HORIZON-EUROHPC-JU-2021-COE-01

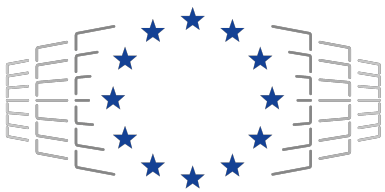
Centre of Excellence in Exascale CFD

CEEC – Centre of Excellence in Exascale CFD

Grant Agreement Number: 101093393

D2.2 – Approach to code generation

WP2: Software and performance engineering



EuroHPC Joint Undertaking

Document Information

Deliverable Number	D2.2
Deliverable Name	Approach to code generation
Due Date	31/12/2023 (PM 8)
Deliverable lead	FAU
Authors	Kajol Kulkarni (FAU)
Responsible Author	Kajol Kulkarni (FAU), kajol.kulkarni@fau.de
Keywords	Automatic code generation techniques
WP	WP2
Nature	R
Dissemination Level	PU
Final Version Date	31/12/2023
Reviewed by	Adam Peplinski (KTH), Niclas Jansson (KTH)
MGT Board Approval	31/12/2023

Acknowledgment:

Funded by the European Union. This work has received funding from the European High Performance Computing Joint Undertaking (JU) and Sweden, Germany, Spain, Greece, and Denmark under grant agreement No 101093393.

Document History

Partner	Date	Comment	Version
FAU	02/12/23	Initial version	0.1
FAU	08/12/23	Corrected version	0.2
FAU	20/12/23	Final version	1.0

Executive Summary

This document outlines the second deliverable for Work Package 2, focused on 'Software and Performance Engineering' within the Centre of Excellence for Exascale Computational Fluid Dynamics (CEEC) project. Within the scope of the CEEC project, the focus is on enhancing computational fluid dynamics (CFD) software frameworks. The goal is to optimize CFD software frameworks to efficiently harness the capabilities of the top European supercomputers for six designated lighthouse cases. These cases specifically tackle intricate problems within the realm of CFD.

Deliverable 2.2, identified as 'Code Generation Techniques,' offers a comprehensive overview of the observations made during the assessment of how code generation methodologies have been applied across diverse applications within CEEC. Within the scope of Work Package 2, this framework assumes a pivotal role, automating repetitive tasks, ensuring coding styles are consistent, and expediting the iterative prototyping process. Moreover, it plays a crucial role in the domain of performance optimization by tailoring code to specific hardware platforms and facilitating the adoption of domain-specific languages tailored to the distinctive requirements of specific problem domains.

The code generation techniques documented in this deliverable reflect the current state of the software frameworks. Project partners will utilize this information to assess possibilities for enhancements and collaboration among the frameworks.

Contents

1 Introduction

The Center of Excellence for Exascale Computational Fluid Dynamics (CEEC) focuses on enhancing computational fluid dynamics software frameworks, enabling six lighthouse cases to efficiently leverage the capabilities of leading EU supercomputers. These scenarios address complex challenges in the CFD field. The simulations demand substantial computing resources to achieve high-fidelity results and meaningful insights. CEEC's efforts are focused on optimizing CFD software frameworks to meet these computational demands, ensuring that the simulations run efficiently and effectively on the advanced hardware of the top EU supercomputers.

The deliverable presents an in-depth examination of the current situation of five major codes in CEEC: FLEXI, Alya, Nek5000/NekRS, Neko, and waLBerla. The assessment specifically focuses on the incorporation of code generation techniques within these codes. It aims to elucidate how effectively these codes leverage automated code generation to enhance productivity, improve code quality, and optimize performance. For codes that do utilize such techniques, the document delves into the strategies employed and their impact on efficiency. For codes that do not currently leverage code generation, the report explores the potential benefits and possibilities for incorporating these techniques in the future.

In the goal of optimizing computational fluid dynamics (CFD) algorithms, much effort is given to customizing code for various architectures, such as graphics processing units (GPUs) and central processing units (CPUs). This process involves a meticulous examination of key factors to ensure the efficient utilization of hardware resources and the seamless adaptation of code to varying computing environments. Furthermore, the generated code's characteristics across different dimensions – portability on diverse architectures, productivity, performance, and scalability, as well as energy efficiency on various HPC systems – merit thorough exploration. Understanding how the generated code aligns with these criteria is essential for gauging its adaptability, usability, and impact on overall system efficiency.

By strengthening the code generation, our objective is to equip code developers with the essential tools to address challenges and fully harness the capabilities inherent in code generation techniques for scientific simulations. The meticulous analysis presented in the subsequent sections delves into the current state of the code generation and performance tracking infrastructure for FLEXI, Alya, Nek5000/NekRS, Neko, and waLBerla. This examination not only forms the basis for potential advancements in both code development and performance analysis but also sheds light on the nuanced considerations in tailoring code for diverse architectures. As we explore the generated code's portability, productivity, performance, scalability, and energy efficiency on various HPC systems, the insights derived from this analysis will prove instrumental for project partners.

2 FLEXI

Currently, FLEXI does not use automatic code generation, and there are no plans for it in the near future. The focus is on supporting both CPU and GPU architectures by using dedicated implementations for each. The next step involves adding abstraction layers to handle different architectures and vendors more efficiently. While there's a potential interest in exploring automatic code generation for optimization, there are no definite plans in that direction at this time. The immediate goal is to improve and adapt the existing codebase for diverse hardware setups.

3 Alya

Alya, is a powerful multi-physics simulation tool designed for efficient supercomputing, emphasizing heterogeneous architectures. Although, it does not use code generation techniques. Alya excels in engineering simulations, utilizing finite element formulations, unstructured meshes, mesh deformation, and parallel solvers. It prioritizes features like hierarchical parallelism, efficient multiphysics coupling, and comprehensive parallelism in simulation workflows. The software follows continuous integration practices and is portable across architectures

4 Nek5000/NekRS

In the realm of NekRS, automatic code generation, strictly defined, may not be present in the conventional sense. Instead, the framework employs a dynamic approach to optimization during runtime, referred to as runtime specialization. This involves the selection of specialized kernels from a manually pre-programmed search space based on performance metrics. The manually crafted search space accounts for various parameters such as polynomial order in Spectral Element Methods (SEM), the distinction between low and high polynomial orders, the choice between GPU and CPU execution, the specific type of GPU in use, and the precision (single or double) required for computations. Approximately 10 crucial kernels, integral to the system's functionality, have multiple versions, each tailored to different configurations. These kernel versions are packaged with the code, and at runtime, the framework intelligently chooses the most performance-efficient version for a given problem.

Notably, the code generation process at runtime utilizes the OCCA kernel language [4] to translate the code into the specific target device language. While this may not align with the traditional notion of automatic code generation, it effectively achieves adaptability and optimization for diverse hardware and problem scenarios.

Nek5000 do no employ automatic code generation, users are required to manually construct and implement the code for their simulations.

5 Neko

Neko does not utilize automatic code generation for optimizing its application. However, tailoring the code for different architectures, including GPUs and CPUs, by employing separate backends and implementing hand-tuned code for each architecture. As Neko does not engage in automatic code generation, specific insights into the generated code's portability, productivity, performance, scalability, and energy efficiency on different HPC systems are not applicable. The optimization strategy centers around manual tuning for diverse architectures, reflecting a meticulous approach to address specific hardware nuances and maximize overall application efficiency.

6 waLBerla

Walberla employs automatic code generation to optimise the framework. The *lbmpy* module of waLBerla plays a crucial role in automatically generating code for enhanced portability across various architectures, including GPUs. Demonstrations have showcased scalable simulations with adaptive mesh refinement and load balancing on up to 2 million CPU threads (on Juqueen) for trillion-unknown meshes.

lbmpy framework is written in Python, and it generates code in C/C++/LLVM for CPUs, as well as in CUDA or OpenCL for GPUs. The *lbmpy* package allows for the specification of the lattice Boltzmann (LB) scheme in a high-level symbolic representation using *SymPy*. The hardware- and problem-specific transformations are then applied automatically, generating highly efficient code for CPU and GPU. This approach eliminates the need for manual optimization and ensures flexibility, maintainability, and high performance on different architectures.

The CFD application uses this code generation framework to obtain scalable and efficient LBM CUDA kernels. *lbmpy* enables the formulation of LBM methods, such as the Partially Saturated Cells Method (PSM), through a symbolic representation, generating optimized and parallel compute kernels. These generated compute kernels are seamlessly integrated into the simulation framework within waLBerla.

lbmpy utilizes automated code transformations for optimized compute kernels across diverse architectures, employing techniques such as loop unrolling and hardware-specific intrinsic. For GPU implementation, waLBerla facilitates communication hiding, crucial for multi-GPU simulations with MPI, by dividing the iteration region into inner and outer parts. The framework prioritizes considerations such as optimizing compute kernels, minimizing memory footprint, and enabling efficient inter-processor communication for overall improved performance [2]. Further more, waLBerla uses continuous integration (CI) testing methodologies to test the generated code.

lbmpy leverages a high-level symbolic representation and automated code transformations, ensuring portability across various architectures and seamless integration into existing HPC software as external C++ files. Built on the pystencils framework, *lbmpy* employs *SymPy*, enabling interactive development and prototyping on a single workstation. This approach, coupled with *lbmpy's* automation of development tasks, such as loop splitting and equation reformulation, enhances productivity by minimizing manual optimization efforts for efficient LBM code development.

lbmpy's code, drawing on extensive experience in stencil code optimization, is finely tuned for maximum efficiency and leverages a robust background in systematic performance engineering. Its support for high parallelism crucially contributes to scalability on modern computer architectures. Furthermore, validation across compilers and hardware platforms consistently demonstrates high performance and scalability, complementing its prowess in lattice Boltzmann simulations achieved through automated transformations, optimized kernels, and metaprogramming, as well as the use of the framework for efficient multi-GPU simulations [1, 2].

Fluid-particle simulations on CPU-GPU architectures align closely with A100 GPU memory bandwidth, underscoring GPU superiority. Minimal overhead in run times and efficient CPU-GPU communication in the hybrid setup emphasize its effectiveness. Promising weak scaling performance indicates success for large-scale multiphysics simulations with the hybrid approach [3].

7 Conclusion and Outlook

This deliverable offers a comprehensive summary of the current state of code generation on different CEEC applications. In this comprehensive assessment of five major codes within CEEC: FLEXI, Alya, Nek5000/NekRS, Neko, and waLBerla—the focus has been on the incorporation of code generation techniques to enhance productivity, code quality, and performance. The analysis provides valuable insights into the current state of these codes, their strategies, and the impact on computational efficiency.

FLEXI, a high-order accurate solver, prioritizes support for both CPU and GPU architectures without currently utilizing automatic code generation. The emphasis is on adding abstraction layers for efficient handling of diverse hardware setups, with potential interest in future exploration of code generation techniques.

Alya, designed for efficient supercomputing, excels in engineering simulations but does not currently employ code generation. It follows continuous integration practices, ensuring portability across architectures, and presents a solid foundation for exascale readiness.

Moreover, in NekRS, automatic code generation is absent, but run-time optimization dynamically selects specialized kernels from a manually programmed space, considering factors like polynomial order and hardware specifications, enhancing adaptability for diverse scenarios. Code is generated at runtime through OCCA kernel language to specific target devices, optimizing performance.

Neko adopts a manual tuning approach for different architectures, without using automatic code generation. Specific insights into portability, productivity, and energy efficiency on diverse HPC systems are not applicable.

In contrast, waLBerla stands out with its *lbmpy* module, employing automatic code generation for enhanced portability across various architectures, including GPUs. This approach, based on a high-level symbolic representation, eliminates the need for manual optimization and ensures flexibility, maintainability, and high performance on different platforms.

The analysis underscores the diverse strategies employed by CEEC codes in optimizing computational fluid dynamics algorithms for various architectures. While manual tuning remains a prevalent approach, the success of waLBerla's *lbmpy* module showcases the potential benefits of automated code generation. The outlook for CFD simulations involves a continued exploration of code generation techniques to address the increasing demands for high-fidelity results and meaningful insights on advanced HPC systems.

Future efforts could involve collaborative initiatives to share best practices and methodologies for code optimization. The incorporation of automatic code generation, as demonstrated by waLBerla, could be explored further by other codes to enhance adaptability, usability, and overall system efficiency. Additionally, ongoing advancements in hardware technologies, such as GPUs, warrant continuous adaptation of code to ensure optimal performance and scalability.

In conclusion, by prioritizing the enhancement of code generation techniques, the five codes FLEXI, Alya, Nek5000/NekRS, Neko, and waLBerla can foster a more robust and efficient development process. This will contribute to advancing work package 2 and enable researchers to tackle the complex physical problems addressed in the lighthouse cases with excellent efficiency and stable codes.

Bibliography

[1] Martin Bauer, Harald Köstler, and Ulrich Rüde. lbmpy: Automatic code generation for efficient parallel lattice boltzmann methods. *Journal of Computational Science*, 49:101269, 2021.

[2] M. Holzer, M. Bauer, H. Köstler, and U. Rüde. Highly efficient lattice boltzmann multiphase simulations of immiscible fluids at high-density ratios on cpus and gpus through code generation. *The International Journal of High Performance Computing Applications*, 35(4):413–427, 2021.

[3] Samuel Kemmler, Christoph Rettinger, and Harald Köstler. Efficient and scalable hybrid fluid-particle simulations with geometrically resolved particles on heterogeneous cpu-gpu architectures. *Journal of Computational Science 49*, 2023.

[4] David S Medina, Amik St-Cyr, and Tim Warburton. Occa: A unified approach to multi-threading languages. *arXiv preprint arXiv:1403.0968*, 2014.