D1.3 Advanced Source-to-Source Translation Tools

Nabeeh Jumah Julian Kunkel

Workpackage:	WP1 Towards higher-level code design
Responsible institution:	Kunkel
Contributing institutions:	Universität Hamburg, RIKEN, IPSL
Date of submission:	February 2019

Contents

Abstract

One of the goals of the project AIMES is to improve the programming of the earth system models, which includes improving the code quality, and hence, the maintainability of the source code and the productivity of the scientists. To realize this we developed the GGDML language extensions, which we described in the previous reports. However, another important point concerning the advanced programming techniques is the performance portability of the model code. To provide such a feature, we designed our techniques, and hence tools, to translate the GGDML code through source-to-source translation tools, which apply some transformations to optimize the code for a specific architecture. The developed tools apply a set of user-driven optimization procedures to exploit the different hardware features of the underlying architecture.

In this report we focus on the efforts done under this project deliverable. We describe the requirements we collected to design the tools and the technique. Then we describe some optimizations that the tools apply within a description of the translation process. Some experimental results are also discussed to demonstrate performance impact on different architectures.

1 Introduction

1.1 Relation to the Project

This report describes the developed tools and demonstrates how to use them to create a own meta-dsl, dialects and back-end-ends to transform them.
The following text is the description of the project proposal for this task and deliverable:
In this task, the source-to-source translation tool is developed that converts code enriched with the meta-dsl (and derived formulations of dialects) flexibly into various target representations. It will be a light-weight tool to allow users to understand and maintain it in the future. Using back-ends, the abstract DSL formulation is convertible into a broad range of different approaches with limited effort. The tool will be flexible and modular to allow exchange of the meta-dsl and back-ends. Since back- ends are tightly coupled with the language specification each meta-dsl requires its own back-ends. We use a test-driven approach during the development and apply the tool to a synthetic meta-dsl which also serves as documentation for developing own meta-dsls and dialects. While the tool is developed by UHHSC, the other groups provide back-ends as targets for the con- version of the meta-dsl for icosahedral models. We evaluate existing DSLs, application frameworks and libraries that have been developed within previous projects such as ICOMEX and JST CREST and Physis, HybridFortran and GRIDTools as potential targets and provide translations for relevant application operators. During this process, we also anticipate to identify limitations in potential back- ends, for example, the Physis DSL assumes regular Cartesian meshes, while some of the target applications employ unstructured list vectors to represent partial differential equations, which is not possible to express in Physis. While these limitations restrict certain constructs of the meta-dsl for certain back-ends, the do not restrict the generality of the meta-dsl and the concept.

1.2 Higher-Level Coding Reviewed

In the work-package WP1 of the project AIMES, we developed a set of language extensions to support the development of earth system models, especially modeling with icosahedral grids. The extensions were developed based on the three icosahedral models DYNAMICO, ICON and NICAM which use icosahedral grids. The extensions were basically developed for each of the three models, while keeping in mind to specify a common set of the suggested extensions as long as possible. This common set of language extensions serves as the basis for the domain-specific language extensions that we specified as GGDML. For further details, please refer to the report D1.2 of the project. GGDML abstracts the scientific concept of the grid and provides the necessary code like specifiers, expressions, iterator to access and manipulate variables and grids from a scientific point of view.
The three icosahedral models that were used to guide the development of the language extensions were written in Fortran. However, the nature of the solution, in which we depend on extending the grammar of the general-purpose modeling language (GPL), allows to use the technique with different languages. That is because generally the grammars of the programming languages can be extended, and the new rules could exhibit different semantical features.

The language extensions that we developed for the models which are written in the Fortran language, are usable in other languages. For example, we have used this set of language extensions to write a test-code which uses the C language as the basic modeling language. In this case, the application's code is written mainly with C, and some parts use the GGDML extensions.

The implementation of the source-to-source translation tools provides the way to allow using the language extensions to extend a specific modeling language. For example, to process the test-code that we developed

with C and GGDML, a module was developed to handle the grammar of the C language. To use the language extensions to extend another language, the translation tools implementation needs to support that language.

2 Design Drivers

In this section we review the requirements which where specified at the beginning of the project to develop the language extensions and the translation tools design. We review here the requirements related to the translation process and derived tools. For the whole set of requirements, please refer to report D1.1 of the project.

2.1 Functional Requirements

We review here the functional requirements which drove the development of the translation tools.

RFPD **Parse extended language**

Parsing the developed extension of the general-purpose language must be doable by the tools for further processing.

RFRS **Read source files in project structures**

The code extensions of the GPL must be able to be integrated in the existing directory structures for the application projects. That means, tools must be able to read input source code from the models code repository and process them during source-to-source translation considering dependencies.

RFWS **Generate code output in project structures**

After source code tree contents are processed the solution should write the generated GPL code into an output code tree that fits into the existing project structure.

RFGC **Generate code for target machines**

The tools must be able to generate code for various back-ends that can be chosen upon compile time.

RFRC **Configurable process**

The translation process must be a configurable process (supporting RFGC). The tools must be configurable and particularly, the configuration and language must work together to allow flexible use across systems.

RFOP **Optimizing high-level code**

The tools must provide high-level optimizations that normally are implemented manually and cannot be conducted by compilers of the GPL. That means, the tools must apply the relevant optimization procedures and rewrite code in a way to get optimal performance on the target machine.

RNBI **Tools can be integrated into existing build systems**

The tools should be able to be integrated into existing build systems e.g. make. Simple calls to the translation tools should allow translating the source code to make it ready for compiling.

RNFL **Flexibility**

The translation technique and tools should allow modifying the DSL. Processing the DSL should not support a strict constant set of language constructs, it should rather support DSL modifications. Language processing module should be replaceable to provide alternative implementations.

2.2 Non-Functional Requirements

The non-functional requirements, which drove and guided the development of the translation tools are discussed in this section.

RNPP **Performance portability**

Performance portable code can target multiple architectures. On each supported architecture, performance portable code can optimally use the hardware features.

One of our goals in the AIMES project is to give the capability for scientists to write performance portable code. Atmospheric/climate applications are highly demanding for computing resources. However, rewriting parts of a model's code for different architectures is the price of getting performance.

One of the requirements while developing our translation technique and tools is to allow using the same source code on different architectures, while optimally using their capabilities.

RNCT **Compilation time**

Runtime of the tool must be in the order of the regular compile time (2x of current compile time is acceptable).

RNTS **Tools simplicity**

The use of the tool should not be a concern for scientists. Calling the tool to translate code by scientists should be simple enough and should not present any new learning burden or complicated build procedures.

RNTM **Tools maintainability**

Maintainability of the tools is an important point to consider. The tools should not need complicated maintainability efforts to keep them doing their work.

3 Translation Process

In this section, we discuss the concepts and some high-level details concerning the translation process. We describe the concept of 'separation of concerns' which was an important point to consider in the design phase. Tools design at the high-level is discussed briefly, and further discussion on configuration files is then provided. Some optimization procedures are discussed briefly to have a better overview on the tools and their design.

3.1 Separation of Concerns

In the approach that we use in AIMES project, we commit to the principle of separation of concerns. Climate scientists do not need to learn optimization techniques and programming models. They rather focus on the scientific side. This is done by a flexible DSL that abstracts domain concepts. Source code is developed with the high-level DSL. To get an optimized code based on the high-level code, translation tools transform the source code and apply optimization procedures. The optimization procedures are guided by information delivered through configuration files. Those configuration files are prepared by scientific programmers, who have the expertise with the target architectures. They know how to optimize code for those architectures. Based on this knowledge, they provided the necessary sections and options to generate a code version that optimally uses the features of a target machine.

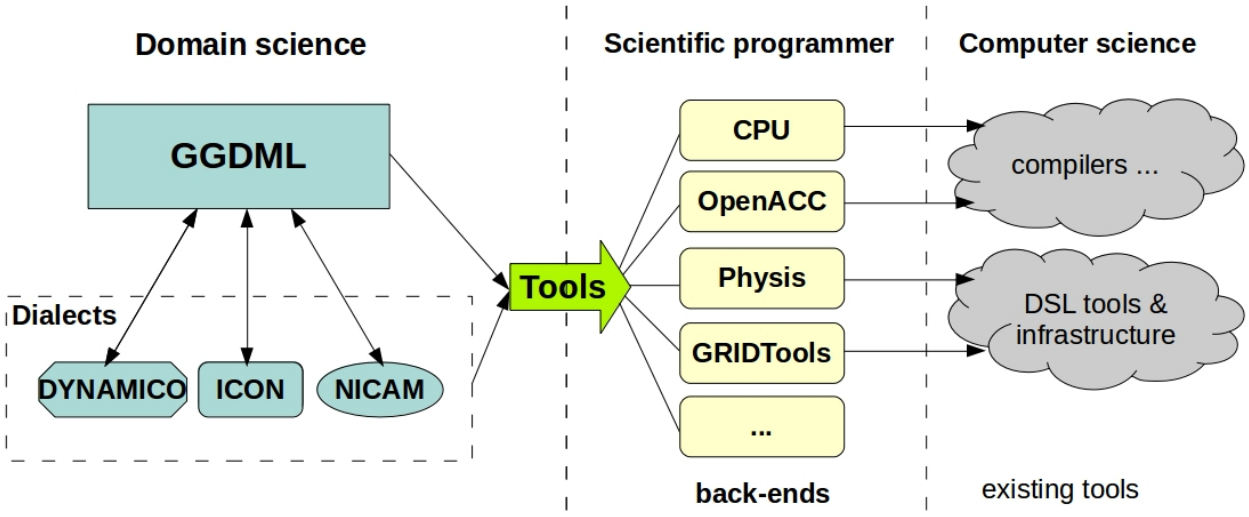

Figure 1: Separation of concerns

3.2 Tools Design

The high-level model code is developed with GGDML mixed together with general-purpose language. Translating this code into GPL code which can be used by compilers, passes through a set of steps. When calling the translation tool, the user needs to tell it where the root of the source code is located, besides to the language

handler modules, and the configuration file to use during the code translation process. Besides to translating complete trees, the tools can process one source file, the name of which can be passed as an alternative to the tree root folder. However, the tree translation allows to exploit optimizations available at the inter-file level between the source code files through inlining function calls and loop fusions.
As a first step, the tool loads necessary modules, which allow it to deal with the DSL extensions and the general-purpose language. Then, the specified configuration file is loaded and parsed, such that the modules are configured to parse the code according to the user needs. After the modules are loaded and configured, the source code is loaded and parsed. An AST structure is generated for each of the different source files. The configuration of the modules allows the translation tool to apply different optimization procedures. Some optimizations are applied within each AST, and some are done based on analyzing the set of ASTs. As a last step, the tool generates the final code from the ASTs. A new version of each file is generated mirroring the original model code tree. The new code takes the optimizations into account for the target architecture and configuration. Figure 2 shows the discussed concepts. Several relevant high-level optimizations[1] are implemented directly in the tool, which is extensible to allow the integration of even more optimizations.

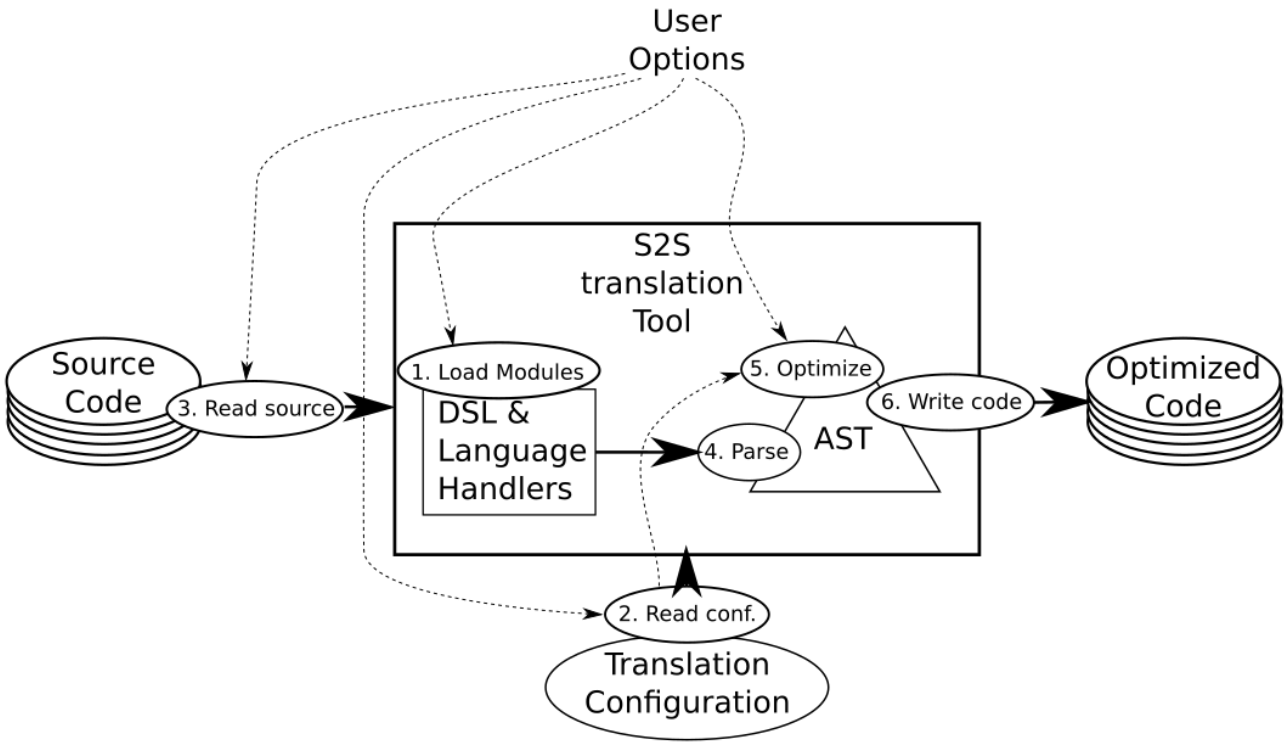

Figure 2: Translation Process

3.3 Configuration Files

The translation process that we use in our approach is controlled by the users. Besides to the role of the scientists who develop the scientific problem within the source code, expert scientific programmers provide optimization information to guide the high-level code translation. Configuration files contain this information. Different sections of the configuration files drive the different optimization procedures. Some sections are essential, and some can be ignored. When a section is missing, then a default optimization procedure is used. For example, if the local domain is not defined, then automatic domain decomposition is used.
Listing 1 is a sample from a configuration file demonstrating how the user provides different options to guide the code transformation process.

Listing 1: Example configuration file contents

```
...
RANGE OF YD= 0 TO GRIDY
RANGE OF XD= 0 TO GRIDX
...
CBLOCKING:
```

[1] We consider optimizations high-level that cannot be executed by the compiler due to the restrictions of a language's semantics.

```
XD=10000
ENDCBLOCKING
...
INDEXOPERATORS:
 edge_right(): XD=$XD+1
...
```

In this sample configuration part, different information are provided. The ranges of a grid specifying the global problem domain are shown. The values GRIDX and GRIDY are values that can be defined for compile-time processing. A few lines later we see information regarding blocking configuration. It tells the tool to apply blocking to the X dimension with a blocking factor of 10000. The definition line below defines the extension *edge_right* to allow access to the right edge by incrementing the index of the X dimension.
In the remainder of this section we describe some optimizations that can be applied through the tools and the configuration files.

3.4 Blocking

Memory hierarchies on the different architectures allow to reduce the need to data traffic between the processing units and the memory. However, the code should be written in a way that allows to optimally use those hardware features. Reducing the data movement from/to the memory is a key optimization for stencil computations as a result of the arithmetic intensity of such codes.
Data locality of the stencil applications allow to benefit much of memory hierarchies, if the code is written to exploit those memory hierarchies. Blocking is one technique to increase the reuse of the data while still being stored in the caches.
Our translation tool is designed to transform the code to apply blocking based on some input from the user through the configuration files. Blocking can be disabled if it is not desired to be applied. A kernel to compute the divergence at each grid cell is shown in Listing 2 using the GGDML language extensions mixed with C language.

Listing 2: Example kernel using C with GGDML

```
foreach c in grid
{
  float df=(f_F[c.east_edge()]-f_F[c.west_edge()])/dx;
  float dg=(f_G[c.north_edge()]-f_G[c.south_edge()])/dy;
  f_HT[c]=df+dg;
}
```

Applying the blocking using the blocking information provided within Listing 1 will result in generating the following code in Listing 3.

Listing 3: Example configuration file contents

```
...
        for (size_t blk_start = (0); blk_start < (GRIDX); blk_start += 10000) {
            size_t blk_end = GRIDX;
            if ((blk_end - blk_start) > 10000)
                blk_end = blk_start + 10000;
...

#pragma omp simd
                for (size_t XD_index = blk_start; XD_index < blk_end; XD_index++) {
...
```

Experiments were done to evaluate the impact of applying blocking to the generated code within the translation tools. Section 5.1 demonstrates the results.

3.5 Memory Layout

Choosing the right layout of the field data in memory is a key decision to optimally use the memory bandwidth. This is important for memory-bound computations, which is the case for stencil computations. A data layout that matches the most time consuming memory access patterns is essential to use the caching and the vectorization features of the underlying architecture.
Our translation tool provides a flexible user-driven memory layout transformation optimizer. It can apply a wide range of transformations ranging from simple index interchange to much complex formula based transformations,

e.g., space-filling curves. To further control the optimal use of memory layout, the user is also given the possibility to control loop interchange. The sum of the related transformation procedures within the translation tools allows to optimally use memory bandwidth and vector units.
The following code (Listing 4) demonstrates the index transformation technique. The two indices are transformed into one index in this example. Users provide the formula, and the tool uses it to transform the indices.

Listing 4: Example code generated with index transformation

```
...
#pragma omp for
          for (size_t YD_index = (0); YD_index < (local_Y_Cregion); YD_index++) {
#pragma omp simd
              for (size_t XD_index = blk_start; XD_index < blk_end; XD_index++) {

                 float df = (f_F[(YD_index + 1) * (GRIDX + 3) + (XD_index + 1) + 1] -
                             f_F[(YD_index + 1) * (GRIDX + 3) + (XD_index) + 1]) * invdx;
                 float dg = (f_G[((YD_index + 1) + 1) * (GRIDX + 3) + (XD_index) + 1] -
                             f_G[(YD_index + 1) * (GRIDX + 3) + (XD_index) + 1]) * invdy;
                 f_HT[(YD_index + 1) * (GRIDX + 3) + (XD_index) + 1] = df + dg;
              }
          }
...
```

Experiments were done to evaluate the impact of choosing the right memory layout to get optimal performance. Section 5.2 shows some experiments using those capabilities of the translation tool.

3.6 Inter-Kernel Optimization

To further improve the data reuse while being in the cache, further optimizations besides to blocking are applicable. Blocking allows to reuse data that need to be accessed within a kernel, however, there are opportunities some times to reuse data by transforming multiple kernels. Some consecutive kernels can be fused in some form to exploit such opportunities. Surely, this is done after some decisions are made regarding the feasibility of the merge and the data dependencies to guarantee the correctness of the computational results.
Our translation tool does some analysis to check the possibilities to fuse loops, and even check the possibilities to inline functions between source code modules, such that it allows further loop fusions. The tool provides a list of those possibilities to the user who can then choose as arguments which to apply, and then again the tool applies the selected optimization (inlining or fusion). This strategy provides the opportunity for a later automatic selection of the optimizations.
Listing 5 shows two kernels to compute the two components of the flux.

Listing 5: Example code with two kernels to compute flux components

```
...

#pragma omp parallel for
       for (size_t YD_index = (0); YD_index < (local_Y_Eregion); YD_index++) {
#pragma omp simd
          for (size_t XD_index = blk_start; XD_index < blk_end; XD_index++) {

             f_F[YD_index][XD_index] = f_U[YD_index][XD_index] *
                                       (f_H[YD_index][XD_index] +
                                        f_H[YD_index][(XD_index) + (-1)])
                                       * onehalf;

          }
       }
...

#pragma omp parallel for
       for (size_t YD_index = (0); YD_index < (local_Y_Eregion); YD_index++) {
#pragma omp simd
          for (size_t XD_index = blk_start; XD_index < blk_end; XD_index++) {

             f_G[YD_index][XD_index] = f_V[YD_index][XD_index] *
                                       (f_H[YD_index][(XD_index] +
                                        f_H[(YD_index) + (-1)][XD_index])
                                       * onehalf;
          }
       }
...
```

Listing 6 shows the resulting code after merging the two kernels.

Listing 6: Merged version of the flux computation kernels

```
...

#pragma omp parallel for
      for (size_t YD_index = (0); YD_index < (local_Y_Eregion); YD_index++) {
#pragma omp simd
          for (size_t XD_index = blk_start; XD_index < blk_end; XD_index++) {

             f_F[YD_index][XD_index] = f_U[YD_index][XD_index] *
                                       (f_H[YD_index][XD_index] +
                                        f_H[YD_index][(XD_index) + (-1)])
                                       * onehalf;

             f_G[YD_index][XD_index] = f_V[YD_index][XD_index] *
                                       (f_H[YD_index][(XD_index] +
                                        f_H[(YD_index) + (-1)][XD_index])
                                       * onehalf;
          }
      }
...
```

Experiments in Section 5.3 demonstrate the impact of using the inter-kernel optimization technique.

3.7 Scaling with Multiple-Node Runs

In the exascale era, using the features of a single machine is not enough but many nodes must be clustered together with powerful interconnects. So, codes that scale over multiple nodes is essential to run models which demand more computational resources than a single node.
To support scaling models written with GGDML over multiple nodes, we allow generating code to handle the necessary inter-node communication regarding the specified multi-node configurations. This includes domain decomposition and communication between the nodes.
Our translation tools allow both automatic and user-guided domain decomposition. The tool also detects the necessary halo-exchange operations through extracting the semantics from the GGDML language extensions. Then, those extracted information drive the generation of code that handles the communication. The whole procedure is configurable and controlled by the user. So, the user can provide the necessary instructions to initialize the communication library and the code patterns to handle the different halo patterns. This flexibility allows the user to choose whatever communication library of interest without changing the translation tool itself.
To examine the success of that technique, we used MPI and measured performance to test the scalability of the generated codes, and GASPI as an alternative to test the possibility to change the communication library.
The following code Listing 7 demonstrates generating communication code to exchange the halo data between the running processes.

Listing 7: Example generated code to handle communication of halo data

```
...
       //part of the halo exchange code
       if (f_G_dirty_flag[11] == 1) {
           if (mpi_world_size > 1) {
               comm_tag++;
               int pp = mpi_rank != 0 ? mpi_rank - 1 : mpi_world_size - 1;
               int np = mpi_rank != mpi_world_size - 1 ? mpi_rank + 1 : 0;
               MPI_Isend(f_G[0], GRIDX + 1, MPI_FLOAT, pp,
                         comm_tag, MPI_COMM_WORLD, &mpi_requests[0]);
               MPI_Irecv(f_G[local_Y_Eregion], GRIDX + 1, MPI_FLOAT, np,
                         comm_tag, MPI_COMM_WORLD, &mpi_requests[1]);
               MPI_Waitall(2, mpi_requests, MPI_STATUSES_IGNORE);

...

#pragma omp parallel for
      for (size_t YD_index = (0); YD_index < (local_Y_Cregion); YD_index++) {
#pragma omp simd
          for (size_t XD_index = blk_start; XD_index < blk_end; XD_index++) {

                  float df = (f_F[YD_index][(XD_index) + (1)] -
                              f_F[YD_index][XD_index]) * invdx;
```

```
                    float dg = (f_G[(YD_index) + (1)][XD_index] -
                                f_G[YD_index][XD_index]) * invdy;
                    f_HT[YD_index][XD_index] = df + dg;

                }
            }
    ...
```

Results of scaling on multiple nodes are shown in Section 5.4.

4 Test-Application

To test our techniques and tools, we developed a test application that solves shallow water equations. A regular grid configuration was used for the purpose of the run experiments. An explicit time-stepping scheme with the finite difference method are used.
The application runs mainly eight kernels to compute flux components, velocity components, and the surface level. Different grid widths were tested to understand performance with blocking and single/multiple node configurations.
To access the code of the test code, it is available online under
`https://github.com/aimes-project/ShallowWaterEquations`.

5 Experiments

In this section, we demonstrate experimental results of using the tools and the discussed translation technique. Experiments included different optimization procedures. Configurations were prepared for different experiments targeting different architectures. The multi-core processor experiments were run on Broadwell processors, GPU experiments were run on P100 GPUs, besides to experiments run on Aurora-SX vector engine. We used likwid, nvprof, and ftrace tools to record measurements on the Broadwell, P100, and Aurora respectively.

5.1 Blocking

In this section we discuss some experiments that we did to better understand how to transform the source code of a model to use blocking. This optimization allows the models to improve the use of the caches.
In first experiments, we generated two versions of the code for the Broadwell multi-core processor: with and without blocking. We ran the codes and recorded the performance of the test application using different grid widths. The results are shown in Figure 3. The results show that the performance drops when the code is run with wider grids. Blocking allowed to hold performance at a specific level instead of dropping, as data within the caches are reused. That is because the blocking takes into account the cache sizes, and hence maximizes the data reuse while they are still in the caches.
To better explore the blocking optimization, we also fixed the grid width and varied the blocking factor. The results are shown in Figure 4. The experimental results supported our theoretical estimations of the optimal blocking factors to maximize the data reuse within caches. Our theoretical estimations relate the cache sizes and the stencil structure to find the optimal blocking factor.
Similar experiments were repeated again with GPUs. We generated code for GPUs through the necessary configurations. Then we ran the two code versions (with and without blocking) on a P100 GPU. The results are shown in Figure 5. Again the caching allowed to keep performance over wider grids. However, we notice that the blocking is not necessary for smaller grids, in the contrary, it harms performance.
To find optimal blocking factor, we have done some theoretical estimations using the cache size and the structure of the stencil, and we tried to check the impact of the blocking factor experimentally. We ran the code with a fixed grid width, while varying the blocking factor over a wide spectrum of values. Figure 6 shows that small blocking factors yield much less performance in comparison to the optimal factor. Factors higher than that optimal value lead also to degraded performance.

5.2 Vectorization and Memory Layout

To understand the impact of the memory layout of the model code, we prepared different configurations to test different memory layout. First, we tested the impact of the access patterns and the distances between the data element on the performance of the model code. Table 1 shows the recorded application performance under three tested configurations, under two different architectures. In the first configuration, we investigated using contiguous unit-stride arrays to store the data of the model fields. The second configuration emulates

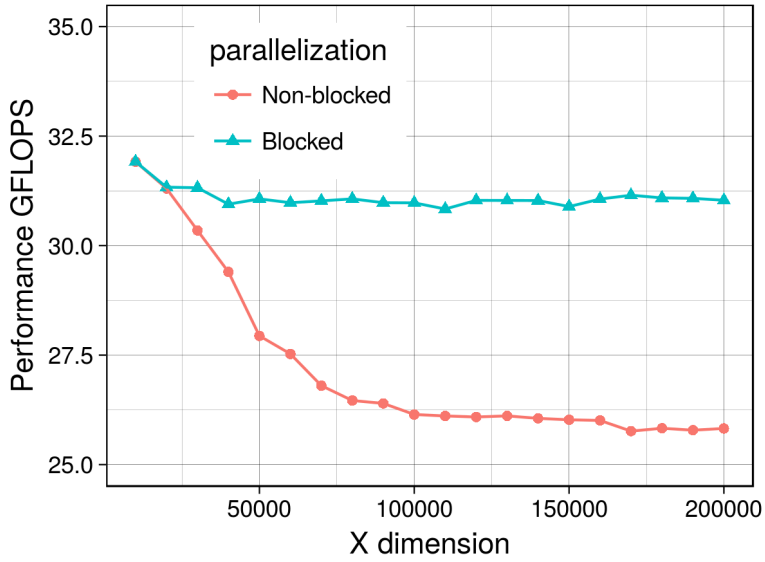

Figure 3: Performance measurements with variable grid width with and without blocking on Broadwell processor

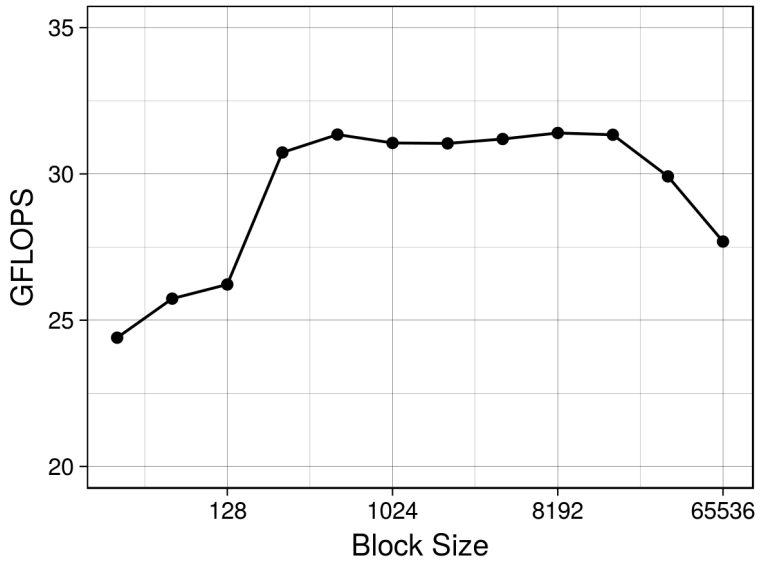

Figure 4: Performance measurements with variable blocking factor on Broadwell processor

the AoS structure[2], with a constant short distance separating the consecutive field data elements. In the third configuration, the code accessed scattered data elements separated with long distances.

Architecture	GFLOPS		
	Scattered	Short distance	Contiguous
Broadwell	3	13	25
NEC Aurora	80	161	322

Table 1: Performance measurements of both code versions with variable grid width with and without blocking on Broadwell processor

Looking at the Broadwell results, we find that the unit-stride code version performs well on the processor, taken into account the maximum memory bandwidth and the arithmetic intensity of the code. However, separating

[2]With AoS (Array of structures) data is interleaved such that consecutive data elements of the same field are separated with a constant distance, were data of other fields fit within those gaps separating field elements

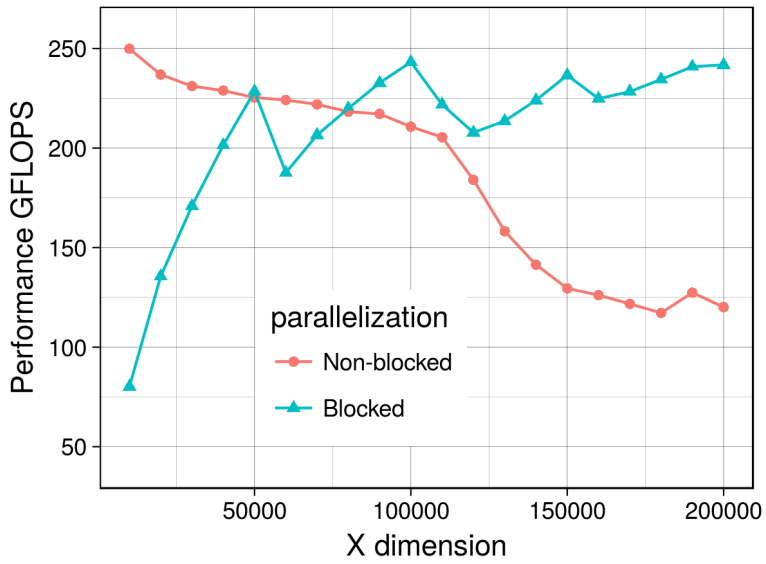

Figure 5: Performance measurements with variable grid width with and without blocking on P100 GPU

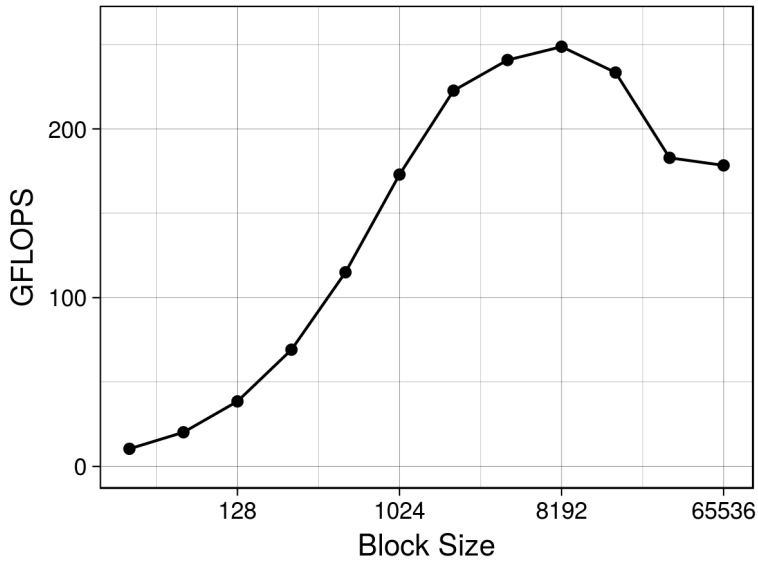

Figure 6: Performance measurements with variable blocking factor on P100 GPU

each data element (single precision floating point) with four bytes from the previous element reduced the performance of the application to the half. This is a result of two factors: first is the misuse of the caches, and the second is suboptimal use of the vector units. The details of the measurements show that some of the kernels were not vectorized. This was even much worse using the distant data elements. In that case the performance dropped vary much and non of the kernels were vectorized. So, the AVX2 features were not used, besides to the bad use of the caches.

Again, looking at the Aurora vector engine results, we find that the unit-stride code version performs well. The performance drops with nearly the same ratio as the Broadwell for the constant short distance separated values. Those reductions are again because of the bad use of the memory bandwidth. However, the performance drop ratio in the version with the scattered data elements is less compared to the Broadwell. That is because the vector units are still working, but with degraded performance of the data exchange between the vector units and the memory.

The results lead to focus on matching the right loop order and the memory access patterns. Our translation tool allows the user to control the loop order transformations and to fully control the data layout of the model fields.

5.3 Inter-kernel Optimization

To evaluate the impact of the inter-kernel optimization of loop fusions to exploit cache reuse, we ran some experiments on three different architectures: multi-core processors, GPUs, and vector engines. The eight kernels of the original source code are merged in some way to keep the consistency of the results. The new code resulting from the kernel merge is run to measure performance gain. This process was on the three architectures. Table 2 shows the results.

Architecture	Theoretical Memory bandwidth (GB/s)	Before merge		After merge	
		Measured memory throughput (GB/s)	GFLOPS	Measured memory throughput (GB/s)	GFLOPS
Broadwell	77	62	24	60	31
P100 GPU	500	380	149	389	221
NEC Aurora	1,200	961	322	911	453

Table 2: Performance measurements of both code versions with variable grid width with and without blocking on Broadwell processor

The table shows the maximum memory bandwidth of each architecture, and the achieved memory throughput. The arithmetic intensity of the code explains the achieved performance in terms of GFLOPS. The code runs approximately at comparable memory throughput before and after the kernel merges. However, we found that the efficiency of using that achievable memory throughput can be improved. This is done by using the data that is stored in the cache to compute more expressions while still being in the cache. This reduces the access to the memory and increases the arithmetic intensity.
The numbers in the table show the improvement in the application performance.

5.4 Scaling with Multiple-Node Runs

To experimentally examine the scalability of the code that the translation tool generates, we tested multiple node configurations. We generated code for multiple nodes with multi-core processors (Broadwell), using OpenMP besides to MPI. Also, we generated code for multiple nodes with GPUs (P100), using OpenACC besides to MPI. Figure 7 shows the measured performance of running the application on different numbers of nodes. The measurements were taken for 1,10,20...100 nodes. On the Broadwell processor, we used 36 OpenMP threads on each node.

6 Summary

In this report we discussed the development of the source-to-source translation tools that translate the high-level code, which uses the language extensions, into optimized code. We discussed the reqirements that guided the development of the needed tools. Then an overview of the high-level design of the translation technique was presented. Following the discription, we discussed some details regarding some optimization procedures that the tools apply during the code translation process. Some experimental results were presented to reflect the impact of the applied optimization procedures.

The analysis of the performance of the generated codes for the different architectures shows that the tools allowed to achieve near optimal performance. The tested code belongs to the stencil computation family of applications and hence is memory bound. Achieving a high percentage of the maximum memory bandwidth of an architecture that aligns with the theoretical estimations according to the arithmetic intensity of the code leads to the results that the optimization at the kernel level is nearly optimal. Besides to kernel level optimization, through the inter-kernel optimization we could increase the arithmetic intensity at the application level. Thus, the tools allowed to achieve also near optimal performance at the application level.

The measured performance, taking into account the maximum memory throughput on the different architectures, shows that the tools succeeded to provide performance portability. We used a single source code. Using this code, we could generate code for multi-core processors, which efficiently used the memory bandwidth. The same is true for the GPUs and the vector engines, where we also used the memory bandwidth efficiently. Therefore, the ratios of the achieved performance on one architecture to another reflects the ratio between the maximum memory throughputs of those architectures.

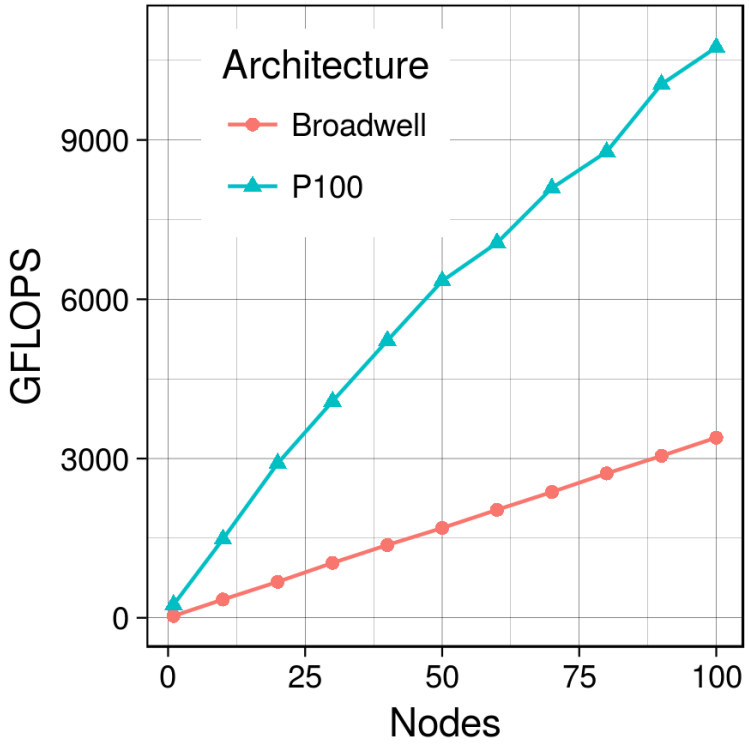

Figure 7: Performance measurements of both code versions with variable grid width with and without blocking on Broadwell processor

Important optimizations to let stencil operations run efficiently on the differnt architectures are implemented. Such optimizations are normally applied manually by scientists, which needs scientists to spend time and learn technical details regarding architectures and optimization. Thanks to the developed tools, scientists can focus their efforts on the scientific problems. The tools can add the optimization through the code translation process.

Acknowledgement

This work was supported by the German Research Foundation (DFG) through the Priority Programme 1648 „Software for Exascale Computing“ (SPPEXA).