ABSTRACT

Approaching the exascale era, scientific codes are forced to adapt in order to harness the computational resources. Research groups typically do not develop an application code completely on their own, but use well-optimised libraries as back end. ExaHyPE as a solver engine for hyperbolic PDEs addresses this need and will provide scalable and efficient kernels for large-scale computer systems. A key step towards a high-performance code is achieving good single-core performance. Hardware-aware programming, smart memory layouts and vectorisation provide the means to it. For this purpose, ExaHyPE employs a code generator, which couples driver routines with back end assembly code generation. This report documents the current status of the code generator and presents first performance results.

1. INTRODUCTION

The ExaHyPE project develops an engine that utilises the high-order discontinuous Galerkin (DG) finite element method [3, 2] in order to solve hyperbolic PDEs. It provides a foundation to simulate astrophysics or geoscience scenarios at exascale level. ExaHyPE comprises the following basic building blocks.

- **Toolkit.** The toolkit acts as the user interface. It reads in a specification file, selects solver kernels and outputs glue code. The specification file contains parameters such as the computational domain, the order of the DG polynomial, and optimisation options.

- **Adaptive Mesh Management.** The adaptive Cartesian grid is implemented by the Peano framework [6].

- **Solver Kernels.** As part of an engine, there is a stock of generic, non-optimised compute kernels. A suited subset is selected and combined through the glue code of the toolkit. This subset is selected in accordance with the specification file. In particular, the subset depends on the type of problem (linear/non-linear PDE) specified by the user.

- **Code Generator.** The code generator outputs optimised solver kernels for one particular specification file. For this set-up, the code generator decides on the memory layout, computation schemes, makefile options and prefetching strategies.

ExaHyPE has to provide efficient solver kernels in order to utilise the computational resources of large-scale computer systems. The code generator of the ExaHyPE engine provides a means to replace the generic, non-optimised solver kernels with optimised ones. The code generator is therefore the key ingredient for achieving good single-core performance.

Section 2 introduces the ADER discontinuous Galerkin (DG) method as the numerical scheme. Section 3 discusses the current status of the code generator and presents first performance results.

2. ADER DG SCHEME

The ExaHyPE engine addresses hyperbolic conservation laws in 2 or 3 space dimensions. In the remainder of this report, the focus is on nonlinear hyperbolic conservation laws of the form

$$\frac{\partial Q}{\partial t} + \nabla \cdot F(Q) = 0, \quad x \in \Omega, t \in \mathbb{R}_{\geq 0}$$

subject to an initial condition $Q(x, 0)$, where $x \in \Omega$, and boundary conditions $Q(x, t) = Q_{\text{Bnd}}(x, t)$, where $x \in \partial \Omega$. Here $Q \in \mathbb{R}^V$ denotes the $V$-valued vector of conserved quantities, $F(Q)$ a non-linear flux tensor, and $\Omega$ the two-dimensional and three-dimensional computational domain, respectively.

Following [3, 2], ExaHyPE employs a Cartesian grid to discretise the computational domain

$$\Omega = \bigcup_k T^k.$$ 

The elements $T^k$ are cubes in 3D or squares in 2D. ExaHyPE uses nodal DG to represent the solution approximation in each element through a polynomial of degree $N$

$$u^k_i(x, t^n) = \sum_{i=0}^{N} \Phi_i(x) \hat{u}_i^n, \quad x \in T^k.$$ 

The basis functions $\Phi_i(x)$ are given by the tensor product of the Lagrange interpolation polynomials evaluated at the Gauss-Legendre nodes; $\hat{u}_i^n$ denotes the degrees of freedom at time $t^n$. The direct sum of the element-wise DG polynomials gives the global discrete solution.

ExaHyPE employs a predictor-corrector scheme to evolve the solution. First, the predictor computes an element-local space-time DG solution $q_h^k$. Second, the corrector incorporates influences from neighbouring cells into the element-
local solution \( q_h \) and updates thereby the true DG solution \( u_h \).

For a unified numerical scheme, the elements \( T^k \) are mapped onto a space-time reference element. The spatial coordinates in the reference elements are \( \xi = (\xi, \eta) \) in 2D and \( \xi = (\xi, \eta, \zeta) \) in 3D, respectively; the temporal variable is \( t = t^n + \tau \Delta t \). Given \( d \) space dimensions, \( T_E = [0,1]^d \) describes the space reference element and \( T_E \times [0,1] \) the space-time reference element. In the reference element, the PDE reads as

\[
\frac{\partial Q}{\partial \tau} + \nabla \xi \cdot F'(Q) = 0
\]  

(2)

where \( \nabla \xi = \partial \xi / \partial x \cdot \nabla \) is the gradient with respect to the reference coordinates and \( F'(Q) = \Delta \xi (\partial \xi / \partial x)^T \cdot F(Q) \) is the volume flux in the reference element.

Multiplying the PDE in Equation (2) with space-time test functions \( \Theta_h = \Theta_h(\xi, \tau) \) and integrating over the space-time reference element yields the weak form

\[
\int_0^1 \int_{T_E} \Theta_h \frac{\partial Q}{\partial \tau} d\xi d\tau + \int_0^1 \int_{T_E} \Theta_h \nabla \xi \cdot F'_h(Q) d\xi d\tau = 0. \tag{3}
\]

### 2.1 Space-time Predictor

The space-time predictor computes a local solution in the space-time reference element. Given some space-time test functions \( \Theta(\xi, \tau) \), the predictor solution in reference coordinates is

\[
q_h(\xi, \tau) = \sum_{l=0}^N \Theta_l(\xi, \tau) \hat{q}_l \tag{4}
\]

and the space-time volume flux is

\[
F'_h(\xi, \tau) = \sum_{l=0}^N \Theta_l(\xi, \tau) F'_l. \tag{5}
\]

The latter amounts to the evaluation of the flux function at the Gauss-Legendre nodes, i.e. \( F'_l = F'(\hat{q}_l) \). Inserting the predictor solution \( q_h \) into Equation (3) yields

\[
\int_0^1 \int_{T_E} \Theta_h \frac{\partial q_h}{\partial \tau} d\xi d\tau + \int_0^1 \int_{T_E} \Theta_h \nabla \xi \cdot F'_h d\xi d\tau = 0.
\]

Integrating the first term by parts in time results in

\[
\left[ \int_{T_E} \Theta_h \frac{\partial q_h}{\partial \tau} d\xi \right]_0^1 - \int_0^1 \int_{T_E} \frac{\partial \Theta_h}{\partial \tau} q_h d\xi d\tau + \int_0^1 \int_{T_E} \Theta_h \nabla \xi \cdot F'_h d\xi d\tau = 0.
\]

The first term is now in the form to connect the predictor solution and the DG solution. At the beginning of each time step, the DG solution serves as the initial condition for the space-time predictor:

\[
\left[ \int_{T_E} \Theta_h \frac{\partial q_h}{\partial \tau} d\xi \right]_0^1 - \int_0^1 \int_{T_E} \frac{\partial \Theta_h}{\partial \tau} q_h d\xi d\tau + \int_0^1 \int_{T_E} \Theta_h \nabla \xi \cdot F'_h d\xi d\tau = 0.
\]

Inserting Equation (4) and Equation (5) leads to the space-time predictor solver kernel used in ExaHyPE:

\[
\sum_{l=0}^N \left( \int_{T_E} \Theta_h(\xi, 1) \Theta_l(\xi, 1) d\xi - \int_0^1 \int_{T_E} \frac{\partial \Theta_h}{\partial \tau} \Theta_l d\xi d\tau \right) \hat{q}_l = \begin{cases} 0 & \text{if } \kappa_1 \in \text{rhs0} \\
\sum_{l=0}^N \left( \int_{T_E} \Theta_h(\xi, 0) \Theta_l d\xi \right) \bar{q}_l \end{cases}
\]

(6)

The space-time predictor is the computationally most expensive component in the ADER-DG scheme. It is split up into three solver kernels.

- **Picard Loop.** Based on Equation (6), Picard iterations\(^1\) approximate the space-time predictor coefficients \( \hat{q}_l \). As \( K_1 \) is entirely based on the reference element, its inverse is precomputed.

\[
K_1 \hat{q}_l^{r+1} = \text{rhs0} - \nabla \xi K_1 F'(\bar{q}_l^{r+1}) \Rightarrow \hat{q}_l^{r+1} = K_1^{-1}(\text{rhs0} - \nabla \xi K_1 F'(\bar{q}_l^{r+1}))
\]

After \( r = N \) iterations, the space-time evolved predictor coefficients are known.

- **Time Averaging.** Time-averaging of the volume flux and the predictor over \([t^n, t^{n+1}]\) concludes the time integration.

\[
\bar{F} = \sum_l w_l F'_l \quad \bar{q} = \sum_l w_l \hat{q}_l \tag{7}
\]

- **Boundary Extrapolation.** As ExaHyPE uses nodal DG evaluated at the Gauss-Legendre nodes, the predictor and the flux values attained at the boundaries must still be computed. For this purpose, the values

\(^1\)In the linear case, the Picard loop is replaced by the Cauchy-Kowalewski time evolution.
are extrapolation to the left and to the right end of the reference element
\[ \hat{q}^- = \sum_i \Phi_i(0) \hat{q}_i, \]
\[ \hat{q}^+ = \sum_i \Phi_i(1) \hat{q}_i. \]

The flux is extrapolated analogously.

The predictor solution is local to an element and does not consider fluxes from neighbouring elements. The corrector stage augments the predictor solution by such contributions and thereby advances the DG solution to the next time step.

2.2 Corrector

The weak form of the PDE in Equation (1) is obtained by multiplying with a spatial test function \( \Phi \), integrating over the space-time control volume \( T^k \times [t^n, t^{n+1}] \), and reducing smoothness requirements on the flux function through integrating the flux divergence term
\[ \int_{t^n}^{t^{n+1}} \int_{T^k} \Phi \frac{\partial \mathbf{Q}}{\partial t} \, dx \, dt + \int_{t^n}^{t^{n+1}} \int_{\partial T^k} \Phi \mathbf{F}(\mathbf{Q}) \cdot \mathbf{n} \, dS \, dt \]
\[ - \int_{t^n}^{t^{n+1}} \int_{T^k} \nabla \Phi \mathbf{F}(\mathbf{Q}) \, dx \, dt = 0 \]

where \( \partial T^k \) denotes the boundary of the element \( T^k \) and \( \mathbf{n} \) its outward-pointing unit normal vector.

Inserting the predictor polynomial \( \mathbf{q}_k \) given in Equation (4) yields the discretised form. The boundary flux term amounts to a Riemann problem given the time-averaged boundary data \( \hat{q}^- \), \( \hat{q}^+ \), extrapolated from the left and the right element onto the face. As such, it is replaced with a numerical flux function in normal direction \( G(\hat{q}^-, \hat{q}^+ \cdot \mathbf{n}) \). Altogether, the ADER-DG scheme reads
\[ \sum_i \left( \int_{q^k} \Phi_i \Phi_i dx \right) \left( \hat{u}_i^{n+1} - \hat{u}_i^n \right) \]
\[ + \int_{t^n}^{t^{n+1}} \int_{\partial T^k} \Theta_i G(\hat{q}^-, \hat{q}^+) \cdot \mathbf{n} \, dS \, dt \]
\[ - \int_{t^n}^{t^{n+1}} \int_{T^k} \nabla \Phi_i \cdot \mathbf{F}(\mathbf{q}_k) \, dx \, dt = 0 \]

(8)

As the mass matrix \( \mathbf{M} \) is diagonal and assembles the tensor product of \( N \) Gauss-Legendre weights \( \mathbf{w} \), the explicit scheme becomes
\[ \mathbf{M} \Delta \hat{\mathbf{u}} = \mathbf{K} \hat{\mathbf{F}^*} - \Delta \tilde{\mathbf{f}} \]
\[ \Rightarrow \hat{\mathbf{u}}^{n+1} = \hat{\mathbf{u}}^n + \frac{\Delta t}{\mathbf{w}} (\mathbf{K} \hat{\mathbf{F}^*} - \Delta \tilde{\mathbf{f}}). \]

The corrector comprises two solver kernels.

- **Volume Kernel.** The volume integral computes the correction term \( \mathbf{K} \hat{\mathbf{F}^*} \) and is the computationally second-most expensive component.
- **Boundary Kernel.** The surface integral including the Riemann solver constitutes the boundary kernel, which computes the term \( \Delta \tilde{\mathbf{f}} \). For this purpose, ExaHyPE utilises a Rusanov solver.

Eventually the incorporation of the weighted correction term concludes one time step.

3. PROJECT STATUS

ExaHyPE’s code generator aims at replacing the generic solver kernels with vectorised, optimised version. For this purpose, the following questions have been tackled.

- What is the structure of the matrices and tensors?
- What is the favourable memory layout for each kernel? How can conflicting memory layouts be resolved?
- What is the access pattern to the tensors?
- What is the best way to multiply the square system matrix with a tensor slice?
- What parts of the code can be handled by an auto-vectorising compiler?

Due to usage of Cartesian grids, the solver kernels rely on dense, small-sized tensor operations. The most prominent operation is the matrix-tensor multiply. In this setup, the matrix resembles a square system matrix such as \( \mathbf{K} \). The tensor comprises degrees of freedom of the quantities in one element. In a 2D domain, for example, the tensor constitutes a three-way tensor holding the degrees of freedom for the quantities in \( x \)- and \( y \)-direction.

**Structure.** The system matrices are small, the tensors are tall and thin. In the nonlinear case all structures are dense. There is no sparsity pattern that could be taken into account for the computations. Due to the small size of the matrices, they can be assumed to reside in the cache.

**Memory Layout.** The tensors store the degrees of freedom of all quantities in each spatial direction, possibly also in time. For a 2D setup, a spatial tensor is a three-way tensors with the dimensions (i) the quantities \( Q \), (ii) the degrees of freedom in \( x \)-direction \( \text{DOFx} \), and (iii) the degrees of freedom in \( y \)-direction \( \text{DOFy} \). The dimensions of the tensor can be reordered freely, which yields a smattering of different ordering schemes. The solver kernels indeed favour certain ordering schemes. The space-time predictor exhibits weighted matrix-tensor multiplies. Since the weights apply to the quantities, the preferred ordering is \( (Q, \text{DOFx}, \text{DOFy}) \). For the boundary kernel, by contrast, the data should be ordered as \( (\text{DOFx}, Q_{Bnd}) \) and \( (\text{DOFy}, Q_{Bnd}) \). The volume integral
works on dimension-dependent flux tensors, which can be split and ordered independently. Figure 7 illustrates this. In summary, the individual solver kernels favour conflicting ordering schemes. A switch between ordering schemes typically requires that the tensor is transposed.

**Access Pattern.** Closely related to the different ordering schemes of the tensor dimensions is the access pattern to the tensors. For a 2D setup, the simplest occurrence of a weighted matrix-tensor multiply is

\[
C(:, :, j) = \alpha(j)B(:, :, j)A(:, :)
\]

followed with

\[
C(:, i, :) = \alpha(i)B(:, i, :)A(:, :)
\]

This operation is prominent in the space-time predictor. The scalar \(\alpha\) is a quadrature weight. The matrix \(A(:, :)\) is a fixed square system matrix. The tensor \(B(:, :, :)\) stores the degrees of freedom for all quantities. Here, its first dimension holds the quantities; its other two dimensions hold the spatial degrees of freedom in the \(x\)- and the \(y\)-direction, respectively. Due to the Cartesian grid set-up, the directions (here \(x\)- and \(y\)-direction) are processed one after another and demand different slices \(B(:, :, j)\) and \(B(:, i, :)\) of the tensor. As a result, the way the tensors are accessed depends on the spatial direction.

**Multiplication Order.** There are two orders in which matrices can be multiplied, namely \(AB\) vs \(B^T A^T\). For ExaHyPE, this affects the choice of the memory layout. Section 3.3 takes up on this topic.

### 3.1 Code Generator

Due to the access patterns to the tensors, the optimised kernels process the matrix-tensor multiplies in the form of a sequence of small-sized strided matrix-matrix multiplies. For this purpose, the code generator holds sequences of configurations. Consider for example a matrix-vector multiply. A configuration then specifies the dimensions of the matrix and the one of the vector. Based on these configurations, the code generator outputs two types of files.

- **Implementation files.** The code generator holds sequences of configurations. Each configuration yields an implementation file. Such an implementation file is typically given in the form of assembly code or close-to-assembly code. The latter can be intrinsics code or simple-structured C code that can be processed by an auto-vectorising compiler.

- **Driver files.** Driver files wrap the calls to the implementation files. The tensor set-up performs the very same operation on one data structure, but with different start addresses. A large part of code generator is therefore dedicated to the computation of the start addresses for the given configurations.

The combination of driver routines and implementation files will substitute the generic solver kernels. ExaHyPE’s code generator utilises Intel’s libxsmm as assembly code generation back end. Most of the implementation files are generated with Intel’s libxsmm. Given a sequence of slices of a tensor, the matrix-tensor multiply becomes a sequence of dense weighted matrix-matrix multiplications.

### 3.2 Performance of Selected Kernels

This section compares the generic kernels compiled with -O3 and the optimised version. All measurements have been executed on an Intel Xeon E5-2690v3 (“Haswell”) operating at 2.6 GHz. One core processes 2 vector instructions per cycle and exhibits 256-bit wide AVX instructions so that 4 double-precision operands can be processed simultaneously. The peak performance of one core calculates as 2.6 Gflops/s. The test case covers the 3D Euler equations of fluid dynamics with five variables for the polynomial degrees 3 up to 8.

**Picard Loop.** Based on the DG solution of the previous time step, the Picard loop computes the space-time evolved coefficients \(\hat{q}\) of the predictor polynomial. Figure 1 shows the runtime measurements for the 3D case. In the latest version of ExaHyPE, only the generic C++ kernels are in use; the generic Fortran kernels are only a legacy version. While the generic C++ kernels process matrix-tensor multiplies in an element-by-element fashion, the generic Fortran kernels use built-in intrinsics. For higher orders the optimised version yields a speedup of a factor of roughly 2 over the Fortran version. The optimised version implements the Picard loop as follows.

\[
\begin{align*}
\text{rhs} & = \sum_{l=0}^{N} \left( \int_{s^k} \Theta_k(\xi, 0) \Phi_l d\xi \right) \hat{u}_l^s \\
\text{for } r = 0, \ldots, N \text{ do } & \\
K_r^s & = \nabla_q K_{\xi} \\
\text{rhs} & = \text{rhs} - K_r^s F^*(\hat{q}_r) \\
\hat{q}_{r+1} & = \text{rhs} \left( K_r^s \right)^{-1}
\end{align*}
\]

Figure 1: Performance comparison of various implementations of the Picard loop with or without the evaluation of the 3D Euler flux function.
As Intel’s libxsmm does not support arbitrary scalars, the d-dimensional discrete scalar product over the reference element is incorporated into the auxiliary matrix $K_s^\xi$. The right-hand side is accumulated through a sequence of gemm calls. ExaHyPE stores all matrices and tensor slices in column major format. In contrast to Equation 9, $\mathbf{rhs}_0$ and $\mathbf{rhs}$ therefore are implicitly transposed so that $(K_s^{-1})^T$ is multiplied from the right. Internally, the optimised version holds both $K_s^{-1}$ and its transpose $(K_s^{-1})^T$ in memory to avoid any transpose operations.

The optimised version excels at order 7 because the dimension of the tensor slices exactly equals the SIMD width. Order 8, by contrast, falls off. For compatibility with the generic kernels, only some data structures have been padded; in particular the data structure holding the frequently used $\mathbf{rhs}$ remains unpadded.

The coupling of the optimised kernels and the user-defined flux function imposes an unfavourable memory layout for the flux tensor. ExaHyPE will provide fully optimised examples with an optimised data layout of the flux tensor and a vectorised evaluation of the flux function.

Anticipating Section 3.3, changing the multiplication order from $K_s^\xi F^\ast(q_i^r)$ to $F^\ast(q_i^r)K_s^\xi$ in conjunction with batched BLAS promises further speedup. Moreover, alignment of the data structures is yet to be done.

**Volume Kernel.** The volume kernel computes the contribution of the volume integral as given in Equation 8 to the update tensor $\Delta \hat{u}^n$. Similar to the runtime measurements of the Picard loop, the optimised version is about a factor of 2 faster than the Fortran version, see Figure 2. The optimised version implements the volume integral as follows.

$$K_s^\xi = \sum_{\mathbf{q}_i^r} K_s^\xi$$

$$\Delta \hat{u}^n = \hat{F}^\ast K_s^\xi$$

Following Paragraph 3, the spatial directions are processed successively. Due to the assumption of isotropic elements, the volume kernel works independent of the spatial direction being accessed. Following the workaround proposed by [5], the d-dimensional discrete scalar product over the reference element is incorporated into the auxiliary matrix $K_s^\xi$ and used repeatedly. The spatial contributions are therefore interleaved rather than processed one after another.

While all system matrices and the tensor slices $\mathbf{F}^\ast$ are padded, the update tensor $\Delta \hat{u}^n$ is not. The transition from order 7 to order 8 suggests that the introduction of padding for the update tensor gives further speedup. Moreover, all tensors are unaligned.

The manual version of the optimised kernels demonstrates the feasibility of the batched BLAS idea, see Section 3.3. This version is hand-written assembly code where 6 matrix-matrix multiplies are computed jointly. The hand-written code covers only the x-direction. The runtime reported is only an extrapolated value, assuming that processing each direction takes equally long. Possible memory effects are therefore not included.

**Time Averaging.** The time-integrated fluxes and predictor coefficients are obtained by averaging the space-time evolved quantities. The optimised kernels interpret Equation (7) as a sequence of matrix-matrix multiplies of a $Q$-by-$N$ with an $N$-by-1 matrix, which is handed over to Intel’s libxsmm. Figure 3 presents the runtime measurements.

### 3.3 Optimising Matrix Dimensions

Currently, a typical operation is a matrix-matrix multiply of a small-sized system matrix with a tensor slice. This section examines optimisation opportunities based on matrix sizes. Greater matrix sizes promise higher arithmetic intensity, which in turn promises better performance.

Intel’s libxsmm does not support arbitrary scalars and is not going to cover arbitrary scalars in the near future [4].
This performance test therefore compares only unweighted matrix-matrix multiplies
\[ C_{M \times N} = A_{M \times M} B_{K \times N}. \]
One matrix is a slice of a tensor; the other matrix is a square system matrix. As a consequence, without padding, the two options are
\[ C_{M \times N} = A_{M \times M} B_{K \times N} \quad \text{and} \quad C_{M \times N} = A_{M \times N} B_{N \times N}. \]

Figure 4 lists the results for an average of ten runs on an Intel Xeon E3-1235v3 (“Haswell”). From a performance perspective the conclusions are as follows.

1. For very small-sized structures, the performance is poor in any case.
2. The system matrices should be padded in order to avoid performance valleys.
3. The plateaus are at \( M = 28 \) and \( M = 32 \) even if \( N \) attains small values. The size of the square matrix takes a back seat to the structure of the tensor slice.

The size of the system matrix is fixed and prescribed by the user. Typical dimensions are 4 × 4 up to 10 × 10. The only opportunity for optimisation is padding, which has already been introduced. The second matrix, i.e. the tensor slice, behaves differently. One dimension of the tensor slice is fixed, of course. The other one, however, is not. This offers an opportunity for tuning. Looking at the figures, the key question is: How can the tensor slice be enlarged to hit a plateau? Section 3.4 takes up on this topic and discusses ideas on how this can be integrated into the code generator.

was passiert bei prefetching?

Table 1: Efficiency of the optimised kernels with respect to the theoretical peak performance of one core. The flop count for the Picard loop neglects the evaluation of the flux function and the computations of \( \text{rhs0} \). The runtime of the Picard loop covers \( \text{rhs0} \), but not the evaluation of the flux function. The manual version is hand-written assembly code which mimics batched BLAS. Only the \( x \)-direction with 6 jointly computed matrix-matrix multiplies has been implemented, the results reported here are extrapolated.

<table>
<thead>
<tr>
<th>Time Averaging</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>- flop count</td>
<td>8960</td>
<td>22500</td>
<td>47520</td>
<td>89180</td>
<td>153600</td>
<td>247860</td>
</tr>
<tr>
<td>- runtime [\mu s] generated</td>
<td>2.18</td>
<td>6.08</td>
<td>11.6</td>
<td>22.66</td>
<td>40.61</td>
<td>59.38</td>
</tr>
<tr>
<td>- efficiency [%] generated</td>
<td>19.76</td>
<td>17.78</td>
<td>18.89</td>
<td>18.94</td>
<td>18.17</td>
<td>20.05</td>
</tr>
</tbody>
</table>
| Figure 5: Storing frontal slices in column-major order.

3.4 Matricisation

From a performance perspective, it is better to compute one matrix-matrix multiply with one tall and thin tensor slice rather than a sequence of matrix-matrix multiplies with all small-sized matrices. Matricisation offers a means to jointly compute multiple matrix-matrix multiply, see Figure 6. The tensor slices, however, are scaled with the Gauss-Legendre quadrature weights. In fact it is the product of quadrature weights that impedes matricisation in a straightforward manner because of the dependence of the weights on the tensors slices.

There are three main ideas to nevertheless realise matricisation in ExaHyPE.
Figure 4: Performance of the dense matrix-matrix multiply in GFlops/s of \((M \times M)(M \times N)\) matrices (left) and \((M \times N)(N \times N)\) matrices (right).

\begin{verbatim}
! x - direction
DO j = 1 , nDOF(2)
  a = weight(j)/dx
  C(:,j) = C(:,j) +
          a * MATMUL(Kxi, F(:,1,:))
ENDDO

! y - direction
DO i = 1 , nDOF(1)
  a = weight(i)/dx
  C(:,i,:) = C(:,i,:) +
             a * MATMUL(Kxi, F(:,2,:))
ENDDO
\end{verbatim}

Figure 7: The generic code base (left) exhibits strided accesses to \(F\). Splitting \(F\) into \(F_x\), \(F_y\) and reordering \(F_y\) yields uniform access patterns enabling matricisation for both \(x\)- and \(y\)-direction (right).

1. **Splitting.** Data structures such as the flux tensors hold dimension-dependent data. The flux tensor can therefore be split into smaller tensors for each direction, which in turn reordered in an advantageous way. The code generator already implements this idea.

   Recall that the way the tensors are accessed depends on the space direction. Wherever applicable, the tensors are therefore rearranged or reshaped to match the frontal variant of matricisation, illustrated in Figure 6. This holds for the flux tensors present in the space-time predictor and the volume integral. Figure 7 gives the basic idea. Splitting up tensors into smaller tensors \(F_x, F_y\) enables direction-dependent reordering of the tensor variables. The drawback, however, is that already small structures become even smaller. In terms of performance, it would be advantageous to group multiple operation of this type.

2. **Batched BLAS.** Multiple operations of the same kind can be glued together. Instead of a sequence of calls with different start addresses, the tensor slices should be considered as one joint operation. One idea of consolidating the quadrature weights and the matricisation idea is mimicking batched BLAS [1]. Note that this idea takes place within a single element.

3. **Patches.** Reintroducing patches gives the opportunity to perform operations across multiple elements. The introduction of patches is not planned for the time being. Libxsmm does not offer an interface to incorporate arbitrary scalars into the matrix-matrix multiply. The workaround is the scaling of the system matrix before the gemm call. As multiple elements of the same spatial dimensions require the same scaled system matrix, these operations could be glued together. Diese Aussage sollte ohne Hack nicht umsetzbar sein. Peano gibt keine Garantien über die Speicheradressen.

4. **CONCLUSION AND OUTLOOK**

The first version of the code generator is simplistic and mostly mimics the matrix-matrix multiplies of the generic code base. The driver files enable the usage of libxsmm as an assembly code generation back end. First optimisation steps include the padding of system matrices, the padding of tensors, and dimension-dependent splitting and reordering of tensor data. A pending step is the reintegration of the generated files into the Peano framework.

In terms of performance the quadrature weights pose a challenge. The current workaround of first scaling the system matrix with a quadrature weight and then calling the libxsmm-generated gemm implementation cannot be consolidated with
the matricisation idea. The next steps will therefore address the integration of the quadrature weights.

5. REFERENCES


