The SAC Story: From Functional Programming with Curly Brackets to High Performance Computing

Clemens Grelck

University of Amsterdam

MSc Course

Parallel Functional Programming

Chalmers University of Technology
Göteborg, Sweden
May 26, 2015

SAC: High Productivity meets High Performance

Functional Programming with Curly Brackets ?

Design Rationale of SAC

Data-Parallel Functional Array Programming in SAC

Abstraction and Composition

Case Study: Generic Convolution

Compilation Challenge

Does it Work ? Some Experimental Evaluation

Summary and Conclusion

Functional Programming with Curly Brackets ??

What the heck....

Functional Programming with Curly Brackets ??

What the heck....

Imagine...

- (you want to design a new functional language)

Functional Programming with Curly Brackets ??

What the heck....

Imagine...

- (you want to design a new functional language)
- you want to seduce "curly bracket" programmers

Functional Programming with Curly Brackets ??

What the heck....

Imagine...

- (you want to design a new functional language)
- you want to seduce "curly bracket" programmers
- you want people on non-functional programming conferences to "understand" your code

Functional Programming with Curly Brackets ??

What the heck....

Imagine...

- (you want to design a new functional language)
- you want to seduce "curly bracket" programmers
- you want people on non-functional programming conferences to "understand" your code
- you want to exploit functional semantics for compiler optimisation and parallelisation

Functional Programming with Curly Brackets ??

What the heck....

Imagine...

- (you want to design a new functional language)
- you want to seduce "curly bracket" programmers
- you want people on non-functional programming conferences to "understand" your code
- you want to exploit functional semantics for compiler optimisation and parallelisation
- you are pragmatic

Functional Programming with Curly Brackets !!

...then your Factorial function could look like this:

```
int fac( int n)
{
  if (n <= 1) f = 1;
  else f = n * fac(n-1);

  return f;
}
```

Functional Programming with Curly Brackets !!

...then your Factorial function could look like this:

```
int fac( int n)
{
  if (n <= 1) f = 1;
  else f = n * fac(n-1);

  return f;
}
```

...or like this:

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

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:

```
{
  ...
  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:

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

Functional pseudo code:

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

Functional Semantics of SAC

SAC:

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

Functional pseudo code:

```
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 f = 1 in
  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
  let f = fac_while f n
  in  f
```

Functional Programming with Curly Brackets

Execution model:

- **NOT:** Step-wise modification of state
- **BUT:** Context-free substitution of expressions

Functional Programming with Curly Brackets

Execution model:

- **NOT:** Step-wise modification of state
- **BUT:** Context-free substitution of expressions

Role of variables:

- **NOT:** Names of a memory locations
- **BUT:** Placeholders for values

Functional Programming with Curly Brackets

Execution model:

- **NOT:** Step-wise modification of state
- **BUT:** Context-free substitution of expressions

Role of variables:

- **NOT:** Names of a memory locations
- **BUT:** Placeholders for values

Role of functions:

- **NOT:** Subroutines with side-effects
- **BUT:** Mappings of argument values to result values

Interesting but ....

Interesting but ....

... why should I use SAC ?

... is there anything SAC can do better ?

SAC: High Productivity meets High Performance

Functional Programming with Curly Brackets ?

Design Rationale of SAC

Data-Parallel Functional Array Programming in SAC

Abstraction and Composition

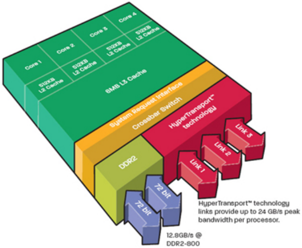
Case Study: Generic Convolution

Compilation Challenge

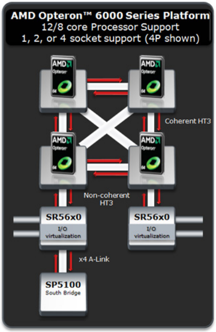
Does it Work ? Some Experimental Evaluation

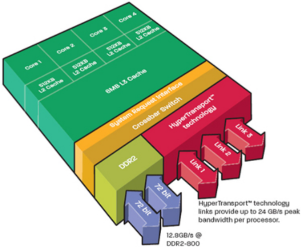
Summary and Conclusion

Today's Hardware: The Multi-/Many-Core Zoo

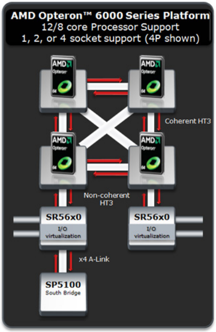


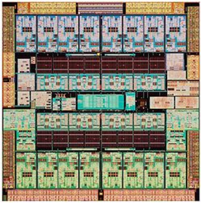
Today's Hardware: The Multi-/Many-Core Zoo

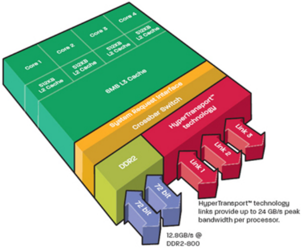




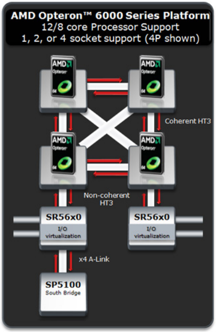
Today's Hardware: The Multi-/Many-Core Zoo

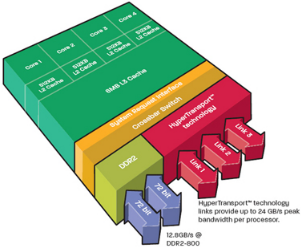


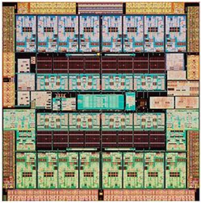




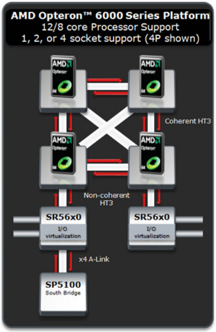
Today's Hardware: The Multi-/Many-Core Zoo

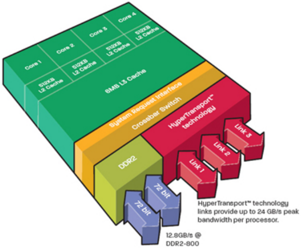


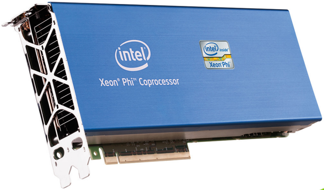


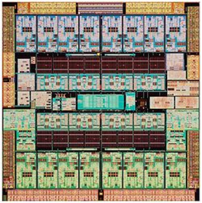


Today's Hardware: The Multi-/Many-Core Zoo









Design Rationale of SAC

Hardware in the many-core era is a zoo:

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

Design Rationale of SAC

Hardware in the many-core era is a zoo:

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

SAC: Genericity through abstraction

Design Rationale of SAC

Hardware in the many-core era is a zoo:

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

SAC: Genericity through abstraction

- Program what to compute, not exactly how

Design Rationale of SAC

Hardware in the many-core era is a zoo:

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

SAC: Genericity through abstraction

- Program what to compute, not exactly how
- Leave execution organisation to compiler and runtime system

Design Rationale of SAC

Hardware in the many-core era is a zoo:

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

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

Hardware in the many-core era is a zoo:

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

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

Design Rationale of SAC

Hardware in the many-core era is a zoo:

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

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

Design Rationale of SAC

Hardware in the many-core era is a zoo:

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

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

What Does Data Parallel Really Mean ?

Factorial recursive:

```
int fac( int n)
{
  if (n <= 1) f = 1;
  else f = n * fac(n-1);

  return f;
}
```

Factorial iterative:

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

What Does Data Parallel Really Mean ?

Factorial recursive:

```
int fac( int n)
{
  if (n <= 1) f = 1;
  else f = n * fac(n-1);

  return f;
}
```

Factorial iterative:

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

Factorial data parallel:

```
int fac( int n)
{
  return prod( 1 + iota( n));
}
```

What Does Data Parallel Really Mean ?

Factorial recursive:

```
int fac( int n)
{
  if (n <= 1) f = 1;
  else f = n * fac(n-1);

  return f;
}
```

Factorial iterative:

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

Factorial data parallel:

```
int fac( int n)
{
  return prod( 1 + iota( n));
}
```

10

What Does Data Parallel Really Mean ?

Factorial recursive:

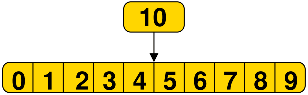
```
int fac( int n)
{
  if (n <= 1) f = 1;
  else f = n * fac(n-1);

  return f;
}
```

Factorial iterative:

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

Factorial data parallel:

```
int fac( int n)
{
  return prod( 1 + iota( n));
}
```



What Does Data Parallel Really Mean ?

Factorial recursive:

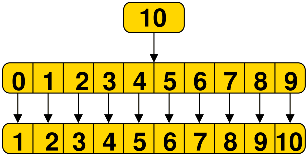
```
int fac( int n)
{
  if (n <= 1) f = 1;
  else f = n * fac(n-1);

  return f;
}
```

Factorial iterative:

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

Factorial data parallel:

```
int fac( int n)
{
  return prod( 1 + iota( n));
}
```



What Does Data Parallel Really Mean ?

Factorial recursive:

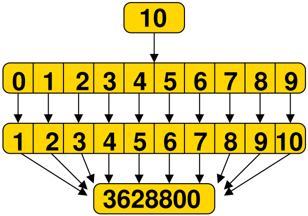
```
int fac( int n)
{
  if (n <= 1) f = 1;
  else f = n * fac(n-1);

  return f;
}
```

Factorial iterative:

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

Factorial data parallel:

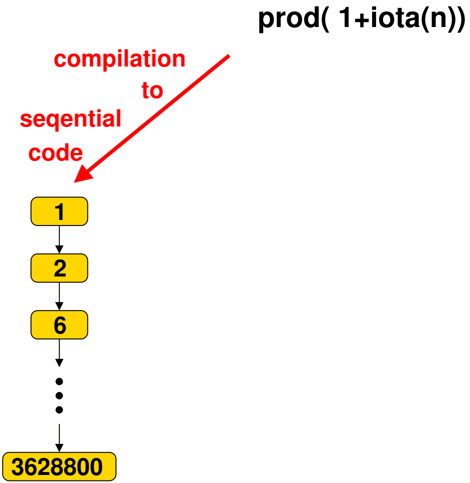
```
int fac( int n)
{
  return prod( 1 + iota( n));
}
```



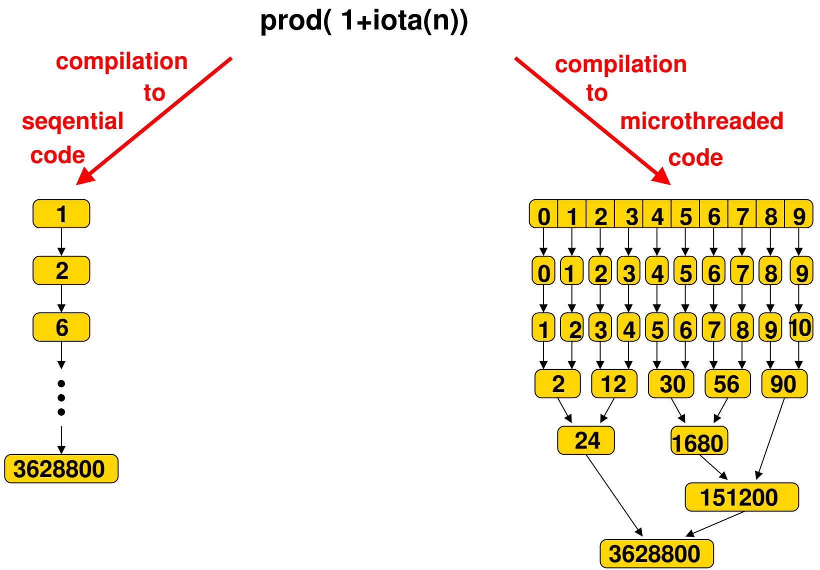
The Essence of Data Parallel Programming

prod(1+iota(n))

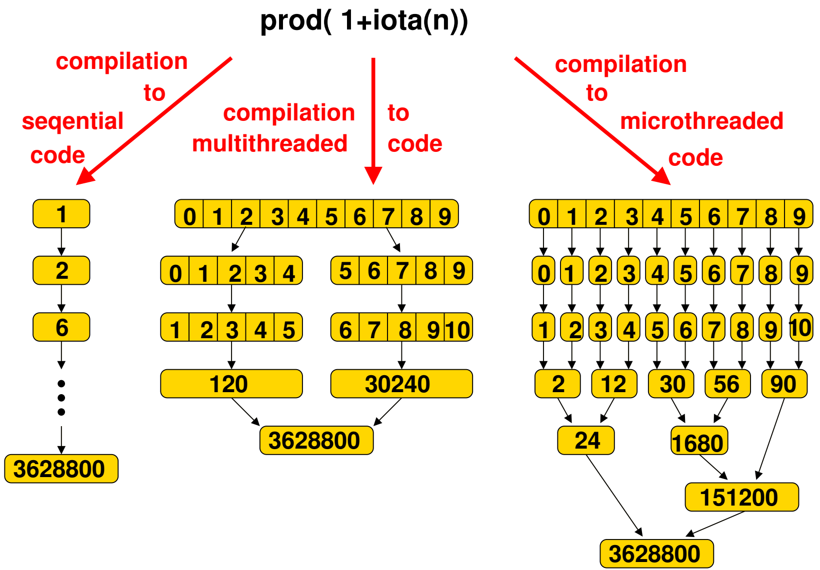
The Essence of Data Parallel Programming



The Essence of Data Parallel Programming



The Essence of Data Parallel Programming



SAC: High Productivity meets High Performance

Functional Programming with Curly Brackets ?

Design Rationale of SAC

Data-Parallel Functional Array Programming in SAC

Abstraction and Composition

Case Study: Generic Convolution

Compilation Challenge

Does it Work ? Some Experimental Evaluation

Summary and Conclusion

Multidimensional Arrays in SAC

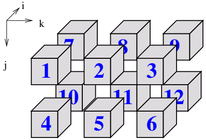
$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

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]

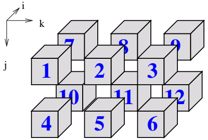


dim: 3
shape: [2,2,3]
data: [1,2,3,4,5,6,7,8,9,10,11,12]

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]

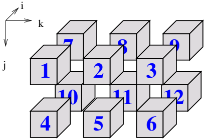
[1, 2, 3, 4, 5, 6]

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.)

SAC: High Productivity meets High Performance

Functional Programming with Curly Brackets ?

Design Rationale of SAC

Data-Parallel Functional Array Programming in SAC

Abstraction and Composition

Case Study: Generic Convolution

Compilation Challenge

Does it Work ? Some Experimental Evaluation

Summary and Conclusion

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);
}
```

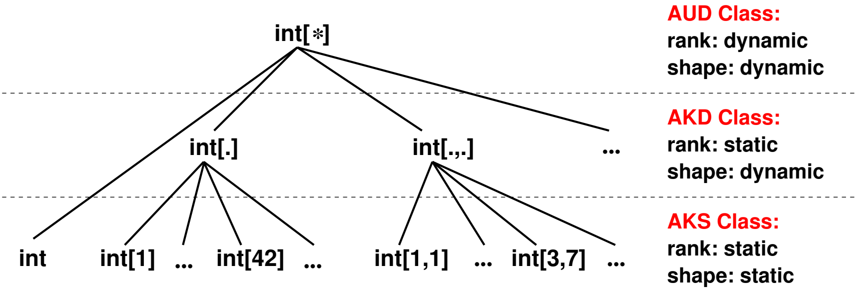
Principle of Abstraction

```
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
- Function dispatch:
  - as static as possible
  - as dynamic as needed

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

SAC: High Productivity meets High Performance

Functional Programming with Curly Brackets ?

Design Rationale of SAC

Data-Parallel Functional Array Programming in SAC

Abstraction and Composition

Case Study: Generic Convolution

Compilation Challenge

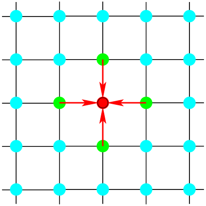
Does it Work ? Some Experimental Evaluation

Summary and Conclusion

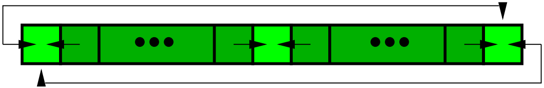
Case Study: Convolution

Algorithmic principle:

Compute weighted sums
of neighbouring elements



Periodic boundary conditions (1-dimensional):



Convolution Step in SAC

1-dimensional:

```
double[.] convolution_step (double[.] A)
{
  R = with {
         ...
      }
  return R;
}
```

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 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;
}
```

Convergence check:

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

Summary: Power of Abstraction

Functional array programming in SAC:

- High productivity software engineering and maintenance
- High confidence in correctness of code
- Programming by abstraction
- Programming by composition of abstractions
- High readability of code
- Entirely architecture- and resource-agnostic

Summary: Power of Abstraction

Functional array programming in SAC:

- High productivity software engineering and maintenance
- High confidence in correctness of code
- Programming by abstraction
- Programming by composition of abstractions
- High readability of code
- Entirely architecture- and resource-agnostic

Opportunities for compiler and runtime system:

- Aggressive machine-independent optimisation exploiting compositional, side-effect-free semantics
- Machine-specific customisation and adaptation
- Automatic granularity control:
  Customised adaptation to concrete computing architecture
- Automatic resource management:
  memory, cores, nodes, energy, ...

SAC: High Productivity meets High Performance

Functional Programming with Curly Brackets ?

Design Rationale of SAC

Data-Parallel Functional Array Programming in SAC

Abstraction and Composition

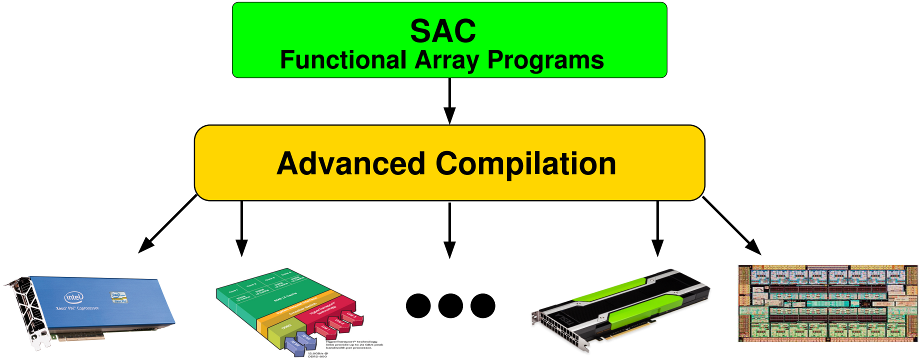
Case Study: Generic Convolution

Compilation Challenge

Does it Work ? Some Experimental Evaluation

Summary and Conclusion

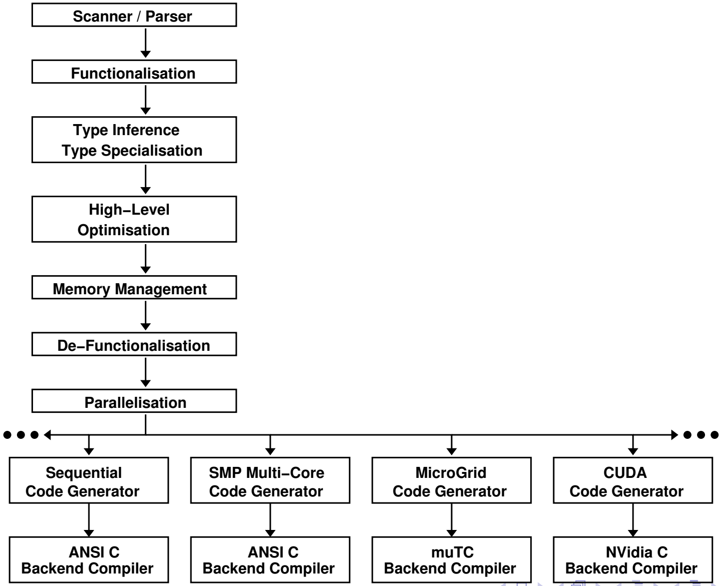
Compilation Challenge



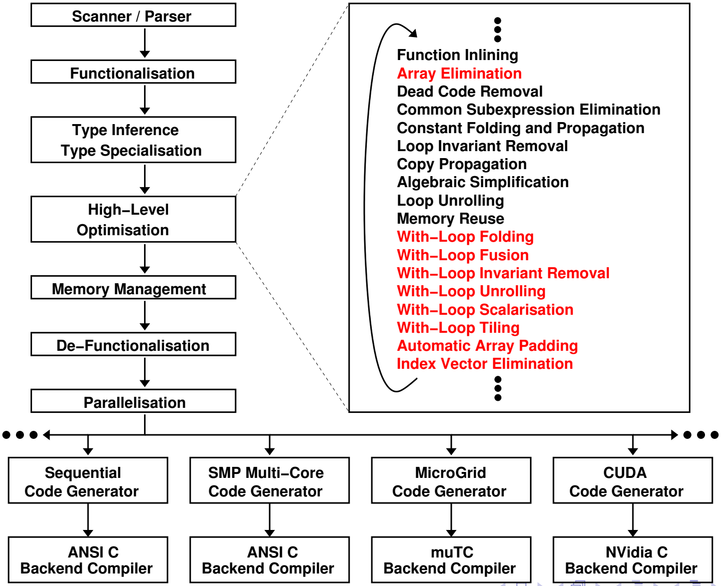
And achieve reasonably high performance....

Compilation Challenge

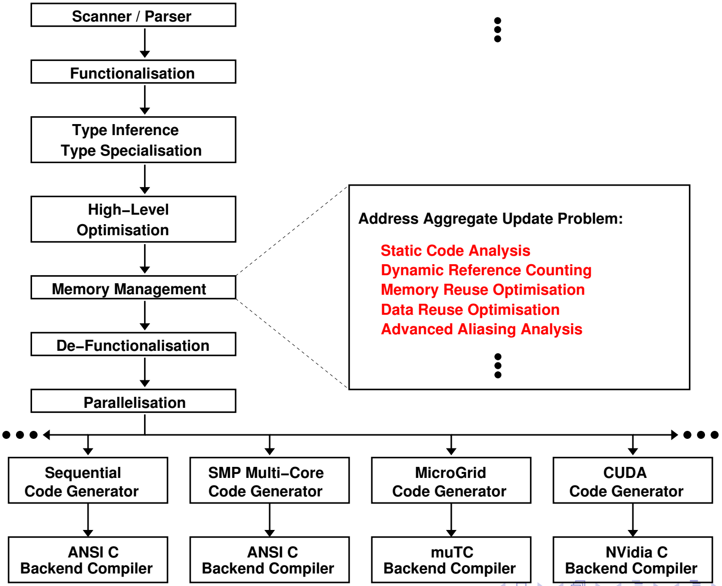
Challenge Taken: the **SAC** Compiler



Challenge Taken: the **SAC** Compiler



Challenge Taken: the **SAC** Compiler



Challenge Taken: the **SAC** Compiler

Compiler fact sheet:

- Around 300,000 lines of code
- Around 1000 files:
  - + standard prelude
  - + standard library
- Around 250 compiler passes
- Complete compiler construction toolkit as side product:
  - re-used in other compiler research projects
  - re-used in teaching compiler courses (Bachelor/Master)

Challenge Taken: the **SAC** Compiler

Compiler fact sheet:

- Around 300,000 lines of code
- Around 1000 files:
  - + standard prelude
  - + standard library
- Around 250 compiler passes
- Complete compiler construction toolkit as side product:
  - re-used in other compiler research projects
  - re-used in teaching compiler courses (Bachelor/Master)

Where's the trick ?

- Purely functional semantics benefits large-scale program transformation
- Stringent language–compiler co-design with one goal: high performance in parallel execution of array programs

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)

SAC: High Productivity meets High Performance

Functional Programming with Curly Brackets ?

Design Rationale of SAC

Data-Parallel Functional Array Programming in SAC

Abstraction and Composition

Case Study: Generic Convolution

Compilation Challenge

Does it Work ? Some Experimental Evaluation

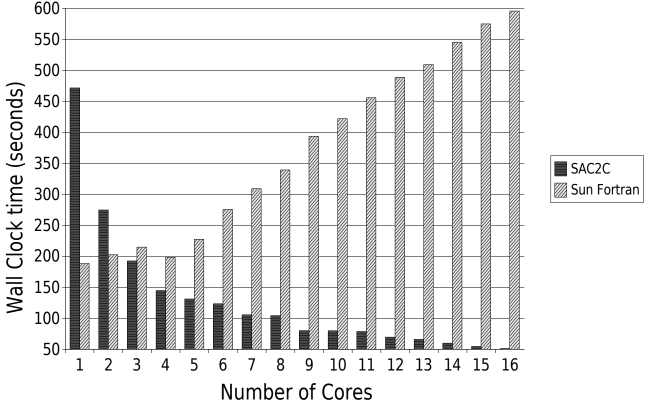
Summary and Conclusion

Experiment: **SAC** on x86 Multi-Core Multi-Processor

Machine:

- 4 AMD Opteron 8356 processors
- 4 fully-fledged cores each

Unsteady Shock Wave Simulation:

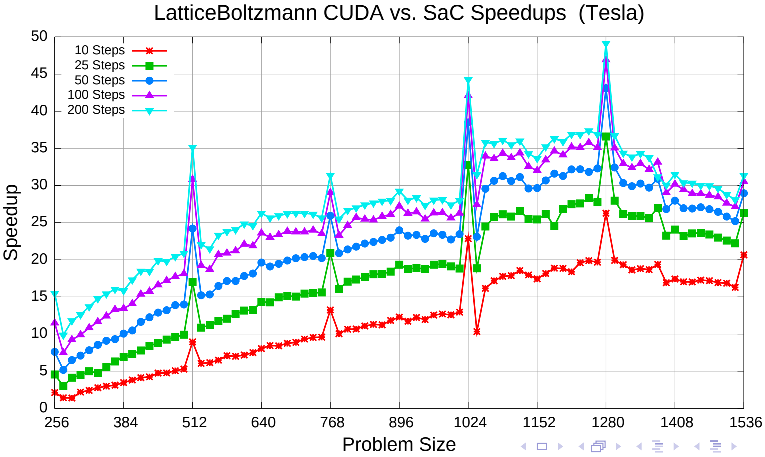


Experiment: **SAC** on Graphics Accelerator

Machine:

- NVidia Tesla GPU

Lattice-Boltzmann:



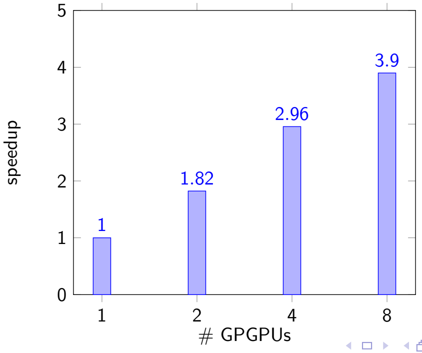
Experiment: **SAC** on Multiple Graphics Accelerators

Machine:

- 8 NVidia GeForce GTX 580

Convolution kernel:

- 8000x8000 matrix, 10000 iterations
- requires data exchange between GPGPUs after each iteration

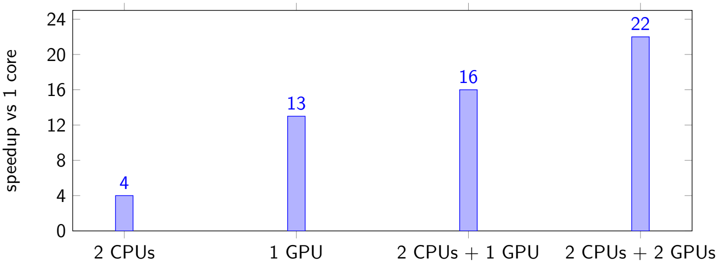


Experiment: **SAC** on Heterogeneous System

Machine:

- 2 quad-core Intel Xeon processors
- 2 NVidia GTX480 GPUs

Convolution kernel:

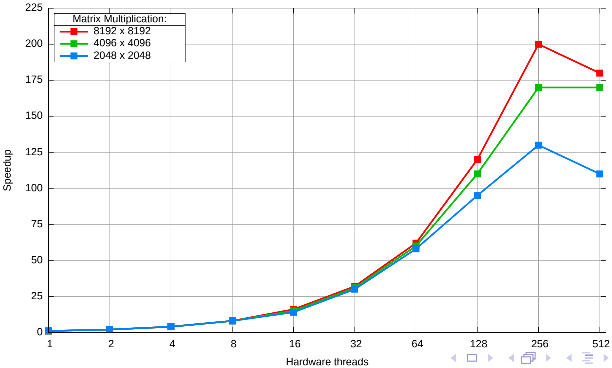


Experiment: **SAC** on Ultra Sparc T3-4 Server

Machine:

- 4 Oracle Ultra Sparc T3 processors
- $4 \times 16 \; = \; 64$ cores
- $4 \times 16 \times 8 \; = \; 512$ hardware threads

Matrix multiplication:



SAC: High Productivity meets High Performance

Functional Programming with Curly Brackets ?

Design Rationale of SAC

Data-Parallel Functional Array Programming in SAC

Abstraction and Composition

Case Study: Generic Convolution

Compilation Challenge

Does it Work ? Some Experimental Evaluation

Summary and Conclusion

Summary

Language design:

- Functional state-less semantics with C-like syntax
- Data parallel array programming
- Abstraction and composition
- Shape-generic programming
- Index-free programming

Summary

Language design:

- Functional state-less semantics with C-like syntax
- Data parallel array programming
- Abstraction and composition
- Shape-generic programming
- Index-free programming

Language implementation:

- Fully-fledged compiler
- Automatic parallelisation
- Automatic memory management
- High-level program transformation
- Large-scale machine-independent optimisation
- Performance competitive with the “real” curly brackets!!

The End

Questions ?

Check out www.sac-home.org !!

Bonus Slides: Input and Output

Example:

```
import StdIO: all;
import ArrayIO: all;

int main()
{
  a = 42;
  b = [1,2,3,4,5];

  errcode, outfile = fopen( "filename", "w");

  fprintf( outfile, "a = %d\n", a);
  fprint( outfile, b);

  fclose( outfile);

  return 0;
}
```

Digression: Input and Output

Example functionalised by compiler:

```
FileSystem, int main( FileSystem theFileSystem)
{
  a = 42;
  b = [1,2,3,4,5];

  theFileSystem, errcode, outfile
    = fopen( theFileSystem, "filename", "w");

  outfile = fprintf( outfile, "a = %d\n", a);
  outfile = fprint( outfile, b);

  theFileSystem = fclose( theFileSystem, outfile);

  return( theFileSystem, 0);
}
```

FileSystem and File are uniqueness types.