Advances in Dynamic Compilation for Functional Data Parallel Array Processing

Clemens Grelck, Heinz Wiesinger

Universiteit van Amsterdam

3rd HiPEAC Workshop on
Dynamic Compilation Everywhere
DCE 2014
Wien, Austria
January 21, 2014

What is ...

Functional Data Parallel Array Processing ?

- Programming with multidimensional, immutable arrays
- Abstract from structural properties of arrays
- Treat arrays as abstract values
- Storage, layout, operations, ... all implicit

Single Assignment C (SAC)

Language:

- Purely functional programming language
- Generic data-parallel array processing
- All arrays immutable
- Syntax imitates ISO C
    - Assignment sequences (→ let-expressions)
    - Branches (→ conditional expressions)
    - Loops (→ tail-recursive functions)

Single Assignment C (SAC)

Language:

- Purely functional programming language
- Generic data-parallel array processing
- All arrays immutable
- Syntax imitates ISO C
  - Assignment sequences (→ let-expressions)
  - Branches (→ conditional expressions)
  - Loops (→ tail-recursive functions)

Compiler:

- Highly optimising compiler
- Performance competitive with Fortran or C
- Automatic parallelisation for
  - Symmetric multicore multiprocessors
  - Graphics accelerators
  - Heterogeneous systems

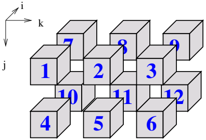
Multidimensional Array Calculus: Rank and Shape

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

rank: 1
shape: [6]

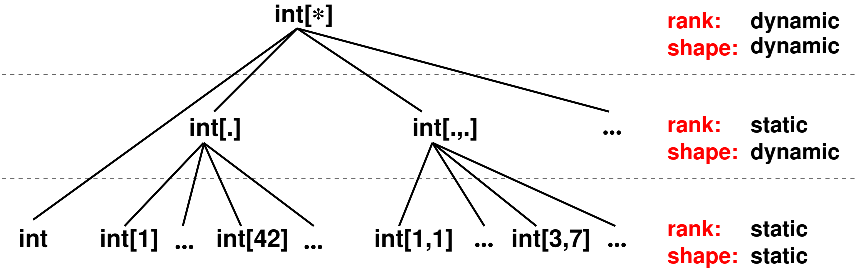
$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

rank: 2
shape: [3,3]

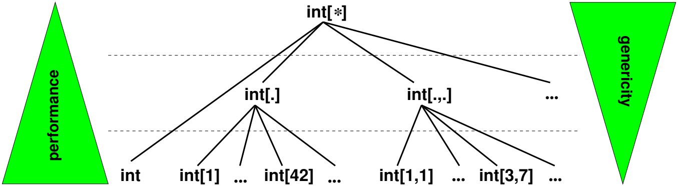


rank: 3
shape: [2,2,3]

Shapely Array Type Hierarchy



Genericity vs Performance Trade-Off



What software engineering principles demand:

- rank- and shpahe-generic programs
- wide-spread applicability
- code reuse

What the machine needs for performance:

- code customised to processed data
- exploit compile time knowledge for optimisation
- overcome abstraction boundaries

How Can We Reconcile Genericity and Performance ?

Observation:

- Often small numbers of different shapes prevail.
- Specialisation for concrete ranks and shapes !!
- Effectively apply large-scale static optimisation !!

How Can We Reconcile Genericity and Performance ?

Observation:

- Often small numbers of different shapes prevail.
- Specialisation for concrete ranks and shapes !!
- Effectively apply large-scale static optimisation !!

Limitations:

- Code obfuscation
- Arrays obtained from external source (e.g. file)
- Functions called from external environment (e.g. C code)

How Can We Reconcile Genericity and Performance ?

Observation:

- Often small numbers of different shapes prevail.
- Specialisation for concrete ranks and shapes !!
- Effectively apply large-scale static optimisation !!

Limitations:

- Code obfuscation
- Arrays obtained from external source (e.g. file)
- Functions called from external environment (e.g. C code)

Solution:

Dynamic Compilation to the rescue !

Solution: Adaptive Runtime Specialisation

Observations:

- Various compute cores available even in modest systems
- Even with linear speedups one or two cores less hardly matter

Solution: Adaptive Runtime Specialisation

Observations:

- Various compute cores available even in modest systems
- Even with linear speedups one or two cores less hardly matter

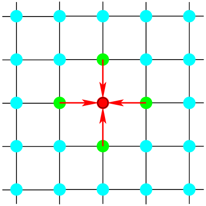
Idea:

- Set one core aside for dynamic code adaptation
- Incrementally generate more efficient code as shape information becomes available
- Accumulate adapted code in running process through dynamic linking
- Use adapted code as soon as available through dynamic dispatch

Case Study: Convolution with Convergence Check

Algorithmic principle:

Iteratively compute the weighted sums of neighbouring elements with cyclic neighbourhoods and dynamic convergence check

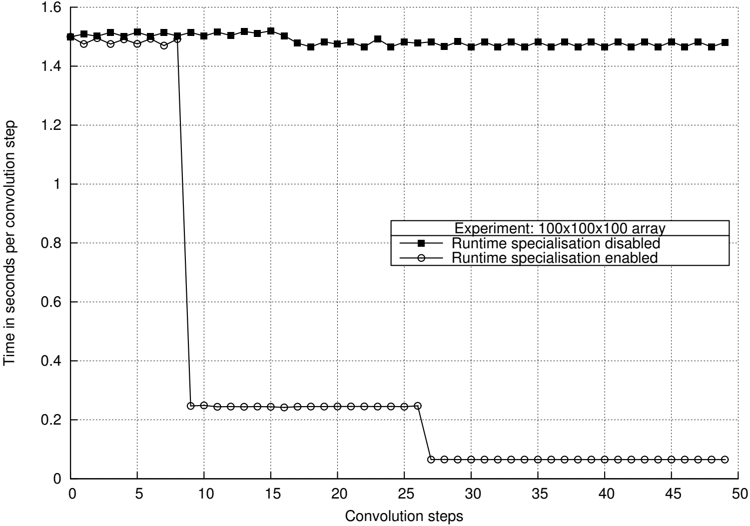


Implementation in SAC:

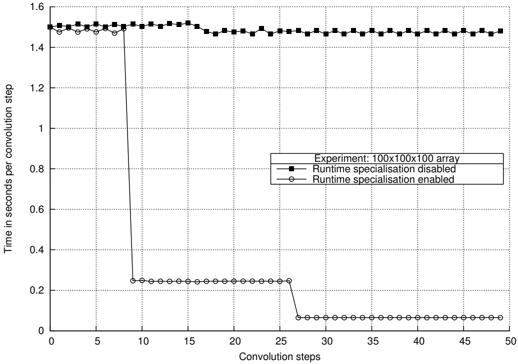
```
double[*] convolution (double[*] A, double eps)
{
  do {
    A_old = A;
    A = convolution_step( A_old);
  }
  while (!is_convergent( A, A_old, eps));

  return A;
}
```

Evaluation Example: 3-d Convolution 100x100x100



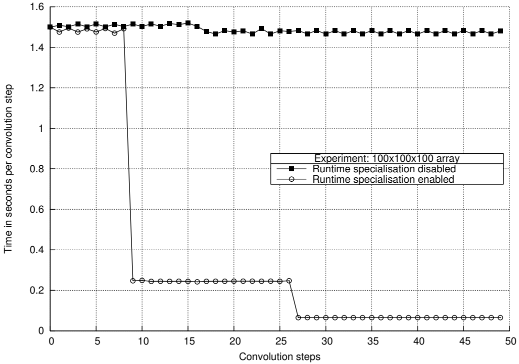
Availability of Adapted Code



Key question:

How can we speed up the availability of adapted code ?

Availability of Adapted Code

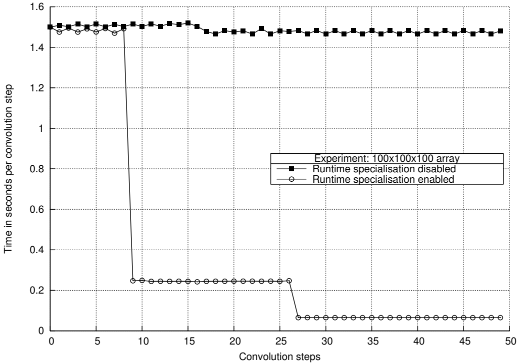


Key question:

How can we speed up the availability of adapted code ?

- Accelerate SAC compiler ?

Availability of Adapted Code

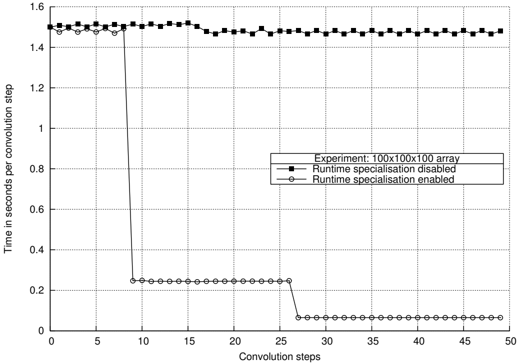


Key question:

How can we speed up the availability of adapted code ?

- Accelerate SAC compiler ?
- Hahaha .....

Availability of Adapted Code



Key question:

How can we speed up the availability of adapted code ?

- Accelerate SAC compiler ?
- Hahaha .....

Subject of this talk: 4 complementary proposals to speed up availability of adapted code

Idea 1: **Manifold** Asynchronous Adaptive Specialisation

Key observations:

- Specialisation requests come in in bursts
- Dynamic specialisation often exhibits a great deal of sharing among functions from same module
- Specialising two functions in conjunction may take the same time as each individually

Idea 1: **Manifold** Asynchronous Adaptive Specialisation

Key observations:

- Specialisation requests come in in bursts
- Dynamic specialisation often exhibits a great deal of sharing among functions from same module
- Specialising two functions in conjunction may take the same time as each individually

Our solution:

- Postpone triggering a specialisation by some (small) amount of time
- Expect more specialisation requests before cut-off time
- Specialise multiple functions together

Idea 2: **Prioritised** Asynchronous Adaptive Specialisation

Observation:

- Different specialisations yield different performance improvements
- Can we choose the (presumably) most effective one?

Idea 2: **Prioritised** Asynchronous Adaptive Specialisation

Observation:

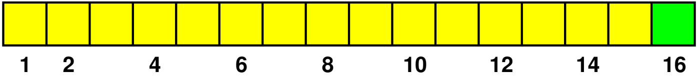
- Different specialisations yield different performance improvements
- Can we choose the (presumably) most effective one?

Solution:

- Turn specialisation request queue into priority queue
- Create buckets of functions from same module
- Gather statistical profiling data regarding effectiveness of specialisation

Idea 3: **Parallel** Asynchronous Adaptive Specialisation

First approach: dedicated specialisation core:

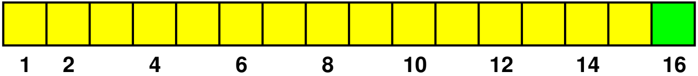


Key observations:

- Dynamic specialisations are time-consuming
- Adapted functions only become available with delay
- Insight 1: One specialisation core only is suboptimal

Idea 3: **Parallel** Asynchronous Adaptive Specialisation

First approach: dedicated specialisation core:

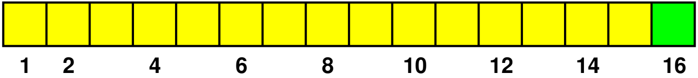


Key observations:

- Dynamic specialisations are time-consuming
- Adapted functions only become available with delay
- Insight 1: One specialisation core only is suboptimal
- Insight 2: Any fixed number of specialisation cores is suboptimal since specialisation competes with core application for resources

Idea 3: **Parallel** Asynchronous Adaptive Specialisation

First approach: dedicated specialisation core:

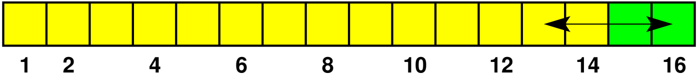


Key observations:

- Dynamic specialisations are time-consuming
- Adapted functions only become available with delay
- Insight 1: One specialisation core only is suboptimal
- Insight 2: Any fixed number of specialisation cores is suboptimal since specialisation competes with core application for resources

Solution:

- Dynamically adjust number of specialisation cores:



Idea 4: **Persistent** Asynchronous Adaptive Specialisation

Key observations:

- Dynamic code adaptation is for one program run
- Insight: the very same dynamic specialisations are built again and again

Idea 4: **Persistent** Asynchronous Adaptive Specialisation

Key observations:

- Dynamic code adaptation is for one program run
- Insight: the very same dynamic specialisations are built again and again

Solution:

- Store dynamic specialisations in installation-wide persistent storage
- Incrementally update the binary format of a module with new specialisations as they materialise
- Use replacement policies as in cache memories (e.g. least recently used)
- Learn which shapes are relevant.

Conclusion and Future Work

Conclusion:

- **Time to availability of specialisations is crucial !**
- We propose 4 complementary measures:
  - Manifold .....
  - Prioritised .....
  - Parallel .....
  - Persistent .....

..... asynchronous adaptive specialisation

Future work:

- Complete implementation(s)
- Conduct more case studies
- Do extensive evaluation
- Give a talk at DCE 2015