Generating Highly Parallel Geometric Multigrid Solvers with the ExaStencils Approach

Sebastian Kuckuk†, Christian Schmitt‡
† System Simulation, ‡ Hardware/Software Co-Design,
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
ESPT, New Orleans, LA, USA; November 17, 2014
Motivation

void Smoother() {
    for (int fragmentIdx = 0; fragmentIdx < 1; ++fragmentIdx) {
        if (isValidForSubdomain) {
            for (int x = iterationOffsetBegin; x < iterationOffsetEnd; x += 1) {
                #pragma omp parallel for schedule(static) num_threads(27)
                __m256d vec0 = _mm256_set1_pd(2.000000e-01);
                __m256d vec8 = _mm256_loadu_pd((&slottedFieldData_Solution[curSlot_Solution[1]%2][1][((x+(y*12))+(z*108))+16])
                    + slottedFieldData_Solution[slot][1](((12095040+y)+x)+(z*102960))), 1));
                __m256d vec6 = _mm256_loadu_pd((&slottedFieldData_Solution[curSlot_Solution[1]%2][1][((x+(y*12))+(z*108))+125])
                    + slottedFieldData_Solution[slot][1](((12095040+y)+x)+(z*102960))), 1));
                __m256d vec5 = _mm256_loadu_pd((&slottedFieldData_Solution[curSlot_Solution[1]%2][1][((x+(y*12))+(z*108))+123])
                    + slottedFieldData_Solution[slot][1](((12095040+y)+x)+(z*102960))), 1));
                __m256d vec10 = _mm256_loadu_pd((&slottedFieldData_Solution[curSlot_Solution[1]%2][1][((x+(y*12))+(z*108))+124])
                    + slottedFieldData_Solution[slot][1](((12095040+y)+x)+(z*102960))), 1));
                __m256d vec11 = _mm256_loadu_pd((&slottedFieldData_Solution[curSlot_Solution[1]%2][1][((x+(y*12))+(z*108))
                    +112]) + slottedFieldData_Solution[slot][1](((12095040+y)+x)+(z*102960))), 1));
                if (isValidForSubdomain) {
                    for (int x = iterationOffsetBegin; x < iterationOffsetEnd; x += 4) {
                        if (isValidForSubdomain) {
                            if (neighbor_isValid[0]&&neighbor_isRemote[0]) {
                                for (int w = 0; (w<1); w += 1) {
                                    for (int z = 1; (z <98); z += 1) {
                                        for (int y = 1; (y <98); y += 1) {
                                            for (int x = 4; (x <5); x += 1) {
                                                _mm256_storeu_pd((&slottedFieldData_Solution[curSlot_Solution[1]%2][1][((x+(y*12))+(z*108))
                                                    +124]) *2.000000e-01, vec10);
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}

reqOutstanding_Send_0[1] = true; } } } ;
reqOutstanding_Recv_0[0] = true; } } } ;
buffer_Send_0[3] = fieldData_RHS[1][((w*9409)+x)+(y*97)+(z*97)] ;
for ( ; (x<_upper) ; x += 1) {
    _mm256_storeu_pd((&slottedFieldData_Solution[curSlot_Solution[1]%2][1][((x+(y*12))+(z*108))
        +232]) *1.333333e-01, vec10);
}

/courses FAU Generating Highly Parallel Geometric Multigrid Solvers with the ExaStencils Approach
Sebastian Kuckuk, Christian Schmitt | FAU |
Motivation

void Smoother() {
    for (int fragmentIdx = 0; fragmentIdx < 1; ++fragmentIdx) {
        if (isValidForSubdomain) {
            for (int z = i + 0; (z < 100); z += 1) {
                for (int y = 1; (y < 98); y += 1) {
                    for (int x = 1; (x < 5); x += 1) {
                        buffer_Send_0[x][y][z] = std::min((i + 3), iterationOffsetEnd[0] + 7);
                    }
                }
            }
        }
    }
}

// SIMD extensions

#pragma omp critical
MPI_Isend(buffer_Send_0[1], 9409, MPI_DOUBLE, neighbor_remoteRank[1], ((unsigned int)commId << 20) + ((unsigned int)(neighbor_fragCommId[1]) << 10), mpiCommunicator, &mpiRequest_Send_0[1];

#pragma omp critical
for (int fragmentIdx = 0; fragmentIdx < 1; ++fragmentIdx) {
    if (isValidForSubdomain) {
        for (int z = 1; (z < 100); z += 1) {
            for (int y = 1; (y < 98); y += 1) {
                for (int x = 100; (x < 101); x += 1) {
                    buffer_Send_0[x][y][z] = std::min((i + 3), iterationOffsetEnd[0] + 7);
                }
            }
        }
    }
}

#pragma omp critical
for (int fragmentIdx = 0; fragmentIdx < 1; ++fragmentIdx) {
    if (isValidForSubdomain) {
        for (int z = 1; (z < 100); z += 1) {
            for (int y = 1; (y < 98); y += 1) {
                for (int x = (iterationOffsetBegin[0] + 3) & (~3); (x < _upper); x += 1) {
                    _upper = std::min((i + 3), iterationOffsetEnd[0] + 7);
                }
            }
        }
    }
}

#pragma omp critical
MPI_Isend(buffer_Send_0[3], 9409, MPI_DOUBLE, neighbor_remoteRank[3], ((unsigned int)commId << 20) + ((unsigned int)(neighbor_fragCommId[3]) << 10), mpiCommunicator, &mpiRequest_Send_0[3];

SIMD extensions
Motivation

Why not concentrate on the algorithmic description?

```plaintext
Function Smoother () : Unit {
    communicate Solution
    loop over Solution {
        Solution = Solution + 0.8 * (1.0 / diag(Laplace)) * 
                    (RHS - Laplace * Solution)
    }}
```
Motivation

Why not concentrate on the algorithmic description?

```java
Function Smoother () : Unit {
    communicate Solution
    loop over Solution {
        Solution = Solution + 0.8 * (1.0 / diag(Laplace)) * (RHS - Laplace * Solution)
    }
}
```

- **Productivity**
  - Algorithm description at high-level
  - Hide low-level details from programmer

- **Portability**
  - Support different target platforms from the same description
  - Support different target languages from the same description

- **Performance**
  - Portable: high performance on different target platforms
  - Competitive: comparable performance to hand-written code
ExaSlang
ExaSlang Layer 4

Properties

- Procedural
- Statically typed
- External DSL
- Syntax partly inspired by Scala
ExaSlang Layer 4

Properties

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

Specification of

- Operations depending on the multigrid level
- Loops over computational domain
- Communication and data exchange
- Interface to 3rd-party code
Data Types

Simple and aggregate data types

- Real, Integer, String, Boolean
- Complex<Real>, Complex<Integer>

Algorithmic data types

Field

- Correspond to discretized (mathematical) variables
- Communication scheme via Layout
- Specify Slot number for multiple copies

Stencil

- Correspond to discretized (mathematical) operators
- (Nearly) arbitrary expressions possible
Computations

Loop over computational domain split into loop over fragments

- Fragments stem from distribution across different cluster nodes
- Corresponds to global operation
- Optionally: reduction operators

and loop over <field>

- Iteration over parts of fields possible
- Corresponds to local operation
- Optionally: Reduction operators

```plaintext
Function NormResidual @(coarsest and finest) () : Real {
    Variable res : Real = 0
    loop over fragments with reduction(+ : res) {
        loop over Residual @current with reduction(+ : res) {
            res += Residual @current * Residual @current
        }
    }
    return ( sqrt(res) )
}
```
ExaStencils Framework

Abstract workflow:

- Algorithmic description
- Intermediate representation
- Parsing
- Pretty printing
- C++ output
ExaStencils Framework

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

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

is being 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) {
            // ....
        }
    }
}
```
Using a simple 1-step concept, we can do some refinements, e.g.,

```c
1  loop over Solution {
2    // ....
3 }
```

is being processed to

```c
1  for (int z = start_z; z < stop_z; z += 1) {
2    for (int y = start_y; y < stop_y; y += 1) {
3      for (int x = start_x; x < stop_x; x += 1) {
4        // ....
5      }
6  }
7 }
```


→ Very cumbersome with 1-step approach. Need something more flexible!
ExaStencils Framework
ExaStencils 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
ExaStencils 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 printed 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
ExaStencils Framework

Transformations

• Transform program state into 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
ExaStencils Framework

Transformations

- Transform program state into 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
ExaStencils 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
ExaStencils 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 + r)
})

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

Implemented workflow:

Algorithmic description

Parsing

Prettyprinting

C++ output
Function Smoother (): Unit {
    communicate Solution
    loop over Solution {
        Solution = Solution + 0.8 * (1.0 / diag(Laplace)) * (RHS - Laplace * Solution)
    }
}

- Program control flow and calculations can be transformed to C++
- What about parallelism?
  Affects nearly all parts of the generated program
  Requires major changes and concepts for
  - distributing the data and the work
  - keeping the data synchronized
ExaStencils Framework

```c
Function Smoother () : Unit {
  communicate Solution
  loop over Solution {
    Solution = Solution + 0.8 * (1.0 / diag(Laplace)) * (RHS - Laplace * Solution)
  }
}
```

- Program control flow and calculations can be transformed to C++
- What about parallelism?
  - Affects nearly all parts of the generated program
  - Requires major changes and concepts for
    - distributing the data and the work
    - keeping the data synchronized
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 parallelization strategy, e.g. using MPI and OMP
  - Each **block** corresponds to one *MPI* rank
  - Each **fragment** corresponds to one *OMP* rank
  - Pure *MPI* corresponds to one **fragment** per **block**
  - Pure *OMP* corresponds to one **block**
  - Hybrid *MPI/OMP* corresponds to multiple **blocks** and multiple **fragments** per **block**
  - Optional transformation: aggregate all **fragments** within one block and *OMP* parallelize field operations directly

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

- Two different ways:
  - Load domain information from file (work in progress)
  - If supported by domain shape and partition:
    generate req. information on the fly

- All domains are specified in Layer 4

```plaintext
1 Domain global< [ -1, -1, -1 ] to [ 1, 1, 1 ] >
2 Domain sthSmaller< [ -0.75, -0.75, -1 ] to [ 0.75, 0.75, 1 ] >
```

- Actual partition is specified through the number of blocks and fragments in each dimension
- Using this information domain setup code is generated
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 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 – Requirements

• 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
Communication – User Interface

- Communication statements are added automatically (wip) or specified by the user

```c
/* communicates all applicable layers */
communicate Solution@current

/* communicates only ghost layers */
communicate ghost of Solution[curSlot]@current

/* communicates duplicate and first two ghost layers */
communicate dup ghost [ 0, 1 ] of Solution[curSlot]@current

/* starts asynchronous communicate */
begin communicate 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
Communication – Node Progression

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

For each neighbor:

For duplicate and/or ghost:
Results
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 work in progress
Generated Lines of Code

<table>
<thead>
<tr>
<th>Method</th>
<th>ExaSlang 4</th>
<th>C++ Pure MPI</th>
<th>C++ Hybrid MPI/OMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobi</td>
<td>244</td>
<td>11,259</td>
<td>13,432</td>
</tr>
<tr>
<td>Gauss-Seidel</td>
<td>236</td>
<td>9,600</td>
<td>11,320</td>
</tr>
<tr>
<td>Red-Black GS</td>
<td>240</td>
<td>9,776</td>
<td>12,887</td>
</tr>
</tbody>
</table>
Weak Scalability

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

![Graph showing weak scalability with different core counts and combinations of OpenMP and MPI.]
Weak Scalability

- Time to solution
- $V(3,3)$ with Jacobi and CG

![Graph showing weak scalability with various configurations of MPI and OpenMP processes.]
Conclusion
Conclusion

• Transformation framework for generating massively parallel geometric multigrid solvers from an abstract representation
• Specialized DSLs allow for concise algorithmic descriptions (**Productivity**)
• Generated code is automatically tuned towards target platforms using, e.g., OpenMP, MPI and intrinsics (**Portability**)
• Performance and scalability are not yet optimal but already quite good (**Performance**)

Sebastian Kuckuk, Christian Schmitt | FAU | Generating Highly Parallel Geometric Multigrid Solvers with the ExaStencils Approach
Next Steps

- Applications
- Abstract DSL layers (ExaSlang 1 to 3)
- Target platform description language
- End-user tool support (e.g. editors)
- Variant generation and exploration (in combination with...)
- Convergence analysis
- Extended performance optimizations (e.g. color-partitioned memory) and examination of their impacts on, e.g., communication functions
- More in-depth analysis of performance potentials in overlapping communication and computation
- (Multi-)GPU support
- (HHG data structures)
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).

**Contributors (in alphabetical order):**
Sven Apel, Matthias Bolten, Alexander Grebahn, Armin Größlinger,
Frank Hannig, Harald Köstler, Stefan Kronawitter, Sebastian Kuckuk,
Christian Lengauer, Hannah Rittich, Ulrich Rüde, Christian Schmitt, Jürgen Teich