ExaSlang and the ExaStencils code generator

Christian Schmitt¹, Stefan Kronawitter², Sebastian Kuckuk³, Frank Hannig¹, Jürgen Teich¹, Christian Lengauer², Harald Köstler³, Ulrich Rüde³
¹ Hardware/Software Co-Design, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
² Chair of Programming, University of Passau
³ System Simulation, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

Seminar Advanced Stencil-Code Engineering, Schloss Dagstuhl; April 14, 2015
Outline

The ExaStencils DSL

Transformation Framework

Polyhedral Optimizations

Traditional Optimizations

Partitioning the Computational Domain(s)

Communication

Results

Conclusion & Future Work
Overall goal

It’s all about simplicity!

Randall Munroe.  *xkcd: Manuals*. Licensed under Creative Commons Attribution-NonCommercial 2.5 License. 2014. URL: http://xkcd.com/1343/
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

Target Platform Description

abstract
problem
formulation

concrete
solver
implementation

ExaSlang: Layers

Continuous Domain & Continuous Model (Layer 1)

Specification of
- size and structure of computational domain
- variables
- functions and operators (pre-defined functions and operators also available)
- mathematical problem

Discrete Domain & Discrete Model (Layer 2)

Discretization of
- computational domain into fragments (e.g., triangles)
- variables to fields
  - specification of data types
  - selection of discretized location (cell-based or node-based)

Transformation of energy functional to PDE or weak form
ExaSlang: Layers

Algorithmic Components & Parameters (Layer 3)

Specification of

• mathematical operators
• multigrid components (e.g., selection of smoother)
• operations in matrix notation

Complete Program Specification (Layer 4)

Specification of

• complete multigrid V-cycle, or
• custom cycle types
• operations depending on the multigrid level
• loops over computational domain
• communication and data exchange
• interface to 3rd-party code
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
  }
}
Properties

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

```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: Level Specifications**

Multigrid is inherently hierarchical and recursive

→ We need
  - multigrid recursion exit condition
  - access to other levels’ data & functions

→ Additionally, we want
  - relative addressing
  - aliases for certain levels
  - variable definitions per level

**Implementation**

- Numerical values, e.g., @0 for bottom level
- Aliases, e.g., @all, @current, @coarser, @coarsest
- Simple expressions, e.g., @coarsest + 1
- Lists, e.g., @(1, 3, 5)
- Ranges, e.g., @(1 to 5)
- Negations, e.g., @(1 to 5, not(3))
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

Using a simple 1-step concept, we can do some refinements, e.g.,

```java
loop over Solution {
    // ....
}
```

can be processed to

```java
for (int z = start_z; z < stop_z; z += 1) {
    for (int y = start_y; y < stop_y; y += 1) {
        for (int x = start_x; x < stop_x; x += 1) {
            // ....
        }
    }
}
```
Transformation Framework

Using a simple 1-step concept, we can do some refinements, e.g.,

```c
loop over Solution {
    // ....
}
```

can be processed to

```c
for (int z = start_z; z < stop_z; z += 1) {
    for (int y = start_y; y < stop_y; y += 1) {
        for (int x = start_x; x < stop_x; x += 1) {
            // ....
        }
    }
}
```


→ Very cumbersome with 1-step approach. Need something more flexible!
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
- Central instance keeps track of generated program: StateManager
- Variant generation by duplicating program at different transformation stages
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

Transactions

- Before execution, a snapshot of the program state is made
- May be committed or aborted

Checkpoints

- A copy of program state during compilation
- Restoration of program states
- Acceleration of variant generation for design space exploration
Transformation Framework

Example transformations:

```plaintext
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:

Algorithmic description

Parsing

Prettyprinting

C++ output

IR
L4
IR
IR
IR
IR
Polyhedral Optimizations
Motivation

Jacobi smoother

![Diagram showing input and result for Jacobi smoother](image-url)
Motivation

Jacobi smoother
Motivation

Jacobi smoother

input

result

x

y
Motivation

Jacobi smoother – temporal blocking
Motivation

Jacobi smoother – temporal blocking

\[ x \]

\[ y \]

input \quad \text{intermediate} \quad \text{result}
Motivation

Jacobi smoother – temporal blocking
Polyhedron Model

```c
for (int i=1; i<=n; ++i)
    for (int j=1; j<=n-i+1; ++j)
        a[i][j] =
            a[i-1][j] + a[i][j-1];
```
Polyhedron Model

```c
for (int i=1; i<=n; ++i)
    for (int j=1; j<=n-i+1; ++j)
        a[i][j] =
            a[i-1][j] + a[i][j-1];
```

Iteration domain

1 ≤ i ≤ n
1 ≤ j ≤ n − i + 1
for (int i=1; i<=n; ++i)
    for (int j=1; j<=n-i+1; ++j)
        a[i][j] =
            a[i-1][j] + a[i][j-1];

Iteration domain

\[
\begin{align*}
1 \leq i & \leq n \\
1 \leq j & \leq n - i + 1
\end{align*}
\]

Dependences

\[
\begin{align*}
(i,j) & \rightarrow (i+1,j) \\
(i,j) & \rightarrow (i,j+1)
\end{align*}
\]
Polyhedron Model

```c
for (int i=1; i<=n; ++i)
    for (int j=1; j<=n-i+1; ++j)
        a[i][j] =
            a[i-1][j] + a[i][j-1];
```

Iteration domain

![Iteration domain diagram]

1 ≤ i ≤ n
1 ≤ j ≤ n − i + 1

Transformation

\[ t = i + j - 1 \]
\[ p = j \]

Dependences

\[ (i, j) \rightarrow (i+1, j) \]
\[ (i, j) \rightarrow (i, j+1) \]

\[ (t, p) \rightarrow (t+1, p) \]
\[ (t, p) \rightarrow (t+1, p+1) \]
Polyhedron Model

for (int i=1; i<=n; ++i)
    for (int j=1; j<=n-i+1; ++j)
        a[i][j] =
            a[i-1][j] + a[i][j-1];

for (int t=1; t<=n; ++t)
    #pragma omp parallel for
    for (int p=1; p<=t; ++p)
        a[t-p+1][p] = ..;

Iteration domain

\[
\begin{align*}
1 & \leq i \leq n \\
1 & \leq j \leq n - i + 1
\end{align*}
\]

Transformation

\[
\begin{align*}
t & = i + j - 1 \\
p & = j
\end{align*}
\]

\[
\begin{align*}
1 & \leq t \leq n \\
1 & \leq p \leq t
\end{align*}
\]

Dependences

\[
\begin{align*}
(i,j) & \rightarrow (i+1,j) \\
(i,j) & \rightarrow (i,j+1)
\end{align*}
\]

\[
\begin{align*}
(t,p) & \rightarrow (t+1,p) \\
(t,p) & \rightarrow (t+1,p+1)
\end{align*}
\]
Constraints on Input

Data structures

- Only scalars and arrays allowed
- Alias information must be available
Constraints on Input

Data structures

- Only scalars and arrays allowed
- Alias information must be available

Loop bounds and array subscripts

Must be affine in surrounding loop variables and parameters

\[ \text{Iteration domain is } \mathbb{Z}\text{-polyhedron} \]

```c
for (int i=0; i<n; ++i)
    for (int j=0; j<=i; ++j) {
        a[(i*i+i)/2+j] = // linearized triangle
        b[2*j][i-j];
        c[n*i] = d[c[j]];
    }
```

Marked parts are not allowed
Model Extraction

Iteration domain

- Described by a single loop over <field>
- No need to deal with nested loops
- Subsequent loops may be merged
Model Extraction

Iteration domain

• Described by a single loop over <field>
• No need to deal with nested loops
• Subsequent loops may be merged

Memory accesses

Modelling takes place before accesses are linearized
→ Allows modelling code using, e.g., triangular fields

\[ a[(y*y+y)/2+x] \]
vs.
\[ a[y][x] \]
Model Optimization

Optimization steps

1. Compute dependences
2. Eliminate dead statement instances
3. Search an optimal schedule
   - use generic input to scheduler, or
   - preprocess input to obtain better results for stencil codes
4. Tile dimensions
5. Recreate AST
Model Optimization

Optimization steps

1. Compute dependences
2. Eliminate dead statement instances
3. Search an optimal schedule
   - use generic input to scheduler, or
   - preprocess input to obtain better results for stencil codes, or
   - perform a complete search space exploration
4. Tile dimensions
5. Recreate AST
Model Optimization

Optimization steps

1. Compute dependences
2. Eliminate dead statement instances
3. Search an optimal schedule
   - use generic input to scheduler, or
   - preprocess input to obtain better results for stencil codes, or
   - perform a complete search space exploration
4. Tile dimensions
5. Recreate AST

Optimization levels

0. Disable polyhedral optimizations
1. Perform dependence analysis only
2. Optimize schedule, but do not tile
3. All
Traditional Optimizations
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

1. Linearize accesses
   \[ a\{x\}{y\}{z} \]
   \[ a\{H z *U1R + y I *U1R + x\} \]

2. Simplify index expressions
   \[ a\{z *R6R1TT + y *U1R\} \]
   \[ a\{z *R6R1TT + y *U1R + x + R6R6UW\} \]

3. Extract common subexpression
   \[ a_p \] Fa\{z *R6R1TT + y *U1R\}[a_p \{x\}] for H int x \] NNI {NN a_p \{x\} NN a_p \{x + R6R6UW\} NN}
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
   
   $a[z \cdot R6R1TT + y \cdot U1R + x]$
   
   $a[z \cdot R6R1TT + y \cdot U1R + x + R6R6UW]$

2. Simplify index expressions

3. Extract common subexpression
   
   $a_p[f_a[z \cdot R6R1TT + y \cdot U1R]]$
   
   for $\text{int } x \text{ NNI }$
   
   $a_p[x] \text{ NNI}$
   
   $a_p[x + R6R6UW] \text{ NNI}$
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

\[
\begin{align*}
    a[(z \times 512 + y) \times 512 + x] \\
    a[((z+1) \times 512 + (y+1)) \times 512 + (x+1)]
\end{align*}
\]
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
   
   $a[\text{z * 512 + y * 512 + x}]$
   $a[\text{(z+1) * 512 + (y+1) * 512 + (x+1)}]$

2. Simplify index expressions
   
   $a[\text{z * 262144 + y * 512 + x}]$
   $a[\text{z * 262144 + y * 512 + x + 262657}]$
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 = ..) {
      .. $a_p[x]$ ..
      .. $a_p[x + 262657]$ ..
   }
Arithmetic Simplifications

Optimizations

- Convert division by a constant into multiplication by its inverse
- Evaluate subexpressions as far as possible
- Apply law of distributivity in order to
  - factor out repeated loads of the same array element
  - reduce the number of multiplications required

Example

\[ 2.0 \times (a[i]/2.0 + a[i+1]/4.0 + a[i-1]/4.0 - a[i]) \]

is transformed to

\[ -a[i] + 0.5 \times (a[i+1] + a[i-1]) \]
Vectorization

- Using vector units is mandatory to achieve best performance
- Contemporary compilers are unable to emit efficient vector code

⇒ Explicit vectorization during generation
Vectorization

- Using vector units is mandatory to achieve best performance
- Contemporary compilers are unable to emit efficient vector code

→ Explicit vectorization during generation

Advantages

- Ensure data alignment
- Evaluate multiple instruction combinations
- Usage of data dependence analysis from polyhedral optimizations
Vectorization

- Using vector units is mandatory to achieve best performance
- Contemporary compilers are unable to emit efficient vector code

→ Explicit vectorization during generation

Advantages

- Ensure data alignment
- Evaluate multiple instruction combinations
- Usage of data dependence analysis from polyhedral optimizations

Supported vector units

SSE3, AVX, AVX2, QPX
Vectorization

• Using vector units is mandatory to achieve best performance
• Contemporary compilers are unable to emit efficient vector code

→ Explicit vectorization during generation

Advantages

• Ensure data alignment
• Evaluate multiple instruction combinations
• Usage of data dependence analysis from polyhedral optimizations

Supported vector units

SSE3, AVX, AVX2, QPX, (NEON)
Vectorization Example

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

is transformed to

```c
vector4double vec0 = vec_splats(0.2);
vector4double vec6 = vec_lda(0, &soli_p[lower+72]);
vector4double vec8 = vec_lda(0, &soli_p[lower+76]);
for (int i = lower; i < upper; i += 4) {
    vector4double vec1 = vec_lda(0, &rhs_p[i]);
    vector4double vec2 = vec_lda(0, &soli_p[i+4]);
    vector4double vec3 = vec_lda(0, &soli_p[i+148]);
    vector4double vec5 = vec6;
    vec6 = vec8;
    vector4double vec4 = vec_sldw(vec5, vec6, 3);
    vec8 = vec_lda(0, &soli_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_staa(vec9, 0, &sol0_p[i+76]);
}
```
Loop unrolling

Idea

Duplicate loop body to reduce branch penalty, condition evaluation, ...

```plaintext
for (i = ..; ..; i += 1)
    { S(i); T(i); }
```
Loop unrolling

Idea

Duplicate loop body to reduce branch penalty, condition evaluation, ...

```c
for (i = ..; ..; i += 1)
    { S(i); T(i); }
```

Two unrolling modes supported:

- Duplicate the whole body at once

```c
for (i = ..; ..; i += 2)
    { S(i); T(i); S(i+1); T(i+1); }
```
Loop unrolling

Idea

Duplicate loop body to reduce branch penalty, condition evaluation, ...

```
for (i = ..; ..; i += 1)
{ S(i); T(i); }
```

Two unrolling modes supported:

- Duplicate the whole body at once
  ```
  for (i = ..; ..; i += 2)
  { S(i); T(i); S(i+1); T(i+1); }
  ```

- Duplicate each statement in-place
  - requires renaming of local variables
  - loop must be parallel
  - could be better on in-order architectures

  ```
  for (i = ..; ..; i += 2)
  { S(i); S(i+1); T(i); T(i+1); }
  ```
Loop unrolling

Interpolation

Three fourth of the iterations of a 2D-interpolation perform semantically equivalent load operations:

```c
for (int y = ..; ..; y += 1)
    for (int x = ..; ..; x += 1)
        .. a[y/2][x/2]
        + a[y/2][x/2 + x%2]
        + a[y/2 + y%2][x/2]
        + a[y/2 + y%2][x/2 + x%2];
```

Number of unnecessary loads is even higher in 3D
Loop unrolling

Interpolation

Three fourth of the iterations of a 2D-interpolation perform semantically equivalent load operations:

```c
for (int y = ..; ..; y += 1)
    for (int x = ..; ..; x += 1)
        .. a[y/2][x/2]
        + a[y/2][x/2 + x%2]
        + a[y/2 + y%2][x/2]
        + a[y/2 + y%2][x/2 + x%2];
```

Number of unnecessary loads is even higher in 3D

Optimization

1. Unroll each loop in interpolation once
2. Precompute the value of $x\%2$ and $y\%2$
3. Factor out identical array accesses
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

```c
/* 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 - Requirements

• Data needs to be exchanged between different processes . . .
  • . . . locally and/or remotely (i.e. between fragments within and/or across blocks)
  • . . . with different patterns (e.g. 6P/26P in 3D)
  • . . . for specific regions in the grids (e.g. when using temporal blocking)
  • . . . for multiple data layouts
  • . . . using special construct such as MPI data types if reasonable
  • . . . asynchronously

• However, it is not feasible to . . .
  • . . . implement every possible case
  • . . . extensively use templates and defines
  • . . . trade variability for performance (e.g. using PGAS)
Communication - Layouts

- Fields (data mapped to fragments) are (logically) split into different regions: padding (P), ghost (G), duplicate (D) and inner (I) points.

- Duplicate points allow for intuitive mapping between levels.
Field layouts are declared in an intuitive way on Layer 4.

/* used for, e.g., the right hand side */
Layout NoComm< Real, Node >@all {
  ghostLayers = [ 0, 0, 0 ]
  duplicateLayers = [ 1, 1, 1 ]
}

/*@ used for, e.g., the solution and the residual */
Layout BasicComm< Real, Node >@all {
  ghostLayers = [ 1, 1, 1 ] with communication
  duplicateLayers = [ 1, 1, 1 ] with communication
}

/*@ used for fields with cell centered values and, e.g.,
temporal blocking or larger stencils */
Layout TempBlocking< Real, Cell >@all {
  ghostLayers = [ 3, 3, 3 ] with communication
  duplicateLayers = [ 0, 0, 0 ]
}
Communication - Field Declarations

- Field layouts are then used when declaring fields

```c
/* Field ident< domain, layout, bc >@level */

Field  RHS< global, NoComm, None >@all

Field  Residual< global, BasicComm, 0, 0 >@all

Field  Solution< global, BasicComm, 
0.0 >[2]@coarsest to (finest - 1))
Field  Solution< global, BasicComm, 
geometricCoordinate_x() * geometricCoordinate_y() >[2]@finest
```
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`
- Loop over fragments
- Function call
- Exchange data function
- Condition statement
  - `iv.IsValidForSubdomain(...)`
- For each neighbor, either remote sends or remote recvs
  - Remote sends
    - Copy to send buffer
    - Remote send
    - Wait for transfer
  - Remote receives
    - Copy from recv buffer
    - Remote recv
    - Wait for transfer

For duplicate and/or ghost elements.
Results
Node-Level Performance

3D 7-point Jacobi smoother

Intel IvyBridge EP

![Graph showing performance data for different core counts and MLUP/s values. The x-axis represents cores, ranging from 1 to 10, and the y-axis represents MLUP/s, ranging from 0 to 700. The graph includes a line marked 'no opts'.]
Node-Level Performance

3D 7-point Jacobi smoother

Intel IvyBridge EP

![Graph showing MLUP/s on the y-axis and Cores on the x-axis, with two lines for 'no opts' and 'poly only'.]
Node-Level Performance

3D 7-point Jacobi smoother

Intel IvyBridge EP

C. Schmitt, S. Kronawitter, S. Kuckuk | ExaStencils | ExaSlang and the ExaStencils code generator
Node-Level Performance

3D 7-point Jacobi smoother

Intel IvyBridge (consumer)

![Graph showing performance comparison between hand-tuned and generated methods.](image-url)
Node-Level Performance

3D 7-point Jacobi smoother

Intel IvyBridge (consumer)

generated: 8 LOC ExaSlang $\leftrightarrow$ 4215 LOC C++ (up to 1108 characters per line)
Showcase: Code Generation for FPGAs

Motivation

- Show flexibility of generator by adding external HLS tools into workflow
- Explore FPGAs as a viable alternative to standard accelerators

Mapping

- Resolve stencil applications per multigrid levels
- Map \texttt{loop over} statements to separate IP cores
- Dependency analysis: Add fields to IP core inputs/outputs
- Calculate field (stream) sizes for IP core
- Replace \texttt{loop over} statements with kernel statements
- Connect IP cores with streams and duplicate streams accordingly
- Add iteration intervals from simulation

## Results

### Resource usage on FPGAs for double precision

<table>
<thead>
<tr>
<th>FPGA</th>
<th>LUTs</th>
<th>FFs</th>
<th>DSPs</th>
<th>BRAMs</th>
<th>( F_{\text{max}} ) [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kintex-7</td>
<td>140%</td>
<td>43%</td>
<td>111%</td>
<td>124%</td>
<td>232.0</td>
</tr>
<tr>
<td>Virtex-7</td>
<td>73%</td>
<td>29%</td>
<td>33%</td>
<td>53%</td>
<td>229.4</td>
</tr>
</tbody>
</table>

- Double precision not possible for Kintex-7
- More stages could be added for Virtex-7

### Performance figures for a single V-cycle

<table>
<thead>
<tr>
<th>Target</th>
<th>Runtime [ms]</th>
<th>Throughput [V-cycles/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA(^2)</td>
<td>83.1</td>
<td>12.3</td>
</tr>
<tr>
<td>Intel i7</td>
<td>223.1</td>
<td>4.5</td>
</tr>
</tbody>
</table>

\(^2\)Performance is the same for Kintex-7 (XC7K325T) and Virtex-7 (XC7VX485T). Single precision on Kintex-7, double precision on Virtex-7.
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
ExaStencils Framework: Comparison of Lines of Code

ExaSlang 4 | C++ Pure MPI | C++ Hybrid MPI/OMP
---|---|---
Jacobi | 244 | 11,259 | 13,432
Gauss-Seidel | 236 | 9,600 | 11,320
Red-Black GS | 240 | 9,776 | 12,887

C. Schmitt, S. Kronawitter, S. Kuckuk | ExaStencils | ExaSlang and the ExaStencils code generator
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
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).