Declarative Array Programming with SAC — Single Assignment C

Clemens Grelck

University of Amsterdam
Informatics Institute
Computer Systems Architecture Group

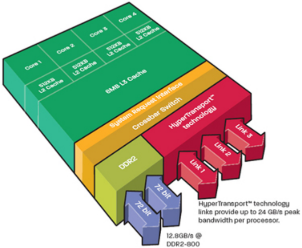
ASCI Course A24

A Programmer's Guide for
Modern High-Performance Computing Architectures
Delft, November 29, 2012

Single Assignment C: Outline

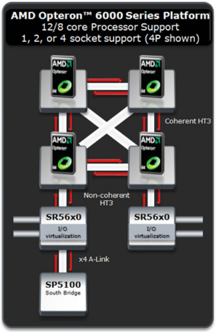
The Free Lunch is Over: Many-Core to the Rescue

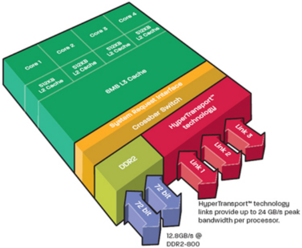
The many-core hardware zoo:



The Free Lunch is Over: Many-Core to the Rescue

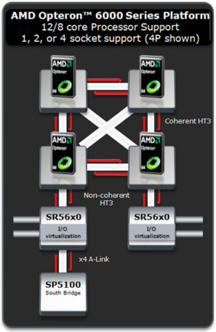
The many-core hardware zoo:

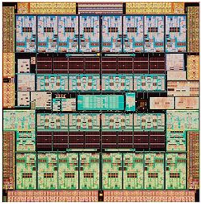


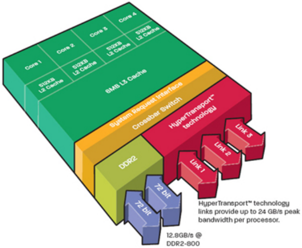


The Free Lunch is Over: Many-Core to the Rescue

The many-core hardware zoo:

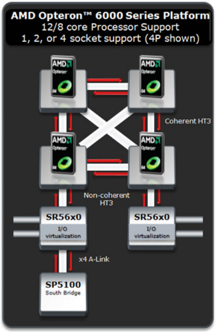


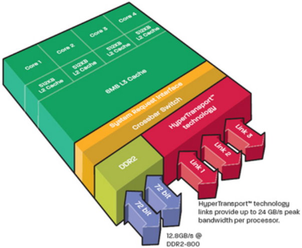


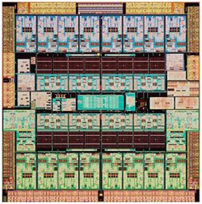


The Free Lunch is Over: Many-Core to the Rescue

The many-core hardware zoo:

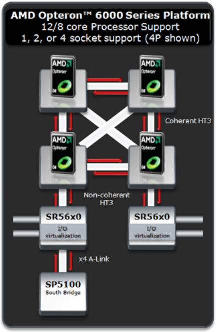


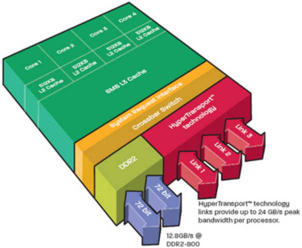


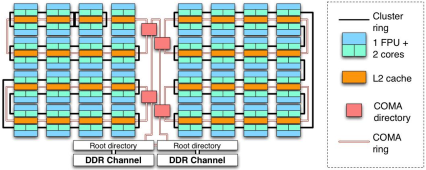


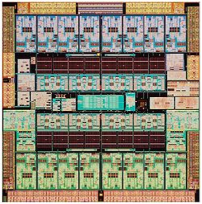
The Free Lunch is Over: Many-Core to the Rescue

The many-core hardware zoo:









Design Rationale of SAC

Hardware in the many-core era is a zoo:

- Vastly different numbers of cores
- Vastly different core architectures: power, genericity
- Vastly different memory architectures

Design Rationale of SAC

Hardware in the many-core era is a zoo:

- Vastly different numbers of cores
- Vastly different core architectures: power, genericity
- Vastly different memory architectures

Programming diverse hardware is uneconomic:

- Diverse low-level programming models
- Each requires expert knowledge
- Heterogeneous combinations of the above ?

Design Rationale of SAC

Genericity through abstraction:

- Program what to compute, not exactly how
- Leave execution organisation to compiler and runtime system
- Put expert knowledge into compiler, not into applications

Design Rationale of SAC

Genericity through abstraction:

- Program what to compute, not exactly how
- Leave execution organisation to compiler and runtime system
- Put expert knowledge into compiler, not into applications
- Let programs remain architecture-agnostic
- Compile one source to diverse target hardware
- Pursue data-parallel approach to implicitly promote concurrency

What Does Data Parallel Really Mean ?

Factorial imperative:

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

Factorial functional:

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

What Does Data Parallel Really Mean ?

Factorial imperative:

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

Factorial functional:

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

n: 10

f: 1

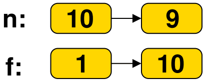
What Does Data Parallel Really Mean ?

Factorial imperative:

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

Factorial functional:

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



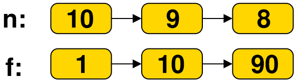
What Does Data Parallel Really Mean ?

Factorial imperative:

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

Factorial functional:

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



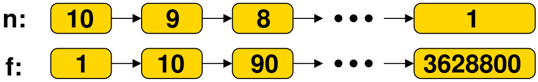
What Does Data Parallel Really Mean ?

Factorial imperative:

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

Factorial functional:

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



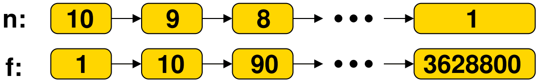
What Does Data Parallel Really Mean ?

Factorial imperative:

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

Factorial functional:

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



Data parallel:

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

What Does Data Parallel Really Mean ?

Factorial imperative:

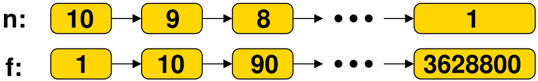
```
int fac( int n)
{
  f = 1;
  while (n > 1) {
    f = f * n;
    n = n - 1;
  }
  return f;
}
```

Data parallel:

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

Factorial functional:

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



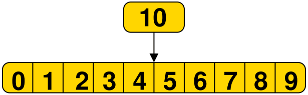
What Does Data Parallel Really Mean ?

Factorial imperative:

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

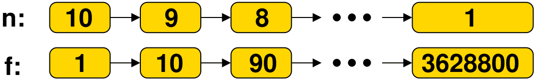
Data parallel:

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



Factorial functional:

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



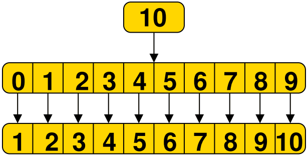
What Does Data Parallel Really Mean ?

Factorial imperative:

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

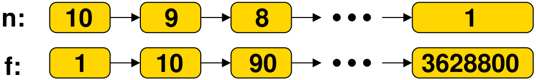
Data parallel:

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



Factorial functional:

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



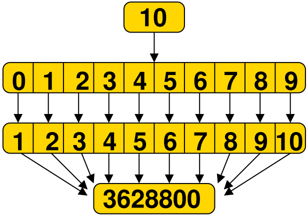
What Does Data Parallel Really Mean ?

Factorial imperative:

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

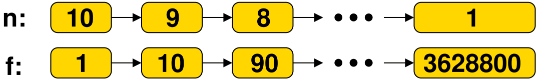
Data parallel:

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



Factorial functional:

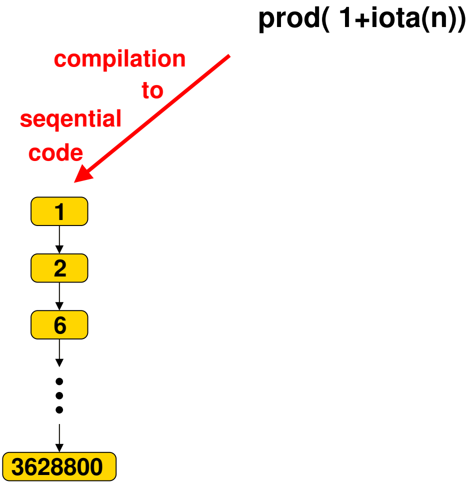
```
fac n = if n <= 1
        then 1
        else n * fac (n - 1)
```



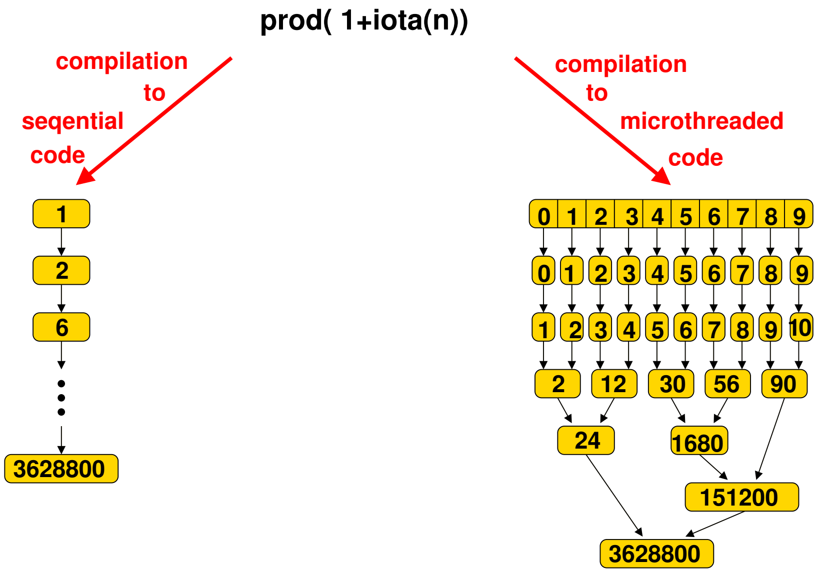
The Essence of Data Parallel Programming

prod(1+iota(n))

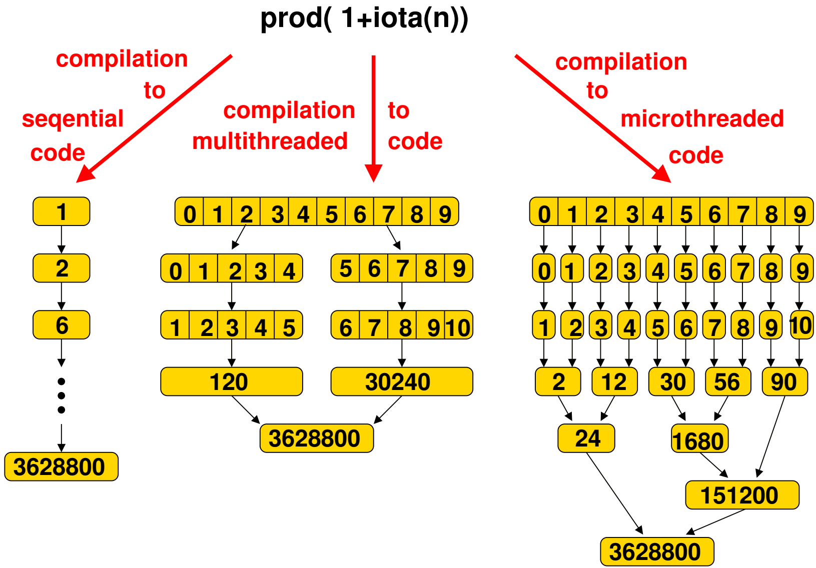
The Essence of Data Parallel Programming



The Essence of Data Parallel Programming



The Essence of Data Parallel Programming

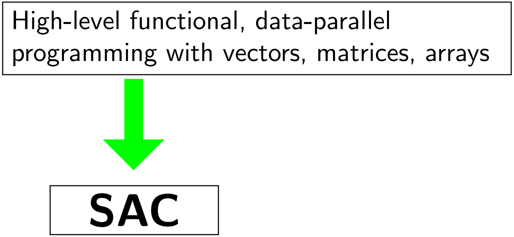


Single Assignment C: Outline

SAC — Design Space

SAC

SAC — Design Space



SAC — Design Space

High-level functional, data-parallel
programming with vectors, matrices, arrays

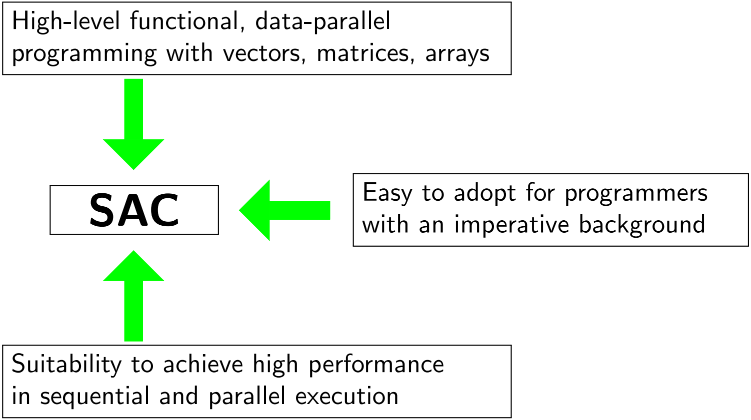
↓

SAC

↑

Suitability to achieve high performance
in sequential and parallel execution

SAC — Design Space



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

Functional Semantics of SAC

SAC: → **Functional pseudo code:**

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

```
...
let a = 5
in let b = 7
in let a = a + b
in a
```

Functional Semantics of SAC

SAC: ➡ **Functional pseudo code:**

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

```
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
```

Functional Semantics of SAC

SAC:

```
int fac( int n)
{
  f = 1;

  while (n>1) {
    f = f * n;
    n = n - 1;
  }

  return( f);
}
```

Functional pseudo code:

```
fun fac n =
  let rec fac_while f n =
    if n>1
    then let f = f * n
         in let n = n - 1
         in fac_while f n
    else f
  in
    fac_while 1 n
```

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 in SAC:
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 in SAC:
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

Single Assignment C: Outline

Multidimensional Arrays in SAC

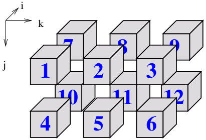
$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

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

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]

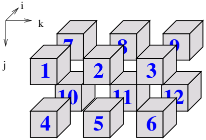


dim: 3
shape: [2,2,3]
data: [1,2,3,4,5,6,7,8,9,10,11,12]

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]

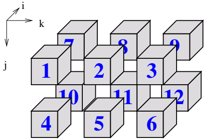
[1, 2, 3, 4, 5, 6]

dim: 1
shape: [6]
data: [1,2,3,4,5,6]

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]

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

dim: 1
shape: [6]
data: [1,2,3,4,5,6]

42

dim: 0
shape: [ ]
data: [42]

Built-in Array Operations

- Defining a vector:

```
vec = [1,2,3,4,5,6];
```

Built-in Array Operations

- Defining a vector:

```
vec = [1,2,3,4,5,6];
```

- Defining a higher-dimensional array:

```
mat = [vec,vec];
mat = reshape( [3,2], vec);
```

Built-in Array Operations

- Defining a vector:
```
vec = [1,2,3,4,5,6];
```
- Defining a higher-dimensional array:
```
mat = [vec,vec];
mat = reshape( [3,2], vec);
```
- Querying for the shape of an array:
```
shp = shape( mat);
```
→ [3,2]

Built-in Array Operations

- Defining a vector:
  ```
  vec = [1,2,3,4,5,6];
  ```
- Defining a higher-dimensional array:
  ```
  mat = [vec,vec];
  mat = reshape( [3,2], vec);
  ```
- Querying for the shape of an array:
  ```
  shp = shape( mat);
  ```
  → [3,2]
- Querying for the rank of an array:
  ```
  rank = dim( mat);
  ```
  → 2

Built-in Array Operations

- Defining a vector:
  ```
  vec = [1,2,3,4,5,6];
  ```
- Defining a higher-dimensional array:
  ```
  mat = [vec,vec];
  mat = reshape( [3,2], vec);
  ```
- Querying for the shape of an array:
  ```
  shp = shape( mat);    ➡ [3,2]
  ```
- Querying for the rank of an array:
  ```
  rank = dim( mat);    ➡ 2
  ```
- Selecting elements:
  ```
  x = sel( [4], vec);    ➡ 5
  y = sel( [2,1], mat);    ➡ 6
  x = vec[[4]];    ➡ 5
  y = mat[[2,1]];    ➡ 6
  ```

With-Loops: Versatile Array Comprehensions

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

- Multidimensional array comprehensions
- Mapping from index domain into value domain

[0,0]	[0,1]	[0,2]	[0,3]
[1,0]	[1,1]	[1,2]	[1,3]
[2,0]	[2,1]	[2,2]	[2,3]
[3,0]	[3,1]	[3,2]	[3,3]
[4,0]	[4,1]	[4,2]	[4,3]

def	def	def	def
def	e([1,1])	e([1,2])	e([1,3])
def	e([2,1])	e([2,2])	e([2,3])
def	e([3,1])	e([3,2])	e([3,3])
def	def	def	def

index domain **value domain**

With-Loops: Modarray Variant

```
A  =  with {
        ([1,1] <= iv < [3,4]) :  e(iv);
      }:  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   =   with {
          ([1,1] <= iv < [3,4]) :   e(iv);
        }:  fold( ⊕, neutr );
```

$$A = neutr \oplus e([1,1]) \oplus e([1,2]) \oplus e([1,3]) \oplus e([2,1]) \oplus e([2,2]) \oplus e([2,3])$$

($\oplus$ denotes associative, commutative binary function.)

Single Assignment C: Outline

Principle of Abstraction

Element-wise subtraction of arrays:

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

Principle of Abstraction

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

Shape-generic code

```
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);
}
```

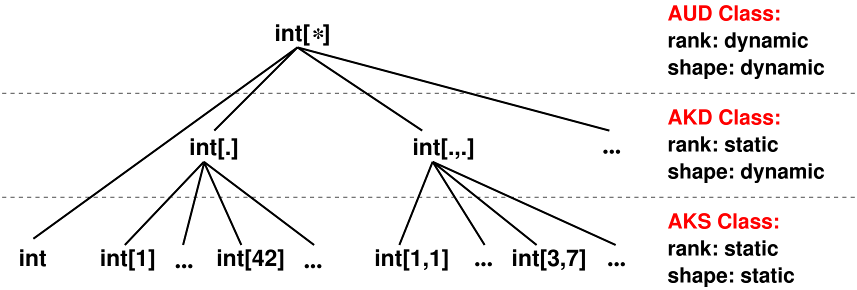
Principle of Abstraction

```
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);
}
```

Shapely Array Type Hierarchy With Subtyping



AUD	:	Array of Unknown Dimension
AKD	:	Array of Known Dimension
AKS	:	Array of Known Shape

Function Overloading

Example:

```
int[20,20] (-) (int[20,20] A, int[20,20] B) {...}

int[.,.]   (-) (int[.,.] A,   int[.,.] B) {...}

int[*]     (-) (int[*] A,     int[*] B) {...}
```

Features:

- Multiple function definitions with same name, but
  - different numbers of arguments
  - different base types
  - different shapely types
- No restriction on function semantics
- Argument subtyping must be monotonous
- Dynamic function dispatch

Principle of Composition

Characteristics:

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

Example: convergence test

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

Principle of Composition

Example: convergence test

```
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:

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

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 )
```

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:

```
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_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])
        ⇓
false
```

Shape-Generic Programming

2-dimensional convergence test:

$$\texttt{is\_convergent(} \begin{pmatrix} 1 & 2 \\ 3 & 8 \end{pmatrix}, \begin{pmatrix} 3 & 2 \\ 1 & 7 \end{pmatrix}, 3 \texttt{)}$$

Shape-Generic Programming

2-dimensional convergence test:

$$\texttt{is\_convergent(} \begin{pmatrix} 1 & 2 \\ 3 & 8 \end{pmatrix}, \begin{pmatrix} 3 & 2 \\ 1 & 7 \end{pmatrix}, \texttt{3)}$$

3-dimensional convergence test:

$$\texttt{is\_convergent(} \left(\begin{matrix} \begin{pmatrix} 1 & 2 \\ 3 & 8 \end{pmatrix} \\ \begin{pmatrix} 6 & 7 \\ 2 & 8 \end{pmatrix} \end{matrix} \right), \left(\begin{matrix} \begin{pmatrix} 2 & 1 \\ 0 & 8 \end{pmatrix} \\ \begin{pmatrix} 1 & 1 \\ 3 & 7 \end{pmatrix} \end{matrix} \right), \texttt{3)}$$

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
- Composition of building blocks
  - Rapid prototyping
  - High confidence in correctness
  - Good readability of code

The Power of With-Loops

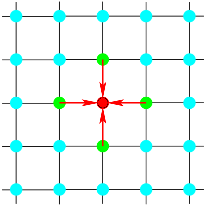
- NO large collection of built-in operations
  - Simplified compiler design
- INSTEAD: **library of array operations**
  - Improved maintainability
  - Improved extensibility
- Composition of building blocks
  - Rapid prototyping
  - High confidence in correctness
  - Good readability of code
- General intermediate representation for array operations
  - Basis for code optimization
  - Basis for implicit parallelization

Single Assignment C: Outline

Case Study: Convolution

Algorithmic principle:

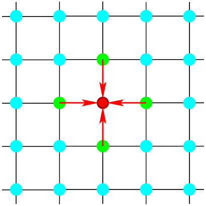
Compute weighted sums
of neighbouring elements



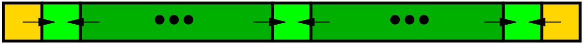
Case Study: Convolution

Algorithmic principle:

Compute weighted sums
of neighbouring elements



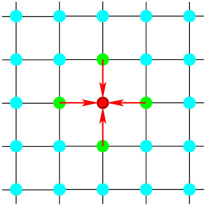
Fixed boundary conditions (1-dimensional):



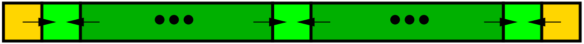
Case Study: Convolution

Algorithmic principle:

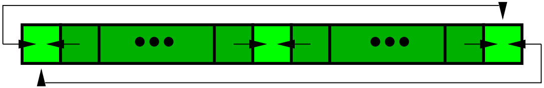
Compute weighted sums of neighbouring elements



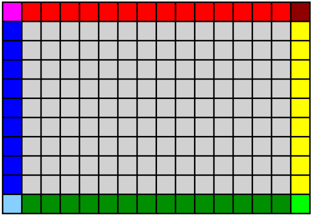
Fixed boundary conditions (1-dimensional):



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 = R + rotate( i, 1, A) + rotate( i, -1, A);
  }
  return( R / tod( 2 * dim(A) + 1));
}
```

Convolution in SaC

Fixed number of iterations:

```
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:

```
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:

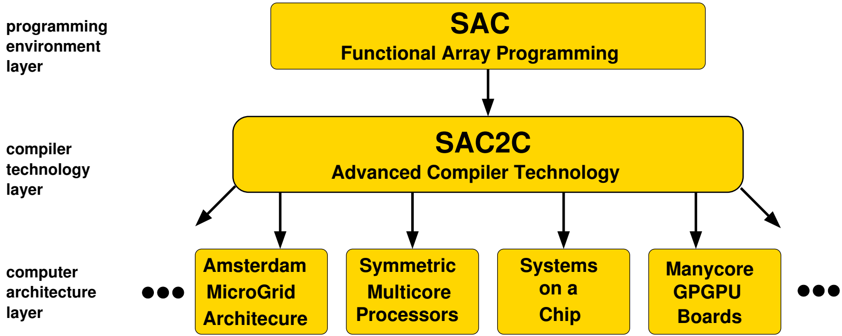
```
double[*] convolution (double[*] A, double eps)
{
  do {
    A_old = A;
    A = convolution_step( A_old);
  }
  while (!is_convergent( A, A_old, eps));

  return( A);
}
```

Convergence criterion:

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

Single Assignment C: Outline

Compilation Challenge



Some Compilation Challenges

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

Some 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?

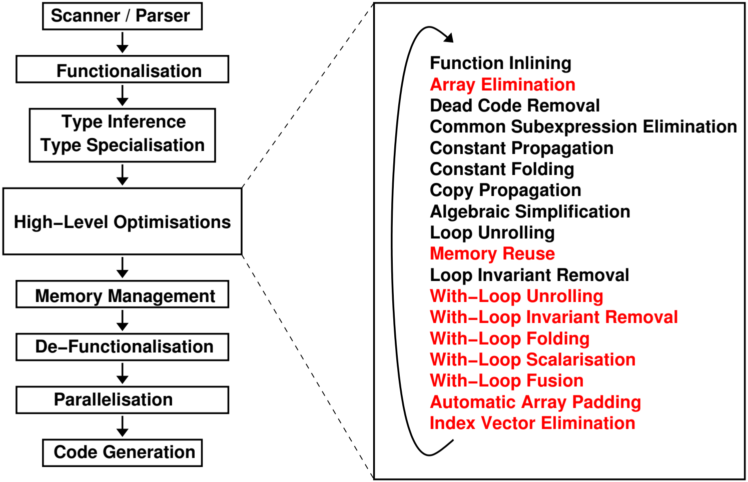
Some Compilation Challenges

- **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?

Some 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: Organisation of Concurrent Execution**
  - How to schedule index spaces to threads ?
  - When to synchronise (and when not) ?

Challenge 5: Implementing a Fully-Fledged Compiler



Compiler Engineering

sac2c is a large-scale compilation technology project

- **SAC** compiler + runtime library:
  - 300,000 lines of code
  - about 1000 files
  - about 250 compiler passes
  - + standard prelude
  - + standard library
- More than 15 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 (2003–2012)
- University of Amsterdam, Netherlands (since 2008)
- Heriot-Watt University, Scotland (since 2011)

Always Looking for New Faces !!

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 !!