



Master Thesis

Investigating Matrix Assembly in High Performance CFD Codes

Lukas Petermann
lukas.petermann@ovgu.de

October 12, 2026

First Reviewer:
Prof. Dr. Michael Kuhn

Second Reviewer:
Prof. Dr. Hartwig Anzt

Supervisor:
Dr. Gregor Olenik

Abstract

The finite volume method (FVM) is one of the most popular method in the field of computational fluid dynamics. In terms of FVM-based fluid flow simulations, OpenFOAM is one of the most prominent software projects of this decade. The discretisation, also referred to as *matrix assembly*, of partial differential equations like the momentum equation is a central aspect to the FVM. Due to eager evaluation of the differential operators and the lack of official GPU support in OpenFOAM, this matrix assembly shows poor performance. As a representative of approaches that introduce GPU acceleration by refactoring the OpenFOAM code base, NeoN solves some of these problems, but is still limited by static compilation and template-laden development of modern C++.

This thesis presents JuNe, an interface that embeds Julia into NeoN and utilises Julia’s just-in-time compilation to fuse operators at runtime. The performance aspects of both the matrix assembly and the operator fusion are evaluated on the lid-driven cavity and DrivAer cases using four different matrix assembly strategies, namely cell-based, face-based, global face-based, and batched face-based. These evaluations were performed in a serial, multi-threaded, and GPU context on contemporary hardware like the NVIDIA H200 GPU or the AMD EPYC CPU. In addition to the quantitative experiments, the impact of the interface on the development effort is discussed. The results show that the overhead of the proposed operator fusion strategy is negligible. The fused cell-based strategy of NeoN outperforms JuNe in a serial and multi-threaded context. In the context of GPU matrix assembly, the face-based strategy of JuNe shows the highest performance. Overall, the interface is considered to have a positive impact on the development efforts.

Contents

1	Introduction	1
1.1	Goals	4
1.2	Structure	5
2	Background	7
2.1	The Finite Volume Method	7
2.2	Governing Equations	7
2.2.1	Continuity Equation	8
2.2.2	Momentum Equation	8
2.3	Discretisation of the Momentum Equation	8
2.3.1	The Transient Operator	9
2.3.2	The Divergence Operator	11
2.3.3	The Laplacian Operator	11
2.4	The Roofline Model & Arithmetic Intensity	12
3	Related Work	15
3.1	OpenFOAM Fused	15
3.2	DSL Optimisation in NeoN	15
3.3	XCALibre.jl	16
4	Methods	19
4.1	Operators	20
4.1.1	Div	20
4.1.2	Laplacian	21
4.1.3	The Transient Operator	21
4.1.4	Operator Fusing	22
4.2	Operator Schemes	23
4.2.1	Div Schemes: Gauss Linear, Upwind	23
4.2.2	Laplacian Schemes: Uncorrected	23
4.2.3	Transient Schemes: Backward Euler	24
4.2.4	Abstraction of Interpolation Schemes	25
4.3	Matrix Assembly Strategies	25
4.3.1	Cell-Based	26
4.3.2	Face-Based	28

4.3.3	Global Face-Based	29
4.3.4	Batched Face-Based	30
4.4	Arithmetic Intensities	31
4.4.1	Cell-Based, Internal Faces	31
4.4.2	Face-Based, Internal Faces	32
4.4.3	Face-Based, Boundary Faces	32
4.5	JuNe: Integrating Julia into NeoN	33
4.6	Summary	34
5	Implementation	37
5.1	OpenFOAM Cases	37
5.2	Case Generation	38
5.3	Interpolation Schemes	39
5.3.1	Div Schemes	39
5.3.2	Temporal Integration Schemes	40
5.4	Operators	41
5.4.1	Div	42
5.4.2	Laplacian	45
5.4.3	The Transient Operator	47
5.4.4	Operator Fusing	48
5.5	Standalone Julia	51
5.5.1	Preprocessing	51
5.5.2	Runtime Scheme Abstractions	53
5.6	JuNe	56
5.6.1	Integration into NeoN	57
5.6.2	Data Preparation	58
5.6.3	Passing Data between NeoN and JuNe	60
5.6.4	Data Reception in Julia	61
5.6.5	Matrix Assembly Strategies	63
5.7	Developing Julia for Performance	69
5.8	Summary	72
6	Evaluation	75
6.1	General Setup	75
6.1.1	Metrics	75
6.1.2	Test Cases	77
6.1.3	IcoFoam	79
6.1.4	Evaluations Structure	80
6.2	Standalone Julia	80
6.2.1	Variations of Operator Calculations	80
6.2.2	Serial Matrix Assembly Strategies	82
6.2.3	Multi-Threaded Matrix Assembly Strategies	83
6.2.4	GPU Matrix Assembly Strategies	86
6.2.5	Summary	88
6.3	Evaluation of JuNe	88
6.3.1	CPU Matrix Assembly Strategies	89

6.3.2	GPU Matrix Assembly Strategies	92
6.3.3	Integration Benchmark	94
6.3.4	Summary	96
6.4	Roofline Model	97
6.5	Compilation Times	98
6.6	Summary	99
7	Discussion	101
7.1	Development Effort	101
7.2	How Does the Interface Affect Development?	103
7.3	Summary	105
8	Conclusion	107
8.1	Research Questions Findings	107
8.1.1	RQ I: Operator Calculation & Fusing	107
8.1.2	RQ II: Matrix Assembly Strategies	108
8.1.3	RQ III: Integration into CFD Frameworks	109
8.2	Limitations	110
8.3	Future Work	110
	Bibliography	113
	A. Memory Footprints	120
	B. Multi-Threaded Results	123
	C. Usage Of Gen. AI	125

1 Introduction

The simulation of physical phenomena has been an important engineering design tool for decades. Here, computational fluid dynamics (CFD) is one prominent example. Therein, numerical analyses are performed to investigate the behaviour of various types of flows. In the industry, a popular choice is the finite volume method (FVM) due to its simplicity, inherent robustness, and ability to process unstructured polygonal geometries [Moukalled et al., 2015]. Central to FVM is the transformation of partial differential equations (PDEs) into sets of linear equations in the form of matrices, which will henceforth be referred to as *matrix assembly*. One of the most prominent FVM-based open-source software projects is OpenFOAM [Jasak et al., 2007]. It has been a staple in the CFD community for a long time, with its first public release dating back to 2004, though it has been in closed development for far longer.

With the recent trends in GPU computing and increase in computational demand, the lack of official GPU support, despite numerous efforts, has become apparent. As of now, the only means of parallelisation integrated in OpenFOAM is distributed processing using the message passing interface (MPI). However, it is useful to invest development effort towards GPU support for OpenFOAM. The entire simulation process involves repeatedly performing a relatively small workload for millions of elements during assembly and subsequently solving the resulting linear system. GPUs excel at both of these tasks.

Two approaches of adding GPU support to OpenFOAM are currently acknowledged by the community: First, OpenFOAM can read plugins at runtime, which can include custom-written solvers that offload computations to the GPU. However, the matrix assembly is still performed on the OpenFOAM side, which limits the attainable speedup of this approach. Notable representatives of this category are the OpenFOAM-Ginkgo-Layer [Olenik et al., 2024] and petsc4FOAM [Bna et al., 2020]. Second, GPU support is introduced by refactoring parts of the OpenFOAM codebase itself. One example for refactoring parts of OpenFOAM is NeoN [Olenik et al., 2025]. Designed for portability and modularity, NeoN provides functionality for all hardware and software vendors while employing state-of-the-art simulation software systems like Kokkos [Trott et al., 2022], Ginkgo [Anzt et al., 2022], and PETSc [Balay et al., 1997]. Moreover, NeoN provides a domain-specific language (DSL) that allows efficient and developer-friendly encoding of complex partial differential equations. OpenFOAM also provides a DSL for defining PDEs, allowing the abstraction of low-level details such as MPI communication patterns or looping strategies during assembly.

In the field of high performance computing (HPC), the predominantly used languages are C/C++. Naturally, this applies not only to NeoN and its included DSL, but also OpenFOAM. In OpenFOAM, IcoFoam is the typically used solver application for incompressible Navier-Stokes equations using the Pressure-Implicit with Splitting of Operators (PISO) [Issa et al., 1986] algorithm. Listing 1.1 shows how the momentum equation is defined by the DSLs of NeoN and IcoFoam, respectively.

```

1 dsl::Expression<scalar> momentum(
2   dsl::imp::ddt(U)
3   + dsl::imp::div(phi, U)
4   - dsl::imp::laplacian(gamma, U)
5 );

```

(a) NeoN

```

1 fvVectorMatrix momentum(
2   fvm::ddt(U)
3   + fvm::div(phi, U)
4   - fvm::laplacian(gamma, U)
5 );

```

(b) IcoFoam

Listing 1.1.: Equivalent implementations of the momentum equation in NeoN and IcoFoam.

Though the definitions of the momentum equation shown in Listing 1.1 seem identical, the implementation of IcoFoam has a substantial caveat: Each operator of the momentum is evaluated eagerly, i.e., each operator iterates over the entire mesh and returns its own coefficient matrix. This is illustrated in Figure 1.2a. For the momentum equation, this results in three iterations and three intermediate matrices, which are subsequently combined into the final linear system. The operators perform comparably few operations in relation to the data required to fully process the mesh. Consequentially, the matrix assembly kernel implemented in OpenFOAM is memory bound. Further considering that the implementation of each operator discretisation iterates over the same mesh elements, two matrix (de-)allocations of the total three in the momentum equation above are unnecessary. Consequently, calculating each operator in a separate iteration over the mesh reduces the arithmetic intensity even further.

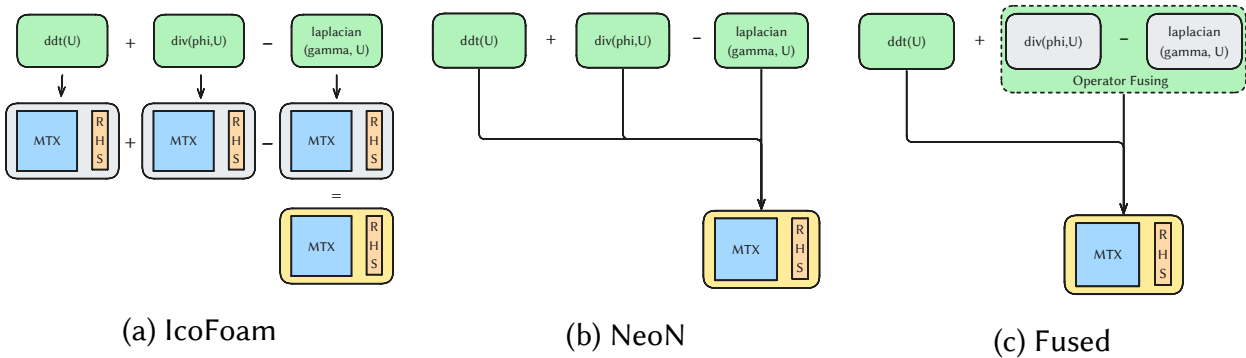


Figure 1.2.: DSL evaluation approaches. The PDE operators are displayed in the top row. The middle row denotes temporary linear systems created by individual operators. Drawn on the bottom are the finally assembled linear systems of the momentum equation.

Contrasting this, NeoN implements lazily evaluated operators in its DSL implementation, which avoids the usage of unnecessary intermediate matrices. This is shown in Figure 1.2b: The operators, instead of creating temporary matrices, write into the same linear system. Considering an improvement to the number of passes over the mesh, one of the current NeoN development efforts is to extend the DSL with an optimisation step that joins multiple operators into a single one. The current implementation still requires substantial development effort, as virtually all combinations of operators would have to be implemented both into the DSL and as a joint operation. Furthermore, the operator calculations depend on a set of case-specific configuration settings which are not available during compilation. For instance, the scheme selected for the divergence operator $div(phi, U)$, is only known at runtime. According to the OpenFOAM user

guide¹, this divergence scheme can be chosen out of a list of 14 options. To accommodate this variability, modern CFD cores like NeoN are often implemented using template meta-programming, e.g., a template is defined for each valid combination of interpolation scheme and operator. However, because there are more instances of this runtime variability besides the interpolation scheme, the number of templates to implement grows substantially. Even only providing all combinations for the divergence and time integration schemes of the transient operator $\text{ddt}(U)$ would require around 60 template definitions. Developing a vast amount of templates not only increases mental load for the developer and the likelihood of bugs, it also diminishes overall maintainability of the code base. Lastly, achieving this variability through lots of additional, but rather similar, code also leads to increased compilation times.

To briefly summarise the current state: Compared to OpenFOAM's matrix assembly, the lazy implementation of NeoN significantly reduces the memory footprint. An experimental operator fusing strategy is being integrated into NeoN, which reduces the passes over the mesh at the cost of caveats resulting from implementing a large amount of templates to account runtime variabilities. Consequentially, the compilation times grow, and the development speed is diminished. Overall, this approach is not scalable, which sufficiently motivates taking a different approach into consideration.

In an optimal scenario, the matrix assembly is (1) implemented lazily to reduce the memory footprint, (2) able to fuse operators to limit the number of passes over the mesh to a minimum, while (3) being able to realise runtime-only simulation parameters without significantly impeding the development effort.

Assuming it is possible to implement these three items in C++, it would entail major refactors of the NeoN codebase. A less invasive approach would be to leverage language interoperability between C++ and a more dynamic language. A common use case for this is calling a compiled C++ binary from Python for a performance-critical task, but the direction is the reverse of the use case presented in this work. A more appropriate example for this use case is the molecular simulation software LAMMPS [Thompson et al., 2022]. Therein, Python scripts can be integrated into the LAMMPS input script which are then dynamically executed. However embedding Python into LAMMPS is not motivated by performance implications, but to overcome the limitations of the input script syntax.

Embedding Julia instead of Python into C++ is a more performance-oriented alternative. Julia has already been proven to be a valid choice in the context of HPC, performing Euler-Lagrange particle tracking at exascale and cataloguing the visible universe using Bayesian inference at petascale [Regier et al., 2019; Schmalfuß et al., 2026]. Furthermore, Julia is also a promising solution considering the three requirements of an optimal scenario stated above: First, C++ and Julia are both implemented on the LLVM compiler backend. This simplifies the transfer of data across language boundaries, enabling Julia to use the same shared linear system used in NeoN and, therefore, the lazy evaluation as well. Second, and of most interest for this work, Julia is a just-in-time (JIT) compiled language, which lays the foundation for an operator fusing approach that does not require explicit implementation of fused operators. Instead, the individual operators need to be implemented only once and can be fused at runtime with negligible overhead.

¹<https://www.openfoam.com/documentation/guides/v2012/doc/guide-schemes-divergence.html>

Lastly, any runtime variability is handled by the JIT compiler, removing the need for a large quantity of templates.

1.1 Goals

While scientific software written primarily in Julia already exists [Medina et al., 2025; Schmalfuß et al., 2026], embedding Julia into C++ is an entirely novel approach in the context of computational fluid dynamics. The primary goal of this work is the improvement of the compute performance of modern CFD cores in combination with a flexible and developer-friendly approach to implement typical compute kernels. This is achieved by presenting JuNe: interoperability between Julia and NeoN for efficient matrix assembly with operator fusing. To evaluate the effectiveness of JuNe, the following research questions (RQs) are evaluated in this thesis:

RQ I: Operator Calculation & Fusing:

- RQ I.a *What are the limitations and performance implications of the operator fusing strategy?*
- RQ I.b *What is the cost of different runtime or just-in-time abstraction approaches, e.g., for interpolation schemes?*

The first research question is focused on the PDE operator evaluations. As mentioned above, an operator fusing approach is developed to reduce passes over the mesh without explicitly creating an implementation for every possible combination of fusable operators. RQ I.a specifically deals with possible caveats resulting from fusing the operators automatically from an implementation perspective. Additionally, the operator fusing needs to be quantitatively evaluated as well. The evaluation of the operator fusing is connected to the evaluation of different ways to evaluate schemes for the operators.

RQ II: Matrix Assembly Strategies:

- RQ II.a *What are the performance implications of different matrix assembly strategies?*
- RQ II.b *What is the near-optimal performance of typical CFD kernels?*

Because various matrix assembly strategies exist, each with its own advantages and caveats, one objective of this work is to quantitatively compare them with respect to scalability and flexibility of operator fusing. To obtain a broad overview, this evaluation includes the matrix assembly of standalone Julia, standalone NeoN, and JuNe.

RQ III: Integration into CFD Frameworks

- RQ III.a *What is a good strategy to interface the matrix assembly with existing CFD frameworks?*
- RQ III.b *What is the performance overhead of the interface between C++ and Julia?*
- RQ III.c *How does the interface affect the development effort?*

The third research question is arguably the most important one, as it deals with the primary contribution of this work: extracting the matrix assembly of NeoN into the Julia language. In this work, the implemented interface and extracted Julia matrix assembly are referred to as JuNe. In principle, while the shared LLVM backend is promising, the practical implementation of such an interface needs evaluation as it has never been done before in this context. Here, the primary evaluation is whether or not the matrix assembly extraction is useful in context of the whole system. This evaluation naturally includes the aspect of efficiency, but also involves

considerations about the development process. The answer to RQ III.c will be discussed in the form of a qualitative reflection of the author in Chapter 7.

The goal of this work is not to rewrite an entire CFD simulation in Julia, but to extract the matrix assembly which can be plugged into an existing system. To keep the scope narrow, this work is focused on incompressible fluid flow problems.

1.2 Structure

The remainder of this thesis is structured as follows. Chapter 2 provides explanations of FVM fundamentals regarding the discretisation of the relevant PDEs into a set of linear equations. Chapter 3 outlines related work and other approaches to operator fusing. Thereafter, the research concept is outlined in Chapter 4 and followed by the respective implementations in Chapter 5. The proposed evaluation procedure and the results thereof are presented in Chapter 6. The implementation process, especially with respect to RQ III.c, is discussed in Chapter 7, before closing this work in Chapter 8 by evaluating the state of the research questions based on our findings in the preceding chapters and giving some remarks on future research endeavours.

2 Background

This chapter aims to deliver a necessary knowledge basis for the contents of this thesis. First, the finite volume method and the governing equations of this work are introduced. Thereafter, the mathematical steps included in the discretisation of differential equations are presented. Additionally, the momentum equation, as well as the included operators and a selection of schemes are presented. The chapter closes with performance considerations in the field of HPC.

2.1 The Finite Volume Method

Solving partial differential equations (PDEs) such as Navier-Stokes are generally considered to be rather difficult². Therefore, multiple approximation strategies have been developed over the years. While methods like finite difference [Zhou, 1993], finite elements [Morgan and Peraire, 1998] or Lattice Boltzmann [Chen and Doolen, 1998] are popular, this work focuses on the finite volume method (FVM).

The finite volume method relies on subdividing the problem domain into smaller subdomains called control volumes. These control volumes are referred to as *cells*. Each cell has a volume V_C , and is enclosed by a set of faces F_C . In the context of FVM, a flux is the rate with which a quantity general quantity ϕ crosses a face. Each face f has an area A_f and together with its normal vector, the outward-pointing surface vector S_f is defined as:

$$S_f = n_f A_f \quad (1)$$

The direction of S_f indicates whether a flux enters or leaves the cell.

All faces are either considered internal or boundary face. An internal face is shared by two neighbouring cells. Adapting standard OpenFOAM terminology, one is called the owner O and the other the neighbour N . A boundary face is located at the border of a geometry and therefore has no neighbouring cell N .

A number of variations of FVM exist, e.g., vertex-centric, cell-centric and face-centric [Burcet et al., 2024]. These variations indicate where the discrete values of ϕ are stored. In this thesis, only the primarily used cell-centric variant is considered. Consequently, the field value is stored at the centre of the cells.

2.2 Governing Equations

Fluid flow is governed by the conservation laws of physics [Versteeg and Malalasekera, 2007]. The generic conservation equation for a transported quantity is given in Eq. (2):

²<https://www.claymath.org/millennium/navier-stokes-equation/>

$$\frac{\partial(\rho\phi)}{\partial t} + \nabla \cdot (\rho \mathbf{U} \phi) = \nabla \cdot (\Gamma \phi \nabla \phi) + S_\phi \quad (2)$$

By changing the values of ϕ and Γ , the conservation of mass and the conservation of momentum can be obtained.

2.2.1 Continuity Equation

Setting $\phi = 1$ in Eq. (2) yields the continuity equation in Eq. (3).

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{U}) = 0 \quad (3)$$

As the focus of this thesis is incompressible FVM, the density ρ becomes constant and Eq. (3) simplifies to Eq. (4). The continuity equation states that the velocity field $\mathbf{U}$ is divergence-free, i.e., the net mass flux over the boundary of every cell must be zero.

$$\nabla \cdot \mathbf{U} = 0 \quad (4)$$

2.2.2 Momentum Equation

The second conservation law, the conservation of momentum, is obtained by setting $\phi = \mathbf{U}$ and $\Gamma = \nu$:

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot (\mathbf{U} \cdot \mathbf{U}) = -\nabla p + \nabla \cdot (\nu \nabla^2 \mathbf{U}) + S_U \quad (5)$$

The transient term $\frac{\partial \mathbf{U}}{\partial t}$ describes the change of velocity over time. The convection term $\nabla \cdot (\mathbf{U} \cdot \mathbf{U})$ transports momentum with the mass flow, while the diffusive term $\nabla \cdot (\nu \nabla^2 \mathbf{U})$ represents viscous transport. S_U contains source terms. It is not possible to solve Eq. (5) independently because the pressure is typically unknown. Additionally, an arbitrary solution to Eq. (5) might not satisfy the divergence-free constraint of Eq. (4). These sets of equations can be rewritten in matrix form, which is shown in Eq. (6) [Moukalled et al., 2015, Chapter 15]. Because $\mathbf{A}$ includes a zero block, which indicates a saddle point problem, the system cannot be computed iteratively.

$$\mathbf{A} \mathbf{U} = \begin{pmatrix} \mathbf{F} & \mathbf{B}^T \\ \mathbf{B} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \mathbf{v} \\ \mathbf{p} \end{pmatrix} = \begin{pmatrix} \mathbf{f} \\ \mathbf{0} \end{pmatrix} \quad (6)$$

Instead, a solution to the pressure term is needed to solve for the velocity, which in turn is needed to solve for the pressure. The idea is, therefore, to split the operators in Eq. (5) into two terms: The pressure term and the momentum equation. Because this thesis primarily focuses on the fusing of PDE operators and matrix assembly, representatively on the momentum equation, the pressure equation is not further detailed here.

2.3 Discretisation of the Momentum Equation

All of the terms included in the momentum equation are discretised separately. Later, their respective contributions are combined into an equation for each cell. In conjunction, all these

equations create coefficients that are finally assembled into a matrix. In the following, the discretisation of each operator is detailed.

Evaluating the general transport equation in Eq. (2) by integrating over each control volume V yields Eq. (7).

$$\int_V \frac{\partial(\rho\phi)}{\partial t} dV + \underbrace{\int_V \nabla \cdot (\rho \mathbf{U} \phi) dV}_{\text{Spatial Operators}} = \int_V \nabla \cdot (\Gamma \phi \nabla \phi) dV + S_\phi \quad (7)$$

At this point, the equation is spatially limited to the cell's extent, but still continuous. Gauss' Divergence Theorem [Katz, 1979] is then applied to the spatial operators. The theorem states, that the flux of the surface area A_f is equal to the total flow out of the control volume V , thereby moving the evaluation from the cell volume to its faces. This gives Eq. (8)

$$\frac{\partial}{\partial t} \int_V \rho \phi dV + \sum_{f \in F_C} (\rho \mathbf{U} \phi)_f \cdot A_f = \sum_{f \in F_C} \Gamma (\nabla \phi)_f \cdot A_f + \int_V S_\phi V \quad (8)$$

Here, approximations are required for the time integral, the face values, the face gradients, and the source integral. The following sections are dedicated to the operators, their discretisation and the schemes used in this work.

2.3.1 The Transient Operator

In steady state flows the transient term can be omitted. However, in simulations with a rate of change over time, the transient term is needed. Other than the spatial operators, the transient term is not discretised using neighbouring control volumes, but instead derived from the local value at one or multiple time steps.

Regardless of the scheme used, the transient equation is integrated over a time interval $[t - \Delta t/2, t + \Delta t/2]$. Further, by treating the cell volume as a constant, the integral turn into a difference of face fluxes [Moukalled et al., 2015, Section 13.3]. The semi-discretised equation transient equation is shown in Eq. (9).

$$\int_{t-\frac{\Delta t}{2}}^{t+\frac{\Delta t}{2}} \frac{\partial(\rho\phi)}{\partial t} V_C dt = V_C (\rho_C \phi_C)^{t+\frac{\Delta t}{2}} - V_C (\rho_C \phi_C)^{t-\frac{\Delta t}{2}} = 0 \quad (9)$$

To obtain the fully discretised transient equation, an interpolation profile for the values at $t + \Delta t/2$ and $t - \Delta t/2$ is needed. Regardless of the scheme chosen, the fully discretised transient operator can be described through a summation of fluxes [Moukalled et al., 2015, Chapter 13]:

$$\text{Flux} = \text{FluxC} \phi_C + \text{FluxC}^{t-1} \phi_C^{t-1} + \text{FluxV} \quad (10)$$

And, algebraically:

$$\begin{aligned} a_C &= a_C + \text{FluxC} \\ b_C &= b_C - \text{FluxC}^{t-1} \phi_C^{t-1} - \text{FluxV} \end{aligned} \quad (11)$$

2.3 Discretisation of the Momentum Equation

Therein Flux_C represents the flux linearisation coefficient of the cell C . Flux_V denotes the non-linearised parts and depends on the chosen transient scheme. In the matrix assembled during discretisation, a_C is stored on the diagonal entry corresponding to the cell C . b_C is stored insight the right hand side vector. In this work, two schemes are considered: The first and second order of the backward differentiation formula, which is also referred to as *Backward Euler*. Both schemes are derived using a Taylor series expansion of $\frac{\partial(\rho\phi_C)}{\partial t}$.

2.3.1.1 Backward Euler First-Order

Taking the first Taylor expansion derivative of Eq. (8) yields Eq. (12):

$$\frac{(\rho_C\phi_C)^t - (\rho_C\phi_C)^{t-\Delta t}}{\Delta t}V_C + \mathbb{S} = 0, \quad (12)$$

with $\mathbb{S}$ being the contributions of the spatial terms. The algebraic form of the transient equation using the first-order backward Euler (BDF1) scheme is given in Eq. (13).

$$(a_C^t + a_C)\phi_C + \sum_{F \in \text{Nei}(C)} a_F\phi_F = b_C + a_C^{t-1}\phi^{t-1}, \quad (13)$$

whereas the coefficients are defined as:

$$a_C^t = \frac{\rho_C V_C}{\Delta t}, \quad a_C^{t-1} = -\frac{\rho_C^{t-1} V_C}{\Delta t} \quad (14)$$

The BDF1 scheme is numerically stable and bounded by the values of its spatial neighbours at the current time step. However, a high accuracy can only be achieved with very small values of Δt .

2.3.1.2 Backward Euler Second-Order

For the second-order backward Euler (BDF2), the second Taylor expansion derivative of Eq. (8) is taken, resulting in Eq. (15).

$$\frac{3(\rho_C\phi_C)^t - 4(\rho_C\phi_C)^{t-\Delta t} + (\rho_C\phi_C)^{t-2\Delta t}}{2\Delta t}V_C + \mathbb{S} = 0, \quad (15)$$

with $\mathbb{S}$ being the contributions of the spatial terms. The algebraic form of the transient equation using BDF2 scheme is given in Eq. (16).

$$(a_C^t + a_C)\phi_C + \sum_{F \in \text{Nei}(C)} a_F\phi_F = b_C - a_C^{t-1}\phi^{t-1} - a_C^{t-2}\phi^{t-2}, \quad (16)$$

whereas the coefficients are defined as:

$$a_C^t = \frac{3\rho_C V_C}{2\Delta t}, \quad a_C^{t-1} = -\frac{4\rho_C^{t-1} V_C}{2\Delta t}, \quad a_C^{t-2} = \frac{\rho_C^{t-2} V_C}{2\Delta t} \quad (17)$$

The BDF2 scheme is stable but not guaranteed to be bounded [Moukalled et al., 2015, Section 13.2.6].

2.3.2 The Divergence Operator

As introduced at the end of Section 2.3, the face values need approximation because the convective term, the div operator, is evaluated as a sum of face fluxes. For only one velocity component u , i.e., one dimensional velocity, the operator is once again given by Eq. (18)

$$\int_{V_C} \nabla \cdot (\rho \mathbf{U} u) \approx \sum_{f \in F_C} \dot{m} u_f, \quad \text{with } \dot{m} = (\rho \mathbf{U}) \cdot \mathbf{S}_f \quad (18)$$

Here, the velocity at the face u_f needs to be determined through a selected interpolation scheme called divScheme. For this work, two interpolation schemes are considered: Gauss linear and the upwind scheme.

2.3.2.1 Gauss Linear

The Gauss linear scheme, also known as central differencing scheme, linearly interpolates between two values stored at neighbouring cell centres. The face value is calculated using Eq. (19).

$$u_f = \frac{1}{2}u_C + \frac{1}{2}u_N \quad (19)$$

This scheme is generally second-order accurate in space. However, it is not always bounded, i.e., it does not always yield physical results. This case can occur then the convection overpowers the diffusion [Moukalled et al., 2015, Section 11.2]

2.3.2.2 Upwind

The main problem of the Gauss linear scheme is that it disregards flux direction. In contrast, using the upwind scheme, the face value depends on the upwind value, i.e., the value at the cell from which the mass flow originates. The velocity value u_f at the face f between two cells C, N is therefore calculated as shown in Eq. (20).

$$u_f = \begin{cases} u_C & \text{if } \dot{m}_f > 0 \\ u_N & \text{otherwise} \end{cases} \quad (20)$$

2.3.3 The Laplacian Operator

The diffusive term is discretised as a Laplacian.

2.3.3.1 Uncorrected

For orthogonal faces, the diffusive flux can be approximated by Eq. (21):

$$(\Gamma \nabla u)_f \cdot \mathbf{S}_f \approx d_f (u_C - u_N), \quad \text{with } d_f = \Gamma \frac{A_f}{\delta_{CN}} \quad (21)$$

In Eq. (21), δ_{CN} is the distance between the cell centres of C and N measured in the face-normal direction. This approximation works only for orthogonal, or very slightly non-orthogonal faces.

2.3 Discretisation of the Momentum Equation

Because meshes are usually non-orthogonal in the industry, it is necessary to include an orthogonality correction term. For this, a variety of approaches exist. Because only the uncorrected scheme is considered in this work, only the minimum correction approach is detailed representatively as a counter example.

The diffusive flux evaluation with the correction term $\mathbf{E}_f$ is given by Eq. (22).

$$(\Gamma \nabla u)_f \cdot (\mathbf{S}_f - \mathbf{E}_f) \quad (22)$$

2.3.3.2 Minimum Correction

As the name suggests, the idea of this approach is to limit the orthogonal correction to a minimum. The calculation of $\mathbf{E}_f$ is given in Eq. (23).

$$\mathbf{E}_f = (\mathbf{e} \cdot \mathbf{S}_f) \mathbf{e} = (S_f \cos \theta) \mathbf{e}, \quad (23)$$

with $\mathbf{e}$ being the vector from the centre of C to the face and θ being the angle between the outward pointing vector $\mathbf{S}_f$ and the orthogonal vector to the neighbouring cell centre N .

2.4 The Roofline Model & Arithmetic Intensity

The roofline model relates the computational performance of an algorithm to the underlying memory traffic [Williams et al., 2008]. With the model, the measured performance of a kernel can easily be compared to the peak floating-point performance or peak memory bandwidth of the used hardware. As reported in [Williams et al., 2008], the memory traffic between caches and memory is measured instead of the traffic between the processor and the caches. The authors use *operational intensity* to describe the bandwidth requirements of a kernel to allow for memory optimisations to be reflected in the model. In the context of this thesis, *operational intensity* is referred to as *arithmetic intensity*.

The roofline model itself displays the floating-point performance, operational intensity, and memory performance in a two-dimensional graph, as illustrated in Figure 2.1. The peak floating-point performance can be obtained by consulting the hardware specifications or by running dedicated benchmarks. Like most of the kernels investigated in this thesis, the ones considered in [Williams et al., 2008] do not fit inside the the caches. Therefore, the peak memory performance is also obtained through the hardware specifications, though it can alternatively be obtained through microbenchmarks like STREAM [McCalpin, 1995].

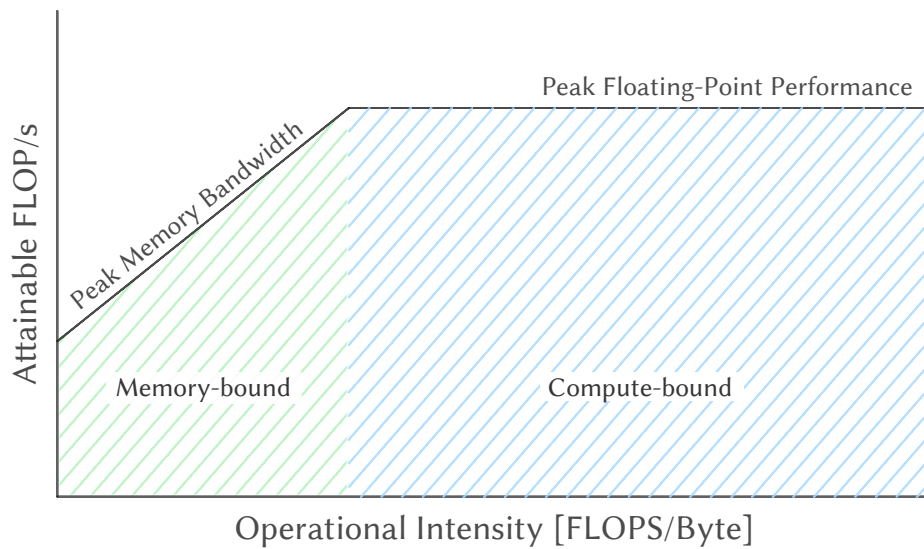


Figure 2.1.: The roofline model.

The peak memory bandwidth is a function of FLOPS per second and FLOPS per byte, resulting in a 45-degree diagonal line in the model. Both lines, the peak bandwidth and the peak floating-point performance, intersect at the so-called *ridge point* which provides additional insight on the overall machine-specific performance: The more the ridge point is located towards the right side of the model, the higher the operational intensity of a kernel needs to be to achieve maximum performance. Conversely, the intensity of a kernel can more easily saturate the machine's performance when the ridge point is placed more towards the left. In this context, a kernel is often referred to as memory-bound or compute-bound, stating that the theoretical maximum is limited by either the peak bandwidth or the peak floating-point performance of the system.

3 Related Work

This chapter presents other research or development projects that are thematically or conceptually related to this work. For the sake of completeness, note that Ansys Fluent and Moldex3D are existing CFD FVM alternatives to OpenFOAM. However, since both are closed source and proprietary, their matrix assembly or potential operator fusing strategies cannot be discussed here.

3.1 OpenFOAM Fused

Version v2412 of OpenFOAM introduces a preliminary implementation of fused operators³. As of now, only explicit operations are implemented. Fused operations are specified the same way as without fusing inside the fvSchemes(see Listing 3.1).

```
1  divSchemes {
2      div(phi,U)      bounded fusedGauss linearUpwind grad(U);
3  }
4  laplacianSchemes {
5      laplace          fusedGauss linear corrected;
6  }
```

Listing 3.1.: OpenFOAM operator fusing, shortened example for simplicity.

These fused operations seem to slightly improve performance, however, they do not solve our aforementioned problems with the OpenFOAM operator calculations. The fused operations are implemented alongside of their non-fused counterparts, thus increasing the template count, and thereby the compilation times. The implementation efforts are also increased, with no hint towards a potential strategy for replacing equivalent non-fused operations with their fused ones. It is reported that while the fused operations are *functionally* equivalent, the order of operations is not which leads to different truncation errors.

A year later, in version v2512, fusing of field operations have been introduced via C++ expression templates⁴. This contribution is quite similar to the contributions of this thesis, ignoring the fact that this still involves heavy template usage.

3.2 DSL Optimisation in NeoN

As mentioned in Chapter 1, NeoN recently introduced an experimental operator fusing concept that relies on explicitly specifying which operators can be fused. Through the

³<https://www.openfoam.com/news/main-news/openfoam-v2412/numerics#fused>

⁴<https://www.openfoam.com/news/main-news/openfoam-v2512/numerics#expressions>

`dsl::optimize` method, applicable operators are replaced by a semantically equivalent fused variant. `dsl::optimize` applies a range of optimisers to the expression. Each optimiser implements the same interface, receives an expression, and returns a transformed expression, which can be turned into an optimisation pipeline. Currently, the default pipeline contains only the `DivLapOptimizer` rule. It checks for the existence of both the implicit `DivOperator` and the implicit `LaplacianOperator`. If either term is absent, no transformation of the expression is performed. Otherwise, constructs a `GaussGreenDivLaplacian` operator that inherits the same configurations and replaces the two original operators with the fused one.

A minimal example of this DSL optimisation is shown in Listing 3.2 on the example of steady-state convection-diffusion. The two operators would be fused into one, and only one function is called during matrix assembly instead of two.

```
1 auto pde = dsl::imp::div(phi, U) - dsl::imp::laplacian(gamma, U);
2 auto optimized = dsl::optimize(pde);
```

Listing 3.2.: DSL optimisation in NeoN

The main disadvantage of this fusing strategy is, as mentioned in Chapter 1, the need to explicitly define fusable operator combinations, implementing the fused operators in addition to the unfused ones, and enabling this entire process through lots of templates.

3.3 XCALibre.jl

XCALibre.jl is a general purpose library for FVM CFD simulations [Medina et al., 2025]. The authors acknowledge that CFD systems are predominantly written in statically compiled languages for performance reasons. Noting that these codes are limited in their capabilities to adopt recent trends, XCALibre.jl is entirely written in Julia. Since the project is dynamically compiled, the main problems of C++ OpenFOAM backends discussed in Chapter 1 are not as relevant here. Furthermore, XCALibre.jl is entirely based on OpenFOAM. The conversion from the OpenFOAM `polyMesh` directory into their own mesh format is implemented, but any kind of scheme definition is handled programmatically:

```
1 schemes = (
2     U = Schemes(time=Euler, divergence=Upwind, gradient=Gauss),
3     p = Schemes(time=CrankNicolson, gradient=Gauss)
4 )
```

Listing 3.3.: XCALibre scheme definitions, extracted from the *2D_Cavity.jl* example⁵.

Similarly to the implementations of this work, ExCALibre includes GPU support for multiple vendors through the `KernelAbstractions` package. Considering the matrix assembly and operator evaluations, it seems that the implementation of EXCALibre is more similar to the experimental operator fusing of NeoN: The discretisation is performed in a single loop over the mesh that includes a nested iteration over all PDE terms. Thereby, their operator fusion strategy

⁵https://github.com/mberto79/XCALibre.jl/blob/main/examples/2D_cavity.jl

is realised not by fusing the operators into a single callable object, but rather through a list of terms. The advantage over NeoN, however, is that the fusion of operators does not require explicit configuration of valid combinations.

4 Methods

Transforming partial differential equations into sets of linear equations is called discretisation. It is a core element of the finite volume method (FVM). During this transformation, non-zero matrix coefficients are synthesised by applying the PDE operators like the divergence or Laplacian operator to a flow field and the specified mesh geometry. The mesh cells and faces are uniquely mapped to diagonal and off-diagonal entries. In this work, this entire process is called *matrix assembly*.

This matrix assembly is performed frequently inside iterative numerical approaches. For instance, consider the Pressure-Implicit with Splitting of Operators (PISO) algorithm used by IcoFoam as illustrated in Figure 4.1. The highlighted nodes mark algorithm steps wherein matrices are assembled. The solver runs for a pre-defined number of time steps. Each iteration, one momentum predictor step is followed by two corrector steps. The momentum predictor first assembles and solves the momentum equation, yielding a new mass velocity field which is subsequently used by the corrector steps. After the corrector steps, the systems are checked for convergence using the mass conservation equation. If the mass conservation equation is not satisfied, the inner loop continues. Otherwise, the simulation either stops because the time limit has been reached, or the results become inputs for the next iteration.

Because the inner loop may, even for geometrically simple cases, need thousands of iterations to converge, the matrix assembly would consequently also be called thousands of times [Serra and Semiao, 2021]. As mentioned in Chapter 1, the eager evaluation of IcoFoam increases the total number of assembled matrices per inner loop by the number of differential operators. NeoN solves this by performing the matrix assembly lazily but is limited due to runtime-only variables like the interpolation schemes. Thus, NeoN reduces the number of allocated matrices to one, but still passes over the matrix several

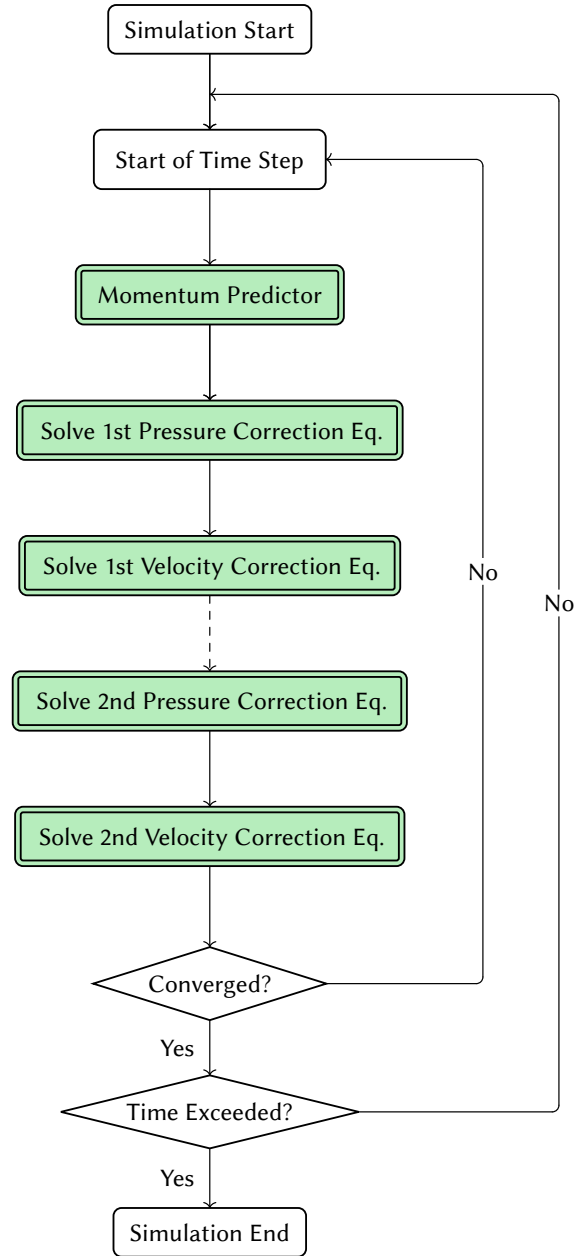


Figure 4.1.: PISO algorithm flow chart.

Taken and modified from
[Rana et al., 2022].

times, once for every operator.. Considering these simulations often run for thousands of time steps, it is obvious that reducing both the number of allocations and passes over the coefficient matrix will substantially reduce the time until solution.

Against this background, the delegation of the matrix assembly step to Julia is proposed. The matrix assembly can be further deconstructed into major building blocks. First, the iterations over the mesh geometry can be implemented in different ways, which are referred to as *assembly strategies* in this work. The strategies each have different performance implications with respect to cache locality, memory coalescence, and parallel efficiency. Second, multiple approaches of calculating the discretised PDE operators exist as well. Especially interesting for this work, of course, is fusing the operators to reduce the number of passes over the mesh. Lastly, numeric differences in the resulting assembled operators between assembly performed in standalone Julia and the extracted assembly as part of an interface are worth investigating. Through the research questions presented in Section 1.1, these three primary research topics of this work will be thoroughly evaluated.

The structure of this chapter conceptually graduates from matrix assembly internals towards the larger system that encompasses the matrix assembly. First, the PDE operators used in this work are presented, followed by the proposed approach of operator fusing. The interpolation schemes used by the operators and different variations of calculating them are detailed in Section 4.2. Thereafter, Section 4.3 presents four matrix assembly strategies and their arithmetic intensities. Lastly, the approach of extracting the matrix assembly step to Julia is reported. The chapter closes with a brief summary of the implemented strategies.

4.1 Operators

For the evaluation of the first research question, which deals with operator fusing and runtime abstractions of schemes, a set of representative PDE operators is selected. Using all the operators from the momentum equation seems like the obvious choice. However, the transient operator cannot be calculated in a face-based strategy without needlessly overcomplicating the algorithm. Therefore, the spatial operators, i.e., the divergence and Laplacian operators, are used for most of the evaluations in this work. The transient operator is only included in the final integration benchmark.

4.1.1 Div

In the cell-centric finite volume method, the field values are stored at the center of the control volumes, the cells. Part of the discretisation process, however, involves applying Gauss' divergence theorem [Katz, 1979] which allows expressing of a volume integral as an integral over the surface area. This surface integral is then approximated by taking the sum of values at the faces. Because the values are stored at the cell centres, they have to be interpolated to the faces. This is achieved through a selection of and an appropriate `divScheme`, i.e., an interpolation profile between the values stored at the centres. For a given internal face, the contributions of the owner and the neighbouring cell are calculated. Both are also applied to the two corresponding off-diagonal entries. Listing 4.1 shows the div operator calculation.

```

1  weight = div.scheme(faceFlux)
2  ownFluxContrib = -faceFlux * weight * d.scale
3  neiFluxContrib = +faceFlux * (1.0 - weight) * d.scale

```

Listing 4.1.: Calculation of the divergence operator.

The function stored under `div.scheme` can take the form of either the Linear or Upwind scheme. Both functions share the idea that an interpolation weight is calculated based on the flux at the cell face. The following two subsections briefly explain both. `d.scale` is simply any scalar factor the operator might be multiplied with. Note that it is generally possible to apply an operator to a field, in which case `d.scale` would not be scalar. For the sake of simplicity, this is not considered here.

4.1.2 Laplacian

For the calculation of the Laplacian operator, the diffusion coefficient Γ , a delta coefficient, and the magnitude of the face area are needed. A general calculation thereof is shown in Listing 4.2. The `deltaCoeff` stores the distance between the two cells of a face. `magFaceArea` describes the size of the face. Together with the diffusion coefficient and the distance to the neighbouring cell, it describes the size of the diffusive flux, i.e., how much of a variable crosses a face per time step. Like before, `lap.scale` is a factor the operator might be multiplied with. Like the div operator, the `diffusive_flux` calculated by the Laplacian operator contributes to the four matrix entries corresponding to both cells and the off-diagonal entries of the face.

```

1  diffusive_flux = gamma * deltaCoeff * magFaceArea * lap.scale

```

Listing 4.2.: Calculation of the Laplacian operator.

4.1.3 The Transient Operator

The transient operator describes the rate of change for a given variable ϕ inside a cell. Naturally, this operator only applies to simulations that are transient themselves. In steady-state simulations, the transient operator is simply deleted. In contrast to the spatial operators, the transient operator thereby focuses on the physical change of the same cell through time instead of the geometric relationships between neighbouring ones. In the implementation, this is realised by the `previousValues` variable. Here it represents a variably sized collection of previous values of ϕ . The cell volume is included as a scaling factor because larger cells contribute more than smaller ones. Another contrast to the spatial operators, the transient operator only contributes to the diagonal entries of the coefficient matrix and the right-hand side. This is shown in Listing 4.3. The operator relies more on the chosen scheme than the spatial operators, i.e., calculation of the transient operator is entirely comprised of the scheme.

```

1  diagContrib, RHSContrib = ddt.scheme(volume, previousValues) * ddt.scale

```

Listing 4.3.: Calculation of the transient operator.

4.1.4 Operator Fusing

In the context of this work, the term operator fusing is defined as the joint calculation of multiple, possibly all, operators inside one assembly loop instead of calculating each operator separately.

The concept of fused operators is not as prominent in FVM as it is in the finite difference (FDM) or finite element methods (FEM) [Carrascal-Manzanares Carlos et al., 2018]. In a broader sense, matrix-free methods (MFM), where matrix assembly is avoided and the coefficients instead computed inside the solver, used in both FEM and FDM could be compared to the definition of operator fusing in this work [Chen et al., 2024; Schussnig et al., 2025]. In MFM, the assembly of the coefficient matrix is directly included in the solver step of the algorithm. Therefore, operator calculations are inherently fused by design. More similar to the methodology of this work is the operator fusing proposed in [Huang et al., 2025]. Therein, the operators are fused using a pre-defined set of rules that classify each operator based on its inputs and outputs.

To accomplish this operator fusing, the PDE can be re-represented as a binary tree wherein the leaves are concrete operators. Any runtime schemes and scaling factors are simply members in the corresponding data structure (see Figure 4.2). In the example below, the slightly modified momentum equation from Chapter 1, is transformed into the binary tree structure. In contrast, NeoN implements the Expression as a flat list.

```

1  // case/system/fvSchemes:
2  divSchemes {
3    div(phi,U) Gauss linear;
4  }
5  // main.cpp:
6  Expression momentum (
7    2.0 * ddt(U)
8    + div(phi, U)
9    - laplacian(nu, U)
10 );

```

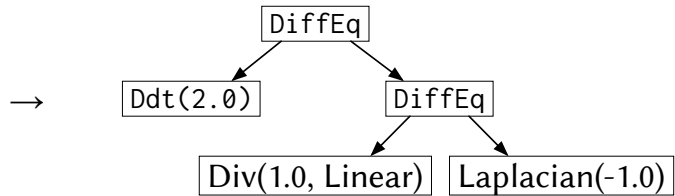


Figure 4.2.: Julia fused operator representation of example PDE.

This equation tree can then be evaluated in a bottom-up manner, i.e., evaluations and intermediate results start at the leaves and are propagated toward the root node. This requires that all operators implement matching function definitions. Implementation specifics thereof are detailed in Section 5.4.4. Passing arguments that might not be needed might seem inefficient, but has little performance impact since Julia defaults to passing arguments by sharing: A called function receives a local binding to the same object as the caller, which generally works similarly to passing by reference. Mutations to the object are visible to the caller, but assignment of a new value is only visible to the local function scope.

Furthermore, NeoN is evaluated with respect to their DSL-based approach to operator fusing to directly compare the different approaches.

4.2 Operator Schemes

In the preceding sections, three operators are presented. All three have in common that some properties like the flux need to be interpolated, e.g., from cell centre to the cell faces for the spatial operators and between distinct time steps for the transient operator. The convective and diffusive fluxes need to be interpolated from the cells to the faces in the calculation of the div and Laplacian operators. For the transient operator, the interpolation is used to derive the next value from previous ones in a certain way. The schemes used in this work are: Upwind and Gauss linear (div), Backward Euler (ddt), and the corrected and uncorrected surface normal gradient schemes (Laplace).

4.2.1 Div Schemes: Gauss Linear, Upwind

All div interpolation schemes revolve around the flux interpolation between two cells at the face (see Eq. (24)).

$$\phi_f = g\phi_C + (1 - g)\phi_N \quad (24)$$

This calculation is also found in Listing 4.1. The interpolation schemes differ in the calculation of g .

In general, the Gauss linear scheme, also called central differencing scheme, performs a weighted average to interpolate between two cells at the face. Applied to uniform meshes, both cells contribute equally to the face flux with $g = \frac{1}{2}$. On non-uniform meshes, the weighted sum is calculated, favouring the cell that is farther away from the face. This scheme is second-order accurate [Moukalled et al., 2015].

Interpolating between cells with no directional bias may work for symmetrical terms. However, for the convective term and the div operator, this is not adequate [Moukalled et al., 2015]. A better solution might be the upwind scheme. The idea is that the face flux needs to depend on the direction of the flow. Therefore, from which cell flux is taken from depends on the sign of the owner cell, i.e., the direction of the majority of the flow. Eq. (25) shows the calculation of g using the upwind scheme.

$$g = \begin{cases} 1 & \text{if flow towards neighbour N} \\ 0 & \text{otherwise} \end{cases} \quad (25)$$

The upwind scheme may be chosen in favour over the linear scheme due to its increased numerical stability in simulation. However, this also reduces the accuracy to first-order. This trade-off needs to be considered when designing the simulation [Epikhin, 2026].

4.2.2 Laplacian Schemes: Uncorrected

In OpenFOAM, the Laplacian scheme is a combination of an interpolation and surface-normal gradient scheme. The definition inside the fvSchemes file is shown in Listing 4.4.

```

1  laplacianSchemes {
2    laplacian(gamma,phi) Gauss <interpolation scheme> <snGrad scheme>
3  }

```

Listing 4.4.: Laplacian schemes in OpenFOAM.

All options are based on Gauss' divergence theorem [Katz, 1979]. The interpolation scheme can be any scheme also chosen for the div operator, e.g., upwind, linear, or any other that is not detailed in this work. The snGrad scheme specifies how the angle of the face between two cells is handled during flux computation. While NeoN provides functionality for multiple schemes, only the uncorrected scheme is implemented in this work. The uncorrected gradient scheme factors in the orthogonality of the face by multiplying the right hand side of Eq. (26) with $\alpha = \frac{1}{\cos(\theta)}$.

$$\nabla_f \phi = \alpha \frac{\phi_C - \phi_N}{|d|} \quad (26)$$

4.2.3 Transient Schemes: Backward Euler

Implemented in this work are the first and second-order backward Euler schemes, which are also referred to as BDF1 and BDF2, respectively (see Section 2.3.1). The backward Euler scheme approximates the derivative of the next time step using values from the previous time steps. The order of the transient scheme relates to how many previous time steps are used for the calculation of the next. For instance, the first-order scheme only uses the previous time step, or initial guesses, while the second-order takes the preceding two. Naturally, BDF2 cannot be calculated in the first time step. In those cases, a fallback to BDF1 is usually implemented (see Listing 4.3). The calculation of the first-order and second-order backward Euler schemes are illustrated in Listing 4.5 and Listing 4.6. Therein, oldPhi and oldOldPhi correspond to the previousValues variable of Listing 4.3.

```

1  a0a1 = 1.0 / Δt;
2  diagContrib = cellVolume * a0a1
3  rhsContrib  = cellVolume * a0a1 * oldPhi;

```

Listing 4.5.: First-order backward Euler.

The backwards Euler scheme is an implicit scheme. This entails that the value to interpolate occurs on both sides of the linear system. Furthermore, the second-order is not guaranteed to be bounded, i.e., the interpolated value is not guaranteed to remain in the range specified by the values used to calculate it [Hundsdoerfer et al., 2006].

```

1  a0 = 1.5 / Δt
2  a1 = 2.0 / Δt
3  a2 = -0.5 / Δt
4  diagContrib = cellVolume * a0
5  rhsContr = cellVolume * (a1 * oldPhi + a2 * oldOldPhi)

```

Listing 4.6.: Second-order backward Euler.

4.2.4 Abstraction of Interpolation Schemes

Regarding the runtime abstraction of different interpolation schemes, upwind and Gauss linear are selected as representatives. The choice of specific operators is not as important for the underlying question of how different variants of calculating the operators have an impact on performance.

In total, four variants of operator calculation are considered here for research question RQ I.b: *Pre-calculating* the interpolation weights, *dynamically* calculating the interpolation by calling a function that is passed as an argument, *statically* hard-coding the interpolation function inside the assembly loop, and finally, interpolating inside the *fused* operator calculations.

Including the fused variant directly addresses RQ I.b by juxtaposing our implementation of operator fusing against multiple baselines. At a minimum, comparable results will establish the viability of using fused operators in terms of runtime abstraction cost.

4.3 Matrix Assembly Strategies

This section presents the concept for the evaluation of different matrix assembly strategies. In this work, *matrix assembly* is defined as the calculation of non-zero coefficients from a flow field and a geometric mesh into a set of linear equations called the coefficient matrix. Therein, the *assembly strategy* refers to the order in which the geometric mesh is processed.

At the time of writing this thesis, the author of this work is not aware of any scientific literature on performance differences of different assembly strategies for FVM CFD applications. Additionally, evaluating different looping strategies is directly related to our research question RQ II.a. Implementation details of the assembly strategies are presented in Section 5.6.5.

In the following, four assembly strategies are discussed in detail: cell-based, face-based, global face-based, and batched face-based.

4.3.1 Cell-Based

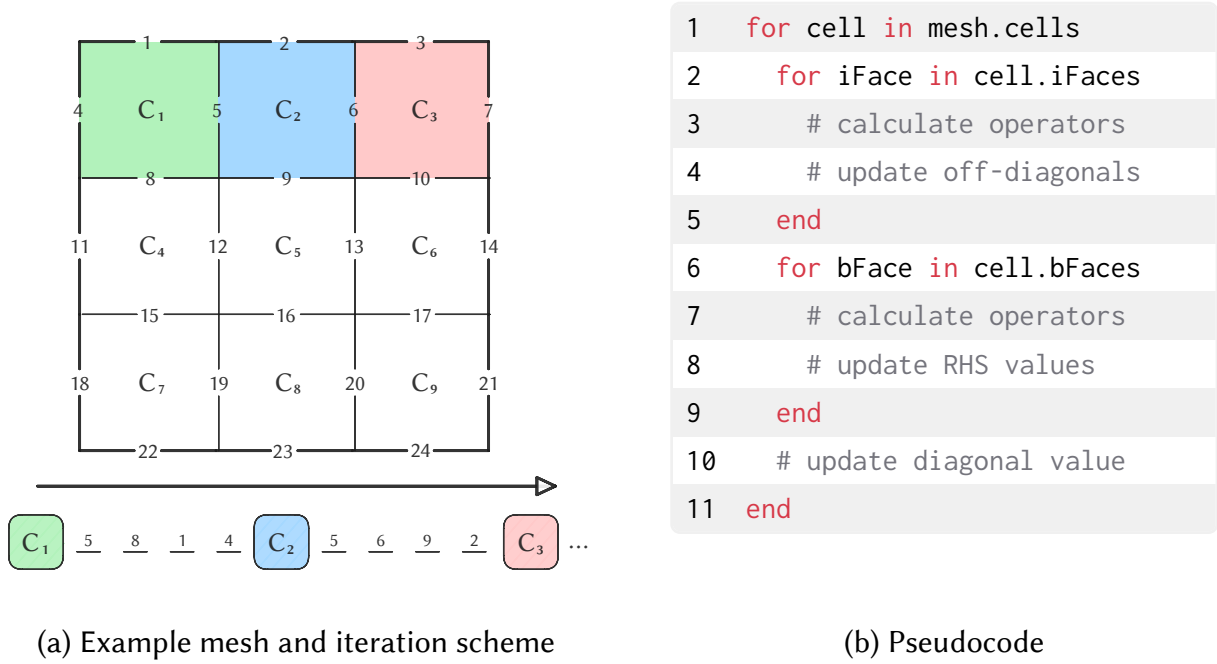


Figure 4.3.: Cell-based assembly. C_i and $-f-$ denote cell and face indices, respectively. The iteration order is illustrated below the example mesh.

In the cell-based assembly strategy, matrix coefficients are calculated by primarily iterating over the mesh cells, and secondarily over the cell's faces. This iteration scheme is illustrated in Figure 4.3a. The iteration order is to be read from left to right: C_1 is the active cell of the first iteration. The first inner loop iterates over the internal faces adjacent to C_1 , in this case $\{5, 8\}$. The second goes over the boundary faces $\{1, 4\}$. The outer loop continues with C_2 , in which the inner loops process the faces $\{5, 6, 9\}$ and $\{2\}$. Note that in Figure 4.3a, the faces are iterated over in ascending order for visualisation purposes. Generally speaking, the iteration order is arbitrary.

The pseudocode of the cell-based strategy in Figure 4.3b of the cell-based strategy illustrates the iteration procedure of Figure 4.3a. Each internal face represents the neighbourhood between two cells and corresponds to off-diagonal matrix coefficients. The contribution to the diagonal entry which corresponds to the active cell itself can be accumulated in a temporary variable and updated only once at the end of each iteration.

The access pattern of the cell-based strategy is shown in Figure 4.4. Therein, one advantage of the cell-based strategy becomes apparent: the iteration for C_i only writes coefficients into a single matrix row. Therefore, potential race conditions do not need to be prevented in a parallel context. Assuming an implementation similar to the pseudocode in Figure 4.3b, the write operations into the RHS vector can also be performed without atomics. Here, it is interesting to note that NeoN does this differently: Instead of processing the boundary faces inside the cell loop, they are instead iterated over in a separate loop. In that case, atomic operations are necessary again because a cell can have multiple boundary faces at edges and corners of the mesh. Furthermore, the computed coefficients are written into contiguous chunks of memory inside the coefficient matrix which positively affects performance due to the locality of data.

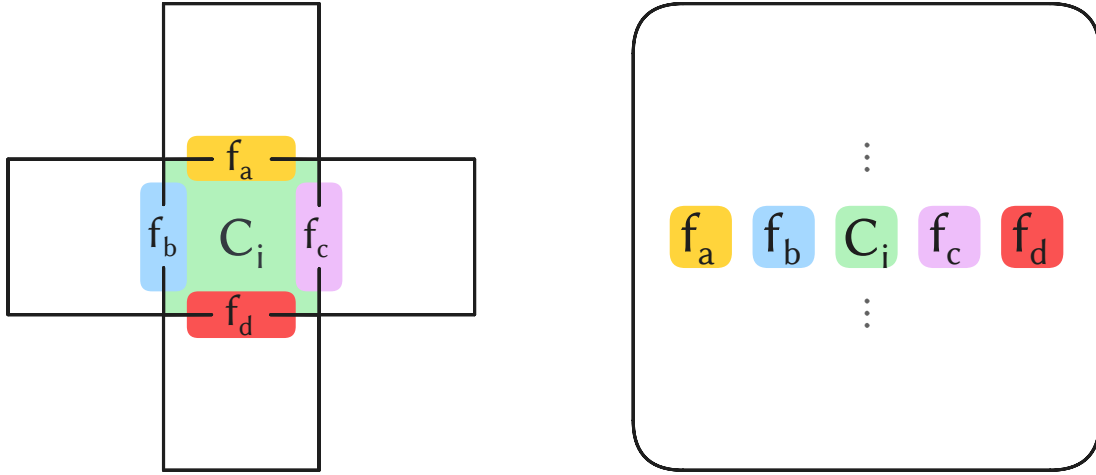
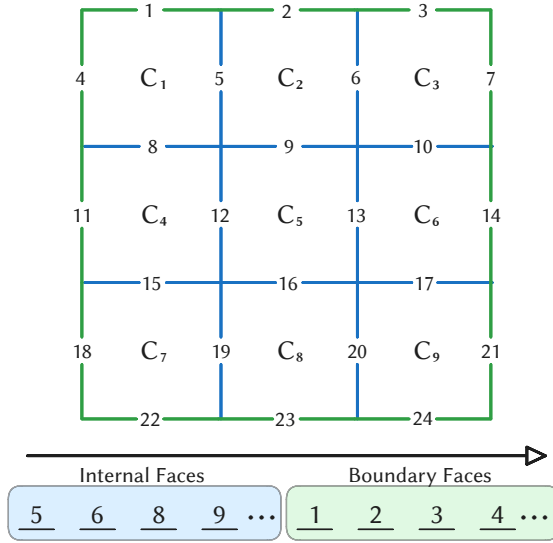


Figure 4.4.: Cell-based coefficient matrix access pattern. The left side shows a two-dimensional stencil of the mesh and the entries accessed in the coefficient matrix are drawn on the right.

Further, operators like the transient term $\text{ddt}(U)$ that only write into the matrix diagonal can only be calculated with the cell-centric strategy. When all operators are processed with the same strategy, the transient operator would have to be calculated outside of the face loops (see Figure 4.3b, line 11). Lastly, each internal face is processed once for each cell that borders it. Because operators like the divergence are not applied symmetrically, it is important to consider if the current cell owns the face or not. A check for face ownership is used to determine which values to update the diagonal and corresponding off-diagonal entries with. The implementation thereof is explained in Section 5.4.1.

While the cell-based strategy performs contiguous write accesses into the coefficient matrix, the performed read operations are not necessarily contiguous. Each cell holds a vector of adjacent face indices over which the internal loop iterates. This indirection often leads to poor memory locality. Additionally, the face indices of this small 2D example in Figure 4.5a might fit into the same cache line, but for larger three-dimensional meshes, the face indices are likely spaced further apart, which negatively affects performance due to frequent cache misses.

4.3.2 Face-Based



(a) Iteration schematic

```

1  for iFace in mesh.iFaces
2      # calculate operators
3      # update matrix values
4  end
5  for bFace in mesh.bFaces
6      # calculate operators
7      # update matrix values
8      # update RHS values
9  end

```

(b) Pseudocode

Figure 4.5.: Face-based assembly. C_i and $-f-$ denote cell and face indices, respectively. The iteration order is illustrated below the iteration scheme.

The face-based looping strategy is the one predominantly used in OpenFOAM. The assembly is split in two iterations; one loop for the internal faces, and one for all boundary faces. This behaviour is illustrated in Figure 4.5a. The internal faces $\{5, 6, 8, 9, \dots\}$ are processed in one pass and the boundary faces $\{1, 2, 3, 4, \dots\}$ afterwards. Note that this numbering is chosen for the simplicity of visualisation, and that realistic meshes are oftentimes sorted differently. For instance, in NeoN the boundary faces are sorted after the internal ones. In contrast to Figure 4.5b, the loops can then simply increment a counter to be used as the active face index for the current iteration. The pseudocode in Figure 4.5b is otherwise generally similar to the loop body of the cell-based strategy. The key difference is that here, the faces are taken from the entire mesh, and not singular cells.

In this strategy, every face is only calculated once. The matrix access pattern is illustrated in Figure 4.6. For a given face f_n between its owning and neighbouring cells, in this example C_a and C_b , the face-based strategy writes coefficient contributions into four entries: the two diagonal entries that correspond to the owner and neighbour, and two off-diagonal entries that represent the connection between the cells. Specifically, if the two cells correspond to the matrix diagonal entries A_{ii} and A_{jj} , then the off-diagonal entries that correspond to the face would be A_{ij} and A_{ji} , as illustrated by the right side of Figure 4.6.

Note the dotted gray elements, C_x and f_k , in Figure 4.6. Because both f_k and f_n are connected to C_a , both iterations would write into the corresponding matrix entry. This is why, in a parallel context, atomic operations are necessary to prevent race conditions when multiple faces can write into shared diagonal entries.

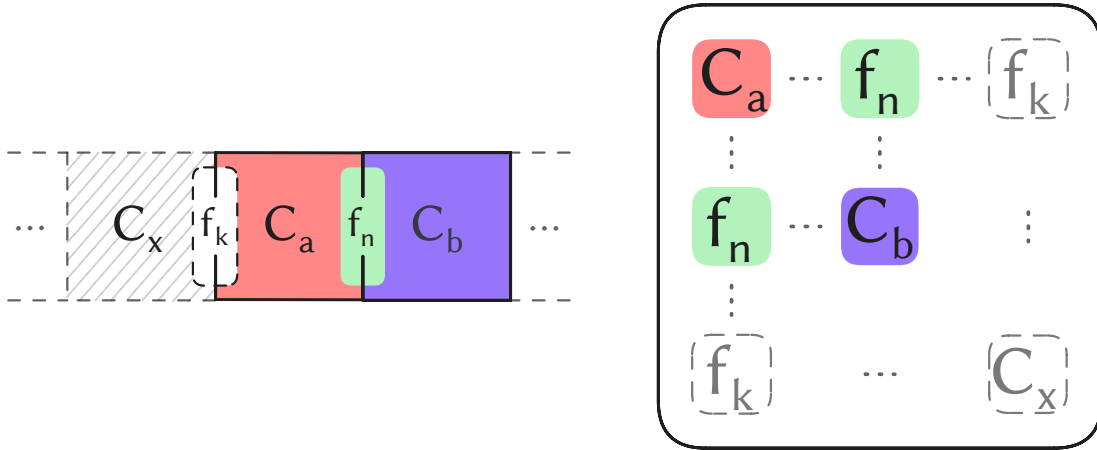
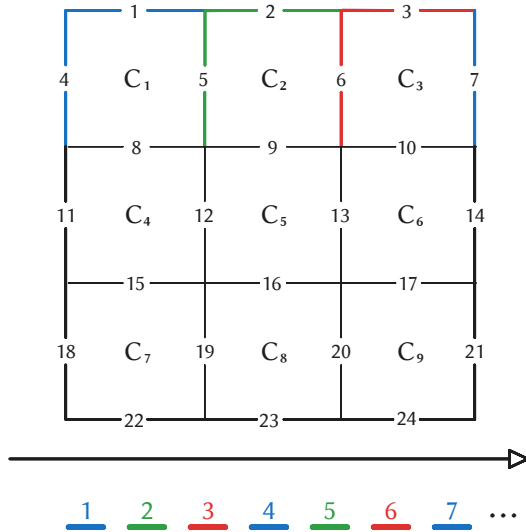


Figure 4.6.: Face-based coefficient matrix access pattern. The left side shows a slice of a one-dimensional mesh and the right side the corresponding matrix accesses.

Using the face-based strategy, calculating the transient operator is difficult because it only needs to be calculated once per cell. It is of course possible to mark a cell as processed and omit all subsequent visits, but this would pose further complications for the operator fusing. Compared to the cell-based strategy, this one functions entirely branchless on an algorithmic level.

4.3.3 Global Face-Based



(a) Iteration schematic

```

1  for face in mesh.faces
2      if face is internal
3          # calculate operators
4          # update matrix values
5      else
6          # calculate operators
7          # update matrix values
8          # update RHS values
9      end
10 end

```

(b) Pseudocode

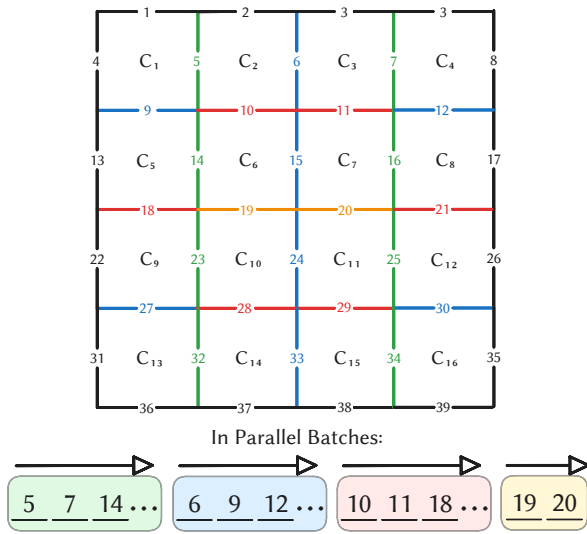
Figure 4.7.: Global face-based assembly. C_i and $-f-$ denote cell and face indices respectively. The iteration order is illustrated below the iteration scheme.

Closely related to the face-based strategy, the global face-based strategy moves the internal/boundary branch inside the loop. The iteration schema is illustrated in Figure 4.7a: All face indices are processed in ascending order without prior sorting. The pseudocode in Figure 4.7b shows that inside the loop, faces are differentiated between internal and boundary.

Conceptually, this can be done by checking if the face has a neighbour. Alternatively, and this is the way NeoN does this, the iteration counter can be compared to the number of internal faces.

The only difference between the global face-based strategy and the face-based one is that this strategy does not function branchless. Otherwise, the same advantages and disadvantages of the face-based strategy apply here; The write operations into the coefficient matrix are scattered, while the read operations are generally close to the previous iteration. Writing into the matrix diagonal entries and RHS requires atomics in a parallel context because cells can be shared between faces.

4.3.4 Batched Face-Based



(a) Iteration schematic

```

1  batches = colourEdges(mesh)
2  for batch in batches
3      # in parallel:
4      for iFace in batch
5          # calculate operators
6          # update matrix values
7      end
8  end
9  for bFace in boundaryFaces
10     # calculate operators
11     # update matrix values
12     # update RHS values
13 end

```

(b) Pseudocode

Figure 4.8.: Batched face-based assembly. C_i and $-f-$ denote cell and face indices, respectively. The iteration order is illustrated below the iteration scheme.

In the batched strategy, all internal faces are divided into groups using an edge colouring approach. Commonly applied to graph problems, an edge colouring algorithm assigns each edge of the graph a colour so that all edges of a given node have different colours. Here, the mesh cells and faces correspond to graph nodes and edges, respectively. The edge colouring illustrated in Figure 4.8a could be the result of a greedy approach that sequentially assigns the lowest possible id to a face, given the previously coloured faces of a given cell. In the example, the mesh has been partitioned into four clusters of internal faces: $\{5, 7, 14, \dots\}$, $\{6, 9, 12, \dots\}$, $\{10, 11, 18, \dots\}$, and $\{19, 20\}$.

As stated in the preceding two sections, face-based strategies require atomic operations to prevent race conditions when writing into diagonal entries of the coefficient matrix because multiple faces can belong to the same cell. Precisely this caveat is solved by applying the edge colouring algorithm: All faces within a batch can be processed in parallel without atomic opera-

tions. The boundary faces, representing only a fraction of all faces, are excluded in the colouring procedure and are handled the same way as the other face-based strategies.

The disadvantage of this strategy is that the contiguity is deteriorated by the edge colouring. In the other face-based strategies, the iteration over faces happened in a contiguous range $[1, \#InternalFaces]$. As drawn in Figure 4.8a, the iteration now happens in irregularly sized steps. Similar to the cell-based strategy, this may have negative effects on cache locality. Furthermore, implementing this strategy in an efficient manner requires elaborate splitting and rearranging of input data.

It is worth noting that the currently used greedy edge colouring may result in rather unbalanced batch sizes, depending on the mesh geometry. For the WindsorBody case introduced in [Varney, 2020], four colours contain the majority of faces, while the remainder is sorted into up to ten batches of rapidly decreasing size. In contrast, the geometrically much simpler lid-driven cavity case results in an equal subdivision of faces.

4.4 Arithmetic Intensities

RQ II.b focuses on the optimal performance of typical CFD kernels like, in this case, the matrix assembly strategies. Calculating the arithmetic intensity (AI) provides an indicator for whether the given assembly strategy is memory bound or compute bound. Before the following sections detail the AI per assembly strategy, some remarks regarding the calculation procedure are necessary.

Some architectures combine operations in the form of $d = a * b + c$ into one fused multiply-add (FMA) operation, virtually halving the floating point operations (FLOPs) in the process. For the sake of generality, the definition of [Golub and Van Loan, 2013] is adopted, which includes floating point add, multiply, subtract, and divide, and further counts FMAs as two separate FLOPs. The arithmetic intensity is given with respect to the calculation of a single iteration. Introducing the transient operator in this would require mixing of the strategies. As this would complicate the calculations, the AI is derived using only the divergence and Laplacian operators and their respective implementations in Listing 5.6, Listing 5.8, and Listing 5.9.

The face-based strategies share the same arithmetic intensity inside the loop and the cell-based strategy employs face-based processing of the boundary faces. Therefore, to avoid redundancy, the following sections present the arithmetic intensity for cell-based internal face, face-based internal face, and face-based boundary face calculations.

4.4.1 Cell-Based, Internal Faces

In addition to the operators, the arithmetic intensity of the cell-based strategy is based on its implementation in Listing 5.24. A structured mesh with six internal faces per cell is assumed for the derivation of the formula in Eq. (27). In the internal face loop, the spatial operators perform 12 FLOPs, and lines {16, 17} contribute four, bringing the internal loop to a total of $6 * 16 = 96$ FLOPs. In the cell loop, the lines {26, 27} add six FLOPs to the total. Note that the RHS update in line 30 corresponds to the transient operator and is therefore not included here.

Every read access of array entries is counted towards the memory traffic. The operator evaluation call reads five double-precision values, and the index calculation thereafter adds 4 byte through a 32-bit integer. Outside of the internal face loop, the `cellFacesSegments` vector is read from twice and the `rowOffs` and `diagOffs` vectors are accessed once each. Between those four vector accesses, two 32-bit integer and one unsigned 8-bit integer are loaded, resulting in a total of 13 bytes which brings the total to $6 * 44 + 13 = 277$ read bytes. The number of store operations into the coefficient matrix equate to $6 * 3 + 3 = 21$. Only double-precision values are stored. The total number of stored bytes is therefore 168.

$$AI_{\text{cell, internal}} = \frac{\#FLOPs}{\text{Memory Traffic}} = \frac{102}{277 + 168} \simeq 0.229 \quad (27)$$

4.4.2 Face-Based, Internal Faces

The formula of arithmetic intensity for the face-based strategy is derived from the aforementioned operator implementations and the processing of internal faces Listing 5.26.

The divergence and Laplacian operators perform a total of 12 FLOPs. Twelve additive or subtractive assignment updates to the matrix coefficients are performed in lines {15,17,19,21}, totalling 42 FLOPs. Every read access array entries is counted towards the memory traffic. In contrast to the cell-based strategy, no ownership sign is needed. Thus, the operator evaluation call only reads four doubles. The preceding array accesses and subsequent index calculations read four 32-bit and four 8-bit integers, resulting in $4 * 8 + 4 * 4 + 4 * 1 = 52$ of total read bytes per internal face. No RHS updates are calculated here, which reduces the number of memory writes to $4 * 3 = 12$. Only double-precision values are stored. The total number of stored bytes is therefore 96.

$$AI_{\text{face, internal}} = \frac{\#FLOPs}{\text{Memory Traffic}} = \frac{42}{52 + 96} \simeq 0.283 \quad (28)$$

4.4.3 Face-Based, Boundary Faces

The arithmetic intensity is derived from the operator boundary implementations in Listing 5.7, Listing 5.10, and the face-based boundary face calculations presented in Listing 5.28. The boundary implementations for the div and Laplacian operators perform 25 and 27 FLOPs, respectively. Updating the coefficient matrices and right-hand-sides performs a sum of 12 FLOPs, bringing the total to 64. Eleven double-precision floats are read for the operator evaluation. Thereafter, two 32-bit and one unsigned 8-bit integers are loaded. The total read bytes is therefore $11 * 8 + 2 * 4 + 1 = 97$. The amount of written bytes coincides with the twelve updates to matrix coefficients and RHS values and results in $8 * 12 = 96$ bytes.

$$AI_{\text{face, boundary}} = \frac{\#FLOPs}{\text{Memory Traffic}} = \frac{64}{97 + 96} \simeq 0.331 \quad (29)$$

4.5 JuNe: Integrating Julia into NeoN

In an effort to answer research questions RQ III.a - RQ III.c, an interface between NeoN, representing C++ CFD Systems, and the Julia matrix assembly is developed. The development in itself is a rather explorative effort towards embedding Julia packages into C++ CFD system, as this has, to our knowledge, never been done before. Prior benchmarks focus on the efficiency of matrix assembly and operator fusing in both languages and enable evaluating the developed interface in isolation. In the remainder of this work, the interface between the Julia matrix assembly and NeoN will also be referred to as *JuNe*.

To establish any connection between C++ and Julia, several options exist. However, most focus on providing C++ functionality for Julia. Therefore, packages like `CxxWrap.jl`⁶, `Clang.jl`⁷ or `Cbinding.jl`⁸ are not an option for us here. To our knowledge, only two packages implement the correct interface direction for our use-case. First, the C API can be used by including `julia.h` inside the C++ project. Alternatively, `jluna`⁹ is a package that provides quality-of-life wrappers around the C API. At the time of writing this, the last commit to the `jluna` repo dates back to april of 2024. Additionally, multiple issues regarding missing support of Julia 1.11, thrown errors in multi-threaded contexts, and failing compatibility with `CUDA.jl` are open in the repository. Since all three of those are required, the C API is ultimately chosen.

One important aspect to note is that the operator calculations of the interface differ from the operators calculations performed in the standalone Julia matrix assembly. The standalone assembly is implemented based on a rather simple prototype, while NeoN is much more developed. Therefore, the standalone matrix assembly is not integrated into NeoN using the interface. Instead, a second, more concise assembly is implemented alongside of it.

Research question RQ III.c focuses less on the quantitative effectiveness of the extracted matrix assembly, but rather on the qualitative part of developing it. Included therein is the time spent waiting for the program to compile. While the compilation time plays a secondary role for the simulations and benchmarks, it is a central part of the development process. Furthermore, as stated in the Introduction, enabling runtime variability through lots of templates drastically increases compilation time. Removing all templates that have now become obsolete would be a major undertaking in the NeoN codebase. As this is out of scope for this thesis, the increase in compilation time induced by the interface can instead be measured. It will be considered a success, if the interface does not increase the time by a substantial amount. The results will be presented at the end of Chapter 6 and pose a segue into Chapter 7, wherein the development efforts will be discussed more qualitatively.

Implementing the batched face-based strategy into JuNe would require extensive development on the NeoN code base. The edge colouring algorithm, additional abstractions to allow for the resulting partition and the entire flow of data through the application would be affected by this implementation. Complex refactoring activities like this are outside the scope of this thesis, but

⁶<https://github.com/JuliaInterop/CxxWrap.jl>

⁷<https://github.com/JuliaInterop/Clang.jl>

⁸<https://github.com/analytech-solutions/CBinding.jl>

⁹<https://github.com/Clemapfel/jluna>

will be considered for future work. Conclusively, the batched strategy is only implemented in the standalone Julia matrix assembly.

4.6 Summary

This chapter opened with the contextualisation that improving the matrix assembly of typical CFD simulations is meaningful because it may be calculated hundreds, if not thousands, of times. Thereafter, the PDE operators of the momentum equation and one of the main contributions of this work, the operator fusing, are presented in Section 4.1. These operators rely on a case-specific selection of schemes which are detailed in Section 4.2. That section also includes remarks on the various ways of evaluating these schemes, which directly relates to RQ I.b. Section 4.3 provides insights on the four matrix assembly strategies evaluated in this work. The section thereafter derives the arithmetic intensities of each strategy.

Table 4.1 acts as a brief summary of sections Section 4.3 and Section 4.4. The cell-based, face-based, and global face-based strategies are implemented in CPU, multi-threaded, and GPU. The batched face-based strategy is only implemented in the standalone Julia for the reasons previously detailed. Because it is inherently designed for parallel use, the usefulness of a serial implementation is questionable. Still, the serial implementation is created for the calculation of the speedup metric. This is marked in Table 4.1 as (J). The arithmetic intensity is reported with respect to processing the spatial operators div and Laplace with JuNe, as derived in Section 4.4. More specifically, all but the bottom row report the intensity of processing the internal faces. The intensity of processing boundary faces is reported in the bottom row for the sake of completeness. As discussed at the end of Section 4.5, no batched strategy is implemented into JuNe. Regardless, an arithmetic intensity for the batched face-based strategy is presented, as the loop-internal implementations would not differ from the face-based or global face-based strategies. Note that the internal arithmetic intensities for the three face-based strategies are identical. This is explained by the fact that the calculations on the faces are identical, and that the strategies primarily differ by the outer structure. The right-most column gives the corresponding section wherein each assembly strategy is presented.

Table 4.1.: Assembly Strategies. Strategies that are implemented in standalone Julia and JuNe are marked with J and N, respectively. The serial batched face-based strategy was implemented, though it is designed for parallel use. *Arithmetic Intensity* is calculated with respect to the work inside each iteration.

Assembly Strategy	CPU		GPU	Arithmetic Intensity (JuNe)	Section
	Serial	Threaded			
Cell-Based	J, N	J, N	J, N	0.229 FLOP/Byte	4.3.1
Face-Based	J, N	J, N	J, N	0.283 FLOP/Byte	4.3.2
Global Face-Based	J, N	J, N	J, N	0.283 FLOP/Byte	4.3.3
Batched Face-Based	(J)	J	J	0.283 FLOP/Byte	4.3.4
Boundary Faces	J, N	J, N	J, N	0.331 FLOP/Byte	

Before closing this chapter, the primary contribution of this thesis is presented in Section 4.5, namely the integration of Julia matrix assembly into NeoN. The next chapter will elaborate on the implementation details of the conceptualisations discussed in this chapter.

5 Implementation

In this chapter, the implementation specifics for the concept presented in the preceding chapter are detailed.

The C++ side of the interface implementation can be found on the `enh/julia` branch of the NeoN GitHub repository¹⁰. Tests and benchmarks are implemented on the `enh/julia` branch of NeoFOAM¹¹. The extracted Julia matrix assembly can be found in the `JuNe.jl` subdirectory of the GitHub repository¹². Alongside thereof, the code for the standalone matrix assembly, benchmarking results, and more can be found as well.

5.1 OpenFOAM Cases

OpenFOAM is an open-source CFD simulation system written in C++. The simulation process always starts with the case definition. Listing 5.1 shows the basic directory structure of an OpenFOAM case.

Each case includes a `system` and a `constant` directory, as well as an initial time directory and, optionally, multiple restart time directories. The `system` directory contains configurations for all OpenFOAM operations and plugins are in some capacity useful for the simulation. For instance the `blockMesh` operation reads its configuration from the `blockMeshDict` to generate the mesh and write its data into the `constant` directory. Conceptually, `blockMesh` takes a defined geometry and decomposes it into a collection of at least one hexahedral block. Lots of other functionalities are included in `blockMesh`. However, the knowledge that it decomposes a geometry into smaller control volumes, the cells, is sufficient for the context of this work.

1	case
2	├─ 0
3	│ ├─ T
4	│ ├─ U
5	│ └─ p
6	├─ constant
7	│ └─ polyMesh
8	│ │ └─ boundary
9	│ │ └─ faces
10	│ │ └─ neighbour
11	│ │ └─ owner
12	│ └─ points
13	└─ transportProperties
14	├─ system
15	│ └─ blockMeshDict
16	│ └─ controlDict
17	│ └─ decomposeParDict
18	│ └─ fvSchemes
19	└─ fvSolution

Listing 5.1.: OpenFOAM case structure.

¹⁰<https://github.com/exasim-project/NeoN>

¹¹<https://github.com/exasim-project/NeoFOAM>

¹²<https://github.com/lupeterm/FVM-MA>

Out of the other files in the system directory, `fvSchemes` is most relevant for this work. Therein, the various interpolation and time integration schemes used by the PDE operators are defined. To add more context, these are the schemes discussed in Chapter 1 that are only known at runtime. A small example is illustrated in Listing 5.2. For the transient operator `ddt`, the implicit first-order Euler integration scheme is selected. Moreover, as `divSchemes` show, different interpolation schemes can be selected for each transported variable. Here, for `div(phi, U)` the linear scheme is chosen, while the temperature `T` will be transported using the upwind scheme.

```

1  ddtSchemes {
2      default          Euler;
3  }
4  divSchemes {
5      div(phi,U)        Gauss linear;
6      div(phi,T)        Gauss upwind;
7  }
```

Listing 5.2.: Scheme definitions in `fvSchemes`.

The constant directory describes physical constants, model parameters, and holds the computation mesh of the simulation. The files therein are assembled into the mesh. OpenFOAM meshes are geometric structures that hold collections of cells, faces, and boundaries. Conceptually, the geometry of a simulation is subdivided into smaller sub-geometries called cells. Faces represent the boundaries between two adjacent cells, one of which is considered the owner and the other the neighbour. The vector of three-dimensional coordinates in points defines the mesh vertices, i.e., the cell corners. The `faces` file contains an array of corner points of each face. Note, that while faces are usually rectangular, these face shapes are generally arbitrary and depend on the problem complexity and design.

The `owner` and `neighbour` files include a list of indices that map from a face index to the corresponding owner and neighbour cell index. `transportProperties` defines any physical constants related to viscosity, diffusivity, and conductivity. In Listing 5.1, the time directory `0` defines the initial temperature, velocity, and pressure fields. OpenFOAM differentiates between surface, volume, and patch fields. Volume fields store the field data at the cell centres and surface fields at the face centre. A patch is a subset of a surface field, typically corresponding to a domain boundary. Volume fields relate values to cells, surface fields relate to faces, and patches correspond to boundaries.

5.2 Case Generation

As explained above, `blockMesh` can be used to generate meshes of defined sizes. However, manually creating multiple meshes for parameter studies of varying sizes is tedious. Motivated by this, the OpenFOAM benchmark runner (OBR) [Gärtner et al., 2025] was developed. This tool allows reading meta configuration YAML files to easily perform a range of operations. Using the provided generator functionality, `blockMesh` can be automatically performed on a range of mesh dimensions. The meta configuration shown in Listing 5.3 describes loading a case definition

of the lid-driven cavity case from a specified repository. Thereafter, OBR calls `blockMesh` to decompose the original geometry into increasingly finer meshes. In addition to the original mesh with 100^3 (1M) cells, three meshes with 8 million, 27 million, and 64 million cells are generated. Similar to `blockMesh`, OBR includes many other functionalities which are beyond the scope of this work.

```

1  case:
2      type: GitRepo
3      url: https://develop.openfoam.com/committees/hpc.git
4      folder: Lid_driven_cavity-3d/S
5      commit: f9594d16aa6993bb3690ec47b2ca624b37ea40cd
6      post-build:
7          ...
8  - operation: blockMesh
9      schema: "{cells}"
10     generator:
11         key: cells
12         values: [100, 200, 300, 400]
13         template:
14             - modifyBlock: "(100 100 100)->(cells cells cells)"

```

Listing 5.3.: Generation of multiple Lid-driven cavity cases using OBR. The post-build configuration is omitted for brevity.

5.3 Interpolation Schemes

In this section, the implementations of interpolation schemes are presented. As discussed in Section 4.2.2, only the uncorrected Laplacian scheme is implemented. More specifically, it is implemented directly into the operator without an explicit implementation as a scheme. Therefore, only the div and transient schemes are detailed here. Note that an array of other schemes exist, but only the most fundamental schemes are discussed in this work.

5.3.1 Div Schemes

The implemented interpolation schemes for the div operator are illustrated in Listing 5.4. First, an abstract `DivScheme` type is defined from which the `upwind` and `linear` structs inherit. Both schemes are defined as so-called *functor structs*, i.e., structs that are callable. The `upwind` is implemented using a ternary operator, returning 1.0 if the flow is directed towards the neighbouring cell, and 0.0 otherwise. The `linear` scheme always returns 0.5 in this current implementation, representing linear interpolation in orthogonal grids. The type annotation specifies `P` to be a subset of `AbstractFloat`, i.e., this implementation works for floats of different precisions. Implemented in Julia are 16, 32, 64, and arbitrary-precision floats using `BigFloat`. While the functionality exists, all calculations in this work are performed using 64-bit precision.

Furthermore, the function calls are additionally modified with the `@inline` macro. This suggests to the JIT compiler the function body should be inserted into the call site in place of the call itself. This may be beneficial for small functions in a high-performance context. However, the compiler may ignore the suggestion, should the inlining be evaluated as maleficial. This inlining is also used in the evaluation of operators and the operator fusing in Section 5.4.4.

```

1  abstract type DivScheme{P} end
2
3  struct upwind{P} <: DivScheme{P} end
4  @inline (u::upwind{P})(ϕ) where {P<:AbstractFloat} = ϕ > 0 ? 1.0 : 0.0
5
6  struct linear{P} <: DivScheme{P} end
7  @inline (u::linear{P})(ϕ) where {P<:AbstractFloat} = 0.5

```

Listing 5.4.: Implemented div schemes.

5.3.2 Temporal Integration Schemes

Like the div schemes, the transient schemes are implemented as functor structs that inherit from the same abstract type, `DdtScheme`. This is shown in Listing 5.1, with the first-order Backward Differentiation Formula (BDF) on the left and the second-order on the right. In this work, these two are also referred to as BDF1 and BDF2. The first-order scheme accepts the value from the previous time step `oldPhi` as input, while the second-order scheme accepts the values from the previous two. Otherwise, the calculations performed are exactly those previously outlined in Section 4.2.3. The `.*` syntax seen for example in Listing 5.1a:12 denotes a broadcasted multiplication, i.e., the diagonal contribution is multiplied with each element in `oldPhi`. This could even be further shortened into `oldPhi .*= diagCon`. Regardless, this was not implemented to preserve a more descriptive data flow through variable naming.

```

1  abstract type DdtScheme{P} end
2
3
4  struct BDF1{P} <: DdtScheme{P}
5      a0a1::P
6  end
7  @inline function (bdf::BDF1{P})(
8      volume::P,
9      oldPhi::Vector{P}
10 ) where {P<:AbstractFloat}
11     diagCon = volume * bdf.a0a1
12     rhsCon = diagCon .* oldPhi
13     return diagCon, rhsCon
14 end
15

```

(a) first-order

```

1  struct BDF2{P} <: DdtScheme{P}
2      a0::P; a1::P; a2::P
3  end
4  @inline function (bdf::BDF2)(
5      volume::P,
6      oldPhi::Vector{P},
7      oldOldPhi::Vector{P}
8  ) where {P<:AbstractFloat}
9      diagCon = volume * bdf.a0
10     rhsCon = volume * (
11         bdf.a1 .* oldPhi
12         + bdf.a2 .* oldOldPhi
13     )
14     return diagCon, rhsCon
15 end

```

(b) second-order

Listing 5.1.: Struct definitions of the backward Euler transient schemes.

However, the implementation here differs slightly from the previous remarks. That is because the coefficients are automatically divided by Δt during instantiation of the operator, as shown in Listing 5.5. Here, it is important to note that BDF2 is a regular function that merely shadows the name of the BDF2 struct. This is somewhat necessary due to the fact that custom constructors are not a thing in Julia like they are in other languages. This specific implementation of the function call is also important for another reason, which will be explained in more detail in Section 5.6.2. For now, it suffices to say that this definition is necessary for the API between NeoN and JuNe. An analogous function for the first-order backward Euler scheme is implemented, but not shown here for the sake of brevity.

```

1  function BDF2(Δt::P)::BDF2{P} where {P<:AbstractFloat}
2      return BDF2{P}(1.5 / Δt, 2.0 / Δt, -0.5 / Δt)
3  end

```

Listing 5.5.: BDF2 instantiation.

5.4 Operators

In this section, the implementation details of the selected PDE operators are explained. Four matrix assembly strategies were presented in the preceding chapter, one cell-based and three face-based ones. The face-based strategies perform the same operator calculations, but the cell-based strategy is slightly different. Therefore, a dedicated operator implementation for each strategy is created. Furthermore, different implementations are produced, distinguishing between internal faces and boundary faces. Because it is not reasonable to calculate the transient

5.4 Operators

operator in a face-based strategy, it is only implemented for the cell-based strategy. To calculate the transient operator in a face-based strategy would require performing the evaluation only once for each owning cell. A naive implementation that uses a bit array to check if a cell has been processed, would not only increase the memory footprint, but also add overhead by introducing branches into the loop.

Listing 5.2 shows the struct definitions of the momentum equation operators. As discussed in Section 4.2.2, only the uncorrected gradient scheme is implemented in this work. Therefore, no dedicated struct member is implemented for the Laplacian operator.

1 <code>struct Ddt{P,S}</code>	1 <code>struct Div{P,S}</code>	1 <code>struct Laplace{P}</code>
2 <code>scheme::S</code>	2 <code>scheme::S</code>	2
3 <code>scale::P</code>	3 <code>scale::P</code>	3 <code>scale::P</code>
4 <code>end</code>	4 <code>end</code>	4 <code>end</code>
(a) Transient Operator	(b) Divergence Operator	(c) Laplacian Operator

Listing 5.2.: Struct definitions of all momentum Equation PDE operators.

In the GPU implementation, all operators share that non-scalar input arguments need to be split and passed as individual scalars. This applies to, for instance, the phi of previous time steps or reference values at the boundaries (see Listing 5.10, Listing 5.7, and Listing 5.3). When trying to pass data by ranged index slicing into the operators inside a CUDA kernel, the JIT compilation throws an error, reporting that the compilation resulted in an invalid LLVM intermediate representation. This is likely caused by the fact that taking a slice of an array creates a small array in temporary memory. More importantly, slices are dynamically sized, and allocating dynamically sized arrays is forbidden inside kernels.

Similarly, ranged assignments of the form `foo[start:stop] = bar[start:stop]` cannot be performed atomically. Therefore, these assignments need to be unrolled in a parallel context, i.e., `foo[start] = bar[start]; ... foo[end] = bar[end]`. For this reason, some values calculated by the operators are passed per dimension instead of a three-element vector.

To support the operator fusing approach presented in Section 5.4.4, all fusable operators need to implement the same function definition. The concrete result of this can be seen by comparing Listing 5.6 and Listing 5.9: Arguments that are used by another operator, but are needed for the calculation of another are denoted by underscores.

In the following sections, the implementation details of the PDE operators used in this work are discussed, followed by the implementation of operator fusing.

5.4.1 Div

Listing 5.6 shows the implementation of the div operator call for internal faces in the face-based strategy. Therefore, three scalars are unused here, as marked with the underscore. The `divScheme` stored as a struct member is called to calculate the interpolation weight.

```

1  function (div::Div{P,S})(
2      faceFlux::P,
3      _::P,
4      _::P,
5      _::P,
6      ownFluxContrib::P,
7      neiFluxContrib::P
8  ) where {P<:AbstractFloat,S}
9      weights_f = div.scheme(faceFlux)
10     ownFluxContrib += -faceFlux * weights_f * div.scale
11     neiFluxContrib += +faceFlux * (1.0 - weights_f) * div.scale
12     return ownFluxContrib, neiFluxContrib
13 end

```

Listing 5.6.: Face-based div calculation for internal faces. `_::P` denotes variables only needed for the Laplacian operator.

Listing 5.7 illustrates the calculation of the div operator at the boundary faces. The first five inputs all correspond to the boundary condition (BC) defined in the case. In certain boundary conditions, imaginary cells beyond the mesh boundary may be defined to still allow interpolation to the boundary face. Some of these BC may define a fixed value for these imaginary cells, which is what `refVal` represents. Alternatively, instead of defining a fixed value, a gradient can be used to define how the value at the boundary cell may be transported through the boundary face. `valFrac` is a scalar between zero and one which specifies how much of the reference value is fixed. Lastly, the delta coefficient variable `dCoeff` is used to account for any non-orthogonality of boundary faces.

This implementation is used for both CPU serial and multi-threaded. Lines 13-15 can be joined into a compound statement like `RHS = bflux * valFrac .* refVal + invFrac .* refGrad ./ dCoeff`. It is split, however, because ranged assignments cannot be made atomic in a parallel implementation. Note that lines 13 through 18 are split for readability.

5.4 Operators

```
1  function (div::Div{P,S})(
2      refVal::AbstractVector{P},
3      refGrad::AbstractVector{P},
4      bflux::P,
5      valFrac::P,
6      dCoeff::P,
7      _::P, _::P,
8      valueDiag::P, valueRHSx::P, valueRHSy::P, valueRHSz::P
9  ) where {P<:AbstractFloat,S}
10     invFrac = 1.0 - valFrac
11     valueDiag -= bflux * invFrac * div.scale
12     RHSx = bflux * valFrac * refVal[1] + invFrac * refGrad[1] / dCoeff
13     RHSy = bflux * valFrac * refVal[2] + invFrac * refGrad[2] / dCoeff
14     RHSz = bflux * valFrac * refVal[3] + invFrac * refGrad[3] / dCoeff
15     valueRHSx -= RHSx * div.scale
16     valueRHSy -= RHSy * div.scale
17     valueRHSz -= RHSz * div.scale
18     return valueDiag, valueRHSx, valueRHSy, valueRHSz
19 end
```

Listing 5.7.: Face-based div calculation for boundary faces. `_::P` denotes variables only needed for the Laplacian operator.

The cell-based implementation for the div operator is shown in Listing 5.8. Overall, the same calculations are performed as in Listing 5.6. The only addition to the face-based implementation is that the operator checks whether the cell owns the current internal face, or not. The ownership is currently evaluated against a sign of type `AbstractFloat`, which includes `Float32` and `Float64`. Conceptually, a single bit would suffice. However, because NeoN currently implements this as a double, it is also implemented as such here.

```

1  function (div::Div{P,S})(
2      faceFlux::P,
3      _::P,
4      _::P,
5      _::P,
6      diagContrib::P,
7      offdiag::P,
8      sign::P
9  ) where {P<:AbstractFloat,S}
10     weights_f = div.scheme(faceFlux)
11     if sign > 0 # cell owns the face
12         diagContrib += faceFlux * weights_f * div.scale
13         offdiag      += faceFlux * (1.0 - weights_f) * div.scale
14     else      # cell is the neighbour
15         diagContrib += -faceFlux * (1.0 - weights_f) * div.scale
16         offdiag      += -faceFlux * weights_f * div.scale
17     end
18     return diagContrib, offdiag
19 end

```

Listing 5.8.: Cell-based div calculation for internal faces. `_::P` denotes variables only needed for the Laplacian operator.

5.4.2 Laplacian

Listing 5.9 shows the implementation of the Laplace operator for internal faces in the face-based strategy. The needed arguments here are the inverse to the div operator (see Listing 5.6): The first argument, the interpolated face flux, is not needed for the Laplacian operator. Instead, the diffusion coefficient γ , the delta coefficient, and the magnitude of the face area are used. They are multiplied together with a potential scaling factor as specified in the original PDE and added to the owner and neighbour contributions. For the sake of brevity, the cell-based implementation is not shown here. The only difference to the face-based implementation in Listing 5.9 is an additional input argument `_::P` which corresponds to the face ownership sign (see Listing 5.8). In any case, it is not used in the Laplacian operator.

5.4 Operators

```
1  function (lap::Laplace{P})(
2      _::P,
3      gamma::P,
4      deltaCoeff::P,
5      magFaceArea::P,
6      ownFluxContrib::P,
7      neiFluxContrib::P
8      #_::P in the cell-based implementation
9  ) where {P<:AbstractFloat}
10     flux = gamma * deltaCoeff * magFaceArea * lap.scale
11     return ownFluxContrib + flux, neiFluxContrib + flux
12 end
```

Listing 5.9.: Face-based Laplacian calculation for internal faces. `_::P` denotes variables only needed for the Div operator.

The calculation of the Laplace operator at the boundaries is shown in Listing 5.10. Generally, the previous remarks on the boundary calculation of the divergence operator apply here as well. The first five input arguments correspond to the defined boundary condition. The third argument is only used in the div calculations.

```

1  function (lap::Laplace{P})(
2      refVal::AbstractVector{P},
3      refGrad::AbstractVector{P},
4      _::P,
5      valFrac::P,
6      deltaCoeff::P,
7      gamma::P,
8      magFaceArea::P,
9      valueDiag::P, valueRHSx::P, valueRHSy::P, valueRHSz::P
10 ) where {P<:AbstractFloat}
11     invFrac = 1.0 - valFrac
12     flux = gamma * magFaceArea
13     valueDiag -= flux * valFrac * deltaCoeff * lap.scale
14     x = flux * (valFrac * deltaCoeff * refVal[1] + invFrac * refGrad[1])
15     y = flux * (valFrac * deltaCoeff * refVal[2] + invFrac * refGrad[2])
16     z = flux * (valFrac * deltaCoeff * refVal[3] + invFrac * refGrad[3])
17     valueRHSx -= x * lap.scale
18     valueRHSy -= y * lap.scale
19     valueRHSz -= z * lap.scale
20     return valueDiag, valueRHSx, valueRHSy, valueRHSz
21 end

```

Listing 5.10.: Face-based Laplacian calculation of boundary faces. `_::P` denotes variables only needed for the Div operator.

5.4.3 The Transient Operator

Because the transient operator describes the change over time of a transported variable for each cell, no flux or neighbour relations are necessary. Therefore, and in contrast to the spatial operators, the transient operator does not operate on the mesh faces and only modifies diagonal entries of the coefficient matrix. Consequently, no face-based variant for this operator is implemented. The second-order backward Euler scheme needs an additional value; the ϕ from two time steps ago. Writing a generalised transient operator using type unions like `phi_t2::Union{Nothing, Vector{P}}` would require branches to check for `Nothing`. Therefore, instead of creating different implementations based on strategy, different implementations based on the chosen transient scheme are written. The implementations of the transient operator are shown in Listing 5.3.

```

1  function (ddt::Ddt{P,BDF1{P}})(
2    oldVector::Vector{P},
3
4    volume::P,
5    valueDiag::P,
6    valueRHSx::P,
7    valueRHSy::P,
8    valueRHSz::P
9  ) where {P<:AbstractFloat}
10  ret = ddt.scheme(
11    volume, oldVector
12
13  ) * ddt.scale
14  return valueDiag + ret[1],
15    valueRHSx + ret[2][1],
16    valueRHSy + ret[2][2],
17    valueRHSz + ret[2][3]
18  end

```

(a) Transient operator with BDF1

```

1  function (ddt::Ddt{P,BDF2{P}})(
2    oldVector::Vector{P},
3    oldOldVector::Vector{P},
4    volume::P,
5    valueDiag::P,
6    valueRHSx::P,
7    valueRHSy::P,
8    valueRHSz::P
9  ) where {P<:AbstractFloat}
10  ret = ddt.scheme(
11    volume, oldVector,
12    oldOldVector
13  ) * ddt.scale
14  return valueDiag + ret[1],
15    valueRHSx + ret[2][1],
16    valueRHSy + ret[2][2],
17    valueRHSz + ret[2][3]
18  end

```

(b) Transient operator with BDF2

Listing 5.3.: Cell-Based transient operator calculation using backward Euler schemes. Lines are left empty on the left to highlight differences.

Note that the tuple-scalar multiplications in line 13 are normally not allowed by the compiler. A dedicated auxiliary utility function that overloads `Base.*` was implemented and is shown in Listing 5.11: the first argument is simply a scalar multiplication, whereas the second is obtained through a broadcasted multiplication of `a` and all the entries in `b[2]`.

```

1  Base.*(a::P, b::Tuple{P, Vector{P}}) where {P} = a * b[1], a .* b[2]

```

Listing 5.11.: Utility function that enables tuple-scalar multiplication.

5.4.4 Operator Fusing

One of the goals of this work is the fusing of the momentum equation PDE operators. To achieve this, a recursive data structure `DiffEq` is defined in Listing 5.12. The struct is essentially a binary tree wherein the nodes are either a PDE operator or another `DiffEq`. Here, this tree is also referred to as an expression tree.

The operator nodes inside the tree are evaluated by propagating matrix coefficient contributions in a post-order traversal. The evaluation call to `DiffEq` is called with inputs that are needed by the operators, as well as the two contribution values, `ownerContrib` and `neighContrib`. These values are either zero, for the tree root, or intermediate results during the evaluation. Overall, the

operator fusing is designed so the unfused operator calculations can be replaced by the singular function call to DiffEq, as shown representatively on the face-based strategy in Figure 5.4. The left side illustrates the unfused assembly of the spatial operators through multiple iterations over the mesh. On the right, the spatial operators are evaluated together inside the internal face loop.



Figure 5.4.: Assembly of the spatial operators unfused (left) and using operator fusing (right).

To enable bottom-up evaluation of the expression tree, all operators that are evaluated from the same point inside the assembly loop need to implement the same function definition. For instance, in the cell-based strategy, the transient operator is evaluated outside of the internal faces loop while both spatial operators are evaluated inside it. In that case, the spatial operators share the same inputs. For the example in Figure 5.4, wherein the spatial operators of the momentum equation are fused, both would have the same function definition as shown in Listing 5.12. In addition to the operator-specific arguments, the coefficient contributions are also found in the function definition.

Note that only the internal face-based evaluation is shown for the sake of brevity. The cell-based and face-based boundary cases are implemented the same way, mirroring the function definitions of Listing 5.24, Listing 5.7, and Listing 5.3. Across all operators and DiffEq, a total of 16 functions were implemented. Currently, the transient operator is always called on its own. No corresponding DiffEq function call is therefore added.

5.4 Operators

```
1  struct DiffEq{A,B}
2      a::A
3      b::B
4  end
5  @inline function (eq::DiffEq)(
6      faceFlux,
7      gamma,
8      deltaCoeffs,
9      magFaceArea,
10     ownerContrib,
11     neighContrib
12 )
13     ownerContrib, neighContrib = eq.a(
14         faceFlux, gamma, deltaCoeffs, magFaceArea, ownerContrib, neighContrib
15     )
16     ownerContrib, neighContrib = eq.b(
17         faceFlux, gamma, deltaCoeffs, magFaceArea, ownerContrib, neighContrib
18     )
19     return ownerContrib, neighContrib
20 end
```

Listing 5.12.: The abstract DiffEq struct used for operator fusing and the evaluation call for the face-based strategy.

Because the transient operator and the spatial operators need to be handled separately in the cell-based strategy, auxiliary functions for DiffEq are implemented (see Listing 5.13). Similar to the fused operator calculation, both functions are evaluated bottom-up: The existence of the transient operator is propagated upwards by the `hasTransient` function, returning `true` for the transient operator itself, and `false` otherwise. The `splitTempSpat` function propagates a tuple of sub-expressions upward. The first argument of this tuple will be all temporal operators, while the second contains all spatial operators. Necessary for this to work is a third function that overloads the addition operator for generic tuples.

```

1  hasTransient(t::DiffEq) = hasTransient(t.a) || hasTransient(t.b)
2  hasTransient(t::Ddt) = true
3  hasTransient(t::Union{Div,Laplace}) = false
4
5  splitTempSpat(t::DiffEq) = splitTempSpat(t.a) + splitTempSpat(t.b)
6  splitTempSpat(t::Ddt) = t, nothing
7  splitTempSpat(t::Union{Div,Laplace}) = nothing, t
8
9  Base.:+(a::Tuple, b::Tuple) = a[1] + b[1], a[2] + b[2]

```

Listing 5.13.: DiffEq helper functions.

Analogous to the previously detailed operators, the implementation of DiffEq includes a struct, and a functor definition. Additionally, and for the sake of elegance alone, we overload addition of operators and Fused structs to easily construct what essentially represents the PDE's syntax tree. The functor arguments need to match that of the operator functors.

5.5 Standalone Julia

The primary goal of this section is to report details of the implementations created for the evaluation of RQ I.b and, in part, RQ I.a. The standalone Julia implementation is based on an existing prototype¹³ written in C++ and extended where necessary. For instance, the prototype only calculates two-dimensional heat diffusion, i.e., only the Laplacian operator is implemented. Furthermore, some modifications with respect to generality of the parsers were necessary due to the coexistence of multiple OpenFOAM standards and versions. The chronologically next step after case definition and generation is reading it to prepare for simulation. Therefore, some preprocessing strategies are explained first, followed by the implementation variants of the operators.

Because the algorithmic structure of the matrix assembly in standalone Julia does not differ from the one implemented in JuNe, their details are not shown here. The primary difference between the standalone implementation and JuNe is the underlying data layout. This difference also leads to slightly different memory footprints (cmp. Table A1 & Table A2). Reporting their respective implementations in addition to the pseudocodes in Section 4.3 and the operator calculations presented in Section 5.5.2 has little additional benefit with respect to the research questions. Instead, the focus of this section is pre-processing steps and the abstraction of interpolation schemes at runtime.

5.5.1 Preprocessing

Interpolation weights for the Upwind and central difference divSchemes are pre-calculated. The weights are merely used for the evaluation of the operator calculation variants described in Section 4.1. Furthermore, the sparsity pattern of the coefficient matrix can be calculated in

¹³<https://github.com/chihtaw/FVM-CFD-prototype>

column-major order once and re-used in future assembly steps, assuming the mesh does not change. The sparsity pattern depends on the matrix format employed during the solver steps. Analogous to the interpolation schemes, the matrix format might also be generalised by a template parameter. For the sake of simplicity, only the compressed-sparse-row (CSR) matrix format will be used in this work. However, note that templates usage regarding different matrix formats, too, can be reduced through delegation to Julia. Preprocessing the sparsity pattern reduces arithmetic operations inside the assembly, and no subsequent sorting is necessary. The edge colouring used for the batched face-based assembly strategy needs to be calculated only once as well.

Figure 5.5 relates the OpenFOAM mesh to coefficient matrix entries. Each cell and its adjacent faces F_c correspond to $2 * |F_c| + 1$ non-zero coefficient matrix entries: The diagonal entry for the cell itself, and two off-diagonal entries per face. In memory, the coefficient matrix A is stored as a contiguous one-dimensional vector. The entries in the source vector b can be scalar or three-dimensional. For instance, when the transported variable is the velocity, it is three-dimensional, while transported temperature would result in a scalar vector.

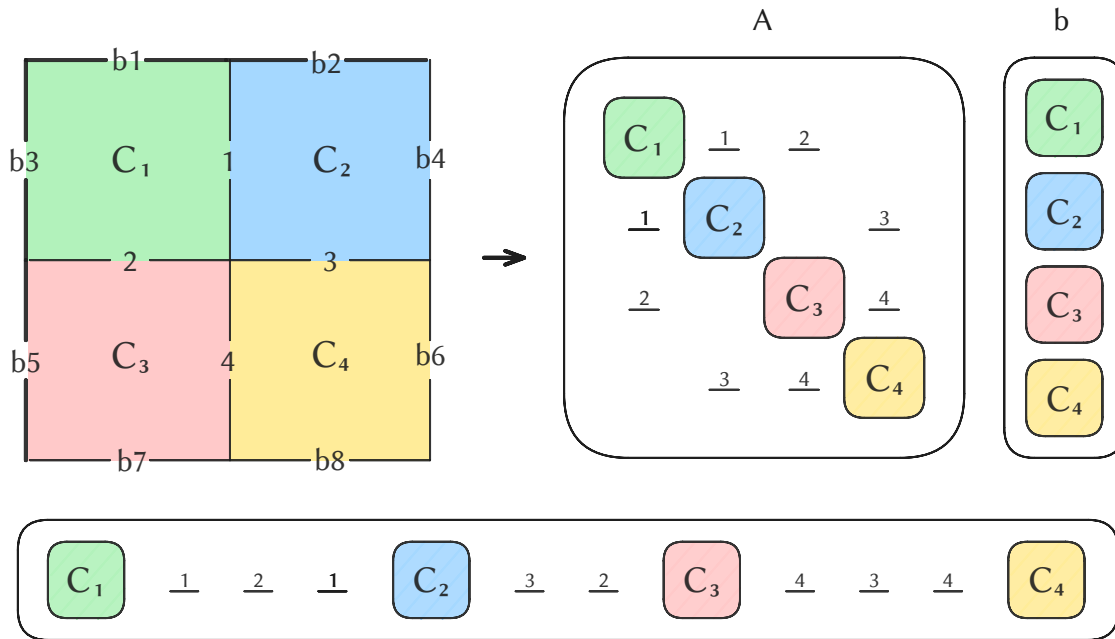


Figure 5.5.: Correspondence of the OpenFOAM mesh (top left) to the coefficient matrix A and RHS (top right), and the storage layout of A (bottom).

For the cell-based strategy only, every cell gets assigned a `rowOffset` value which specifies its exact position inside the matrix. All faces have four indexing-related values. Two for the owning cell and its adjacent neighbour, and two offsets that determine the off-diagonal position with respect to a given cell. Consider the matrix diagrams of Figure 5.5 and assume face `-1-` is owned by C_1 , and C_2 is the neighbour. The owner and neighbour indices in the one-based Julia vector would then be 1 and 5, respectively. Further, the off-diagonal entry of face `-1-` in relation to its owner is 2. Similarly, the off-diagonal in relation to the neighbour cell C_2 would be 4.

When offloading the matrix assembly to the GPU, some vectors need to be flattened. CUDA only allows two-dimensional vectors if all vectors of the second axis statically have the same length. Since, for instance, the number of internal faces of each cell is generally variable this is not allowed. Instead, the vector of corresponding face indices is flattened and used in combination with an auxiliary offset vector that stores the start index for each cell.

5.5.2 Runtime Scheme Abstractions

This section is focused on the operator calculations in the standalone Julia matrix assembly. More specifically, the following details the implementation of Section 4.2.4 which relates to RQ I.b. The transient operator is implemented, but remains unused in the standalone Julia assembly. Therefore, only the fused implementation of Laplacian and div operators are shown in the following. Before presenting the hard-coded, dynamic, and pre-calculated variations, the fused implementation is detailed in Listing 5.6. For the sake of brevity, the examples are only shown representatively for the face-based internal loop.

On the left, the diffusive flux is calculated by multiplying the diffusion coefficient ν with the gDiff input argument. The geometric difference here is a pre-calculated value that describes the weight of the face, i.e., large faces have large gDiffs , and thus contribute more. The Laplacian operator has no implemented scheme here, as stated in Section 4.2.2.

```

1  function (t::Laplace{P})(
2      _::SVector{3,P},
3      _::SVector{3,P},
4      Sf::SVector{3,P},
5      nu::P,
6      gDiff::P,
7      ownCont::P,
8      neiCont::P
9  ) where {P<:AbstractFloat}
10     diffFlux = nu * gDiff
11     ownCont += diffFlux
12     neiCont += diffFlux
13     return ownCont, neiCont
14 end

```

(a) Laplacian Operator

```

1  function (d::Div{P,S})(
2      U_c::SVector{3,P},
3      U_n::SVector{3,P},
4      Sf::SVector{3,P},
5      _::P,
6      _::P,
7      ownCont::P,
8      neiCont::P
9  ) where {P<:AbstractFloat,S}
10     Uf = 0.5(U_c + U_n)
11      $\phi$ ::P = dot(Uf, Sf)
12     g = d.scheme( $\phi$ )
13     ownCont +=  $\phi$  * g
14     neiCont += - $\phi$  * (1 - g)
15     return ownCont, neiCont
16 end

```

(b) Divergence Operator

Listing 5.6.: Fused calculation of the spatial operators.

The implementation of the hard-coded operator calculation calculates both function bodies of Listing 5.6 in succession inside the respective strategy loop (see Listing 5.14). Additionally, the

upwind interpolation scheme is hard-coded into the function in line 6. A second implementation is created for the linear scheme. The only real benefit of hard-coding the scheme into the assembly is that it is self-documenting, i.e., it is clear that the upwind scheme is called in Listing 5.14. The natural caveat of this is that different schemes need to then be realised in different assembly functions which leads to a lot of unnecessary duplicated code. Realistically, this abstraction is best suited for a baseline for the following benchmarks, as this the abstraction with the lowest computational overhead.

```

1  function assemble(...)
2      for iFace in 1:numInternalFaces
3          iOwner = faces.iOwner[iFace]
4          iNeighbor = faces.iNeighbor[iFace]
5          diffFlux = nu[iOwner] * faces.gDiff[iFace]
6          Uf = 0.5(U[iOwner] + U[iNeighbor])
7           $\phi f$  = Uf * faces.Sf[iFace]
8          g = upwind( $\phi f$ )
9          ownCont =  $\phi f$  * g + diffFlux
10         neiCont = - $\phi f$  * (1.0 - g) - diffFlux
...     Update coefficient matrix entries...
15     end
16 end

```

Listing 5.14.: Hard-coded evaluation of the interpolation scheme. The difference between the other abstractions is highlighted.

In contrast to the hard-coded calculation, the scheme is dynamically evaluated in Listing 5.15 by passing the scheme as a function into the assembly function. Otherwise, the implementations are identical. This abstraction, other than the hard-coded variant, needs only one implementation of the given assembly method because the concrete scheme is hidden inside the passed function. This naturally requires an instantiation of this function object in the steps leading up to the assembly. It is nonetheless the closest relative to the fused implementation.

```

1  function assemble(..., scheme::Function, ...)
2      for iFace in 1:numInternalFaces
3          iOwner = faces.iOwner[iFace]
4          iNeighbor = faces.iNeighbor[iFace]
5          diffFlux = nu[iOwner] * faces.gDiff[iFace]
6          Uf = 0.5(U[iOwner] + U[iNeighbor])
7           $\phi f$  = Uf * faces.Sf[iFace]
8          g = scheme( $\phi f$ )
9          ownCont =  $\phi f$  * g + diffFlux
10         neiCont = - $\phi f$  * (1.0 - g) - diffFlux
...     Update coefficient matrix entries...
15     end
16 end

```

Listing 5.15.: Dynamic evaluation of the interpolation scheme. The difference between the other abstractions is highlighted.

Lastly, the weights of the scheme can be computed during preprocessing. This is also the current implementation in NeoN. The weights of the given interpolation scheme are pre-processed and passed into the assembly function as a vector with an element for each internal face. This replaces the previous function call with an array read access (see Listing 5.16:10). Similar to the dynamic variant, the abstraction of the concrete scheme is pulled out of the assembly method which increases dynamism and reduces duplicated code. However, computing and storing the factors in an extra array further increased the memory footprint, thus reducing the arithmetic intensity.

```

1  function assemble(..., weights::Vector{Float64}, ...)
2      for iFace in 1:numInternalFaces
3          iOwner = faces.iOwner[iFace]
4          iNeighbor = faces.iNeighbor[iFace]
5          diff_flux = nu[iOwner] * faces.gDiff[iFace]
6          Uf = 0.5(U[iOwner] + U[iNeighbor])
7           $\phi f$  = Uf * faces.Sf[iFace]
8          g = weights[iFace]
9          ownCont =  $\phi f$  * g + diff_flux
10         neiCont = - $\phi f$  * (1.0 - g) - diff_flux
...     Update coefficient matrix entries...
15     end
16 end

```

Listing 5.16.: Pre-calculated evaluation of the interpolation scheme. The difference between the other abstractions is highlighted.

5.6 JuNe

This section presents the implementation details of the interface between Julia and NeoN, JuNe. First, some general information about inter-language differences and the process of building NeoN with Julia is reported. Thereafter, the subsections conceptually graduate from the creation of a PDE expression in NeoN towards the discretisation during the extracted Julia matrix assembly.

Concerning the interface development, there are a number of potential pitfalls. First, C++ internally stores its data in row-major while Julia does so in column-major. As illustrated in Figure 5.7, a NeoN `Vector<Vec3>` of length four would, trying to preserve two-dimensionality, be a 3x4 Matrix on the Julia side. Further, to access the first sub-vector $\{v_{11}, v_{12}, v_{13}\}$, non-contiguous accesses at multiples of the first dimension would either have to be accepted, or the data would have to be transposed. The cost of the latter approach naturally increases with problem size, and is therefore not a viable option when developing for performance. However, realising that a contiguous vector of also contiguous data, `Vec3` in this case aliasing as `double[3]`, allows us to discard the previous options and instead treat it as a flat one-dimensional vector. The C API can be used to create a thin wrapper around the two-dimensional vector using its flattened size. The key is that the entire chunk of data needs to be contiguous. As a counter example, lists in Python only store references to the contained data, and therefore do not guarantee contiguity. The downside of treating a two-dimensional vector as one-dimensional in Julia is that memory accesses are slightly more complicated. Especially the conversion from zero-based indexing to one-based in combination with 2D to 1D index conversion is prone to errors. In the future, some low-cost vector Wrappers around the passed data that hide these caveats are likely to improve development and reduce the likelihood of one-off indexing mistakes.

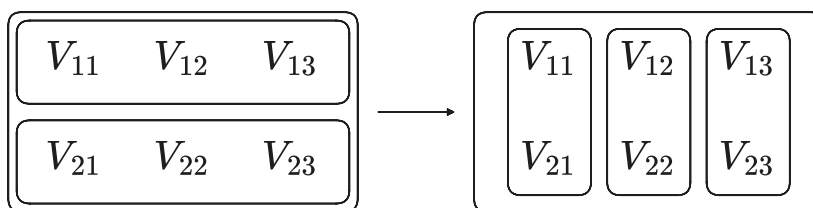


Figure 5.7.: C++ row-major (left) vs. Julia column-major memory layout (right). Note, that the element notation is unchanged to illustrate how the Julia memory layout affects C++-allocated data. Inside Julia code, the indices would be transposed.

5.6.1 Integration into NeoN

The first step of integrating Julia into NeoN is to include it inside NeoN’s CMake build process. Following the pre-existing modular setup, the Julia assembly can be toggled through the corresponding CMake build variable `-DNeoN_WITH_JULIA=ON`. If the variable is set and Julia is installed, Julia is included in the build process, as illustrated in Listing 5.17. The first `execute_process` uses the installed Julia to dynamically obtain its binary directory. Afterwards, the `JULIA_DIR` is used to make `julia.h` publicly available inside the code base of NeoN. For the NeoFOAM build, a second CMake variable `JUNE_MODULE_INIT` is necessary: the path to a `init.jl` file inside the JuNe module. Because of compatibility constraints of LLVM v19.1.7 and CUDA 12.5.1, Julia version 1.11 is used for the implementation.

```

1  if(NeoN_WITH_JULIA)
2      execute_process(
3          COMMAND julialauncher +1.11 -e "print(
4              rsplit(Sys.BINDIR, '/', limit=2)[1]
5          )"
6          OUTPUT_VARIABLE JULIA_DIR
7          OUTPUT_STRIP_TRAILING_WHITESPACE
8      )
9      target_include_directories(NeoN_public_api INTERFACE
10         "${JULIA_DIR}/include/julia"
11     )
12     target_link_libraries(NeoN_public_api INTERFACE
13         "${JULIA_DIR}/lib/libjulia.so"
14     )
15     target_link_options(NeoN_public_api INTERFACE
16         "-Wl,-rpath,${JULIA_DIR}/lib"
17     )
18 endif()

```

Listing 5.17.: CMake integration of Julia into NeoN.

With this CMake integration of Julia, the development of the interface can begin. Listing 5.18 shows the basic structure of a NeoFOAM executable using Julia. First the Julia header is included. The compiler macro `JULIA_DEFINE_FAST_TLS` enables a more efficient implementation of the thread-local storage. Inside the main function, Julia has to be initialised first. Then, the previously discussed CMake build variable is used to initialise the JuNe module. The `fmt` library is used for the string formatting. At the end of the program, the Julia context is safely closed using the `jl_atexit_hook(0)`.

```

1  #include <julia.h>
2  JULIA_DEFINE_FAST_TLS;
3  int main() {
4      jl_init();
5      auto path =
6          fmt::format(fmt::runtime("include(\"{}\)"), JUNE_MODULE_INIT);
7      jl_eval_string(path.c_str());
...
100  jl_atexit_hook(0);
101  }
```

Listing 5.18.: Julia initialisation, loading of JuNe, and teardown.

5.6.2 Data Preparation

NeoN' vector and array classes are implemented as wrapper classes around `Kokkos::View` with additional functionality. One of these is to return a view on the underlying data. A similar function is implemented into the vector and array classes that returns a thin wrapper around the data. The implementation thereof is shown in Listing 5.19. The implementation has to differentiate between where the data is allocated, i.e., on the CPU or on the GPU. GPU device pointers need to be passed as void pointers (see Listing 5.19:2), while CPU host can be passed with the thin wrapper. In NeoN, the abstraction of the execution context and the data location is implemented in the `Executor` class. Aside from the `GPUExecutor` shown in Listing 5.19:1, `SerialExecutor` and `CPUExecutor` exist, denoting CPU-serial and CPU-parallel execution. For the latter, either OpenMP [Dagum and Menon, 1998] or Threads can be chosen.

```

1  if (std::holds_alternative<GPUExecutor>(exec_)) {
2      return jl_box_voidpointer((void*)data());
3  }
4  // Data is allocated on CPU
5  jl_value_t* type;
6  if constexpr (std::is_same_v<VectorValueType, NeoN::Vec3>)
7  {
8      type = jl_apply_array_type((jl_value_t*)jl_float64_type, 1);
9  }
10 else if constexpr (std::is_same_v<VectorValueType, label>)
11 {
12     type = jl_apply_array_type((jl_value_t*)jl_int32_type, 1);
13 }
...                               ... other VectorValueTypes ...
24 jl_array_t* julia_ptr = jl_ptr_to_array_1d(type, data(), size(), 0);
25 return (jl_value_t*)julia_ptr;

```

Listing 5.19.: Wrapping a NeoN Vector<Vec3> to be passed to Julia. `data()` and `size()` return a pointer to the vector elements and its size, respectively.

The `jl_ptr_to_array_1d` function takes four input arguments: The corresponding Julia data type, a pointer to the data, the number of elements, and an integer that specifies whether Julia should own this vector or not. Here, all data is owned by NeoN, i.e., Julia’s garbage collector does not free the memory of, for instance, the coefficient matrix after the assembly function ends.

5.6.2.1 NeoN::Expression to Julia Fused OP

The `NeoN::Expression` object is constructed by overloading arithmetic operators to combine the individual operator objects. Each NeoN operator carries a Julia string representation of itself as a member which is filled with runtime variables such as the interpolation schemes. Any mathematical factor an operator is multiplied with is not a runtime variable, but is also formatted into the string representation. Inside this overloading of arithmetic operators, the operator-specific strings are concatenated and stored at the `Expression` (see Listing 5.20:14). This string representation needs to be valid Julia code because it will be parsed into the operators and `DiffEq` object presented in Section 5.4 (see Listing 5.20:18).

Working with string formatting that needs to be successfully parsed is rather fragile from a software development point of view. A different, more stable, solution would likely be required in a production setting. However, since this method has negligible impact on performance and because further development yields no further improvement with respect to the research questions, the string formatting and parsing is used.

```

1  // in main.cpp
2  auto somePDE = 3 * div(phi, U) - 0.5 * laplace(nu, U);
3
4  // in laplacianOperator.hpp, factor := -0.5
5  juliaEvalString_ = std::format("Laplace({})", factor);
6
7  // in divOperator.hpp, scheme := "upwind", factor := 3.0
8  juliaEvalString_ = std::format(
9      "Div({}{Float64})({}), {}",
10     scheme, factor
11 );
12
13 // in NeoN::Expression, operator-()
14 expr.juliaOP() = lhs.juliaOP() + " + " + rhs.juliaOP();
15
16 // in Julia:
17 pde = eval(Meta.parse(pString))
18 // DiffEq(Div(upwind(), 3.0), Laplace(-0.5))

```

Listing 5.20.: String representation from NeoN::Expression. `juliaOP` is the getter function for `juliaEvalString`.

The next logical step after the conversion and preparation of data is the transfer from NeoN to Julia.

5.6.3 Passing Data between NeoN and JuNe

Passing the thin wrappers around data across language boundaries to Julia is as simple as filling a `j1_value_t*` array, and using it as an argument for `j1_call` (see Listing 5.21). Non-vector variables like the number of internal faces in Listing 5.21:2 need to be boxed, i.e., the value of a primitive type is wrapped in an object to allow referencing. For each primitive data type, a C API boxing function in the form of `j1_box_<dtype>` exists. Strings are handled differently. A `std::string` needs to first be transformed into a C string, i.e., null-terminated character arrays, and subsequently converted into a Julia string (see Listing 5.20:10). The `assemble` function is implemented inside a Julia module. To access it, the module has to be loaded first. Afterwards, the Julia matrix assembly is called. Note that the implementation shown in Listing 5.21 is simplified slightly; the presented implementation would always perform the face-based matrix assembly. Depending on the iteration strategy and the Executor of the underlying NeoN::LinearSystem, a different strategy or its GPU equivalent is chosen.

```

1  jl_value_t* args[nInputs] = {
2      jl_box_int32(nInternalFaces),
3      owners_jptr,
4      ...
5      ...
10     jl_cstr_to_string(fusedOPString.c_str()),
11     ...
23 }
24 jl_module_t* mod = (jl_module_t*)jl_eval_string("JuNe");
25 jl_function_t* func = jl_get_function(mod, "assemble");
26 jl_call(func, args, ninputs);

```

Listing 5.21.: Preparing arguments and calling our Julia module from NeoN.

It is important to note, that interfacing Julia with C++ is not entirely thread-safe¹⁴, and that Julia should only be called from the same thread that originally called `jl_init()`. Nevertheless, calling a Julia function like this is synchronous, i.e., the program only continues when `jl_call` finishes. However, this also holds true if the called function throws an exception. It is therefore useful to follow function calls with `jl_exception_occurred()`, which facilitates error handling and logging, as shown in Listing 5.22. If an exception occurs during the Julia call, it will be stored in global scope. This allows calling Julia again to obtain this error and further process it. Otherwise, it is likely that little to no error output will be shown. In Listing 5.22:35, `NF_ERROR_EXIT` is a macro that forwards an error message to `NeoN::Logging` and exits the program.

```

26 jl_call(func, args, ninputs);
27 if (jl_exception_occurred()) {
28     const char* p = jl_string_ptr(
29         jl_eval_string(
30             "sprintf(showerror, ccall(:jl_exception_occurred, Any, ()))"
31         )
32     );
33     // handle error here. For instance, in the context of NeoN:
34     NF_ERROR_EXIT(std::string(p));
35 }

```

Listing 5.22.: Julia interface exception handling.

5.6.4 Data Reception in Julia

This section details the Julia side of the interface. Listing 5.8 shows the entry points for the face-based strategy, differentiated by CPU and GPU. Listing 5.8b is the exact counterpart to the function call shown in Listing 5.21. While this is the face-based entry point, the same structure

¹⁴<https://docs.julialang.org/en/v1/manual/embedding/#Thread-safety>

is applied for all other entry points. In the following, these code blocks will be referred to as the general entry-point to avoid confusion.

CPU-allocated vectors can simply be accepted as `Vector{T}`. For CUDA device vectors, extra conversions need to be performed, as shown in Listing 5.8c. The GPU version needs to accept a void pointer, which is reinterpreted as a CUDA pointer and finally wrapped into a `CUDA.CuArray`. The `toGPU` function shown in Listing 5.8a requires the vector size and element data type. For both the CPU and GPU case, the data types need to be manually set according to the data types on the C++ side, as Julia cannot automatically detect the data type from simply a pointer. In both cases, the string representation of the PDE is parsed and evaluated into a `DiffEq` (see Listing 5.8b:24 and Listing 5.8c:27). Because the transient operator needs to be handled differently, the PDE operators are split using the auxiliary function defined in Listing 5.13. The CPU and GPU implementations continue in Listing 5.23 and Listing 5.9, respectively.

```

1  function toGPU(
2      p::Ptr{Cvoid}, N, T
3  )
4      ptr = reinterpret(CuPtr{T}, p)
5      return unsafe_wrap(
6          CuArray, ptr, Int64(N);
7          own=false
8      )
9  end

```

(a) Auxiliary function to transform void pointers to CuArrays.

```

1  function assemble(
2      numInteriorFaces::Int32,
3      owners::Vector{Int32},
4      ...
5      ...
6      ...
7      ...
8      ...
9      ...
10     pString::String,
11     ...
12     ...
13     ...
14     ...
15     ...
16     ...
17     ...
18     ...
19     ...
20     ...
21     ...
22     ...
23 )
24     pde = eval(parse(pString))
25     ddt, spat = splitTempSpat(pde)
26     ...
27     ...

```

(b) CPU

```

1  function assemble_gpu(
2      numInteriorFaces::Int32,
3      _owners::Ptr{Cvoid},
4      _neighbour::Ptr{Cvoid},
5      _diagOffs::Ptr{Cvoid},
6      _ownOffs::Ptr{Cvoid},
7      _neiOffs::Ptr{Cvoid},
8      _rowOffs::Ptr{Cvoid},
9      _values::Ptr{Cvoid},
10     pString::String,
11     _faceFlux::Ptr{Cvoid},
12     _gamma::Ptr{Cvoid},
13     ...
14     ...
15     ...
16     ...
17     ...
18     ...
19     ...
20     ...
21     ...
22     ...
23 )
24     owners = toGPU(
25         _owners,
26         numInteriorFaces,
27         Int32
28     )
29     pde = eval(parse(pString))
30     ddt, spat = splitTempSpat(pde)
31     ...
32     ...
33     ...
34     ...
35     ...
36     ...
37     ...
38     ...
39     ...
40     ...
41     ...
42     ...
43     ...
44     ...
45     ...
46     ...
47     ...
48     ...
49     ...
50     ...
51     ...
52     ...
53     ...
54     ...
55     ...
56     ...
57     ...
58     ...
59     ...
60     ...
61     ...
62     ...
63     ...
64     ...
65     ...
66     ...
67     ...
68     ...
69     ...
70     ...
71     ...
72     ...
73     ...
74     ...
75     ...
76     ...
77     ...
78     ...
79     ...
80     ...
81     ...
82     ...
83     ...
84     ...
85     ...
86     ...
87     ...
88     ...
89     ...
90     ...
91     ...
92     ...
93     ...
94     ...
95     ...
96     ...
97     ...
98     ...
99     ...
100    ...

```

(c) GPU

Listing 5.8.: The Julia side of JuNe. Shown is the face-based entry point. For information on `splitTempSpat`, see Section 5.4.4. `Meta.parse` was shortened to `parse` for brevity.

Having successfully passed C++ data across language boundaries and transformed a string representation of the PDE into a Julia struct, the next topic is the matrix assembly implemented into JuNe.

5.6.5 Matrix Assembly Strategies

As outlined in Chapter 4, three assembly strategies in serial, multi-threaded, and on the GPU are implemented, resulting in a total of nine methods. To reiterate, the batched face-based strategy is not implemented into JuNe due to refactoring complexity, but will be considered for future work. Overall, the assembly strategy implementations on the Julia side of the interface are designed to mirror the original one on the NeoN side.

A small selection of Julia packages for the purpose of GPU development exist. On one hand, vendor-specific packages exist, e.g., `CUDA.jl`, `AMDGPU.jl`, `Metal.jl` or `oneAPI.jl`. Out of these options, `CUDA.jl` is by far the most developed. Alternatively, packages like `KernelAbstractions.jl`, like the name might suggest, hide the vendor-specifics behind abstractions. Ultimately, `KernelAbstractions` was chosen because of the intention to extend the interface to other vendors in future releases. Furthermore, the API is slightly more developer-friendly than that of `CUDA.jl`. In terms of performance, small-scale evaluations not detailed here have shown a negligible difference between `CUDA` and `KernelAbstractions`. Note, that while support for other GPU vendors is planned in the future, only NVIDIA GPUs will be used in this work.

As previously said in Section 5.4, passing array slices per ranged indexing is not possible in a GPU context. Additionally, because slice assignments like `foo[1:3] = bar[1:3]` cannot be made atomic, they have to be split into individual assignments, e.g., `foo[1] = bar[1]`; `foo[2] = bar[2]`; `foo[3] = bar[3]`. The following subsections provide implementation details on each matrix assembly strategy. To reduce unnecessary repetition therein, only the serial and GPU implementations are shown here. In general, the process of introducing multi-threading into the assembly strategies involves two steps: Prepending `@batch` to the main assembly loop, and safe-guarding unsafe potential race conditions with `@atomic`. These differences are annotated in the serial implementations through comments.

In Section 4.4, the arithmetic intensities are presented for the internal face processing of the cell-based and the general face-based strategies, and the boundary face processing of the face-based strategy for the sake of conciseness. The same reasoning is applied here: The implemented face-based strategies differ only on a superficial level, i.e., that the faces are differentiated in a merged loop. Reducing redundancy in this section, the loops of the face-based strategy are presented in respective subsections alongside of the cell-based calculations of inner faces. The complete implementations are found in the Github repository¹⁵

Each of the following subsections follows the same structure: The respective processing of a single element is presented. Before that, the scaffolding where this processing is embedded is detailed for the GPU, CPU-serial and CPU-threaded execution contexts.

¹⁵<https://github.com/lupeterm/FVM-MA>

Listing 5.24 shows the processing of a single cell with the cell-based strategy. Inside it, all internal faces of the current cell are processed with the fused spatial operators, in this case the div and Laplacian operators. In lines {10, 19, 23, 30}, conversions from two-dimensional and zero-based to flattened and one-based indexing is performed. Contributions to the diagonal coefficient for the cell are accumulated in the `diagValue` variable. Off-diagonal entries, corresponding to individual internal faces, are written directly into the coefficients vector `vals`. No atomic operations are needed because each cell writes in disjoint coefficient matrix entries.

```

...                (continued from Listing 5.23 or Listing 5.9b)
2  diagValue = 0.0
3  startIdx = cellFacesSegments[celli]
4  numInternalFaces = cellFacesSegments[celli+1] - startIdx
5  for i in 1:numInternalFaces
6      faceIdx = cellFacesValues[startIdx+i] + 1
7      dVal, offDiagValue = spatial(      # Fused: div + Laplacian
8          faceFlux[faceIdx],
9          gamma[faceIdx],
10         deltaCoeffs[faceIdx],
11         magFaceArea[faceIdx],
12         0.0, 0.0,                      # Initial values
13         faceSignV[startIdx+i]
14     )
15     fIdx = matrixColumnIdxV[startIdx+i] * 3 + 1
16     vals[fIdx:fIdx+2] .+= offDiagValue      $
17     diagValue      -= dVal
18 end
19 cellIdx = celli * 3 - 2
20 diagValue, rx, ry, rz = ddt(              # ddt, here with BDF1
21     oldVectors[cellIdx:cellIdx+2],        $
22     volumes[celli],
23     diagValue, 0.0, 0.0, 0.0             # Initial values
24 )
25 diagIdx = (rowOffs[celli] + diagOffs[celli]) * 3 + 1
26 vals[diagIdx:diagIdx+2] .+= diagValue      $
27 RHS[cellIdx:cellIdx+2] += [rx, ry, rz]     $

```

Listing 5.24.: Serial cell-based processing of one cell. `celli` is the given cell index as specified in Listing 5.9b or Listing 5.23. The ranged indexings in lines marked with \$ would have to be split in the GPU implementation.

5.6.5.2 Processing of Internal Faces

Listing 5.10a shows the kernel call to the internal face computation in the face-based strategy. Like the cell-based strategy, the backend is explicitly specified to be CUDA, and the kernel call itself is supplied with a work group size of 64, the appropriate inputs, and the total number of elements to process, `numInteriorFaces`. Afterwards, the kernel call is synchronised to avoid race conditions or premature returns. Note that the synchronisation is not always necessary here. For instance, both the internal faces kernel and the boundary faces kernel presented in the next section can be processed in parallel. Therefore, both kernels can be started, and only need to be synchronised once at the end.

In Listing 5.10b, the GPU scaffolding around the internal face processing presented in Listing 5.26 is illustrated. Analogous to the cell-based strategy, the global index of the internal face is obtained by reading the `@index` macro.

... (continued from Listing 5.8c)	1 @kernel function faceInternal(
31 backend = CUDABackend()	2 @Const(owner),
32 faceInternal(backend, 64)(	3 @Const(neighbour),
33 internalFaceInputs...;	
34 ndrange=numInteriorFaces	19)
35)	20 iFace = @index(Global)
36 synchronize(backend)	... (continued in Listing 5.26)
37 end # function assemble_gpu	47 end # function faceInternal

(a) Kernel call.

(b) GPU scaffolding.

Listing 5.10.: GPU call hierarchy of the face-based kernel for internal faces. In (a), `internalFaceInputs` is a placeholder for all input arguments for the sake of brevity.

Listing 5.25 shows the CPU implementation of the loop around Listing 5.26. Like before, the difference between the serial and multi-threaded implementation is highlighted.

... (continued from Listing 5.8b)
2 @batch for iFace in 1:numInteriorFaces
... (continued in Listing 5.26)
18 end
19 end # function assemble

Listing 5.25.: CPU scaffolding of the face-based strategy processing internal faces. Differences between serial and multi-threaded are highlighted.

Listing 5.26 presents the calculation of a single internal face in a face-based strategy. The implementation is overall the same for serial, multi-threaded and GPU. The differences between serial and parallel implementations are highlighted. Furthermore, like in the cell-based strategy, the lines marked with $\S$ do not compile in the GPU implementation, and need to be split into separate assignments. The lines additionally marked with $\mathbb{P}$ need to be split because ranged

assignments cannot be made atomic. As a reminder, only the spatial operators are evaluated here, because the transient operator is only calculated in the cell-based strategy. The index calculations in lines {20, 22, 31, 33, 35, 37} are necessary to transform zero-based, two-dimensional C++ indices to one-based flattened Julia indices.

In contrast to the cell-based strategy, no face sign needs to be passed into the operators. Instead, the owner and neighbour contributions are both computed by the fused operators and applied to the four coefficient matrix entries. The atomic assignments write into the matrix diagonal entries that correspond to the owner and neighbour cell, while the remaining assignments update the off-diagonal entries related to the face,

```

...           (continued from Listing 5.25 or Listing 5.10b)
20  iOwner = owner[iFace] + 1
21  rowOwnStart = rowOffs[iOwner]
22  iNeighbor = neighbour[iFace] + 1
23  rowNeiStart = rowOffs[iNeighbor]
24  ownContrib, neiContrib = spatial(      # Fused: div + Laplacian
25      faceFlux[iFace],
26      gamma[iFace],
27      deltaCoeffs[iFace],
28      magFaceArea[iFace],
29      0.0, 0.0                          # Initial values
30  )
31  idx = (rowNeiStart + neiOffs[iFace]) * 3 + 1
32  vals[idx:idx+2] .+= ownContrib          $
33  idx = (rowOwnStart + diagOffs[iOwner]) * 3 + 1
34  @atomic vals[idx:idx+2] .-= ownContrib  $,P
35  idx = (rowOwnStart + ownOffs[iFace]) * 3 + 1
36  vals[idx:idx+2] .+= neiContrib          $
37  idx = (rowNeiStart + diagOffs[iNeighbor]) * 3 + 1
38  @atomic vals[idx:idx+2] .-= neiContrib  $,P

```

Listing 5.26.: Face-based processing of one internal face. `iFace` is the internal face index as specified in Listing 5.10b and Listing 5.25. Lines marked with $\$$ need to be split in the GPU implementation because ranged indexing is forbidden. Lines marked with $\mathbb{P}$ need to be split in any parallel context, because of `@atomic`.

5.6.5.3 Processing of Boundary Faces

Generally, the implementations for the processing of boundary faces are very similar to the internal faces in the previous section: Listing 5.11a shows the call of the boundary face kernel in Listing 5.11b, summarising all inputs as `boundaryFaceInputs`, specifying the total number of items to be processed, and synchronising afterwards.

```

... (continued from Listing 5.8c)
31 backend = CUDABackend()
32 faceBoundary(backend, 64)(
33     boundaryFaceInputs...;
34     ndrange=numBoundaryFaces
35 )
36 synchronize(backend)
37 end # function assemble_gpu

```

(a) Kernel call.

```

1 @kernel function faceBoundary(
2     @Const(owner),
3     @Const(diagOffs),
...
19 )
20 bcfaci = @index(Global)
... (continued in Listing 5.28)
47 end # function faceBoundary

```

(b) GPU scaffolding.

Listing 5.11.: GPU call hierarchy of the face-based kernel for boundary faces. In (a), `boundaryFaceInputs` is a placeholder for all input arguments for the sake of brevity.

In Listing 5.27, the loop around Listing 5.28 is shown. It is placed directly after transforming the PDE string into a `DiffEq`. Like before, the `@batch` is highlighted, showing the difference between the serial and multi-threaded implementation.

```

... (continued from Listing 5.8b)
2 @batch for bcfaci in 1:numBoundaryFaces
... (continued in Listing 5.28)
18 end
19 end # function assemble

```

Listing 5.27.: CPU scaffolding of the face-based strategy processing boundary faces. Differences between serial and multi-threaded are highlighted.

Listing 5.28 reports the implementation of processing a single boundary face in the face-based strategy. To reiterate, only the spatial PDE operators `div` and `Laplace` are evaluated in the face-based strategy. Moreover, the lines marked with $\S$ are split in the GPU implementation due to otherwise invalid LLVM IR. $\mathbb{P}$ marks lines that split in multi-threaded or GPU parallelisation because ranged assignments in the form of `foo[start:stop] = bar[start:stop]` cannot be made atomic. This is also the reason why the three values for the RHS are not implemented as a triple or vector: The subsequent assignments are split in two out of three implementations. Consequently, the values would have to be assigned individually. Note that the splitting of lines also applies to the right sides of the assignment in lines 35 and 36.

This implementation shows two additional variables that have not been referenced before: boundary values `bValues` and boundary RHS `bRhs`. In a distributed context, the original mesh is decomposed into subdomains. When processing a subdomain, a former internal face might now be considered a boundary face. The `bValues` and `bRhs` vectors are used in the communication and synchronisation between neighbouring subdomains. Since this work does not focus on distributed processing, these two variables have no semantic effect here. They are nonetheless implemented to be as close to the NeoN implementation as possible.

```

... (continued from Listing 5.27 or Listing 5.11b)
20 start = bcfaci * 3 - 2
21 valueDiag, valueRHSx, valueRHSy, valueRHSz = spatial(
22     refValue[start:start+2], $
23     refGradient[start:start+2], $
24     bfaceFlux[bcfacei],
25     valueFractions[bcfacei],
26     bdeltaCoeffs[bcfacei],
27     bgamma[bcfacei],
28     magFaceArea[numInteriorFaces+bcfacei],
29     0.0, 0.0, 0.0, 0.0
30 )
31 own = bFaceOwners[bcfacei] + 1
32 idx = (rowOffs[own] + diagOffs[own]) * 3 + 1
33 @atomic vals[idx:idx+2] .+= valueDiag $, P
34 bValues[bcfacei*3-2:bcfacei*3] .+= valueDiag $
35 @atomic RHS[own*3-2:own*3] += [valueRHSx, valueRHSy, valueRHSz] $, P
36 bRhs[bcfacei*3-2:bcfacei*3] += [valueRHSx, valueRHSy, valueRHSz] $

```

Listing 5.28.: Face-based processing of one boundary face. `bcfacei` is the boundary face index as specified in Listing 5.11b and Listing 5.27. Lines marked with $\$$ need to be split in the GPU implementation because ranged indexing is forbidden. Lines marked with $\mathbb{P}$ need to be split in any parallel context, because of `@atomic`.

This concludes the implementation details regarding the functionality of JuNe. Because this work takes place in the realm of high-performance computing, the next section presents remarks regarding the performance aspect of matrix assembly in particular and Julia in general.

5.7 Developing Julia for Performance

Julia, being community driven with a rich ecosystem, provides lots of different functionalities and user-written software. Navigating this ecosystem can be challenging. This section aims to provide a brief summary of important details that can impact the performance.

Given that the construction of sparse matrices is a core element of this work, it is reasonable to use the built-in `SparseArrays` module, wherein a sparse Compressed Sparse Column (CSC) matrix data type is provided. However, inserting a value at index (i, j) is slow because the corresponding row and column indices have to be searched for i and j , and because the insertion between existing values means that all elements sorted thereafter have to be moved. To avoid expensively moving millions of elements during assembly, the matrix can be constructed using the sparsity pattern and a coefficient vector filled with zeros. However, updating the coefficients during assembly, even without moving and potential reallocations, still requires lots of searches

for the correct index. If the use case allows it, and it does in the context of this work, it is most efficient to handle the coefficients as a vector and access it through precomputed offset arrays.

Julia is a garbage-collected and just-in-time compiled language. Implemented into Julia is a stop-the-world garbage collector (GC), i.e., all running threads are paused during an active collection cycle [Bezanson et al., 2018; Koolen and Deits, 2019]. Therefore, frequent collections by the GC should be avoided in an HPC context by reducing the amount of dynamic allocations to a minimum. In [Regier et al., 2019, Section 5.2], the authors report individual steps that incrementally improved the performance towards petascale: small arrays were replaced by tuples and scalars to move allocations from the heap to the stack. For arrays where this was not possible, they replaced the built-in dynamically sized vectors with statically sized arrays of the `StaticArrays.jl` package. Lastly, the allocations of larger arrays were moved outside of hot loops. The authors report a 5x speedup by applying these techniques.

The matrix assembly implemented into JuNe has the advantage that NeoN owns the vast majority of data. Therefore, no significant impact of the garbage collection was observed. Conversely, the GC has more impact on the standalone matrix assembly because Julia owns and manages all data.

For instance, consider the benchmark result in Listing 5.29. The run was executed on a structured mesh with 8M cells and took 16 seconds for one assembly. Listing 5.29:2 states that only one run was performed. By default, `BenchmarkTools` performs as many runs as possible in a five-second window, or just a single one. As reported by the built-in benchmarking library `BenchmarkTools`, the function spent almost half of the total processing time in garbage collection. The metrics provided below indicate why: during matrix assembly, almost 14 GiB of data are allocated in nearly 400 million allocations. This very early implementation did not use the separated row, column and value vectors, nor was an operator other than the Laplacian implemented. Regarding the techniques discussed by [Regier et al., 2019], only the larger arrays were allocated outside of the hot loop, i.e., smaller temporary arrays are neither statically sized, nor replaced with tuples or scalars.

```
1 julia> BenchmarkTools.@benchmark cellBasedAssembly(cellBasedInput...)
2   BenchmarkTools.Trial: 1 sample with 1 evaluation per sample.
3   Single result which took 16.058 s (45.37% GC) to evaluate,
4   with a memory estimate of 13.91 GiB, over 393.192.487 allocations.
```

Listing 5.29.: Benchmark of an early, un-optimised version of the standalone Julia cell-based matrix assembly.

In contrast, Listing 5.30 shows the benchmark results for the optimised implementation running on the same mesh. This version performs more calculations and is about 22x faster. Comparing the metrics, the new version performs only a fraction of the allocations, consuming less than 1% of the old implementation. A range of techniques were applied to obtain this speedup, but reducing the allocations enough to not trigger the garbage collection has had the most impact.

```

1 julia> @benchmark FusedCellBasedAssembly($input, $prep..., $pde)
2 BenchmarkTools.Trial: 7 samples with 1 evaluation per sample.
3 Range (min ... max): 728.486 ms ... 732.302 ms | GC (min ... max): 0.00% ... 0.00%
4 Time (median): 729.778 ms | GC (median): 0.00%
5 Time (mean ± v): 730.417 ms ± 1.528 ms | GC (mean ± v): 0.00% ± 0.00%
6 Memory estimate: 7.25 MiB, allocs estimate: 237608.

```

Listing 5.30.: Benchmark of the optimised standalone Julia cell-based matrix assembly

Another factor that is likely implicitly related to the garbage collection is type stability of the program. As stated multiple times throughout this work, Julia uses JIT compilation. Thereby, the type system is not statically enforced as it is in languages like C++ or Rust. However, while this is convenient in terms of development effort, disregarding type stability can negatively impact performance [Pelenitsyn et al., 2021]. When the type of a variable cannot be inferred by the compiler, and is thus labelled as a type union, this uncertainty is propagated everywhere this variable is used. Consequently, the compiler cannot fully optimise the code around it. Furthermore, uncertain types may need to be boxed or otherwise transformed, which may increase heap allocations and, thereby, trigger additional garbage collections. With packages like `Cthulhu.jl`¹⁶, these type uncertainties can be investigated. The primary macro `@descend` traverses the call tree and marks all statements where the compiler is not able to infer a single type.

Naturally, some cases exist where a variable can have multiple types. For instance, in `NeoN` the coefficients are implemented as an abstract `AssemblyValueType`. This type can either be a `NeoN::Vec3` or a `NeoN::scalar`. Applying this concept to Julia, the solution is to reduce ambiguity inside the function body by moving the ambiguity to the function definition. In C++, defining multiple functions with different types is called overloading, while in Julia this is referred to as multiple dispatch. An example for this can be seen in Listing 5.13: different implementations are defined for `DiffEq` and the operators. This also acts as a reminder that the relative impact on performance has to be considered when implementing these optimisation techniques. The splitting of operators is only called once per assembly and the expression tree is rather small. The type union likely has no considerable impact here.

A detail that has more impact on the standalone Julia assembly than on `JuNe` is the usage of vectors. Specifically, variables like the velocity `U` in Listing 5.15 are collections of many small vectors. `StaticArrays.jl`¹⁷ is a package that provides basic functionality around small, statically sized arrays. Changing the data type from `Vector{Vector{Float64}}` to `Vector{SVector{3, Float64}}` may further improve the performance. `SVector{3, Float64}` describes a vector of size $(N,3)$ that is both immutable and not re-sizeable. The mutable, but still statically sized counterpart is `MVector`. This technique is only used in the standalone Julia assembly because all two-dimensional `NeoN` vectors are treated as flat one-dimensional arrays on the Julia side. Furthermore, a construction of a vector of `SVectors` would likely either copy the data, or transfer ownership to Julia.

¹⁶<https://github.com/JuliaDebug/Cthulhu.jl>¹⁷<https://github.com/JuliaArrays/StaticArrays.jl>

Using the built-in Threads module is a simple way of introducing multi-threaded parallelism into already existing code. However, for loops that already have rather short processing times per iteration, this module may introduce performance degradation due to scheduling overhead. Figure 5.12 illustrates a small comparison of the Threads and Polyester¹⁸ modules. JuNe is executed representatively with the face-based strategy on a mesh with ~ 17.5 M cells. The graphs synergistically show better performance using Polyester.@batch over Threads.@threads. Being more suitable for the type of parallelised iterations in this work, the Polyester task scheduling includes lower overhead than the general-purpose scheduler used by Threads.@threads.

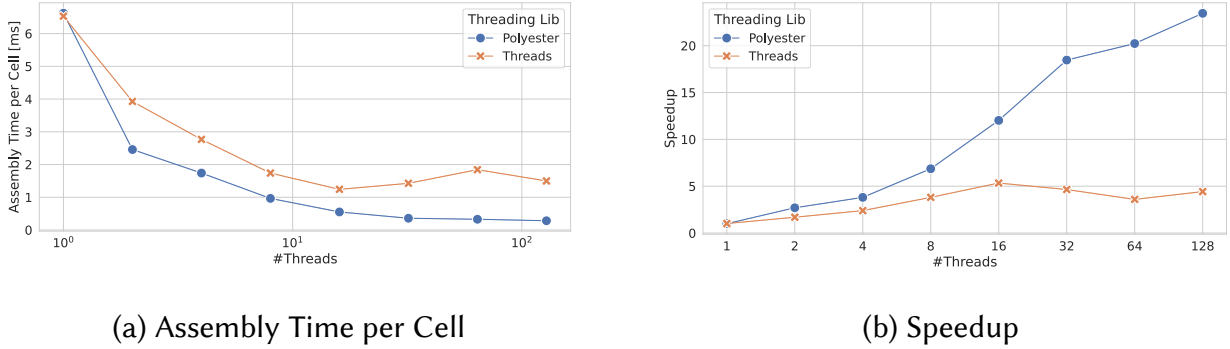


Figure 5.12.: Comparison of the built-in Threads and the Polyester module. The LDC case with 260^3 (~ 17.5 M) cells was used for this illustration.

Another noteworthy detail is the mapping of Julia threads to hardware threads. Allocating a cluster node with 128 hardware threads and starting a Julia REPL with 128 Julia threads does not create a 1:1 mapping. Without interference, some hardware threads may have no Julia thread, while a noticeable amount gets mapped to multiple Julia threads, which can lead to contention and performance degradation. The module ThreadPinning¹⁹ can be used to pin Julia threads to the available hardware using `pinthreads(:cores)`. When performing benchmarks on a HPC cluster, it might be necessary to perform the pinning based on the NUMA affinity mask with `:affinitymask` instead of the physical cores.

5.8 Summary

In the preceding sections, details about OpenFOAM cases, their definition and generation for the purpose of benchmarking are given. The OpenFOAM benchmark runner (OBR) is used to bulk-generate an array of cases with increasing sizes (see Section 5.2). Divergence schemes and time integration schemes are discussed in Section 5.3. No schemes for the Laplacian operator are detailed, as only the uncorrected scheme is implemented implicitly into the operator without an explicit scheme implementation. Section 5.4 presents implementation details for the momentum equation operators themselves. Additionally included therein is the specification for the proposed operator fusing approach: A recursive data structure `DiffEq` is defined to facilitate recursive evaluation of operators and accumulated coefficients at the root (see Section 5.4.4). Some preprocessing strategies are presented in context of the standalone Julia assembly, followed by the runtime scheme abstractions regarding RQ I.b in Section 5.5.2. Thereafter, the internals of

¹⁸<https://github.com/JuliaSIMD/Polyester.jl>

¹⁹<https://github.com/carstenbauer/ThreadPinning.jl>

JuNe are discussed in subsections dedicated to the general integration into NeoN (Section 5.6.1), preparing the data transfer and passing to Julia over the common C API (Section 5.6.2 & Section 5.6.3), how the data is received on the Julia side (Section 5.6.4), and, finally, the matrix assembly strategies implemented into JuNe (Section 5.6.5). Lastly, a set of remarks regarding the development of high-performance Julia code is compiled in Section 5.7.

6 Evaluation

In this chapter, the evaluations conducted in order to answer the research questions defined in Chapter 1 are presented. First, the used hardware setup, as well as the selected metrics and benchmarking cases are reported, followed by the experimental structure.

6.1 General Setup

All benchmarks are performed on the COMA HPC cluster of TU Munich Campus Heilbronn. Two nodes are primarily used for benchmarks and evaluations. The hardware specifications can be seen in Table 6.1.

Table 6.1.: Hardware specification of the COMA cluster nodes.

Node	gpu-nvidia-h100	gpu-nvidia-h200
System Memory	2 TB	2.3 TB
CPU	Intel Xeon Gold 6548N	AMD EPYC 9555 Turing
Cores	32	64
Threads	64	128
L2 Cache Size	2 MB / core	1 MB / core
L3 Cache Size	60 MB	256 MB
Peak Bandwidth	332.8 GB/s	576.0 GB/s
GPU	NVIDIA H100 NVL	NVIDIA H200 NVL
Memory	94 GB	141 GB
Peak Bandwidth	3.9 TB/s	4.8 TB/s

The configuration of the software used in this work follows: Julia 1.11, LLVM/Clang 19.1.7, CUDA 12.5.1, Cmake 3.31.6, OpenFOAM v2412, Open MPI 5.0.7. The versions of secondary dependencies of NeoN and JuNe can be found in the respective repositories.

6.1.1 Metrics

This section presents metrics to measure and discuss performance throughout this work.

6.1.1.1 Speedup

The speedup metric shown in Eq. (30) generally describes the runtime improvement compared to a baseline implementation T_{ref} . Usually, this baseline is the best serial implementation $T(1)$, but other baselines can be used as well.

$$S_N = \frac{T_{\text{ref}}}{T(N)} \quad (30)$$

6.1 General Setup

The ideal case for a parallel implementation running with N threads would be a speedup of N . Oftentimes this is not simple to achieve because of additional communication and memory access overhead caused by additional workers. Speedup values of greater than N are considered *superlinear*. The general consensus for the origin of superlinear speedup is that more memory throughput is accessible in a parallel context [Axtmann and Rist, 2016; Ristov et al., 2016]. In a recent OpenFOAM combustion study, superlinear speedups of 340% are attributed to better utilisation of especially the L3 cache. Modifying the case setup to optimise the elements per parallel worker has also a significant impact [Galeazzo et al., 2024].

6.1.1.2 Parallel Efficiency

The parallel efficiency relates the achieved speedup to the number of processing elements as

$$E_N = \frac{S(N)}{N} \quad (31)$$

The efficiency is given as a percentage between 0 and 100. However, considering a superlinear speedup, the efficiency can exceed 100% as well. Analogous to the speedup, the ideal efficiency would be 100%, but is oftentimes lower due to additional communication overhead.

6.1.1.3 Matrix Assembly Throughput

Oftentimes, processing times are used as the primary metric to measure workload per time. In this work, the primary used metric is the matrix assembly throughput, or just assembly throughput for short. The main advantage of using the throughput over plain assembly times is that the size of the assembled matrix is already included. The number of non-zero coefficients (NNZ) is multiplied with the size of the coefficients in number of bytes. The standalone Julia implementation writes scalar coefficients into the matrix. Conversely, and primarily to mirror the assembly of NeoN, the assembly implemented in JuNe writes three values per coefficient. The result is divided by the assembly time T .

$$\frac{\text{NNZ} * \text{sizeof}(\text{ValueType})}{T} \quad (32)$$

In this work, the assembly throughput is reported in GB/s. The throughput can be used as the basis for calculating the speedup and efficiency metrics, but also allows deeper insight considering the hardware limits given in Table 6.1.

6.1.1.4 Compilation Time

In this thesis, compilation time is considered to be the required time to perform a fresh build of NeoN, i.e., the build directory is deleted between repeated compilations. Builds with and without Julia are performed under the same conditions.

$$\Delta T = T_{\text{JuNe}} - T_{\text{NeoN}} \quad (33)$$

Their difference as defined by Eq. (33) quantifies the overhead introduced by including Julia in the build process.

6.1.1.5 Coefficient of Equivalence

The speedup metric above quantifies the assembly time improvement when adding additional workers in a CPU-parallel context. The coefficient of equivalence (COE) expresses how many CPU cores are required to provide the same throughput as one GPU [Bnà et al., 2025].

In Eq. (34), the number of matrix assembly operations per assembly time is defined as $FVOPS = N_{\text{cells}} * \frac{1}{T}$. FVOPS is therefore a cell-throughput metric and not a count of floating-point operations. The COE is calculated as

$$COE = \frac{FVOPS_{\text{GPU}}/n\text{GPUs}}{FVOPS_{\text{CPU}}/n\text{CPUs}}. \quad (34)$$

As only single-GPU matrix assembly is implemented in this work, the equation simplifies to

$$COE = \frac{FVOPS_{\text{GPU}}}{FVOPS_{\text{CPU}}/n\text{CPUs}}. \quad (35)$$

Because this metric is only used to allow comparison of distributed IcoFoam and GPU-parallel NeoN or JuNe, the nCPUs variable refers to the number of CPU cores.

6.1.2 Test Cases

The research questions presented in Section 1.1 are evaluated using two different test cases: The Lid-Driven Cavity (LDC) [Burggraf, 1966] and DrivAer cases [Heft et al., 2012; Lesnik et al., 2026]. Their use across the benchmarks is summarised in Table 6.2. The following sections explain the cases in detail.

6.1.2.1 Lid-Driven Cavity

The LDC case defines a cubic cavity where one wall is continuously moved in one direction, inducing vorticular movement in the contained fluid. Therein included is usually a primary vortex in and around the centre, but smaller secondary vortices can occur near corners when the dimensionless Reynolds Number is increased (see Figure 6.1). It is a common benchmark and validation case because it is easy to scale and experimental data for validation exists: For this reason, the LDC is frequently used in the “LRZ Gitlab CI” pipeline of the NeoFOAM project²⁰ to validate correctness. However, it has some drawbacks: Being a very structured mesh leads to well banded coefficient matrices, and most of the computational cost is in the linear solver for the Poisson equation which is not representative for industrial simulation cases.

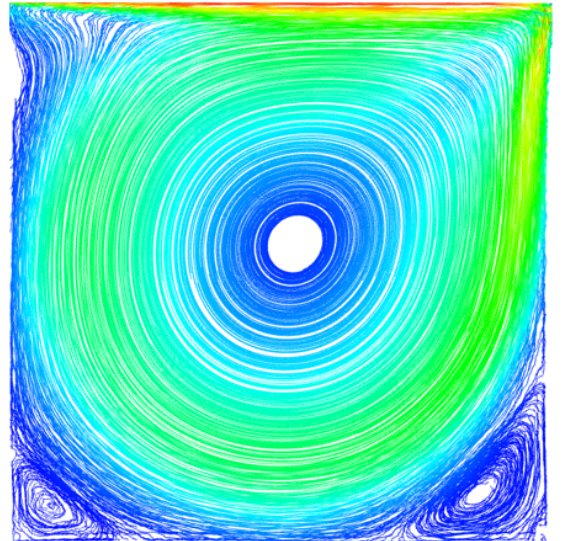


Figure 6.1.: Flow field of a two-dimensional LDC at moderate Reynolds number ($Re = 1000$) [Sume-saraayi and Ihmsen, 2021].

²⁰<https://github.com/exasim-project/NeoFOAM>

To prepare for evaluations, an array of LDC cases is created following the descriptions of Section 5.2. The mesh sizes first increase in steps of $(i \cdot 10)^3$ until 200^3 , and $(i \cdot 20)^3$ thereafter up to 400^3 , i.e., 64 million cells total.

All LDC benchmarks were run with the same fvSchemes configuration shown in Listing 6.1: The ddtScheme Euler refers to the first-order backward Euler transient scheme. upwind is selected for the divScheme and the laplacianScheme is uncorrected.

```
1 ddtSchemes {
2     ddt(U) Euler;
3 }
4 divSchemes {
5     div(phi,U) Gauss upwind;
6 }
7 laplacianSchemes {
8     laplacian(nu,U) Gauss linear uncorrected;
9 }
```

Listing 6.1.: fvSchemes configuration for the LDC benchmarks.

6.1.2.2 DrivAer

The DrivAer case, as illustrated in Figure 6.2, defines the body of a realistic car [Heft et al., 2012]. It is an established benchmarking case for the AutoCFD²¹ workshop and the OpenFOAM HPC Challenge [Lesnik et al., 2026].

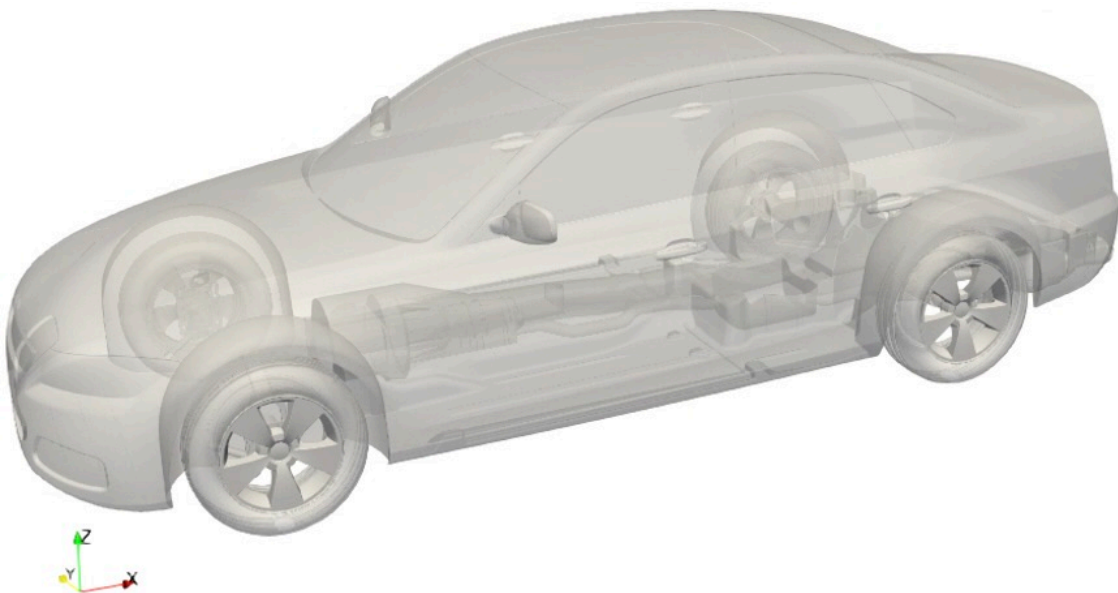


Figure 6.2.: The DrivAer case. Figure taken from OpenFOAM²²

²¹<https://autocfd.org>

²²<https://develop.openfoam.com/committees/hpc/-/tree/develop/incompressible/simpleFoam/occDrivAerStaticMesh>

The default configuration of the DrivAer case is more complex than the LDC. Not all schemes and boundary conditions implemented into NeoN are implemented into JuNe. To navigate around these missing functionalities, the fvSchemes and boundary conditions are modified to mirror those used for the LDC in Listing 6.1. This entails that any simulations performed on this case configuration will not be stable enough for productive simulations. The correctness of JuNe is validated by comparing the assembled matrix with the one assembled by NeoN, which uses the same case configuration. Therefore, this approach is sufficient for the sake of evaluating matrix assembly, operator fusing, and the comparison with NeoN. The increased complexity of the computational grid compared to the LDC also implies that the bandwidth of the assembled coefficient matrix is greater which impacts cache utilisation.

6.1.3 IcoFoam

As described in Chapter 4, four different matrix assembly strategies, variations of operator calculations, the fused operators, and the extracted matrix assembly implemented in JuNe are evaluated. Additionally, further context is added to the results by also evaluating OpenFOAM's incompressible flow solver IcoFoam. The IcoFoam source code is modified to only measure the assembly time, exiting the program before entering the PISO loop, as seen in Listing 6.2.

```

1  int main(int argc, char *argv[]){
2      MPI_Barrier(MPI_COMM_WORLD);
3      Foam::clockTime t;
4      fvVectorMatrix UEqn(
5          fvm::ddt(U)
6          + fvm::div(phi, U)
7          - fvm::laplacian(nu, U)
8      );
9      MPI_Barrier(MPI_COMM_WORLD);
10     Info<< "Momentum " << t.elapsedTime() << std::endl;
11     return 0;
12 }
```

Listing 6.2.: IcoFoam evaluation setup.

Note that the momentum equation in Listing 6.2 also shows the time derivative term `ddt(U)`. Since evaluating the `ddt` operator requires only cell values, a face-based implementation of it would introduce unnecessary overhead and overcomplicate the algorithm. Therefore, it is not included in benchmarks that highlight strategy differences. Instead, it is included in the final interface benchmarks, where the performance differences of NeoN matrix assembly with and without Julia are compared. Including the transient operator in those calculations will be especially interesting because the operator fusing needs to also be able to handle a mixture of spatial and temporal operators including their various schemes.

6.1.4 Evaluations Structure

Table 6.2 details which cases are used by the respective evaluations. Furthermore, the mapping to the research questions introduced in Section 1.1 is given in the third column. While no matrix assembly benchmarks are conducted for the compilation times, they are nonetheless included in this table for the documentation of related research questions. Note that Table 6.2 does not contain RQ III.a because it is primarily addressed in Chapter 7.

Table 6.2.: Research questions and test cases addressed by each benchmark.

Section	Benchmark	Primary RQ(s)	LDC	DrivAer
6.2.1	Standalone Julia Operator Variants	RQ I.a, RQ I.b	✓	–
6.2.2 - 6.2.4	Standalone Julia Assembly Strategies	RQ II.a	✓	–
6.3	NeoN vs. JuNe	RQ II.a, RQ III.b	✓	–
6.3.3	Integration Benchmark	RQ I.a, RQ II.a, RQ III.b	✓	✓
6.4	JuNe Roofline Model	RQ II.b	–	–
6.5	Compilation Times	RQ III.b, RQ III.c	–	–

6.2 Standalone Julia

This section presents the experimental results of the matrix assembly implemented in standalone Julia. The contents of this section are two-fold: First, the differences between operator calculation variants are highlighted in Section 6.2.1. Second, the matrix assembly strategies are evaluated on the LDC in a serial, multi-threaded, and GPU implementation in Sections 6.2.2-6.2.4.

6.2.1 Variations of Operator Calculations

The benchmarks regarding operator calculations are conducted on all lid-driven cavity cases on the H200 node of the COMA cluster. As described in Section 4.2.4, four variations of interpolation scheme abstractions are considered: *pre-calculating* scheme weights, evaluating the schemes *dynamically* via passed function pointer, *statically* calling the scheme inside the assembly loop, and the *operator fusing* approach presented in this work. All calculate the spatial operators of the momentum equation, $\text{div}(\phi, U)$ and $\text{laplacian}(\nu, U)$. The benchmarks further differentiate between the upwind and linear div schemes. In the following, the fused variant only calculates the upwind scheme. Because it is computationally slightly more intensive than the central differencing scheme, the resulting values serve as a representative upper bound and are thereby useful for the comparison to the other variants.

All these individual benchmark runs are executed with the CPU-serial (Figure 6.3), multi-threaded (Figure 6.4), and GPU (Figure 6.5) implementations. Therein, the boxes illustrate the range of assembly throughputs over all mesh sizes. The performance differences of the calculation variants provides insights towards RQ I.b, while the results of the operator fusing strategy in relation to the other variants works towards the answer of RQ I.a.

In the serial implementation illustrated in Figure 6.3, the cell-based strategy throughputs are considerably lower than the face-based strategies, but also show less deviation from the median. Across the three strategies, the pre-calculated variant shows slightly lower values. The differences between the dynamic and the static scheme calculation are negligible. The fused scheme calculation shows comparable values with the other variants. In the cell-based strategy, the fused variant shows the lowest median throughput, though the entire range encompasses the ranges of the other variants. In the face-based strategies, the fused variant shows significantly higher throughputs with a median between approximately 3.3 GB/s and just above 4 GB/s. The batched strategy is not included in Figure 6.3 because it is not designed for serial processing.

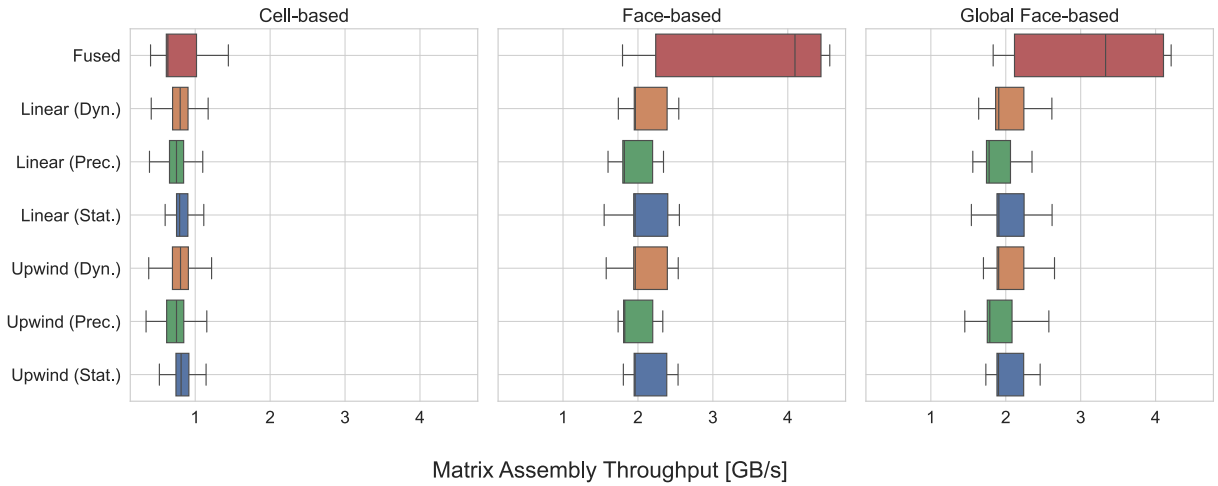


Figure 6.3.: Serial matrix assembly throughput with different operator calculation variants over all mesh sizes of the lid-driven cavity case.

Figure 6.4 illustrates the multi-threaded results for the scheme variations benchmark. Overall, all calculation variants are close to a common median inside the respective strategy: The batched and cell-based strategies have an average median of approximately 5 GB/s. Slightly greater and closer to 10 GB/s are the median values of the face-based and global face-based strategies. The fused calculations show negligible differences to the other variants for the cell-based and face-based strategies, while the throughput values of the batched and global face-based strategies are slightly smaller.

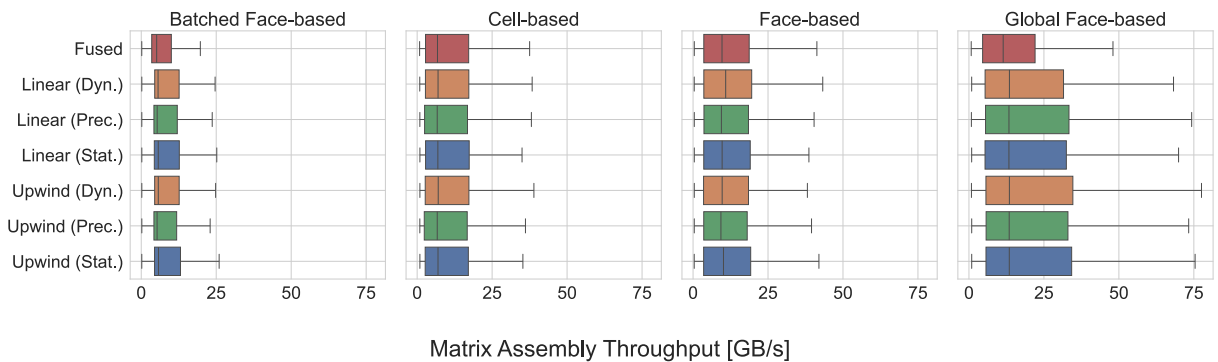


Figure 6.4.: Multi-threaded matrix assembly throughput with different operator calculation variants. Illustrated here are the throughputs for 128 threads over all mesh sizes.

The GPU results of Figure 6.5 show different relative rankings between the assembly strategies. Here, the batched strategy has by far the highest results with median values between 400-450 GB/s. In contrast, the other strategies show median throughput results around or lower than 200 GB/s. Like in the multi-threaded results, all calculation variants are roughly clustered together. Especially the face-based and global face-based strategies show insignificant differences between the variants. For the batched strategy, the lower bound seems equal, while the variants primarily differ in the upper end of the assembly throughput range. Again, the fused calculations show similar median values and ranges to the other variants.

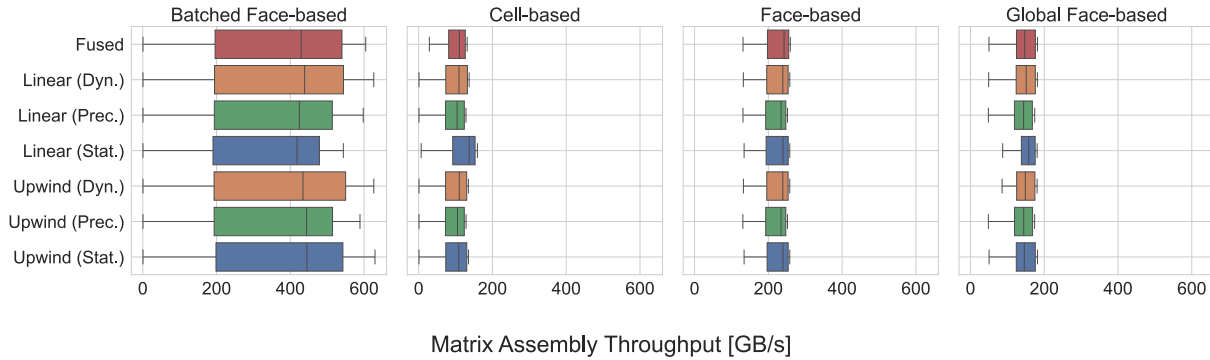


Figure 6.5.: GPU matrix assembly throughput with different operator calculation variants over all LDC mesh sizes.

6.2.2 Serial Matrix Assembly Strategies

Figure 6.6 shows the assembly throughput of the standalone Julia serial matrix assembly strategies for different grid sizes on Intel (left) and AMD (right) CPUs. In contrast to Section 6.2.1, only the upwind interpolation scheme is used here. The batched strategy is not included, as it is an inherently parallel approach. For the Intel CPU, the face-based strategies show considerably higher throughput results than the cell-based strategy while following a similar curve. Both throughputs increase quickly until respective peak values of around 3.5 GB/s before decreasing to a throughput of approximately 2.7 GB/s. The higher throughput values can be explained by considering the memory footprint of the LDC for smaller mesh sizes in combination with the cache sizes of the CPU. The total memory consumption for the mesh size at the peak throughput (50^3) is approximately 39 MB and can be fully loaded into the L3 cache of the Intel CPU. The LDC with 60^3 (216k) cell consumes a total of roughly 67 MB, thereby exceeding the L3 cache size by 7 MB which is still sufficient to store most of the vectors in the cache and achieve a slight improvement in assembly throughput over the 50^3 (125k) LDC. The throughput values then decrease for larger meshes, as only some vectors or parts thereof can be stored in L3, avoiding expensive memory reads. The cell-based strategy displays a similar phenomenon, albeit with a much smaller peak: until the cell count of 50^3 , the throughput very slowly increases to 1.3 GB/s and then decreases again to roughly 1 GB/s, where it then stays for growing meshes.

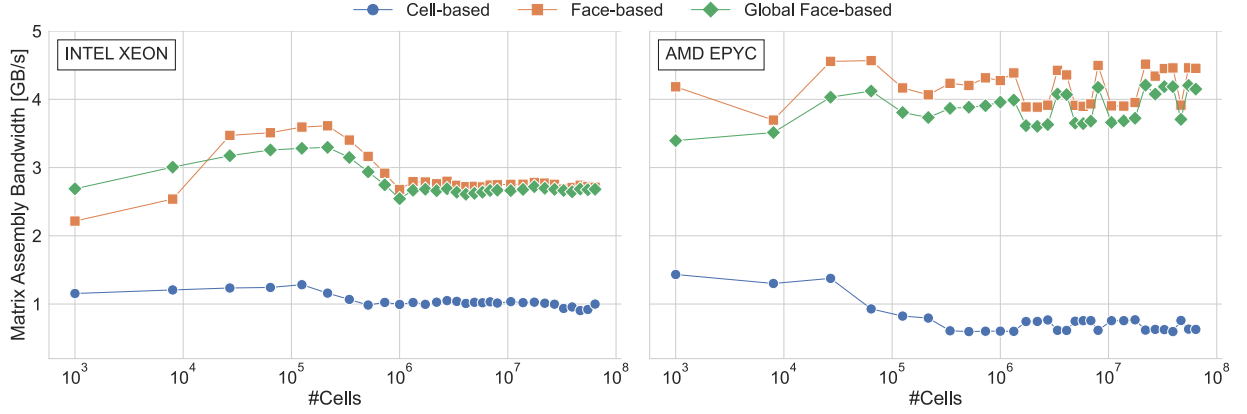


Figure 6.6.: Matrix assembly throughput of the serial matrix assembly strategies implemented in standalone Julia on the Intel XEON CPU.

In contrast to the Intel CPU results, the results on the AMD CPU on the right show overall higher throughputs for both face-based strategies. The cell-based strategy shows lower results, performing similarly for small meshes but then decreasing to approximately 0.75 GB/s. Another noticeable difference between the CPUs are the fluctuations on the right. The relatively stable throughput values face-based strategies are interrupted by sudden drops in performance. For the cell-based strategy, the performance suddenly increases for the same meshes. This phenomenon does not occur in the results on the left. This may be related to some caching behaviour: the L3 cache of the AMD EPYC CPU fits 256 MB, i.e., the cases smaller than or equal to 90^3 (729k) cells fit completely (see Table A1). It is possible that for certain mesh sizes, the vectors have inefficient sizes, e.g., due to the 400k cell increase from 110^3 to 120^3 , all vectors increase in size so that the combination of previously L3-fitting vectors do not fit anymore. Alternatively, this phenomenon can entirely be algorithmic in nature, given that the cell-based throughput values increase for the mesh sizes where the face-based values decrease. Regardless, the face-based strategies show significantly higher throughputs in the serial implementation with final throughput values of approx. 2.7 GB/s for the Intel CPU and values ranging between 3.7-4.5 GB/s for the AMD CPU.

6.2.3 Multi-Threaded Matrix Assembly Strategies

Figure 6.7 shows the multi-threaded matrix assembly throughputs on the AMD EPYC CPU running with 128 threads for different mesh sizes. Directly noticeable in Figure 6.7 is the high throughput of the multi-threaded cell-based strategies, which are in contrast to serial results. Until a cell count of 60^3 (216k), the throughput grows rapidly. Between 10^5 – 10^6 , the throughputs fluctuate between values of 130 GB/s and the global peak of 176 GB/s. This can be explained by the memory consumption of the cell-based strategy and the size of the LLC of the AMD CPU. In Appendix A, Table A1 shows the memory consumption of each LDC mesh. The mesh with 90^3 (729k) cells consumes approximately 225 MB and thereby completely fits inside the 256MB L3 cache (see Table 6.1). For larger meshes, the values rapidly drop again to average values between 18 GB/s and 55 GB/s and reach 21 GB/s for the largest LDC. In contrast, the face-based and global face-based strategies show considerably lower throughputs for meshes smaller than 130^3 (~ 2.2M) cells, but slightly higher results for most of the larger mesh sizes. The global face-based strategy reaches its peak throughput of 60 GB/s at 150^3 (~ 3.4M) cells, while the regular

face-based strategy peaks at approximately 48 GB/s for the second-largest LDC. The batched face-based strategy shows the overall lowest throughputs. The peak throughput is reached at 130^3 cells with a value of 24 GB/s. All face-based strategies seem to show increased fluctuations for cell counts greater than 1M.

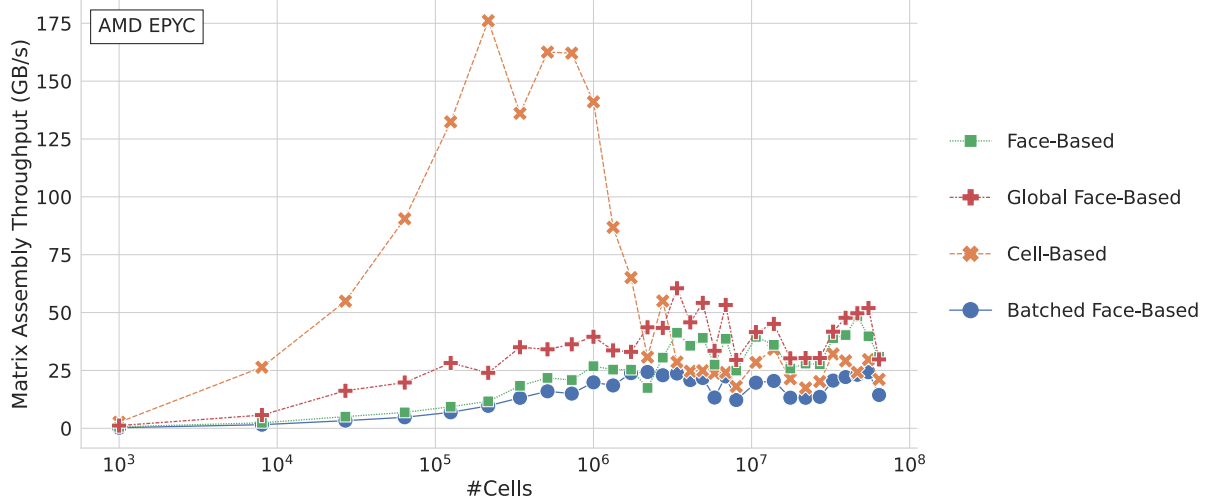


Figure 6.7.: Matrix assembly throughputs of the CPU-parallel implementation with 128 threads on AMD EPYC.

The Intel results in Figure 6.8 show an overall similar trend, but with overall lower throughputs. Like in Figure 6.7, the cell-based strategy displays a large throughput peak, reaching 75 GB/s at 60^3 cells. According to Table A1, the memory consumption of the mesh with 60^3 cells only slightly exceeds the L3 cache size of 60 MB. Considering that the performance drop of the cell-based strategy coincides with the mesh size that exceeds the L3 cache size, this might be a possible reason for the peak. Additionally, the cell-based strategy performs memory reads more locally compared to the face-based strategy. For meshes larger than one million cells, the throughput decreases to values mostly ranging between 10 GB/s and 19 GB/s. The face-based and global face-based strategies stay close together for most of the larger meshes, with throughputs around 10 GB/s to 23 GB/s for cell counts greater than four million. The batched strategy is the slowest strategy on the Intel CPU as well, displaying an average range of 5-10 GB/s.

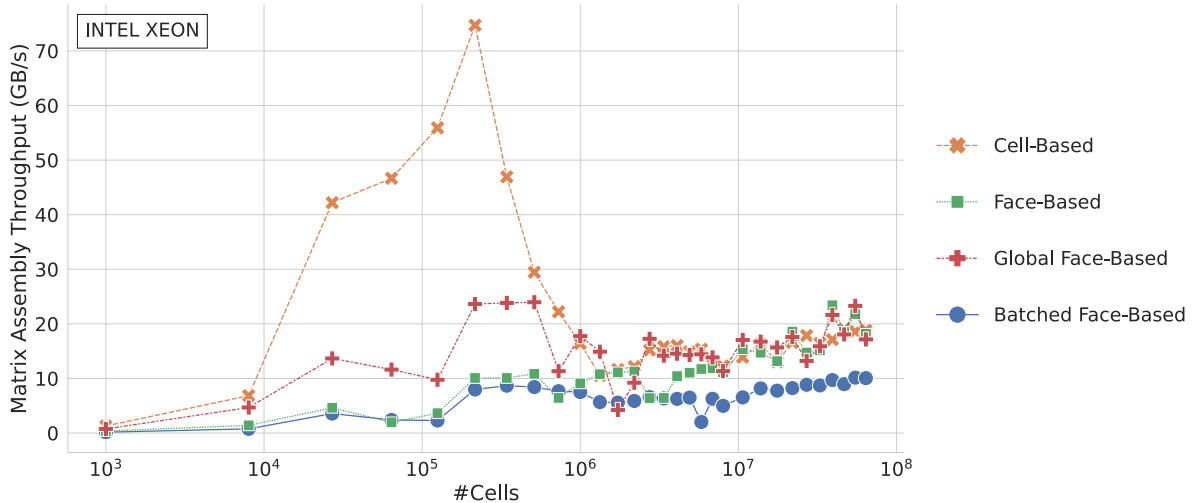


Figure 6.8.: Matrix assembly throughputs of the CPU-parallel implementation with 128 threads on Intel XEON.

Comparing Figure 6.7 and Figure 6.8, the results on the AMD CPU shows higher overall throughputs in nearly every data point. The peaks of the cell-based strategy running on the AMD CPU have more than twice the throughput of the Intel results. Similar things show when comparing the face-based strategies. The primary similarity between the graphs are the overall trends: a considerable peak, followed by rapid decline for the cell-based strategy, increased fluctuation for larger meshes for the face-based strategies, and the relative order of the strategies is the same as well.

Figure 6.9 shows the parallel speedup for the mesh with 216k (60^3) cells, as this mesh corresponds to the highest peak of the cell-based strategy in Figure 6.7. The scaling of the cell-based strategy is closely aligned to the ideal speedup for both CPUs. For thread counts smaller than 32, the AMD CPU decreases in parallel efficiency, but then rapidly increases to a slightly superlinear speedup. The curves of the face-based strategies are overall similar, showing an initial slowdown for two threads, but then slowly increasing. The peak speedup of the face-based strategies is reached by the global strategy with nearly 8x processing 216k cells with 128 threads. The other face-based strategies seem to converge to a speedup of 3-4 for 128 threads.

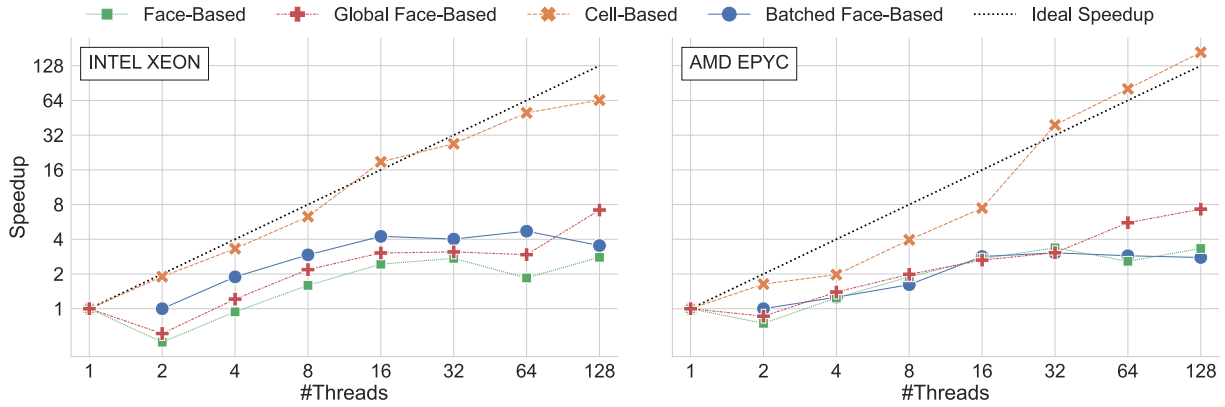


Figure 6.9.: Parallel speedup of the CPU-parallel matrix assembly for the LDC with 216k cells across increasing thread counts. Two threads are considered the baseline for the batched strategy.

Overall, the cell-based strategy shows the highest throughputs and speedups for the mesh sizes around the observed peak. However, its scaling results have to be interpreted with care because it also has the lowest serial throughput of all strategies. The large relative speedup therefore partly results from a weaker baseline and does not directly translate to the same advantage for larger meshes. Considering the absolute throughputs, the cell-based strategy is most suitable for smaller meshes, while the regular face-based strategies show better results for the majority of larger meshes.

6.2.4 GPU Matrix Assembly Strategies

The GPU scaling benchmark is performed using the fused variants of all four matrix assembly strategies on the LDC cases. Figure 6.10 shows the resulting matrix assembly throughputs for mesh sizes ranging from 1k to 64M cells.

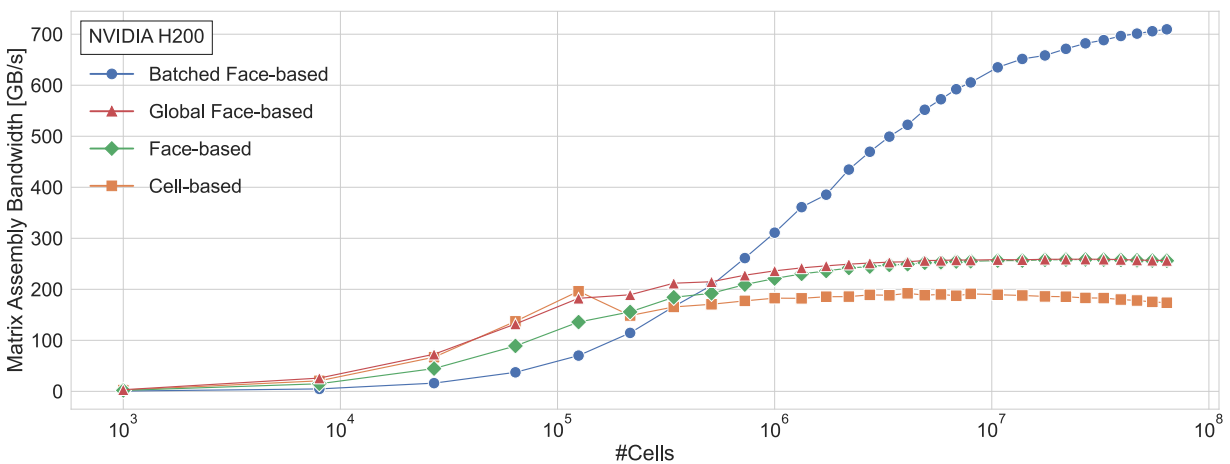


Figure 6.10.: Matrix assembly throughputs of the fused GPU assembly strategies across increasing LDC mesh sizes running on the NVIDIA H200 GPU.

For small meshes, all strategies show low throughputs because the amount of available work is insufficient to utilise the GPU. At mesh sizes of 100k cells or greater, the throughput of the strategies start to separate more clearly. While most strategies reach some plateau under 300 GB/s, the batched strategy throughput keeps increasing. The batched face-based strategy shows the overall highest throughputs, peaking at ~64M cells with a value of 700 GB/s. Overall, this strategy continues to scale rather smoothly for all mesh sizes. The peak throughput of 700 GB/s is close to the upper limit for matrix assembly throughput of a single H200 GPU due to the memory limitations of the device: Running the benchmark with a 125M (500^3) cell LDC results in an error stating that at least one of the vectors could not be allocated. Further, no other meshes with more than 64M cells were tested. The face-based strategy shows the second-highest throughputs with peak values of approximately 250 GB/s. The global face-based strategy follows an almost identical curve, also converging to a peak throughput of approximately 250 GB/s. Both strategies still require atomic updates to shared matrix entries, which also limits their throughput compared to the batched strategy. The cell-based strategy shows the overall lowest throughputs of the four strategies. The majority of throughput values range between 150 GB/s and 200 GB/s for mesh sizes starting at 10^6 cells. While the write accesses of the cell-based strategy are performed non-atomically in disjoint matrix rows, the read memory accesses are more scattered due to the internal face loop (see Section 4.3.1).

Figure 6.11 illustrates the throughput difference between the H200 and H100 GPUs for the standalone matrix assembly strategies. Given that the difference never drops below zero, the H200 shows consistently higher throughputs than the H100. Especially pronounced is the difference of the batched face-based strategy for large meshes: The strategy reaches peak throughput differences of almost 100 GB/s for the cell count range of $280^3 - 320^3$ (~ 23M - 33M). The cell-based and face-based strategies also show some peaks, usually jumping from differences below 20 GB/s to peak values between approximately 35-55 GB/s. These peaks only occur for mesh sizes greater or equal than 10^6 . The differences between the GPUs for smaller meshes never exceed 20 GB/s. The global face-based strategy shows the overall lowest differences with peak values of approximately 30 GB/s.

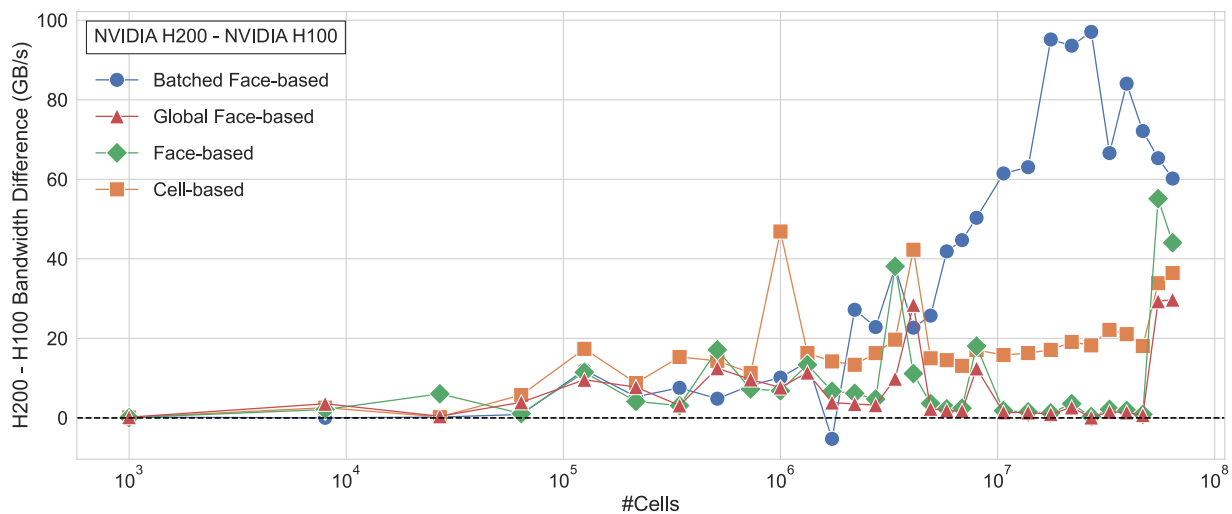


Figure 6.11.: Matrix assembly throughput differences of the fused GPU assembly strategies across increasing LDC mesh sizes.

Overall, the GPU results contrast with the multi-threaded CPU results, where batching introduces additional overhead and performs worse than the regular face-based strategies. For the largest GPU cases, the batched strategy achieves more than twice the throughput of the face-based strategy and roughly five times that of the cell-based strategy. Thus, the most suitable matrix assembly strategy depends strongly on the target architecture.

6.2.5 Summary

Regarding RQ I.a, the throughputs of the fused operator implementation are comparable to the other variants for most implementations and therefore a viable solution from a performance aspect. Discussions regarding non-quantitative aspects are deferred to Chapter 7. Considering RQ I.b, the differences between operator calculation variants, the pre-calculated variant shows slightly lower, but overall similar, throughput values over all implementations. The difference between the static and the dynamic scheme evaluation is negligible across all implementations and strategies. This is likely caused by the JIT compilation of Julia: After the first execution, the matrix assembly including the scheme calculation is compiled in a way that all but removes the overhead of a function call. The calculation differences are most pronounced in the serial implementation. However, considering the small throughputs of smaller than 3 GB/s and the large range of mesh sizes, these differences may be attributed to noise. It is possible that the operator evaluations inside the matrix assembly are simply not computationally expensive enough to highlight significant differences between the variants.

Considering the differences between assembly strategies, the face-based and global face-based strategies show significantly higher results than the cell-based strategy in the serial implementation. Conversely, the cell-based strategy performs better in a multi-threaded context for mesh sizes below 130^3 (~ 2.2M) cells, while the throughputs of all strategies are in the same range of 12-55 GB/s for larger meshes. The cell-based strategy is the only one achieving speedups near the ideal values, though the lowest serial throughput values have to be taken into consideration. The GPU implementation shows yet another relative ranking of the strategies: the batched face-based strategy shows the highest throughput values for meshes greater than 70^3 (343k) cells with a peak of slightly above 700 GB/s while the other strategies show consistent values in the range of 150-270 GB/s. To conclude this section and provide a partial answer to RQ II.a, the best-performing assembly strategy seems to depend on the computational context and mesh size.

6.3 Evaluation of JuNe

This section discusses the benchmark results for the evaluation of JuNe, the Julia matrix assembly interfaced from NeoN. The following subsections detail results for the CPU and GPU implementation, comparing both to NeoN. Thereafter, the integration benchmark performed on the largest LDC and the DrivAer case is presented. Note that the batched face-based strategy is not included in the evaluations of JuNe because it was not implemented due to the required extensive refactoring efforts on the NeoN code base.

The time-to-first-execution (TTFX) is a noteworthy aspect of the implemented interface and Julia development in general. Because of Julia's JIT compilation, the first execution takes more time than subsequent ones, especially when compiling a GPU kernel. However, on the larger

scope of FVM simulations, this first compilation adds a small constant overhead in relation to the total simulation time. Therefore, and to focus purely on the matrix assembly, TTFX is not considered here.

6.3.1 CPU Matrix Assembly Strategies

Here, the CPU results are discussed. Figure 6.12 shows the matrix assembly throughputs of the serial implementation on the Intel (left) and AMD (right) CPUs. Overall, the results are structured similarly between the CPUs, with the primary difference being that the throughputs on AMD EPYC are considerably higher. Another noticeable difference is that the throughputs on the AMD CPU are overall rather steady while Intel results are rather unstable for smaller meshes and only settle after a small jump at around 10^6 cells. The results for both CPUs in Figure 6.12 show a similar peak between 10^4 and 10^5 cells for the fused cell-based strategy of NeoN. For both vendors, the fused cell-based strategy implemented in NeoN shows the highest throughputs, with peak values 2.71 GB/s for the Intel and 6.56 GB/s for the AMD CPU. Directly below is its non-fused counterpart of the cell-based strategy. The fused face-based strategy shows comparably average results with peak values at around 1.2 GB/s. The rest of the strategies, i.e., the non-fused face-based strategies of NeoN and all JuNe matrix assembly strategies are very close in terms of throughput. For smaller meshes until 10^5 , the face-based and global face-based NeoN implementations show much better results, but the difference is negligible for larger meshes. The strategies cluster together at peak throughputs of around 1 GB/s.

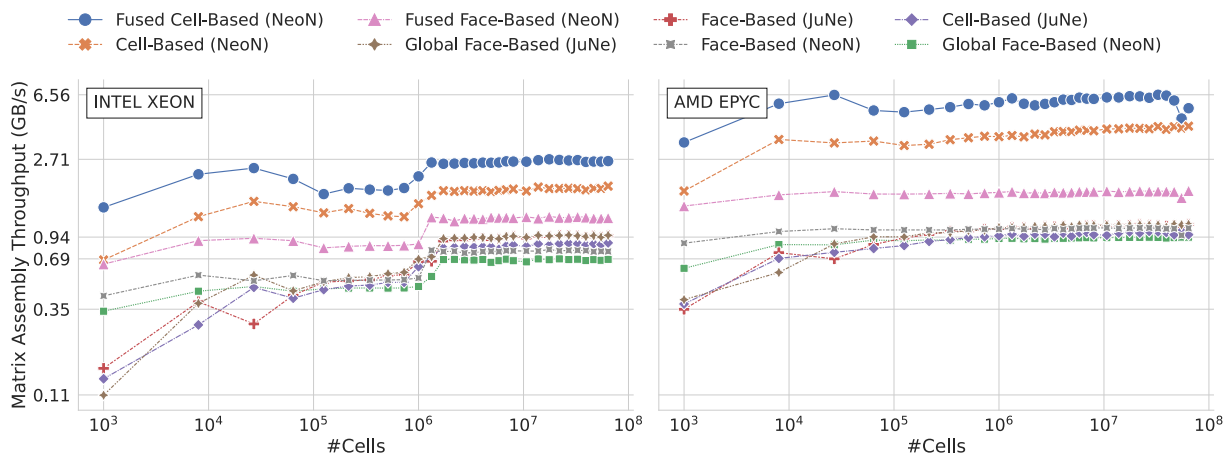


Figure 6.12.: Serial assembly throughput of all strategies implemented in JuNe and NeoN.

Figure 6.13 shows the matrix assembly throughputs for the multi-threaded CPU implementations running with 128 threads on AMD EPYC. Like the serial implementation results, the fused cell-based strategy of NeoN shows the overall highest results. Especially pronounced is a spike for mesh sizes between $5 \cdot 10^4$ and $2 \cdot 10^6$ (30^3 - 120^3) cells. A similar spike can be found in Figure 6.7 and Figure 6.8 for the cell-based strategy of the standalone Julia implementation. Here however, no such spike can be seen for the cell-based strategy of JuNe. These spikes may be related to some caching behaviour, i.e., pieces of the data can be stored in the cache reducing the time needed to fetch it. This behaviour can be seen in the fused cell-based, the fused face-

6.3 Evaluation of JuNe

based and, less pronounced, in the face-based strategy of NeoN. Furthermore, the spikes seem to be less prominent running on the Intel CPU.

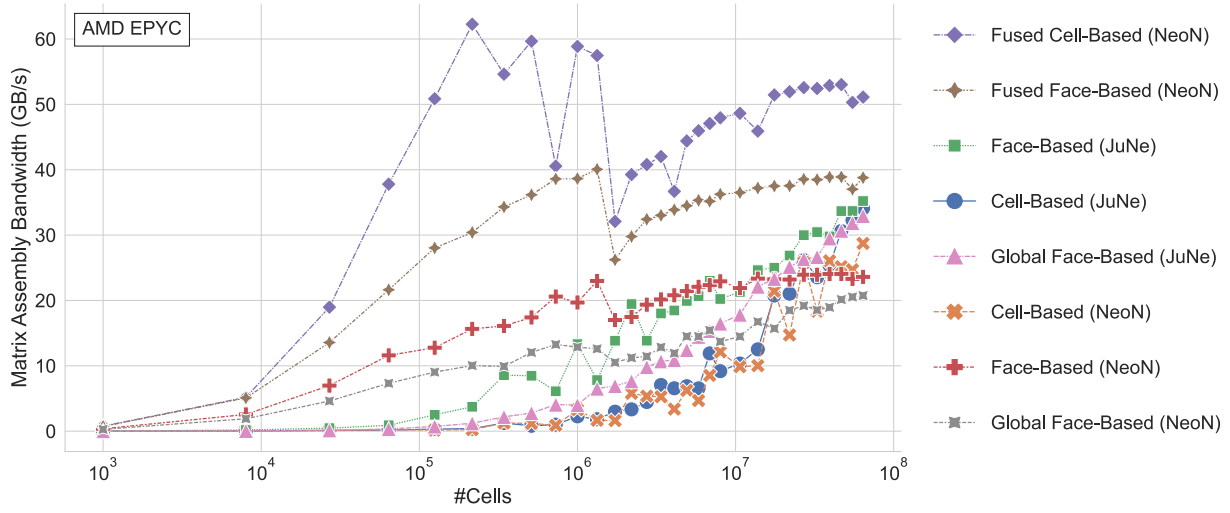


Figure 6.13.: Bandwidth of all assembly strategies using the CPU-parallel implementation with 128 threads on AMD EPYC.

Figure 6.14 illustrates the assembly throughput difference between both CPUs. For mesh sizes below one million cells, the graphs look similar to the AMD-only plot, albeit with smaller throughput values. The peak values of NeoN's fused cell-based, fused face-based and face-based strategies indicate that the AMD CPU reaches approximately twice the assembly throughput for smaller meshes. The order of strategies, from highest throughput to lowest, is still preserved. For meshes with more than a million cells, the AMD still yields higher throughputs, though the Intel results follow similar trends. This is indicated by most strategies staying in the range of 0-17 GB/s without significant growth or decrease.

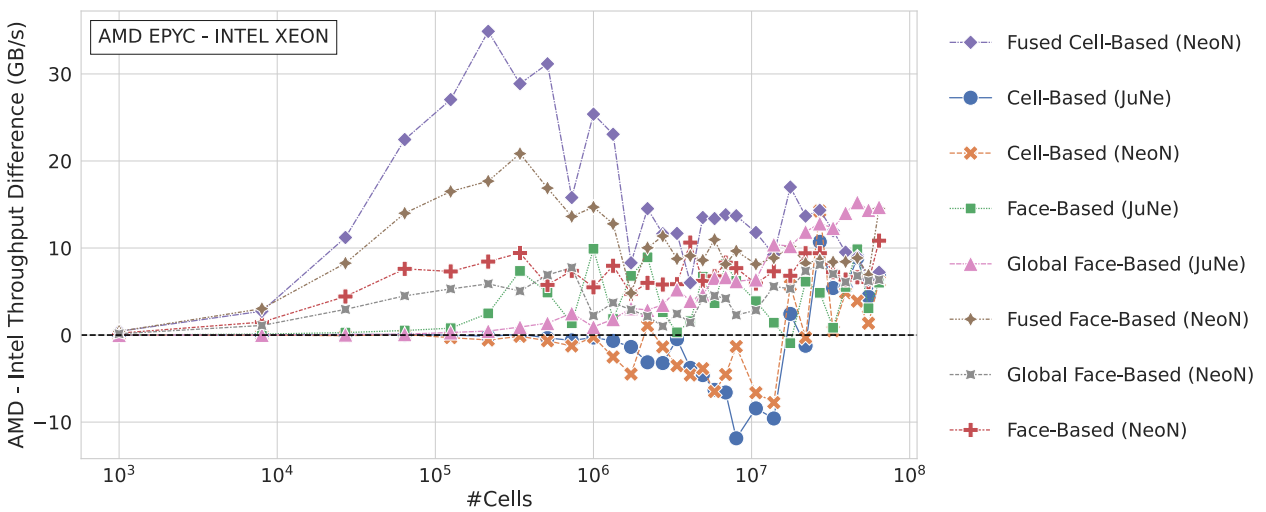


Figure 6.14.: Matrix assembly throughput difference between the Intel and AMD CPUs for 128 threads.

Figure 6.15 offers a different view on the multi-threaded results by showing the speedup with respect to the assembly throughput. Through a similar reasoning to the one applied to Figure 6.9, the speedup plot for the interfaced assembly strategies is calculated for the LDC with 110^3 cells: this particular mesh size seems interesting because it is in the peak throughputs for the top three strategies, and because the throughput significantly decreases at 120^3 ($\sim 1.7M$) cells.

All JuNe matrix assembly strategies display superlinear speedup for thread counts smaller than 32. Especially pronounced is the speedup of the JuNe's cell-based strategy running with 2-16 threads on the AMD CPU. Like the speedup of the standalone Julia in Figure 6.9, this super-linear speedup has to be examined carefully, as the cell-based strategy has some of the lowest throughputs in both the serial and multi-threaded implementation. All strategies either stagnate or drastically lose efficiency when running with 128 threads. The cell-based strategies of JuNe and NeoN show this rapid decline in speedup. However, the fused cell-based strategy of NeoN seems to have more parallel efficiency, as the speedup only stagnates instead of dropping.

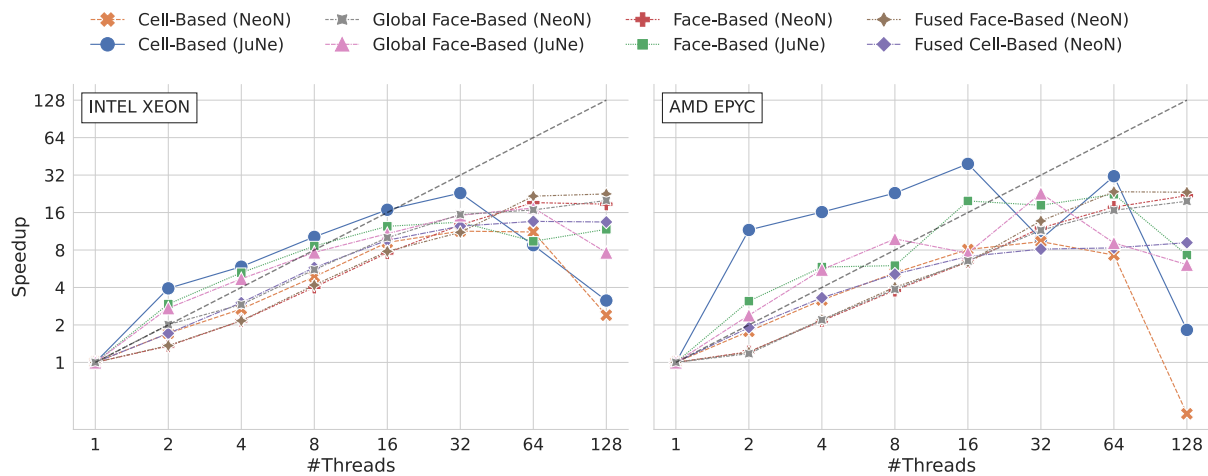


Figure 6.15.: Parallel speedup with respect to matrix assembly throughput on the LDC with 110^3 ($\sim 1.3M$) cells. The dotted diagonal line denotes ideal speedup.

Overall, the CPU benchmarks show that the fused cell-based strategy of NeoN achieves the highest serial and multi-threaded matrix assembly throughput on both CPUs. Although strategies running on AMD EPYC consistently reach higher throughputs than on the Intel CPU the relative ranking and general trends remain similar across both CPUs. Considering the throughput and scaling plots of Figure 6.13 and Figure 6.15, JuNe matrix assembly portrays better scaling than NeoN for smaller thread counts, while NeoN adapts better to thread counts above 32. However, Figure 6.16 illustrates the throughputs of all strategies with 16 threads. While the fused cell-based strategy is still the best-performing, the throughputs of JuNe's cell-based strategy have increased significantly in comparison to 128 threads. Comparing the performance of 16 threads and 128 threads, the results show that the fused face-based strategy performs better on a higher thread count. This is supported by the illustrations in Appendix B and widened to include all of NeoN's fused strategies. In conclusion, the fused cell-based strategy of NeoN shows the overall best throughput results for serial and multi-threaded matrix assembly.

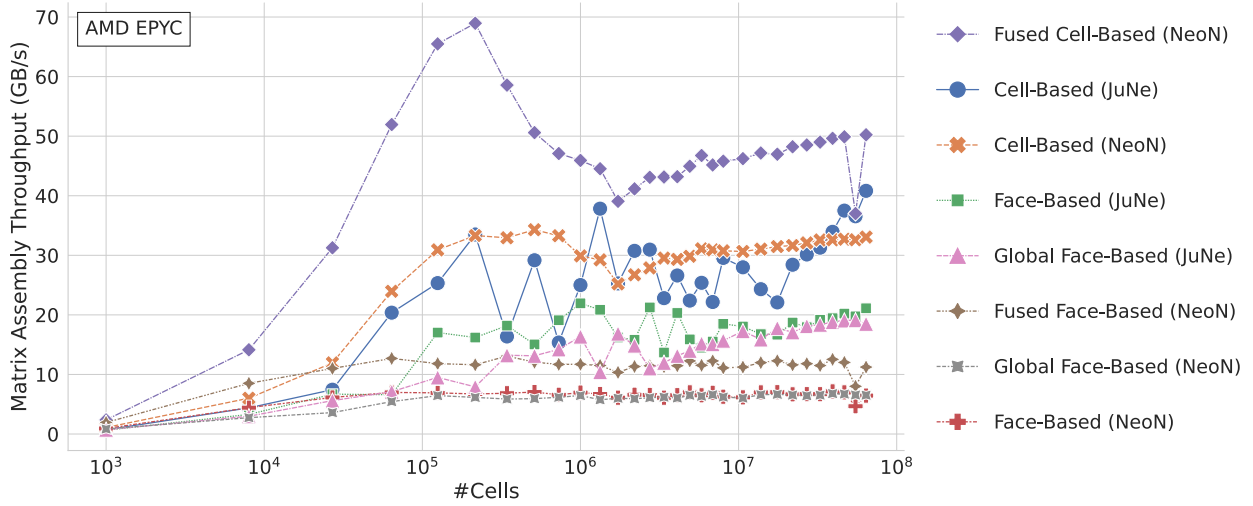


Figure 6.16.: Bandwidth of all assembly strategies using the CPU-parallel implementation with 16 threads on AMD EPYC.

6.3.2 GPU Matrix Assembly Strategies

Figure 6.17 illustrates the achieved throughput during matrix assembly using the H200 GPU. Overall, the global face-based and face-based strategies of NeoN show the lowest throughput, with most values ranging between 50-80 GB/s. As expected, the difference in implementation between the face-based and global face-based strategy has negligible effect. Both strategies show slight drops, especially at mesh sizes 140^3 , 320^3 , and 360^3 . Cell-based is the fastest non-fused strategy of NeoN. While the overall shape is similar to the non-fused face-based strategies, the throughput values are higher with peak values of 130 GB/s. The fused face-based strategy of NeoN shows a speedup of 2x compared to the non-fused face-based strategies, which is the expected result from fusing two operators and thus performing only one pass over the mesh. However, the small fluctuations observed for certain mesh sizes in the non-fused face-based results also occur more pronounced in the fused face-based strategy results. The throughput is mostly higher than the cell-based strategy, but decreases considerably for the largest three meshes to similar values of approximately 130 GB/s.

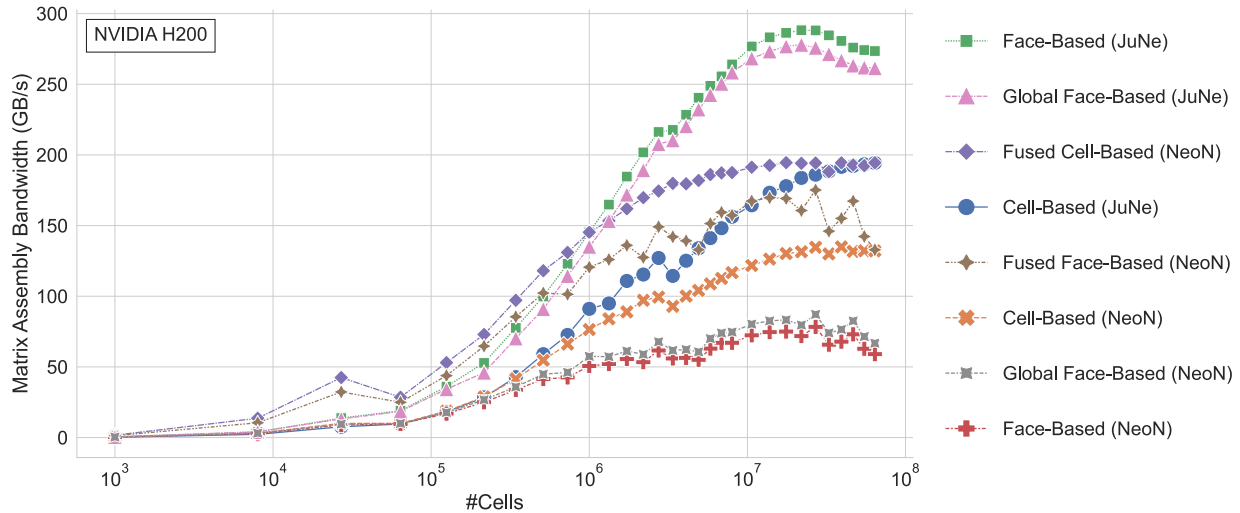


Figure 6.17.: NeoN vs. JuNe: Matrix assembly throughput for all GPU assembly strategies.

For smaller meshes, the cell-based strategy of JuNe shows similar values to the non-fused strategies of NeoN. At around 10^6 cells, the values increase to throughputs similar to the fused face-based strategy of NeoN. Finally, the performance does not decrease as drastically as the fused face-based strategy, and stops at just under 200 GB/s.

The fused cell-based strategy also has its peak throughput at just under 200 GB/s. For mesh sizes until 10^6 , the throughputs are the highest by a small amount. The fused cell-based strategy shows a peak throughput of about 70 GB/s higher than the non-fused counterpart, resulting in a speedup of approximately 1.5x. Overall, fusing the face-based strategy seems to be more effective than fusing the cell-based one. Both cell-based implementations in NeoN show considerably greater throughputs than the face-based strategies.

The overall highest throughputs show the face-based and global face-based strategies implemented into JuNe. Both face-based strategies show peak throughput values between 275-290 GB/s and seem to converge in the range of 260-275 GB/s.

It is reasonable that the fused cell-based strategy of NeoN and the cell-based strategy of JuNe produce similar results, considering the functionality is close to identical and both implementations employ operator fusion. This makes the difference in throughput between the fused face-based strategies even more noticeable. Following the assumption that the fused implementations of similar strategies would show similar results, it is curious why both face-based strategies of JuNe report throughputs that are approximately 1.5x that of the fused face-based strategy of NeoN. Further investigation into possible reasons for this 1.5x difference is out of scope for this work. However, it shall be considered for future research.

Figure 6.18 illustrates the throughput differences between the GPUs in place of the H100 GPU results. The hardware differences are overall rather small compared to the peak values of the H200. The values exceed 10 GB/s only once for JuNe's cell-based strategy on the second-largest LDC. NeoN's non-fused face-based strategy shows the overall smallest differences between the GPUs, showing values of ± 2 GB/s. This range is only left for the largest meshes, where the H100 performs better by approximately 7 GB/s. While some noticeable peaks are shown for smaller

meshes, the respective mesh sizes are simply too small to fully saturate the GPUs. Overall however, the hardware differences do not seem to follow a specific pattern.

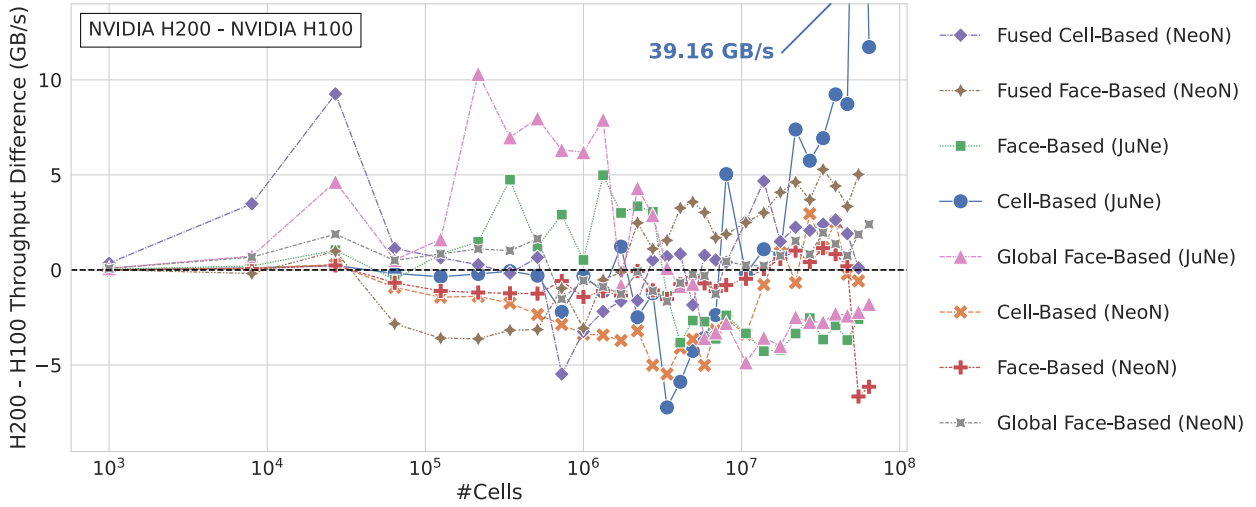


Figure 6.18.: NeoN vs. JuNe: Difference in matrix assembly throughput between the H200 and H100 GPU. The y axis was limited to better visualise the majority of data points. One outlier is given by the annotation.

6.3.3 Integration Benchmark

Previously, the matrix assembly strategies are evaluated using only the divergence and Laplacian operators. These investigations are reasonable and necessary for answering the research questions posed in Section 1.1. However, the full momentum equation is central to the motivation of this work, and not evaluating JuNe on it would render this work incomplete. As a reminder, the momentum equation is shown in Listing 6.3. As this thesis is partially motivated by introducing GPU support into CFD software, the integration benchmark is only performed on GPU. Lastly, as stated at the beginning of this chapter, the incompressible solver IcoFoam is included in this benchmark as well to provide additional performance context.

```

1 dsl::Expression<NeoN::scalar> MomentumEquation (
2     dsl::imp::ddt(U)
3     + dsl::imp::div(phi, U)
4     - dsl::imp::laplacian(nu, U)
5 );

```

Listing 6.3.: The momentum equation as defined in NeoN.

The set of benchmarks that evaluate the matrix assembly of the full momentum equation on NeoN and JuNe will in the remainder of this work be referred to as *integration benchmark*. The two cases selected for this integration benchmark are summarised in Table 6.3. The lid-driven cavity case is well-suited for scaling studies and fine-grained comparisons of algorithms. However, because the case is geometrically too simple and does not occur in the industry,

additional cases are needed. The DrivAer case is therefore included in this benchmark. To achieve comparable results between the cases, the evaluations are all performed on meshes with roughly 64M cells. The largest LDC previously used has exactly that amount and the coarsest DrivAer mesh has 65.3M cells.

Table 6.3.: Cases used for the integration benchmark.

Case Name	#Cells	#Non-Zeros	#Faces	#Boundary Faces
Lid-Driven Cavity	64.000.000	447.040.000	191.520.000	960.000
DrivAer	65.334.765	460.041.233	197.353.234	5.918.634

Table 6.4 shows the results of the matrix assembly strategies on the largest LDC with the full momentum equation. The results in Table 6.4 effectively mirror the results for the 64M LDC shown in Figure 6.17: JuNe’s face-based strategy achieves the overall best results with a total speedup of 4.63x over the non-fused NeoN implementation. The throughput result of approximately 273 GB/s is very similar to the assembly results calculating only the spatial operators in Figure 6.17. Considering that the throughput values did not noticeably change after additionally calculating the transient operator further indicates that the implemented operator fusing strategy has negligible overhead. The fused strategies implemented in NeoN show moderate speedups with 1.47x and 2.25x for the cell-based and face-based strategies, respectively.

Table 6.4.: LDC GPU integration benchmark on the H200 GPU. Speedup is calculated with respect to the corresponding non-fused NeoN strategy. The COE is calculated with respect to IcoFoam with 64 MPI cores (see Eq. (34)).

Impl.	Strategy	Assembly Time	Throughput (GB/s)	Speedup	COE
JuNe	Cell-Based	55.22 ms	194.30	1.47x	482.7
	Face-Based	39.23 ms	273.49	4.63x	679.5
	Global Face-Based	41.04 ms	261.43	3.91x	649.5
NeoN, Fused	Fused Face-Based	80.75 ms	132.87	2.25x	330.1
	Fused Cell-Based	55.18 ms	194.43	1.47x	483.0
NeoN	Cell-Based	81.25 ms	132.05	–	328.0
	Face-Based	181.78 ms	59.02	–	146.6
	Global Face-Based	160.64 ms	66.79	–	165.9

Table 6.5 summarises the results of the integration benchmark performed on the DrivAer case. The relative order of all strategies remains the same as before: the fused face-based strategies of JuNe show the overall highest values. Compared to the integration benchmark performed on the lid-driven cavity, however, the fused strategies of NeoN have significantly decreased in performance. The non-fused strategies of NeoN also decreased in performance, with the cell-based strategy decreasing the most from 132 GB/s to 81 GB/s. Judging by the COE values, IcoFoam also significantly reduced in performance compared to the LDC case. Naturally, the dramatic increase in mesh complexity is the most likely cause for this. Interestingly, the face-based strategies implemented in JuNe seem to have increased in assembly time by roughly

10 ms, while the cell-based strategy of JuNe even slightly decreased in assembly time. The cell-based strategy implementation has one important advantage over the face-based strategies: all operators can be evaluated in a single pass over the mesh. The expected speedup obtained by fusing the momentum equation would thereby be roughly 3x. Combined with the slight performance decline of the face-based strategies and the slight performance incline of the cell-based strategy of JuNe, this may be an indicator that the cell-based strategy is a more stable choice of assembly strategy when simulating a variety of different, complex meshes. However, this hypothesis would have to be verified by future research endeavours.

Table 6.5.: DrivAer GPU integration benchmark on the H200 GPU. Speedup is calculated with respect to the corresponding non-fused NeoN strategy. The COE is calculated with respect to IcoFoam with 64 MPI cores (see Eq. (34)).

Impl.	Strategy	Assembly Time	Throughput (GB/s)	Speedup	COE
JuNe	Cell-Based	51.38 ms	208.81	2.57x	1096.8
	Face-Based	49.69 ms	215.92	4.68x	1134.2
	Global Face-Based	52.12 ms	205.85	3.96x	1081.3
NeoN, Fused	Fused Face-Based	500.25 ms	21.45	0.46x	112.7
	Fused Cell-Based	480.93 ms	22.96	0.27x	117.2
NeoN	Cell-Based	132.00 ms	81.28	–	426.9
	Face-Based	232.61 ms	46.12	–	242.3
	Global Face-Based	206.14 ms	53.56	–	273.4

6.3.4 Summary

The fused strategies of NeoN consistently show higher throughputs than JuNe in the serial implementation. In the multi-threaded context at the hardware limit of 128 threads, the fused strategies of NeoN produce significantly higher results while all other strategies are clustered together in a corridor of approximately 15 GB/s. Performing the assembly strategies in parallel at the speedup peak of 16 threads shows overall slightly higher values and a different order of strategies: While the fused cell-based strategy of NeoN still performs the best, the fused face-based strategy throughputs have significantly decreased. Conversely, the cell-based strategy of JuNe even produces better results for the four largest cases compared to the fused counterpart of NeoN. Despite functionally equivalent implementations, the Julia assembly strategies do not seem to perform as effectively as C++ in a multi-threaded context.

The GPU implementation is a direct contrast to the multi-threaded assembly results: The face-based strategies of JuNe show the highest throughputs for medium- to large-sized meshes and the cell-based implementation produces similar values to the fused counterpart of NeoN. The non-fused strategies of NeoN show the overall lowest values, although the throughput of the cell-based strategy on the 64M mesh is similar to the fused face-based strategy.

Introducing the additional transient operator has an insignificant impact on performance compared to the results of Section 6.3.2. In contrast, the additional mesh complexity of the

DrivAer case dramatically reduced the performance of the fused strategies of NeoN. While the face-based strategies of JuNe had a performance decrease of approximately 25%, the cell-based strategy slightly gained performance, possibly indicating better stability with respect to mesh complexity.

6.4 Roofline Model

In this section, the theoretical performance limits are juxtaposed against the measured performance of the strategies. The findings thereof provide an approximative answer to RQ II.b. To create a roofline model for the H200 GPU, the total bandwidth is needed. Within this chapter, the assembly throughput has referred to the size of the assembled matrix per time of assembly. The total bandwidth is computed from the throughput values, arithmetic intensities of Section 4.4, and the LDC memory footprint in Table A2. The ratio of memory read traffic to memory write traffic is calculated as follows: for the face-based strategies, the reads and writes of the internal and boundary faces are multiplied with the respective amount of faces. For the cell-based strategy, the number of cells and the number of boundary faces are used for the weighted sum. The ratios for the strategies are shown below:

$$\begin{aligned} \left(\frac{\text{Read}}{\text{Write}} \right)_{\text{face-based}} &= \frac{52 * I + 97 * B}{96 * I + 96 * B} = 0.543 \\ \left(\frac{\text{Read}}{\text{Write}} \right)_{\text{cell-based}} &= \frac{277 * C + 97 * B}{168 * C + 96 * B} = 1.648, \end{aligned} \tag{36}$$

with I being the number of internal faces, B the number of boundary faces, and C the number of cells (see Table 6.3). Note that the face-based ratio also applies the global face-based strategy, as the arithmetic intensity is the same. Using the ratios, the total bandwidth can be calculated from the benchmark results using $B_{\text{total}} = B_{\text{write}} + B_{\text{write}} * \left(\frac{\text{Read}}{\text{Write}} \right)$. For the cell-based strategy, the bandwidth is 513 GB/s and the bandwidth for the face-based strategies it is 511 GB/s. Multiplying these bandwidths with the arithmetic intensities reported in Table 4.1 gives the attained performance.

Figure 6.19 shows the roofline model for the assembly strategies implemented into JuNe, based on the size of the 64M cell LDC and the throughputs reported in Section 6.3.2. Consequently, the transient operator is not included in these results. Considering the minimal impact on the results in the integration benchmark, the impact on the roofline is likely minimal as well. The roofline model shows that all three strategies are significantly memory bound. The cell-based strategy achieves a performance of 115 GFLOP/s while the face-based strategies show a higher performance of 143 GFLOP/s. For the current implementation and experimental configuration of case and scheme selection, the theoretical maximum performance is approximately 1.14 TFLOP/s for the cell-based strategy and approximately 1.4 TFLOP/s for the face-based strategies.

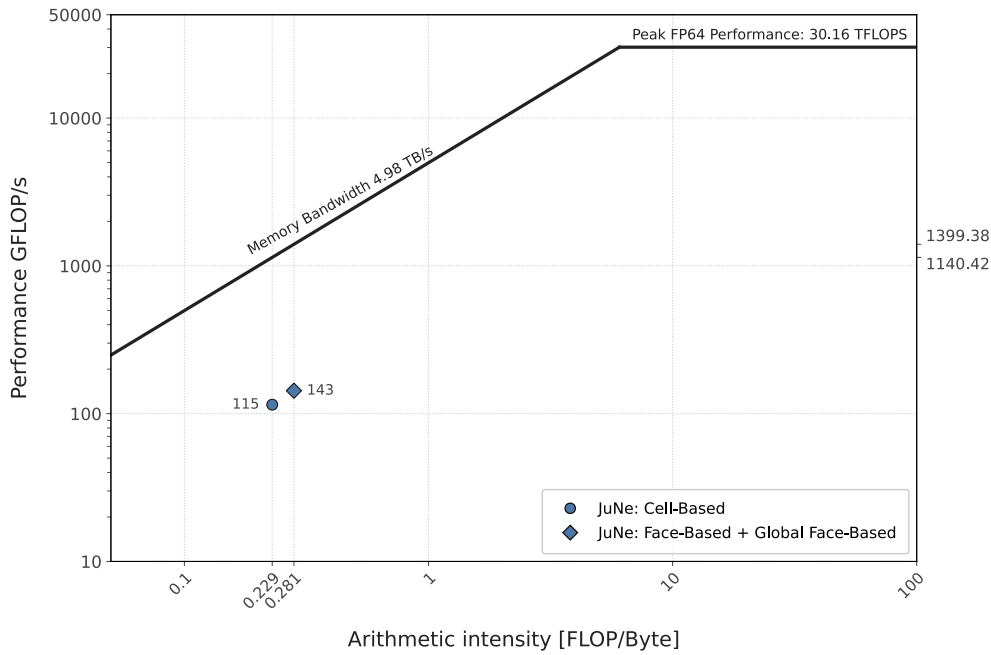


Figure 6.19.: Roofline model for the LDC case with 64M cells on the NVIDIA H200.

6.5 Compilation Times

This section provides argument towards the research questions RQ III.b and RQ III.c. Figure 6.20 presents the compilation times of NeoN, as well as the relative increase in compilation time when compiling with Julia enabled. A selection of different build options is considered, namely building with or without Julia matrix assembly, and building for GPU or CPU. The figure further differentiates between the two hardware nodes used throughout this evaluation and the number of CPUs used for the parallel build process, 32 or 64.

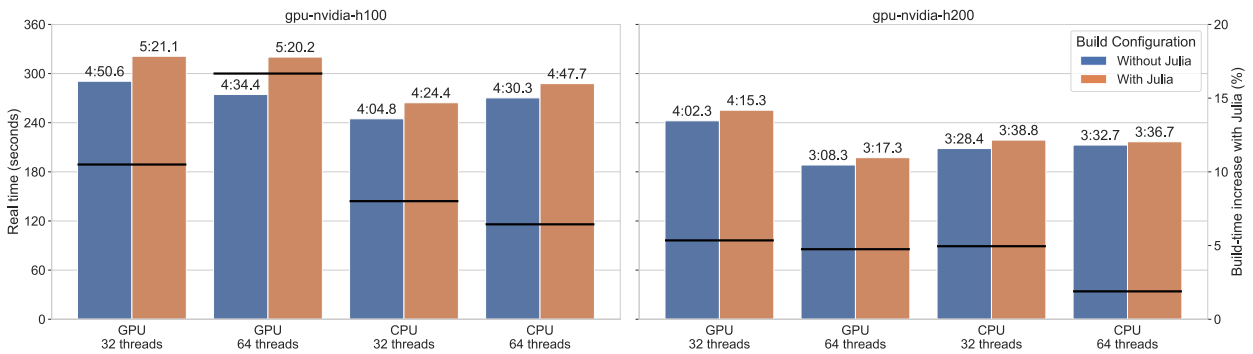


Figure 6.20.: Compilation time comparison, building NeoN with and without Julia. The duration is displayed above the bars in the format of min:sec:ms. H100 and H200 refer to the cluster node (see Table 6.1). The horizontal bars show the percentage overhead introduced by Julia.

Building with Julia enabled consistently increases the compilation time across all build permutations. The largest increase is for the GPU build on the H100 node with 64 parallel workers with a difference of roughly 46 seconds, or a relative increase of approximately 17%. The largest difference besides the H100 GPU builds are the H100 CPU builds with a difference of 17-20 seconds.

The relative compilation time differences in the H200 results range between approximately 2% to 6%. On a 3:30 min compilation time on average, an additional 10 seconds equates to an increase of less than 5%. Considering the speedups reported in Table 6.5 and Table 6.4, even the maximum of 17% increase in compilation time is worth taking into account, especially considering these compilation times are taken from a fresh build. Subsequent compilation times are significantly lower. To conclude, the achieved assembly times and speedups of JuNe overshadow the increase in compilation time.

6.6 Summary

This chapter presented the typical metrics and benchmarking cases in the field of HPC FVM, and the results of the experiments conducted in this work. The structure of evaluations including the focused research questions is given in Table 6.2.

Section 6.2.1 gave the results for the evaluations dealing with matrix assembly implemented in standalone Julia. Specifically, four different variants of evaluating the PDE operators and their interpolation schemes are compared for the answer to RQ I.b: calculating schemes dynamically through function arguments, pre-calculating the interpolation weights, statically encoding the interpolation schemes into the assembly, and the operator fusion strategy. Following this, the section also provided insights towards RQ I.a by evaluating the performance of the operator fusion strategy. The operator fusing shows virtually zero overhead compared to the three other calculation variants, and even better throughput values in the serial implementation. Overall, the way of evaluating the operators does not seem to have a significant impact on matrix assembly performance: The differences are most pronounced in the serial implementation and the upper bound of the GPU implementation. Otherwise, the computational impact is likely too small to be reflected in the results over the entire range of LDC mesh sizes.

Section 6.2.2 reported the standalone Julia matrix assembly strategy evaluation on the lid-driven cavity case of up to 64M cells, providing insights for RQ II.a. In the standalone Julia matrix assembly, the mesh size and the computational context determine which strategy is the most efficient. In the serial implementation, the face-based and global face-based strategies show the highest throughput values, while the cell-based strategy shows dramatic peaks for medium-sized meshes (see Section 6.2.3) in the multi-threaded implementation. In the GPU implementation, the batched face-based strategy shows the best results with a peak throughput of 700 GB/s.

In Section 6.3, the interfaced matrix assembly, JuNe, is juxtaposed against the fused and non-fused assembly strategies of NeoN. In the serial implementation, the NeoN matrix assembly throughputs consistently outperform the results of JuNe. The multi-threaded results also show considerably higher results for the fused strategies of NeoN than the ones implemented in JuNe. The speedup values shown in Figure 6.15 show linear and superlinear results for JuNe's cell-based strategy, which is relativised by the considerably low serial throughput values. In contrast to the CPU results, the GPU implementation of the JuNe face-based assembly strategies shows speedups of close to 2x in comparison to the fused counterparts of NeoN. The face-based strategies reach peak values of 270 GB/s while the fused face-based strategy of NeoN does not exceed 170.

6.6 Summary

Results for the integration benchmark, in which the full momentum equation is evaluated on the largest LDC and the coarse DrivAer case, are presented in Section 6.3.3. The LDC benchmark mirrors the results for the largest LDC shown in Section 6.3. The dramatic increase in mesh complexity of the DrivAer case leads to a throughput decrease of most strategies. Most noticeable is that the fused strategies of NeoN report a slowdown compared to the non-fused counterparts. IcoFoam also increases in assembly time, resulting in COE values of greater than 1k for the assembly strategies implemented in JuNe. The cell-based strategy slightly increases in throughput compared to the LDC integration benchmark. This leads to the hypothesis that this strategy may handle increased mesh complexities better due to the fusing of all three operators into the cell-based iteration.

Section 6.4 combined measurements from the preceding benchmarks, the hardware specification of the H200 GPU and the arithmetic intensities derived in Section 4.4 to show the roofline model for JuNe. Giving an approximative answer to RQ II.b, the roofline diagram is used to obtain the theoretical maximum performance of the three strategies: approximately 1.1 TFLOP/s and 1.4 TFLOP/s are the maxima of the cell-based strategy and face-based strategies, respectively.

Transitioning into the retrospective discussions about the development effort using Julia in Chapter 7, the compilation times of NeoN are reported in Section 6.5. A selection of build configurations is timed: building NeoN with or without Julia, performing the build on 32 or 64 threads, building for CPU or GPU, and differentiating between the two nodes of the HPC cluster. The increase in compilation time with Julia are overall considered to be insignificant, especially considering the dramatic speedups reported in Table 6.5 and the fact that subsequent builds will not take as long as building from a clean state.

7 Discussion

This chapter discusses the required development efforts porting the FVM-based matrix assembly from C++ into the Julia programming language. Specifically, the creation of the bridge between the two languages and the development using Julia on the example of matrix assembly strategies are of special interest. As part of this chapter, RQ III.c is discussed: *How does the interface affect the development effort?*. This question is answered in the form of a reflective case study about the development of JuNe, compared to the quantitative results obtained by the evaluations conducted in Chapter 6. Naturally, parts of this discussion will be subjective in nature.

7.1 Development Effort

Julia is currently a popular language with a very active community. Sometimes, even several packages have been developed independently for the same use case. However, an active community with lots of development effort does not automatically follow good development practices such as comprehensive documentation. This unfortunately applies to parts of the Julia stdlib as well, which significantly slowed down the development of this research. A chapter dedicated to embedded Julia exists in the documentation²³. However, quite frequently, the author had to consult the source code of the language in order to fully grasp the C API. For instance, primitive data types have to be boxed before passing them from C++ to Julia (see Section 5.6.3). In the documentation, this is only illustrated for double values, and the correct boxing for types like `uint8_t` or `int32_t` had to be searched for in the repository. Even more inconvenient is the lack of documentation covering the passing of GPU vectors to Julia, as it is realised differently than sending CPU vectors (see Section 5.6.2).

During the development of JuNe, a test-driven approach was used to build the individual building blocks. The unit tests incrementally build in complexity from sending a scalar value over sending a string to sending multiple vectors of different types. Regardless, assembling the interface from these unit tests was not straightforward.

The development of JuNe was slowed down by a plethora of bugs that occurred by the non-intuitive use of the interface. Predominantly, switching between zero-based indexing of C++ and one-based indexing of Julia was the cause of numerous off-by-one errors. The difficulty of this is increased by the fact that some variables (e.g., the diagonal offsets) need to start from zero on both sides. The same holds for treating a two-dimensional vector as one-dimensional in addition to the off-by-one indexing. This applies to the vectoring storing the matrix coefficients, as well as other inputs like the values of previous iterations needed by the transient operator

Another potential point of failure is that the function call on the NeoN side and the function definition on the Julia side need to be identical in order of arguments. Because no language server exists that can modify function definitions and calls over language boundaries, manual

²³<https://docs.julialang.org/en/v1/manual/embedding/>

modifications on both sides are required. This increases the likelihood of bugs and errors and overall negatively impacts the development. Switching the order of two arguments on one side might, in the best scenario, lead to an error that the supplied arguments do not match the type. Worse is accidentally switching two arguments of the same type. Even if the swap results in a previously not occurring error, resulting debugging effort is tedious.

The development of the GPU implementation of the interface comes with additional difficulties as well. Naturally, the issues of the previous paragraph apply here, too. Additional complications arise with a hardware issue with one of the GPU nodes of the HPC cluster where the GPU was not detected by Kokkos at runtime. The resulting Error thrown by Julia stated that CUDA could not interpret the passed argument as a valid GPU pointer. As discussed in Section 5.6.2, GPU vectors are handled differently than CPU vectors: GPU vectors are passed as pointers, the vector length, and are finally interpreted into a Julia CUDA object while the CPU vectors can simply be passed as a thinly wrapped vector (see Listing 5.8). This vector length either corresponds to the number of internal faces, the number of boundary faces, their sum, or the number of cells in the mesh. Manually passing these values into the reinterpretation helper method leaves room for copy-paste mistakes. More than once, a vector was accidentally given the incorrect size, leading to out of bounds errors in the best case, and incorrect coefficients in the worst. This is also affected by potential changes to the data model on the NeoN side. During development on JuNe, vectors that initially held some data for all faces were split into the internal and boundary part. Naturally, if this change goes unnoticed, the assembly call likely throws an error or returns incorrect coefficients as well.

Related to the broader topic of API changes is the interaction with the active development process of NeoN. Because the interface is currently implemented independently of the matrix assembly in NeoN, any changes to the input data are not always obvious. For instance, obtaining the delta coefficients for the calculation of the Laplacian operator changed fundamentally, and got merged into the JuNe feature branch of NeoN because of an unrelated needed functionality. Consequently, the coefficients calculated by the Laplacian operator were incorrect.

On the other hand, the overall development speed of Julia is significantly faster than C++. Especially the time until a first working prototype is considerably shorter due to the simple and non-verbose syntax, lack of static compilation, and dynamic type system. Additional advantages include the simple syntax comparable to Python, but achieving processing times that are comparable to C++. Not needing to wait for a full re-compilation of NeoN in order to debug the Julia side of the interface is exceptionally convenient.

Another advantage of the Julia programming is the REPL or read-eval-print-loop, i.e., an interactive shell executed in the terminal. It is comprised of three main parts: the built-in package manager, a shell dedicated for documentation and helpful examples, and a primary shell similar to the Python REPL. Because the experience gained through package management and the documentation shell are primarily superficial, the focus of this paragraph is the REPL. The main advantage is that it allows for quick experiments and small-scale debugging, whereas the same testing would require writing an entry point in C++ and the compilation thereof. The operator fusion implementation presented in Section 5.4.4 was initially designed in the REPL. Packages and entire files can be loaded into the REPL, allowing manual end to end testing. The only considerably caveat of the REPL is that it may have to be restarted in some scenarios:

After working inside an active REPL for a prolonged duration, some values or functions may get cached and the only way to change their value is by restarting the session. The second one primarily applies to local development, i.e., not on a HPC cluster. Since it is also possible to run benchmarks through the REPL, it is not uncommon that the memory limitations of the local machine is reached or exceeded, causing the system to freeze. Closing the REPL to free memory was therefore not a rare occurrence. Overall, the REPL aids the development process by facilitating rapid experimentation through an interactive shell.

7.2 How Does the Interface Affect Development?

One motivation of extracting the matrix assembly out of C++ to Julia was to improve development effort by reducing the amount of templates and, consequently, the complexity of introducing new features and the overall compilation time. The argument regarding the time until the first working solution is also included in the improved development effort. However, this cannot entirely be attributed to the interface, as it is more an advantage of the language itself. As for the development effort of the interface, it is reasonable to state that the overall difficulty is roughly equal and that the interface trades higher development speeds for potentially more subtle bugs at runtime, e.g. the bugs related to indexing or correctly sizing the vector as mentioned in Section 7.1.

Note that these bugs oftentimes stem from human carelessness, e.g., copy-and-paste mistakes or accidentally changing the order of arguments only on one side of the interface, i.e., either only on the NeoN-caller side or only on the Julia-callee side. This might be a case where agentic AI development might introduce fewer bugs. However, the research questions targeting effort of development are rendered inconsequential when AI development is introduced into this argument. Thus, the effect of AI shall not be considered here.

The benefits and drawbacks of the cross language development are summarised in Table 7.1. Since RQ III.c is a rather subjective question, the answer is offered subjectively as well.

Table 7.1.: Weighing advantages and disadvantages of the extracted matrix assembly. Written in bold are the arguments that are most significant to the author, given the experience of developing this interface.

Advantage	Disadvantage
Higher development speeds	Poor documentation for niche topics
No C++ rebuild for Julia changes	Potentially more subtle bugs
Fast computation, yet simple syntax	Language server cannot help

Some of these arguments are more impactful in specific phases of the development process. For instance, the poor documentation negatively impacted particularly the beginning of the development. Conversely, the impact of *potentially more subtle bugs* is negligible in the beginning, because the interface has not been developed yet. Lastly, some arguments have a uniform impact over all phases of development. *No C++ rebuild for Julia changes* falls in this category because it equally applies to the early stages of unit testing as well as the later stages of debugging the implemented matrix assembly. The *higher development speeds* generally applies

7.2 How Does the Interface Affect Development?

to all stages. Here, however, it specifically refers to the time until a first working prototype. Therefore, it applies primarily for the middle stage of development, as the underlying structure of the interface exists and the development shifts to the Julia side of the interface. In the case of embedding Julia into C++, the development speed at an early stage is limited until the C++ side of the interface is fully developed. Additionally, the potential scarcity of documentation further limits the development speeds.

Because these arguments fall into different stages of the development process, an answer to RQ III.c is given with respect to the individual stages and illustrated in Figure 7.1.

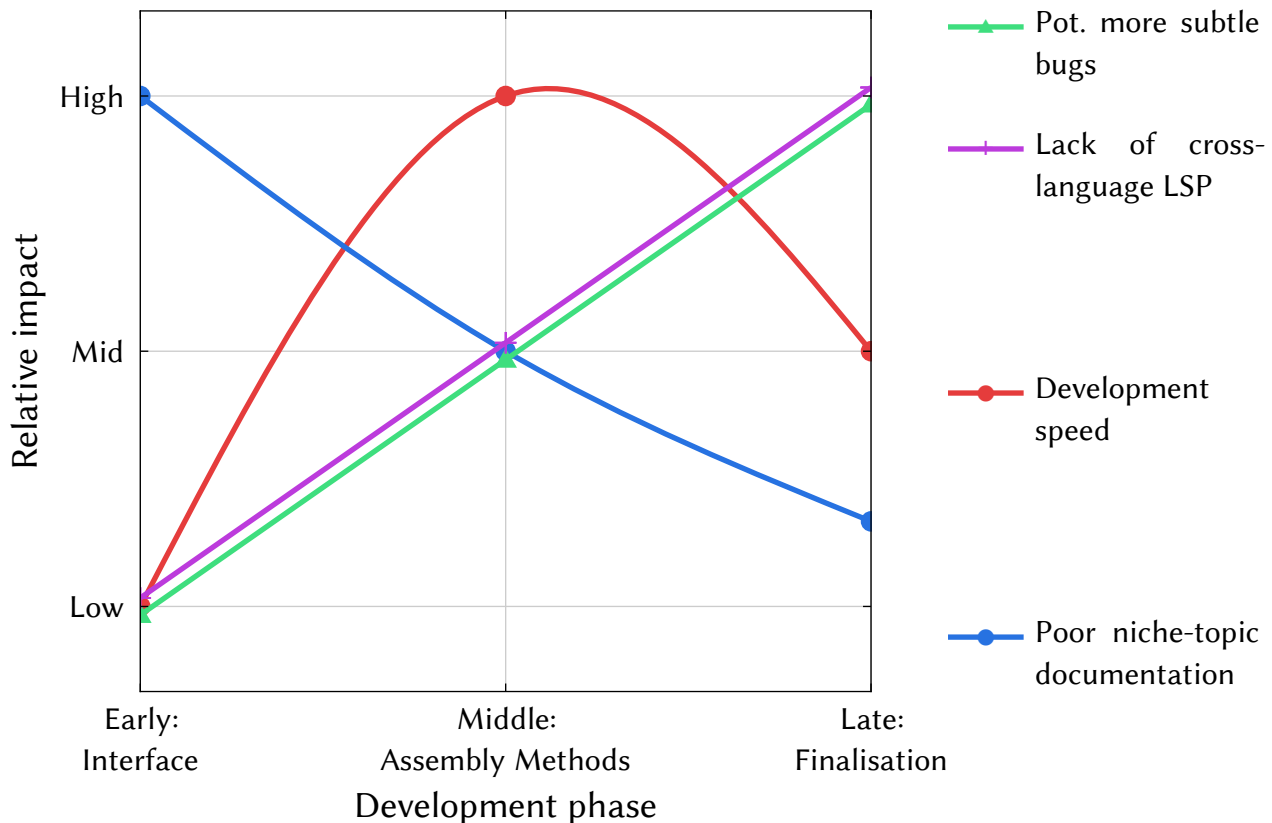


Figure 7.1.: Impact of advantages and disadvantages on each phase of the development process. For the sake of clearer visualisation, *Development speed* also includes the factors *no C++ rebuild for Julia changes* and *fast computation, yet simple syntax*.

The early stage of development is impacted most by *poor documentation for niche topics*. At this stage, the other factors may not have as much impact because the early stage is dominated by development of the C++ side of the interface. Therefore, the impact of the missing cross-language LSP, development and calculation speeds of Julia, and the potential of subtle bugs is limited. RQ III.c asks how the interface affects the development effort. Arguably, the interface does not fully exist at this stage, and the main detriment to development effort is the lack of documentation, which is not caused by the interface itself. However, foreign documentation is part of every development process and shall thus be included. Therefore, the interface has an overall negative impact on the development process in the early stage of development.

In the middle stage, the development of the assembly methods, most of the development is spent on the Julia side of the interface. As most of the niche topic problems are likely solved

at that point, the impact of scarce documentation decreases. With more development on the Julia side, the impact of the potential bugs and the lack of a language server become more apparent, though the advantages of Julia development become more impactful as well. Overall, the interface positively affects the development process in this stage.

The last phase of development, the finalisation, deals with the minor adjustments to enable successful benchmarks. During the development of this work, this stage proceeds in an iterative fashion: First, running benchmarks and comparing the assembled operators generated by the different backends between the interface and pure NeoN. Any difference between them usually stem from an incorrect implementation or some missing functionality on the Julia side. Regarding the latter, and considering the scope of this thesis, missing functionality is not implemented. Because this missing functionality primarily relates to schemes and boundary conditions not considered for this work, the case definitions are modified instead. The benchmark is performed and, if the difference of the resulting matrices exceeds $e-12$, the next set of issues is investigated, continuing the cycle. The effect on overall development process of this phase depends on the number of bugs accidentally introduced into the interface. In the best case, little to no further work is needed. In the worst case, numerous iterations of the debugging cycle are necessary. The work on the bridge between C++ and Julia is likely minimal at this stage and the development speed of Julia is not as impactful compared to the middle stage of development. Therefore, the impact of the late stage has an overall neutral impact on the overall development process.

In summary, the impact of the three stages are negative, positive, and neutral. However, the middle and late stages proceed for every new assembly strategy to be implemented. Conversely, the relative impact of the early stage decreases and shifts more towards the later stages because the underlying concepts of the interface are not likely to change. Therefore, and to answer RQ III.c, the interface has an overall positive effect on the development process.

7.3 Summary

This chapter discussed the development process of JuNe, especially focusing on the advantages and disadvantages specific to Julia, but also in context of the interface to C++. Section 7.1 introduced a compilation of arguments that are used to answer how the interface affects the development effort. Most notably included in these arguments are the potential lack of documentation for niche topics, bugs stemming from language differences, but also positive features like the increased development speeds and quality-of-life features of Julia compared to C++. Section 7.2 divides the development process into three phases and assigns each entry of Table 7.1 an impact score to each phase. Concluding Chapter 7 and answering RQ III.c, the interface overall positively affects the development efforts. The following chapter provides a synopsis of this work.

8 Conclusion

Introducing GPU support into OpenFOAM has been an objective by numerous research projects in recent years. Two major approaches for this exist: adding new or modified functionality through plugins, and development on the OpenFOAM codebase itself. NeoN is selected as the representative for the latter. For a simulation, the time spent on the discretisation of the underlying PDE, the matrix assembly, and the solver varies for the different use cases. Surveys indicate that one can assume that they contribute in equal parts. NeoN’s DSL solves some of the caveats of the matrix assembly implemented into lcoFoam. However, the overall arithmetic intensity and memory boundedness motivate further improvements. Realising that performing multiple passes over the mesh, once for each PDE operator, is inefficient, an experimental fusing of operators was introduced into NeoN. While it reduces the iterations to one, the implementation is verbose: fused operators are created through template-laden code in addition to the non-fused ones. In an ideal solution, the operator fusing would happen dynamically, without explicit definitions of each fused operator. Because this is not possible in C++, a different solution is needed. This thesis proposes the extraction of the matrix assembly out of NeoN into the programming language Julia. Julia’s just-in-time compilation enables dynamic operator fusion and recent studies have shown Julia to be exascale and petascale-ready [Regier et al., 2019; Schmalfuß et al., 2026]. The extracted Julia matrix assembly is accessed from an C API interface implemented into NeoN’s DSL. This novel Julia matrix assembly with dynamic operator fusing embedded into NeoN is called **JuNe**.

8.1 Research Questions Findings

In order to thoroughly evaluate this matrix assembly system, including the operator fusing strategy, a collection of research questions is compiled in Section 1.1. The presented thesis evaluates the research questions experimentally, the answers are synthesised in the next sections. Thereafter, some limitations to the evaluation are discussed and future research directions are outlined.

8.1.1 RQ I: Operator Calculation & Fusing

RQ I.a: What are the limitations and performance implications of the operator fusing strategy?

The operator fusing strategy is explained in detail in Section 5.4.4. The limitations of the fusing strategy primarily stem from its implementation: NeoN generally supports that the factor of an operator can be a field, thereby possibly applying different values depending on the position in the mesh. Currently, only scalar factors for the operators are supported by JuNe. Furthermore, as illustrated by the operator implementations in Section 5.4, the fusing strategy requires all fusable operators to have matching function arguments. As of now, this is not a problem since this only affects the two spatial operators. Should more operators be implemented in the future, this

may introduce conflicting function definitions that need to be differentiated. The performance implications of the fusing strategy are evaluated in Section 6.2.1. Therein, it was compared against three other ways of evaluating the operators, namely pre-calculating the scheme values, hard-coding the scheme function call, and passing the scheme as a function into the assembly. agree on the overhead of the proposed operator fusing strategy being negligible. In some cases, like the serial implementation or the GPU implementation of the batched strategy, it even shows better results than other strategies.

RQ I.b: What is the cost of different runtime or just-in-time abstraction approaches, e.g., for interpolation schemes?

The differences between the calculation abstractions is overall small. The results for the serial implementation shown in Figure 6.3 reveal the most pronounced differences. The dynamic calculation variant shows the highest assembly throughput values by a small margin. The pre-calculation is the overall worst variant because of additional passes over the mesh and additional memory footprint inside the assembly. The difference between the selected interpolation schemes are negligible, likely due to the insignificant amount of work that is further optimised by the JIT compiler.

8.1.2 RQ II: Matrix Assembly Strategies

RQ II.a: What are the performance implications of different matrix assembly strategies?

The answer to this questions is obtained through the evaluations of the standalone Julia assembly in Section 6.2.2 and the comparison of JuNe and NeoN in Section 6.3. On a high level, the answer to this question is that the best-performing assembly strategy depends on the computational context. The results of the standalone assembly and JuNe agree on the best-performing strategy in the multi-threaded and the GPU context; face-based strategies outperform the cell-based strategy in the serial standalone implementation. Conversely, the results of the serial interfaced matrix assembly reveal higher throughput values of the cell-based strategies of NeoN and JuNe. The cell-based strategy seems to exploit small to medium-sized meshes fitting in the L3 cache the most (see Figures 6.7, 6.8, 6.13, 6.14). In the standalone GPU assembly, the batched face-based strategy shows the highest results by a significant amount. The batched strategy was not implemented into JuNe due to the extensive refactoring of NeoN that would be required. The GPU results show that the face-based strategies of JuNe yield the highest throughputs. The fused strategies of NeoN seem to be the more appropriate choice for performing the matrix assembly in a CPU context, while JuNe performs better on the GPU.

These findings of the GPU implementation are supported by the integration benchmark in Section 6.3.3, in which the full momentum equation is evaluated on a 64M cell lid-driven cavity and the coarse DrivAer case. The LDC results mirror the results of the preceding evaluations. All strategies show improvements in comparison to the respective non-fused counterparts of NeoN, with the fused cell-based strategies of NeoN and JuNe performing nearly identically and the face-based strategies of JuNe achieving a speedup of 4.63 over non-fused NeoN. The results of the DrivAer benchmark show different results: the fused strategies of NeoN significantly decreased in performance and are even inferior to the non-fused strategies. The face-based strategies of JuNe slightly decrease in performance, but continue to show speedups of up to 4.6x compared

to non-fused NeoN. JuNe’s cell-based strategy slightly improved in performance, hinting at possibly better adaptability towards changing mesh complexity. The best-performing assembly strategies with respect to the execution context are summarised in Table 8.1.

Table 8.1.: Best-performing assembly strategy for each computational context. The highlighted strategies indicate the strategy that performs best out of all strategies of NeoN and June for that given context. Note that the standalone section is to be read separately.

Impl.	Context	Strategy
JuNe	Serial	Face-based
	Multi-Threaded	Cell-based
	GPU	Face-based
NeoN	Serial	Fused Cell-based
	Multi-Threaded	Fused Cell-based
	GPU	Fused Cell-based
Standalone Julia	Serial	Face-based
	Multi-Threaded	Cell-based
	GPU	Batched Face-based

RQ II.b What is the near-optimal performance of typical CFD kernels?

The arithmetic intensities of the matrix assembly strategies implemented into JuNe are derived in Section 4.4. Using the measurements from the experiments obtained in Section 6.3, the roofline model for the assembly strategies in JuNe is shown in Figure 2.1. According to the hardware specifications of the NVIDIA H200, the theoretical peak performance of the cell-based and the face-based strategies are approximately 1.14 TFLOP/s and 1.4 TFLOP/s, respectively. The matrix assembly is located far left of the ridge point and is thereby memory-bound.

8.1.3 RQ III: Integration into CFD Frameworks

RQ III.a: What is a good strategy to interface the matrix assembly with existing CFD frameworks?

As discussed in Section 4.5, the options of embedding Julia into C++ are limited: most projects focus on embedding statically compiled C++ into Julia and other projects that embed Julia into C++ are not maintained anymore. Therefore, the only viable option for the integration of Julia into NeoN is the built-in C API. While the interface is currently implemented into the PDESolver class of NeoFOAM, the most appropriate location would be the Expression class of NeoN because it is the class that is closest to the individual operators.

RQ III.b: What is the performance overhead of the interface between C++ and Julia?

This question is primarily impacted by two factors: the compilation time of NeoN including Julia and the first execution of the matrix assembly. First, the compilation time of a clean build is measured in Section 6.5. The increase in compilation time is overall insignificant, especially considering that subsequent build times are lower and because of the one-time increase in compilation time are overshadowed by the significant decrease in assembly time, for instance

of the GPU face-based strategy. Second, the runtime of the first execution is slightly prolonged due to the JIT compilation of Julia. This is especially pronounced in the context of GPU matrix assembly. However, as mentioned in Chapter 4, the matrix assembly is executed hundreds of times over the course of the simulation. Therefore, the initial JIT compilation time is negligible.

RQ III.c: How does the interface affect the development effort?

The impact on the development effort is discussed in Chapter 7. Therein, the key factors during the development of JuNe are identified retrospectively and mapped onto three phases of development with respect to severity of impact. Overall, the development speeds of Julia and the lack of documentation of niche topics impacted the development process the most. However, because the early stage, where the lack of documentation is most present, is the comparatively shortest stage, the interface is considered to have an overall positive impact on the development.

8.2 Limitations

The main limitation of this work is that the evaluations were performed on a single node of an HPC system. Especially in the context of high-performance computing, evaluating the multi-GPU scaling is standard procedure.

The current process of passing parsable strings from NeoN to Julia is unstable from a software development perspective. Additionally, as stated earlier in this chapter, factors of the PDE operators cannot be passed as a field. A more stable solution to both problems would involve defining some sort of operator definition struct that can be passed alongside of the other arguments.

8.3 Future Work

In Chapter 1, it was reported that the ratio between the time needed for matrix assembly and solver differs on the use case, but may be considered to be of the same magnitude. As it was not the focus of this work to consider the broader scope of the entire simulation, it is nonetheless interesting to re-evaluate this statement with respect to the performance improvement of the operator fusing strategy.

As mentioned in the limitations above, the proceedings of this thesis were not evaluated in a distributed context. Introducing MPI support will be considered for future work. In the same vein, multi-GPU support will be considered as well. NeoN already provides support for both. However, introducing these new functionalities into JuNe, especially the MPI support, includes all three phases of the development process discussed in Section 7.2 and is therefore likely inhibited by the lack of documentation regarding embedded Julia.

Related to the performance improvement, the batched face-based strategy showed promising results in the evaluation of the GPU standalone Julia implementation. Ultimately, it was not integrated into JuNe as this would require significant refactoring of the NeoN software stack.. In future research, it would be interesting to undergo this refactoring process and compare the performance of the batched strategy in JuNe against the results of standalone Julia. Furthermore, a deeper investigation of the reasons behind the 4x speedup of the face-based strategy of JuNe over the non-fused strategy of NeoN is interesting.

Especially the cell-based strategy allows the evaluation of all operators in a single pass over the mesh. Considering that the matrix assembly is memory bound, the possibility of avoiding the matrix assembly entirely and instead evaluating the fused operators inside the linear solver is an attractive strategy. This strategy is referred to as matrix-free and usually not used by FVM and rather by FEM or FDM. In theory, introducing matrix-free into the linear solver requires refactoring. The main challenge is that the linear solver is usually preceded by one or multiple matrix preconditioning steps, which require an explicitly assembled matrix. Strategies involving mixed mesh granularities exist to tackle this exact problem, e.g., using a coarse mesh for discretisation and creation of a preconditioner that can be used in the linear solver of a finer mesh. Regardless, it is interesting to evaluate the performance of such strategies using JuNe.

Bibliography

- [Anzt et al., 2022] Anzt, H., Cojean, T., Flegar, G., Göbel, F., Grützmacher, T., Nayak, P., Ribizel, T., Tsai, Y. M., and Quintana-Ortí, E. S. (2022). Ginkgo: A Modern Linear Operator Algebra Framework for High Performance Computing. *ACM Transactions on Mathematical Software*, 48(1).
- [Axtmann and Rist, 2016] Axtmann, G., and Rist, U. (2016). Scalability of OpenFOAM with Large Eddy Simulations and DNS on High-Performance Systems. In *High Performance Computing in Science and Engineering 16*, pages 413–424. Springer International Publishing.
- [Balay et al., 1997] Balay, S., Gropp, W., McInnes, L. C., and Smith, B. (1997). Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. In *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press.
- [Bezanson et al., 2018] Bezanson, J., Chen, J., Chung, B., Karpinski, S., Shah, V. B., Vitek, J., and Zoubritzky, L. (2018). Julia: dynamism and performance reconciled by design. *Proc. ACM Program. Lang.*, 2(OOPSLA).
- [Bna et al., 2020] Bna, S., Spisso, I., Olesen, M., and Rossi, G. (2020, June). PETSc4FOAM: a library to plug-in PETSc into the OpenFOAM framework. Retrieved from <https://doi.org/10.5281/zenodo.3923780>
- [Bnà et al., 2025] Bnà, S., Giaquinto, G., Fadiga, E., Zanelli, T., and Bottau, F. (2025). SPUMA: a minimally invasive approach to the GPU porting of OPENFOAM. Retrieved from <https://arxiv.org/abs/2512.22215>
- [Burcet et al., 2024] Burcet, M., Oliveira, B., Afonso, J. C., and Zlotnik, S. (2024). A face-centred finite volume approach for coupled transport phenomena and fluid flow. *Applied Mathematical Modelling*, 125, 293–312.
- [Burggraf, 1966] Burggraf, O. R. (1966). Analytical and numerical studies of the structure of steady separated flows. *Journal of Fluid Mechanics*, 24(1), 113–151.
- [Carrascal-Manzanares Carlos et al., 2018] Carrascal-Manzanares Carlos, Imperiale Alexandre, Rougeron Gilles, Bergeaud Vincent, and Lacassagne Lionel. (2018). A Fast Implementation of a Spectral Finite Elements Method on CPU and GPU Applied to Ultrasound Propagation. *Advances in Parallel Computing*. IOS Press.
- [Chen et al., 2024] Chen, A., Erickson, B. A., Kozdon, J. E., and Choi, J. (2024). Matrix-free SBP-SAT finite difference methods and the multigrid preconditioner on GPUs. In *Proceedings of*

- the 38th ACM International Conference on Supercomputing*, pages 400–412. New York, NY, USA: Association for Computing Machinery.
- [Chen and Doolen, 1998] Chen, S., and Doolen, G. D. (1998). LATTICE BOLTZMANN METHOD FOR FLUID FLOWS. *Annual Review of Fluid Mechanics*, 30(Volume30, 1998), 329–364. Journal Article.
- [Dagum and Menon, 1998] Dagum, L., and Menon, R. (1998). OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1), 46–55.
- [Epikhin, 2026] Epikhin, A. (2026). Implementation of Numerical Schemes for the Computation of Incompressible Flows in OpenFOAM. In *Advances in Computational Heat and Mass Transfer II*, pages 72–83. Cham: Springer Nature Switzerland.
- [Galeazzo et al., 2024] Galeazzo, F. C. C., Weiß, R. G., Lesnik, S., Rusche, H., and Ruopp, A. (2024). Understanding superlinear speedup in current HPC architectures. *IOP Conference Series: Materials Science and Engineering*, 1312(1), 12009.
- [Golub and Van Loan, 2013] Golub, G. H., and Van Loan, C. F. (2013). *Matrix Computations - 4th Edition*. Philadelphia, PA: Johns Hopkins University Press.
- [Gärtner et al., 2025] Gärtner, J. W., Olenik, G., Fadel, M. E., Petermann, L., Kronenburg, A., Marschall, H., and Anzt, H. (2025). Testing Strategies for OpenFOAM Projects. *Openfoam® Journal*, 5, 115–130.
- [Heft et al., 2012] Heft, A. I., Indinger, T., and Adams, N. A. (2012). Introduction of a New Realistic Generic Car Model for Aerodynamic Investigations. In *SAE 2012 World Congress & Exhibition*. SAE International.
- [Huang et al., 2025] Huang, S., Cheng, Y., Mittra, R., Zhang, X., and Zhang, X. (2025). Accelerating GPU-Based Parallel FDTD With Advanced Operator Fusion. *IEEE Antennas and Wireless Propagation Letters*, 24(9), 2914–2918.
- [Hundsdorfer et al., 2006] Hundsdorfer, W., Ruuth, S., and Spiteri, R. (2006). Monotonicity-Preserving Linear Multistep Methods. *SIAM Journal on Numerical Analysis*, 41, 605–623.
- [Issa et al., 1986] Issa, R. I., Gosman, A. D., and Watkins, A. P. (1986). The computation of compressible and incompressible recirculating flows by a non-iterative implicit scheme. *Journal of Computational Physics*, 62(1), 66–82.
- [Jasak et al., 2007] Jasak, H., Jemcov, A., and Tuković, Ž. (2007). OpenFOAM: A C++ Library for Complex Physics Simulations.
- [Katz, 1979] Katz, V. J. (1979). The History of Stokes' Theorem. *Mathematics Magazine*, 52(3), 146–156.
- [Koolen and Deits, 2019] Koolen, T., and Deits, R. (2019). Julia for robotics: simulation and real-time control in a high-level programming language. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 604–611.
- [Lesnik et al., 2026] Lesnik, S., Olenik, G., and Wassermann, M. (2026). The First OpenFOAM HPC Challenge (OHC-1). Retrieved from <https://arxiv.org/abs/2603.27565>

- [McCalpin, 1995] McCalpin, J. D. (1995). Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, 19–25.
- [Medina et al., 2025] Medina, H., Ellis, C. D., Mazin, T., Osborn, O., Ward, T., Ambrose, S., Aleksandrova, S., Rothwell, B., and Eastwick, C. (2025). XCALibre.jl: A Julia XPU unstructured finite volume Computational Fluid Dynamics library. *Journal of Open Source Software*, 10(107), 7441.
- [Morgan and Peraire, 1998] Morgan, K., and Peraire, J. (1998). Unstructured grid finite-element methods for fluid mechanics. *Reports on Progress in Physics*, 61(6), 569.
- [Moukalled et al., 2015] Moukalled, F., Mangani, L., and Darwish, M. (2015). *The Finite Volume Method in Computational Fluid Dynamics: An Advanced Introduction with OpenFOAM and Matlab*. Springer Publishing Company, Incorporated.
- [Olenik et al., 2024] Olenik, G., Koch, M., Boutanios, Z., and Anzt, H. (2024). Towards a platform-portable linear algebra backend for OpenFOAM. *Meccanica*.
- [Olenik et al., 2025] Olenik, G., Scheufler, H., Jones, B., Meng, F., Koch, M., Wang, C.-T., Mishchuk, R., Gerlero, G., Weiss, G., and Hetmann, H. (2025, January). NeoFOAM: A Modern CFD Core. Retrieved from <https://doi.org/10.5281/zenodo.14608521>
- [Pelenitsyn et al., 2021] Pelenitsyn, A., Belyakova, J., Chung, B., Tate, R., and Vitek, J. (2021). Type stability in Julia: avoiding performance pathologies in JIT compilation. *Proc. ACM Program. Lang.*, 5(OOPSLA).
- [Rana et al., 2022] Rana, K., Mavrič, B., Zahoor, R., and Sarler, B. (2022). A meshless solution of the compressible viscous flow in axisymmetric tubes with varying cross-sections. *Engineering Analysis with Boundary Elements*, 143, 340–352.
- [Regier et al., 2019] Regier, J., Fischer, K., Pamnany, K., Noack, A., Revels, J., Lam, M., Howard, S., Giordano, R., Schlegel, D., McAuliffe, J., Thomas, R., and Prabhat. (2019). Cataloging the visible universe through Bayesian inference in Julia at petascale. *Journal of Parallel and Distributed Computing*, 127, 89–104.
- [Ristov et al., 2016] Ristov, S., Prodan, R., Gusev, M., and Skala, K. (2016). Superlinear speedup in HPC systems: Why and when?. In *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 889–898.
- [Schmalfuß et al., 2026] Schmalfuß, S., Lesnik, S., Rusche, H., and Niedermeier, D. (2026). SCALE-TRACK: Asynchronous Euler-Lagrange particle tracking on heterogeneous computing architecture. Retrieved from <https://arxiv.org/abs/2603.26691>
- [Schussnig et al., 2025] Schussnig, R., Fehn, N., Munch, P., and Kronbichler, M. (2025). Matrix-free higher-order finite element methods for hyperelasticity. *Computer Methods in Applied Mechanics and Engineering*, 435, 117600.
- [Serra and Semiao, 2021] Serra, N., and Semiao, V. (2021). ESIMPLE, a new pressure–velocity coupling algorithm for built-environment CFD simulations. *Building and Environment*, 204, 108170.

- [Sumesaraayi and Ihmsen, 2021] Sumesaraayi, S. G., and Ihmsen, M. (2021). Lid-driven Cavity (2D). Retrieved from <https://www.fifty2.eu/innovation/lid-driven-cavity-2d-in-preonlab/>
- [Thompson et al., 2022] Thompson, A. P., Aktulga, H. M., Berger, R., Bolintineanu, D. S., Brown, W. M., Crozier, P. S., in 't Veld, P. J., Kohlmeyer, A., Moore, S. G., Nguyen, T. D., Shan, R., Stevens, M. J., Tranchida, J., Trott, C., and Plimpton, S. J. (2022). LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales. *Computer Physics Communications*, 271, 108171.
- [Trott et al., 2022] Trott, C. R., Lebrun-Grandié, D., Arndt, D., Ciesko, J., Dang, V., Ellingwood, N., Gayatri, R., Harvey, E., Hollman, D. S., Ibanez, D., Liber, N., Madsen, J., Miles, J., Poliakoff, D., Powell, A., Rajamanickam, S., Simberg, M., Sunderland, D., Turcksin, B., and Wilke, J. (2022). Kokkos 3: Programming Model Extensions for the Exascale Era. *IEEE Transactions on Parallel and Distributed Systems*, 33(4), 805–817.
- [Varney, 2020] Varney, M. (2020). Base drag reduction for squareback road vehicles.
- [Versteeg and Malalasekera, 2007] Versteeg, H. K., and Malalasekera, W. (2007). An introduction to computational fluid dynamics - the finite volume method.
- [Williams et al., 2008] Williams, S. W., Waterman, A., and Patterson, D. A. (2008). *Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures* (technical report No. UCB/EECS-2008-134). Retrieved from <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-134.html>
- [Zhou, 1993] Zhou, P.-b. (1993). Finite Difference Method. In *Numerical Analysis of Electromagnetic Fields*, pages 63–94. Berlin, Heidelberg: Springer Berlin Heidelberg.

Appendix A. Memory Footprints

Table A1.: LDC memory footprint in the standalone Julia assembly methods. Some values like the memory footprint of the boundaries are not reported, but included in the total.

LDC Size	Face Data (mb)	Cell Data (mb)	Values (mb)	RHS (mb)	Total (mb)
10^3	0.21	0.02	0.05	0.01	0.32
20^3	1.61	0.16	0.43	0.06	2.52
30^3	5.36	0.54	1.47	0.22	8.45
40^3	12.60	1.28	3.51	0.51	19.94
50^3	24.48	2.50	6.88	1.00	38.86
Intel XEON L3 Cache Size (60 mb)					
60^3	42.16	4.32	11.92	1.73	67.05
70^3	66.80	6.86	18.97	2.74	106.35
80^3	99.53	10.24	28.36	4.10	158.62
90^3	141.52	14.58	40.44	5.83	225.70
AMD EPYC L3 Cache Size (256 mb)					
100^3	193.92	20.00	55.52	8.00	309.44
110^3	257.88	26.62	73.96	10.65	411.69
120^3	334.54	34.56	96.08	13.82	534.30
130^3	425.07	43.94	122.22	17.58	679.11
140^3	530.61	54.88	152.72	21.95	847.97
150^3	652.32	67.50	187.92	27.00	1042.74
160^3	791.35	81.92	228.15	32.77	1265.25
170^3	948.84	98.26	273.74	39.30	1517.37
180^3	1125.96	116.64	325.04	46.66	1800.92
190^3	1323.86	137.18	382.37	54.87	2117.77
200^3	1543.68	160.00	446.08	64.00	2469.76
220^3	2053.71	212.96	593.96	85.18	3286.55
240^3	2665.27	276.48	771.38	110.59	4266.09
260^3	3387.57	351.52	981.01	140.61	5423.14
280^3	4229.84	439.04	1225.55	175.62	6772.51
300^3	5201.28	540.00	1507.68	216.00	8328.96
320^3	6311.12	655.36	1830.09	262.14	10107.29
340^3	7568.56	786.08	2195.48	314.43	12122.28
360^3	8982.84	933.12	2606.52	373.25	14388.71
380^3	10563.15	1097.44	3065.90	438.98	16921.37
400^3	12318.72	1280.00	3576.32	512.00	19735.04

Table A2.: LDC memory footprint in the JuNe assembly methods. Some values like the memory footprint of the boundaries are not reported, but included in the total.

LDC Size	Face Data (mb)	Cell Data (mb)	Values (mb)	RHS (mb)	Total (mb)
10^3	0.05	53.01	0.15	0.02	53.39
20^3	0.40	53.06	1.29	0.19	55.89
Intel XEON L3 Cache Size (60 mb)					
30^3	1.34	53.22	4.41	0.65	62.49
40^3	3.15	53.51	10.52	1.54	75.19
50^3	6.12	54.00	20.64	3.00	95.97
60^3	10.54	54.73	35.77	5.18	126.83
70^3	16.70	55.74	56.92	8.23	169.76
80^3	24.88	57.10	85.09	12.29	226.75
AMD EPYC L3 Cache Size (256 mb)					
90^3	35.38	58.83	121.31	17.50	299.79
100^3	48.48	61.00	166.56	24.00	390.88
110^3	64.47	63.65	221.87	31.94	502.01
120^3	83.64	66.82	288.23	41.47	635.16
130^3	106.27	70.58	366.66	52.73	792.34
140^3	132.65	74.95	458.17	65.86	975.53
150^3	163.08	80.00	563.76	81.00	1186.73
160^3	197.84	85.77	684.44	98.30	1427.92
170^3	237.21	92.30	821.22	117.91	1701.11
180^3	281.49	99.66	975.11	139.97	2008.28
190^3	330.96	107.87	1147.11	164.62	2351.41
200^3	385.92	117.00	1338.24	192.00	2732.52
220^3	513.43	138.18	1781.89	255.55	3616.60
240^3	666.32	163.59	2314.14	331.78	4676.44
260^3	846.89	193.61	2943.03	421.82	5927.98
280^3	1057.46	228.62	3676.65	526.85	7387.16
300^3	1300.32	269.00	4523.04	648.00	9069.92
320^3	1577.78	315.14	5490.28	786.43	10992.19
340^3	1892.14	367.43	6586.43	943.30	13169.90
360^3	2245.71	426.25	7819.55	1119.74	15619.00
380^3	2640.79	491.98	9197.70	1316.93	18355.41
400^3	3079.68	565.00	10728.96	1536.00	21395.08

Appendix B. Multi-Threaded Results

The following figures illustrate the multi-threaded results over all thread counts and LDC mesh sizes. Each strategy of JuNe is compared to the fused counterpart of NeoN.

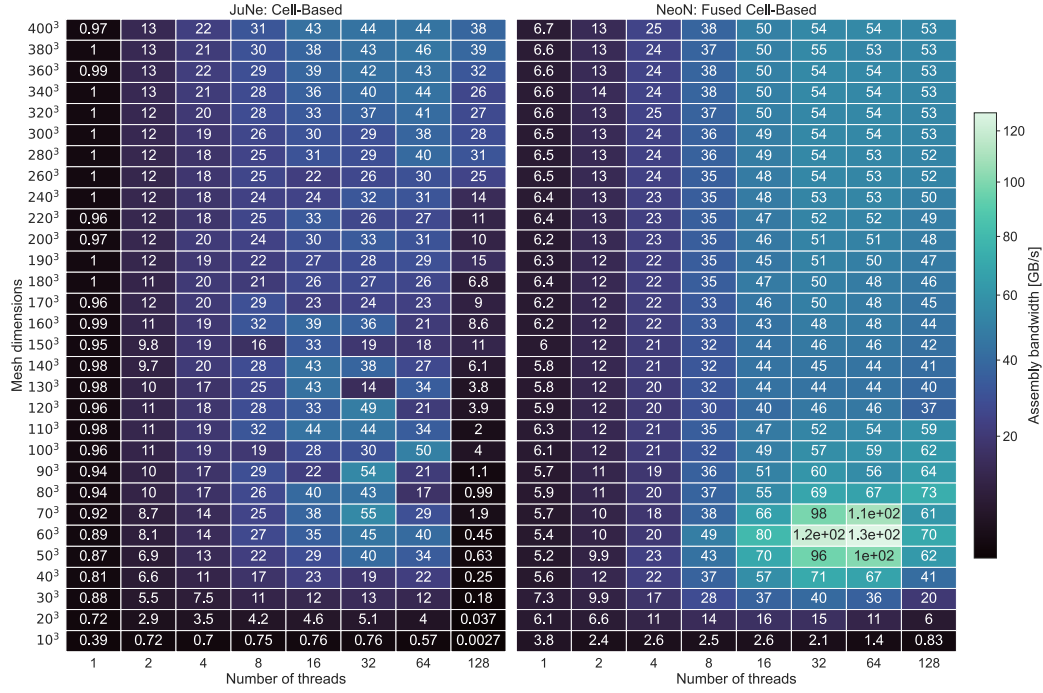


Figure 2.1.: JuNe cell-based and NeoN fused cell-based.

B. Multi-Threaded Results

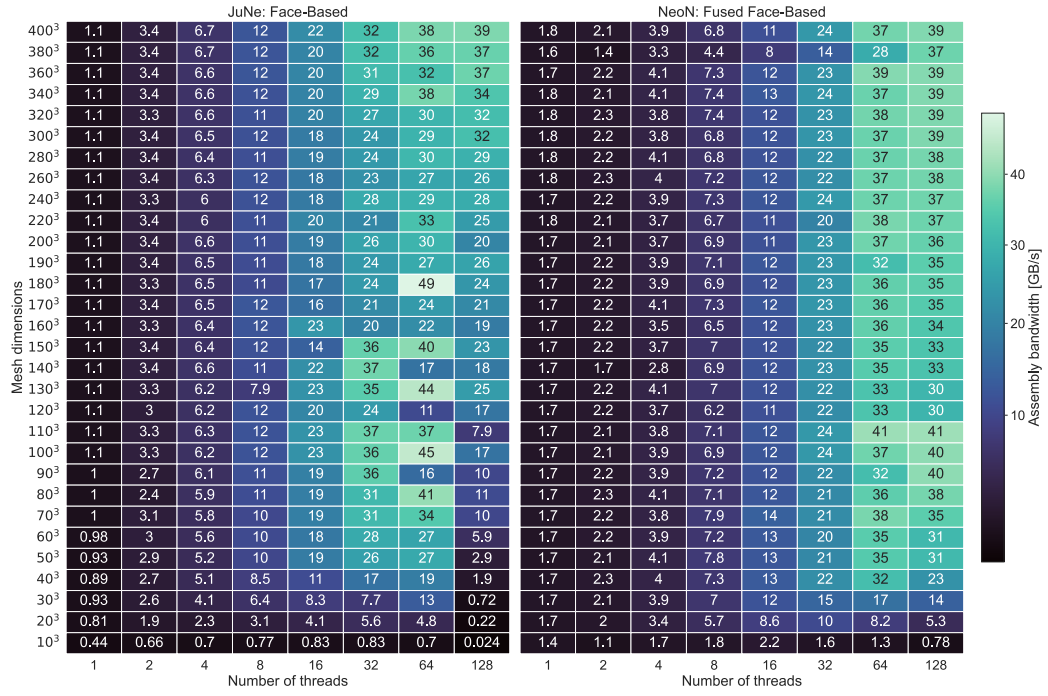


Figure 2.2.: JuNe face-based and NeoN fused face-based.

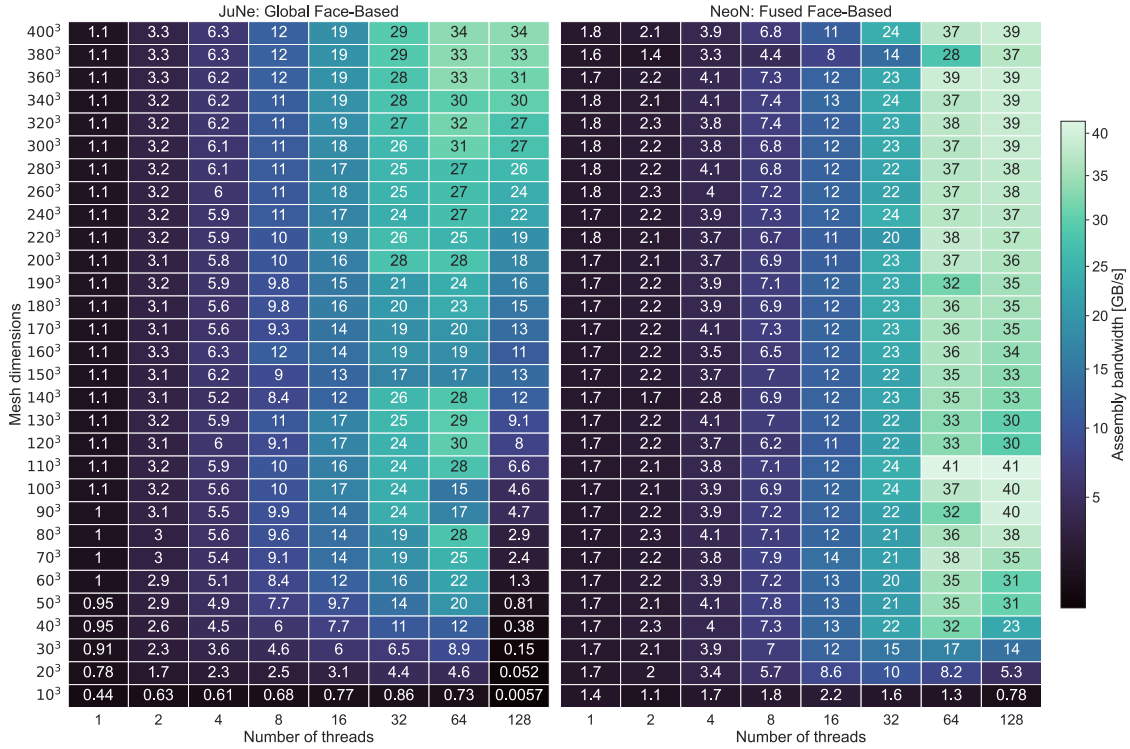


Figure 2.3.: JuNe global face-based and NeoN fused face-based.

Appendix C. Usage Of Gen. AI

Some parts of this work were accelerated through the usage of generative artificial intelligence, specifically GPT 5.5 model of OpenAI Codex. Let it be known that the scientific contributions were still performed by the author: all implementations were created without AI, nor did the author use it for writing this thesis. AI was used for three tasks:

- (1) Extending manual research tasks, e.g., searching the Julia source code for things that are poorly documented or performing some broad-scope research to fill any gaps manual research might have missed. For instance, after scouring usual sources like Google Scholar, the AI was given a task like “Find any scientific literature that comments on the topic of [...]”.
- (2) The generation of a few figures. The author believes that spending an inordinate amount of time generating figures is a waste and, given the time constraints of this thesis, is more reasonably invested into more important things. At no point did the AI create figures completely on its own; the author always gave very specific instructions on what to draw. Often, the process of using AI to create figures took multiple iterations. For instance, in Section 6.3.2 it is discussed that the difference between the GPUs is rather small, and that only the results for the H200 GPU are explicitly shown at that point. Further motivated to increase readability, the AI was given an array of instructions: “Create a second variant of the plot that only considers the H200 results. Also, move the legend to the right, outside the plot. Lastly, order the legend items according to the right-most values, i.e., from top to bottom: Face-Based (JuNe), Global Face-Based (JuNe), [...]” Follow-up instructions were minor formatting and placement modifications.
- (3) Performing reviews on chapter drafts in addition to reviews performed by the supervisors. The AI was specifically tasked to log every mistake and sort them into one of four categories:
 1. Syntax mistakes, e.g., typos, unfinished sentences
 2. Lack of clarity, e.g., unclear or ambiguous explanations
 3. Interrupted train of thought, e.g., jumping between topics or missing introductions of new topics
 4. Bad references, e.g., claims without references or links that point to the wrong section

Statement of Authorship

Thesis Title: Investigating Matrix Assembly for High-Performance CFD Codes

Name: Lukas Petermann

Date of Birth: 01.02.1999

Matriculation Number: 222902

I herewith assure that I wrote the present thesis independently, that the thesis has not been partially or fully submitted as graded academic work and that I have used no other means than the ones indicated. I have indicated all parts of the work in which sources are used according to their wording or to their meaning.

I am aware of the fact that violations of copyright can lead to injunctive relief and claims for damages of the author as well as a penalty by the law enforcement agency.

Furthermore, I confirm that I am aware that the use of content (including but not limited to text, figures, images and code) generated by artificial intelligence (AI) in the thesis must be disclosed. In those cases I have specified the AI system used, I have marked the specific sections of the thesis where AI-generated content was used, and I have provided a brief explanation of the level of detail at which the AI system was used to generate the content. I also stated the reason for using the tools.

I assure that even if a generative AI system has been used, the scientific contribution has been made entirely by myself.

Magdeburg, October 12, 2026

Signature