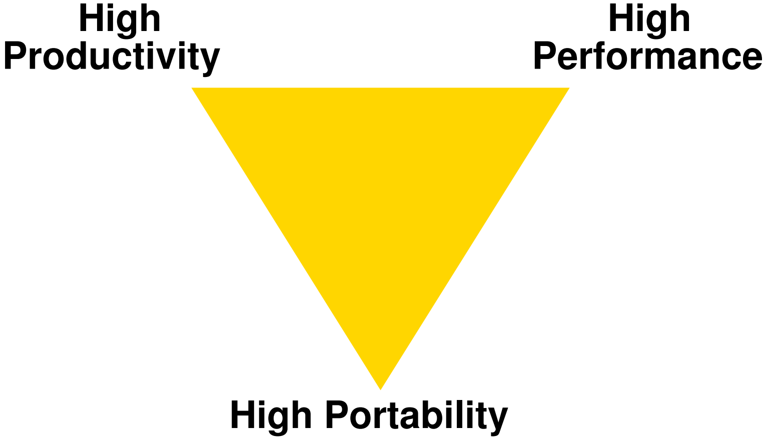
Single Assignment C: A High Productivity Language for High Performance Computing

Clemens Grelck

Universiteit van Amsterdam

Workshop
Trends in High-Performance Distributed Computing
Amsterdam, Netherlands
March 14, 2012

The HP^3 Triangle in High Performance Computing

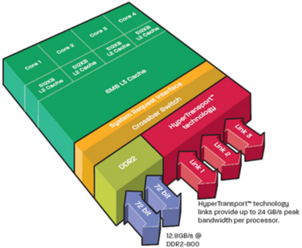


HPC in the Age of Multi- and Many-Core

Welcome to the Cluster Node Hardware Zoo!!

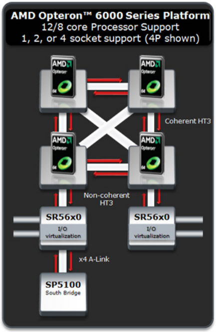
HPC in the Age of Multi- and Many-Core

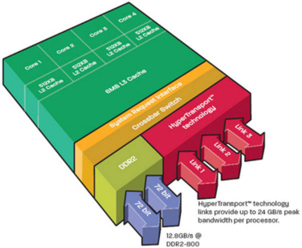
Welcome to the Cluster Node Hardware Zoo!!



HPC in the Age of Multi- and Many-Core

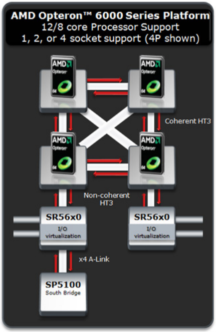
Welcome to the Cluster Node Hardware Zoo!!

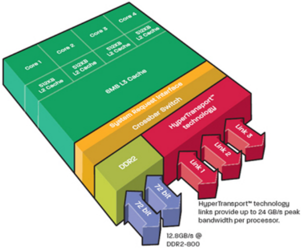


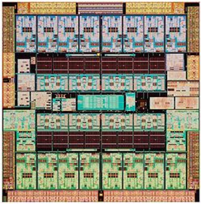


HPC in the Age of Multi- and Many-Core

Welcome to the Cluster Node Hardware Zoo!!

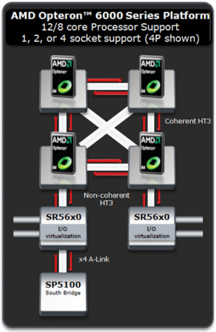


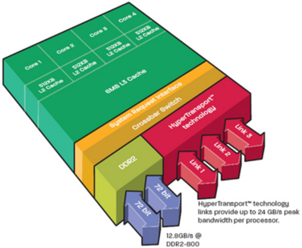


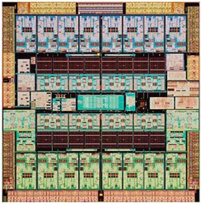


HPC in the Age of Multi- and Many-Core

Welcome to the Cluster Node Hardware Zoo!!

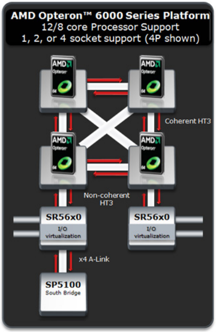


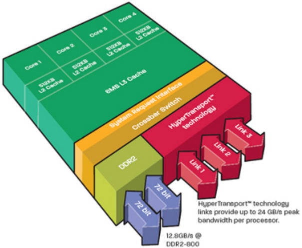


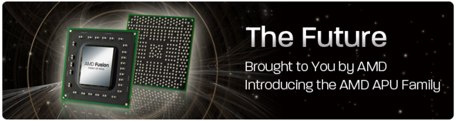


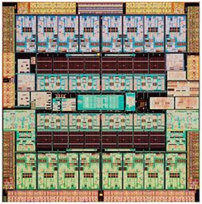
HPC in the Age of Multi- and Many-Core

Welcome to the Cluster Node Hardware Zoo!!



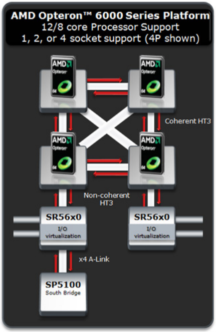


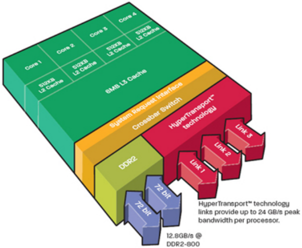


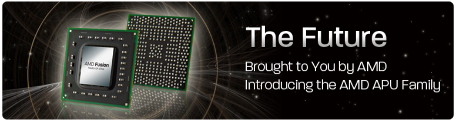


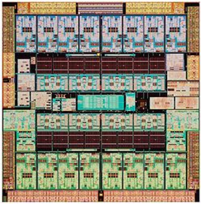
HPC in the Age of Multi- and Many-Core

Welcome to the Cluster Node Hardware Zoo!!









HPC in the Age of Multi- and Many-Core

Compute node hardware is a zoo:

- Vastly different numbers of cores
- Vastly different core architectures: general, restricted
- Vastly different memory architectures

HPC in the Age of Multi- and Many-Core

Compute node hardware is a zoo:

- Vastly different numbers of cores
- Vastly different core architectures: general, restricted
- Vastly different memory architectures

Programming diverse hardware is uneconomic:

- Various low-level programming models
- Each requires different expert knowledge
- Heterogeneous combinations of the above ?
- Cumbersome, error-prone and inefficient

An Alternative: Single Assignment C (SAC)

Credo: abstraction, abstraction, abstraction

- Program what to compute, not exactly how
- Leave concrete organisation of execution to compiler and runtime system
- Put expert knowledge into tools, not into applications
- Architecture-agnostic programming for portability
- Compile one source to diverse target hardware
- Automatically manage resources: memory, cores, etc

SAC — Design Space

$\boxed{\textbf{SAC}}$

SAC — Design Space

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

$\Downarrow$

SAC

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

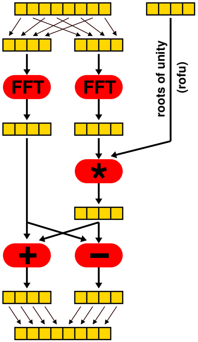
High-level functional, data-parallel
programming with vectors, matrices, arrays

SAC

Easy to adopt for programmers
with an imperative background

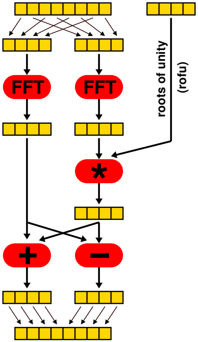
Suitability to achieve high performance
in sequential and parallel execution

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;
}
```

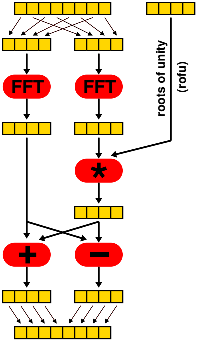
Case Study: 1-Dimensional Complex FFT (NAS-FT)




```
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;
}
```

Now, what is functional array programming ?

Functional Array Programming

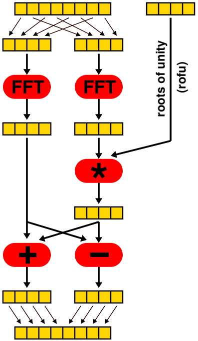



```
complex[.] FFT(complex[.] v, complex[.] rofu)
{
  even = condense(2, v);
  odd  = condense(2, drop( [1], v));

  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;
}
```

Role of Functions:

- Map argument values to result values
- No side effects
- Call-by-value parameter passing

Functional Array Programming

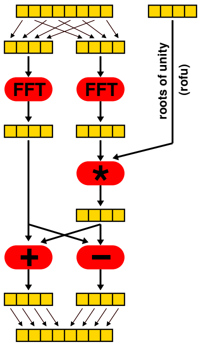



```
complex[.] FFT(complex[.] v, complex[.] rofu)
{
  even = condense(2, v);
  odd  = condense(2, drop( [1], v));

  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;
}
```

Role of Variables:

- Variables are placeholders for values
- Variables do not denote memory locations
- Automatic memory management

Functional Array Programming

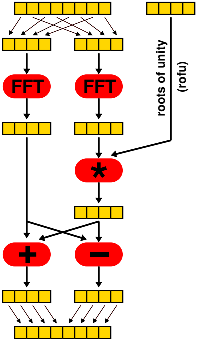



```
complex[.] FFT(complex[.] v, complex[.] rofu)
{
  even = condense(2, v);
  odd  = condense(2, drop( [1], v));

  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;
}
```

Execution Model:

- Contextfree substitution of expressions
- No side-effects

Functional Array Programming

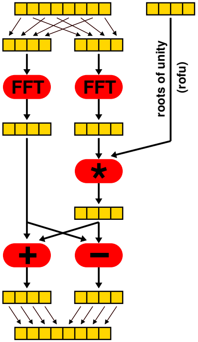



```
complex[.] FFT(complex[.] v, complex[.] rofu)
{
  even = condense(2, v);
  odd  = condense(2, drop( [1], v));

  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;
}
```

Control flow constructs:

- Branches are syntactic sugar for conditional expressions
- Loops are syntactic sugar for tail-end recursive functions
- Data flow determines execution order

Functional Array Programming



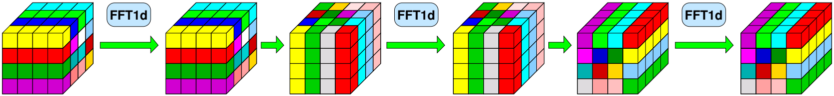

```
complex[.] FFT(complex[.] v, complex[.] rofu)
{
  even = condense(2, v);
  odd  = condense(2, drop( [1], v));

  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;
}
```

Nature of Arrays:

- Pure values, mapping indices to (other) values
- No state, no fixed memory representation
- Maybe no memory manifestation at all

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
```

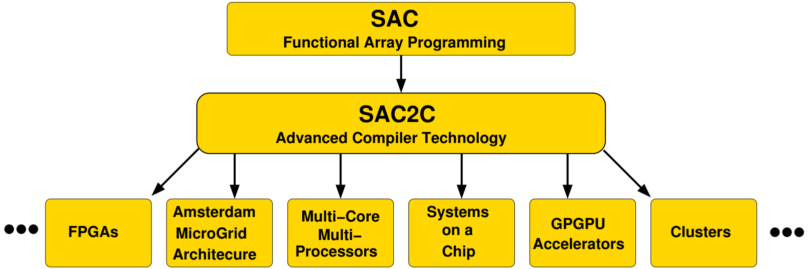
```
subroutine cffts3(is, d, x, xout, y)
implicit none
include 'global.h'
integer is, d(3), logd(3)
double complex x(d(1),d(2),d(3))
double complex xout(d(1),d(2),d(3))
double complex y(fftblockpad, d(3), 2)
integer i, j, k, ii
do i = 1, 3
   logd(i) = ilog2(d(i))
end do
do j = 1, d(2)
  do ii = 0, d(1) - 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
```

The Power of Abstraction

- Programming by composition of building blocks:
  - Rapid prototyping
  - High confidence in correctness
  - Good readability of code
  - Plenty of code reuse opportunities
- Opportunities for compiler and runtime system:
  - Target-independent optimisation
  - Code generation for variety of target architectures
  - Automatic parallelisation
  - Automatic memory management
- Result:
  - Sufficient performance
  - For a range of architectures
  - Without extra effort

Compilation Challenge



And achieve reasonable performance....

Goal: achieve 90% with 10% of the effort

SAC as a Compiler Technology Project

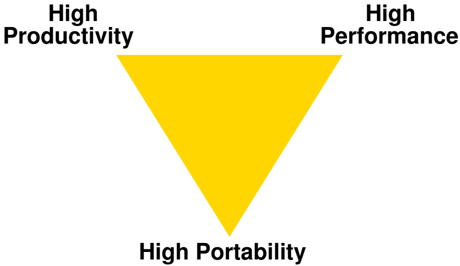
Some Figures:

- **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

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)

Conclusion



Single Assignment C:

- reconcile productivity, portability and performance
- one source for all architectures
- functional array programming
- exposure of fine-grained concurrency
- automatically sequentialising compiler
- automatic resource management

www.sac-home.org