Data Parallel Array Programming in SAC

— Single Assignment C —

Clemens Grelck

University of Amsterdam
Informatics Institute
Computer Systems Architecture Group

ASCI Winter School 2010
Think Data Parallel!
Think Arrays!
or: everything is an array

- Vectors are arrays.
- Matrices are arrays.
- Tensors are arrays.
- ....... are arrays.
Think Data Parallel!
Think Arrays!
or: everything is an array

- Vectors are arrays.
- Matrices are arrays.
- Tensors are arrays.
- ........ are arrays.
- Operations map arrays to arrays.
- Even scalars are arrays.
- Even iteration spaces are arrays.
Implementations of Factorial

**Imperative:**

```c
int fac(int n)
{
    f = 1;
    while (n > 1) {
        f = f * n;
        n = n - 1;
    }
    return n;
}
```

**Functional:**

```c
fac n = if n <= 1
    then 1
    else n * fac(n - 1)
```
Implementations of Factorial

**Imperative:**

```c
int fac( int n)
{
    f = 1;
    while (n > 1) {
        f = f * n;
        n = n - 1;
    }
    return n;
}
```

**Functional:**

```haskell
fac n = if n <= 1
       then 1
       else n * fac (n - 1)
```

1 2 6 3628800

Data Parallel:

```haskell
fac n = prod ( 1 + iota ( n ));
```

1 2 6 3628800

Data Parallel Array Programming in SAC
Implementations of Factorial

**Imperative:**

```c
int fac( int n)
{
    f = 1;
    while (n > 1) {
        f = f * n;
        n = n - 1;
    }
    return n;
}
```

**Functional:**

```c
fac n = if n <= 1
    then 1
    else n * fac (n - 1)
```

**Data parallel:**

```c
fac n = prod( 1 + iota( n));
```

```
1 2 6 3628800
2 10
0 1 3 4 5 7 8 9
1 3 4 5 7 8 96
```

Clemens Grelck, University of Amsterdam

Data Parallel Array Programming in SAC
Implementations of Factorial

**Imperative:**

```c
int fac(int n) {
    f = 1;
    while (n > 1) {
        f = f * n;
        n = n - 1;
    }
    return n;
}
```

**Functional:**

```c
fac n = if n <= 1
    then 1
    else n * fac(n - 1)
```

**Data parallel:**

```c
fac n = prod(1 + iota(n));
```

Data Parallel Array Programming in SAC
Implementations of Factorial

**Imperative:**

```c
int fac(int n)
{
    f = 1;
    while (n > 1) {
        f = f * n;
        n = n - 1;
    }
    return n;
}
```

**Functional:**

```c
fac n = if n <= 1
    then 1
    else n * fac(n - 1)
```

**Data parallel:**

```c
fac n = prod(1 + iota(n));
```

---

map-reduce style!!

![Diagram of data parallel implementation]

Clemens Grelck, University of Amsterdam
Data Parallel Array Programming in SAC
The Essence of Data Parallel Programming

\[ \text{prod}(1 + \text{iota}(n)) \]
The Essence of Data Parallel Programming

\[ \text{prod}(1 + \text{iota}(n)) \]

Compilation to sequential code:

\[ 1 \]
\[ 2 \]
\[ 6 \]
\[ \vdots \]
\[ 3628800 \]
The Essence of Data Parallel Programming

prod(1+iota(n))

Compilation to sequential code

Compilation to microthreaded code

3628800
The Essence of Data Parallel Programming

prod(1+iota(n))

Compilation to sequential code

Compilation to multithreaded code

Compilation to microthreaded code

1 -> 0123456789 -> 012345678910 -> 30240 -> 3628800

2 -> 12345 -> 678910 -> 3628800

6 -> ...

3628800

0123456789

24 -> 1680 -> 151200

3628800

Clemens Grelck, University of Amsterdam

Data Parallel Array Programming in SAC
SAC — Design Space

SAC

Clemens Grelck, University of Amsterdam

Data Parallel Array Programming in SAC
High-level functional programming with vectors, matrices, arrays

SAC
SAC — Design Space

High-level functional programming with vectors, matrices, arrays

SAC

Suitability to achieve high performance in sequential and parallel execution
SAC — Design Space

High-level functional programming with vectors, matrices, arrays

SAC

Easy to adopt for programmers with an imperative background

Suitability to achieve high performance in sequential and parallel execution
SAC at a glance

C
- goto
- break
- continue
- global vars
- side-effects
- pointers

SAC
- bool
- multidimensional stateless arrays
- modules
- I/O system
- function overloading
- multiple return values
- values
- functions
- assignments
- conditionals
- loops
- operators
- double
- float
- int
- char

Clemens Grelck, University of Amsterdam
Data Parallel Array Programming in SAC
Introductory Example: gcd in SAC

```c
int gcd( int high, int low)
{
    if (high < low) {
        mem = low;
        low = high;
        high = mem;
    }
    while (low != 0) {
        quotient, remainder = diffmod( high, low);
        high = low;
        low = remainder;
    }
    return( high);
}
```

```c
int diffmod( int x, int y)
{
    quot = x / y;
    remain = x % y;
    return (quot, remain);
}
```

```c
int main ()
{
    return ( gcd ( 22 , 27));
}
```

Clemens Grelck, University of Amsterdam
Data Parallel Array Programming in SAC
Introductory Example: gcd in SAC

```c
int gcd(int high, int low)
{
    if (high < low) {
        mem = low;
        low = high;
        high = mem;
    }
    while (low != 0) {
        quotient, remainder = diffmod(high, low);
        high = low;
        low = remainder;
    }
    return(high);
}

int, int diffmod(int x, int y)
{
    quot = x / y;
    remain = x % y;
    return(quot, remain);
}

int main()
{
    return(gcd(22, 27));
}
```
What is Functional Programming?

**Execution Model:**

**Imperative programming:**
Sequence of instructions
that step-wise manipulate the program state

**Functional programming:**
Context-free substitution of expressions
until fixed point is reached
SAC:

{ 
  ...
  a = 5;
  b = 7;
  a = a + b;
  return (a);
}

Functional pseudo code:

... 
let a = 5 
in let b = 7 
in let a = a + b 
in a
Functional Semantics of SAC

**SAC:**

```c
int fac ( int n) {
    if (n >1) {
        r = fac (n -1);
        f = n * r;
    }
    else {
        f = 1;
    }
    return( f);
}
```

**Functional pseudo code:**

```plaintext
fun fac n =
    if n>1
    then let r = fac (n-1) in let f = n * r in f
    else let val f = 1 in f
```

Clemens Grelck, University of Amsterdam

Data Parallel Array Programming in SAC
The Role of Functions

Mathematics:
context-free mapping of argument values to result values
The Role of Functions

**Mathematics:**
context-free mapping of argument values to result values

**Imperative programming:**
subroutine with side-effects on global state
The Role of Functions

**Mathematics:**
context-free mapping of argument values to result values

**Imperative programming:**
subroutine with side-effects on global state

**Functional programming:**
context-free mapping of argument values to result values
The Role of Variables

Mathematics:

name/placeholder of a value
The Role of Variables

Mathematics:
name/placeholder of a value

Imperative programming:
name of a memory location
The Role of Variables

**Mathematics:**
name/placeholder of a value

**Imperative programming:**
name of a memory location

**Functional programming:**
name/placeholder of a value
The Role of Arrays

**Mathematics:**
functions from indices to values
The Role of Arrays

**Mathematics:**
functions from indices to values

**Imperative programming:**
contiguous fragments of addressable memory
The Role of Arrays

**Mathematics:**
functions from indices to values

**Imperative programming:**
contiguous fragments of addressable memory

**Functional programming in SAC:**
stateless multidimensional indexable collections of values
Why Stateless Arrays?

```c
int foo( int a)
{
    a = a + 1;
    return( a);
}

int bar( int a)
{
    a = 1;
    b = foo( a);
    print( a);
    print( b);
    return( 0);
}
```
Why Stateless Arrays?

```c
int foo( int a)
{
    a = a + 1;
    return( a);
}

int bar( int a)
{
    a = 1;
    b = foo( a);
    print( a);          "1"
    print( b);          "2"
    return( 0);
}
```

Side-effects are error-prone!!

Arrays must be passed call-by-value!!
Why Stateless Arrays?

```c
{
    return( a);
}

{
    a = [1,2,3,4,5] ;
    b = foo( a);
    print( a);
    print( b);
    return( 0);
}
```

Side-effects are error-prone!!
Arrays must be passed call-by-value!!

Clemens Grelck, University of Amsterdam
Data Parallel Array Programming in SAC
Why Stateless Arrays?

```c
{
    return( a);
}

{
    a = [1,2,3,4,5] ;
    b = foo( a);
    print( a);
    print( b);
    return( 0);
}
```

Side-effects are error-prone!!
Why Stateless Arrays?

```c
{
    return( a);
}

{
    a = [1,2,3,4,5] ;
    b = foo( a);
    print( a);
    print( b);
    return( 0);
}
```

Side-effects are error-prone!!

Arrays must be passed call-by-value!!
Multidimensional Arrays in SAC

\[
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{bmatrix}
\]
dim: 2
shape: [3,3]
data: [1,2,3,4,5,6,7,8,9]
Multidimensional Arrays in SAC

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

- **dim:** 2
- **shape:** [3,3]
- **data:** [1,2,3,4,5,6,7,8,9]

- **dim:** 3
- **shape:** [2,2,3]
- **data:** [1,2,3,4,5,6,7,8,9,10,11,12]

Clemens Grelck, University of Amsterdam
Data Parallel Array Programming in SAC
Multidimensional Arrays in SAC

\[
\begin{pmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{pmatrix}
\]

\text{dim: 2} \\
\text{shape: [3,3]} \\
\text{data: [1,2,3,4,5,6,7,8,9]}

\text{dim: 3} \\
\text{shape: [2,2,3]} \\
\text{data: [1,2,3,4,5,6,7,8,9,10,11,12]}

\text{dim: 1} \\
\text{shape: [6]} \\
\text{data: [1,2,3,4,5,6]}
Multidimensional Arrays in SAC

\[
\begin{pmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9 \\
\end{pmatrix}
\]

\begin{itemize}
\item dim: 2
\item shape: [3,3]
\item data: [1,2,3,4,5,6,7,8,9]
\end{itemize}

\[
\begin{pmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9 \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
10 & 11 \\
12 \\
\end{pmatrix}
\]

\begin{itemize}
\item dim: 3
\item shape: [2,2,3]
\item data: [1,2,3,4,5,6,7,8,9,10,11,12]
\end{itemize}

\[
[1,2,3,4,5,6]
\]

\begin{itemize}
\item dim: 1
\item shape: [6]
\item data: [1,2,3,4,5,6]
\end{itemize}

\[
42
\]

\begin{itemize}
\item dim: 0
\item shape: []
\item data: [42]
\end{itemize}
With-Loops: Genarray Variant

\[
A = \text{with } \{
    ([1,1] \leq iv < [3,4]) : e(iv);
\} : \text{genarray}([4,5], \text{def});
\]

\[
A = \begin{pmatrix}
    \text{def} & \text{def} & \text{def} & \text{def} & \text{def} \\
    \text{def} & e([1,1]) & e([1,2]) & e([1,3]) & \text{def} \\
    \text{def} & e([2,1]) & e([2,2]) & e([2,3]) & \text{def} \\
    \text{def} & \text{def} & \text{def} & \text{def} & \text{def}
\end{pmatrix}
\]
With-Loops: Modarray Variant

\[
A = \text{with} \{
\quad ([1,1] <= iv < [3,4]) : \ e(iv);
\} : \ \text{modarray}( B );
\]

\[
A = \begin{pmatrix}
B[[0,0]] & B[[0,1]] & B[[0,2]] & B[[0,3]] & B[[0,4]] \\
B[[1,0]] & e([1,1]) & e([1,2]) & e([1,3]) & B[[1,4]] \\
B[[2,0]] & e([2,1]) & e([2,2]) & e([2,3]) & B[[2,4]] \\
B[[3,0]] & B[[3,1]] & B[[3,2]] & B[[3,3]] & B[[3,4]]
\end{pmatrix}
\]
With-Loops: Fold Variant

\[
A = \text{with} \begin{cases}
([1,1] \leq iv < [3,4]): e(iv); \\
\end{cases}
\text{fold}(\oplus, \text{neutr});
\]

\[
A = \text{neutr} \oplus e([1,1]) \oplus e([1,2]) \oplus e([1,3]) \\
\oplus e([2,1]) \oplus e([2,2]) \oplus e([2,3])
\]

(\oplus\text{ denotes associative, commutative binary function.})
Grid Generators

\[ A = \text{with} \{ \\
( [2,1] \leq iv < [8,11] \text{ step } [2,3]):\ \\
e(\text{iv}); \\
\} : \text{genarray}([10,13], \text{def}); \]
Grid Width Generators

\[ A = \text{with} \{ \]
\[ ([2,1] \leq iv < [8,11] \text{ step } [3,4] \text{ width } [2,3]): \]
\[ e(iv); \]
\[ \}: \text{genarray( [10,13], \text{ def } );} \]
Element-wise subtraction of arrays:

```sac
{
    res = with {
        ([0,0] <= iv < [20,20]) : A[iv] - B[iv];
    }: genarray( [20,20], 0);
    return( res);
}
```
Principle of Abstraction

\[
\text{int}[20,20] \ (-) (\text{int}[20,20] \ A, \ \text{int}[20,20] \ B) \\
\{
\begin{align*}
\text{res} &= \text{with} \ {}
\begin{align*}
(0,0) &\leq iv < (20,20) : A[iv] - B[iv];
\end{align*}
\}\begin{align*}
&\text{genarray}([20,20] \ , \ 0); \\
&\text{return} (\ \text{res});
\end{align*}
\}
\]

Shape-generic code

\[
\text{int}[.,.] \ (-) (\text{int}[.,.] \ A, \ \text{int}[.,.] \ B) \\
\{
\begin{align*}
\text{shp} &= \text{min} (\ \text{shape}(A), \ \text{shape}(B)); \\
\text{res} &= \text{with} \ {}
\begin{align*}
(0,0) &\leq iv < \text{shp} : A[iv] - B[iv];
\end{align*}
\}\begin{align*}
&\text{genarray}(\ \text{shp}, \ 0); \\
&\text{return} (\ \text{res});
\end{align*}
\}
\]
Principle of Abstraction

int[. , .] (-) (int[. , .] A, int[. , .] B)
{
    shp = min( shape(A), shape(B) );
    res = with {
        ([0,0] <= iv < shp) : A[iv] - B[iv];
    }: genarray( shp, 0);
    return( res);
}

Rank-generic code

int[*] (-) (int[*] A, int[*] B)
{
    shp = min( shape(A), shape(B) );
    res = with {
        (0*shp <= iv < shp) : A[iv] - B[iv];
    }: genarray( shp, 0);
    return( res);
}
A Hierarchy of Array Types with Subtyping

```
int[*]
  /
  int[+]
/    /
int = int[] int[.]

int[0] ... int[42] ...
int[0,0] ... int[3,7] ...
```

Clemens Grelck, University of Amsterdam
Data Parallel Array Programming in SAC
Principle of Composition

Characteristics:

- Step-wise composition of functions
- from previously defined functions
- or basic building blocks (with-loop defined)

Example: convergence test

```c
bool is_convergent (double [*] new, double [*] old, double eps)
{
    return ( all ( abs ( new - old ) < eps ));
}
```
Principle of Composition

Example: convergence test

```c
bool is_convergent (double [*] new, double [*] old, double eps)
{
    return ( all ( abs ( new - old ) < eps ));
}
```

Advantages:

- Rapid prototyping
- High confidence in correctness
- Good readability of code
Execution through Context-Free Substitution

**Convergence Test:**

\[
\text{is\_convergent( } [1,2,3,8], [3,2,1,4], 3 \text{ )}
\]
Execution through Context-Free Substitution

**Convergence Test:**

```python
is_convergent([1,2,3,8], [3,2,1,4], 3 )
```

Downward arrow:

```python
all( abs([1,2,3,8] - [3,2,1,4]) < 3 )
```
Execution through Context-Free Substitution

Convergence Test:

\[
\text{is\_convergent}( [1,2,3,8], [3,2,1,4], 3 )
\]

\[
\text{all}( \text{abs}( [1,2,3,8] - [3,2,1,4]) < 3 )
\]

\[
\text{all}( \text{abs}( [-2,0,2,4]) < 3 )
\]
Execution through Context-Free Substitution

**Convergence Test:**

```
is_convergent([1,2,3,8], [3,2,1,4], 3)
```

```
all(abs([1,2,3,8] - [3,2,1,4]) < 3)
```

```
all(abs([-2,0,2,4]) < 3)
```

```
all([2,0,2,4] < 3)
```
Execution through Context-Free Substitution

**Convergence Test:**

```plaintext
is_convergent( [1,2,3,8], [3,2,1,4], 3 )
```

```
all( abs( [1,2,3,8] - [3,2,1,4]) < 3 )
```

```
all( abs( [-2,0,2,4]) < 3 )
```

```
all( [2,0,2,4] < 3 )
```

```
all( [true, true, true, false])
```
Execution through Context-Free Substitution

Convergence Test:

\[
is_{\text{convergent}}( [1,2,3,8], [3,2,1,4], 3 )
\]

\[
\text{all}( \text{abs}( [1,2,3,8] - [3,2,1,4]) < 3 )
\]

\[
\text{all}( \text{abs}( [-2,0,2,4]) < 3 )
\]

\[
\text{all}( [2,0,2,4] < 3 )
\]

\[
\text{all}( [\text{true, true, true, false}])
\]

false
Shape-Generic Programming

2-dimensional convergence test:

\[
\text{is\_convergent}( \begin{pmatrix} 1 & 2 \\ 3 & 8 \end{pmatrix}, \begin{pmatrix} 3 & 2 \\ 1 & 7 \end{pmatrix}, 3 )
\]
Shape-Generic Programming

2-dimensional convergence test:

\[ \text{is\_convergent}( \begin{pmatrix} 1 & 2 \\ 3 & 8 \end{pmatrix}, \begin{pmatrix} 3 & 2 \\ 1 & 7 \end{pmatrix}, 3 ) \]

3-dimensional convergence test:

\[ \text{is\_convergent}( \begin{pmatrix} \begin{pmatrix} 1 & 2 \\ 3 & 8 \end{pmatrix} \\ \begin{pmatrix} 6 & 7 \\ 2 & 8 \end{pmatrix} \end{pmatrix}, \begin{pmatrix} \begin{pmatrix} 2 & 1 \\ 0 & 8 \end{pmatrix} \\ \begin{pmatrix} 1 & 1 \\ 3 & 7 \end{pmatrix} \end{pmatrix}, 3 ) \]
The Power of With-Loops

▶ NO large collection of built-in operations
  ▶ Simplified compiler design
The Power of With-Loops

- NO large collection of built-in operations
  - Simplified compiler design
- INSTEAD: library of array operations
  - Improved maintainability
  - Improved extensibility
The Power of With-Loops

- NO large collection of built-in operations
  - Simplified compiler design
- INSTEAD: library of array operations
  - Improved maintainability
  - Improved extensibility
- General intermediate representation for array operations
  - Basis for code optimization
  - Basis for implicit parallelization
Case Study: Convolution

Algorithmic principle:

Compute weighted sums of neighbouring elements
Case Study: Convolution

Algorithmic principle:
Compute weighted sums of neighbouring elements

Fixed boundary conditions (1-dimensional):
Case Study: Convolution

Algorithmic principle:

Compute weighted sums of neighbouring elements

Fixed boundary conditions (1-dimensional):

Periodic boundary conditions (1-dimensional):
Case Study: Convolution

Problem:

- 9 different situations in 2-dimensional grids
- 27 different situations in 3-dimensional grids
- ...
Convolution Step in SaC

1-dimensional:

double[.] convolution_step (double[.] A)
{
    R = A + rotate( 1, A) + rotate( -1, A);
    return ( R / 3.0);
}
Convolution Step in SaC

1-dimensional:

double[.] convolution_step (double[.] A)
{
    R = A + rotate(1, A) + rotate(-1, A);
    return (R / 3.0);
}

N-dimensional:

double[*] convolution_step (double[*] A)
{
    R = A;
    for (i=0; i<dim(A); i++) {
        R = rotate(i, 1, R) + rotate(i, -1, R);
    }
    return (R / tod(2 * dim(A) + 1));
}
Fixed number of iterations:

```c
double [*] convolution (double [*] A, int iter)
{
    for (i=0; i<iter; i++) {
        A = convolution_step( A);
    }

    return( A);
}
```
Convolution in SaC

Variable number of iterations with convergence check:

```c
double[*] convolution (double[*] A, double eps)
{
    do {
        A_old = A;
        A = convolution_step( A_old);
    } while (!is_convergent( A, A_old, eps));

    return( A);
}
```
Convolution in SaC

Variable number of iterations with convergence test:

```c
double[*] convolution (double[*] A, double eps) {
    do {
        A_old = A;
        A = convolution_step( A_old);
    } while (!is_convergent( A, A_old, eps));

    return( A);
}
```

Convergence criterion:

```c
bool
is_convergent (double[*] new, double[*] old, double eps) {
    return( any( abs( new - old) >= eps));
}
```
Compilation Challenge

SAC
Functional Array Programming

SAC2C
Advanced Compiler Technology

Amsterdam MicroGrid Architecture
Symmetric Multicore Processors
Systems on a Chip
Manycore GPGPU Boards
Compilation Challenges

- **Challenge 1: Stateless Arrays**
  - How to avoid copying?
  - How to avoid boxing small arrays?
  - How to do memory management efficiently?
Compilation Challenges

- **Challenge 1: Stateless Arrays**
  - How to avoid copying?
  - How to avoid boxing small arrays?
  - How to do memory management efficiently?

- **Challenge 2: Compositional Specifications**
  - How to avoid temporary arrays?
  - How to avoid multiple array traversals?
Compilation Challenges

- **Challenge 1: Stateless Arrays**
  - How to avoid copying?
  - How to avoid boxing small arrays?
  - How to do memory management efficiently?

- **Challenge 2: Compositional Specifications**
  - How to avoid temporary arrays?
  - How to avoid multiple array traversals?

- **Challenge 3: Shape-Invariant Specifications**
  - How to generate efficient loop nestings?
  - How to represent arrays with different static knowledge?
Compilation Challenges

- **Challenge 1: Stateless Arrays**
  - How to avoid copying?
  - How to avoid boxing small arrays?
  - How to do memory management efficiently?

- **Challenge 2: Compositional Specifications**
  - How to avoid temporary arrays?
  - How to avoid multiple array traversals?

- **Challenge 3: Shape-Invariant Specifications**
  - How to generate efficient loop nestings?
  - How to represent arrays with different static knowledge?

- **Challenge 4: Efficient Organisation of Concurrent Execution**
  - How to schedule index spaces to threads?
  - When to synchronise (and when not)?
Challenge 5: Implementing a Fully-Fledged Compiler

Scanner / Parser

Function Inlining
- Array Elimination
- Dead Code Removal
- Common Subexpression Elimination
- Constant Propagation
- Constant Folding
- Copy Propagation
- Algebraic Simplification
- Loop Unrolling
- Memory Reuse
- Loop Invariant Removal
- With-Loop Unrolling
- With-Loop Invariant Removal
- With-Loop Folding
- With-Loop Scalarisation
- With-Loop Fusion
- Automatic Array Padding
- Index Vector Elimination

Type Inference
- Type Specialisation

High-Level Optimisations

Memory Management

De-Functionalisation

Parallelisation

Code Generation
sac2c is a large-scale compilation technology project

- SAC compiler + runtime library:
  - 275,000 lines of code
  - about 1000 files
  - + standard prelude
  - + standard library
sac2c is a large-scale compilation technology project

- **SAC** compiler + runtime library:
  - 275,000 lines of code
  - about 1000 files
  - + standard prelude
  - + standard library

- More than 10 years of research and development
sac2c is a large-scale compilation technology project

- **SAC** compiler + runtime library:
  - 275,000 lines of code
  - about 1000 files
  - + standard prelude
  - + standard library

- More than 10 years of research and development

- Approaching one hundred man years of investment
sac2c is a large-scale compilation technology project

- SAC compiler + runtime library:
  - 275,000 lines of code
  - about 1000 files
  - + standard prelude
  - + standard library

- More than 10 years of research and development
- Approaching one hundred man years of investment
- Complete compiler construction infrastructure
The SAC Project

International partners:

- University of Kiel, Germany (1994–2005)
- University of Toronto, Canada (since 2000)
- University of Lübeck, Germany (2001–2008)
- University of Hertfordshire, England (since 2003)
- University of Amsterdam, Netherlands (since 2008)
Runtime Performance: Standard Multiprocessor

NAS benchmark FT
Runtime Performance: Standard Multiprocessor

**NAS benchmark FT**

- Green dots: vs seq SAC
- Red dots: vs seq Fortran77

**NAS benchmark MG**

- Green dots: vs seq SAC
- Red dots: vs seq Fortran77
Preliminary Performance Results: CUDA

PDE1 benchmark on NVidia 8800GT:

PDE1 CUDA vs. SaC Speedups (8800GT)
Preliminary Performance Results: CUDA

PDE1 benchmark on NVidia Tesla:

![Graph showing PDE1 CUDA vs. SaC Speedups (Tesla) with 10, 50, 100, and 200 steps.]
Preliminary Performance Results: CUDA

Lattice-Boltzmann on NVidia 8800GT:

LatticeBoltzmann CUDA vs. SaC Speedups (8800GT)
Preliminary Performance Results: CUDA

Lattice-Boltzmann on NVidia Tesla:

---

Clemens Grelck, University of Amsterdam  
Data Parallel Array Programming in SAC
Summary

Language design:

- High-level array processing
- Functional state-less semantics but C-like syntax
- Abstraction and composition
- Shape-generic programming
- (Almost) index-free programming
Summary

Language design:
- High-level array processing
- Functional state-less semantics but C-like syntax
- Abstraction and composition
- Shape-generic programming
- (Almost) index-free programming

Language implementation:
- Fully-fledged compiler
- Automatic parallelisation
- Automatic memory management
- High-level program transformation
- Large-scale machine-independent optimisation
The End

Questions ?

Check out www.sac-home.org !!