Sachbericht Teil II

zum Vorhaben

Exascale-fähige Softwarewerkzeuge zur Strömungssimulation im industriellen Design- und Optimierungsprozess - EXASIM Eingehende Darstellung

FKZ: 16ME0676K

Teilvorhaben	Arbeitsgruppe	Adresse	Ansprechperson
	TUM	TUM School of Computation Information and Technology Technische Universität München Bildungscampus 2 74076, Heilbronn	Prof. Dr. Hartwig Anzt hartwig.anzt@tum.de

Richtlinie: Neue Methoden und Technologien für das Exascale-Höchstleistungsrechnen (SCALEXA)

Koordination: Prof. Dr. Hartwig Anzt

Laufzeit: 36 Monate

Autoren: Prof. Dr. Hartwig Anzt, Dr. Gregor Olenik

Inhaltsverzeichnis

1. Einleitung

Im EXASIM-Projekt wurde die Entwicklung und Evaluierung exascale-fähiger Softwaretechnologien für industrielle Anwendungen der Computational Fluid Dynamics (CFD) verfolgt. Im Mittelpunkt stand dabei die Untersuchung potenzieller Leistungsgewinne, die moderne GPU-Hardware für ingenieurwissenschaftliche Strömungssimulationen ermöglichen kann. Innerhalb von EXASIM konzentrierte sich die Arbeit der KIT CSS/TUM COMA-Gruppe auf die Bereitstellung der notwendigen Infrastruktur zur Auslagerung linearer Gleichungslöser auf moderne GPU-basierte HPC-Hardware. Diese linearen Solver bilden eine wesentliche Voraussetzung für die von UCFD durchgeführten Leistungsanalysen industriell relevanter Anwendungsfälle auf Beschleunigerarchitekturen.

Aufgrund seines offenen Quellcodes (veröffentlicht unter der GNU General Public License) sowie seiner weiten Verbreitung in Industrie und Wissenschaft wurde der CPU-basierte Finite-Volumen-Code OpenFOAM® als Referenzplattform und Ausgangspunkt für die Implementierungsarbeiten gewählt. Der Code ist in der Lage, großskalige CFD-Simulationen auf mehreren tausend CPU-Kernen durchzuführen. Die Exascale-Fähigkeit des OpenFOAM-CPU-Codes konnte beispielsweise im EU-geförderten Projekt exaFOAM (2021–2024) demonstriert werden. Zu Beginn des EXASIM-Projekts im Jahr 2022 existierte jedoch noch keine technisch ausgereifte Implementierung für GPU-Architekturen.

Grundsätzlich lassen sich drei architektonische Strategien zur Integration von GPU-Unterstützung in CFD-Codes wie OpenFOAM unterscheiden:

1. Vollständige Refaktorierung bzw. Neuentwicklung bestehender Solver im Sinne einer GPU-first-Architektur.
2. Direktes Portieren bestehender CPU-basierter Solver zur Ausführung auf GPUs ohne grundlegende Änderungen an Code oder Datenstrukturen.
3. Hybrider Ansatz, bei dem ausschließlich besonders rechenintensive Komponenten – insbesondere die linearen Matrix-Solver – auf die GPU ausgelagert werden, während

der verbleibende Programmcode weiterhin auf CPUs ausgeführt wird und nur minimale Änderungen an der bestehenden Codebasis erforderlich sind.

Diese Ansätze unterscheiden sich deutlich hinsichtlich des zu erwartenden Leistungsgewinns und des erforderlichen Implementierungsaufwands. Im Allgemeinen nehmen sowohl die potenziellen Leistungssteigerungen als auch die Implementierungskomplexität von Strategie 1 zu 3 ab.

Der Implementierungsaufwand ist insbesondere vor dem Hintergrund der Größe der OpenFOAM-Codebasis zu bewerten, die etwa 1,5 Millionen Codezeilen umfasst. Eine vollständige Refaktorisierung im Sinne von Strategie 1 würde daher erhebliche personelle und zeitliche Ressourcen erfordern. Zu Projektbeginn wurde als grobe Abschätzung angenommen, dass die Implementierung einer einzelnen Codezeile in einem etablierten Softwareprojekt Kosten von bis zu 100 Euro verursachen kann. Vor diesem Hintergrund erschien eine vollständige Neuentwicklung wirtschaftlich nicht realisierbar.

Strategie 1 würde nicht nur eine Portierung der eigentlichen CFD-Solver auf GPUs erfordern, sondern auch die der zugehörigen Infrastruktur, beispielsweise Randbedingungen, physikalischer Modelle, benutzerdefinierter Funktionen sowie Ein- und Ausgaberoutinen, um eine industrielle Nutzbarkeit sicherzustellen. Zudem ist die Abschätzung der tatsächlich erreichbaren GPU-Leistung für die verschiedenen Strategien grundsätzlich mit Unsicherheiten behaftet. Der hybride Ansatz (Strategie 3) bietet demgegenüber bereits die Möglichkeit, signifikante Leistungsgewinne gegenüber der rein CPU-basierten Implementierung zu erzielen, während gleichzeitig der Großteil der bestehenden Codebasis erhalten bleibt. Darüber hinaus stellt Strategie 3 den am wenigsten invasiven Ansatz dar, um bewährte Vorgehensweisen und etablierte Arbeitsabläufe der bestehenden CFD-Codebasis beizubehalten. Dadurch bleiben vorhandene Simulationssetups und Workflows, die bereits produktiv in industriellen Anwendungen eingesetzt werden, weiterhin nutzbar. Auf Grundlage dieser praktischen Überlegungen entschieden sich die EXASIM-Projektpartner zu Projektbeginn für die Verfolgung des hybriden Ansatzes gemäß Strategie 3. Konkret wurde der OpenFOAM-CPU-Code mit der externen Bibliothek Ginkgo gekoppelt, um die linearen Matrix-Solver auf GPUs auslagern zu können. Zur Kopplung der CPU- und GPU-Komponenten wurde im Rahmen von EXASIM von Olenik et al. [1],[2],[3] ein OpenFOAM-Plug-in mit der Bezeichnung OpenFOAM-Ginkgo-Layer (OGL) entwickelt.

Obwohl dieser hybride Ansatz eine praktikable Möglichkeit zur Nutzung von GPU-Beschleunigung zumindest für Teile des Codes bietet, sind die erzielbaren Leistungsgewinne voraussichtlich geringer als bei einer vollständigen GPU-first-Implementierung (Strategie 1). Das primäre Ziel der vorliegenden Untersuchung besteht daher darin, die Bedingungen zu identifizieren und zu quantifizieren, unter denen der hybride Ansatz tatsächlich signifikante Leistungsverbesserungen ermöglicht, sowie seine praktische Eignung für industrielle CFD-Workflows zu bewerten. Weiterhin sollen potenzielle Anwendungsfälle identifiziert werden, in denen eine vollständige GPU-Portierung erforderlich wäre, um eine leistungsfähige Alternative zur CPU-basierten Implementierung darzustellen. Aufgrund der erwarteten Leistungseinschränkungen bei Simulationsfällen, in denen die Assemblierung der linearen Systemmatrix einen wesentlichen Anteil der Rechenzeit einnimmt, wurde ergänzend eine experimentelle domänenspezifische Sprache (Domain-Specific Language, DSL) mit der Bezeichnung NeoN entwickelt. Ziel dieser Entwicklung ist die Auslagerung der Matrixassemblierung auf GPUs bei möglichst einfacher Integration in bestehende Strömungssolver. Die Funktionsfähigkeit dieses Ansatzes wurde exemplarisch im Demonstrator NeoFOAM aufgezeigt. Alle entwickelten Bibliotheken sind Quelloffen und stehen zur freien Verfügung unter https://github.com/exasim-project und https://github.com/hpsim.

2. Methodik

Im Folgenden wird der im Projekt verwendete Ansatz zur Beschleunigung von CFD-Simulationen mit OpenFOAM detailliert beschrieben.

2.1. Auslagerung mittels des OpenFOAM-Ginkgo-Layers (OGL)

2.1.1. Allgemeiner Ansatz

OpenFOAM (Open Field Operation And Manipulation) ist ein in C++ implementiertes Framework zur Entwicklung numerischer Solver sowie von Pre- und Post-Processing-Werkzeugen für Probleme der Kontinuumsmechanik, insbesondere der Computational Fluid Dynamics (CFD) [4]. Obwohl verschiedene Ansätze zur GPU-Unterstützung für OpenFOAM untersucht wurden, existiert bislang kein allgemein akzeptierter Ansatz [3]. Eine Möglichkeit besteht darin, bestimmte Komponenten – beispielsweise den linearen Solver – über dedizierte Plug-ins auf GPUs auszulagern. Diese Plug-ins sind von OpenFOAM getrennt implementiert. Die jeweiligen Bibliotheken werden zur Laufzeit geladen und stellen zusätzliche Funktionalität bereit. OpenFOAM interagiert somit über eine lose Kopplung mit den Plug-ins, während diese intern eine enge Kopplung verwenden. Die OpenFOAM Ginkgo Layer (OGL) [3][1] ist eines dieser Plug-ins und stellt lineare Solver aus der Ginkgo-Bibliothek bereit. Die Funktionalität von OGL lässt sich in drei Hauptkomponenten unterteilen:

1. Abbildung zwischen den Datenstrukturen von OpenFOAM und Ginkgo, insbesondere für dünnbesetzte Matrizen.
2. Sicherstellung der Persistenz der Ginkgo-Datenstrukturen über die gesamte Simulation.
3. Laufzeitkonfiguration von Ginkgo-Solvern und Präkonditionierern über ogl.

Damit ermöglicht OGL die effiziente Nutzung der plattformportablen Solver von Ginkgo innerhalb von OpenFOAM. Die Bibliothek umfasst etwa 7000 Zeilen C++-Code. Der größte Teil entfällt derzeit auf die Verwaltung persistenter Objekte sowie auf Mechanismen zur Repartitionierung. Dies erfordert detaillierte Kenntnisse der zugrunde liegenden Objektverwaltung von Ginkgo, die über die Ginkgo-API direkt zugänglich ist. Ohne eine enge Kopplung wären deutlich komplexere Schnittstellen und Klassenhierarchien erforderlich, um die Speicherorganisation verschiedener Bibliotheken dynamisch abzubilden. Insgesamt bleibt die Typhierarchie daher relativ flach, da nur ein geringer Grad an Abstraktion notwendig ist. Dieser Abschnitt beschreibt den Offloading-Ansatz mittels der OGL-Bibliothek zur Nutzung der Ginkgo-Solver in OpenFOAM. Darüber hinaus werden einige der damit verbundenen Herausforderungen diskutiert und mögliche Maßnahmen zu deren Begrenzung vorgestellt. Dazu zählen insbesondere die Anbindung an Ginkgo zur Gewährleistung der Plattformportabilität, die Nutzung GPU persistenter Datenstrukturen zur Reduzierung des Datentransfers sowie eine effiziente Implementierung des Abbruchkriteriums, um unnötige Berechnungen der skalierten L1-Norm zu vermeiden.

2.1.2. Prozedur zur GPU-Auslagerung (Offloading)

Nach der Initialisierung der Solverklasse aus dem Plug-in werden folgende zentrale Schritte ausgeführt:

1. Zunächst wird die OpenFOAM-Systemmatrix vom LDU-Format in ein für GPUs geeignetes Format konvertiert. In OGL kann dies das Coordinate-List-Format (COO), das Compressed-Sparse-Row-Format (CSR) oder das Ellpack-Format (ELL) sein.
2. Da das Besetzungsstruktur der Systemmatrix während einer Simulation in der Regel konstant bleibt, werden die Zeilen- und Spaltenindizes auf der GPU persistent gespei-

[1] github.com/hpsim/OGL/

chert und lediglich die Matrixwerte aktualisiert. Hierzu ist eine Abbildung zwischen den Einträgen der LDU-Matrix und den COO- bzw. CSR-Einträgen erforderlich.
3. Zusätzlich werden die rechte Seite der Gleichung sowie der Lösungsvektor auf das Gerät übertragen. Der Lösungsvektor wird nach dem Lösungsschritt an OpenFOAM zurückgegeben und gleichzeitig als Initialschätzung für den nächsten Zeitschritt wiederverwendet.
4. Anschließend werden Solver- und Präkonditioner-Objekte aus der Ginkgo-Bibliothek erzeugt.
5. Schließlich wird der entsprechende lineare Solver aufgerufen und der berechnete Lösungsvektor an OpenFOAM zurückgegeben.

2.1.3. Überblick über die Ginkgo-Lineare-Algebra-Bibliothek

Der hier vorgestellte Offloading-Ansatz basiert auf mehreren zentralen Eigenschaften der Ginkgo-Bibliothek[2] [5]. Ginkgo ist eine moderne C++-Bibliothek für lineare Algebra mit Schwerpunkt auf dünnbesetzten Matrizen und heterogenen Rechnerarchitekturen. Hardwarespezifische Kernel werden in nativen Programmiersprachen implementiert, darunter CUDA (für NVIDIA-GPUs), HIP (für AMD-GPUs), OpenMP (für Mehrkernprozessoren von Intel, AMD oder ARM) sowie SYCL (für Intel-GPUs). Die Bibliothek stellt eine Vielzahl leistungsfähiger Solver- und Präkonditionierer-Algorithmen bereit, die sich für CFD-Simulationen eignen. Darüber hinaus ist Ginkgo ein attraktiver Backend-Kandidat für OpenFOAM, da die Bibliothek Open Source ist und unter der permissiven BSD-3-Clause-Lizenz veröffentlicht wird. Außerdem ist sie Teil des Extreme-scale Scientific Software Development Kit (xSDK) [6] und wurde bereits als Backend in verschiedene Simulationsbibliotheken integriert, darunter deal.ii [7], mfem [8] sowie openCARP [9], [10]. Dies bedeutet zum einen, dass Ginkgo auf vielen großen HPC-Systemen mit xSDK-Unterstützung bereits vorinstalliert ist. Zum anderen kann die Technologie auch in kommerziellen OpenFOAM-Projekten eingesetzt werden. Innerhalb von OGL werden insbesondere folgende zentrale Datenstrukturen von Ginkgo verwendet:

1. Ginkgo-Container-Typen wie `gko::array` und `gko::vector` zur Verwaltung des Datentransfers zwischen den hostbasierten Datenstrukturen von OpenFOAM und den GPU-basierten Daten. Der Typ `gko::vector` wird dabei zur Darstellung verteilter Vektoren in MPI-parallelisierten Simulationen verwendet.
2. Ginkgo-Executor-Klassen zur Abstraktion der Speicheradresse und der verwendeten GPU.
3. Solver- und Präkonditionierer-Klassen aus Ginkgo, die effiziente GPU-orientierte Implementierungen von Algorithmen zur Lösung und Vorkonditionierung linearer Gleichungssysteme bereitstellen.

2.1.4. GPU-Speicher Persistente Datenstrukturen

Ein zentrales Prinzip zur Leistungssteigerung beim Offloading besteht darin, die Kommunikation zwischen CPU-seitigem Speicher (im Folgenden Host) und Beschleuniger (GPU) möglichst gering zu halten. Zur Lösung einer linearen Gleichung

$$Ax = b \tag{1}$$

auf eine dedizierten GPU ergibt sich die zu übertragende Datenmenge für die Systemmatrix zu

$$m_{\mathrm{mat}} = (s_{\mathrm{scalar}} + 2s_{\mathrm{label}})n_{\mathrm{NNZ}} \tag{2}$$

sowie für die Vektoren zu

$$m_{\mathrm{vec}} = 3n_{\mathrm{DOF}}s_{\mathrm{scalar}} \tag{3}$$

[2] github.com/ginkgo-project/ginkgo

Dabei bezeichnen m_{mat} und m_{vec} die übertragenen Speichergrößen in Byte für Matrix bzw. Vektoren. s_{scalar} und s_{label} entsprechen der Größe eines Skalarwerts bzw. des Ganzzahlen-Datentyps in einfacher Genauigkeit. Die Anzahl der Nichtnull-Einträge der Matrix wird durch n_{NNZ} beschrieben, während n_{DOF} die Anzahl der Freiheitsgrade bzw. Rechengitterzellen angibt. Zu beachten ist, dass Gleichung 3 Drei Übertragungen von Skalararrays der Länge n_{DOF} berücksichtigt: Zunächst werden x und b vom Host auf das Gerät kopiert; nach der Lösung des Gleichungssystems muss der Lösungsvektor x wieder auf den Host zurückübertragen werden. In typischen CFD-Anwendungen wird das lineare Gleichungssystem aus Gl. Gleichung 1 jedoch mehrfach innerhalb einer Simulation gelöst, beispielsweise bei instationären Simulationen während der Zeitintegration oder bei stationären Simulationen innerhalb äußerer Iterationsschleifen. Bleibt das Rechengitter während der Simulation unverändert, kann die Datenübertragung zwischen Host und Gerät reduziert werden, indem die zuvor auf das Gerät übertragene Matrix lediglich aktualisiert und der Lösungsvektor als Initialschätzung für den nächsten Aufruf des Gleichungslösers wiederverwendet wird.

$$m_{\mathrm{mat}} = (s_{\mathrm{scalar}} + 2s_{\mathrm{label}})n_{\mathrm{NNZ}} \tag{4}$$

$$m_{\mathrm{vec}} = 2n_{\mathrm{DOF}}s_{\mathrm{scalar}} \tag{5}$$

Für Skalarwerte in doppelter Präzession (double precission) von 8 Byte sowie Labelgrößen von 4 Byte und unter der Annahme eines typischen Verhältnisses von $\frac{n_{\mathrm{NNZ}}}{n_{\mathrm{DOF}}} = 7$[3] ergibt sich ein Verhältnis von etwa 0.52 zwischen vollständiger Datenübertragung bei jedem Aufruf des Gleichungslösers und der Wiederverwendung bereits vorhandener Daten auf dem Gerät. Dadurch kann nahezu die Hälfte des Datentransfers eingespart werden. Weitere Optimierungen wären beispielsweise durch Ausnutzung von Matrixsymmetrien möglich. Verfahren wie zur Datenkompression sind denkbar werden jedoch im Rahmen dieser Arbeit nicht weiter betrachtet. Die Wiederverwendung von Systemmatrix und Vektoren auf der GPU über mehrere Aufrufe des Gleichungslösers hinweg erfordert zusätzliche Funktionalität in OpenFOAM. Da das Objekt `lduMatrixSolver` nach Abschluss eines Gleichungslösers freigegeben wird, würden ohne zusätzliche Maßnahmen auch alle auf der GPU erzeugten Datenstrukturen gelöscht. Zur Vermeidung dieser kostspieligen Neuberechnungen wird das OpenFOAM-Objektregister verwendet, um Smart Pointer auf Ginkgo-Datenstrukturen zu speichern, die Daten auf dem Beschleuniger enthalten. Solange sich der Smart Pointer im Objektregister befindet und seine Referenzzählung nicht auf null sinkt, bleiben die Daten auf dem Gerät erhalten und können bei späteren Gleichungslöseraufrufen wiederverwendet werden. Der vollständige Ablauf für eine GPU-persistente Ginkgo-Matrix ist in Abbildung 1 dargestellt.

[3] Basierend auf einem 3D-Stencil mit sechs Nachbarzellen.

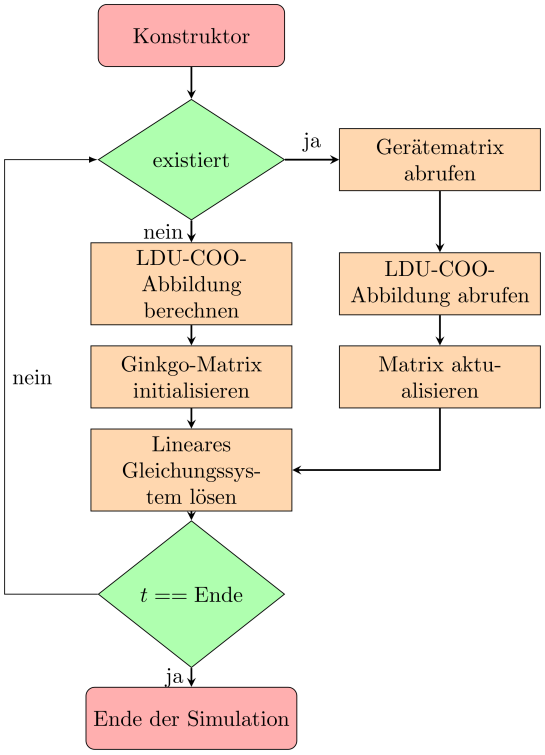

Abbildung 1: Ablauf zur Initialisierung oder Wiederverwendung persistenter Matrizen.

Beim Aufruf des Konstruktors einer entsprechenden Klasse wird zunächst geprüft, ob ein Objekt mit dem gewünschten Namen bereits im Speicher (mittels OpenFOAMs ObjectRegistry) vorhanden ist. Ist dies nicht der Fall, wird eine vollständige Initialisierung durchgeführt. Andernfalls erfolgt eine Aktualisierung unter Wiederverwendung der bestehenden Daten. Zur Verdeutlichung von Abbildung 1 sei ein typischer Ablauf mit mehreren linearen Löseraufrufen betrachtet. Beim ersten Auftreten existiert die persistente Matrix noch nicht, sodass der „nein“-Zweig gewählt wird. Die Initialisierung erfolgt in zwei Schritten. Zunächst wird eine Abbildung (mapping) zwischen den LDU-Koeffizienten von OpenFOAM und den Matrixkoeffizienten in zeilenweiser Anordnung auf dem Gerät berechnet. Anschließend kann die verteilte Ginkgo-Matrix erzeugt und gemeinsam mit allen notwendigen Datenstrukturen in der gespeichert werden. Die erstellte Matrix kann anschließend mit Ginkgo zur Lösung des linearen Systems verwendet werden. Bei nachfolgenden Aufruf des Gleichungslösers existiert das Objekt bereits, sodass der „ja“-Zweig gewählt wird. Die Datenstrukturen werden aus dem der abgerufen und die Matrixkoeffizienten mithilfe der gespeicherten LDU-COO-Abbildung aktualisiert.

2.1.5. Effiziente Implementierung des Abbruchkriteriums

Ein potenziell erheblicher Overhead, insbesondere bei Krylov-Lösern mit Jacobi-Präkonditionierern, kann durch die Auswertung des Abbruchkriteriums in jeder Iteration entstehen. In OpenFOAM basiert dieses auf der Berechnung einer skalierten L1-Norm [11]. Erfolgt diese Berechnung auf der CPU, muss in jeder Solveriteration der vollständige Residuumsvektor von der GPU in den CPU-seitigen Speicher übertragen werden. Daher ist eine dedizierte GPU-Implementierung der skalierten L1-Norm erforderlich, um ein identisches Konvergenzverhalten ohne wiederholte Datenübertragungen zwischen GPU und CPU sicherzustellen. Gleichzeitig sollte der Aufwand für die Berechnung der Residuumsnorm möglichst gering gehalten werden. Ist bekannt, dass eine Mindestanzahl an Iterationen zur Lösung eines bestimmten linearen Problems erforderlich ist, dient eine frühzeitige Auswertung der Resi-

duumsnorm lediglich der Überwachung. Entsprechend kann die Anzahl der Auswertungen reduziert werden, indem die Frequenz der Residuumsnormberechnung verringert wird. Dies kann zwar zu zusätzlichen Solveriterationen führen, bis das Abbruchkriterium erneut geprüft wird, jedoch kann die Gesamtberechnungszeit dennoch sinken, wenn zusätzliche Iterationen günstiger sind als die wiederholte Berechnung der Residuumsnorm. Die optimale Frequenz der Auswertung der Residuumsnorm hängt von zwei Faktoren ab:

1. der Zeit zur Berechnung der Residuumsnorm,
2. den Kosten einer Solveriteration.

Zur Bestimmung der optimalen Frequenz f wird ein einfaches Kostenmodell verwendet. Dieses basiert auf der Anzahl der Iterationen im vorherigen Lösungsschritt, also $n = n^{\{t-1\}}$, sowie auf dem Verhältnis der Rechenkosten einer einzelnen Solveriteration zur Berechnung der skalierten L1-Norm.

2.1.6. Repartitionierung

Bei Simulationen deren Berechnungen auf mehreren Rechenknoten durchgeführt werden, ist es eine Herausforderung, eine optimale Partitionierung des Rechengebiets zu finden, bei der die Kosten der Matrixassemblierung auf der CPU und der Lösung des linearen Systems auf der GPU ausgewogen sind. Die Autoren von [12] bezeichnen dieses Problem als „Over- und Under-Subscription-Challenge“. Im Kontext einer OpenFOAM-Simulation mit einem GPU-Löser-Plug-in lässt sich dies wie folgt illustrieren: Bei Verwendung eines typischen HPC-Knoten mit zwei CPU-Sockets mit jeweils 64 Kernen sowie vier Beschleunigern sind gängige Zerlegungsstrategien eine Partitionierung in (i) $64 = N_{\mathrm{CPU}}$ oder (ii) $4 = N_{\mathrm{GPU}}$ Rechengebiete.

Fall (i) gewährleistet eine optimale Leistung für die Matrixassemblierung auf der CPU, die in der Regel von stärkerer Parallelisierung profitiert. Bei einer naiven Implementierung führt dies jedoch zu einer Überbelegung der GPUs mit $\frac{N_{\mathrm{CPU}}}{N_{\mathrm{GPU}}}$ MPI-Prozessen (Ranks) pro GPU. Insbesondere OpenMPI ist sensitiv bei einer Überbelegung von GPUs, was zu erheblichen Leistungseinbußen führen kann [13]. Zusätzlich entsteht ein Kommunikations-Overhead zwischen Prozessen, die auf demselben Gerät ausgeführt werden.

Fall (ii) vermeidet diese Überbelegung durch eine Partitionierung in N_{GPU} Rechengebiete. Dadurch wird außerdem die Kommunikation zwischen Prozessen auf unterschiedlichen GPUs reduziert, da weniger Zellen an Prozessorgrenzen liegen. Der Nachteil besteht jedoch darin, dass die Matrixassemblierung auf der CPU lediglich N_{GPU} CPU-Kerne nutzt, wodurch eine Unterauslastung der CPU entsteht.

Eine optimale Partitionierung könnte zwar anhand eines Kostenmodells bestimmt werden, das die Kosten der Matrixassemblierung, des linearen Lösers sowie die erwarteten Skalierungseffekte berücksichtigt, würde jedoch eine erhebliche Abweichung vom üblichen Workflow darstellen, bei dem alle verfügbaren Kerne genutzt werden. Innerhalb des EXASIM Projektes wurde daher ein Repartitionierungsansatz untersucht, der die Nachteile von Fall (i) reduziert, während möglichst viele CPU-Kerne genutzt werden. Im folgenden wird das zugrunde liegende Berechnungsverfahren sowie ein einfaches Kostenmodell vorgestellt. Darauf aufbauend wird die Erzeugung und Aktualisierung der verteilten Matrizen auf CPU- (Host) und GPU-Seite (Device) detailliert beschrieben.

2.1.7. Kostenmodel für verteilte heterogene Berechnungen

Das OpenFOAM-Framework implementiert mehrere CFD-Solver auf Basis der Finite-Volumen-Methode (FVM), die typischerweise auf segregierten Projektionsverfahren wie SIMPLE, PISO oder PIMPLE beruhen. Dabei werden lineare Gleichungssysteme für mehrere partielle Diffe-

rentialgleichungen – insbesondere Impuls- und Druckgleichungen – getrennt assembliert und iterativ gelöst.

Der Ablauf des OpenFOAM-Solvers icoFOAM ist in Abbildung 2 dargestellt. Innerhalb jedes Zeitschritts wird zunächst das Gleichungssystem für die Impulsgleichung assembliert und gelöst. Das Ergebnis dient als Prädiktor für das Geschwindigkeitsfeld und als Eingabe für die PISO-Prozedur, in der iterativ die Druckkorrekturgleichung assembliert und gelöst wird. Nach der letzten Iteration der PISO-Schleife wird die Geschwindigkeit durch Anwendung der Druckkorrektur angepasst, sodass ein konservatives Geschwindigkeitsfeld entsteht.

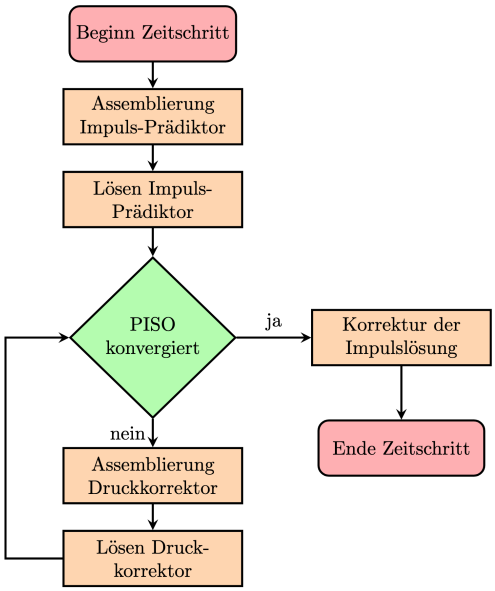

Abbildung 2: Ablaufdiagramm der zentralen Schritte innerhalb eines Zeitschritts im icoFOAM-Löser.

Der Solver icoFOAM dient hier als Modellsystem. Weitere Rechenkosten, etwa durch nicht-orthogonale Korrekturen, Ein-/Ausgabeoperationen oder zusätzliche physikalische Modelle, werden in dieser Betrachtung vernachlässigt. Die vorgeschlagene Vorgehensweise ist jedoch grundsätzlich auch auf andere Solver übertragbar. Unter Vernachlässigung zusätzlicher Kosten ergibt sich die Zeit zur Berechnung eines Zeitschritts als Summe aus Matrixassemblierung und Lösung des linearen Systems:

$$T(n) = T_{\mathrm{AS}}(n) + T_{\mathrm{LS}}(n), \tag{6}$$

wobei n die Anzahl der verwendeten MPI-Ranks bezeichnet.

Die beiden Terme können weiter in einzelne Gleichungen aufgeteilt werden, etwa für Impuls- und Druckgleichung. Im Folgenden wird jedoch angenommen, dass alle Gleichungen ein vergleichbares Skalierungsverhalten aufweisen. Der Beschleunigungsfaktor (Speed-up) der jeweiligen Komponenten ist definiert als $S_{\mathrm{LS}}(n) = T_{\mathrm{LS}}\frac{1}{T_{\mathrm{LS}}}(n)$ und analog $S_{\mathrm{AS}}(n)$. In einem heterogenen Szenario, also der gemeinsamen Verwendung von CPU und GPU-Ressourcen, in dem unterschiedliche Anzahlen an MPI-Ranks für Matrixassemblierung und lineare Lösung verwendet werden, unterscheiden sich diese Skalierungen, sodass gilt $S_{\mathrm{LS}} \neq S_{\mathrm{AS}}$. Dies entspricht dem hier betrachteten Szenario, in dem die Assemblierung auf CPUs erfolgt, während die Lösung des linearen Systems auf GPUs ausgelagert wird. Die maximalen Speed-ups werden für unterschiedliche Anzahlen an MPI Ranks erreicht: $N^*_{\mathrm{LS}} < N^*_{\mathrm{AS}}$ mit $S_{\mathrm{LS}}(n)l = S_{\mathrm{LS}}(N^*_{\mathrm{LS}})$ und $S_{\mathrm{AS}}(n)l = S_{\mathrm{AS}}(N^*_{\mathrm{AS}})$. Bei idealer Skalierung entsprechen diese optimalen Werte den maximal verfügbaren Ressourcen: $N^*_{\mathrm{AS}} = N_{\mathrm{CPU}}$ $N^*_{\mathrm{LS}} = N_{\mathrm{GPU}}$. Betrachtet man erneut die Gesamtlaufzeit

$$T(n) = T_{\mathrm{AS}}(n) + T_{\mathrm{LS}}(n) = T_{\mathrm{AS}}\frac{1}{S_{\mathrm{AS}}}(n) + T_{\mathrm{LS}}\frac{1}{S_{\mathrm{LS}}}(n) \tag{7}$$

so ergibt sich, dass weder $n = N^*_{\mathrm{AS}}$ noch $n = N^*_{\mathrm{LS}}$ die minimale Laufzeit liefern. Eine optimale Laufzeit erfordert stattdessen eine Anzahl an MPI Ranks n, bei der beide Komponenten suboptimale, aber insgesamt günstigere Speed-ups erreichen. Dies bedeutet, dass bei einer einheitlichen Partitionierung zwangsläufig ein Teil der Ressourcen ungenutzt bleibt. Werden jedoch unterschiedliche Anzahl an MPI-Ranks für Assemblierung und lineare Lösung gewählt, können beide Komponenten optimal ausgelastet werden. In diesem Fall ergibt sich

$$T(n_{\mathrm{AS}}, n_{\mathrm{LS}}) = T_{\mathrm{AS}}(n_{\mathrm{AS}}) + T_{\mathrm{LS}}(n_{\mathrm{LS}}) + T_{R(n_{\mathrm{AS}}, n_{\mathrm{LS}})} \tag{8}$$

wobei T_R die Kommunikationskosten zwischen beiden MPI-Kommunikatoren beschreibt. Damit minimiert die Kombination $(N^*_{\mathrm{AS}}, N_{\mathrm{LS}}\, *) = (N_{\mathrm{CPU}}, N_{\mathrm{GPU}})$ die Kosten für Assemblierung und linearen Solver. Es bleibt lediglich der Kommunikationsaufwand T_R zu berücksichtigen. Gleichzeitig kann diese Strategie leicht in bestehende Workflows integriert werden, da vorhandene Zerlegungen in N_{CPU} weiterhin genutzt werden können.

Im Folgenden wird der Begriff **Repartitionierung** für den Prozess verwendet, bei dem eine Zuordnung von einer CPU-Partition, die zur Assemblierung des linearen Gleichungssystems verwendet wird, zu einer separaten GPU-Partition erfolgt, auf der das lineare Gleichungssystem gelöst wird. Jede Partition besteht aus einer Anzahl von Teilgebieten, denen jeweils ein MPI-Rank r zugeordnet ist. Die Anzahl der Teilgebiete in der CPU-Partition n_{CPU} wird durch die Gebietszerlegung bestimmt, die als Vorverarbeitungsschritt mit dem Hilfsprogramm `decomposePar`, einem Bestandteil des OpenFOAM-Frameworks, durchgeführt wird.

Im Gegensatz dazu hängt die Anzahl der Teilgebiete in der GPU-Partition n_{GPU} von einem Repartitionierungsfaktor $\alpha \geq 1$ ab und wird zur Laufzeit bestimmt als $n_{\mathrm{GPU}} = \frac{n_{\mathrm{CPU}}}{\alpha}$ wobei dieser Faktor die Anzahl der Ränge pro GPU definiert. Damit gilt für die Anzahl der Teilgebiete die Beziehung $n_{\mathrm{GPU}} \leq n_{\mathrm{CPU}}$. Um die Abbildung zwischen den beiden Partitionen zu ermöglichen, müssen Verbindungen zwischen den Rängen der CPU-Partition und den Prozessen auf der GPU-Partition definiert werden. Diese Verbindung legt fest, welche Freiheitsgrade (Degrees of Freedom, DOF) des linearen Gleichungssystems welchem Prozess zugeordnet sind. In dieser Arbeit wird eine blockweise Verteilung verwendet, bei der der GPU-Prozess k dieselben Freiheitsgrade besitzt wie die α CPU-MPI Prozesse $\{\alpha k, \alpha k + 1, ..., \alpha k + \alpha - 1\}$. Sowohl für die CPU- als auch für die GPU-Partition werden die Matrixkoeffizienten entsprechend ihren Kommunikationseigenschaften gespeichert. Das bedeutet, dass alle Matrixkoeffizienten, die ausschließlich auf lokale Komponenten wirken, in einer lokalen Matrix gespeichert werden, während Koeffizienten, die Daten von einem anderen Prozess benötigen, in einer separaten nichtlokalen Matrixdatenstruktur abgelegt werden. Eine zusätzliche Einschränkung ergibt sich aus der Tatsache, dass in OpenFOAM die Host-Matrix im LDU-Format gespeichert wird. Daher ist zusätzlich, wie bereits erwähnt, eine Abbildung zwischen den Matrixeinträgen im LDU-Format und den Einträgen des auf der GPU verwendeten Matrixformats (typischerweise CSR) erforderlich.

Das in dieser Arbeit untersuchte Repartitionierungsverfahren lässt sich wie folgt beschreiben:

1. Extraktion der Besetzungsstruktur (Sparsity Pattern) aus der Host-Matrix, einschließlich aller Kopplungsterme mit nichtlokalen Einträgen.
2. Übertragung der lokalen und nichtlokalen Besetzungsstrukturen von den CPU MPI Prozessen an die zugehörigen GPU-Ränge.
3. Auf den GPU-Rängen werden alle empfangenen lokalen Besetzungsstrukturen zu einer gemeinsamen lokalen Besetzungsstruktur zusammengeführt. Konkret fusioniert der GPU-

Prozess k die von den CPU-MPI-Prozessen $\{\alpha k, \alpha k+1, ..., \alpha k+\alpha-1\}$ empfangenen Strukturen. Empfangene nichtlokale Matrixelemente werden in die neue lokale Besetzungsstruktur integriert, sofern sich der ursprüngliche Kommunikationspartner nach der Repartitionierung auf derselben Partition befindet. Andernfalls werden sie in der nichtlokalen Matrix gespeichert.

Zur Illustration zeigt Abbildung 4 die ursprüngliche Struktur der verteilten LDU-Matrix auf dem Host (oben) sowie die daraus resultierende Matrix im COO-Format auf dem Beschleuniger nach der Repartitionierung (unten). Durch die Verringerung der Anzahl der Teilgebiete enthält die resultierende repartitionierte Matrix zudem weniger Koeffizienten in der nichtlokalen Schnittstellenmatrix. Der vorgestellte Ansatz wendet das Repartitionierungsverfahren zunächst auf die Besetzungsstruktur der Matrix an. Dadurch entsteht eine repartitionierte Matrixstruktur, ohne dass die zugehörigen Matrixkoeffizienten berechnet werden müssen. In einem separaten Schritt werden anschließend die Koeffizienten der repartitionierten Matrix auf Basis der Matrixwerte der Host-Matrix gesetzt. Aus Schritt (3) ergibt sich eine Abbildung zwischen der Ordnung der Host-Matrix im LDU-Format und der gewünschten Ordnung der Matrix auf dem Zielprozess, beispielsweise einer zeilenweise angeordneten COO-Matrix. Zur Aktualisierung der Matrixkoeffizienten sendet jeder MPI-Prozess seine Matrixwerte in einen zusammenhängenden Pufferspeicher an den entsprechenden GPU-Prozess. Der empfangende MPI-Prozess ordnet die Daten anschließend anhand der zuvor berechneten Abbildung in die repartitionierte Matrix auf der GPU ein. Der grundlegende Prozess der Repartitionierung der Matrixkoeffizienten ist in Abbildung 3 dargestellt. Der Repartitionierer kombiniert mehrere CPU-basierte LDU-Blöcke, indem sie auf einem entsprechenden GPU-MPI-Rank gesammelt werden. Wie zuvor beschrieben, erfolgt die Zuordnung der CPU- zu GPU-Ranks blockweise. Alternative Verfahren zur Verteilung von Matrixelementen, die potenziell optimalere Partitionierungen liefern könnten (z. B. METIS), sind grundsätzlich möglich. Zusätzlich ist der Repartitionierer dafür verantwortlich, nichtlokale Blöcke zu lokalisieren, deren Kommunikation nach der Repartitionierung vollständig lokal ist. Diese lassen sich identifizieren als Blöcke, bei denen der globale Spaltenindex j innerhalb der Menge der globalen Zeilenindizes $I_{\mathrm{GPU}}(r)$ der fusionierten Besetzungsstruktur auf dem GPU Prozess r liegt: $j \in I_{\mathrm{GPU}}(r) = \cup_{l=0}^{\alpha-1} I_{\mathrm{CPU}}(\alpha r + l)$ Der Repartitionierungsprozess erzeugt drei zentrale Datenstrukturen:

1. Die Besetzungsstruktur der repartitionierten Matrix, bestehend aus Zeilen- und Spaltenindizes.
2. Ein Aktualisierungsmuster U für die Matrixkoeffizienten. Dieses speichert für jeden MPI-Prozess (a) die Zielprozesse der Sendeoperation, (b) Zeiger (Pointer) auf die zu sendenden Host-Daten und die Zielpuffer sowie (c) die entsprechenden Größen der MPI-Kommunikation.
3. Eine Permutationsmatrix, die die ursprüngliche LDU-basierte Sortierung auf eine zeilenweise Sortierung der repartitionierten Gerätematrix abbildet.

Zur Realisierung der unterschiedlichen CPU- und GPU-Partitionen wird ein neuer MPI-Kommunikator erzeugt, indem der ursprüngliche CPU-Kommunikator C in aktive Ränge C_a und inaktive Ränge C_i aufgeteilt wird. Der Standardkommunikator C wird weiterhin für die Aktualisierung der Matrixkoeffizienten benötigt. Der Kommunikator C_a wird an den linearen Löser übergeben, der das verteilte Gleichungssystem behandelt, während die inaktiven Ränge C_i diesen Schritt überspringen. Der Kommunikator C_a wird anschließend ausschließlich auf den GPUs verwendet. Dieses Vorgehen verhindert die Erzeugung leerer Matrizen auf GPUs für nichtbesitzende Ränge, was andernfalls zu erheblichen Performanceeinbußen führen könnte. Um eine optimale Performance zu gewährleisten, werden die Erstellung und die Aktualisierung der Systemmatrix als zwei getrennte Prozesse behandelt. Dadurch kann

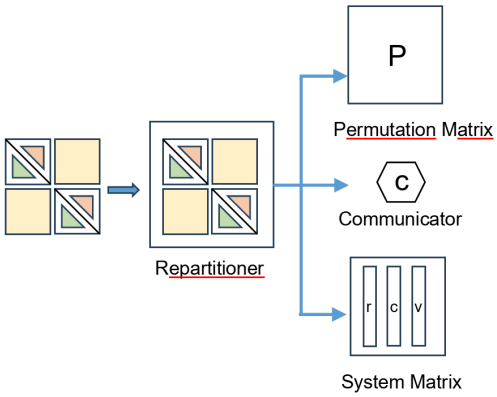

Abbildung 3: Illustration des Repartitionierungsverfahrens, das ein repartitioniertes Besetzungsstruktur, eine Datenstruktur zur Aktualisierung der Matrixkoeffizienten sowie eine Abbildung zwischen den Matrixindizes der LDU- und COO-Formate erzeugt.

die repartitionierte, verteilte Matrix einmal erzeugt und anschließend während der gesamten Simulation durch Aktualisierung ihrer Werte wiederverwendet werden. Der Aktualisierungsprozess der Matrixkoeffizienten ist in Abbildung 5 dargestellt. Falls GPU-aware MPI verfügbar ist, sendet jeder MPI-Prozess auf der CPU seine lokalen Daten direkt an eine Speicheradresse eines temporären Puffers auf der GPU. Andernfalls werden die Daten zunächst auf den CPU-MPI-Prozessen αk gesammelt und anschließend in einem separaten Schritt auf die GPU kopiert. Das Ergebnis ist ein Array, das entsprechend der ursprünglichen LDU-Reihenfolge sowie der Herkunftsränge angeordnet ist. In einem zweiten Schritt werden die Koeffizienten mithilfe der Permutationsmatrix P neu angeordnet, um die von der linearen Algebra-Bibliothek erwartete zeilenweise Ordnung zu erfüllen.

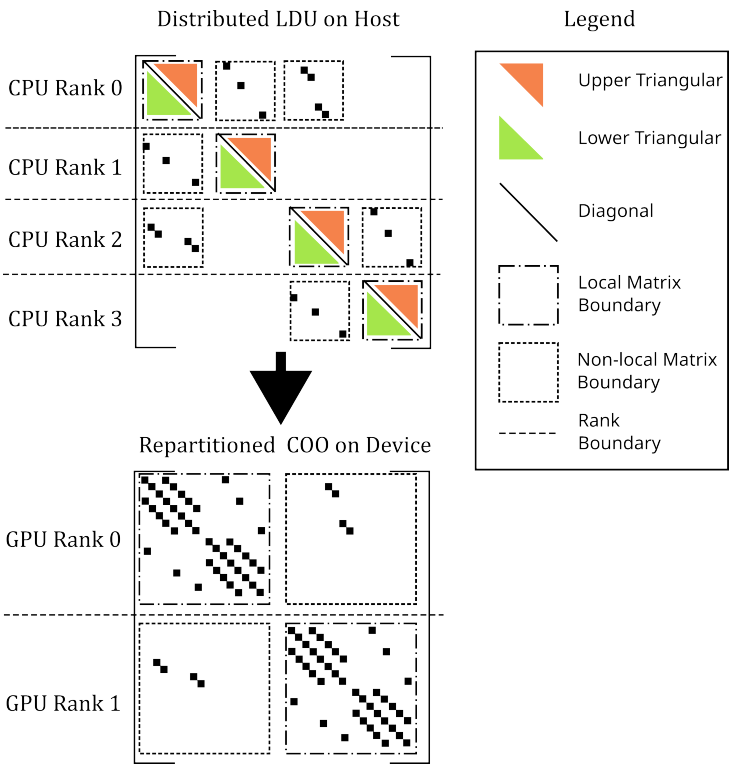

Abbildung 4: Struktur der verteilten Matrix im LDU-Format im CPU-Seitigen Speicher (oben) und nach der Repartitionierung auf dem Beschleuniger (unten), mit $\alpha = 2$.

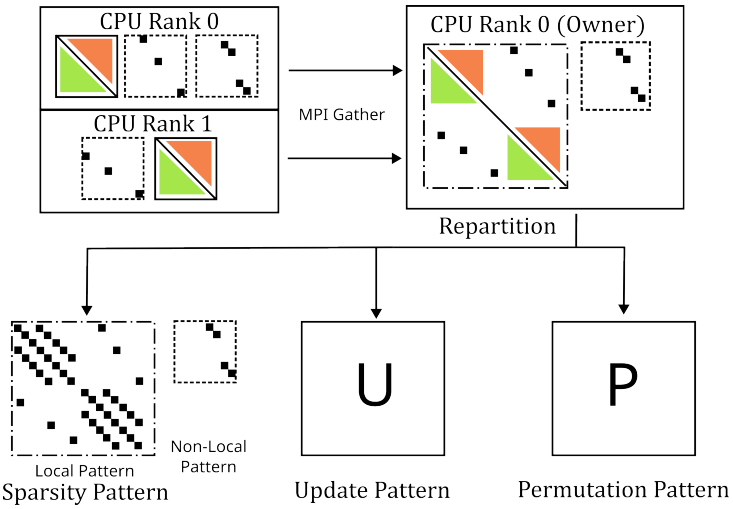

Abbildung 5: Repartitionierungsprozess mit erstellten Datenstrukturen Sparsity,Update und Permutation Pattern, für die nachfolgenden Updateschritte im Verlauf einer Simulation

Abbildung 6 zeigt die erreichte Rechenleistung am Beispiel des DrivAer-Testfalls unter Verwendung von Krylov-Lösern auf bis zu 48 GPUs. Wie zu erkennen ist, ist eine Leistungssteigerung von bis zu einem Faktor von 7 zwischen dem unrepartitionierten Fall (ranksPerGPU=1) und dem vollständig repartitionierten Fall (ranksPerGPU=18) zu beobachten. Gegenüber einer exklusiven Ausführung auf CPUs ist bei Verwendung von Krylov-Lösern ein Speedup von einem Faktor von 2-3 zu beobachten.

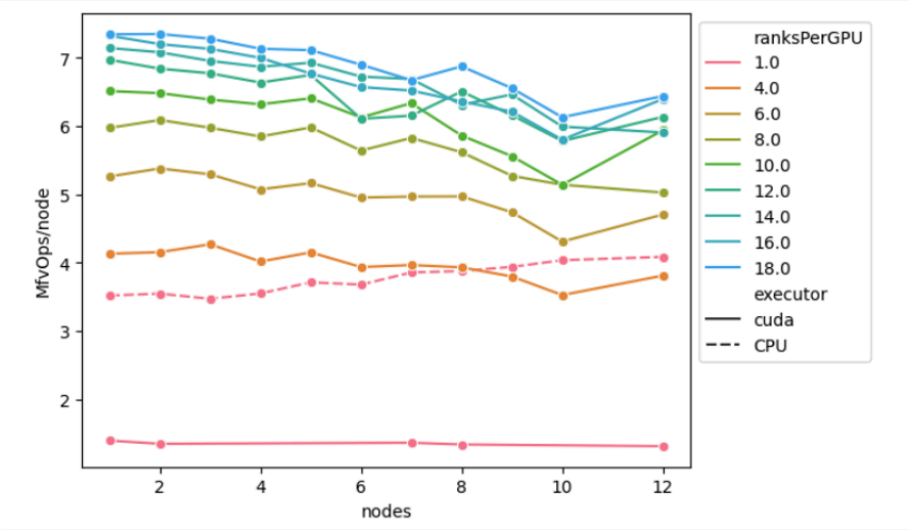

Abbildung 6: Einfluss der Repartitionierungsprozedur auf die Rechenleistung am Beispiel des DrivAer-Testfalls unter Verwendung von Krylov-Lösern.

2.2. Matrix Assemblierung in NeoN

NeoN[4] bedient den Bedarf nach einer vielseitigen CFD Bibliothek die GPU-native Datenstrukturen und Finite-Volumen-Operatorabstraktionen innerhalb eines einheitlichen Programmiermodells implementiert. Durch die Nutzung portabler Parallelisierungsbibliotheken wie Kokkos [5] sowie einer domänenspezifischen Schnittstelle für Diskretisierungsoperatoren ermöglicht NeoN die Entwicklung moderner CFD-Solver, wie im Demonstrator NeoFOAM [6] gezeigt. Diese Solver können effizient auf unterschiedlichen CPU-GPU-Architekturen ausgeführt werden, ohne an eine bestimmte Hardwareplattform gebunden zu sein oder mehrere Implementierungen in verschiedenen Programmiersprachen pflegen zu müssen. Gleichzeitig unterstützt NeoN eine schrittweise Modernisierung bestehender Finite-Volumen-Frameworks. Über begleitende Projekte wie NeoFOAM können auch bestehende Solver, einschließlich OpenFOAM-basierter Codes, GPU-native Datenstrukturen und performanceportable Operatorabstraktionen mit minimalem Refaktorisierungsaufwand nutzen.

NeoN ist als modulare CFD-Bibliothek konzipiert, die die Entwicklung von Finite-Volumen-Solvern mithilfe einer in C++ eingebetteten domänenspezifischen Sprache (DSL) ermöglicht. Der Entwurf adressiert typische Einschränkungen traditioneller Finite-Volumen-Infrastrukturen, bei denen Gleichungsterme häufig unmittelbar ausgewertet (eager evaluation) und direkt in dünnbesetzte Matrizen assembliert werden. Dies führt zu unnötigen temporären Datenstrukturen und erhöhter Nutzung der Speicherbandbreite. Zudem werden Matrizen z.B. in OpenFOAM in spezifischen Formaten wie LDU gespeichert, was die Interoperabilität mit externen linearen Solverbibliotheken erschwert. NeoN adressiert diese Probleme durch eine modulare, operatorbasierte DSL bei der Gleichungsterme nicht unmittelbar (lazy evaluation) ausgewertet werden. Dies hat den Vorteil, dass Gleichungen optimiert werden können, indem Gleichungsterm fusioniert werden. Weiterhin unterstützt NeoN die Assemblierung dünnbesetzter Matrizen in standardisierten Formaten wie COO und CSR. Dadurch wird die Integration externer Solver-Backends verbessert. Zur Lösung der resultierenden linearen Gleichungssysteme integriert NeoN die Bibliothek Ginkgo [4] und stellt damit skalierbare iterative Solver und Präkonditionierer für heterogene Architekturen bereit. Erste Leistungstests zeigen

[4] github.com/exasim-project/NeoN

vielversprechende Ergebnisse mit einer Geschwindigkeitsverbesserung der Assemblierung von linearen Gleichungssystemen um bis zu eine Größenordnung.

2.3. Research Software Engineering Aspects

Der folgende Abschnitt beschreibt die Automatisierung von Unit- und Integrationstests sowie Benchmarking für das OpenFOAM Ginkgo Layer[5] (OGL) [2] unter Verwendung der Softwarebibliothek OpenFOAM Benchmark Runner (OBR)[6] vgl. [14] und die Teststrategie für die NeoN-Bibliothek.

Eine zentrale Herausforderung beim Testen von OpenFOAM-Fällen für OGL besteht darin, sicherzustellen, dass sich Testfälle vor der Ausführung in einem konsistenten Zustand befinden. Dazu gehören beispielsweise die Gittererzeugung mittels , die Zerlegung des Gitters sowie das Entfernen generierter Dateien oder modifizierter Konfigurationseinträge. Dies ist insbesondere bei Benchmarks relevant, bei denen Gittergröße oder Zeitschritte innerhalb desselben Testfalls variiert werden und dadurch Änderungen an den auf der Festplatte gespeicherten Dateien erforderlich sind. Diese und ähnliche Probleme werden durch die im EXASIM Projekt entwickelte OBR-Software adressiert, [14]. Diese wurde auch im Rahmen von OpenFOAMs RSE SIG im April 2024 vorgestellt[7].

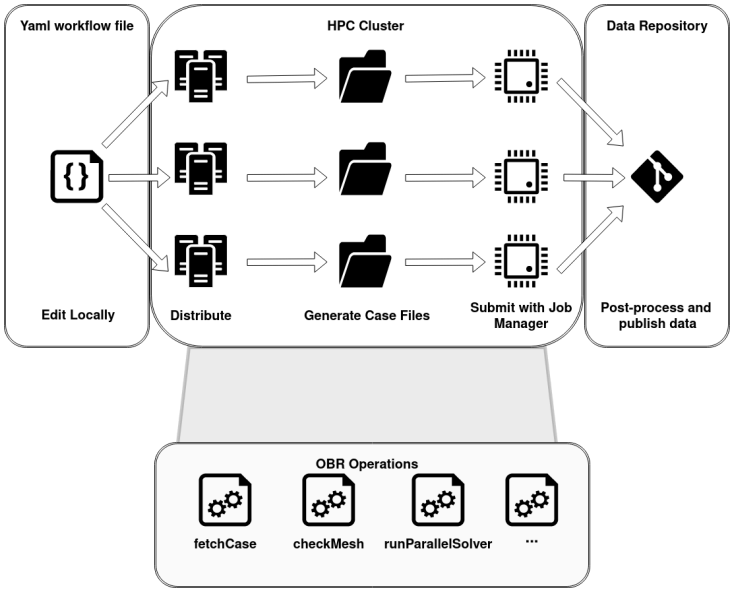

Abbildung 7: Wesentliche Schritte bei der Erstellung der Test und Benchmarkfälle mittels OBR

Zur Sicherstellung der Codequalität verwenden OGL und NeoN GitHub Actions[8] als zentrale Testumgebung zur Ausführung der Testläufe. Hier werden verschiedene Testarten ausgeführt, darunter statische Codeanalysen zur Sicherstellung korrekter Formatierung, Unit-Tests sowie Integrationstests. Die Testworkflows werden in YAML-Dateien im Verzeichnis `.github/workflows` definiert und typischerweise bei neuen Pull-Requests oder nach dem Push neuer Commits auf den GitHub-Server ausgeführt. Die CI/CD-Pipeline für OGL und NeoN besteht zunächst aus dem Kompilieren der Bibliothek mit verschiedenen Compilern wie GCC, Clang, MSCVC, sowie GPU-spezifischen Compilern wie nvcc, icpx etc. Erst nach erfolgreichem Build werden Integrationstests gestartet, die auch die Korrektheit mit OpenFOAM

[5] github.com/hpsim/OGL
[6] github.com/exasim-project/OBR
[7] https://wiki.openfoam.com/Research_Software_Engineering_Special_Interest_Group
[8] https://github.com/features/actions

als Referenzfall testen. OGL und NeoN verfügen über umfangreiche Dokumentation (siehe https://exasim-project.com/NeoN/latest/)

3. Vergleich zur ursprünglichen Vorhabenbeschreibung

Im Folgenden werden die Ergebnisse im Vergleich zu den ursprünglichen Zielen an deren Erfüllung die Arbeitsgruppe SCC/KIT bzw. COMA/TUM maßgeblich beteiligt war, diskutiert.

Ziel: Implementierung eines für industrielle Strömungssimulationen relevanten HPC-Workflows mit automatisierter Testfallsuite inklusive Simulationsgebiet- und Randbedingungsoptimierung
Ergebnis: Im Projekt wurde eine automatisierte Testfallsuite mit derzeit acht Benchmarkfällen aufgebaut und öffentlich bereitgestellt1. Zur Durchführung parametrischer Studien wurde das Orchestrierungsframework OpenFOAM Benchmark Runner (OBR2) entwickelt, das auf der signac-Toolbox basiert und eine reproduzierbare, automatisierte Ausführung von Simulationen auf HPC-Systemen einschließlich Post-Processing ermöglicht3. Teile des entwickelten Workflows wurden bereits produktiv für die Generierung des DrivAerML-Datensatzes genutzt4. Darüber hinaus wurden im Rahmen von OpenFOAM-Workshops mit dem Workflow generierte Performance-Studien präsentiert
Ziel: Das Erreichen von mindestens 0.1 TFLOP/s in den linearen Lösern auf einer Beschleunigerkarte und mindestens 2 TFLOP/s für eine auf mehrere Karten verteilte Simulation.
Ergebnis: Tests mit dem LidDrivenCavity-Benchmark zeigen, dass die Zielgrößen von 0.1 TFLOP/s (Single-GPU) und 2 TFLOP/s (Multi-GPU) in den linearen Lösern erreicht werden. Ergebnisse dieser Untersuchungen wurden unter anderem im Kontext einer Studie zu Repartitionierung auf der ISC präsentiert 6. Die Werte werden nur für Testfälle mit einer hohen Anzahl von Druckiterationen im linearen Löser erreicht. So erreicht der Impinging-Jet-Fall knapp 2 TFLOP/s bei der Ausführung auf 96 GPUs. Als Hauptmetrik für alle weiteren Performancevergleiche wurde die in der OpenFOAM-Community etablierte Kennzahl FVOPS (Finite Volume Operations per Second) von Galeazzo et al. [15] verwendet, da diese einfacher zu bestimmen und zwischen Fällen besser vergleichbar ist. Die gemessenen CPU-Ergebnisse für FVOPS/device stimmen mit den Untersuchungen von Galeazzo et al. [15] überein, während die GPU-Ergebnisse Verbesserungen um etwa eine Größenordnung zeigen.
Ziel: Eine Erhöhung der Rechenleistung des linearen Lösers auf einem einzelnen Rechenknotens unter Einsatz von bis zu acht GPUs um einen Faktor von mindestens 25 im Vergleich zum ausschließlichen Einsatz aller CPU-Kerne eines typischen HPC-Knotens mit 32 CPU-Kernen.
Ergebnis: Für den linearen Löser allein zeigen die Performanzmessungen sehr hohe Beschleunigungsfaktoren. Beispielsweise erreicht der ImpingingJet-Testfall einen Performanzfaktor von bis zu 85 beim Vergleich von 8 GPUs gegenüber 32 CPU-Kernen. Dieser Wert bezieht sich jedoch ausschließlich auf den linearen Löser und ist daher nur eingeschränkt für eine Bewertung der gesamten Kosteneffizienz einer CFD-Simulation geeignet. Für eine realistische Bewertung muss der Vergleich auf Basis eines vollständigen Zeitschritts erfolgen, der sowohl Matrix-Assemblierung als auch die Lösung der linearen Gleichungssysteme umfasst. Hierfür wird der Coefficient of Equivalence (COE) verwendet, der angibt, wie viele CPU-Kerne die Rechenleistung einer GPU für einen vollständigen Zeitschritt erreicht. Der Vergleich zwischen einer Konfiguration mit acht GPUs und einem typischen CPU-Knoten mit 32 Kernen ergibt sich dann direkt aus dem Ausdruck COE · 8 GPUs / 32 CPUs. Auf dieser

Grundlage erreicht der hybride OGL-Ansatz lediglich Beschleunigungsfaktoren im Bereich von etwa 3 bis 10. Erst vollständig GPU-native Implementierungen wie SPUMA oder NeoN erreichen Werte von über 25.
Ziel: Eine Steigerung der Energieeffizienz der linearen Löser um den Faktor zwei gegenüber CPU-basierten Lösern.
Ergebnis: Der Energieverbrauch einer der verwendeten A100-40GB GPUs ist mit etwa 400 W rund 57-mal höher als der eines einzelnen Intel Xeon Platinum 8368 CPU-Kerns (≈7 W). Um eine Verdopplung der Energieeffizienz zu erreichen, muss die GPU folglich eine mindestens 114-fach höhere Rechenleistung pro Kern erzielen. Analog zum vorherigen Punkt wird dieser Wert erreicht, wenn ausschließlich die linearen Löser betrachtet werden. Wird jedoch der gesamte Zeitschritt inklusive Matrix-Assemblierung berücksichtigt, kann der hybride OGL-Ansatz dieses Ziel nicht erreichen. Eine entsprechende Verbesserung der Energieeffizienz wird erst durch vollständig GPU-native Implementierungen erreicht, bei denen neben dem linearen Löser auch die Assemblierung auf der GPU erfolgt, wie beispielsweise bei SPUMA oder NeoN.
Ziel: Nachweis von starker Skalierung auf bis zu 64 GPUs und schwacher Skalierung auf bis zu 200 GPUs.
Ergebnis: Im Rahmen des H3 Workshops[9] auf der ISC 2025 wurden Ergebnisse bzgl. starker Skalierung auf bis zu 64 GPUs für den lidDrivenCavity3D, WindsorBody und DrivAer vorgestellt und in Form eines Konferenzpapers veröffentlicht [16]. Weiterhin wurde schwache Skalierung auf bis zu 16000 GPUs getestet. Bei ausreichender Saturierung der einzelnen GPUs ist eine parallele Effizienz von über 80% erreichbar.
Ziel: Verringerung der Rechenzeitkosten durch Verwendung von Algorithmen mit gemischter Präzisions; hier wird ein Speedup von zirka 1.1 erwartet.
Ergebnis: Die Verwendung von Algorithmen mit gemischter Präzision wurde in Kooperation mit Prof. E. Quintana-Ortí untersucht. Erste Ergebnisse für RISC-V wurden auf der Konferenz HPC Asia 2026[10] vorgestellt. Auf den getesteten Maschinen wurde ein Speedup von ca 1.5 erreicht.
Ziel Im Vergleich zu petsc4FOAM (Stand September 2022) erwarten wir eine deutliche Verringerung des Overheads von 30% bei der Auslagerung der Berechnungen und eine Verbesserung der Performanz ausgelagerter linearer Löser um zirka 10% sowie eine bessere Einhaltung der OpenFOAM-Standards bezüglich Logging und Löserparametereinstellungen.
Ergebnis: Erste Vergleichsstudien zwischen OpenFOAM mit petsc4FOAM und OGL mit Ginkgo wurden im Jahr 2023 durchgeführt. Zunächst zeigte sich eine vergleichbare Performanz der Ansätze. Durch die Implementierung verschiedener Optimierungen, darunter effizientere Abbruchkriterien für lineare Löser, GPU-persistente Datenstrukturen sowie Repartitionierungsschemata, konnte die Performanz anschließend im Jahr 2025 um einen Faktor von etwa 3–6 gesteigert werden.
Ziel: Eine Verbesserung der Lastverteilung zwischen der Druck- und der Impuls- sowie Skalartransportgleichungen durch den Einsatz von hybrider OpenMP/MPI Parallelisierung oder durch verbesserte Partitionierung der Teilrechengebiete einer Simulation im Vergleich zu

[9] https://icl.utk.edu/~luszczek/conf/2025/h3/
[10] https://riscv.epcc.ed.ac.uk/community/workshops/hpcasia26-workshop/

herkömmlichen OpenFOAM Variante eine Speedup um den Faktor von 1.25. (Auf einem Knoten mit 8 GPUs)
Ergebnis: Die implementierten Repartitionierungsverfahren für heterogene CPU/GPU-Architekturen verbessern die Lastverteilung zwischen CPU und GPU deutlich. Insbesondere Simulationsfälle mit hohem Anteil an Matrixassemblierung profitieren von diesem Ansatz. Im Idealfall wurden Beschleunigungen von etwa Faktor 2 für den LidDrivenCavity3D-Fall und bis zu Faktor 7 für den DrivAer-Fall im Vergleich zur nicht repartitionierten Variante gemessen (vgl. [16]). Darüber hinaus wurden erste Tests zur hybriden MPI/OpenMP-Parallelisierung mit der NeoN-Bibliothek durchgeführt. Die bisherigen OpenMP-Ergebnisse zeigen eine gute Skalierbarkeit bis zu 16 Threads. Für eine abschließende Bewertung ist jedoch noch die Implementierung der MPI-Parallelisierung erforderlich.
Ziel: Verringerung des Bedarfs an Rechenknoten um einen Faktor 3 bei gleicher Turnaround-Zeit bzw. Beschleunigung um einen Faktor 3 bei gleicher Anzahl von Knoten.
Ergebnis: Auf dem verwendeten HoreKa-Cluster stehen pro Rechenknoten 76 CPU-Kerne beziehungsweise 4 GPUs zur Verfügung. Um die Ziele zu erfüllen, muss eine GPU die Rechenleistung von mindestens COE ≈ 3·76/4 ≈ 57 CPU-Kernen erreichen. Diese Zielgröße wird von dem hybriden OGL-Ansatz nicht erreicht. Erst vollständig GPU-native Implementierungen erreichen entsprechende COE-Werte und erfüllen damit dieses Projektziel.
Ziel: Eine starke Verbesserung der Konvergenzeigenschaften (bis zu 40% weniger Iterationen) der verteilten Krylov Löser durch Verwendung von speziellen parallelen Algorithmen. Hier wird ein Speedup von insgesamt zirka 1.5 des linearen Lösers erwartet.
Erste Untersuchungen zu Multi-Level-Schwarz- sowie globalen Multigrid-Präkonditionierern zeigten positive Effekte auf das Konvergenzverhalten der Solver. In einigen Fällen konnte die Anzahl der benötigten Iterationen etwa halbiert werden. Der erhöhte Aufwand für Aufbau und Kommunikation der Verfahren führte jedoch insgesamt zu keiner Verbesserung der Gesamtperformanz.

3.1. Wesentliche Leistungsdaten

Im Folgenden werden wesentliche Leistungsmerkmale des OGL Bibliothek dargestellt. Eine genauere Betrachtung erfolgt im Bericht des Projektpartners Upstream CFD der mit Leistungsbewertung betraut war. Abbildung 8 zeigt die erreichte Leistung in TFlops in den linearen Lösern für den LidDrivenCavity3D Fall für mehrere Auflösungen. Für Fälle ab einer Auflösung von 420^3 Gitterzellen sind Leistungen von $> 0.1\mathrm{TFlops}$ in der Spitze pro GPU messbar.

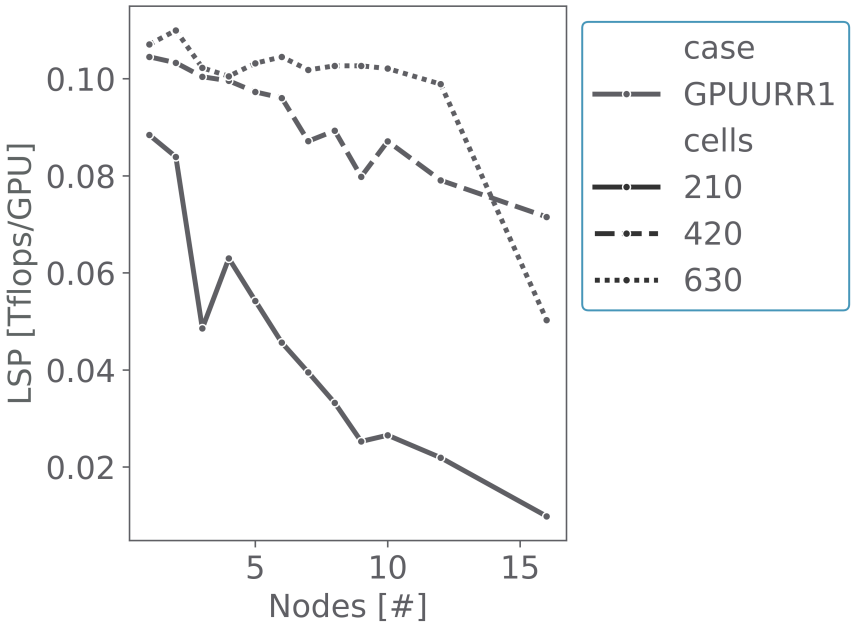

Abbildung 8: Performanz der linearen Löser pro GPU

Abbildung 9 zeigt die erzielte Gesamtleistung der linearen Löser für den LidDrivenCavity3D Fall bei einer Auflösung von 630^3 unter Verwendung verschiedener Partitionsschemata in einem starken Skalierungstest. Bei verteilten Rechnungen ab einer Anzahl von 5 Knoten (20GPUs) wurden mehr als 2TFlops und bei 16 Knoten (64GPUs) mehr als 5Flops Rechenleistung erreicht.

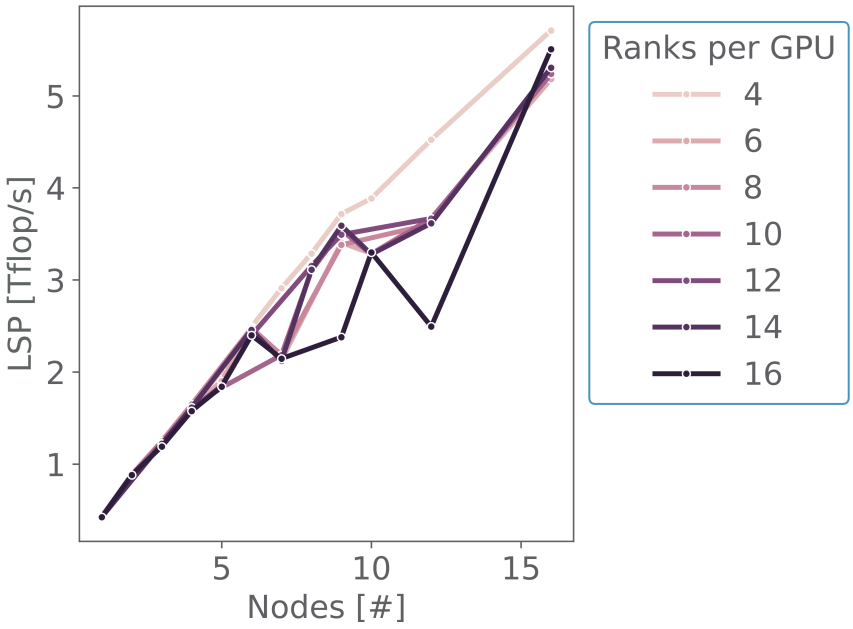

Abbildung 9: Gesamtleistung der linearen Löser in einem starken Skalierungsszenario bis 64 GPUs

Abbildung 10 zeigt einen Vergleich der erreichten Beschleunigung des gesamten Simulationsworkflows mittels petsc4FOAM (P4F) und OGL anhand des LidDrivenCavity3D Falls im Vergleich zu OpenFOAM auf CPUs. Der Speedup wurde sowohl zu Beginn des EXASIM-Projekts

mit den ersten Multi-GPU-fähigen Prototypen von OGL und petsc4FOAM als auch am Ende des Projekts nach Implementierung der beschriebenen Leistungsoptimierungen evaluiert.

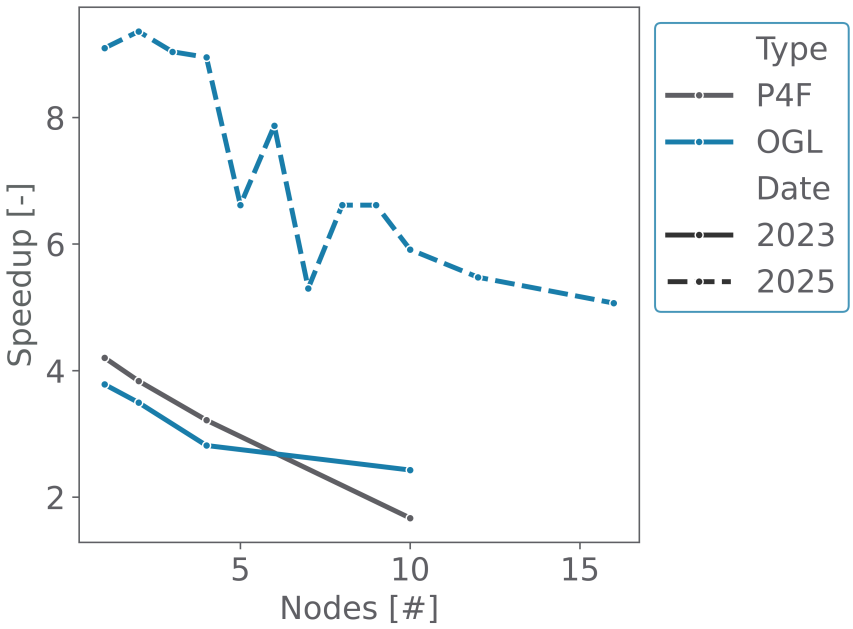

Abbildung 10: Vergleich der erreichten Beschleunigung zwischen petsc4Foam bei Projektbeginn und OGL erstes Jahr und bei Projektende

4. Verwendung der Zuwendung

Die bewilligten Zuwendungen wurden gemäß dem Projektantrag vorwiegend für Personalmittel eingesetzt. Dies war notwendig, um die notwendigen Implementierungen durchzuführen, sowie um die implementierte Software zu testen. Ein Teil der Zuwendungen wurde für Reisekosten aufgewendet, um zum Beispiel den Fortschritt auf der HPC-Statuskonferenz 2023–2025 sowie den OpenFOAM-Workshop 2025 vorzustellen. Die implementierten technischen Werkzeuge sind von großem Nutzen und sollen in zukünftigen Arbeiten weiterentwickelt werden. In naher Zukunft sollen die Erkenntnisse aus OGL und NeoN für einen kompletten CFD Workflow in industrie-relevanten Szenarien vereint werden.

5. Während der Durchführung des Vorhabens dem Zuwendungsempfänger bekannt gewordenen Fortschritt auf dem Gebiet des Vorhabens bei anderen Stellen

Siehe Diskussionen zu SPUMA im Bericht des Projektpartners Upstream CFD GmbH.

Bibliografie

1. Olenik G, Kashi A, Nayak P, et al IMPROVING LINEAR SOLVER PERFORMANCE BY OFFLOADING COMPUTATIONS TO GPGPUS WITH GINKGO
2. Olenik G, Koch M, Boutanios Z, Anzt H (2024) Towards a platform-portable linear algebra backend for OpenFOAM. Meccanica. https://doi.org/10.1007/s11012-024-01806-1
3. Olenik G, Koch M, Boutanios Z, Anzt H (2025) Towards a platform-portable linear algebra backend for OpenFOAM. Meccanica 60:1659–1672

4. Jasak H, Jemcov A, Tukovic Z, others (2007) OpenFOAM: A C++ library for complex physics simulations. In: International workshop on coupled methods in numerical dynamics. S 1–20
5. Anzt H, Cojean T, Flegar G, et al (2022) Ginkgo: A Modern Linear Operator Algebra Framework for High Performance Computing. ACM Trans Math Software 48:1–33. https://doi.org/10.1145/3480935
6. Bartlett R, Demeshko I, Gamblin T, et al (2017) xSDK foundations: Toward an extreme-scale scientific software development kit. arXiv preprint arXiv:170208425
7. Arndt D, Bangerth W, Davydov D, et al (2021) The deal.II finite element library: Design, features, and insights. Comput Math Appl 81:407–422. https://doi.org/10.1016/j.camwa.2020.02.022
8. Anderson R, Andrej J, Barker A, et al (2021) MFEM: A Modular Finite Element Methods Library. Comput Math Appl 81:42–74. https://doi.org/10.1016/j.camwa.2020.06.009
9. Plank G, Loewe A, Neic A, et al (2021) The openCARP Simulation Environment for Cardiac Electrophysiology. Comput Methods Programs Biomed 208:106223. https://doi.org/10.1016/j.cmpb.2021.106223
10. openCARP consortium, Augustin C, Boyle PM, et al (2024) openCARP. https://git.opencarp.org/openCARP/openCARP
11. Bnà S, Spisso I, Olesen M, Rossi G (2020) PETSc4FOAM: A Library to plug-in PETSc into the OpenFOAM Framework. PRACE White Paper
12. Mills RT, Adams MF, Balay S, et al (2021) Toward performance-portable PETSc for GPU-based exascale systems. Parallel Computing 108:102831. https://doi.org/https://doi.org/10.1016/j.parco.2021.102831
13. Bierbaum J, Planeta M, Härtig H (2022) Towards efficient oversubscription: on the cost and benefit of event-based communication in MPI. In: 2022 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS). S 1–10
14. Gärtner JW, Olenik G, Fadeli ME, et al (2025) Testing Strategies for OpenFOAM Projects. OpenFOAM® Journal 5:115–130. https://doi.org/10.51560/ofj.v5.134
15. Galeazzo FCC, Weiß RG, Lesnik S, et al (2024) Understanding superlinear speedup in current HPC architectures. In: IOP Conference Series: Materials Science and Engineering. S 12009
16. Olenik G, Koch M, Anzt H (2025) Investigating Matrix Repartitioning to Address the Over and Undersubscription Challenge for a GPU-Based CFD Solver. In: International Conference on High Performance Computing. S 468–479