Declarative Array Programming with Single Assignment C (SAC)

Language Design and Compiler Technology

Clemens Grelck

Universiteit van Amsterdam

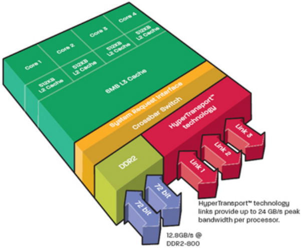
2nd HIPERFIT Workshop

Copenhagen, Denmark

Dec 1/2, 2011

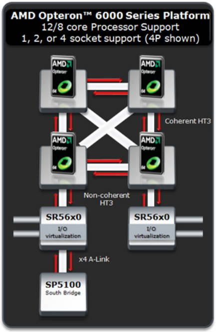
The Free Lunch is Over: Many-Core to the Rescue

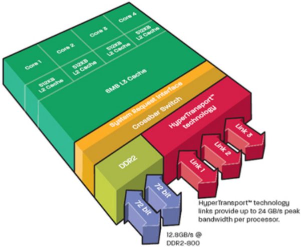
The many-core hardware zoo:



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

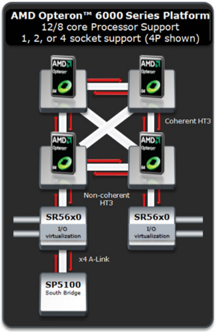
The many-core hardware zoo:

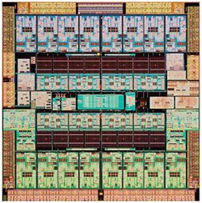


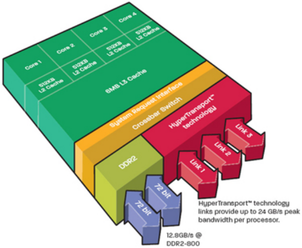


The Free Lunch is Over: Many-Core to the Rescue

The many-core hardware zoo:

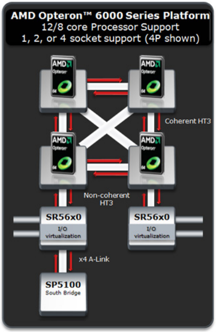


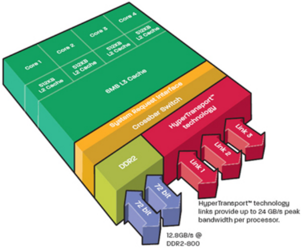


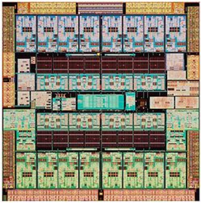


The Free Lunch is Over: Many-Core to the Rescue

The many-core hardware zoo:

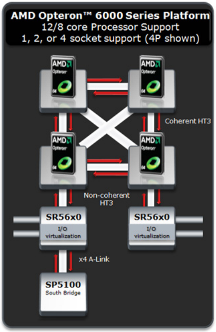


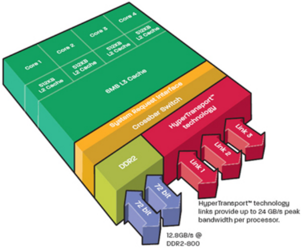


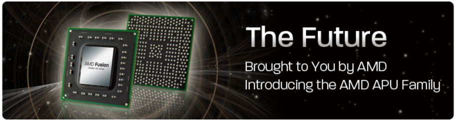


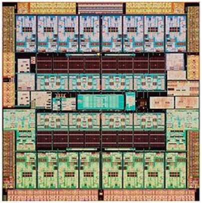
The Free Lunch is Over: Many-Core to the Rescue

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

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
- Promote multidimensional arrays as main data structure
- Pursue data-parallel approach to automatically exploit concurrency

Why (Data Parallel) Array Programming ?

Factorial imperative:

```
int fac( int n)
{
  int 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)
```

Why (Data Parallel) Array Programming ?

Factorial imperative:

```
int fac( int n)
{
  int 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)
```

Factorial data parallel:

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

Why (Data Parallel) Array Programming ?

Factorial imperative:

```
int fac( int n)
{
  int 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)
```

Factorial data parallel:

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

10

Why (Data Parallel) Array Programming ?

Factorial imperative:

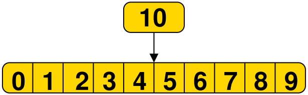
```
int fac( int n)
{
  int f = 1;
  while (n > 1) {
    f = f * n;
    n = n - 1;
  }
  return f;
}
```

Factorial functional:

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

Factorial data parallel:

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



Why (Data Parallel) Array Programming ?

Factorial imperative:

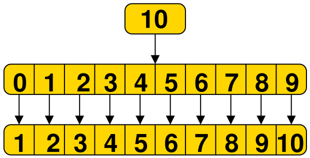
```
int fac( int n)
{
  int f = 1;
  while (n > 1) {
    f = f * n;
    n = n - 1;
  }
  return f;
}
```

Factorial functional:

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

Factorial data parallel:

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



Why (Data Parallel) Array Programming ?

Factorial imperative:

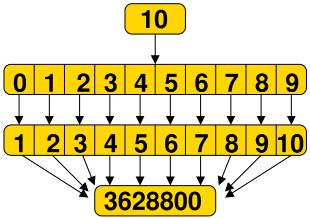
```
int fac( int n)
{
  int f = 1;
  while (n > 1) {
    f = f * n;
    n = n - 1;
  }
  return f;
}
```

Factorial functional:

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

Factorial data parallel:

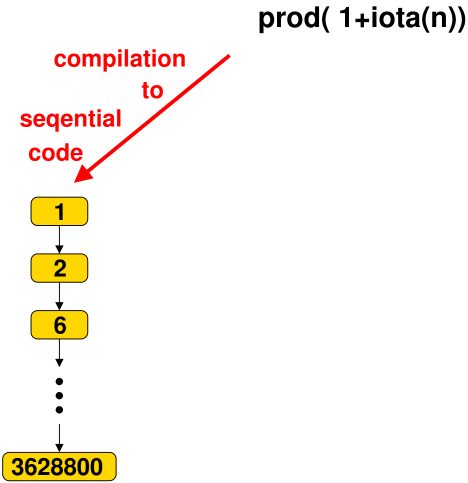
```
fac n = prod( 1 + iota( n));
```



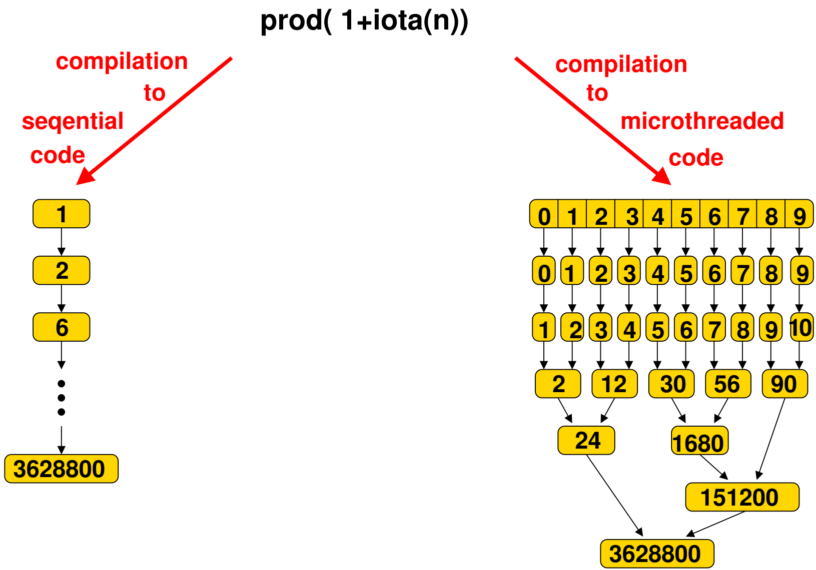
The Essence of (Data Parallel) Array Programming

prod(1+iota(n))

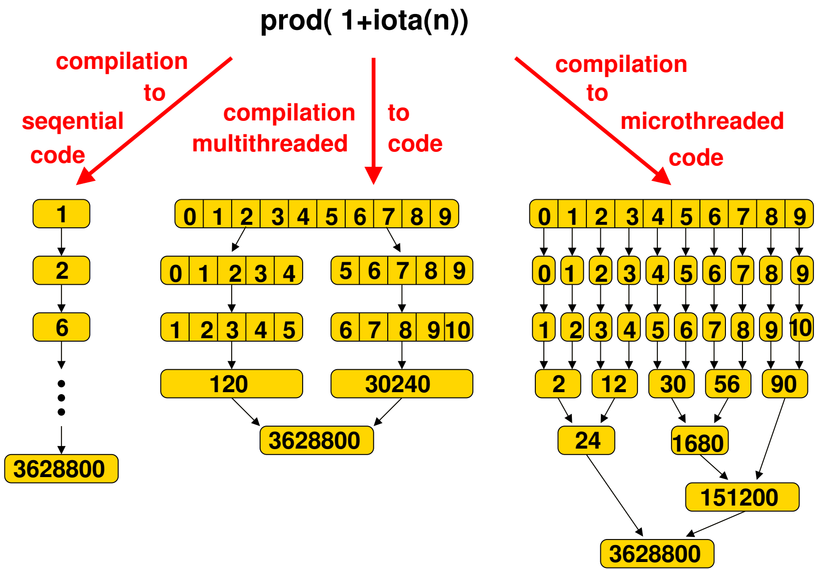
The Essence of Data Parallel Programming



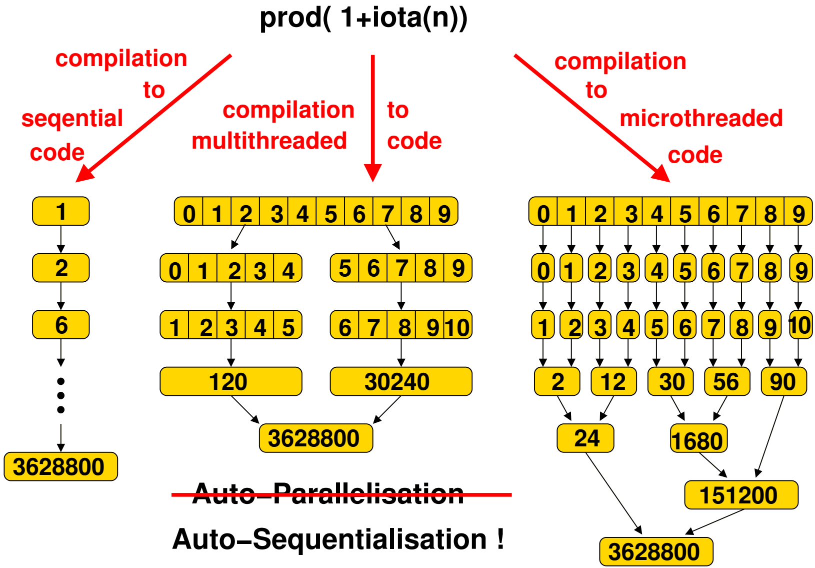
The Essence of Data Parallel Programming



The Essence of Data Parallel Programming



The Essence of Data Parallel Programming



SAC — Design Space

SAC

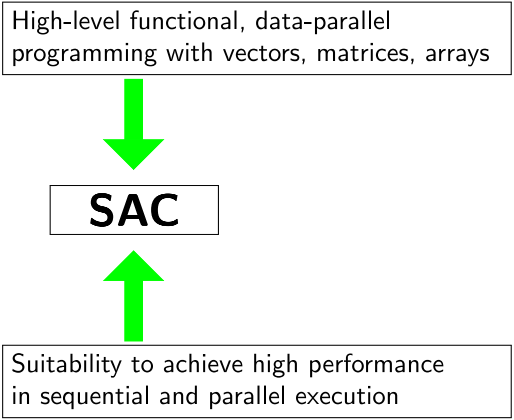
SAC — Design Space

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

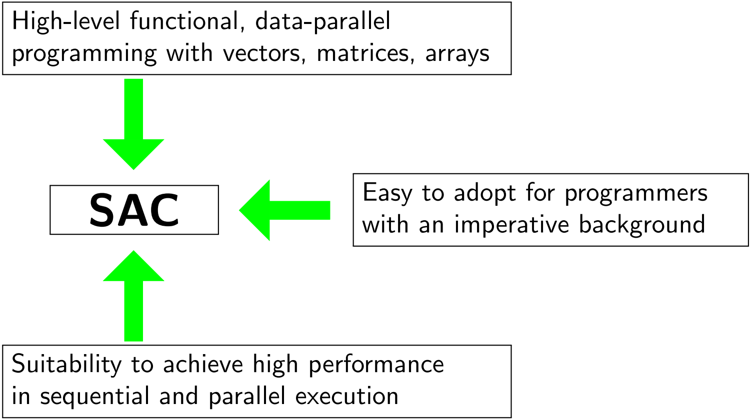
↓

SAC

SAC — Design Space



SAC — Design Space



Introductory Example: gcd in SAC

Euclid's algorithm:

```
int gcd( int high, int low)
{
  if (high < low) {
    mem  = low;
    low  = high;
    high = mem;
  }
  while (low != 0) {
    remain = high % low;
    high   = low;
    low    = remain;
  }
  return high;
}
```

What means Functional Array Programming ?

- **Execution Model**
  - Contextfree substitution of expressions

What means Functional Array Programming ?

- **Execution Model**
  - Contextfree substitution of expressions
- **Role of Functions**
  - Map argument values to result values
  - No side effects
  - Call-by-value parameter passing

What means Functional Array Programming ?

- **Execution Model**
  - Contextfree substitution of expressions
- **Role of Functions**
  - Map argument values to result values
  - No side effects
  - Call-by-value parameter passing
- **Role of Variables**
  - Variables are placeholders for values
  - Variables do not denote memory locations

What means Functional Array Programming ?

- **Execution Model**
  - Contextfree substitution of expressions
- **Role of Functions**
  - Map argument values to result values
  - No side effects
  - Call-by-value parameter passing
- **Role of Variables**
  - Variables are placeholders for values
  - Variables do not denote memory locations
- **Control flow constructs**
  - Branches are syntactic sugar for conditional expressions
  - Loops are syntactic sugar for tail-end recursive functions
  - Data flow determines execution order

What means Functional Array Programming ?

- **Execution Model**
  - Contextfree substitution of expressions
- **Role of Functions**
  - Map argument values to result values
  - No side effects
  - Call-by-value parameter passing
- **Role of Variables**
  - Variables are placeholders for values
  - Variables do not denote memory locations
- **Control flow constructs**
  - Branches are syntactic sugar for conditional expressions
  - Loops are syntactic sugar for tail-end recursive functions
  - Data flow determines execution order
- **Nature of Arrays**
  - Pure values, mapping indices to (other) values
  - No state, no fixed memory representation

Calculus of Multidimensional Arrays

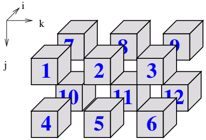
$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

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

Calculus of Multidimensional Arrays

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

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

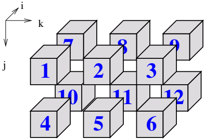


dim: 3
shape: [2,2,3]
data: [1,2,3,4,5,6,7,8,9,10,11,12]

Calculus of Multidimensional Arrays

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

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]

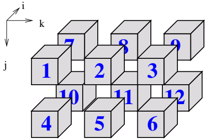
[1, 2, 3, 4, 5, 6]

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

Calculus of Multidimensional Arrays

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

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]

index domain

➡

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

value domain

With-Loops: Versatile Array Comprehensions

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

Variations:

- Multiple generators
- Strided generators
- Multiple operators
- Other defaults
- Reductions
- etc

Principle of Abstraction

Characteristics:

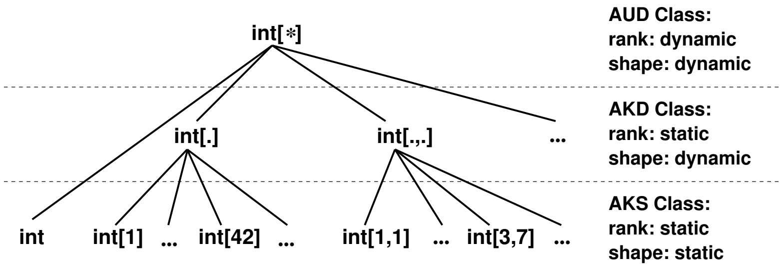
- Use with-loops to define elementary array operations
- Array versions of scalar built-in functions and operators
- Structural operations like rotation and shifting
- Standard reductions
- and much more

Principle of Abstraction

Characteristics:

- Use with-loops to define elementary array operations
- Array versions of scalar built-in functions and operators
- Structural operations like rotation and shifting
- Standard reductions
- and much more

Hierarchy of array types with subtyping and overloading:



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));
}
```

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- and Rank-Generic Programming

2-dimensional convergence test:

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

Shape- and Rank-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:

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

The Power of Abstraction and Composition

- NO large collection of built-in operations
  - Simplified compiler design

The Power of Abstraction and Composition

- NO large collection of built-in operations
  - Simplified compiler design
- INSTEAD: **library of array operations**
  - Improved maintainability
  - Improved extensibility

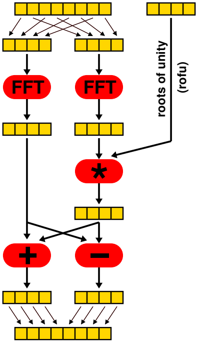
The Power of Abstraction and Composition

- 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 Abstraction and Composition

- 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

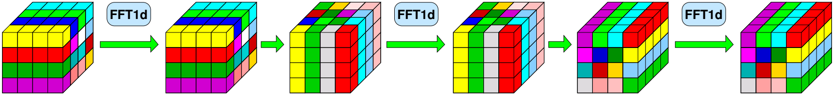
Case Study: 1-Dimensional Complex FFT (NAS-FT)




```
complex[.] FFT(complex[.] v, complex[.] rofu)
{
  even = condense(2, v);
  odd  = condense(2, drop( [1], v));

  even = FFT( even, rofu);
  odd  = FFT( odd,  rofu);

  rofu = condense( len(rofu) / len(odd), rofu)

  left  = even + odd * rofu;
  right = even - odd * rofu;

  return left ++ right;
}
```

Case Study: 3-Dimensional Complex FFT (NAS-FT)

Algorithmic idea:



Implementation:

```
complex[.,.,.] FFT( complex[.,.,.] a, complex[.] rofu)
{
  b = { [.,y,z] -> FFT( a[.,y,z], rofu) };
  c = { [x,.,z] -> FFT( b[x,.,z], rofu) };
  d = { [x,y,.] -> FFT( c[x,y,.], rofu) };

  return d;
}

typedef double[2] complex;
```

The Same in Fortran

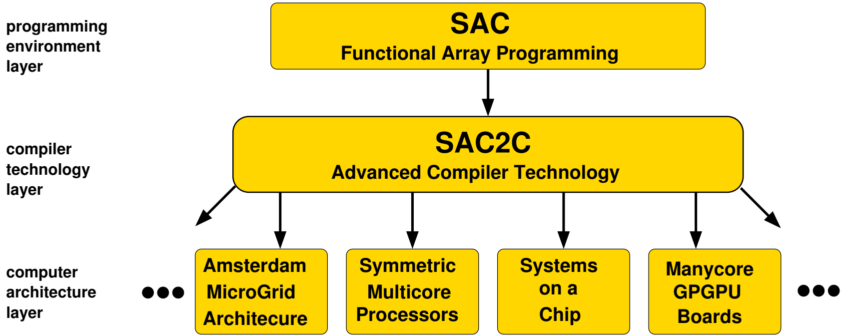
```
subroutine fft(dir, x1, x2)
implicit none
include 'global.h'
integer dir
double complex x1(ntotal), x2(ntotal)
double complex scratch(fftblockpad_default
>                      *maxdim*2)
if (dir .eq. 1) then
   call cffts1(1, dims(1,1), x1, x1, scratch)
   call cffts2(1, dims(1,2), x1, x1, scratch)
   call cffts3(1, dims(1,3), x1, x2, scratch)
else
   call cffts3(-1, dims(1,3), x1, x1, scratch)
   call cffts2(-1, dims(1,2), x1, x1, scratch)
   call cffts1(-1, dims(1,1), x1, x2, scratch)
endif
return
end
subroutine cffts1(is, d, x, xout, y)
implicit none
include 'global.h'
integer is, d(3), logd(3)
double complex x(d(1),d(2),d(3))
double complex xout(d(1),d(2),d(3))
double complex y(fftblockpad, d(1), 2)
integer i, j, k, jj
do i = 1, 3
   logd(i) = ilog2(d(i))
end do
do k = 1, d(3)
   do jj = 0, d(2) - fftblock, fftblock
      do j = 1, fftblock
         do i = 1, d(1)
            y(j,i,1) = x(i,j+jj,k)
         enddo
      enddo
```

```
      call cfftz (is, logd(1),
>                 d(1), y, y(1,1,2))
      do j = 1, fftblock
         do i = 1, d(1)
            xout(i,j+jj,k) = y(j,i,1)
         enddo
      enddo
   enddo
enddo
return
end
subroutine cffts2(is, d, x, xout, y)
implicit none
include 'global.h'
integer is, d(3), logd(3)
double complex x(d(1),d(2),d(3))
double complex xout(d(1),d(2),d(3))
double complex y(fftblockpad, d(2), 2)
integer i, j, k, ii
do i = 1, 3
   logd(i) = ilog2(d(i))
end do
do k = 1, d(3)
  do ii = 0, d(1) - fftblock, fftblock
    do j = 1, d(2)
      do i = 1, fftblock
        y(i,j,1) = x(i+ii,j,k)
      enddo
    enddo
    call cfftz (is, logd(2),
>               d(2), y, y(1, 1, 2))
    do j = 1, d(2)
      do i = 1, fftblock
        xout(i+ii,j,k) = y(i,j,1)
      enddo
    enddo
  enddo
enddo
return
end
```

```
subroutine cffts3(is, d, x, xout, y)
implicit none
include 'global.h'
integer is, d(3), logd(3)
double complex x(d(1),d(2),d(3))
double complex xout(d(1),d(2),d(3))
double complex y(fftblockpad, d(3), 2)
integer i, j, k, ii
do i = 1, 3
   logd(i) = ilog2(d(i))
end do
do j = 1, d(2)
  do ii = 0, d(1) - fftblock, fftblock
    do k = 1, d(3)
      do i = 1, fftblock
        y(i,k,1) = x(i+ii,j,k)
      enddo
    enddo
    call cfftz (is, logd(3),
>               d(3), y, y(1, 1, 2))
    do k = 1, d(3)
      do i = 1, fftblock
        xout(i+ii,j,k) = y(i,k,1)
      enddo
    enddo
  enddo
enddo
return
end
subroutine cfftz (is, m, n, x, y)
implicit none
include 'global.h'
integer is,m,n,i,j,l,mx
double complex x, y
dimension x(fftblockpad,n), y(fftblockpad,n)
mx = u(1)
do l = 1, m, 2
  call fftz2 (is, l, m, n, fftblock,
>             fftblockpad, u, x, y)
```

```
  if (l .eq. m) goto 160
  call fftz2 (is, l + 1, m, n, ff
>             fftblockpad, u, y,
enddo
goto 180
160 do j = 1, n
  do i = 1, fftblock
    x(i,j) = y(i,j)
  enddo
enddo
180 continue
return
end
subroutine fftz2 (is, l, m, n, ny
implicit none
integer is,k,l,m,n,ny,ny1,n1,li,l
double complex u,x,y,u1,x11,x21
dimension u(n), x(ny1,n), y(ny1,n
n1 = n / 2
lk = 2 ** (l - 1)
li = 2 ** (m - l)
lj = 2 * lk
ku = li + 1
do i = 0, li - 1
  i11 = i * lk + 1
  i12 = i11 + n1
  i21 = i * lj + 1
  i22 = i21 + lk
  if (is .ge. 1) then
    u1 = u(ku+i)
  else
    u1 = dconjg (u(ku+i))
  endif
  do k = 0, lk - 1
    do j = 1, ny
      x11 = x(j,i11+k)
      x21 = x(j,i12+k)
      y(j,i21+k) = x11 + x21
      y(j,i22+k) = u1 * (x11 - x2
    enddo
  enddo
enddo
return
end
```

Compilation Challenge



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?

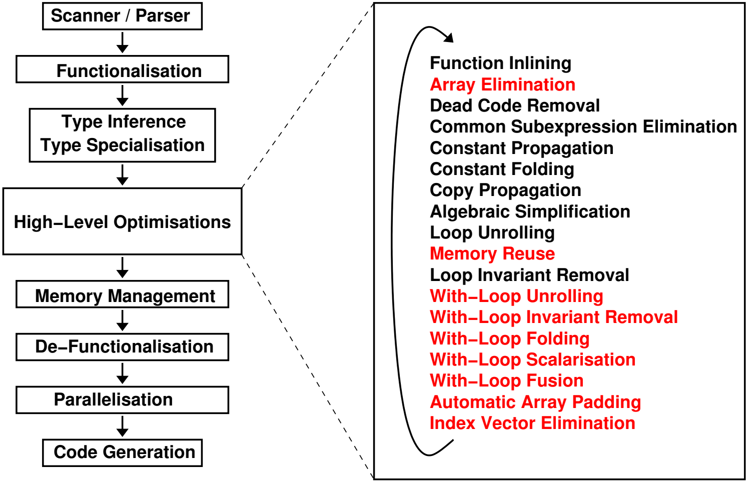
Compilation Challenges

- **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: Organisation of Concurrent Execution**
  - How to schedule index spaces to threads ?
  - When to synchronise (and when not) ?
  - Where does parallel execution pay off ?
  - Granularity control ?

Challenge 5: Implementing a Fully-Fledged Compiler



SAC as a Compiler Technology Project

Large-scale (academic) 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
- More than 30 people involved over the years
- Mostly BSc/MSc students, 5 PhDs

The SAC Project: Credits

Involved Universities:

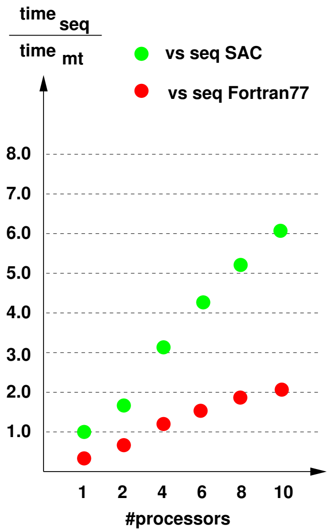
- University of Kiel, Germany (1994–2005)
- University of Toronto, Canada (since 2000)
- University of Lübeck, Germany (2001–2008)
- University of Hertfordshire, England (2004–2012)
- University of Amsterdam, Netherlands (since 2008)
- Heriot-Watt University, Scotland (since 2011)

Main Contributors:

- Sven-Bodo Scholz (Kiel, Herts, Heriot-Watt)
- Clemens Grelck (Kiel, Lübeck, Herts, Amsterdam)

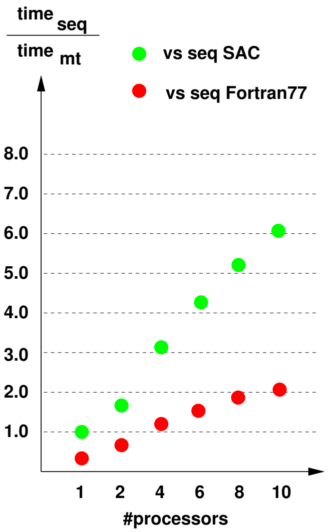
- Stephan Herhut (Kiel, Herts, now at Intel, Santa Clara)
- Kai Trojahner (Lübeck, now at RTT AG, München)
- Dietmar Kreye (Kiel, now at sd&m AG, Hamburg)
- Robert Bernecky (Toronto)
- Jing Guo (Herts)

Runtime Performance: Standard Multiprocessor

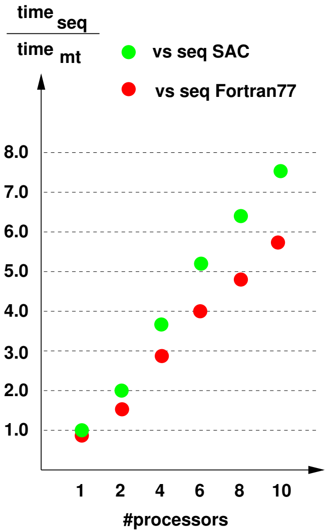


NAS benchmark FT

Runtime Performance: Standard Multiprocessor



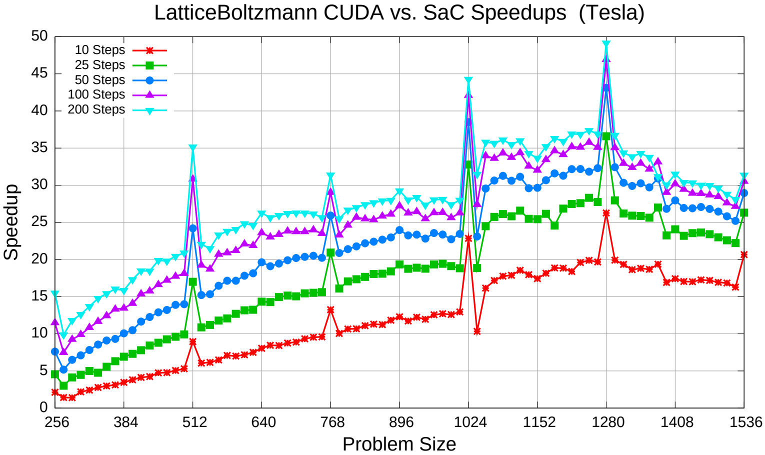
NAS benchmark FT



NAS benchmark MG

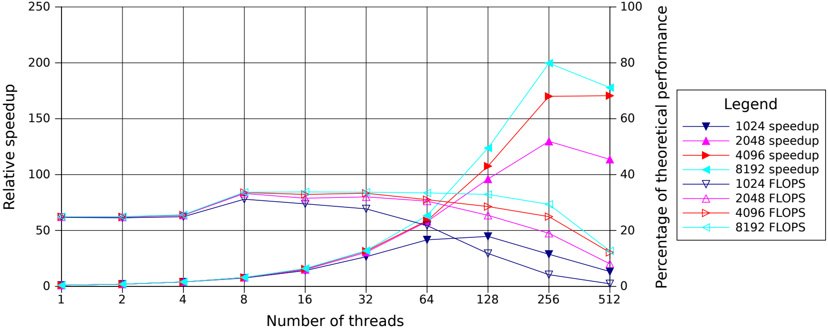
Runtime Performance: NVidia Tesla

Lattice-Boltzmann:



Runtime Performance: Ultra Sparc T3-4 Server

Matrix Multiplication:



Summary

Language design:

- Functional state-less semantics but C-like syntax
- Architecture-agnostic high-level parallel programming
- Shape- and rank-generic array programming
- Index-free (index-less) array programming

Summary

Language design:

- Functional state-less semantics but C-like syntax
- Architecture-agnostic high-level parallel programming
- Shape- and rank-generic array programming
- Index-free (index-less) array programming

Language implementation:

- Fully-fledged compiler, not an embedded DSL
- Large-scale machine-independent optimisation
- Automatic parallelisation for various architectures
- Automatic granularity adaptation and control
- Automatic memory management

The End

Questions ?

Check out www.sac-home.org !!