Node-Level Optimization of a 3D Block-Based Multiresolution Compressible Flow Solver with Emphasis on Performance Portability

Nils Hoppe[1], Igor Pasichnyk[2], Momme Allalen[3], Stefan Adami[1], and Nikolaus A. Adams[1]

[1]Chair of Aerodynamics and Fluid Mechanics, Technical University of Munich, Garching, Germany
[2]IBM Deutschland, Research and Development, Garching, Germany
[3]Bavarian Academy of Sciences and Humanities, Leibniz Supercomputing Centre, Garching, Germany

December 1, 2020

Abstract

Despite the enormous increase in computational power in the last decades, the numerical study of complex flows remains challenging. State-of-the-art techniques to simulate hyperbolic flows with discontinuities rely on computationally demanding nonlinear schemes, such as Riemann solvers with weighted essentially non-oscillatory (WENO) stencils and characteristic decompositioning. To handle this complexity the numerical load can be reduced via a multiresolution (MR) algorithm with local time stepping (LTS) running on modern high-performance computing (HPC) systems. Eventually, the main challenge lies in an efficitent utilization of the available HPC hardware. In this work, we evaluate the performance improvement for a Message Passing Interface (MPI)-parallelized MR solver using single instruction multiple data (SIMD) optimizations. We present straight-forward code modifications that allow for auto-vectorization by the compiler, while maintaining the modularity of the code at comparable performance. We demonstrate performance improvements for representative Euler flow examples on both Intel Haswell and Intel Knights Landing Xeon Phi microarchitecture (KNL) clusters. The tests show single-core speedups of 1.7 (1.9) and average speedups of 1.4 (1.6) for the Haswell (KNL).

Keywords: Computational fluid dynamics; Multiresolution analysis; Parallel algorithms; Software engineering

1 Introduction

The study of flows exhibiting discontinuities such as shock waves or phase interfaces, is a field of intense study in fluid mechanic of diverse application fields.

Complex flow mechanisms are utilized e.g. in the generation of nanoparticles [1] [2] or biomedical procedures such as lithotrispy [3] [4]. In order to deepen the physical understanding of the underlying processes as well as for rapid testing and engineering of improved configurations, accurate and fast numerical simulations are desired. Therein, the combination of employed numerical schemes need to be adjusted to the specific case at hand, otherwise, low-quality or even unphysical solutions might be computed. In addition, the numerical schemes for the simulations of complex flow are computationally costly. Here, we consider conservative high-order finite volume method (FVM) schemes using approximate Riemann solvers [5], low-dissipation weighted essentially non-oscillatory (WENO) stencils [6] and total variation diminishing (TVD) Runge-Kutta (RK) time integration [7].

The high numerical load of the stated methods motivates the usage of compression algorithms. In the last two decades, wavelet-based compression algorithms have been applied to computational fluid dynamics (CFD) simulations. Three main types of wavelet-based algorithms can be identified: Galerkin, collocation or multiresolution (MR) approaches. For an overview of these algorithm we refer to the review paper by Schneider and Vasilyev [8].

In this work, we employ a fully adaptive block-based MR algorithm with local time stepping (LTS). The original MR algorithm developed by Harten [9] uses interpolating wavelets to determine regions in which the solution of the partial differential equation (PDE) is non-smooth. In these regions the resolution of the mesh is refined and vice versa, coarsened accordingly in smooth regions. In his original work, Harten, however, only used the coarse mesh representation to save costly flux evaluations. Instead, so-called fully adaptive MR methods also use the coarse mesh representation to reduce the memory footprint of the finite difference (FD)/FVM scheme. Such methods have been developed by Cohen et al. [10] for hyperbolic and Roussel et al. [11] for parabolic PDE. An additional compression of the temporal integration scheme offers further reduction of the computational load. Taking into account the different length scales at different levels, locally adapted timesteps can be used to advance the solution with an overall reduced number of timesteps [12], and, thus, number of flux evaluations.

Even with the stated compression algorithms, highly resolved three dimensional (3D) simulations require large computation performance of modern high-performance computing (HPC) systems. These compute systems owe their performance mostly to parallel hardware. Therefore, it is crucial to utilize this hardware efficiently. In standard MR algorithms every cell or scale, can be coarsened or refined. An implementation of such an algorithm, however, requires multiple data exchanges with neighboring cells per timestep, opposing efficient parallel execution [13]. To overcome this limitation we follow a block-based FVM variation of the MR algorithm by Han et al. [14]. Consequently, we can make use of the single instruction multiple data (SIMD) capacities of modern central processing unit (CPU).

Similar blocking methods have already proven useful in solving PDE on adaptive grids [15], [16] in the context of adaptive mesh refinement (AMR) algorithms. In particular, their SIMD capabilities were shown [17]. Also in wavelet collocation methods blocking has been introduced to harvest the performance of parallel hardware [18]. Besides the mentioned work of Han et al. Blocking algorithms have just been recently introduced for point-based MR [19].

In this work we show the benefit of SIMD parallelization for a Message Passing Interface (MPI)-parallel MR framework. We present weakly intrusive code optimizations which enable successful auto-vectorization by the compiler, yet the overall code structure with strong modularity and flexibility is unaffected. We demonstrate speed-ups up to a factor of two for different combinations of Riemann solvers, stencils, and cells per Block. Our performance study includes register sizes of 256 and 512 bit. The code framework ALPACA used in this work is published under open-source license and free-to-use for the public[1].

This paper is structured as follows: in the next section a short overview of the underlying physical and numerical models is given. In section 3, we identify the relevant performance bottlenecks and present potential improvements in section 4. The gain in performance is evaluated in section 5 using three representative Euler-flow test cases before we conclude in 6.

2 Numerical Models and Framework

2.1 Underlying Equations

Assuming inviscid flows, the compressible Euler equations govern the dynamics following the conservation law

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \mathbf{F}\left(\mathbf{U}\right) = 0, \tag{1}$$

where $\mathbf{U} = (\rho, \rho\mathbf{v}, E)$ is the state-vector with the density ρ, velocity vector $\mathbf{v}$ and the total energy $E = \rho e + \frac{1}{2}p\,|\mathbf{v}|$. The internal energy e together with ρ further defines the fluid pressure $p(\rho, e)$ via an equation of state (EOS). The flux vector $\mathbf{F}\left(\mathbf{U}\right)$ is given by the governing equation and determines the type-of-solution.

The Euler equations can be extended by source terms for volumetric forces such as gravity and by viscous terms to recover the Navier-Stokes equations. In our code framework, these terms can be switched on or off at compile time. Similarly, the employed EOS can be chosen at runtime. In this paper, however, we restrict ourselves to the stiffened EOS $p(\rho, e) = (\gamma - 1)\rho e - \gamma B$ [20], where γ is the ratio of specific head-capacities and the background pressure constant B is set to zero.

2.2 Riemann Solver with WENO stencils

The governing equations (1) are discretized using the FVM with cell-face Riemann problems defining the fluxes. Here, we focus on two different approximate Riemann solvers: The classical Roe solver [21] and the Harten-Lax-van Leer-contact (HLLC) solver [22]. Both solvers require left and right eigenvectors $K_i^{5\times 5}$ for every cell i to transform the system of equations into characteristic space. The Roe solver additionally requires the corresponding eigenvalues $\lambda_i^{5\times 1}$ and the scalar advection velocity in each cell.

In the Riemann solvers different reconstruction stencils can be used. To demonstrate the modularity of the framework, we use two different high-order reconstruction schemes for the flux calculations, namely the classical fifth order WENO stencil, implemented as in [23], and the sixth order central-upwind weighted essentially non-oscillatory (WENO-CU) stencil according to [24]. Note

[1] `https://nanoshock.de`

that, for the Riemann solver the WENO stencil is applied to the fluxes themselves and for the HLLC solver the WENO-reconstruction gives a high-order estimation of the cell-face states.

2.3 Block-Based MR

The fully adaptive cell- or scale-based MR algorithm [11] [10] allows to concentrate computational power only to relevant regions. We modify this algorithm to work on blocks rather than single scales [14]. Rather than refining or coarsening a single cell the decision is made for a block of cells. Each block consist of the same number of cells independently of its location within the domain or its level of refinement. We restrict ourselves to cubic blocks of cubic cells, i.e. the number of cells per direction is the same. If a block is refined eight (two per dimension) child blocks are created, which overlay the coarser parent block. Naturally this leads to an octree based implementation of the algorithm. We follow standard naming convention, where a *parent* is a block which is overlayed by finer *child* blocks, whereas a *leaf* is the finest block at that location. The PDE is only solved on the leaves. On the coarsest level l_0 the domain is partitioned into a fixed number of blocks according to user input.

As in the standard MR algorithm, a fifth order tensor based prediction is used. The maximum refinement level l_{max} and the MR thresholding ϵ are test case specific user input. The details are computed for the mass and energy only, as in the compressible case the momentum is directly linked to these quantities.

By using blocks a more cache and parallelization friendly data layout is achieved, compared to the single scale-based algorithm. Stencils evaluations in the block-based algorithm require simple array strides, where tree-based neighbor searches and/or data exchange are needed in a scale-based algorithm. However, the changed data layout comes at the cost of a reduced compression rate than in a scale-based MR implementation [18], [14], [19].

The need for neighbor data exchanges is further reduced by equipping every block with a ring of halo cells around its internal cells. The internal cells make up the computational domain without overlaps. The block-based mesh setup is illustrated in Figure 1. This data layout allows to evolve internal cells in time more independently.

The halo cells are filled with values copied from the neighboring block of the same refinement level or from predicted values if the same-level-neighbor does not exist. The halo cell update is shown in Figure 2, cell values are only exchanged within one level of refinement or via parent-child exchanges. The width of the halo layer depends on the prediction and the reconstruction stencil width. For the stencils used in this work four halo cells per spatial dimension are used.

For time integration we use a TVD RK2 LTS routine as given by Domingues et al. [12], which easily adapts to our block-based structure. Time is synchronized by adopting the dyadic cell refinement to adjust the timesteps between l_{i+1} and l_i. Therein, the Courant-Friedrichs-Lewy (CFL) condition is maintained at all scales if the smallest timestep is used as starting value.

Conservation between neighboring blocks of different levels is ensured by adjusting the coarse cell fluxes. This requires the additional storage of a the averaged fine-cell fluxes at the parent-block surfaces.

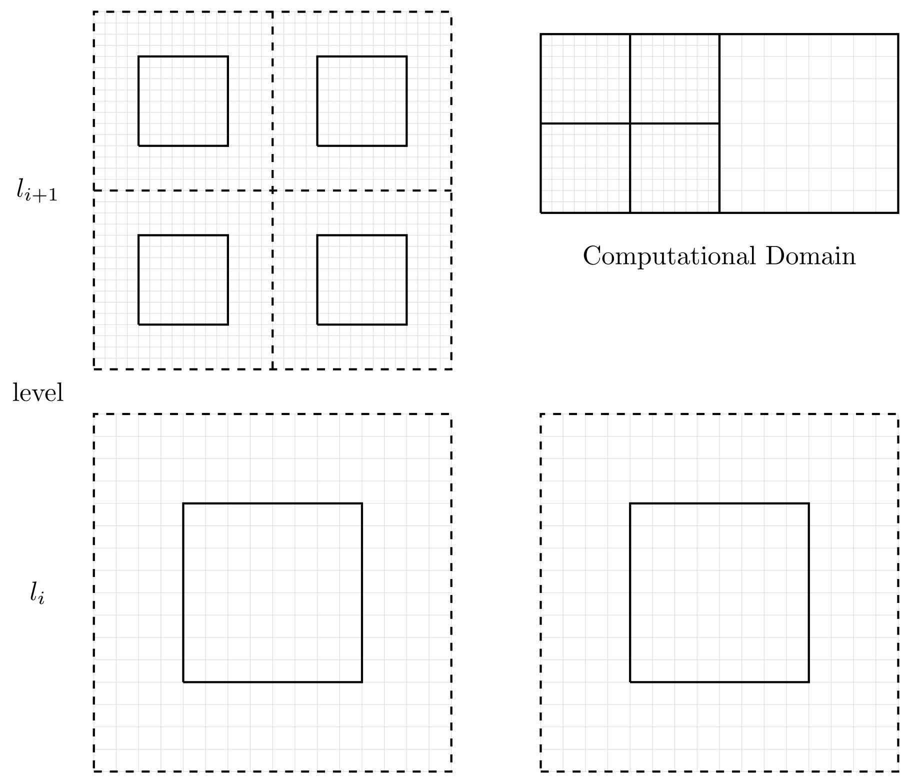


Figure 1: The block-based domain partition for a simple 2D mesh with one level of refinement. The computational domain is distributed into blocks with eight internal cells. In the top right the domain as seen by the user is shown. Only on these cells the PDE is solved numerically. In the left and bottom the representation of the mesh in memory with the halo cells is shown. The fine leaf child blocks overlaying the coarse parent in the left whereas the coarse part is represented by a single coarse leaf on the right.

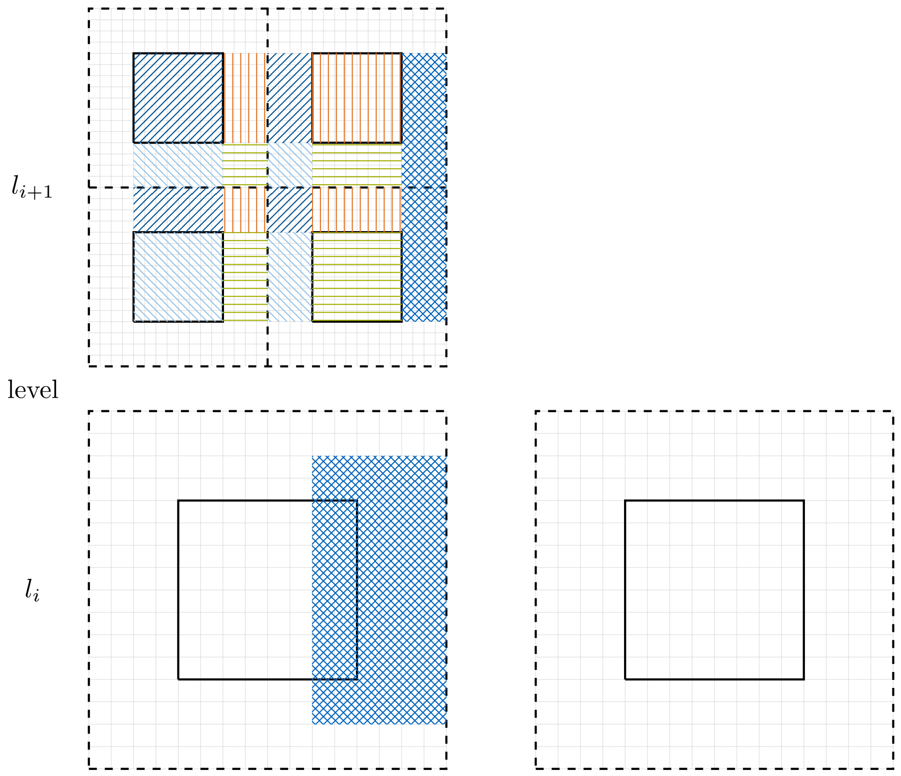


Figure 2: Schematic of the level l_{i+1} halo cell dependencies: The halo cells are filled either with copies of cell values from same level neighbors or via prediction of parent cells. Note, halo cells not colored are filled according to external boundary conditions.

2.4 Code Framework

The block-based MR scheme together with the solvers is implemented oin the open-source code framework ALPACA. The framework is written in modular fashion using object-oriented programming (OOP) with C++11. As mentioned above, at compile-time, we fix the number of cells in a block and the compute kernel, i.e. a combinations of Riemann solver and reconstruction stencil.

Aiming for a convenient use of the program we determine the EOS (and its parameters) as well as the MR thresholding coefficient from user-input at runtime; reducing recompilations for elaborate numerical studies of different fluids.

The MR algorithm in ALPACA is parallelized with MPI. Therein, blocks are the smallest inter-core parallel unit. Leaf-blocks are assigned to MPI ranks according to a 3D Hilbert space-filling curve [25] and parents are assigned to the rank holding the most of their children. Note, no further restrictions for domain partitions are imposed; i.e. we allow arbitrary cuts in the octree. This allows for straight forward load-balancing, as blocks can be distributed freely if the topology has changed after a mesh adaptation.

The overall algorithm follows the data-operator-splitting paradigm [14], i.e. we separate memory transfer functions from rank internal calculations. Thus, the performance bottlenecks of the code is its single-core performance per Block. For this purpose the data (access) layout is designed for optimized SIMD execution, independently of the selected compute kernel.

3 Identification of Bottlenecks

We evaluate the SIMD performance of ALPACA using Intel Advisor 2017[2]. We found that over 70% of the computational time is spent in the compute kernel. For the original implementation, it revealed that just 2% of the complete runtime was spent in vectorized loops. Surpisingly, even the flux computation as main compute kernel did not benefit from SIMD execution. A deeper analysis revealed two major obstracts for automated compiler generated vectorization of the flux computation: the loop structure within the Riemann solvers and the current exception handling.

Firstly, we look at the loop structure in the Riemann solvers. Listing 1 shows the implementation of the main loop nest in the Roe solver class in C++-like pseudo-code. The HLLC solver follows an analogous setup, but without the need for computing the physical flux or the eigenvalues. The vector sizes marked by capital letter `N` indicate compile-time quantities. The second letter (`T` for "total" or `I` for "internal") indicates whether halos cells are included in this array or not, respectively. Here, fives resemble the five conserved quantities in the 3D Euler equations (1). The physical flux in every cell is computed prior to the main loop nest. With each loop iteration, three directional cell-face (numerical) fluxes are determined. Therefore, the eigenvalue vector and eigenvector matrices are computed three times on-the-fly per loop iteration. Note the unified `ComputeFluxes`, which is used for all three directions by simple permutations of the passed loop indices thereby reducing code duplication.

[2]https://software.intel.com/en-us/advisor

Listing 1: Baseline loop nest of the compute kernel.

```
// Inputs from further up the call-stack
Block const& block;
(&double)[5][N_I+1][N_I+1][N_I+1] fluxes_x, fluxes_y, fluxes_z;

//Temporarily allocated
double[5][N_T][N_T][N_T] phys_flux_x, phys_flux_y, phys_flux_z;
double[5][5] eigenvectors_left, eigenvectors_right;
double[5] eigenvalues;

// Determine physical flux - Roe only
(phys_flux_x, phys_flux_y, phys_flux_z) = ComputePhysicalFlux(
    block);

for k, j, i {
   // X DIRECTION
   (eigenvectors_left, eigenvectors_right, eigenvalues) =
       ComputeEigenvaluesX( block, i, j, k );
   fluxes_x[i][j][k] = ComputeFluxes( block, i, j, k, 0,
       phys_flux_x, eigenvectors_left, eigenvectors_right,
       eigenvalues );

   // Y DIRECTION
   (eigenvectors_left, eigenvectors_right, eigenvalues) =
       ComputeEigenvaluesY( block, j, i, k );
   fluxes_y[j][i][k] = ComputeFluxes( block, j, i, k, 1,
       phys_flux_y, eigenvectors_left, eigenvectors_right,
       eigenvalues );

   // Z DIRECTION
   (eigenvectors_left, eigenvectors_right, eigenvalues) =
       ComputeEigenvaluesZ( block, j, k, i );
   fluxes_z[j][k][i] = ComputeFluxes( block, j, k, i, 2,
       phys_flux_z, eigenvectors_left, eigenvectors_right,
       eigenvalues );
}
```

The second issue conflicting automated vectorization deals with exception handling. For the compiler-generated vectorization report it is clear, that functions are auto-vectorized only if exceptions are rigorously prohibited. In ALPACA, the problematic routine stems from a proxy class for the EOS. With it the respective EOS of each cell can be determined at run time. The look-up of the correct EOS used `std::find` routines which do not guarantee `noexcept`.

4 Code Improvements

To reduce the runtime without sacrificing the code quality, i.e. its usability, readability, extendability and portability between different architectures, modifications are limited to standard C++ instructions as well as Open Multi-

Processing (openMP) (SIMD) pragmas instead of hard-coded architecture dependent (low-level) optimizations.

Our performance measurements have identified the Riemann solvers as bottleneck. As a consequence, we have re-designed its structure and slit the computation of eigenvalues, eigenvectors and fluxes. Each direction is now handled separately and the eigendecomposition is temporarily buffered. The resulting structure is shown in Listing 2.

Listing 2: Directionally split compute kernel loop nest.

```
// Other quantities and arrays unchanged.
double[N_I][N_I][N_I][5][5] eigenvectors_left, eigenvectors_right
    ;
double[N_I][5] eigenvalues;

(eigenvectors_left, eigenvectors_right, eigenvalues) =
    ComputeEigenvaluesX( block );
phys_flux_x = ComputePhysicalFluxX( block );
fluxes_x = ComputeFluxes( block, 0, phys_flux_x,
    eigenvectors_left, eigenvectors_right, eigenvalues, cell_size
     );

(eigenvectors_left, eigenvectors_right, eigenvalues) =
    ComputeEigenvaluesY( block );
phys_flux_y = ComputePhysicalFluxY( block );
fluxes_y = ComputeFluxes( block, 1, phys_flux_y,
    eigenvectors_left, eigenvectors_right, eigenvalues, cell_size
     );

// Z analogously.
```

The new structure addresses two issues that inhibit vectorization of the baseline implementation: Loop-nest depth and strided data access. The baseline `ComputeFluxes` function used nested loops, in which only the deepest loop was vectorizable. Grouping it with the computation of the eigenvectors thus required the latter to be executed sequentially. By splitting the loop nest into separate steps both functions can benefit from vectorization. The used WENO stencils inside the `ComputeFluxes` make memory bandwidth problems unlikely, as such stencils require hundreds of floating point operations per five (continuous) data accesses. Furthermore, the compact implementation using index permutation to handle different coordinate direction caused decelerating strided data access. In the split version, a continuous data layout is ensured for each direction.

Finally, analyzing the new loop structure showed a complete auto-vectorization of the `ComputePhysicalFlux` function. Inside the `ComputeEigenvalues` function single elements of one of the block's buffers were obtained in each loop iteration via a `GetBuffer` function call. This constant re-fetching was not auto-vectorizable. Replacing it by a single fetch approach as shown in Listing 3 yields the desired vectorization. Additionally, an exception-safe variant of the EOS proxy, mentioned in section 3, was introduced. With these changes, also the `ComputeEigenvalues` functions were vectorized by the compiler according to the vectorization report.

Listing 3: Fetching of buffers before and after optimization.

```
// Before:
for i, j k
   double const value = coefficient * block.GetBuffer()[i][j][k];

// Now:
double (&cells)[N_T][N_T][N_T] = block.GetBuffer();
for i, j, k
   double const value = coefficient * cells[i][j][k];
```

Another problem for auto-vectorization are unanticipated memory access patterns. Interestingly, we found smart logics as e.g. a running offset in a unified array access implementation are not recognized by the compiler and excluded from auto-optimization. As a remedy, we "simplified" the code and achieved full auto-vectorization also in these parts. Listing 4 shows the original implementation as well as the improved version below.

Listing 4: Vectorization of loop nests with offset indices

```
// Input: direction = 0 (for X), 1 (Y) or 2 (Z)

// Before
int flux_direction[3] = {0};
flux_direction[direction] = 1;
double (&cells)[N_T][N_T][N_T] = block.GetBuffer();

for i,j,k,m
   double const value = cells[i+flux_direction[0]*m][j+
       flux_direction[1]*m][k+flux_direction[2]*m];

// Now
const int x_off = direction==0 ? 1 : 0;
const int y_off = direction==1 ? 1 : 0;
const int z_off = direction==2 ? 1 : 0;
double (&cells)[N_T][N_T][N_T] = block.GetBuffer();

for i,j,k,m
  double const value = cells[i+x_off*m][j+y_off*m][k+z_off*m]
```

5 Optimizations results

5.1 Test case definition

We use three representative test cases to evaluate the impact of the found optimization, the well-know sod shock tube [26], a one-phase viscous Rayleigh-Taylor instability (RTI) [27] and a Gaussian pulse case. Note, all setups are simulated in full 3D domains even though the problem definition might be of lower dimensions. For the Sod test cases, a gas ($\gamma = 1.4$) with discontinuous left $(\rho = 1.0, \mathbf{v} = 0, p = 1.0)_{y \leq 0.5}$ and right $(\rho = 0.25, \mathbf{v} = 0, p = 0.1)_{y > 0.5}$ state is simulated until time $t = 0.2$ with CFL $= 0.6$. At this point the shock wave, con-

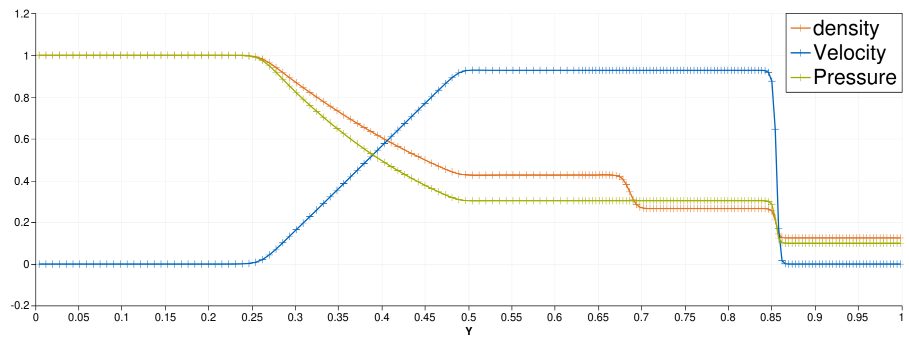


Figure 3: Final state $t = 0.2$ of the Sod shock tube problem using the HLLC Riemann solver with WENO5 reconstruction stencil and 16 internal cell per block. The state values along a line parallel to the Y-axis of the computational domain are plotted. The markers indicate the cell center positions and illustrate the mesh refinement.

tact discontinuity and the rarefaction fan have separated and are clearly distinguishable as shown in Figure 3. The simulated domain is $[0, 0.25] \times [0, 1] \times [0, 0.25]$ and symmetry boundary conditions are applied on all sides. The maximum level of refinement is fixed to $l_{max} = 2$.

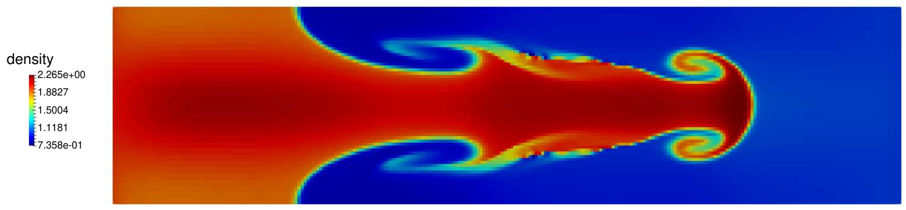
In the RTI test case a heavy fluid layer is accelerated into a lighter one by gravity. By adding a small disturbance to the layered initial condition, a mushroom shaped instability forms and growths continuously, see Figure 4 for a visualization of the heavy (red) and light (blue) fluid at $t = 1.95$. The domain is $[0, 0.25] \times [0, 1] \times [0, 0.25]$ and the initial conditions are

$$\begin{aligned}
\rho &= \begin{cases} 1.0 & y \leq 0.5 \\ 2.0 & y > 0.5 \end{cases} \\
v_x &= 0.0, \\
v_y &= -0.25 * c * \cos(8\pi x) * \cos(8\pi z), \\
v_z &= 0.0, \\
p &= 1.0 + 2.0 * y
\end{aligned} \tag{2}$$

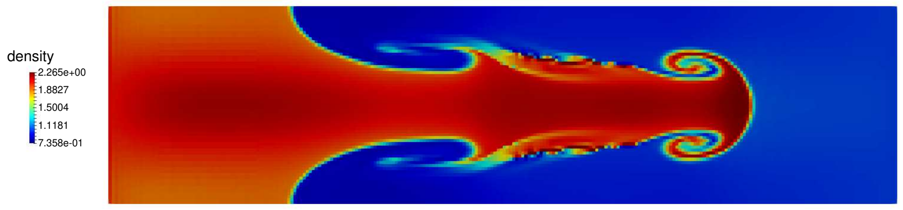
with the speed of sound $c = \sqrt{\gamma p / \rho}$ and $\gamma = 5/3$.

Symmetry boundary conditions are applied, except for top and bottom where the states $(1, 0, 0, 0, 2.5)$ and $(2, 0, 0, 0, 1)$ are imposed, respectively. The simulation is evolved until the final time $t = 1.95$ is reached, the CFL number is fixed to 0.9. The maximum level of refinement is $l_{max} = 3$.

In the Gaussian pulse test case a cubic domain with side length 12 and all outflow boundary conditions is simulated. The initial density and pressure profile follow a Gaussian distribution:



(a) Roe Riemann solver, WENO5 and 16 internal cells.



(b) Roe Riemann solver, WENO-CU6 and 16 internal cells.

Figure 4: Top view of the viscous RTI computed with different reconstruction stencils. Plotted is the density at the final timestep $t = 1.95$.

$$\begin{aligned}
\rho &= 1.0 + 0.5 * e^{-0.693\frac{(x-6.0)^2+(y-6.0)^2+(z-6.0)^2}{2}}, \\
\mathbf{v} &= 0, \qquad\qquad\qquad\qquad\qquad\qquad (3) \\
p &= \frac{1}{\gamma} + 0.5 * e^{-0.693\frac{(x-6.0)^2+(y-6.0)^2+(z-6.0)^2}{2}},
\end{aligned}$$

with $\gamma = 1.4$. The final simulation time is $t = 2.6$ with a CFL number of 0.6 and the maximum refinement level $l_{max} = 6$. The resulting pulses in the energy and velocity magnitude field are shown in Figure 5 together with the refined mesh.

As stated earlier, all test cases were run using all combinations of the two Riemann solvers, the two reconstruction stencils and with 8^3 and 16^3 internal cells per block. The effective resolution, i.e. the comparable resolution on a homogeneous mesh without MR for the cases is summarized in Table 1, together with the effective compression rate.

We verified that the code changes did not affect the accuracy of the numerical solution. For most of the conducted tests the results are identical, only a few cases showed differences on the order of floating point roundoff error.

5.2 Compute architectures

We executed our tests on two homogeneous clusters with different architectures. One is based on Intel Xeon E5-2697 v3 “Haswell (HW)” CPUs with 28 cores at 2.7 GHz and 2.3GB memory and the other cluster consists of Intel Xeon Phi 7210-F hosts “Knights Landing Xeon Phi microarchitecture (KNL)” nodes with 64 CPUs per node at 1.3 GHz with the multi-channel dynamic random-access memory (MCDRAM) in cache mode. During parallel runs each MPI rank was

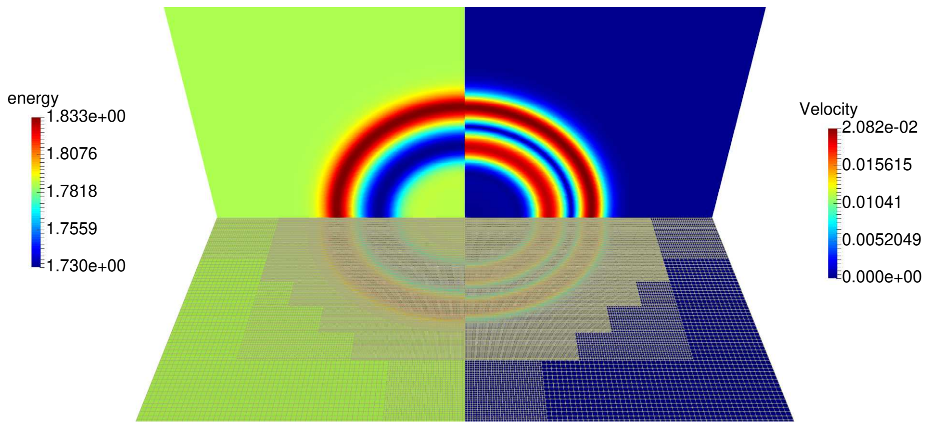


Figure 5: Energy and velocity magnitude of Gaussian pulse at the end of the simulation at $t = 2.5$

Table 1: Specification of the test cases. Compression compares a finest-level homogeneous mesh with the cell average over the whole simulation time over all configurations.

Test case	$\frac{\#\text{cells}^3}{\text{Block}}$	Resolution	Compression [%]
Sod	8	$32 \times 128 \times 32$	58.19
	16	$64 \times 256 \times 64$	43.94
RTI	8	$64 \times 256 \times 64$	52.45
	16	$128 \times 512 \times 128$	43.87
Gauss	8	$512 \times 512 \times 512$	9.29
	16	$1024 \times 1024 \times 1024$	3.29

assigned to one physical core; hyper-threading was not employed. We have compiled the source code with the Intel C++ compiler (ICC) version 16.0 with an underlying installation of the GNU C++ compiler (GCC) version 5.4.0. All binaries were compiled with compiler flags `-ipo`, `-m64`, and `-fp-model precise`. For the optimized version, we also added `-xCORE-AVX2` and `-xMIC-AVX512` flags for the HW and KNL, respectively. Thereby, the full instruction set was used, i.e. `ymm` and `zmm` registers for the HW and the KNL, respectively. This was verified by inspecting the produced assembler code. However, these compiler flags should be used with caution since they affect the truncation error behavior of the numerical scheme [28]. Therefore, we analyzed the speedups with and without the usage of Advanced Vector Extensions (AVX) registers.

For each test case, we used a different number of MPI ranks. For the Sod case only a single core and hence a single MPI rank was used. The RTI case used 28 MPI ranks, which means all cores on one compute node were completely used on the HW. For the runs on the KNL system we used both 28 and 64 ranks to have a direct comparison of rank count and node count. Similarly, we use $(5 \times 28 =)$ 140 ranks for the Gaussian pulse test case, but also executed it with 140 and 320 ranks on five KNL nodes.

5.3 Performance estimation

We ran our test cases with different kernel and block configurations, on both HW and KNL hardware, mentioned earlier. All runtime results were measured with `MPI_Wtime` and were averaged over three (execution) samples. On the HW the standard deviation of any configuration did not exceed 3%. For the KNL, the more ranks were used the higher deviations were measured. The average deviation on the KNL does not exceed 11%, with a few outliers with a maximum deviation of 24% for the RTI testcase. Note, initialization and disk input and output were explicitly excluded from the time measurements.

In the following, we compare the achieved speedups for a range of configurations on both architectures. Therein, speedup is defined as the relative runtime as compared to the same configuration (solver, stencil, cells per block, architecture) using the baseline implementation. The average baseline wall-clock runtime of the three cases on the HW (KNL) was 176s (1634s), 1528s (12831s) and 1310s (3861s) for the Sod, the RTI and the Gaussian Pulse, respectively.

Figures 6 and 7 give an overview of the speedups across configurations and test cases for the HW and the KNL cluster, respectively. For all cases we used the same number of MPI ranks per node. Obviously, the single core runs showed the largest improvement, but for all settings a significant speedup was achieved. Clearly, the speedup for the KNL is larger for all test cases and configurations than on the HW.

Analyzing the effects of the block configuration and the `AVX` compiler flag in detail, we point to Table 2. It depicts the achieved speedups for all test cases averaged over all kernel configurations run on both architectures with two different numbers of MPI ranks on the KNL as described before. We find, that both aspects have a stronger impact on the KNL than on the HW. So does the `AVX` compiler flag show almost no difference on the HW but a factor of up to 0.38 can be observed on the KNL. The increase of the number of cells in a block results in a performance increase for all setups, except for the single core HW case. Again the performance gain is stronger on the KNL than on the HW.

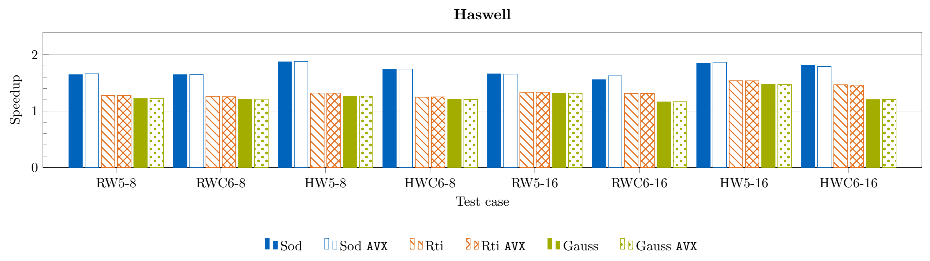


Figure 6: Overview of runtime speedups for all three test cases, four different kernels and two block configurations on the Haswell architecture. In the abbreviations, *R* and *H* denote Roe and the HLLC Riemann solver, respectively. *W5* and *WC6* indicate the WENO5 and the WENO-CU6 stencil and *8* and *16* denote the number of blocks per dimension.

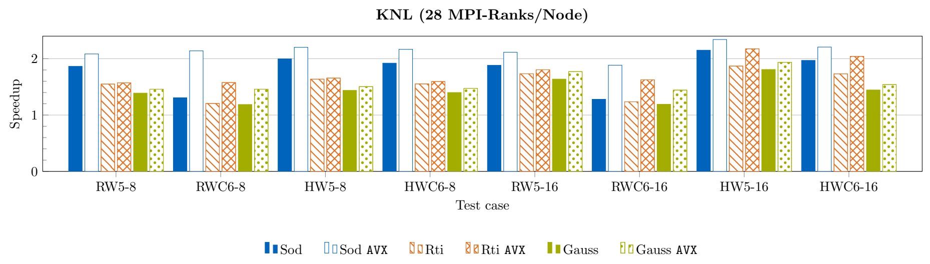


Figure 7: Overview of runtime speedups on the KNL arcitecture running 28 MPI-ranks per KNL node. For abbreviations see Figure 6.

Table 2: Average speedup over all kernel configurations with and without the `AVX` compiler flag.

Cluster	$\frac{\#\text{cells}^3}{\text{Block}}$	Sod `AVX` off	Sod `AVX` on	RTI `AVX` off	RTI `AVX` on	Gauss `AVX` of	Gauss `AVX` on
Haswell	8	1.73	1.73	1.28	1.27	1.23	1.23
	16	1.72	1.73	1.41	1.41	1.29	1.29
KNL28	8	1.77	2.15	1.49	1.60	1.35	1.47
	16	1.82	2.14	1.64	1.91	1.52	1.67
KNL	8	-	-	1.09	1.19	1.16	1.23
	16	-	-	1.37	1.52	1.33	1.43

We refer to Table 3, for a detailed view on the speedups obtained with different kernel configurations. The shown speedups are averaged over all test cases for each kernel and cells-per-block configuration. The impact of `AVX` is also stated separately. As seen before, configurations with 16^3 cell per block show stronger gains than the once with 8^3. The configurations using a WENO5 reconstruction stencil benefit stronger than the WENO-CU6. Furthermore, the HLLC Riemann solver based configurations benefit stronger from the optimizations than the Roe based ones. In the baseline implementation the Roe configurations had a shorter time-to-solution than the HLLC ones (not shown), whereas after the optimizations the tested HLLC configurations show shorter runtimes.

In all configurations the single core Sod test case showed the strongest performance gains. This is explainable, by the lack of MPI-overhead. Similarly, the computation on full KNL nodes did not benefit as strongly from the code changes as the ones on the HW. We account this to MPI overhead, as the numerical load of one rank becomes too small if a whole KNL node is filled purely with MPI. For example, the RTI can only yield 32 blocks per rank, if the mesh is globally refined to the maximum. As seen in Table 1, however, only half of this resolution is used on average. With such low loads the communication and in it particular the sequential neighbor searches become bottlenecks and hinder performance. We credit the higher speedups for 16^3 internal cells per block on the KNL to the wider vector registers, which benefit from longer loop bodies.

6 Summary and Outlook

We have introduced an MPI parallelization of a block-based FVM MR scheme for the simulation of complex flows governed by the compressible Euler or Navier-Stokes equations. Thereby, our implementation allows to utilize modern distributed memory machines efficiently. In particular, the SIMD capabilities of these systems can be harvested. We focus on compiler generated SIMD vectorization, such that all of the modular compute kernels could benefit from vector-

Table 3: Average speedup over all test cases of the kernel and block configurations. For abbreviations see Figure 6.

Configuration	Haswell AVX off	Haswell AVX on	KNL-28 AVX off	KNL-28 AVX on	KNL AVX off	KNL AVX on
RW5-8	1.38	1.39	1.60	1.70	1.15	1.19
RWC6-8	1.37	1.37	1.23	1.72	1.00	1.22
HW5-8	1.49	1.49	1.69	1.79	1.18	1.23
HWC6-8	1.40	1.40	1.62	1.74	1.16	1.22
RW5-16	1.44	1.44	1.75	1.90	1.41	1.50
RWC6-16	1.34	1.37	1.24	1.65	1.07	1.31
HW5-16	1.62	1.62	1.94	2.15	1.58	1.67
HWC6-16	1.49	1.49	1.71	1.93	1.34	1.43

ization. We showed, how to overcome common pitfalls for compiler-generated vectorization and presented the respective performance increases. Over a broad range of configuration and test cases considerable speedups could be achieved. We want to highlight that the proposed (minor) code changes did not alter the general algorithm or the communication strategy and are not custom tailored to a specific compute kernel. The performance gain was validated on two different micro-architectures with 256-bit and 512-bit wide vector registers.

Currently, we further optimize the computational efficiency following a hybrid openMP-parallelization on local compute units. Also, the optimized code is extended for sharp-interface two-phase flows. In which, even more numerical schemes need to be added in a modular fashion.

Acknowledgment

The first, fourht and fifth authors have received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 667483).

The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. for funding this project by providing computing time on the CoolMUC-2 Linux-Cluster at Leibniz Supercomputing Centre.

The authors acknowledge the Bavarian State Ministry of Science and the Arts for partial funding through the Competence Network for Scientific High Performance Computing in Bavaria (KONWIHR).

References

[1] Gabriele C Messina, Philipp Wagener, René Streubel, Alessandro De Giacomo, Antonio Santagata, Giuseppe Compagnini, and Stephan Bar-

cikowski. Pulsed laser ablation of a continuously-fed wire in liquid flow for high-yield production of silver nanoparticles. *Phys. Chem. Chem. Phys.*, 15(9):3093–3098, 2013.

[2] Tatiana E. Itina. On nanoparticle formation by laser ablation in liquids. *Journal of Physical Chemistry C*, 115(12):5044–5048, mar 2011.

[3] Christian Chaussy and Egbert Schmiedt. Urologic Radiology Extracorporeal Shock Wave Lithotripsy (ESWL) for Kidney Stones. An Alternative to Surgery? *Urol Radiol*, 6:80–87, 1984.

[4] Kazuki Maeda, Wayne Kreider, Adam Maxwell, Bryan Cunitz, Tim Colonius, and Michael Bailey. Modeling and experimental analysis of acoustic cavitation bubbles for Burst Wave Lithotripsy. *Journal of Physics: Conference Series*, 656:012027, dec 2015.

[5] E. F. Toro. *Riemann solvers and numerical methods for fluid dynamics: a practical introduction.* Springer, Dordrecht ; New York, 3rd ed edition, 2009.

[6] X.-D. Xu Dong Liu, Stanley Osher, and Tony Chan. Weighted Essentially Non-oscillatory Schemes. *Journal of Computational Physics*, 115:200–212, 1994.

[7] Ami Harten. High resolution schemes for hyperbolic conservation laws. *Journal of Computational Physics*, 49(3):357–393, mar 1983.

[8] Kai Schneider and Oleg V. Vasilyev. Wavelet Methods in Computational Fluid Dynamics. *Annual Review of Fluid Mechanics*, 42(1):473–503, jan 2010.

[9] Ami Harten. Multiresolution algorithms for the numerical solution of hyperbolic conservation laws. *Communications on Pure and Applied Mathematics*, 48(12):1305–1342, sep 1995.

[10] Albert Cohen, Sidi Kaber, Siegfried Müller, and Marie Postel. Fully adaptive multiresolution finite volume schemes for conservation laws. *Mathematics of Computation*, 72(241):183–225, 2003.

[11] Olivier Roussel, Kai Schneider, Alexei Tsigulin, and Henning Bockhorn. A conservative fully adaptive multiresolution algorithm for parabolic PDEs. *Journal of Computational Physics*, 188(2):493–523, jul 2003.

[12] Margarete O. Domingues, Sônia M. Gomes, Olivier Roussel, and Kai Schneider. An adaptive multiresolution scheme with local time stepping for evolutionary PDEs. *Journal of Computational Physics*, 227(8):3758–3780, apr 2008.

[13] K. Brix, S. Melian, S. Müller, and M. Bachmann. Adaptive Multiresolution Methods: Practical issues on Data Structures, Implementation and Parallelization. *ESAIM: Proceedings*, 34:151–183, dec 2011.

[14] L. H. Han, T. Indinger, X. Y. Hu, and N. A. Adams. Wavelet-based adaptive multi-resolution solver on heterogeneous parallel architecture for computational fluid dynamics. In *Computer Science - Research and Development*, volume 26, pages 197–203, jun 2011.

[15] Wolfgang Eckhardt and Tobias Weinzierl. A blocking strategy on multicore architectures for dynamically adaptive PDE solvers. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6067 LNCS, pages 567–575. Springer, Berlin, Heidelberg, 2010.

[16] Martin Schreiber, Tobias Weinzierl, and Hans Joachim Bungartz. Cluster optimization and parallelization of simulations with dynamically adaptive grids. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8097 LNCS, pages 484–496. Springer, Berlin, Heidelberg, 2013.

[17] Chaulio R. Ferreira and Michael Bader. Load Balancing and Patch-Based Parallel Adaptive Mesh Refinement for Tsunami Simulation on Heterogeneous Platforms Using Xeon Phi Coprocessors. In *Proceedings of the Platform for Advanced Scientific Computing Conference on - PASC '17*, pages 1–12, New York, New York, USA, 2017. ACM Press.

[18] Babak Hejazialhosseini, Diego Rossinelli, Michael Bergdorf, and Petros Koumoutsakos. High order finite volume methods on wavelet-adapted grids with local time-stepping on multicore architectures for the simulation of shock-bubble interactions. *Journal of Computational Physics*, 229(22):8364–8383, nov 2010.

[19] Mario Sroka, Thomas Engels, Philipp Krah, Sophie Mutzel, Kai Schneider, and Julius Reiss. An Open and Parallel Multiresolution Framework Using Block-Based Adaptive Grids. pages 305–319. Springer, Cham, 2019.

[20] F. Harlow and A. Amsden. Fluid dynamics. Technical report, Los Alamos National Labs, 1971.

[21] Philip L. Roe. Approximate Riemann solvers, parameter vectors, and difference schemes. *Journal of computational physics*, 43(2):357–372, 1981.

[22] Eleuterio F. Toro, M. Spruce, and W. Speares. Restoration of the contact surface in the HLL-Riemann solver. *Shock waves*, 4(1):25–34, 1994.

[23] Lin Fu, Xiangyu Y. Hu, and Nikolaus A. Adams. A family of high-order targeted ENO schemes for compressible-fluid simulations. *Journal of Computational Physics*, 305:333–359, jan 2016.

[24] X.Y. Hu, Q. Wang, and N.A. Adams. An adaptive central-upwind weighted essentially non-oscillatory scheme. *Journal of Computational Physics*, 229(23):8952–8965, nov 2010.

[25] Michael Bader. *Space-Filling Curves*, volume 9 of *Texts in Computational Science and Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[26] Gary A Sod. A survey of several finite difference methods for systems of nonlinear hyperbolic conservation laws, apr 1978.

[27] Jing Shi, Yong Tao Zhang, and Chi Wang Shu. Resolution of high order WENO schemes for complicated flow structures. *Journal of Computational Physics*, 186(2):690–696, 2003.

[28] Nico Fleischmann, Stefan Adami, and Nikolaus A. Adams. Numerical Symmetry-Preserving Techniques for Low-Dissipation Shock-Capturing Schemes. *Computers & Fluids*, may 2019.