Single Assignment C
Research by Sven Bodo Schulz and Clemens Grelk

• *Single Assignment C: efficient support for high level array operations in a functional setting* [Scholz]

• *SAC – A Functional Array Language for Efficient Multi-threaded Execution* [Grelk and Scholz]

• *Accelerating APL Programs in SAC* [Grelk and Scholz]
SAC Language Overview

• N-dimensional arrays as first-class entities
  – powerful primitives for array-oriented programming
SAC Language Overview

- N-dimensional arrays as first-class entities
  - powerful primitives for array-oriented programming
- Purely functional semantics + static types
  - Useful for implicit parallelism and aggressive optimizations
SAC Language Overview

• N-dimensional arrays as first-class entities
  – powerful primitives for array-oriented programming

• Purely functional semantics + static types
  – Useful for implicit parallelism and aggressive optimizations

• Curly braces syntax
  – don’t scare the “white belt programmer”
A First Look At SAC

```c
int foo(int v, int w) {
    r = v + w;
    r = r + 1;
    return (r);
}
```

It doesn’t look functional!
A First Look At SAC

int foo(int v, int w) {
    r = v + w; // type of local variables inferred
    r = r + 1;
    return (r);
}

It doesn’t look functional!
A First Look At SAC

```c
int foo(int v, int w) { 
    r = v + w;   // type of local variables inferred
    r = r + 1;   // re-assignment treated as implicit let
    return (r);
}
```

It doesn’t look functional!
int foo(int v, int w) {
    r = v + w;       // type of local variables inferred
    r = r + 1;      // re-assignment treated as implicit let
    return (r);
}

let foo (v:int) (w:int) : int =
    let r = v + w in
    let r = r + 1 in r
Let’s generalize to arrays...

```c
int[*] foo(int[*] v, int[*] w) {
    r = v + w;
    r = r + 1;
    return (r);
}
```
Arrays in SAC

- Arrays can be scalars, vectors, matrices, etc...
- Arrays consist of two components:
  \[
  \begin{pmatrix}
  1 & 2 \\
  3 & 4
  \end{pmatrix}
  = \begin{bmatrix} 1,2,3,4 \end{bmatrix}, \begin{bmatrix} 2,2 \end{bmatrix}
  \]
- Specified using reshape(dims, data)

<table>
<thead>
<tr>
<th>Value</th>
<th>Shape</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[]</td>
<td>0</td>
</tr>
<tr>
<td>[1,2]</td>
<td>[2]</td>
<td>1</td>
</tr>
<tr>
<td>reshape([2,2],[1,2,3,4])</td>
<td>[2,2]</td>
<td>2</td>
</tr>
</tbody>
</table>
The Type System

• Primitive types of C
  – int, char, float, double, double, bool, etc...

• Hierarchy of array types
  
  \[ \text{bool}[1,2] \leq \text{bool}[.,.] \leq \text{bool}[+] \leq \text{bool}[\ast] \]
The Heart of Array Processing: with-loops

• We want a small core functionality for expressing shape-invariant array operations
  – *Keeps the language elegant, and helps the optimizing compiler*
The Heart of Array Processing: with-loops

• We want a small core functionality for expressing array operations
  – *Keeps the language elegant, and helps the optimizing compiler*

• SAC defines all array operations in terms of “with-loops” (sort of like an array comprehension)
genarray with-loop

Specifies a mapping from indices to values:

with

\[ \text{index sets : values} \]

genarray (shape)

with

\((\mathbb{0,0} \leq k < \mathbb{1,1}) : 7\)

default: 42

genarray([2,3])

\[ \begin{array}{ccc}
7 & 42 & 42 \\
42 & 42 & 42 \\
\end{array} \]
modarray with-loop

Duplicates old array’s values, with some specified modifications

with

\[
\begin{align*}
\text{index sets} : \text{values} \\
\text{modarray (array)}
\end{align*}
\]

\[
x = [1,2,3]; \\
\text{with} \\
\quad ([0] <= k < [2]): \ 7 \\
\text{modarray (x)}
\]

\[
\rightarrow \begin{bmatrix} 7 & 7 & 3 \end{bmatrix}
\]
fold with-loop

Reduces dimensionality of array.
Requires binary (associative, commutative) operation and a neutral element

with

  index sets : values
fold (op, neutral)

\[
x = [1,2,3]; \\
with \\
  ([0] <= k <= [2]): \ x[k] \\
fold(+, 0)
\]

\[\rightarrow 6\]
with-loop examples (I)

double[+] abs(double[+] a) {
    res = with
        (iv) : abs(a[iv])
    genarray(shape(a));
    return (res)
}

with-loop examples (II)

bool any(bool[*] a) {
    res = with
        (iv < shape(a)) : a[iv]
        fold(||, false);
    return (res)
}
Parallel Execution

• with-loops embody data parallelism (like array comprehensions in Data Parallel Haskell)
• Pool of worker threads (one thread per CPU)
• Array index space is partitioned between workers, and each worker executes identical code
• with-loops are combined by SAC compiler and unnecessary synchronization is optimized away