ExaSlang and the ExaStencils code generator

Sebastian Kuckuk\textsuperscript{1}, Christian Schmitt\textsuperscript{2}, Stefan Kronawitter\textsuperscript{3}
\textsuperscript{1} System Simulation, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
\textsuperscript{2} Hardware/Software Co-Design, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
\textsuperscript{3} Chair of Programming, University of Passau
PASC’15, Zurich, Switzerland; June 02, 2015
Outline

The ExaStencils DSL

Transformation Framework

Automatic Optimizations

Partitioning the Computational Domain(s)

Communication

Results

Conclusion & Future Work
The ExaStencils DSL
ExaSlang

- **ExaStencils language**
- Abstract description for generation of massively parallel geometric multigrid solvers
- Multi-layered structure → hierarchy of domain-specific languages (DSLs)
- Top-down approach: from abstract to concrete
- Very few mandatory specifications at one layer → room for decisions at lower layers based on domain knowledge
- External domain-specific language
  - better reflection of extensive ExaStencils approach
  - enables greater flexibility of different layers
  - eases tailoring of DSL layers to users
  - enables code generation for large variety of target platforms
- Parsing and code transformation framework implemented in Scala

---

ExaSlang: Multi-layered DSL Structure

Different layers of ExaSlang are tailored towards different users and knowledge.

Layer 1: Continuous Domain & Continuous Model
Layer 2: Discrete Domain & Discrete Model
Layer 3: Algorithmic Components & Parameters
Layer 4: Complete Program Specification

abstract problem formulation

Concrete solver implementation

Target Platform Description

Properties

- Procedural
- Statically typed
- External DSL
- Syntax partly inspired by Scala

Function Smoother@((coarsest + 1) to finest)(): Unit {
  communicate ghost of Solution[active]@current
  loop over fragments {
    loop over Solution@current {
      Solution[next]@current = Solution[active]@current
      + (omega * inverse(diag(Laplace@current))
      * (RHS@current - Laplace@current
         * Solution[active]@current))
    }
    advance Solution@current
  }
}
ExaSlang 4: Complete Program Specification

Properties

- Procedural
- Statically typed
- External DSL
- Syntax partly inspired by Scala

**Function** Smoother@((coarsest + 1) to finest)(): Unit {
  communicate ghost of Solution[active]@current
  loop over fragments {
    loop over Solution@current {
      Solution[next]@current = Solution[active]@current
        + (omega * inverse(diag(Laplace@current))
          * (RHS@current - Laplace@current
            * Solution[active]@current))
    }
    advance Solution@current
  }
}
ExaSlang 4: Example

Example: exit multigrid recursion

```
Function WCycle@((all, not(coarsest)))(()) : Unit {
    repeat 4 times {
        Smoother@current()
    }
    UpResidual@current()
    Restriction@current()
    SetSolution@coarser(0)
    repeat 2 times {
        Wcycle@coarser()
    }
    Correction@current()
    repeat 3 times {
        Smoother@current()
    }
}

Function WCycle@coarsest() : Unit {
    /* ...direct solving... */
}
```
Transformation Framework
Transformation Framework

Abstract workflow:

Algorithmic description → Parsing → Intermediate representation → Prettyprinting → C++ output
Transformation Framework

Current workflow

1. DSL input (Layer 4) is parsed
2. Parsed input is checked for errors and transformed into the IR
3. Many smaller, specialized transformations are applied
4. C++ output is prettyprinted
Transformation Framework

Current workflow

1. DSL input (Layer 4) is parsed
2. Parsed input is checked for errors and transformed into the IR
3. Many smaller, specialized transformations are applied
4. C++ output is prettyprinted

Concepts

- Major program modifications take place only in IR
- IR can be transformed to C++ code
- Small transformations can be enabled and arranged according to needs
Transformation Framework

Transformations

- Transform program state to another one
- Are applied to program state in depth-first order
- May be applied to only a part of the program state
- Are grouped together in strategies
Transformation Framework

Transformations

- Transform program state to another one
- Are applied to program state in depth-first order
- May be applied to only a part of the program state
- Are grouped together in strategies

Strategies

- Are applied in transactions
- Standard strategy that linearly executes all transformations is provided
- Custom strategies possible
Transformation Framework

Example transformations:

```javascript
var s = DefaultStrategy("example strategy")

// rename a certain stencil
s += Transformation("rename stencil", {
    case x : Stencil if(x.identifier == "foo") => {
        if(x.entries.length != 7) error("invalid stencil size")
        x.identifier = "bar"; x
    }
})

// evaluate additions
s += Transformation("eval adds", {
    case AdditionExpression(l : IntegerConstant, r : IntegerConstant) => IntegerConstant(l.value + r.value)
})

s.apply // execute transformations sequentially
```
Transformation Framework

Implemented workflow:

1. Algorithmic description
2. Parsing
3. IR
4. L4
5. IR
6. IR
7. IR...
8. IR
9. C++ output
10. Parsing
11. Prettyprinting
Automatic Optimizations
Example: Address Precalculation

Idea: precompute a maximally sized part of the index expression outside the loop

- Standard optimization in production compilers
- Not always applied, since other transformations can stand in its way

→ We implemented a more advanced version directly
Example: Address Precalculation

Idea: precompute a maximally sized part of the index expression outside the loop

- Standard optimization in production compilers
- Not always applied, since other transformations can stand in its way

⇒ We implemented a more advanced version directly

Example for \(a[x][y][z]\) and \(a[x+1][y+1][z+1]\)
Example: Address Precalculation

Idea: precompute a maximally sized part of the index expression outside the loop

- Standard optimization in production compilers
- Not always applied, since other transformations can stand in its way

→ We implemented a more advanced version directly

Example for $a[x][y][z]$ and $a[x+1][y+1][z+1]$

1. Linearize accesses
   
   $$
   a[ ( z \times 512 + y ) \times 512 + x ] \\
   a[ ( (z+1) \times 512 + (y+1) ) \times 512 + (x+1) ]
   $$
Example: Address Precalculation

Idea: precompute a maximally sized part of the index expression outside the loop

- Standard optimization in production compilers
- Not always applied, since other transformations can stand in its way

⇒ We implemented a more advanced version directly

Example for \(a[x][y][z]\) and \(a[x+1][y+1][z+1]\)

1. Linearize accesses
   \[
   \begin{align*}
   a[(z\times512 + y)\times512 + x] \\
   a[(z+1)\times512 + (y+1)]\times512 + (x+1)]
   \end{align*}
   \]

2. Simplify index expressions
   \[
   \begin{align*}
   a[z\times262144 + y\times512 + x] \\
   a[z\times262144 + y\times512 + x + 262657]
   \end{align*}
   \]
Example: Address Precalculation

Idea: precompute a maximally sized part of the index expression outside the loop

- Standard optimization in production compilers
- Not always applied, since other transformations can stand in its way

We implemented a more advanced version directly

Example for $a[x][y][z]$ and $a[x+1][y+1][z+1]$

1. Linearize accesses

   $a[(z \times 512 + y) \times 512 + x]$
   $a[((z+1) \times 512 + (y+1)) \times 512 + (x+1)]$

2. Simplify index expressions

   $a[z \times 262144 + y \times 512 + x]$
   $a[z \times 262144 + y \times 512 + x + 262657]$

3. Extract common subexpression

   $a_p = &a[z \times 262144 + y \times 512];$
   $for \ (int \ x = ..) \ { \$
     ..\ a_p[x] \ .$
     ..\ a_p[x + 262657] \ .$
   \}$
And more

- polyhedral optimizations with different granularity per level
  1. Extract models
  2. Compute dependences
  3. Eliminate dead statement instances
  4. Search an optimal schedule
  5. Tile dimensions
  6. Recreate AST

- loop unrolling
- arithmetic simplifications
- vectorization for SSE3, AVX, AVX2, QPX, (NEON)
- ...

S. Kuckuk | ExaStencils | ExaSlang and the ExaStencils code generator
Vectorization Example

for (int i = ..)
    sol0_p[i+76] = 0.2*(rhs_p[i]+sol1_p[i+4]+sol1_p[i+148]
                     +sol1_p[i+75]+sol1_p[i+76]+sol1_p[i+77]);

is transformed to

vector4double vec0 = vec_splats(0.2);
vector4double vec6 = vec_lda(0, &sol1_p[lower+72]);
vector4double vec8 = vec_lda(0, &sol1_p[lower+76]);
for (int i = lower; i < upper; i += 4) {
    vector4double vec1 = vec_lda(0, &rhs_p[i]);
    vector4double vec2 = vec_lda(0, &sol1_p[i+4]);
    vector4double vec3 = vec_lda(0, &sol1_p[i+148]);
    vector4double vec5 = vec6;
    vec6 = vec8;
    vector4double vec4 = vec_sldw(vec5, vec6, 3);
    vec8 = vec_lda(0, &sol1_p[i+80]);
    vector4double vec7 = vec_sldw(vec6, vec8, 1);
    vector4double vec9 = vec_madd(vec0,
                                    vec_add(vec_add(vec_add(vec_add(
                      vec1, vec2), vec3), vec4), vec7), vec6);
    vec_sta(vec9, 0, &sol0_p[i+76]);
}
Partitioning the Computational Domain(s)
Domain Partitioning - Our Scope

- Uniform grids
- Block-Structured grids
Domain Partitioning - Concept

- Easy for regular domains

Each **domain** consists of one or more **blocks**

Each **block** consists of one or more **fragments**

Each **fragment** consists of several data points / cells

- More complicated for HHG
Domain Partitioning - Mapping to Parallelism

- Domain partition maps directly to different parallelization interfaces, e.g. MPI and OMP:
  - Each **block** corresponds to one *MPI* rank
  - Each **fragment** corresponds to one *OMP* rank
  - Hybrid *MPI/OMP* corresponds to multiple **blocks** and multiple **fragments** per **block**
  - Alternatively: only one **fragment** per **block** and direct parallelization of kernels with *OMP*

- Easy to map to different interfaces, e.g.
  - PGAS
  - MPI and PGAS
  - MPI and CUDA
Domain Partitioning - User Interface

• All domains are specified in Layer 4

/* embedded domains */
Domain global< [ -1, -1, -1 ] to [ 1, 1, 1 ] >
Domain sthSmaller< [ -0.5, -0.5, -1 ] to [ 0.5, 0.5, 1 ] >

/* non-regular shapes */
Domain global< [ 0, 0 ] to [ 2, 2 ] >
Domain lShape< [ 0, 0 ] to [ 1, 1 ]
and [ 0, 1 ] to [ 1, 2 ]
and [ 1, 0 ] to [ 2, 1 ] >

Domain moreComplex from file 'mydomain.exa'

• Actual partition of the domains is specified through the number of fragments in each dimension
• If possible, domain is not loaded from file but our framework generates code to determine req. information at (solver) runtime
Communication
Communication - User Interface

- Communication statements are added automatically when transforming Layer 3 to Layer 4 where they may be reviewed or adapted.
- Ghost and duplicate layers may be synchronized separately or collectively.

```c
/* communicates all applicable layers */
communicate Solution@current
/* communicates only ghost layers */
communicate ghost of Solution[active]@current
/* communicates duplicate and first two ghost layers */
communicate dup, ghost [ 0, 1 ] of Solution[active]@current
/* asynchronous communicate */
begin communicate Residual@current
...
finish communicating Residual@current
```

- Basic (Layer 4) communicate statements are synchronous with respect to the computations.
- Actual realization, i.e. usage of synchronous and/ or asynchronous MPI operations is up to the generator.
Node progression inside our framework is similar to this:

- communicate Solution@current
- ExchangeDataFunction
- FunctionCall
- RemoteSends
- LocalSends
- RemoteRecvs
- LoopOverFragments
- ConditionStatement(iv.IsValidForSubdomain(…))
- CopyToSendBuffer
  - RemoteSend
  - WaitForTransfer
- LoopOverDimensions
  - Direct Copy
- CopyFromRecvBuffer
  - RemoteRecv
  - WaitForTransfer
- RemoteSends
- LocalSends
- RemoteRecvs

for duplicate and/or ghost

for each neighbor
Benchmark Problem and System

- **Target system**
  - JUQUEEN supercomputer located in Jülich, Germany
  - 458,752 cores / 28,672 nodes (1.6 GHz, 16 cores each, four-way multithreading)

- **Regarded problem**
  - 3D finite differences discretization of Poisson’s equation ($\Delta \phi = f$) with Dirichlet boundary conditions
  - V(3,3) cycle, parallel CG as coarse grid solver
  - Jacobi, Gauss-Seidel or red-black Gauss-Seidel smoother
  - pure MPI or hybrid MPI/OMP parallelization
  - 64 threads per node, roughly $10^6$ unknowns per core
  - code optimized through polyhedral loop transformations, 2-way unrolling and address precalculation on finer levels as well as custom MPI data types
  - vectorization and blocking are not yet taken into account
Weak Scalability

- Mean time per V-cycle
- $V(3,3)$ with Jacobi and CG

![Graph showing weak scalability](graph.png)
ExaStencils Framework: Comparison of Lines of Code

- **ExaSlang 4**
  - Jacobi: 244
  - Gauss-Seidel: 236
  - Red-Black GS: 240

- **C++ Pure MPI**
  - Jacobi: 11,259
  - Gauss-Seidel: 9,600
  - Red-Black GS: 9,776

- **C++ Hybrid MPI/OMP**
  - Jacobi: 13,432
  - Gauss-Seidel: 11,320
  - Red-Black GS: 12,887
ExaStencils Framework: Program Sizes during Transformation

![Graph showing program sizes during transformation.](image)
Conclusion

- Hierarchy of languages for generating massively parallel geometric multigrid solvers
- Automatic target-specific refinements and optimizations
- Automatic vectorization and polyhedral optimization
- Flexible transformation framework for implementation of external DSLs
- Code generation for a variety of target platforms
Future Work

- Implementation of ExaSlang layers 1 to 3
- Refinement of TPDL
- Variant generation and exploration
- Support for multi-colored kernels
- (Multi-)GPU support
- PGAS performance evaluation
- Applications
Thanks for listening. Questions?

ExaStencils – Advanced Stencil Code Engineering
http://www.exastencils.org
ExaStencils is funded by the German Research Foundation (DFG) as part of the Priority Program 1648 (Software for Exascale Computing).