Resource-agnostic Programming of Heterogeneous Architectures with Single Assignment C (SAC)

Clemens Grelck

Universiteit van Amsterdam

(Joint work with Miguel Diogo, MSc, and the whole SAC Team)

4th Distributed ASCI Supercomputer Workshop
Delft, Netherlands
February 13, 2013
The Trend on DAS and beyond: Heterogeneity

DAS-1 (1997)
- 200-Mhz Pentium Pro nodes

DAS-2 (2002)
- Dual 1-GHz Pentium III nodes

DAS-3 (2007)
- Dual AMD-Opteron nodes
- Some single core, some dual core
- 2.2-GHz 2.4-GHz 2.6-GHz

DAS-4 (2011)
- ?
Computing in the Age of Multi- and Many-Core

Compute node hardware is a zoo:

- Vastly different numbers of cores
- Vastly different memory architectures
- Accelerator cards: GPGPUs, Xeon Phi
- Heterogeneous chip architectures: AMD Fusion
Computing in the Age of Multi- and Many-Core

Compute node hardware is a zoo:

- Vastly different numbers of cores
- Vastly different memory architectures
- Accelerator cards: GPGPUs, Xeon Phi
- Heterogeneous chip architectures: AMD Fusion

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

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

Clemens Grelck, Universiteit van Amsterdam

Programming Heterogeneous Architectures with SAC
SAC — Design Space

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

SAC

Clemens Grelck, Universiteit van Amsterdam
Programming Heterogeneous Architectures with 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)

```c
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

Clemens Grelck, Universiteit van Amsterdam
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

```c
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

Clemens Grelck, Universiteit van Amsterdam
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

Clemens Grelck, Universiteit van Amsterdam
Case Study: 3-Dimensional Complex FFT (NAS-FT)

Algorithmic idea:

Implementation:

```c
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;

Clemens Grelck, Universiteit van Amsterdam
Programming Heterogeneous Architectures with SAC
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, n) 
  if (dir.eq.1) then 
    call cffts1(is, d1) 
    call cffts2(is, d1, x1, x2) 
    call cffts3(is, d1, x1, x2) 
    call cffts3(-1, d1, x1, x2) 
    endif 
  return 
  enddo 
end subroutine fft 
```

```
subroutine cffts1(is, d1, x1, x2) 
  implicit none 
  include 'global.h' 
  integer is, d1 
  do i = 1, d1 
    xout(i+1, j) = x(i, j) 
  enddo 
end subroutine cffts1 
```

```
subroutine cffts2(is, d1, d2, x1, x2) 
  implicit none 
  include 'global.h' 
  integer is, d1, d2 
  do i = 1, d1 
    do j = 1, d2 
      xout(i, j, k) = x(i, j) 
    enddo 
  enddo 
end subroutine cffts2 
```

```
subroutine cffts3(is, d1, x1, x2) 
  implicit none 
  include 'global.h' 
  integer is, d1 
  do i = 1, d1 
    xout(i, j) = x(i, j) 
  enddo 
end subroutine cffts3 
```

```
if (1.eq.m) goto 160 
```

```
call cfftz(is, logd1) 
subroutine cffts3(is, d1, x1, x2) 
  implicit none 
  include 'global.h' 
  integer is, d1 
  do j = 1, d1 
    y(j, i) = x(j, i) 
  enddo 
end subroutine cffts3 
```

```
if (1.eq.m) goto 160 
```

```
call cffts2(is, 1, 1, m, n, ff 
  double complex scratch(fftblockpad, n) 
  double complex xout(d1, d2, d3) 
  double complex yout(d1, d2, d3) 
  integer i, j, k, li 
  do i = 1, d1 
    do j = 1, d2 
      y(i, j, k) = x(i, j) 
    enddo 
  enddo 
end subroutine cffts2 
```

```
double complex xout(d1, d2, d3) 
  do i = 1, d1 
    double complex yout(d1, d2, d3) 
    do j = 1, d2 
      y(i, j, k) = x(i, j) 
    enddo 
  enddo 
end subroutine cffts2 
```

```
double complex xout(d1, d2, d3) 
  do i = 1, d1 
    do j = 1, d2 
      y(i, j, k) = x(i, j) 
    enddo 
  enddo 
end subroutine cffts2 
```

```
end subroutine fft 
```

```
print *, 'End of program' 
end program main 
```
The Power of Abstraction

- Programming by composition of building blocks:
  - Rapid prototyping
  - Good readability of code
  - High confidence in correctness
  - 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
The Power of Abstraction

- Programming by composition of building blocks:
  - Rapid prototyping
  - Good readability of code
  - High confidence in correctness
  - 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
The Power of Abstraction

- Programming by composition of building blocks:
  - Rapid prototyping
  - Good readability of code
  - High confidence in correctness
  - 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
And achieve reasonable performance....

Goal: achieve 90% with 10% of the effort
Simultaneously use:

- Multiple CPU processors with multiple cores
- Multiple GPU accelerators
SAC Array Comprehensions: With-loops

Internal representation of any array operation:

\[ A = \text{with} \{ \]
\[ (\text{lower\_bound} \leq \text{idxvec} < \text{upper\_bound}) : \text{expr} ; \]
\[ \ldots \]
\[ (\text{lower\_bound} \leq \text{idxvec} < \text{upper\_bound}) : \text{expr} ; \]
\[ \} : \text{genarray}(\text{shape}, \text{default}) \]

Characteristics:

- Disjoint partitions defined by *generators*
- Induction variable: *idxvec*
- Partitions associated with expression
- Periodically sliced and interleaved partitions
Parallelisation of With-Loops

**General approach:**

- **CPU**: Offload model with additional worker threads
- **GPU**: Offload model with dedicated kernel
Parallelisation of With-Loops

General approach:

- **CPU**: Offload model with additional worker threads
- **GPU**: Offload model with dedicated kernel

Compilation process:

- **CPU**: Intertwine generators for canonical order processing
- **GPU**: Turn each generator into one kernel
Parallelisation of With-Loops

General approach:

- **CPU**: Offload model with additional worker threads
- **GPU**: Offload model with dedicated kernel

Compilation process:

- **CPU**: Intertwine generators for canonical order processing
- **GPU**: Turn each generator into one kernel

Solution for heterogeneous architectures:

- Follow both compilation paths at the same time
- Maintain two different versions of each with-loop
Host vs Device Memories

Existing compilation paths:

- **CPU**: everything in host memory
- **GPU**: whole arrays either in host or device memory
- **GPU**: automatic transfers between host and device memory
Host vs Device Memories

Existing compilation paths:

- **CPU**: everything in host memory
- **GPU**: whole arrays either in host or device memory
- **GPU**: automatic transfers between host and device memory

Heterogeneous computing:

- **Distributed arrays** partially stored in each memory
- Control structure to track what data is available where
- Automatic transfers of array slices as necessary
- Decision when transfer is made at runtime
- Caching of array slices in multiple memories
Runtime Organisation

**Host threads:**

- 1 master thread executes sequential parts of code on host
- k worker threads run parallel computations offloaded to host
- 1 controller thread per accelerator controls computations offloaded to that accelerator
Runtime Organisation

Host threads:
▶ 1 master thread executes sequential parts of code on host
▶ k worker threads run parallel computations offloaded to host
▶ 1 controller thread per accelerator controls computations offloaded to that accelerator

Scheduling:
▶ Existing scheduling techniques divide iteration space between threads: static, dynamic, etc
Runtime Organisation

**Host threads:**
- 1 master thread executes sequential parts of code on host
- k worker threads run parallel computations offloaded to host
- 1 controller thread per accelerator controls computations offloaded to that accelerator

**Scheduling:**
- Existing scheduling techniques divide iteration space between threads: static, dynamic, etc
- Observation: GPU threads generally compute much faster than CPU threads
- Idea: monitor and statistically analyse relative performance between GPUs and CPUs, then divide iteration space accordingly
- Use dynamic scheduling within peer groups
- Implementation now: choose for experimentation
Experimental Evaluation

Experimental Setting:

- DAS-4 cluster
- Dual-GPGPU Machine:
  - 2 hexa-core 2.67 GHz CPUs
  - 2 NVidia GTX480 GPGPUs
- Multi-GPGPU Machine:
  - 2 quad-core 2.4 GHz CPUs
  - 8 NVidia GTX580 GPGPUs
Example: Relaxation with Fixed Boundary Conditions

Main computation in intermediate representation:

```plaintext
for (i=0; i<iter; i++) {
    A = with {
        ([0,0] <= x < shape(A)) :
            0.25 * ( A[x+[1,0]] + A[x-[1,0]]
                     + A[x+[0,1]] + A[x-[0,1]]);
    } : modarray( A);
}
```

Memory organisation:

![Diagram showing memory organisation]
Relaxation with Fixed Boundary Conditions

Dual-GPGPU machine, 6000x6000 matrix, 2000 iterations.

Clemens Grelck, Universiteit van Amsterdam
Programming Heterogeneous Architectures with SAC
Relaxation with Fixed Boundary Conditions

Multi-GPGPU machine, 8000x8000 matrix, 10000 iterations.

![Graph showing speedup vs. number of GPGPUs]

- 1 GPGPU: 1.01
- 2 GPGPUs: 1.82
- 4 GPGPUs: 2.96
- 8 GPGPUs: 3.9

Clemens Grelck, Universiteit van Amsterdam
Programming Heterogeneous Architectures with SAC
Conclusion

**Single Assignment C:**
- reconcile productivity, portability and performance
- one source for all architectures
- exposure of fine-grained concurrency
- automatically sequentialising compiler
- automatic resource management
- automatic exploitation of heterogeneous architectures

www.sac-home.org
Conclusion

High Productivity

High Performance

High Portability

Single Assignment C:
▶ reconcile productivity, portability and performance
▶ one source for all architectures
▶ exposure of fine-grained concurrency
▶ automatically sequentialising compiler
▶ automatic resource management
▶ automatic exploitation of heterogeneous architectures
▶ Is it worth it?

www.sac-home.org
Conclusion

Single Assignment C:
- reconcile productivity, portability and performance
- one source for all architectures
- exposure of fine-grained concurrency
- automatically sequentialising compiler
- automatic resource management
- automatic exploitation of heterogeneous architectures
- Is it worth it?
- Future work: expand distributed arrays to clusters

www.sac-home.org