SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Optimization of an ADER-DG-Solver for Hyperbolic Equations Using the BLIS Linear Algebra Framework

Lando Jahn

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Optimization of an ADER-DG-Solver for Hyperbolic Equations Using the BLIS Linear Algebra Framework

Optimierung eines ADER-DG-Lösers für hyperbolische Gleichungen mittels des BLIS Frameworks für lineare Algebra

Author:	Lando Jahn
Supervisor:	Prof. Dr. Michael Bader
Advisor:	Marc Marot-Lassauzaie M.Sc.
Submission Date:	15.08.2023

Acknowledgments

I would like to express my deepest gratitude to my advisor Marc Marot-Lassauzaie for making it possible for me to write my thesis at the Chair of Scientific Computing in Computer Science. He was always there to answer my questions and help me, which I truly appreciate. Additionally, I want to give special thanks to Prof. Dr. Michael Bader and the Chair of Scientific Computing in Computer Science at the Technical University of Munich for enabling me to write this thesis. Lastly, I would like to thank my family and friends for supporting me throughout my studies and this thesis.

Abstract

Many simulation problems can be described using hyperbolic partial differential equations. Solving these with performance in mind while not sacrificing too much in terms of accuracy can be achieved using refined numerical solvers, such as ADER-DG.
ExaHyPE 2 can generate sophisticated C++ code that uses ADER-DG to solve such numerical problems. It often requires many matrix multiplications, so providing an optimized implementation for them can be very beneficial to performance. This is currently done using an older version of the libxsmm code generator driver, which is deprecated, so we explore the use of another implementation: BLIS. During testing, we find that BLIS is not as performant for the smaller matrix multiplications used throughout most problems. On the grounds of this, we also take into consideration Eigen and a newer version of libxsmm that generates code during runtime to determine whether they can achieve performance that is comparable to or better than the existing option.
We explore the performance of all four implementations in exhaustive isolated performance tests with various matrix sizes and environments. During these tests we observe that Eigen and the newer libxsmm version can outperform the existing implementation in certain realistically occurring situations, mainly when the existing implementation does not provide optimizations for the specific microarchitecture. As a result of that and the fact that BLIS may become more performant for smaller matrix multiplications in the future, we incorporate all three of the newly explored implementations in the ExaHyPE 2 generator as an addition to the one that is already included. Consequentially, users of ExaHyPE 2 can easily have such optimizations included in their generated ADER-DG programs by invoking an option.
In performance tests on the entire program, we find that Eigen may sometimes outperform the previously used libxsmm code generator driver, even in cases where it is optimized for the underlying microarchitecture.

Contents

1. Introduction

1.1. Motivation

There are many use cases for physics-based simulations. For example, when wanting to warn of incoming tsunamis, it is important to know where and how they develop and when they will hit the shore. For that, a model that may for instance be based on the shallow water equations can be designed. The shallow water equations are an example of hyperbolic partial differential equations that we may want to solve numerically [46]. In order to use the model to simulate the tsunamis on a computer, we need to discretize the problem (for example in space by dividing the domain into cells and in time by turning continuous time into discrete time steps) and then solve the equations. We do that with a numerical solver, using ADER-DG (**A**rbitrary high-order **DER**ivatives-**D**iscontinuous **G**alerkin [22]), which applies "high order polynomials [...] to span the solution in space and time"[8].
ADER-DG performs a multi-part process for each time step:
First, there is the predictor step: When performing that for one particular cell, we act as if there were no other cells and just compute (by use of the polynomial), what would change within the cell during that time step. This corresponds to a volume integration over time [2][8].
Then, the so-called corrector step is performed, computing the solutions for the discontinuities present at the cell interfaces (i.e. where two cells meet) by use of a Riemann solver [8].
As a final part of this process, the results of both the predictor and the corrector step are then combined to yield the solution for the calculated time step [8].
Since this process is very compute-intensive, it is important to optimize for performance as much as possible, without making the results less accurate. This is why we want to implement the numerical solver in C++, using state-of-the-art frameworks/libraries [8].
Taking this one step further, we do not only want to be able to solve the shallow water equations but many different problems that can be described using hyperbolic partial differential equations. To do all that while staying performant, the project we are working on (ExaHyPE 2, part of the Peano repository, see [52]) offers C++ code generation. ExaHyPE 2 allows the user to simply "describe" the problem using a Python script and some C++ code. They can then choose from different solvers and optimizations for the

generated code. Since this generated code often relies on matrix multiplications, this is one area where optimizations are well fit. For these matrix multiplications, there are currently two options to choose from: One of them is a naïve implementation, without the use of any frameworks/libraries, whilst the other one utilizes Intel's libxsmm, more specifically the libxsmm code generator driver, which creates platform-specific C code that performs matrix multiplications. The problem we are facing with that is that this part of libxsmm is deprecated [46][53][35][39].
To avoid future problems, we want to explore the use of another external framework in the ExaHyPE 2 ADER-DG-solver: BLIS ("BLAS-like Library Instantiation Software" [3]). It is a framework for dense linear algebra operations, which are BLAS-like. More on the specific BLAS operations we use is explained in section 1.2. BLIS offers matrix multiplication kernels for a range of different microarchitectures [6][7][3].
We want to use BLIS since it promises to provide high performance. In the following, we are going to explore the performance challenges we had with that framework and two examples, as well as the reason for them and some alternative implementations that could be used instead [3].

1.2. The Notation for the Matrix Multiplications

In the following two sections, we want to establish the terminology/conventions used throughout the thesis. The goal is to perform BLAS-like general matrix multiplications of the form 1.1 [5].

$$C := \beta \cdot C + \alpha \cdot A \cdot B \tag{1.1}$$

BLAS ("Basic Linear Algebra Subprograms" [51]) provide an interface for linear algebra operations, that are structured into three "levels". Vector-vector operations are included via level-1, level-2 provides matrix-vector operations and level-3 consists of matrix-matrix operations. Since we are working with dense matrix multiplications, we will be using level-3 operations [51].
In Equation 1.1, A is a matrix of size $m \times k$, B of $k \times n$ and C is of size $m \times n$. A little visualization for easy reference follows [5]:

$$A := \underbrace{\begin{pmatrix} a_{1,1} & \cdots & a_{1,k} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,k} \end{pmatrix}}_{\text{k columns}} \left.\vphantom{\begin{pmatrix} a \\ \vdots \\ a \end{pmatrix}}\right\} \text{m rows} \tag{1.2}$$

$$B := \underbrace{\begin{pmatrix} b_{1,1} & \dots & b_{1,n} \\ \vdots & \ddots & \vdots \\ b_{k,1} & \dots & b_{k,n} \end{pmatrix}}_{\text{n columns}} \Bigg\} \text{k rows} \tag{1.3}$$

$$C := \underbrace{\begin{pmatrix} c_{1,1} & \dots & c_{1,n} \\ \vdots & \ddots & \vdots \\ c_{m,1} & \dots & c_{m,n} \end{pmatrix}}_{\text{n columns}} \Bigg\} \text{m rows} \tag{1.4}$$

1.3. The Matrix Storage

Matrixes need to be stored in memory. Since we are dealing with general (not sparse) matrix multiplications, they are best stored consecutively in memory as one single vector. There are different ways to store matrixes in random access memory, we use a column-major format. A "matrix is stored in column-major order if it is stored column by column, starting with the entire first column, followed by the entire second column" [16]. One metric we have to keep in mind in this regard is the leading dimension. "The leading dimension for a two-dimensional array is an increment that is used to find the starting point for the matrix elements in each successive column of the array." [30] In our case, that is the number of rows in a matrix plus some padding we may want to put between the end of a column and the beginning of the next one, for example for alignment purposes. Figure 1.1 shows that graphically. In BLIS, this concept is called column-stride which we will discuss more later. The same concept, but with rows (row-major matrix storage) is also offered by BLIS, however, we will not be using that [7].

1.4. The Parameters for the BLIS General Matrix Multiplication Call

Here, we want to introduce the parameters required for the BLIS matrix multiplication function calls (see Figure 1.2). The first two parameters (`transa` and `transb`) determine whether A or B should be transposed and/or conjugated. `m`, `n`, and `k` are the matrix sizes that were discussed in section 1.2, just like `alpha` and `beta`, which correspond to the scalars α and β from Equation 1.1. The currently used libxsmm code generator driver only supports $\alpha = 1$ and $\beta \in \{0, 1\}$ so that is all we need to support. The pointers to A, B, and C's buffers are `a`, `b`, and `c`. As mentioned, BLIS refers to the leading dimension as column-stride, as it also offers operations on row-major matrixes. That is why we have the row-stride parameters (`rsa`, `rsb`, and `rsc`, which we set to 1 for use with column-major matrix storage) in addition to the column-stride ones (`csa`, `csb`, and `csc`) [5][39].

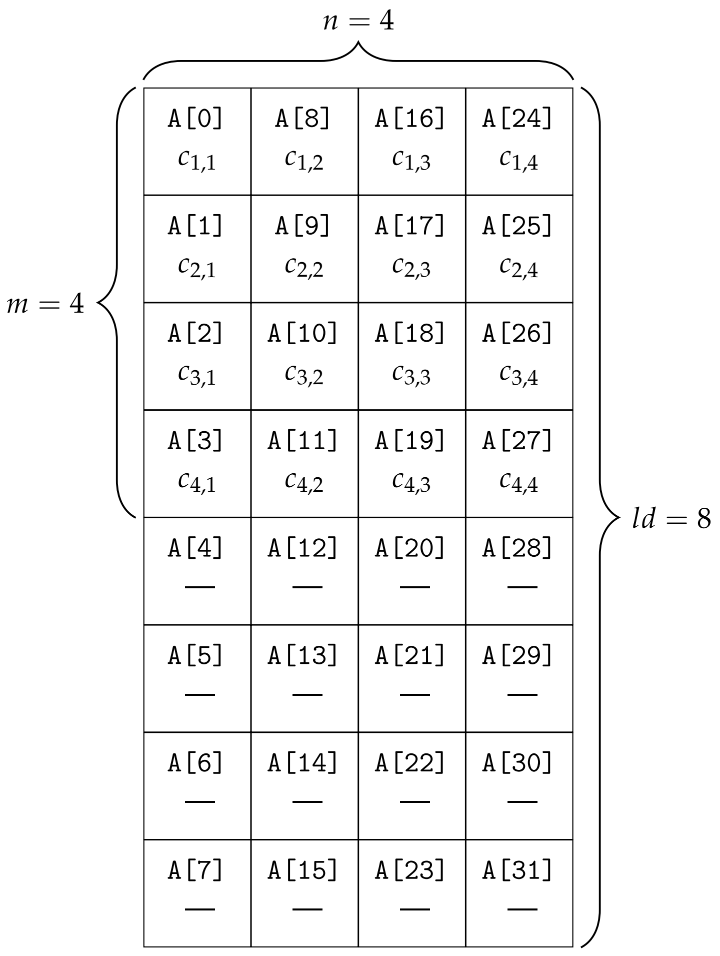

Figure 1.1.: Example of a column-major 4×4 matrix C stored in the 32-byte array `A`; In this case $m = n = 4$ holds and the leading dimension (ld) is 8, so not all entries of `A` are used for C's elements

```c
void bli_dgemm
 (
   trans_t transa,
   trans_t transb,
   dim_t m,
   dim_t n,
   dim_t k,
   double* alpha,
   double* a, inc_t rsa, inc_t csa,
   double* b, inc_t rsb, inc_t csb,
   double* beta,
   double* c, inc_t rsc, inc_t csc
 );
```

Figure 1.2.: A BLIS double precision general matrix multiplication function interface, showing the required arguments (adapted from [5])

2. Testing the Application of BLIS With Two Examples

2.1. Replacing the Matrix Multiplications in the Shallow Water Equations Example

We want to optimize the shallow water equations example using BLIS. The idea is to have ExaHyPE 2 generate C++ code for this example with the libxsmm optimizations enabled. We then look at the interfaces/matrix multiplication calls for the generated functions and swap in calls to BLIS versions of corresponding matrix multiplications. Then we can compare the implementations regarding performance and equivalence of results, to see if optimizing with BLIS is a viable choice here [39][35][3].
To conduct the experiment, we generate code for the shallow water equations example with bathymetry provided at `applications/exahype2/aderdg/SWE/w_bathimetry/` in the Peano repository. After configuring Peano with `--enable-exahype --enable-blockstructured --enable-loadbalancing --with-libxsmm`, we add `use_libxsmm=True` to the kernel optimizations in `SWE.py` to enable the use of libxsmm in the generated code. The generated code can be found in `generated/kernels/aderdg/nonlinear` (starting from the SWE base directory). From there, we can identify the libxsmm matrix multiplications used in `fusedSpaceTimePredictorVolumeIntegral.cpp`. The libxsmm-generated code itself is located inside `asm_fstpvi.c`.
When looking at a particular matrix multiplication, we can tell which arguments the libxsmm-generated code utilizes for the matrix multiplications.
One example: The statement

```
gemm_4_4_4_rhs_y(lFi+((t*1+z)*16+x)*4+256, coeffRhsY, rhs+((t*1+z)*16+
    x)*4);
```

calls the function shown in Figure 2.1.
Since the libxsmm code generator driver gets the arguments at generation time, we cannot see them in the call directly, they have already been included in the source code. Therefore, we look at the generated code to determine the parameters used for the matrix multiplications to use them for the BLIS calls [39].
We can tell that $m = n = k = 4$ since the counting indices for them (`l_n`, `l_m`, `l_k`)

```
void generated::kernels::AderDG::nonlinear::gemm_4_4_4_rhs_y(const
    double* A, const double* B, double* C) {
[...]
  for ( l_n = 0; l_n < 4; l_n++ ) {
    for ( l_k = 0; l_k < 4; l_k++ ) {
      #pragma simd
      for ( l_m = 0; l_m < 4; l_m++ ) {
        C[(l_n*16)+l_m] += A[(l_k*16)+l_m] * B[(l_n*4)+l_k];
      }
    }
  }
[...]
}
```

Figure 2.1.: A matrix multiplication located in `asm_fstpvi.c`, generated by the libxsmm generator driver [39] (shortened); Since we are working with an AMD Zen 2 CPU and libxsmm does not have specific optimizations for that microarchitecture in the generator driver, the target architecture is noarch (no architecture), resulting in no platform-specific optimizations being present in the raw code, the compiler may later apply such optimizations [1][39]

are kept within the range $(0,3)$. For m that is the case, as the number of variables for this SWE model is four. The number is made up of height, velocity in the x and y directions, and bathymetry. As a result of the number of degrees of freedom being four, which is the order (three in this case) plus one, n and k are also four. The matrixes are accessed in column-major order, with a column-stride of 16 for A and C (`csa` and `csc`) and 4 for B (`csb`). E.g. for C, this can be seen, as the column index (`l_n`) is multiplied by 16, indicating that the next column begins `16*sizeof(double)` after the previous one. Since the entries in C are not set to/multiplied by anything beforehand and there is no scaling of entries before adding them, we can tell that `alpha=beta=1`. Lastly, no transpositions or conjugations are done here, so `transa` and `transb` will be `BLIS_NO_TRANSPOSE` [52][5][3].

2.2. BLIS: Typed vs. Object API

2.2.1. The Typed and Object APIs

After getting the required function arguments for the BLIS call, we need to plug them in. BLIS offers two different Application Programming Interfaces (APIs), we need to decide on which one to use [43][3].
The so-called typed and object APIs both allow the user to access the available functions (e.g. general matrix multiplications). As the name suggests, the object API uses objects (structs named `obj_t`) to "[abstract] away properties of vectors and [matrixes] [...], [to] be queried with accessor functions." [3] Among other things, these structs contain a reference to the matrix buffer and information about the matrix size. The typed Application Programming Interface (with the general matrix multiplication function signature shown in Figure 1.2) gets all the information required via a single function call, removing the need to keep track of any structs. Generally, both APIs perform the same operations with similar arguments, just passed at different points in time [3][4].

2.2.2. Considerations As to Why We Chose the Typed API for This Experiment

Both APIs can have sensible applications, in the following we want to provide two points that were considered when choosing the typed API:

Lower (Implementation) Overhead

For the typed API, we don't need to create the structs (of type `obj_t`) to represent the matrixes. Since we are working with an existing project, creating them would be more of an afterthought than how the program is designed. Furthermore, when looking at the libxsmm calls, they are often done multiple times in a row with pointers to different matrixes. Recreating this would require us to reassign a pointer to, or redefine some `obj_t` each time beforehand. These operations can be avoided by utilizing the typed API.

Performance

Going hand in hand with the previous point, one would expect more function calls, as we would have with the object API, to likely not be beneficial for the code's performance. However, it may also be possible that calling the object API implementation of the general matrix multiplication a lot amortizes the overhead from creating the structs, as they already contain the information passed to the typed API each call and can

repeatedly access it. We suppose that big amortization does not take place here, due to the frequent switching of matrixes worked with. To confirm or deny this hypothesis, we run performance tests for model matrix multiplications with both versions of the BLIS Application Programming Interfaces used, respectively. The example we use is a general matrix multiplication with $m = n = k = 4$, where all matrixes are tightly packed column-major matrixes, which leads the column-strides (`csa`, `csb`, `csc`) to be 4, as well as the row-strides (`csa`, `csb`, `csc`) to be 1. `alpha` and `beta` are 1 and the matrixes are not transposed. The appropriate calls for the typed and object APIs can be seen in Figure 2.2. Each implementation is run five million times and the average time per matrix multiplication is calculated. The program is compiled with `-O3`, BLIS for Zen 2 statically linked and the header provided by it is included. The POSIX threads library is linked to as well, as BLIS requires that [3][50][6][18].
We run the program on the local machine from subsection 3.3.1. This results in the object API calls (574 *ns* on average per matrix multiplication) being just a little bit faster than the typed calls (585 *ns* on average per matrix multiplication).

Concluding the Comparison

For this manually performed experiment, we feel like the performance benefit of about 1.9% does not justify the implementation overhead of more function calls, as well as the memory overhead that comes with the structs that have to be stored. It is also possible that the five million iterations exaggerate the amortization since the matrix multiplications are not called as often in a row in the actual example. However, this decision can be reversed pretty easily later on, should one decide to do so.

```
[...]
// typed API
for(int i = 0; i < 5000000; ++i) {
    bli_dgemm(BLIS_NO_TRANSPOSE, BLIS_NO_TRANSPOSE, 4, 4, 4, &one, A,
        1, 4, B, 1, 4, &one, CBLIStyped, 1, 4);
}
[...]
// object API
obj_t a, b, c;
bli_obj_create_without_buffer(BLIS_DOUBLE, 4, 4, &a);
bli_obj_create_without_buffer(BLIS_DOUBLE, 4, 4, &b);
bli_obj_create_without_buffer(BLIS_DOUBLE, 4, 4, &c);

for(int i = 0; i < 5000000; ++i) {
    bli_obj_attach_buffer(A, 1, 4, 1, &a);
    bli_obj_attach_buffer(B, 1, 4, 1, &b);
    bli_obj_attach_buffer(CBLISobj, 1, 4, 1, &c);
    bli_gemm(&BLIS_ONE, &a, &b, &BLIS_ONE, &c);
}
[...]
```

Figure 2.2.: Matrix multiplication calls doing the same thing using the BLIS typed (top) and object (bottom) APIs; Time is also measured (not shown here); For the object calls we create the structs only once, as we reuse them; We reattach the buffers to the structs each iteration to simulate the matrix locations changing and not omit that overhead, even though they are always assigned to the same matrix in this case

2.3. Creating the New API Calls

After deciding on the typed API, we can plug in the parameters to create the call for the example matrix multiplication from section 2.1:

```
bli_dgemm(BLIS_NO_TRANSPOSE, BLIS_NO_TRANSPOSE, 4, 4, 4, &one, lFi+((t
    *1+z)*16+x)*4+256, 1, 16, coeffRhsY, 1, 4, &one, rhs+((t*1+z)*16+x
    )*4, 1, 16);
```

Since BLIS requires `alpha` and `beta` to be passed as a pointer, the `one` constant was previously defined as

```c
double one = 1.0;
```

The same process can be done with the remaining matrix multiplications. Replacing them all yields the BLIS version of the code. We include the BLIS-header (`"blis.h"`) and modify the Makefile to link to the BLIS static library (`libblis.a`) and the POSIX threads library (`-lpthread`), while also including the BLIS `include/zen2` directory. The program can then be recompiled and executed [6][50].

2.4. Checking the Validity of the Results

Changing the implementation of the matrix multiplications should only be done if, given the same input, both versions produce the same output. In the SWE example, we verify this in two ways: optically and data-driven. We first look at the animations rendered from the solution files of the BLIS and libxsmm versions of the program and compare them (see Figure 2.3). We also introduce a small program to test both implementations with many different values. It has a doctest test case for each of the libxsmm-generated matrix multiplications we replace. Inside the test case, we perform 100 iterations where we create four buffers `A`, `B`, `Cxsmm`, and `CBLIS`. The `Cxsmm`/`CBLIS` buffers are filled with zeros in case $\beta = 1$ and ones for $\beta = 0$ (to test that both implementations set C to zero). `A` and `B` are packed with random numbers. We then let the libxsmm-generated function and the corresponding BLIS call (with the same arguments we use inside the experiment, except for the matrix locations) perform a matrix multiplication. Both implementations use the same A and `B` to compute the results for their target buffers (`Cxsmm` and `CBLIS`). The doctest `CHECK` macro is used to test if each entry of the target buffers is equal for both implementations. For the example call from section 2.3, the test case is shown in Figure 2.4. A repository containing all test cases is available at `https://gitlab.lrz.de/landojahn/blis-verification`. The test environment is the same one as described in subsubsection 2.2.2 [10].
Both methods of verification indicate the desired outcome: equivalence of the results of the calculations made by the BLIS and the libxsmm-generated versions.

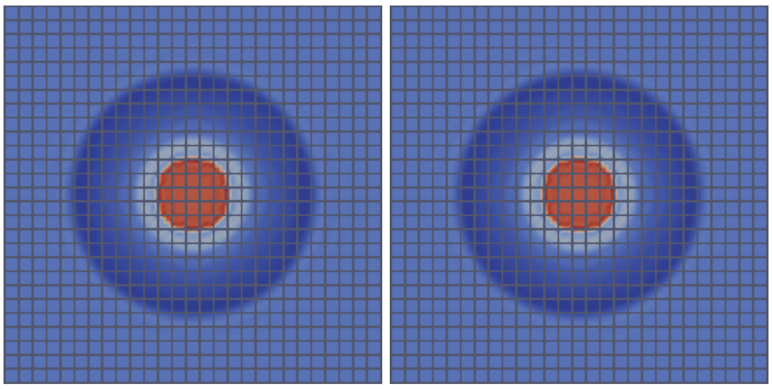

Figure 2.3.: The SWE example with bathymetry computed using BLIS (left) and libxsmm-generated code (right) after a time of $t = 0.0503095$ seconds, rendered with the render script provided by Peano [52] and visualized with ParaView [31]

2.5. The Performance Problem

Even though the results are the same, one thing stands out: the performance. While the libxsmm version of the SWE solver takes 16 seconds to complete, the version with the BLIS typed matrix multiplications completes in 55 seconds. We got these results on the same local machine as described in 3.3.1. This is a problem since we cannot optimize the performance of the program with it becoming about 244% slower with BLIS.
Another example in the same environment, the elastic example, also provided in the Peano repository (at `applications/exahype2/aderdg/Elastic`) shows results hinting in a similar direction: While the libxsmm version takes just four seconds to execute, the BLIS version completes the computations in seven seconds, a performance decrease of 75%.
We want to answer the question: Why is it slower? We are going to explore that along with the performance of some alternatives to BLIS in the next chapter [52].

```
TEST_CASE("gemm_4_4_4_rhs_y") {
  for (int y = 0; y < 100; ++y) { // test 100 different scenarios
    double A[64], B[16], Cxsmm[64], CBLIS[64];

    [...] // Fill buffers with 0 (for C) and random values (for A and
        B)

    gemm_4_4_4_rhs_y(A, B, Cxsmm);

    bli_dgemm(BLIS_NO_TRANSPOSE, BLIS_NO_TRANSPOSE, 4, 4, 4, &one, A,
        1, 16, B, 1, 4, &one, CBLIS, 1, 16);

    for (int i = 0; i < 64; ++i) {
      CHECK(CBLIS[i] == Cxsmm[i]);
    }
  }
}
```

Figure 2.4.: A test case for the example from section 2.3, verifying equivalence between the libxsmm-generated code and the BLIS version, using doctest [10] (shortened)

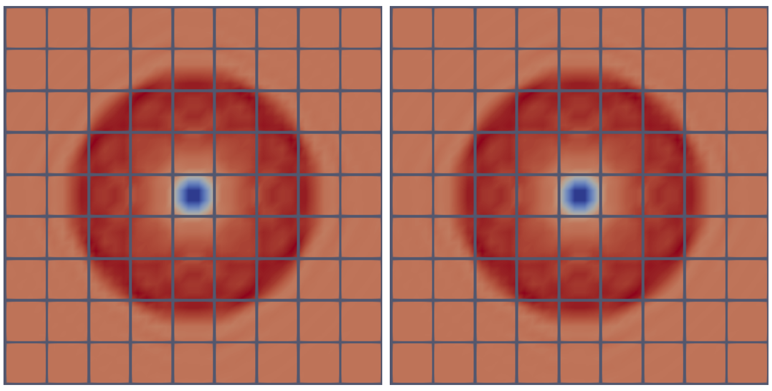

Figure 2.5.: The elastic example computed using BLIS (left) and libxsmm-generated code (right) after a time of $t = 0.03$ seconds, rendered with the render script provided by Peano [52] and visualized with ParaView [31]

3. Evaluating the Performance of Different Matrix Multiplication Implementations

3.1. The Hypothesis: BLIS Is Only Competitive for Bigger Matrix Sizes

One possible hypothesis suggesting why the performance is worse is that BLIS is designed for bigger matrixes, which are not present in the ExaHyPE 2 shallow water equations solver. A result we have seen before hints at this conclusion: The elastic example only takes about 75% longer with BLIS than with the libxsmm-generated code, compared to the 244% increase we saw with the shallow water equations. This may be the case, as the matrixes are larger than in the SWE example. While the shallow water equations work with $m = n = k = 4$, the elastic example uses $m = 9$ and $n = k = 6$ [52].
One possible explanation for this behavior is that BLIS has more initial overhead, with lower computation times afterward, which could make it more performant on bigger matrixes, as the relation of initial overhead to computation time afterward might be smaller. In the following, we want to test whether the hypothesis holds, and if so, what counts as a large enough matrix for BLIS to be viable and competitive.

3.2. Testing the Hypothesis

To test this assumption, we want to compare BLIS' performance when multiplying varying-size matrixes. To have a sensible comparison with fitting categorization and also get information on which implementations would be worth it to integrate into the existing code, we supply results of multiple other implementations' performance on the same problems. This allows us to not only look at the results in an absolute way but also relative to other implementations to see in which scenarios BLIS is competitive and where the other ones are best used.
To represent the current implementation, we use C code generated by the libxsmm code generator driver version included in Peano. Since that is deprecated, we also add the regular version of libxsmm. It does not generate code beforehand, but rather "JIT

(Just-In-Time)" [39] (right when it is needed) during runtime. This allows it to adapt to the microarchitecture used during runtime [39][21].
The last implementation we want to include is Eigen, described by the authors as a "C++ template library for linear algebra: [matrixes], vectors, numerical solvers, and related algorithms." [14]
Since we want to compare the different implementations' performances on various matrix sizes, we need different test cases for the matrix sizes. The libxsmm-generated code distinguishes between them, not by use of function arguments, but rather by the function name, so we need to make a new block of code for each test case or use function pointers. To be as close as possible to the real implementation and not slow the tests down artificially, we decided on the former method. This is reliably accomplished using a Python script, that generates the test code using Jinja2, a template engine that creates code by replacing variables in the provided template. The script also invokes the commands for the libxsmm generator driver to create the matrix multiplication functions appropriate for the target microarchitecture, which is specified beforehand [42].
The generated code then goes through previously defined problem sizes, performs double precision matrix multiplications with each implementation a predefined amount of times (10000 in these experiments), and calculates the average completion time per matrix multiplication. In the final step (for every test case), each implementation's result matrix (C) is compared for equality. The performance results are then plotted using another Python script and Matplotlib. The test cases can be found in the `generated` directory of the repository located at `https://gitlab.lrz.de/landojahn/blis-performance` [49].

3.3. The Test Setups

We run the tests on three different setups with the specifications we describe here.

3.3.1. Local

The local machine we use is running Fedora Linux 37 on an AMD Ryzen 5 5500U with 16 GB of DDR4. As a compiler we use `g++ (GCC)12.2.1` [45][1][20].

3.3.2. CoolMuc-2

We use the CoolMuc-2 cluster of the Leibniz-Rechenzentrum. The programs are executed on an interactive compute node (single-node), which runs SUSE Linux Enterprise

Server 15 SP1 on an Intel® Xeon® E5-2697 v3 with 64 GB of DDR4. Programs are compiled with `icpc (ICC)2021.7.0` [25][33][48][27].

3.3.3. CoolMuc-3

Another cluster of the Leibniz-Rechenzentrum we use is the CoolMuc-3. Again, programs are run on an interactive compute node (single-node) that uses an Intel® Xeon Phi™ 7210F processor with 96 GB of DDR4 and 16 GB of High Bandwidth Memory. As an operating system, SUSE Linux Enterprise Server 12 SP5 is used. Due to the outdated state of the `icpc` C++ compiler on CoolMuc-3, we compile programs on CoolMuc-2 with `icpc (ICC)2021.7.0` to run here [34][26][47][25].

3.4. The Test Results

We generate test cases for the test setups from section 3.3 and compile them with the `-O3` and `-DNDEBUG` flags. The plotted performance results can be seen in Figures 3.1 through 3.5. The problem sizes are ordered according to $m \cdot n \cdot k$ in ascending order, to approximate the complexity of the problem [18][19][24][23].
In the following, we want to describe and interpret some patterns visible throughout. We begin with the three examples, where the libxsmm generator driver did not perform target-specific optimizations (noarch).
Figure 3.1 shows the test results on the local machine (see subsection 3.3.1). We can see BLIS consistently being the slowest implementation tested, up to a problem size of $m = n = k = 16$. Afterward, BLIS performance starts to get competitive, getting closer to the Just-In-Time libxsmm kernel, finally beating it by a tiny margin at $m = n = k = 128$. For smaller problem sizes, we can see that first Eigen and then the JIT libxsmm kernel mostly prevail.
Looking at CoolMuc-2 (see subsection 3.3.2) with general (noarch) libxsmm-generated code (see Figure 3.2), we can again see BLIS becoming competitive only at $m = n = k = 16$ and then getting very close to the JIT libxsmm code. In the smaller problem sizes, Eigen again mostly prevails, while the JIT libxsmm kernel becomes competitive for slightly bigger m, n, and k.
As with the previous examples where the libxsmm-generated code was not target-specific, in the CoolMuc-3 example in Figure 3.3, BLIS again gets competitive at a problem size of $m = n = k = 16$, however this time, Eigen is almost always the fastest before this point.
Looking at the two sets of tests, where the libxsmm generator driver optimized the code for a specific target microarchitecture like Haswell (Figure 3.4), or Knights Landing (Figure 3.5), the results are clear: the libxsmm-generated code is the fastest for every

tested problem size, without exceptions, while BLIS again only begins to become competitive regarding the other implementations at a larger problem size. The case between Eigen and the libxsmm JIT kernel is like in the other CoolMuc-2/-3 examples before, as expected [28][29].
Overall, we can conclude the experiment: the hypothesis from section 3.1 holds in our cases, BLIS only performs competitively with larger matrix multiplication problems, which are not very prevalent in the ADER-DG-solver we are working on. However, by comparing multiple different implementations offering matrix multiplication, we got to see that there are cases in which the code generated by the libxsmm generator driver does not offer optimal performance. This is generally the case when the generated code is not optimized for the specific platform we are working on (Figures 3.1 through 3.3). While not a realistic scenario in itself on CoolMuc-2 and -3, the comparison is still interesting, since there are modern microarchitectures that the libxsmm code generator driver does not support, like AMD Zen 2 (on the local machine from subsection 3.3.1) or newer Intel microarchitectures [39].
This shows potential for improvement on some microarchitectures, especially with Eigen and the JIT libxsmm version. BLIS does not offer competitive performance for the matrix sizes typically used in the solver, however, it might implement faster algorithms for smaller matrixes in the future.

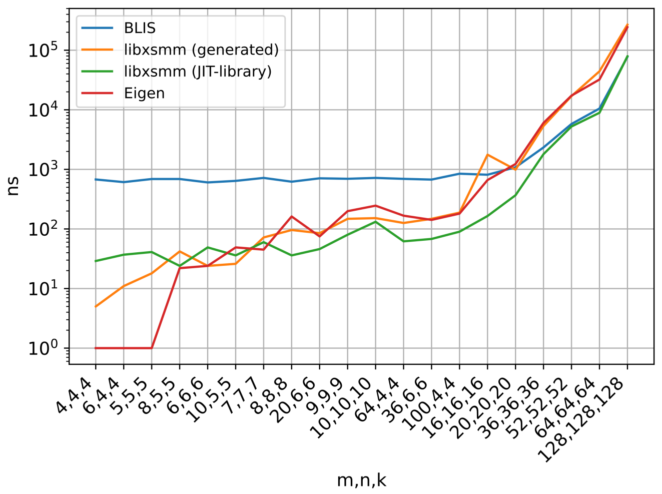

Figure 3.1.: Comparison of computation times [y-axis, logarithmic scale] of different implementations performing matrix multiplications with different sized matrixes [x-axis]; On the local environment (see subsection 3.3.1); The libxsmm-generated code is optimized for noarch (no target specific optimizations performed by the generator, but possibly done by the compiler)

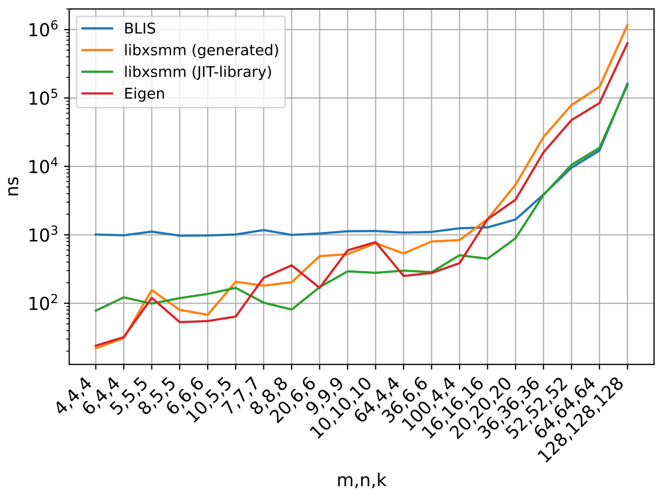

Figure 3.2.: Comparison of computation times [y-axis, logarithmic scale] of different implementations performing matrix multiplications with different sized matrixes [x-axis]; On the CoolMuc-2 environment (see subsection 3.3.2); The libxsmm-generated code is optimized for noarch (no target specific optimizations performed by the generator, but possibly done by the compiler)

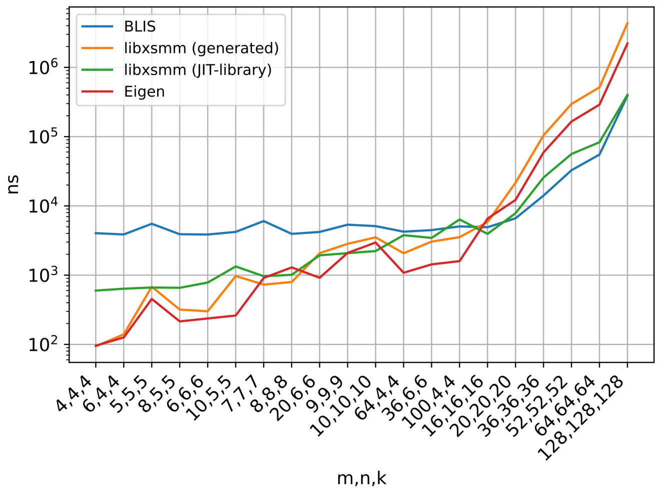

Figure 3.3.: Comparison of computation times [y-axis, logarithmic scale] of different implementations performing matrix multiplications with different sized matrixes [x-axis]; On the CoolMuc-3 environment (see subsection 3.3.3); The libxsmm-generated code is optimized for noarch (no target specific optimizations performed by the generator, but possibly done by the compiler)

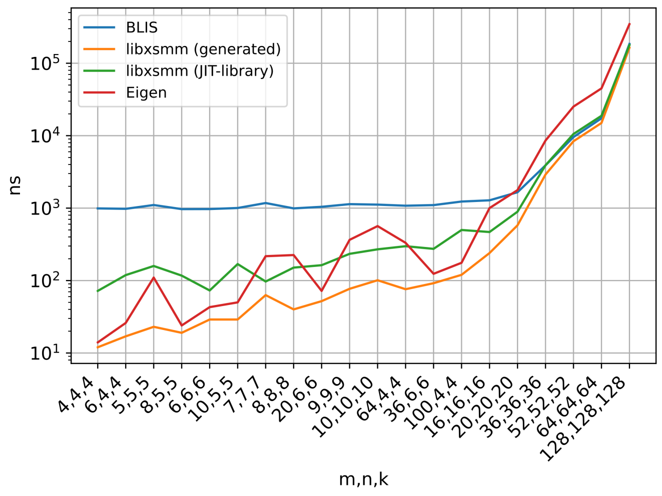

Figure 3.4.: Comparison of computation times [y-axis, logarithmic scale] of different implementations performing matrix multiplications with different sized matrixes [x-axis]; On the CoolMuc-2 environment (see subsection 3.3.2); The libxsmm-generated code is optimized for Haswell [28]

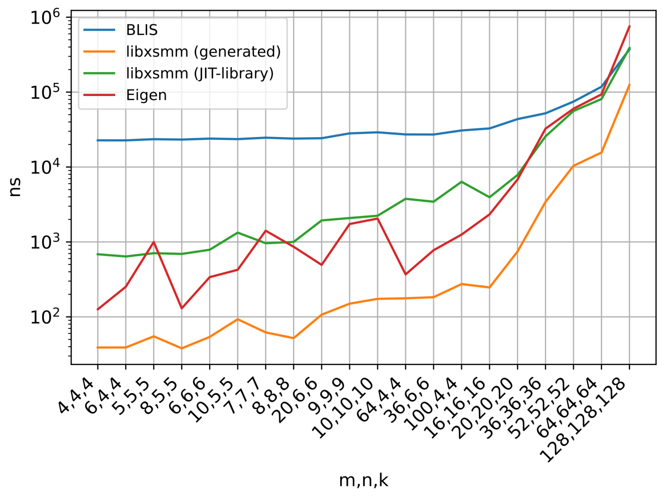

Figure 3.5.: Comparison of computation times [y-axis, logarithmic scale] of different implementations performing matrix multiplications with different sized matrixes [x-axis]; On the CoolMuc-3 environment (see subsection 3.3.3); The libxsmm-generated code is optimized for Knights Landing [29]

3.5. Reasons for the Observed Behavior

We compare the performance profiling for a matrix multiplication problem with $m = n = k = 4$, iterated 10 million times, on the local machine (see subsection 3.3.1, with the `-O3` and `-DNDEBUG` compiler options enabled [18][19]), using either BLIS or libxsmm-generated code to get an idea as to why BLIS has a relatively lower performance for smaller matrix multiplication problems. We can see that the version of the code using the libxsmm-generated matrix multiplications (Figure 3.6) spends almost 99% of its computation time on the core matrix multiplication function (`gemm_4_4_4`) according to the samples taken by the profiler. Contrary to that, the BLIS version (Figure 3.7) spends a lot of computation time on different functions. This may show that BLIS has more initial overhead compared to other implementations, which can make it slower for smaller matrix multiplication problems. That initial overhead seems to be amortized by faster computation times afterward for larger matrix multiplication problems, as indicated by the performance test results in Figures 3.1 through 3.5. We again use a performance profiler to show this in a similar test case with $m = n = k = 128$ and 100000 iterations. The results (see Figure 3.8) strongly indicate superior utilization at that size compared to the smaller size.
We can explain the performance differences between Eigen/the JIT version of libxsmm and the libxsmm-generated code because, without microarchitecture-specific optimizations enabled at generation-time, the libxsmm-generated code essentially is a default matrix multiplication implementation (see Figure 2.1) only optimized by the compiler. Contrary to that, in case the libxsmm code generator driver does optimize for the platform, we see it beat Eigen/the JIT version of libxsmm, presumably because it can generate highly optimized platform-specific assembly instructions.
We observe the libxsmm JIT code often beat Eigen at higher matrix sizes, likely because the Just-In-Time kernel generation allows it to better adapt to the underlying microarchitecture, however, comes with some initial performance overhead that is not amortized in smaller cases.

```
98,61%  gemm_4_4_4
 1,09%  main
 0,13%  do_lookup_x
 [...]
```

Figure 3.6.: Results of the performance profiling (using perf [40]) of the libxsmm-generated matrix multiplication called ten million times with $m = n = k = 4$; compiled with `-fno-inline` [18] for more precise results; both rows and columns have been shortened to give a rough overview

```
16,36%  bli_thrinfo_split
10,06%  bli_dgemm_ex
 9,59%  bli_pool_checkin_block
 8,84%  bli_gemmsup_ref_var2m
 4,68%  bli_l3_basic_check
 2,79%  bli_dgemmsup_rv_haswell_asm_4x4
 2,31%  bli_thrinfo_free
 2,30%  bli_pool_checkout_block
 2,19%  bli_pthread_switch_on
 2,09%  pthread_mutex_lock@@GLIBC_2.2.5
 1,82%  bli_l3_sup_thrinfo_create
 1,78%  bli_dgemmsup_rv_haswell_asm_6x4m
 [...]
```

Figure 3.7.: Results of the performance profiling (using perf [40]) of the BLIS matrix multiplication called ten million times with $m = n = k = 4$; both rows and columns have been shortened to give a rough overview

```
95,42%  bli_dgemmsup_rv_haswell_asm_6x8m
 3,58%  bli_dgemmsup_rv_haswell_asm_2x8
 0,18%  bli_gemmsup_ref_var2m
 [...]
```

Figure 3.8.: Results of the performance profiling (using perf [40]) of the BLIS matrix multiplication called 100000 times with $m = n = k = 128$; both rows and columns have been shortened to give a rough overview

4. Including the New Implementations in the ExaHyPE 2 ADER-DG-Solver

In the performance tests, we compared Eigen, the JIT version of libxsmm, and BLIS to the (already utilized) libxsmm code generator driver. We have seen that it can be beneficial to performance in some cases to have one of the newer implementations handle the matrix multiplications. Consequently, all three of them are included as options in the code-generation procedures for the ADER-DG-solver. That way, the user can choose the best implementation to perform their computations even when working on a microarchitecture where the newer implementations show more potential than the libxsmm-generated matrix multiplications. However, since there are a lot of situations where the existing implementation performs better, we will also leave it as an option.
As discussed, ExaHyPE 2 is part of the Peano project and generates C++ code. In our case, that code is generated by the ExaHyPE 2 ADER-DG kernel generator. It uses Jinja2, a template engine to generate the user-specific code and surrounding files. The driver code, which for example determines what is plugged into the templates is written in Python. Some steps need to be taken to get the code generator to utilize the new implementations [42].
We begin by introducing an optimization option for each new implementation we add. They allow the users to have the resulting program optimized to their choosing. The options are first added to the `ADERDG` class as variables that can be set using the `add_kernel_optimizations` function. These attributes then get passed on to the ExaHyPE 2 ADER-DG kernel generator, more specifically the `generate_aderdg_kernels` function located inside `generator.py`. From thereon, it is packed into the input configuration dictionary (`InputConfig`). In Python, a dictionary is an associative array storing key-value pairs. That makes it a suitable choice for storing options passed to the generator in an easily accessible form, while still making it simple to transfer them onward to other functions or classes [52][44].
Afterward, the dictionary containing the configuration is passed on to the `Controller`. It uses these values and those obtained from the `Configuration` (which also has options for BLIS, Eigen, and libxsmm JIT added to it) to create the configuration passed to the actual models. These models (derived from `AbstractModelBaseClass`) implement the `generateCode` method to instantiate the Jinja2-templates using the configuration

provided [52].
A relevant example of this is the `FusedSpaceTimePredictorVolumeIntegralModel`. Given the configuration, it selects among multiple Jinja2-templates and then renders one of them into C++ code. Jinja2 has access to the configuration inside these templates, allowing it to adjust the code to the requested options. Among other models, the `FusedSpaceTimePredictorVolumeIntegralModel`'s templates use the matrix multiplication macro (in `macros.template` and `matmul.template`) [52].
Inside the macro, we add multiple implementations for the different frameworks/libraries, in addition to the existing ones (using the libxsmm code generator driver and without the use of any frameworks/libraries) to perform the matrix multiplication. Jinja2 selects the implementation according to the configuration. For example, when the configuration has the `useBLIS` option set to `True`, the BLIS implementation is chosen. The `matmulkey`, a key for a dictionary of matrix multiplication configurations has to be passed along with the locations of the matrixes A, B, and C, some additional and some optional arguments each time the macro is used. The entry (which is a `MatmulConfig`) in the dictionary corresponding to the key contains more information for the specific matrix multiplication operation that shall be performed with the use of the macro. This information includes α, β, matrix sizes, and leading dimensions. The dictionary has to be generated beforehand, which is done in the `buildGemmsConfig` function of every model class that uses matrix multiplications in the respective template [52].

4.1. The Matrix Multiplication Template Macro

We now want to address the specific implementations inside the macro further, since they set the options apart. As mentioned, we want to add support for the BLIS, Eigen, and libxsmm JIT matrix multiplications, while keeping the two existing options. When the user has chosen an implementation via the configuration, a specific block of code is generated. We want to explore the particular actions taken for each one more closely. For all options, the locations of A, B, and C are determined using the parameters of the macro, in the way that was already implemented with the libxsmm code generator driver [52].

4.1.1. Using BLIS

As before, we choose the typed API for BLIS. This makes it simple to adapt the macro. Since the floating point precision can vary, we have to determine it beforehand. That is done by reading the information inside the `MatmulConfig` corresponding to the given key. Depending on that, we choose the name of the BLIS gemm function to call: `bli_dgemm` for double and `bli_sgemm` for single precision matrix multiplications. The arguments are similar to what we described in section 2.1. We do not transpose the matrixes, so `transa` and `transb` are set to `BLIS_NO_TRANSPOSE`. We obtain `m`, `n`, and `k` from the appropriate matrix multiplication configuration, just like `alpha` and `beta`. Since BLIS requires `alpha` and `beta` to be passed as a pointer, we declare constants for zero and one (the only valid options for α and β), from which we (depending on α and β) pass the addresses to the function. As in the experiment, the column-stride parameters (`csa`, `csb`, and `csc`) are set to the according leading dimensions fetched from the configuration. Row-strides (`rsa`, `rsb`, and `rsc`) are always `1`, as before [5].

4.1.2. Using Eigen

For Eigen, the macro is a little more extensive. We declare a `Map` for each of the three operand matrixes. `Map`s are a feature of Eigen, which allow for an existing C-style array to be used with Eigen. Since the raw buffers we use here are also just pointers, they are compatible. To create the `Map`s, we need to pass a `Matrix` template parameter. The `Matrix` itself also requires template arguments, which we provide after getting them from the configuration: the precision type (`double` or `float`) and the size. Next, we set the alignment to `Unaligned`, even though information on alignment is partially provided by the configuration (for matrixes A and C). This is done because using it according to the configuration caused errors due to aligned access on unaligned memory for us. Information about the memory layout of the matrixes also has to be included, we use the `Stride` class for that. It allows us to pass the inner- and outer stride. For a column-major matrix (which is the default for Eigen) the inner stride "is the pointer increment between two consecutive entries" [12], while the outer stride "is the pointer increment between two consecutive [...] columns" [12]. That means the inner stride is `1` and the outer stride is the leading dimension, which we obtain from the matrix multiplication configuration. The only non-template argument the `Map` instantiation takes is the pointer to the buffer. We get that from the macro parameters. Afterward, we need to perform the actual matrix multiplication using the `*` operator. Since using operators is not BLAS-like we need to find another way to incorporate α and β. We obtain `alpha` and `beta` from the configuration and generate the code depending on the values. In case `alpha` is one, we don't multiply `A*B` by any scalar. Otherwise, we multiply the result of `A*B` by `alpha`. In case `beta` is zero, we use the `=` operator to assign the result of `A*B` to `C`, otherwise, we use `+=` to add it to `C` [12][13][17][12][15].

4.1.3. Using libxsmm JIT

The libxsmm JIT matrix multiplication calls themselves are the same as before with the libxsmm-generated code, hence we use the same code for the macro. However, some changes in other places need to be made. The macro chooses the matrix multiplication function name based on the baseroutinename of the configuration. There it plugs in the pointers to the buffers for A, B, and C, which are obtained from the macro parameters [52].

4.2. Adjusting the Preamble of the Templates

The files that use the new implementations for matrix multiplications need to have some additional code at the beginning, e.g. to include header files.
For BLIS that means adding `#include "blis.h"` and the two possible options for `alpha` and `beta` (zero or one), since they have to be passed as a pointer. We declare two pairs of them, one of them as `double`s and the other one as `float`s. They are marked with the appropriate suffix `_d` and `_s` and chosen by the matrix multiplication macro according to the floating point precision of the operation.
For Eigen, we just need to include the header (`#include <Eigen/Dense>`).
To increase performance with the libxsmm JIT version, we have it generate all the kernels in the beginning (during runtime) and reuse them every time we need them. To do that, we iterate through all the matrix multiplication configurations used. Based on the information obtained for each configuration we declare three variables for the leading dimensions and one for the flags (this information needs to be passed as a pointer). The relevant flag we use is for β. In case it is zero, we set the value of the flags to four, otherwise, we set it to zero. $\alpha = 1$ is the only case supported by the libxsmm JIT kernels, hence $\alpha = 1$ is assumed and not included as a flag. However, that is not a big problem since the currently used libxsmm code generator driver has the same restriction, and setting α to zero would not make sense in most cases, performing $C := \beta \cdot C$. We use these variables and the matrix sizes to get a libxsmm matrix multiplication kernel (`libxsmm_dmmfunction` or `libxsmm_smmfunction`, depending on the precision) from the `libxsmm_dmmdispatch_v2` or `libxsmm_smmdispatch_v2` functions, respectively. The kernels are named after the baseroutinename from the configuration. This functionality is outsourced to a macro. In addition to that, we include the corresponding header (`#include <libxsmm.h>`) [37][38][35][39].

4.3. Adapting the Environment

For the resulting program to compile, we need to have the appropriate libraries available to include/link to. This is why we create new configuration options (`--with-BLIS`, `--with-Eigen`, and `--with-libxsmmJIT`) in the Peano `configure.ac` script, which can be invoked from the `./configure` command. For each of these options, we clone the

```
bli_{{precisionPrefix}}gemm(BLIS_NO_TRANSPOSE, BLIS_NO_TRANSPOSE, {{M
    }}, {{N}}, {{K}}, {% if alpha == 1 %}&one_{{precisionPrefix}}{%
    elif alpha == 0 %}&zero_{{precisionPrefix}}{% endif %}, {{A}}{% if
     A_shift != '0' %}+{{A_shift}}{% endif %}, 1, {{LDA}}, {{B}}{% if
    B_shift != '0' %}+{{B_shift}}{% endif %}, 1, {{LDB}}, {% if beta
    == 1 %}&one_{{precisionPrefix}}{% elif beta == 0 %}&zero_{{
    precisionPrefix}}{% endif %}, {{C}}{% if C_shift != '0' %}+{{
    C_shift}}{% endif %}, 1, {{LDC}});
```

Figure 4.1.: The templated BLIS matrix multiplication call; `precisionPrefix` is either `d` or `f` for `double` or `float`

```
Eigen::Map<Eigen::Matrix<{{precisionType}}, {{M}}, {{K}}>, Eigen::
    Unaligned, Eigen::Stride<{{LDA}}, 1>> A({{A}}{% if A_shift != '0'
    %}+{{A_shift}}{% endif %});
Eigen::Map<Eigen::Matrix<{{precisionType}}, {{K}}, {{N}}>, Eigen::
    Unaligned, Eigen::Stride<{{LDB}}, 1>> B({{B}}{% if B_shift != '0'
    %}+{{B_shift}}{% endif %});
Eigen::Map<Eigen::Matrix<{{precisionType}}, {{M}}, {{N}}>, Eigen::
    Unaligned, Eigen::Stride<{{LDC}}, 1>> C({{C}}{% if C_shift != '0'
    %}+{{C_shift}}{% endif %});
C {{eigenOp}} {% if alpha != 1%}{{alpha}} * (A * B){% else %}A * B{%
    endif %};
```

Figure 4.2.: The templated Eigen matrix multiplication call; `precisionType` is either `double` or `float`; `eigenOp` is either += or =, according to `beta`

```
{{conf.baseroutinename}}({{A}}{% if A_shift != '0' %}+{{A_shift}}{%
    endif %}, {{B}}{% if B_shift != '0' %}+{{B_shift}}{% endif %}, {{C
    }}{% if C_shift != '0' %}+{{C_shift}}{% endif %});
```

Figure 4.3.: The templated libxsmm JIT matrix multiplication call, which is equal to the existing call for the libxsmm-generated code [52]

```
{% for config in matmulConfigs %}
[...]
const int flags_{{matmulConfig.baseroutinename}}({% if matmulConfig.
    beta == 1 %}0{% else %}4{% endif %}); // 0 for beta = 1, 4 for
    beta = 0
libxsmm_{{precisionPrefix}}mmfunction {{matmulConfig.baseroutinename}}
     = libxsmm_{{precisionPrefix}}mmdispatch_v2({{matmulConfig.M}}, {{
    matmulConfig.N}}, {{matmulConfig.K}}, &lda_{{matmulConfig.
    baseroutinename}}, &ldb_{{matmulConfig.baseroutinename}}, &ldc_{{
    matmulConfig.baseroutinename}}, &flags_{{matmulConfig.
    baseroutinename}});
{% endfor %}
```

Figure 4.4.: Part of the templated libxsmm JIT preamble macro (shortened)

appropriate repository into the Peano `submodules` folder. For BLIS and Eigen that is just their source repository (see [3] and [14]). BLIS also needs to be configured (automatically by invoking `./configure auto`) and built using `make`. We also test BLIS' functionality using `make check`, notifying the user in case something went wrong. Since the JIT version of libxsmm may fall back to a BLAS implementation for certain matrix multiplication cases, we need to provide a BLAS interface in addition to the libxsmm source repository (see [35]). We choose OpenBLAS (see [41]) since it automatically adapts the compilation process to the environment it is run on, simply requiring an invocation of the `make` command. For comparison: LAPACK, which also offers a BLAS interface, demands the specification of a configuration beforehand. We also need to build libxsmm using `make`. As a side note: We cannot reuse the libxsmm version cloned by the other libxsmm option (used for the code generator driver) as it is older than the one we are using here. Everything we described in this section up to this point is done automatically by the `configure.ac` script [3][32][41][35][36].
The `Makefile` for compiling the program also needs to be adapted to include/link to the appropriate resources, which we placed inside the `submodules` directory while performing the configuration. Using the `Makefile` class, we can modify it. The class provides suitable methods, such as one to add a search path for header files. We call these methods from the ExaHyPE 2 `Project` class, where we have access to the solvers used. In case one of them is an ADER-DG-solver, we take actions according to the selected optimizations: With BLIS, we need to add the path to the BLIS include directory as a header search path, link to `libblis.a` and add `-lpthread` (see Figure 4.5). Eigen only requires adding the repository as a header search path, while the libxsmm JIT

version needs to have its include directory added as a header search path, `libxsmm.a` and `libopenblas.a` (from OpenBLAS) linked to, as well as having `-lpthread` added [52][39].

```
self._project.output.makefile.add_header_search_path("␣" +
    pathToExaHyPERoot + "/submodules/blis/include/" + archname)
self._project.output.makefile.add_linker_flag(pathToExaHyPERoot + "/
    submodules/blis/lib/" + archname + "/libblis.a")
self._project.output.makefile.add_library("-lpthread")
```

Figure 4.5.: Adding the required options to the `Makefile` in the BLIS case

4.4. Performance Results for the New Implementation

For some concrete comparisons between the final generated versions with the different matrix multiplication implementations, we again run some examples (without target-specific optimizations enabled, `architecture="noarch"`) on the local and CoolMuc-2 machines (see subsection 3.3.1 and 3.3.2) and interpret the results based on the ranking of the computation times. We do not expect as much of a difference between the various implementations as we saw in section 3.4 because the final program consists of more than just matrix multiplications. The results are prepared in Tables 4.1 and 4.2. We begin with the SWE example (with matrix sizes $m = n = k = 4$) from section 2.5. The results for that example are in line with our expectations, as the way they are ranked is equal to Figures 3.1 and 3.2, considering the matrix sizes.
For the elastic example, working with matrix sizes of $m = 9$ and $n = k = 6$, the results also mostly fit in with the performance results from Figures 3.1 and 3.2.
As one final comparison, we use the computation times for the ADER-DG Acoustic example (with $m = n = k = 4$, located at `applications/exahype2/aderdg/Acoustic`), with the end time extended to 0.015 seconds. Again, the performance order of these implementations is as expected (regarding Figures 3.1 and 3.2).
A version target-optimized for Haswell (by passing the `architecture="hsw"` option to `add_kernel_optimizations`) is also run in the same CoolMuc-2 environment (see Table 4.3). It yields results that partially contradict the ones we got in section 3.4. While we expect the libxsmm-generated code to outperform all other implementations in cases where it is optimized for the underlying microarchitecture (indicated by Figure 3.4), we see Eigen outperform it in the SWE and Acoustic example. The elastic example is the only one where the results are as expected: a substantial performance increase

from the general to the target-specific version. Since individual matrix multiplication performance improves drastically with target-specific optimizations enabled for the libxsmm code generator driver, we do not know how to attribute this inconsistency. The fact that turning on these target-specific optimizations also changes other aspects of the program apart from the matrix multiplications may contribute to the observed results. In this regard, it should also be noted that many of the performance results for the other implementations also became slightly worse when compared to Table 4.2. Appendix A describes how these examples can be recreated.

implementation / example	BLIS	Eigen	libxsmm JIT	libxsmm (generated for noarch)
SWE	55.3	14.6	17.5	15.7
Elastic	6.2	3.2	3.2	3.7
Acoustic	88.8	65.9	68.3	68.1

Table 4.1.: Computation times in seconds for the SWE, Elastic, and Acoustic examples run on the local machine (see subsection 3.3.1) with different matrix multiplication implementations enabled as optimization options; The code has no microarchitecture-specific optimizations enabled

implementation / example	BLIS	Eigen	libxsmm JIT	libxsmm (generated for noarch)
SWE	89.1	20.5	24.8	23.0
Elastic	10.0	4.9	4.8	5.6
Acoustic	138.6	100.8	103.9	101.9

Table 4.2.: Computation times in seconds for the SWE, Elastic, and Acoustic examples run on the CoolMuc-2 machine (see subsection 3.3.2, we use `icpx 2022.1.0` instead of `icpc` [25] and pass `CXXFLAGS="-std=c++20"` during configuration) with different matrix multiplication implementations enabled as optimization options; The code has no microarchitecture-specific optimizations enabled

implementation / example	BLIS	Eigen	libxsmm JIT	libxsmm (generated for hsw)
SWE	89.2	20.8	24.7	24.1
Elastic	9.9	5.2	5.1	5.0
Acoustic	140.2	101.7	106.3	103.6

Table 4.3.: Computation times in seconds for the SWE, Elastic, and Acoustic examples run on the CoolMuc-2 machine (see subsection 3.3.2, we use `icpx 2022.1.0` instead of `icpc` [25] and pass `CXXFLAGS="-std=c++20"` during configuration) with different matrix multiplication implementations enabled as optimization options; The code has microarchitecture-specific optimizations for Haswell enabled

5. Conclusion

Since we were not able to replace the deprecated libxsmm code generator driver with BLIS while keeping up performance, we extended our considerations to Eigen and the JIT version of libxsmm. They showed to be viable optimizations in certain scenarios, beating the existing implementation's performance in some cases, while falling behind in other cases. We consider the expansion a worthwhile improvement for the program, especially since turning on the specific optimization options in the code generator is as simple as with the existing libxsmm code generator driver version. The user only has to pass one flag during configuration and one additional boolean value while selecting the optimization options for ADER-DG. Considering the performance test results in section 3.4, we saw outcomes in the real application tests that were better (from the newer implementations' point of view) than we expected them to be.
We recommend users who want to try one of the new implementations make an attempt with the Eigen option in case the matrixes are a little smaller (e.g. $m = n = k = 4$). We suggest applying the JIT version of libxsmm for slightly bigger matrixes (e.g. $m = 9$, $n = k = 6$). For really large ones (e.g. $m = n = k = 36$ or similar), it might be worth it to use BLIS.
We were able to explore the main goal of this topic, although there certainly are branches that are worth researching in the future that we did not look into further. For instance, the tests were run sequentially and the libraries were introduced into the ExaHyPE 2 ADER-DG-solver in their non-parallelized versions. Since they all offer multithreaded options it would be interesting to explore how this influences relative performance between the libraries and overall program execution time. We briefly tried utilizing the multithreaded version of BLIS for small matrix multiplications (e.g. $m = n = k = 4$) however, we experienced even bigger performance setbacks than with the sequential BLIS version, so we did not pursue this approach further but instead tested other implementations for these matrix sizes. It would however be interesting to analyze the reason for the performance decrease and whether there is a way to improve it. [35][6][11].
Another adaptation we would like to make in the future is extending the CMake configuration (`CMakeLists.txt`) to also include options for cloning (and potentially building) the repositories required by the frameworks/libraries, just like we did with the `configure.ac` script. That would make it easier to use the new optimization options

with CMake [9].
Furthermore, it might be worth trying to determine what causes the alignment issues with Eigen mentioned in subsection 4.1.2. This could improve the performance of the Eigen library.
Investigating the reasons for the difference in performance when comparing the isolated matrix multiplications to the final results will very likely yield interesting results. One might also want to branch out and look into why the performance of the other implementations slightly decreased when turning on target-specific optimizations for Haswell.
Finally, though we were not able to efficiently use BLIS for the matrix sizes mainly present throughout the ExaHyPE 2 ADER-DG-solver, we very much hope to see BLIS release some kernels specifically optimized for smaller matrix multiplications. Provided they use the same API and function signatures, the BLIS-optimized version of the ExaHyPE 2 ADER-DG-solver would automatically be using them as soon as they are available in the BLIS git repository.

List of Figures

List of Tables

Bibliography

[1] Advanced Micro Devices, Inc. (AMD). *AMD Ryzen™ 5 5500U*. URL: `https://www.amd.com/en/product/10856` (visited on 07/05/2023).

[2] C. Bassi, S. Busto, and M. Dumbser. *High order ADER-DG schemes for the simulation of linear seismic waves induced by nonlinear dispersive free-surface water waves*. 2020. arXiv: `2004.03257 [math.NA]`. URL: `https://arxiv.org/abs/2004.03257` (visited on 08/05/2023).

[3] BLIS. *flame/blis: BLAS-like Library Instantiation Software Framework*. URL: `https://github.com/flame/blis` (visited on 07/03/2023).

[4] BLIS. *flame/blis: BLISObjectAPI.md*. URL: `https://github.com/flame/blis/blob/master/docs/BLISObjectAPI.md` (visited on 07/13/2023).

[5] BLIS. *flame/blis: BLISTypedAPI.md*. URL: `https://github.com/flame/blis/blob/master/docs/BLISTypedAPI.md#level-3-operations` (visited on 07/05/2023).

[6] BLIS. *flame/blis: BuildSystem.md*. URL: `https://github.com/flame/blis/blob/master/docs/BuildSystem.md` (visited on 07/03/2023).

[7] BLIS. *flame/blis: KernelsHowTo.md*. URL: `https://github.com/flame/blis/blob/master/docs/KernelsHowTo.md` (visited on 07/23/2023).

[8] D. E. Charrier and T. Weinzierl. "Stop talking to me - a communication-avoiding ADER-DG realisation." In: *CoRR* abs/1801.08682 (2018). arXiv: `1801.08682`. URL: `http://arxiv.org/abs/1801.08682` (visited on 07/12/2023).

[9] CMake. *CMake*. URL: `https://cmake.org/` (visited on 08/03/2023).

[10] doctest. *doctest/doctest: The fastest feature-rich C++11/14/17/20/23 single-header testing framework*. URL: `https://github.com/doctest/doctest` (visited on 07/05/2023).

[11] Eigen. *Eigen and multi-threading*. URL: `https://eigen.tuxfamily.org/dox/TopicMultiThreading.html` (visited on 08/03/2023).

[12] Eigen. *Eigen::Stride< OuterStrideAtCompileTime_, InnerStrideAtCompileTime_ > Class Template Reference*. URL: `https://eigen.tuxfamily.org/dox/classEigen_1_1Stride.html` (visited on 08/01/2023).

[13] Eigen. *Interfacing with raw buffers: the Map class*. URL: `https://eigen.tuxfamily.org/dox/group__TutorialMapClass.html` (visited on 08/01/2023).

[14] Eigen. *libeigen/eigen*. URL: `https://gitlab.com/libeigen/eigen` (visited on 07/19/2023).

[15] Eigen. *Matrix and vector arithmetic*. URL: `https://eigen.tuxfamily.org/dox/group__TutorialMatrixArithmetic.html` (visited on 08/01/2023).

[16] Eigen. *Storage orders*. URL: `https://eigen.tuxfamily.org/dox/group__TopicStorageOrders.html` (visited on 08/01/2023).

[17] Eigen. *The Matrix class*. URL: `https://eigen.tuxfamily.org/dox/group__TutorialMatrixClass.html` (visited on 08/01/2023).

[18] Free Software Foundation, Inc. *3.11 Options That Control Optimization*. URL: `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html` (visited on 08/07/2023).

[19] Free Software Foundation, Inc. *3.13 Options Controlling the Preprocessor*. URL: `https://gcc.gnu.org/onlinedocs/gcc/Preprocessor-Options.html` (visited on 08/08/2023).

[20] Free Software Foundation, Inc. *GCC, the GNU Compiler Collection*. URL: `https://gcc.gnu.org/` (visited on 07/09/2023).

[21] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst. "LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation." In: *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2016, pp. 981–991. DOI: `10.1109/SC.2016.83`. URL: `https://ieeexplore.ieee.org/document/7877162` (visited on 08/06/2023).

[22] V. Hermann. "ADER-DG - Analysis, further Development and Applications." Jan. 2011. URL: `http://nbn-resolving.de/urn:nbn:de:bvb:19-125403` (visited on 07/12/2023).

[23] Intel Corporation. *Intel® C++ Compiler Classic Developer Guide and Reference - D*. URL: `https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-10/d.html` (visited on 08/08/2023).

[24] Intel Corporation. *Intel® C++ Compiler Classic Developer Guide and Reference - O*. URL: `https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-10/o-001.html` (visited on 08/08/2023).

[25] Intel Corporation. *Intel® oneAPI DPC++/C++ Compiler*. URL: `https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html` (visited on 07/10/2023).

[26] Intel Corporation. *Intel® Xeon Phi™ Processor 7210F*. URL: `https://ark.intel.com/content/www/us/en/ark/products/94709/intel-xeon-phi-processor-7210f-16gb-1-30-ghz-64-core.html` (visited on 07/10/2023).

[27] Intel Corporation. *Intel® Xeon® Processor E5-2697 v3*. URL: `https://ark.intel.com/content/www/us/en/ark/products/81059/intel-xeon-processor-e52697-v3-35m-cache-2-60-ghz.html` (visited on 07/10/2023).

[28] Intel Corporation. *Products formerly Haswell*. URL: `https://ark.intel.com/content/www/us/en/ark/products/codename/42174/products-formerly-haswell.html` (visited on 07/20/2023).

[29] Intel Corporation. *Products formerly Knights Landing*. URL: `https://ark.intel.com/content/www/us/en/ark/products/codename/48999/products-formerly-knights-landing.html` (visited on 07/20/2023).

[30] International Business Machines Corporation (IBM). *How Leading Dimension Is Used for Matrices*. 2021. URL: `https://www.ibm.com/docs/en/essl/6.3?topic=matrices-how-leading-dimension-is-used` (visited on 06/30/2023).

[31] Kitware, Inc. *ParaView*. URL: `https://www.paraview.org/` (visited on 07/05/2023).

[32] LAPACK. *Reference-LAPACK/lapack*. URL: `https://github.com/Reference-LAPACK/lapack` (visited on 08/02/2023).

[33] Leibniz-Rechenzentrum. *CoolMUC-2*. URL: `https://doku.lrz.de/coolmuc-2-11484376.html` (visited on 07/10/2023).

[34] Leibniz-Rechenzentrum. *CoolMUC-3*. URL: `https://doku.lrz.de/coolmuc-3-11484375.html` (visited on 07/10/2023).

[35] libxsmm. *Libxsmm/libxsmm: Library for specialized dense and sparse matrix operations, and deep learning primitives*. URL: `https://github.com/libxsmm/libxsmm` (visited on 06/25/2023).

[36] libxsmm. *libxsmm/libxsmm: libxsmm_mm.md*. URL: `https://github.com/libxsmm/libxsmm/blob/main_stable/documentation/libxsmm_mm.md` (visited on 08/02/2023).

[37] libxsmm. *libxsmm/libxsmm: libxsmm_typedefs.h*. URL: `https://github.com/libxsmm/libxsmm/blob/main_stable/include/libxsmm_typedefs.h` (visited on 08/01/2023).

[38] libxsmm. *libxsmm/libxsmm: libxsmm.h*. URL: `https://github.com/libxsmm/libxsmm/blob/main_stable/include/libxsmm.h` (visited on 08/01/2023).

[39] libxsmm. *libxsmm/libxsmm: libxsmm_be.md*. URL: `https://github.com/libxsmm/libxsmm/blob/main_stable/documentation/libxsmm_be.md` (visited on 07/05/2023).

[40] Linux Kernel Organization, Inc. *perf: Linux profiling with performance counters*. URL: `https://perf.wiki.kernel.org/index.php/Main_Page` (visited on 07/24/2023).

[41] OpenBLAS. *xianyi/OpenBLAS*. URL: https://github.com/xianyi/OpenBLAS (visited on 08/02/2023).

[42] Pallets. *Jinja*. URL: https://jinja.palletsprojects.com/en/3.1.x/ (visited on 07/19/2023).

[43] S. E. Peters and M. McClennen. "The Paleobiology Database application programming interface." In: *Paleobiology* 42.1 (2016), pp. 1–7. DOI: 10.1017/pab.2015.39. URL: https://www.cambridge.org/core/journals/paleobiology/article/paleobiology-database-application-programming-interface/4D20F5CAFA1B0AC7033975418668D82B (visited on 07/08/2023).

[44] Python Software Foundation. *5. Data Structures*. URL: https://docs.python.org/3/tutorial/datastructures.html (visited on 08/01/2023).

[45] Red Hat. *Fedora Linux 37*. URL: https://docs.fedoraproject.org/en-US/releases/f37/ (visited on 07/10/2023).

[46] A. Reinarz, D. E. Charrier, M. Bader, L. Bovard, M. Dumbser, K. Duru, F. Fambri, A.-A. Gabriel, J.-M. Gallard, S. K"oppel, L. Krenz, L. Rannabauer, L. Rezzolla, P. Samfass, M. Tavelli, and T. Weinzierl. "ExaHyPE: An engine for parallel dynamically adaptive simulations of wave problems." In: *Computer Physics Communications* 254 (2020), p. 107251. ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2020.107251. URL: https://www.sciencedirect.com/science/article/pii/S001046552030076X (visited on 08/06/2023).

[47] SUSE. *SUSE Linux Enterprise Server 12 SP5*. URL: https://www.suse.com/releasenotes/x86_64/SUSE-SLES/12-SP5/index.html (visited on 07/10/2023).

[48] SUSE. *SUSE Linux Enterprise Server 15 SP1*. URL: https://www.suse.com/releasenotes/x86_64/SUSE-SLES/15-SP1/index.html (visited on 07/10/2023).

[49] The Matplotlib development team. *Matplotlib: Visualization with Python*. URL: https://matplotlib.org/ (visited on 07/19/2023).

[50] The Open Group. *IEEE Portable Applications Standards Committee*. URL: https://collaboration.opengroup.org/external/pasc.org/plato/ (visited on 07/09/2023).

[51] F. G. Van Zee and R. A. van de Geijn. "BLIS: A Framework for Rapidly Instantiating BLAS Functionality." In: *ACM Trans. Math. Softw.* 41.3 (June 2015). ISSN: 0098-3500. DOI: 10.1145/2764454. URL: https://doi.org/10.1145/2764454 (visited on 07/03/2023).

[52] T. Weinzierl. *hpcsoftware/Peano: Peano*. URL: https://gitlab.lrz.de/hpcsoftware/Peano (visited on 07/05/2023).

[53] T. Weinzierl. “The Peano Software—Parallel, Automaton-Based, Dynamically Adaptive Grid Traversals.” In: *ACM Trans. Math. Softw.* 45.2 (Apr. 2019). ISSN: 0098-3500. DOI: `10.1145/3319797`. URL: `https://doi.org/10.1145/3319797` (visited on 08/05/2023).

A. Running the Examples Provided

We provide the examples we tested in section 4.4 (adapted from the ones included in Peano [52]) in a fork of the Peano repository. It can be cloned from `https://gitlab.lrz.de/landojahn/Peano`. To run them, execute the following commands (starting in the base folder, some tools may have to be installed first):

```
git checkout DG
libtoolize; aclocal; autoconf; autoheader;
cp src/config.h.in .;
automake --add-missing
```

The user can then select the optimization options they want to have available by means of the `./configure` command. Add the desired optimizations (`--with-BLIS`, `--with-Eigen`, `--with-libxsmmJIT`, or `--with-libxsmm`, there may be multiple options enabled at the same time) to the base command (`./configure --enable-exahype --enable-blockstructured --enable-loadbalancing`). An invocation and subsequent build may look like this (enabling the use of BLIS):

```
./configure --enable-exahype --enable-blockstructured --enable-
    loadbalancing --with-BLIS
make -j8
```

Then, choose the example from `applications/exahype2/aderdg/`. There are four directories with examples that have certain optimizations enabled. They are called `[optimization]_examples`. Choose an optimization for which the prerequisites have been enabled during configuration and subsequently decide on an example. For instance, choosing BLIS and SWE leads to the following command invocation:

```
cd applications/exahype2/aderdg/
cd BLIS_examples/SWE/
```

From there, set the `PYTHONPATH` environment variable to `../../../../../python/` and run the Python script in the directory using `python3`. For example:

```
export PYTHONPATH=../../../../../python/
python3 SWE.py
```

The program is then generated and built. For the same example, it is run and the solution is rendered using:

```
./SWE
pvpython ../../../../../python/peano4/visualisation/render.py solution-SWE.peano-patch-file
```

The resulting file with the extension `.pvd` can then be opened in ParaView.