>
<td>236.9s</td>
</tr>
</tbody>
</table>

▶ Timings: SAC HAND 5.9s
APEX TD 5.9s
APEX ROT 5.9s
APEX SHIFT 5.9s
Serial Relaxation using Abstract Expressionism and APEX

\[ \text{SHF} \leftarrow \{ N \leftarrow \rho \omega \right. \]
\[ m \leftarrow (0 = \iota(N)) \lor (N-1) = \iota N \]
\[ (m \times \omega) + (\neg m) \times ((1 \text{ shift } \omega) + \neg 1 \text{ shift } \omega) \div 2 \}
\[ \text{shift} \leftarrow \{ ((\times \alpha) \times \rho \omega) \uparrow \alpha \downarrow \omega \} \]

\[ m = (0 == \text{iota}(N)) \lor ((N-1) == \text{iota}(N)); \]
\[ \text{res} = (\text{tod}(m) \times v) + \text{tod}(!m) \times \]
\[ ((\text{shift}([1],v) + \text{shift}([-1],v)))/2.0; \]

- **Shift algorithm in APEX-generated SAC code**
  - APL TD 82.6s
  - APL ROT 203.9s
  - APL SHF 236.9s

- **Timings:**
  - SAC HAND 5.9s
  - APEX TD 5.9s
  - APEX ROT 5.9s
  - APEX SHIFT 5.9s

> ALL inner loops are identical!

Robert Bernecky, Sven-Bodo Scholz

Abstract Expressionism for Parallel Performance
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:
Serial Performance GFLOPS

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

Serial Relaxation Performance (One FPU)

- Theoretical Peak Perf.
- SAC Hand
- APEX Rotate
- APEX Shift
- APEX TakeDrop
- SAC Cond
- C If/then/else
- APL TakeDrop
- APL Rotate
- APL Shift
Hard to do better? SAC/APEX approach maximum GFLOPS rate

Let’s look at parallel execution

Serial Relaxation Performance (One FPU)
Parallel Relaxation Speedup in C

- Open MP

Robert Bernecky, Sven-Bodo Scholz
Abstract Expressionism for Parallel Performance
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 \texttt{SOURCE} code
- See paper for ceremonial details
- Basic idea: Introduce pragmas into \texttt{SOURCE} code
  
  ```c
  #pragma omp parallel for
  ```
  after \textit{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)
Timings: (higher is better)
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 ="
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"
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: (higher is better)
Pessimized Parallel Relaxation in C

Timings: (higher is better)

Relaxation Performance

GFLOP/s

Number of threads

Robert Bernecky, Sven-Bodo Scholz

Abstract Expressionism for Parallel Performance
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

```c
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

Robert Bernecky, Sven-Bodo Scholz

Abstract Expressionism for Parallel Performance
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!)
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
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
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

Relaxation Performance

Number of threads

GFLOP/s

0 2 4 6 8 10 12 14 16

GFLOP/s

1 2 3 4 5 6 7 8

Theoretical Peak Perf.

shifts

hands

rotates

takedrops

handc

conds

ifs

condc

condstc

handstc

ifc

ifstc

Robert Bernecky, Sven-Bodo Scholz

Abstract Expressionism for Parallel Performance
Provide purely functional Intermediate Language (IL)

- Scalarize small arrays, e.g., in Gaussian Elimination pivot, replacing:
  \[
  \text{mat}[\text{rowa}, \text{rowb};] \rightarrow \text{mat}[\text{rowb}, \text{rowa};]
  \]

- ... 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!
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] û mat[rowb,rowa]
  
  by
  
  
  . . . 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!
Provide purely functional Intermediate Language (IL)
Preserve array semantics in IL
Scalarize small arrays, e.g.:

... 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!
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
Provide purely functional Intermediate Language (IL)
Preserve array semantics in IL
Scalarize small arrays, e.g.:
in Gaussian Elimination pivot, replacing:
\[
\text{mat}[\text{rowa},\text{rowb};] \leftarrow \text{mat}[\text{rowb},\text{rowa};]
\]
by
\[
\text{trow} \leftarrow \text{mat}[\text{rowa};] \quad \diamond \quad \text{mat}[\text{rowa};] \leftarrow \text{mat}[\text{rowb};] \quad \diamond
\text{mat}[\text{rowb};] \leftarrow \text{trow}
\]
... gives 2X speedup!
Provide purely functional Intermediate Language (IL)

Preserve array semantics in IL

Scalarize small arrays, e.g.:

- in Gaussian Elimination pivot, replacing:
  \[
  \text{mat}[\text{rowa,rowb}] \rightarrow \text{mat}[\text{rowb,rowa}]
  \]
  by
  \[
  \text{trow} \rightarrow \text{mat}[\text{rowa}] \oplus \text{mat}[\text{rowa}] \rightarrow \text{mat}[\text{rowb}] \oplus \text{mat}[\text{rowb}] \rightarrow \text{trow}
  \]

- ... gives 2X speedup!

- Do scalarization in the compiler, \textit{NOT} in app source code.
Provide purely functional Intermediate Language (IL)
Preserve array semantics in IL
Scalarize small arrays, e.g.:

in Gaussian Elimination pivot, replacing:
mat[\text{rowa}, \text{rowb};] \leftarrow \mat{\text{rowb}, \text{rowa};}

by
\text{trow} \leftarrow \mat{\text{rowa};} \diamond \mat{\text{rowa};} \leftarrow \mat{\text{rowb};} \diamond \mat{\text{rowb};} \leftarrow \text{trow}

... gives 2X speedup!

Do scalarization in the compiler, \textit{NOT} in app source code.

Use array-based optimizations, \textit{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:
    \[
    \text{mat[rowa, rowb;] } \leftrightarrow \text{mat[rowb, rowa;]}
    \]
    by
    \[
    \text{trow} \leftrightarrow \text{mat[rowa;] } \diamond \text{mat[rowa;] } \leftrightarrow \text{mat[rowb;] } \diamond \\
    \text{mat[rowb;] } \leftrightarrow \text{trow}
    \]
  - ...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...
Provide purely functional Intermediate Language (IL)

Preserve array semantics in IL

Scalarize small arrays, e.g.:

in Gaussian Elimination pivot, replacing:

\[
\text{mat[rowa,rowb;]} ← \text{mat[rowb,rowa;]}
\]

by

\[
\text{trow} ← \text{mat[rowa;]} ∧ \text{mat[rowa;]} ← \text{mat[rowb;]} ∧ \text{mat[rowb;]} ← \text{trow}
\]

... 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?